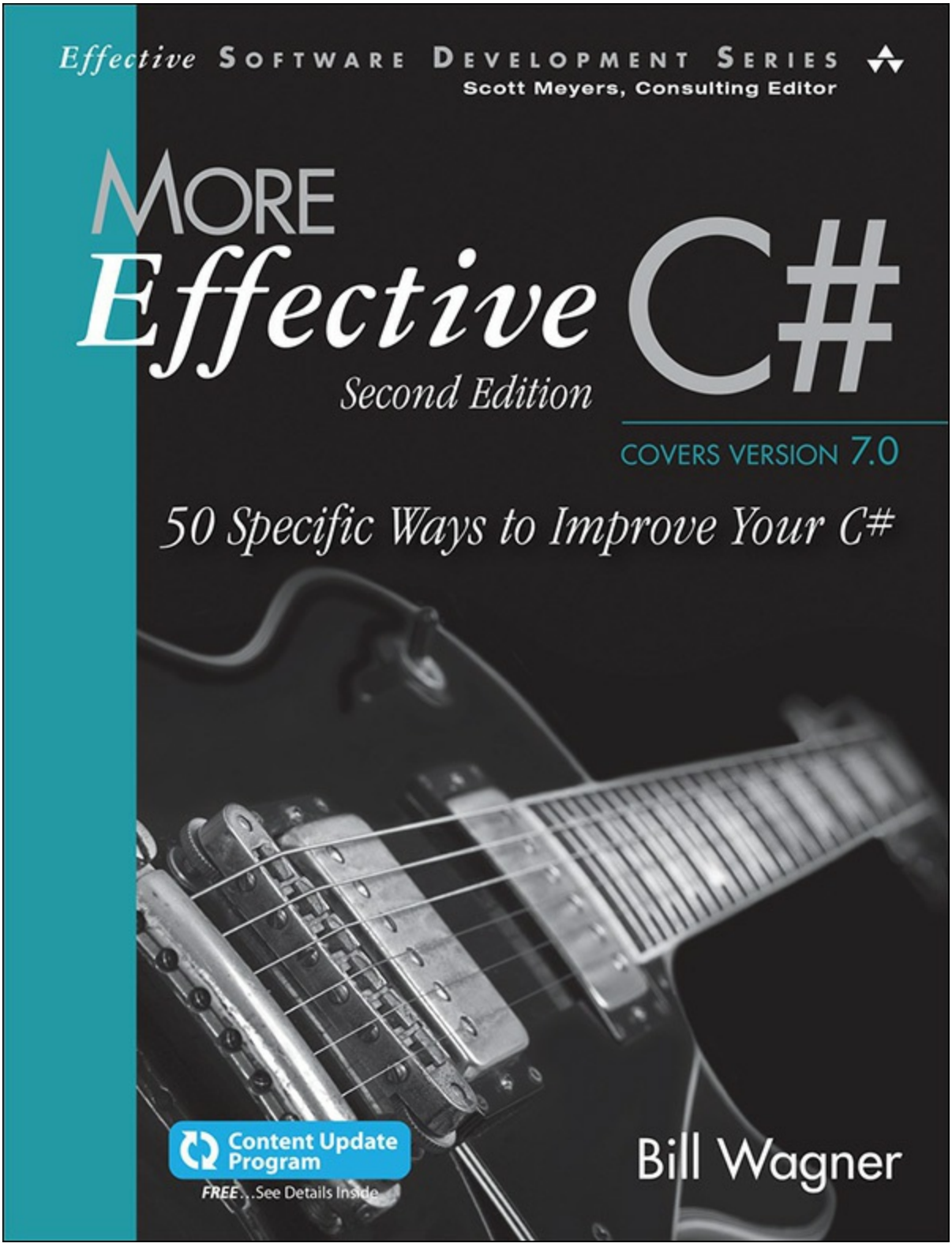

Effective Software Development Series
Scott Meyers, Consulting Editor
More Effective C#
Second Edition
COVERS VERSION 7.0
50 Specific Ways to Improve Your C#
Content Update Program
FREE...See Details Inside
Bill Wagner

Contents

Introduction

C# continues to evolve and change. As the language evolves, the community is also changing. More C# developers are approaching the language as their first professional programming language. These members of our community don't have the preconceptions common in those of us that started using C# after years with another C-based language. Even for those that have been using C# for years, the recent pace of change brings the need for many new habits. The C# language is seeing an increased pace of innovation in the years since the compiler has been Open-Sourced. The review of proposed features now includes the community. The community can participate in the design of the new features.

Changes in recommended architectures and deployments are also changing the language idioms we use. Building applications by composing microservices, distributed programs, 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. Rather, I provide 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 a larger number of items have been removed. With these editions, I've reorganizing 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 C# 7, but it is not an exhaustive treatment of the new language features. Like all books in the Effective Software Development

Series, this book offers practical advice on how to use these features to solve problems you're likely to encounter every day. I specifically cover C# 7 features where new language features introduce new and better ways to write common idioms. Internet searches may still point to earlier solutions that have years of history. I specifically point out older recommendations and 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 them here: https://github.com/BillWagner/MoreEffectiveCSharpAnalyzers. If you have ideas, or want to contribute, 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, this book discusses how you can integrate all the features of the current version of the C# language into your everyday development.

In addition to language features, I assume you have some knowledge of the Common Language Runtime (CLR) and Just-In-Time (JIT) compiler.

About the Content

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. That 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 API design. It's how you communicate to your users your expectations on 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. 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: mutli-threaded parallel execution. You'll see how PLINQ enables easier decomposition of complex algorithms across multiple cores and multiple CPUs.

Chapter 5 discusses how to use C# as a dynamic language. C# is a strongly typed, statically typed language. However, more and more 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. 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 more ways to be involved and help to 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 that will include eliding error recovery code for space. Public methods should validate their parameters and other inputs, but that code is usually elided for space. Similar space considerations remove validation of

method calls, and try/finally clauses that would often be included in complicated algorithms.

I also usually assume most developers can find the appropriate namespace when samples 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 samples. 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, are the result of 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.

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, 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 samples to ensure 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 created the quality of the finished manuscript from the front cover to the back, and everything in between. Curt Johnson continues to do an incredible job marketing technical content. No matter what 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. He 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, is invaluable.

As always, my family gave up time with me so that I could finish this manuscript. My wife, Marlene, gave up 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 finish.

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. Wagner 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.

1. Working with Data Types

C# was originally designed to support object oriented design techniques, where data and functionality are brought together. As it has matured, it has added new idioms to support programming practices that have become more common. One of those trends is to separate data storage concerns from the methods that manipulate that data. This trend is driven by the move to distributed systems, where an application is decomposed into many smaller services that each implement single features, or a small set of related features. Moving to a new strategy for separating concerns gives rise to new programming techniques. Using new programming techniques gives rise to new language features.

In this chapter, you'll learn techniques to separate data from the methods that manipulate or process that data. It's not always objects. Sometimes it's functions and passive data containers.

Item 1: Use Properties Instead of Accessible Data Members

Properties have always been a feature of the C# language. Several enhancements since the initial release of the C# language have made properties even more expressive. You can specify different access restrictions on the getter and setter. Auto properties minimize the hand typing for properties instead of data members, including `readonly` properties. Expression bodied members enable even more concise syntax. If you're still creating public fields in your types, stop now. If you're still creating get and set methods by hand, stop now. Properties let you expose data members as part of your public interface and still provide the encapsulation you want in an object-oriented environment. Properties are language elements that are accessed as though they are data members, but they are implemented as methods.

Some members of a type really are best represented as data: the name of a customer, the (x,y) location of a point, or last year's revenue. Properties enable

you to create an interface that acts like accessing data fields directly but still has all the benefits of a method. Client code accesses properties as though they are accessing public fields. But the actual implementation uses methods, in which you define the behavior of property accessors.

The .NET Framework assumes that you'll use properties for your public data members. In fact, the data binding classes in the .NET Framework support properties, not public data fields. This is true for all the data binding libraries: WPF, Windows Forms, and Web Forms. Data binding ties a property of an object to a user interface control. The data binding mechanism uses reflection to find a named property in a type:

```
textBoxCity.DataBindings.Add("Text",
    address, nameof(City));
```

The previous code binds the Text property of the `textBoxCity` control to the City property of the address object. It will not work with a public data field named City; the Framework Class Library designers did not support that practice. Public data members are bad practice, so support for them was not added. Their decision simply gives you yet another reason to follow the proper object-oriented techniques.

Yes, data binding applies only to those classes that contain elements that are displayed in your user interface logic. But that doesn't mean properties should be used exclusively in UI logic. You should use properties for other classes and structures. Properties are far easier to change as you discover new requirements or behaviors over time. You might soon decide that your customer type should never have a blank name. If you used a public property for Name, that's easy to fix in one location:

```
public class Customer
{
    private string name;
    public string Name
    {
        get => name;
        set
        {
            if (string.IsNullOrWhitespace(value))
                throw new ArgumentException(
                    "Name cannot be blank",
```

```
                    nameof(Name));
            name = value;
        }
        // More Elided.
    }
}
```

If you had used public data members, you're stuck looking for every bit of code that sets a customer's name and fixing it there. That takes more time—much more time.

Because properties are implemented with methods, adding multithreaded support is easier. You can enhance the implementation of the get and set accessors to provide synchronized access to the data (See Item 38 for more details):

```
public class Customer
{
    private object syncHandle = new object();

    private string name;
    public string Name
    {
        get
        {
            lock (syncHandle)
                return name;
        }
        set
        {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException(
                    "Name cannot be blank",
                    nameof(Name));
            lock (syncHandle)
                name = value;
        }
    }
    // More Elided.
}
```

Properties have the language features of methods. Properties can be virtual:

```
public class Customer
{
```

```
    public virtual string Name
    {
        get;
        set;
    }
}
```

You'll notice that the last examples use the implicit property syntax. Creating a property to wrap a backing store is a common pattern. Often, you won't need validation logic in the property getters or setters. The language supports the simplified implicit property syntax to decrease the amount of ceremonial code needed to expose a simple field as a property. The compiler creates a private field (typically called a backing store) for you and implements the obvious logic for both the get and set accessors.

You can extend properties to be abstract and define properties as part of an interface definition, using similar syntax to implicit properties. The example below shows a property definition in a generic interface. Note that while the syntax is consistent with implicit properties, the interface definition below does not include any implementation. It defines a contract that must be satisfied by any type that implements this interface.

```
public interface INameValuePair<T>
{
    string Name { get; }

    T Value { get; set; }
}
```

Properties are full-fledged, first-class language elements that are an extension of methods that access or modify internal data. Anything you can do with member functions, you can do with properties. Properties do not allow one other important source of breakage possible that is possible with fields: You cannot pass a property to a method using the `ref` or `out` keywords.

The accessors for a property are two separate methods that get compiled into your type. You can specify different accessibility modifiers to the get and set accessors in a property in C#. This gives you even greater control over the visibility of those data elements you expose as properties:

```
public class Customer
```

```
{
    public virtual string Name
    {
        get;
        protected set;
    }
    // remaining implementation omitted
}
```

The property syntax extends beyond simple data fields. If your type should contain indexed items as part of its interface, you can use indexers (which are parameterized properties). It's a useful way to create a property that returns the items in a sequence:

```
public int this[int index]
{
    get => theValues[index];
    set => theValues[index] = value;
}
     private int[] theValues = new int[100];

// Accessing an indexer:
int val = someObject[i];
```

Indexers have all the same language support as single-item properties: They are implemented as methods you write, so you can apply any verification or computation inside the indexer. Indexers can be virtual or abstract, can be declared in interfaces, and can be read-only or read-write. Single-dimension indexers with numeric parameters can participate in data binding. Other indexers can use noninteger parameters to define maps:

```
public Address this[string name]
{
    get => addressValues[name];
    set => addressValues[name] = value;
}
private Dictionary<string, Address> addressValues;
```

In keeping with the multidimensional arrays in C#, you can create multidimensional indexers, with similar or different types on each axis:

```
public int this[int x, int y]
    => ComputeValue(x, y);
```

```
public int this[int x, string name]
    => ComputeValue(x, name);
```

Notice that all indexers are declared with the `this` keyword. You cannot name an indexer in C#. Therefore, every different indexer in a type must have distinct parameter lists to avoid ambiguity. Almost all the capabilities for properties apply to indexers. Indexers can be virtual or abstract; indexers can have separate access restrictions for setters and getters. You cannot create implicit indexers as you can with properties.

This property functionality is all well and good, and it's a nice improvement. But you might still be tempted to create an initial implementation using data members and then replace the data members with properties later when you need one of those benefits. That sounds like a reasonable strategy—but it's wrong. Consider this portion of a class definition:

```
// using public data members, bad practice:
public class Customer
{
    public string Name;

    // remaining implementation omitted
}
```

It describes a customer, with a name. You can get or set the name using the familiar member notation:

```
string name = customerOne.Name;
customerOne.Name = "This Company, Inc.";
```

That's simple and straightforward. You are thinking that you could later replace the Name data member with a property, and the code would keep working without any change. Well, that's sort of true. Properties are meant to look like data members when accessed. That's the purpose behind the syntax. But properties are not data. A property access generates different Microsoft Intermediate Language (MSIL) instructions than a data access.

Although properties and data members are source compatible, they are not binary compatible. In the obvious case, this means that when you change from a public data member to the equivalent public property, you must recompile all

code that uses the public data member. C# treats binary assemblies as first-class citizens. One goal of the language is that you can release a single updated assembly without upgrading the entire application. The simple act of changing a data member to a property breaks binary compatibility. It makes upgrading single assemblies that have been deployed much more difficult.

While looking at the IL for a property, you probably wonder about the relative performance of properties and data members. Properties will not be faster than data member access, but they might not be any slower. The JIT compiler does inline some method calls, including property accessors. When the JIT compiler does inline property accessors, the performance of data members and properties is the same. Even when a property accessor has not been inlined, the actual performance difference is the negligible cost of one function call. That is measurable only in a small number of situations.

Properties are methods that can be viewed from the calling code like data. That puts some expectations into your users' heads. They will see a property access as though it was a data access. After all, that's what it looks like. Your property accessors should live up to those expectations. Get accessors should not have observable side effects. Set accessors do modify the state, and users should be able to see those changes.

Property accessors also have performance expectations for your users. A property access looks like a data field access. It should not have performance characteristics that are significantly different than a simple data access. Property accessors should not perform lengthy computations, or make cross-application calls (such as perform database queries), or do other lengthy operations that would be inconsistent with your users' expectations for a property accessor.

Whenever you expose data in your type's public or protected interfaces, use properties. Use an indexer for sequences or dictionaries. All data members should be private, without exception. You immediately get support for data binding, and you make it much easier to make any changes to the implementation of the methods in the future. The extra typing to encapsulate any variable in a property amounts to one or two minutes of your day. Finding that you need to use properties later to correctly express your designs will take hours. Spend a little time now, and save yourself lots of time later.

Item 2: Prefer Implicit Properties for Mutable Data

Additions to the property syntax mean that you can express your design intent clearly using properties. The modern C# syntax also supports changes to your design over time. Start with properties, and you enable many future scenarios.

When you add accessible data to a class, often the property accessors are simple wrappers around your data fields. When that's the case, you can increase the readability of your code by using implicit properties:

```
public string Name { get; set; }
```

The compiler creates the backing field using a compiler-generated name. You can even use the property setter to modify the value of the backing field. Because the name of the backing field is compiler generated, even inside your own class you need to call the property accessor rather than modify the backing field directly. That's not a problem. Calling the property accessor does the same work, and because the generated property accessor is a single assignment statement, it will likely be inlined. The runtime behavior of the implicit property is the same as the runtime behavior of accessing the backing field, even in terms of performance.

Implicit properties support the same property access specifiers as do their explicit counterparts. You can define any more-restrictive `set` accessor you need:

```
public string Name
{
    get;
    protected set;
}
// Or
public string Name
{
    get;
    internal set;
}
// Or
public string Name
{
    get;
```

```
    protected internal set;
}
// Or
public string Name
{
    get;
    private set;
}
// Or
// Can only be set in constructor:
public string Name { get; }
```

Implicit properties create the same pattern of a property with a backing field that you would have typed yourself in previous versions of the language. The advantage is that you are more productive, and your classes are more readable. An implicit property declaration shows anyone reading your code exactly what you intended to produce, and it doesn't clutter the file with extra information that only obscures the real meaning.

Of course, because implicit properties generate the same code as explicit properties, you can use implicit properties to define virtual properties, override virtual properties, or implement a property defined in an interface.

When you create a virtual implicit property, derived classes do not have access to the compiler-generated backing store. However, overrides can access the base property `get` and `set` methods just as they can with any other virtual method:

```
public class BaseType
{
    public virtual string Name
    {
        get;
        protected set;
    }
}

public class DerivedType : BaseType
{
    public override string Name
    {
        get => base.Name;
        protected set
```

```
        {
            if (!string.IsNullOrEmpty(value))
                base.Name = value;
        }
    }
}
```

You gain two additional advantages by using implicit properties. When the time comes to replace the implicit property with a concrete implementation because of data validation or other actions, you are making binary-compatible changes to your class, and your validation will be in only one location.

In earlier versions of the C# language, most developers directly accessed the backing field to modify it in their own class. That practice produces code that distributes the validation and error checking throughout the file. Every change to an implicit property's backing field calls the (possibly private) property accessor. You transform the implicit property accessor to an explicit property accessor, and then you write all the validation logic in the new accessor:

```
// original version
public class Person
{
    public string FirstName { get; set;}
    public string LastName { get; set; }
    public override string ToString() =>
        $"{FirstName} {LastName}";
}

// Later updated for validation
public class Person
{
    public Person(string firstName, string lastName)
    {
        // leverage validation in property setters:
        this.FirstName = firstName;
        this.LastName = lastName;
    }
    private string firstName;
    public string FirstName
    {
        get => firstName;
        set
        {
            if (string.IsNullOrEmpty(value))
```

```
                throw new ArgumentException(
                    "First name cannot be null or empty");
            firstName = value;
        }
    }

    private string lastName;
    public string LastName
    {
        get => lastName;
        private set
        {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException(
                    "Last name cannot be null or empty");
            lastName = value;
        }
    }
    public override string ToString() =>
        $"{FirstName} {LastName}";
}
```

You've created all the validation in one place. If you can continue to use your accessor rather than directly access the backing field, then you can continue to keep all the field validation in one location.

There is one important limitation of implicit properties. You cannot use implicit properties on types that are decorated with the `Serializable` attribute. The persistent file storage format depends on the name of the compiler-generated field used for the backing store. That field name is not guaranteed to remain constant. It may change at any time when you modify the class.

In spite of those two limitations, implicit properties save developer time, produce readable code, and promote a style of development in which all your field modification validation code happens in one location. If you create clearer code, it helps you maintain that code in a better way.

Item 3: Prefer Immutability for Value Types

Immutable types are simple: After they are created, they are constant. If you

validate the parameters used to construct the object, you know that it is in a valid state from that point forward. You cannot change the object's internal state to make it invalid. You save yourself a lot of otherwise necessary error checking by disallowing any state changes after an object has been constructed. Immutable types are inherently thread safe: Multiple readers can access the same contents. If the internal state cannot change, there is no chance for different threads to see inconsistent views of the data. Immutable types can be exported from your objects safely. The caller cannot modify the internal state of your objects. Immutable types work better in hash-based collections. The value returned by `Object.GetHashCode()` must be an instance invariant (see Item 7); that's always true for immutable types.

In practice, it is very difficult to make every type immutableThat's why this recommendation is for both atomic and immutable value types. Decompose your types to the structures that naturally form a single entity. An Address type does form such an entity. An address is a single thing, composed of multiple related fields. A change in one field likely means changes to other fields. A customer type is not an atomic type. A customer type will likely contain many pieces of information: an address, a name, and one or more phone numbers. Any of these independent pieces of information might change. A customer might change phone numbers without moving. A customer might move, yet still keep the same phone number. A customer might change his or her name without moving or changing phone numbers. A customer object is not atomic; it is built from many different immutable types using composition: an address, a name, or a collection of phone number/type pairs. Atomic types are single entities: You would naturally replace the entire contents of an atomic type. The exception would be to change one of its component fields.

Here is a typical implementation of an address that is mutable:

```
// Mutable Address structure.
public struct Address
{
    private string state;
    private int zipCode;

    // Rely on the default system-generated
    // constructor.

    public string Line1 { get; set; }
```

```
    public string Line2 { get; set; }
    public string City { get; set; }
    public string State
    {
        get => state;
        set
        {
            ValidateState(value);
            state = value;
        }
    }

    public int ZipCode
    {
        get => zipCode;
        set
        {
            ValidateZip(value);
            zipCode = value;
        }
    }

    // other details omitted.
}

// Example usage:
Address a1 = new Address();
a1.Line1 = "111 S. Main";
a1.City = "Anytown";
a1.State = "IL";
a1.ZipCode = 61111;
// Modify:
a1.City = "Ann Arbor"; // Zip, State invalid now.
a1.ZipCode = 48103; // State still invalid now.
a1.State = "MI"; // Now fine.
```

Internal state changes mean that it's possible to violate object invariants, at least temporarily. After you have replaced the City field, you have placed a1 in an invalid state. The city has changed and no longer matches the state or ZIP code fields. The code looks harmless enough, but suppose that this fragment is part of a multithreaded program. Any context switch after the city changes and before the state changes would leave the potential for another thread to see an inconsistent view of the data.

Okay, so you think you're not writing a multithreaded program. You can still

get into trouble. Imagine that the ZIP code was invalid and the set threw an exception. You've made only some of the changes you intended, and you've left the system in an invalid state. To fix this problem, you would need to add considerable internal validation code to the address structure. That validation code would add considerable size and complexity. To fully implement exception safety, you would need to create defensive copies around any code block in which you change more than one field. Thread safety would require adding significant thread-synchronization checks on each property accessor, both sets and gets. All in all, it would be a significant undertaking—and one that would likely be extended over time as you add new features.

Instead, if you need Address to be a `struct`, make it immutable. Start by changing all instance fields to read-only for outside uses.

```
public struct Address
{
    // remaining details elided
    public string Line1 { get; }
    public string Line2 { get; }
    public string City { get; }
    public string State { get; }
    public int ZipCode { get; }

    public Address(string line1,
        string line2,
        string city,
        string state,
        int zipCode) :
        this()
    {
        Line1 = line1;
        Line2 = line2;
        City = city;
        ValidateState(state);
        State = state;
        ValidateZip(zipCode);
        ZipCode = zipCode;
    }
}
```

Now you have an immutable type, based on the public interface. To make it useful, you need to add all necessary constructors to initialize the `Address` structure completely. The `Address` structure needs only one additional

constructor, specifying each field. A copy constructor is not needed because the assignment operator is just as efficient. Remember that the default constructor is still accessible. There is a default address where all the strings are null, and the ZIP code is 0:

```
public Address(string line1,
    string line2,
    string city,
    string state,
    int zipCode) :
    this()
{
    Line1 = line1;
    Line2 = line2;
    City = city;
    ValidateState(state);
    State = state;
    ValidateZip(zipCode);
    ZipCode = zipCode;
}
```

Using the immutable type requires a slightly different calling sequence to modify its state. You create a new object rather than modify the existing instance:

```
// Create an address:
Address a2 = new Address("111 S. Main",
    "", "Anytown", "IL", 61111);

// To change, re-initialize:
a2 = new Address(a1.Line1,
    a1.Line, "Ann Arbor", "MI", 48103);
```

The value of a1 is in one of two states: its original location in Anytown, or its updated location in Ann Arbor. You do not modify the existing address to create any of the invalid temporary states from the previous example. Those interim states exist only during the execution of the `Address` constructor and are not visible outside that constructor. As soon as a new Address object is constructed, its value is fixed for all time. It's exception safe: a1 has either its original value or its new value. If an exception is thrown during the construction of the new `Address` object, the original value of a1 is unchanged.

To create an immutable type, you need to ensure that there are no holes that

would allow clients to change your internal state. Value types do not support derived types, so you do not need to defend against derived types modifying fields. But you do need to watch for any fields in an immutable type that are mutable reference types. When you implement your constructors for these types, you need to make a defensive copy of that mutable type. All these examples assume that `Phone` is an immutable value type because we're concerned only with immutability in value types:

```
// Almost immutable: there are holes that would
// allow state changes.
public struct PhoneList
{
    private readonly Phone[] phones;

    public PhoneList(Phone[] ph)
    {
        phones = ph;
    }

    public IEnumerable<Phone> Phones
    {
        get { return phones; }
    }
}

Phone[] phones = new Phone[10];
// initialize phones
PhoneList pl = new PhoneList(phones);

// Modify the phone list:
// also modifies the internals of the (supposedly)
// immutable object.
phones[5] = Phone.GeneratePhoneNumber();
```

The array class is a reference type. The array referenced inside the `PhoneList` structure refers to the same array storage (phones) allocated outside the object. Developers can modify your immutable structure through another variable that refers to the same storage. To remove this possibility, you need to make a defensive copy of the array.. `Array` is a mutable type. One alternative would be to use the `ImmutableArray` class, in the `System.Collections.Immutable` namespace. The previous example shows the pitfalls of a mutable collection. Even more possibilities for mischief exist if the `Phone` type is a mutable reference type. Clients could modify the values

in the collection, even if the collection is protected against any modification. This is easy using the `ImmutableList` collection type:

```
public struct PhoneList
{
    private readonly ImmutableList<Phone> phones;

    public PhoneList(Phone[] ph)
    {
        phones = ph.ToImmutableList();
    }

    public IEnumerable<Phone> Phones => phones;
}
```

The complexity of a type dictates which of three strategies you will use to initialize your immutable type. The `Address` structure defined one constructor to allow clients to initialize an address. Defining the reasonable set of constructors is often the simplest approach.

You can also create factory methods to initialize the structure. Factories make it easier to create common values. The .NET Framework `Color` type follows this strategy to initialize system colors. The static methods `Color.FromKnownColor()` and `Color.FromName()` return a copy of a color value that represents the current value for a given system color.

Third, you can create a mutable companion class for those instances in which multistep operations are necessary to fully construct an immutable type. The .NET string class follows this strategy with the `System.Text.StringBuilder` class. You use the `StringBuilder` class to create a string using multiple operations. After performing all the operations necessary to build the string, you retrieve the immutable string from the `StringBuilder`.

Immutable types are simpler to code and easier to maintain. Don't blindly create `get` and `set` accessors for every property in your type. Your first choice for types that store data should be immutable, atomic value types. You easily can build more complicated structures from these entities.

Item 4: Distinguish Between Value Types and

Reference Types

Value types or reference types? Structs or classes? When should you use each? This isn't C++, in which you define all types as value types and can create references to them. This isn't Java, in which everything is a reference type (unless you are one of the language designers). You must decide how all instances of your type will behave when you create it. It's an important decision to get right the first time. You must live with the consequences of your decision because changing later can cause quite a bit of code to break in subtle ways. It's a simple matter of choosing the `struct` or `class` keyword when you create the type, but it's much more work to update all the clients using your type if you change it later.

It's not as simple as preferring one over the other. The right choice depends on how you expect to use the new type. Value types are not polymorphic. They are better suited to storing the data that your application manipulates. Reference types can be polymorphic and should be used to define the behavior of your application. Consider the expected responsibilities of your new type, and from those responsibilities, decide which type to create. Structs store data. Classes define behavior.

The distinction between value types and reference types was added to .NET and C# because of common problems that occurred in C++ and Java. In C++, all parameters and return values were passed by value. Passing by value is very efficient, but it suffers from one problem: partial copying (sometimes called slicing the object). If you use a derived object where a base object is expected, only the base portion of the object gets copied. You have effectively lost all knowledge that a derived object was ever there. Even calls to virtual functions are sent to the base class version.

The Java language responded by more or less removing value types from the language. All user-defined types are reference types. In the Java language, all parameters and return values are passed by reference. This strategy has the advantage of being consistent, but it's a drain on performance. Let's face it, some types are not polymorphic—they were not intended to be. Java programmers pay a heap allocation and an eventual garbage collection for every object instance. They also pay an extra time cost to dereference every

`this` in order to access any member of an object. All variables are references. In C#, you declare whether a new type should be a value type or a reference type using the `struct` or `class` keywords. Value types should be small, lightweight types. Reference types form your class hierarchy. This section examines different uses for a type so that you understand all the distinctions between value types and reference types.

To start, this type is used as the return value from a method:

```
private MyData myData;
public MyData Foo() => myData;

// call it:
MyData v = Foo();
TotalSum += v.Value;
```

If MyData is a value type, the content of the return is copied into the storage for v. However, if MyData is a reference type, you've exported a reference to an internal variable. You've violated the principle of encapsulation. That may enable callers to modify the object while bypassing your API(see Item 26).

Or, consider this variant:

```
public MyData Foo2() => myData.CreateCopy();

// call it:
MyData v = Foo();
TotalSum += v.Value;
```

Now, v is a copy of the original myData. As a reference type, two objects are created on the heap. You don't have the problem of exposing internal data. Instead, you've created an extra object on the heap. All in all, it's inefficient.

Types that are used to export data through public methods and properties should be value types. But that's not to say that every type returned from a public member should be a value type. There was an assumption in the earlier code snippet that MyData stores values. Its responsibility is to store those values.

But, consider this alternative code snippet:

```
private MyType myType;
public IMyInterface Foo3()
    => myType as IMyInterface;

// call it:
IMyInterface iMe = Foo3();
iMe.DoWork();
```

The myType variable is still returned from the Foo3 method. But this time, instead of accessing the data inside the returned value, the object is accessed to invoke a method through a defined interface

That simple code snippet starts to show you the distinction: Value types store values, and reference types define behavior. Reference types, defined in classes, have a rich palette of mechanisms to define complex behavior. You have inheritance. Mutability is easier to manage for reference types. Interface implementation does not imply possible boxing and unboxing operations. Value types have a simpler palette. You can create a public API that enforces invariants, but complex behaviors are harder to model. When you want to model complex behaviors, create a reference type.

Now look a little deeper at how those types are stored in memory and the performance considerations related to the storage models. Consider this class:

```
public class C
{
    private MyType a = new MyType();
    private MyType b = new MyType();

    // Remaining implementation removed.
}

C cThing = new C();
```

How many objects are created? How big are they? It depends. If MyType is a value type, you've made one allocation. The size of that allocation is twice the size of MyType. However, if MyType is a reference type, you've made three allocations: one for the C object, which is 8 bytes (assuming 32-bit pointers), and two more for each of the MyType objects that are contained in a C object. The difference results because value types are stored inline in an object, whereas reference types are not. Each variable of a reference type holds a

reference, and the storage requires extra allocation.

To drive this point home, consider this allocation:

```
MyType[] arrayOfTypes = new MyType[100];
```

If MyType is a value type, one allocation of 100 times the size of a MyType object occurs. However, if MyType is a reference type, one allocation just occurred. Every element of the array is null. When you initialize each element in the array, you will have performed 101 allocations—and 101 allocations take more time than 1 allocation. Allocating a large number of reference types fragments the heap and slows you down. If you are creating types that are meant to store data values, value types are the way to go. This is the last point, and it's the least important of the reasons to choose reference types or value types. The semantics of the types, discussed earlier, are much more important considerations.

The decision to make a value type or a reference type is an important one. It is a far-reaching change to turn a value type into a reference type. Consider this type:

```
public struct Employee
{
    // Properties elided
    public string Position { get; set; }

    public decimal CurrentPayAmount { get; set; }

    public void Pay(BankAccount b)
        => b.Balance += CurrentPayAmount;
}
```

This simple type contains one method to let you pay your employees. Time passes, and the system runs fairly well. Then you decide that there are different classes of Employees: Salespeople get commissions, and managers get bonuses. You decide to change the Employee type into a class:

```
public class Employee
{
    // Properties elided
    public string Position { get; set; }
```

```
    public decimal CurrentPayAmount { get; set; }

    public virtual void Pay(BankAccount b) =>
        b.Balance += CurrentPayAmount;
}
```

That breaks much of the existing code that uses your customer struct. Return by value becomes return by reference. Parameters that were passed by value are now passed by reference. The behavior of this little snippet changed drastically:

```
Employee e1 = Employees.Find(e => e.Position == "CEO");
BankAccount CEOBankAccount = new BankAccount();
decimal Bonus = 10000;
e1.CurrentPayAmount += Bonus; // Add one time bonus.
e1.Pay(CEOBankAccount);
```

What was a one-time bump in pay to add a bonus just became a permanent raise. Where a copy by value had been used, a reference is now in place. The compiler happily makes the changes for you. The CEO is probably happy, too. The CFO, on the other hand, will report the bug. You just can't change your mind about value and reference types after the fact: It changes behavior.

This problem occurred because the Employee type no longer follows the guidelines for a value type. In addition to storing the data elements that define an employee, you've added responsibilities—in this example, paying the employee. Responsibilities are the domain of class types. Classes can define polymorphic implementations of common responsibilities easily; structs cannot and should be limited to storing values.

The documentation for .NET recommends that you consider the size of a type as a determining factor between value types and reference types. In reality, a much better factor is the use of the type. Types that are simple structures or data carriers are excellent candidates for value types. It's true that value types are more efficient in terms of memory management: There is less heap fragmentation, less garbage, and less indirection. More important, value types are copied when they are returned from methods or properties. There is no danger of exposing references to mutable internal structures enabling unexpected state changes. But you pay in terms of features. Value types have

very limited support for common object-oriented techniques. You cannot create object hierarchies of value types. All value types are automatically sealed. You can create value types that implement interfaces but require boxing, which Item 17 shows causes performance degradation. Think of value types as storage containers, not objects in the OO sense.

You'll create more reference types than value types. If you answer yes to all these questions, you should create a value type. Compare these to the previous Employee example:

1. Is this type's principal responsibility data storage?

2. Can I make this type immutable?

3. Do I expect this type to be small?

4. Is its public interface defined entirely by properties that access its data members?

5. Am I confident that this type will never have subclasses?

6. Am I confident that this type will never be treated polymorphically?

Build low-level data storage types as value types. Build the behavior of your application using reference types. You get the safety of copying data that gets exported from your class objects. You get the memory usage benefits that come with stack-based and inline value storage, and you can utilize standard object-oriented techniques to create the logic of your application. When in doubt about the expected use, use a reference type.

Item 5: Ensure That 0 Is a Valid State for Value Types

The default .NET system initialization sets all objects to all 0s. There is no way for you to prevent other programmers from creating an instance of a value type that is initialized to all 0s. Make that the default value for your type.

One special case is `enums`. Never create an `enum` that does not include 0 as a

valid choice. All `enums` are derived from `System.ValueType`. The values for the enumeration start at 0, but you can modify that behavior:

```
public enum Planet
{
    // Explicitly assign values.
    // Default starts at 0 otherwise.
    Mercury = 1,
    Venus = 2,
    Earth = 3,
    Mars = 4,
    Jupiter = 5,
    Saturn = 6,
    Uranus = 7,
    Neptune = 8
    // First edition included Pluto.
}

Planet sphere = new Planet();
var anotherSphere = default(Planet);
```

Both `sphere` and `anotherSphere` are 0, which is not a valid value. Any code that relies on the (normal) fact that `enums` are restricted to the defined set of enumerated values won't work. When you create your own values for an `enum`, make sure that 0 is one of them. If you use bit patterns in your `enum`, define 0 to be the absence of all the other properties.

As it stands now, you force all users to explicitly initialize the value:

```
Planet sphere2 = Planet.Mars;
```

That makes it harder to build other value types that contain this type:

```
public struct ObservationData
{
    private Planet whichPlanet; //what am I looking at?
    private double magnitude; // perceived brightness.
}
```

Users who create a new `ObservationData` object will create an invalid `Planet` field:

```
ObservationData d = new ObservationData();
```

The newly created ObservationData has a 0 magnitude, which is reasonable. But the planet is invalid. You need to make 0 a valid state. If possible, pick the best default as the value 0. The Planet `enum` does not have an obvious default. It doesn't make any sense to pick some arbitrary planet whenever the user does not. If you run into that situation, use the 0 case for an uninitialized value that can be updated later:

```
public enum Planet
{
    None = 0,
    Mercury = 1,
    Venus = 2,
    Earth = 3,
    Mars = 4,
    Jupiter = 5,
    Saturn = 6,
    Neptune = 7,
    Uranus = 8
}

Planet sphere = new Planet();
```

sphere now contains a value for `None`. Adding this uninitialized default to the Planet `enum` ripples up to the `ObservationData` structure. Newly created `ObservationData` objects have a 0 magnitude and `None` for the target. Add an explicit constructor to let users of your type initialize all the fields explicitly:

```
public struct ObservationData
{
    Planet whichPlanet; //what am I looking at?
    double magnitude; // perceived brightness.

    ObservationData(Planet target,
        double mag)
    {
        whichPlanet = target;
        magnitude = mag;
    }
}
```

But remember that the default constructor is still visible and part of the structure. Users can still create the system-initialized variant, and you can't stop them.

This is still somewhat faulty, because observing nothing doesn't really make sense. You could solve this specific case by changing `ObservationData` to a class, which means that the parameterless constructor does not need to be accessible. But, when you are creating an `enum`, you cannot force other developers to abide by those rules. Enums are a thin wrapper around integers If a set of integral constants does not provide the abstraction you need, you should consider another language feature instead.

Before leaving `enums` to discuss other value types, you need to understand a few special rules for `enums` used as flags. `enums` that use the Flags attribute should always set the None value to 0:

```
[Flags]
public enum Styles
{
    None = 0,
    Flat = 1,
    Sunken = 2,
    Raised = 4,
}
```

Many developers use flags enumerations with the bitwise AND operator. 0 values cause serious problems with bitflags. The following test will never work if Flat has the value of 0:

```
Styles flag = Styles.Sunken;
if ((flag & Styles.Flat) != 0) // Never true if Flat == 0.
    DoFlatThings();
```

If you use Flags, ensure that 0 is valid and that it means "the absence of all flags."

Another common initialization problem involves value types that contain references. `Strings` are a common example:

```
public struct LogMessage
{
    private int ErrLevel;
    private string msg;
}

LogMessage MyMessage = new LogMessage();
```

MyMessage contains a `null` reference in its msg field. There is no way to force a different initialization, but you can localize the problem using properties. You created a property to export the value of msg to all your clients. Add logic to that property to return the empty string instead of `null`:

```
public struct LogMessage
{
    private int ErrLevel;
    private string msg;

    public string Message
    {
        get => get => msg ?? string.Empty;        set => msg =
    }
}
```

You should use this property inside your own type. Doing so localizes the `null` reference check to one location. The `Message` accessor is almost certainly inlined as well, when called from inside your assembly. You'll get efficient code and minimize errors.

The system initializes all instances of value types to 0. There is no way to prevent users from creating instances of value types that are all 0s. If possible, make the all 0 case the natural default. As a special case, `enums` used as flags should ensure that 0 is the absence of all flags.

Item 6: Ensure That Properties Behave Like Data

Properties lead dual lives. From the outside, they appear to be passive data elements. However, on the inside they are implemented as methods. This dual life can lead you to create properties that don't live up to your users' expectations. Developers using your types will assume that accessing a property behaves the same as accessing a data member. If you create properties that don't live up to those assumptions, your users will misuse your types. Property access gives the impression that calling those particular methods will have the same characteristics as accessing a data member directly.

When properties correctly model data members, they live up to client

developers' expectations. First, client developers will believe that subsequent calls to a `get` accessor without any intervening statements will produce the same answer:

```
int someValue = someObject.ImportantProperty;
Debug.Assert(someValue == someObject.ImportantProperty);
```

Of course, multiple threads could violate this assumption, whether you're using properties or fields. But otherwise, repeated calls to the same property should return the same value.

Finally, developers using your type will not expect property accessors to do much work. A property getter should never be an expensive operation. Similarly, property `set` accessors will likely do some validation, but it should not be expensive to call them.

Why do developers using your types have these expectations? It's because they view properties as data. They access properties in tight loops. You've done the same thing with .NET collection classes. Whenever you enumerate an array with a `for` loop, you retrieve the value of the array's `Length` property repeatedly:

```
for (int index = 0; index < myArray.Length; index++)
```

The longer the array, the more times you access the `Length` property. If you had to count all elements every time you accessed the array's `Length` property, every loop would have quadratic performance. No one would use loops.

Living up to your client developers' expectations is not hard. First, use implicit properties. **Implicit properties** are a thin wrapper around a compiler-generated backing store. Their characteristics closely match those of data access. In fact, because the property accessors are simple implementations, they are often inlined. Whenever you can implement properties in your design using implicit properties, you will live up to client expectations.

However, if your properties contain behavior that isn't implemented in implicit properties, that's not always a concern. You'll likely add validation in your property setters, and that will satisfy users' expectations. Earlier I showed you this implementation of a property setter for a `LastName`:

```
public string LastName
{
    // getter elided
    set
    {
        if (string.IsNullOrEmpty(value))
            throw new ArgumentException(
            "last name can't be null or blank");
        lastName = value;
    }
}
```

That validation code doesn't break any of the fundamental assumptions about properties. It executes quickly, and it protects the validity of the object.

Also, property `get` accessors often perform some computation before returning the value. Suppose you have a `Point` class that includes a property for its distance from the origin:

```
public class Point
{
    public int X { get; set; }
    public int Y { get; set; }
    public double Distance => Math.Sqrt(X * X + Y * Y);
    }
}
```

Computing the distance is quick, and your users won't see performance problems if you have implemented `Distance` in this way. However, if `Distance` did turn out to be a bottleneck, you could cache the distance the first time you compute it. Of course, this also means that you need to invalidate the cached value whenever one of the component values changes. (Or you could make `Point` an immutable type.)

```
public class Point
{
    private int xValue;
    public int X
    {
        get => xValue;
        set
        {
            xValue = value;
            distance = default(double?);
```

```
        }
    }
    private int yValue;
    public int Y
    {
        get => yValue;
        set
        {
            yValue = value;
            distance = default(double?);
        }
    }
    private double? distance;
    public double Distance
    {
        get
        {
            if (!distance.HasValue)
                distance = Math.Sqrt(X * X + Y * Y);
            return distance.Value;
        }
    }
}
```

If computing the value returned by a property getter is much more expensive, you should rethink your public interface.

```
// Bad Property Design. Lengthy operation required for getter
public class MyType
{
    // lots elided
    public string ObjectName =>
        RetrieveNameFromRemoteDatabase();
}
```

Users don't expect that accessing a property will require round trips to remote storage, or possible throw an exception. You need to change the public API to meet users' expectations. Every type is different, so your implementation must depend on the usage pattern for the type. You may find that caching the value is the right answer:

```
// One possible path: evaluate once and cache the answer
public class MyType
{
    // lots elided
```

```
    private string objectName;
    public string ObjectName =>
        (objectName != null) ?
        objectName : RetrieveNameFromRemoteDatabase();
}
```

This technique is implemented in the Lazy<T> class in the Framework. The code above could be replaced with:

```
private Lazy<string> lazyObjectName;
public MyType()
{
    lazyObjectName = new Lazy<string>
        (() => RetrieveNameFromRemoteDatabase());
}
public string ObjectName => lazyObjectName.Value;
```

It works well when the `ObjectName` property is needed only occasionally. You save the work of retrieving the value when it's not needed. In return, the first caller to ask for the property pays an extra penalty. If this type almost always uses the `ObjectName` property and it's valid to cache the name, you could load the value in the constructor and use the cached value as the property return value. The preceding code also assumes that `ObjectName` can be safely cached. If other portions of the program or other processes in the system change the remote storage for the object name, then this design fails.

The operations of pulling data from a remote database and later saving changes back to the remote database are common enough, and certainly valid. You can live up to your users' expectations by performing those operations in methods, giving those methods names that match the operations. Here's a different version of `MyType` that lives up to expectations:

```
//Better solution: Use methods to manage cached values
public class MyType
{
    public void LoadFromDatabase()
    {
        ObjectName = RetrieveNameFromRemoteDatabase();
        // other fields elided.
    }

    public void SaveToDatabase()
    {
```

```
            SaveNameToRemoteDatabase(ObjectName);
            // other fields elided.
        }

        // lots elided

        public string ObjectName { get; set; }
    }
```

It's not only `get` accessors that can break client developer assumptions. You can create code that breaks your users' assumptions in the property setter as well. Suppose that `ObjectName` is a read-write property. If the setter wrote the value to the remote database it would break your users' assumptions:

```
public class MyType
{
    // lots elided
    private string objectName;
    public string ObjectName
    {
        get
        {
            if (objectName == null)
                objectName = RetrieveNameFromRemoteDatabase();
            return objectName;
        }
        set
        {
            objectName = value;
            SaveNameToRemoteDatabase(objectName);
        }
    }
}
```

This extra work in the setter breaks several assumptions made by your users. Client code developers won't expect that a setter is making a remote call to a database. It will take longer than they expect. It also has the chance of failing in many ways they don't expect.

Finally, debuggers may invoke your property getters automatically in order to display the value of those properties. If the getters throw, take a long time, or change internal state, that's going to complicate your debugging sessions.

Properties set different expectations for client developers than do methods.

Client developers expect properties to execute quickly and to provide a view into object state. They expect properties to be like data fields, both in behavior and in their performance characteristics. When you create properties that violate those assumptions, you should modify the public interface to create methods that represent the operations that don't live up to users' expectations for properties. That practice lets you return the properties to their purpose of providing a window into object state.

Item 7: Limit Type Scope by Using Anonymous Types

You have more options than ever to create user defined types that represent the objects and data structures in your programs. You can choose classes, structs, tuple types, or anonymous types. Classes and structs provide a rich vocabulary to express your designs. That leads many developers to pick these constructs without considering any other possibilities. That's too bad because classes and structs, while powerful constructs, require more ceremony for simple designs. You can write more readable code by using the simpler constructs of anonymous types or tuples more often. It pays to learn more about what constructs are supported by these types, how they differ, and how they both differ from classes and structs.

Anonymous types are compiler generated immutable reference types. Let's go through that definition bit by bit. You author an anonymous type by declaring a new variable and defining the fields inside { and } characters:

```
var aPoint = new { X = 5, Y = 67 };
```

You've told the compiler several things. You've indicated that you need a new internal sealed class. You've told the compiler that this new type is an immutable type and that it has two public read-only properties surrounding two backing fields (`X, Y`).

You've told the compiler to write something like this for you:

```
internal sealed class AnonymousMumbleMumble
{
    private readonly int x;
```

```
    public int X
    {
        get => x;
    }

    private readonly int y;
    public int Y
    {
        get => y;
    }

    public AnonymousMumbleMumble(int xParm, int yParm)
    {
        x = xParm;
        y = yParm;
    }
    // And free implementations of ==, and GetHashCode()
    // elided.
}
```

Instead of writing this by hand, I'd rather let the compiler write it for me. There are a great many advantages. Most simply, the compiler is faster. I can't type the full class definition nearly as fast as I can type the new expression. Second, the compiler generates the same definition for these repetitive tasks. As developers, we occasionally miss something. This is pretty simple code, so the chances of error aren't very great, but they're not zero. The compiler does not make those human mistakes. Third, letting the compiler generate the code minimizes the amount of code to maintain. No other developer needs to read this code, figure out why you wrote it, figure out what it does, and find where it is used. Because the compiler generates the code, there is less to figure out and less to look at.

The obvious drawback of using anonymous types is that you don't know the name of the type. Because you don't name the type, you can't use an anonymous type as a parameter to a method or as its return value. Still, there are ways to work with single objects or sequences of anonymous types. You can write methods or expressions that work with anonymous types inside a method. You must define them as lambda expressions or anonymous delegates so that you can define them inside the body of the method where the anonymous type was created. If you mix in generic methods that contain function parameters, you can create methods that work with anonymous methods. For example, you can

double both the `X` and `Y` values for a `Point` by creating a transform method:

```
static T Transform<T>(T element, Func<T, T> transformFunc)
{
    return transformFunc(element);
}
```

You can pass an anonymous type to the `Transform` method:

```
var aPoint = new { X = 5, Y = 67 };
var anotherPoint = Transform(aPoint, (p) =>
    new { X = p.X * 2, Y = p.Y * 2 });
```

Of course, complicated algorithms will require complicated lambda expressions, and probably multiple calls to various generic methods. But it's only more of the simple example I've shown. That is what makes anonymous types great vehicles for storing interim results. The scope of an anonymous type is limited to the method where it is defined. The anonymous type can store results from the first phase of an algorithm and pass those interim results into the second phase. Using generic methods and lambdas means that you can define any necessary transformations on those anonymous types within the scope of the method where the anonymous type is defined.

What's more, because the interim results are stored in anonymous types, those types do not pollute the application's namespace. You can have the compiler create these simple types and shield developers from needing to understand them to understand the application. Anonymous types are scoped inside the method where they are declared. Using an anonymous type clearly shows other developers that a particular type is scoped inside that single method.

You may have noticed that I use some vague words earlier when I describe how the compiler defines an anonymous type. The compiler generates "something like" what I wrote when you tell it you need an anonymous type. The compiler adds some features that you can't write yourself. Anonymous types are immutable types that support object initializer syntax. If you create your own immutable type, you must hand-code your constructors so that client code can initialize every field or property in that type. Hand-coded immutable types would not support object initializer syntax, because there are no accessible property setters. Still, you can and must use object initializer syntax

when you construct an instance of an anonymous type. The compiler creates a public constructor that sets each of the properties, and it substitutes a constructor call for the property setters at the call point.

For example, suppose you have this call:

```
var aPoint = new { X = 5, Y = 67 };
```

It is translated by the compiler into this:

```
AnonymousMumbleMumble aPoint = new AnonymousMumbleMumble(5, 67)
```

The only way you can create an immutable type that supports object initializer syntax is to use an anonymous type. Hand-coded types do not get the same compiler magic.

Finally, I've said that there is less runtime cost for anonymous types than you might have thought. You might naively think that each time you new up any anonymous type, the compiler blindly defines a new anonymous type. Well, the compiler is a little smarter than that. Whenever you create the same anonymous type, the compiler reuses the same anonymous type as before.

I need to be a little more precise about what the compiler views as the same anonymous types if they're used in different locations. First, obviously that happens only if the multiple copies of the anonymous type are declared in the same assembly.

Second, for two anonymous types to be considered the same, the property names and types must match, and the properties must be in the same order. The following two declarations produce two different anonymous types:

```
var aPoint = new { X = 5, Y = 67 };
var anotherPoint = new { Y = 12, X = 16 };
```

By putting the properties in different orders, you have created two different anonymous types. You would get the same anonymous type only by ensuring that all properties are declared in the same order every time you declare an object that is meant to represent the same concept.

Before we leave anonymous types, a special case deserves mention. Because anonymous types follow value semantics for equality, they can be used as composite keys. For example, suppose you need to group customers by salesperson and ZIP code. You could run this query:

```
var query = from c in customers
            group c by new { c.SalesRep, c.ZipCode };
```

This query produces a dictionary in which the keys are a pair of `SalesRep` and `ZipCode`. The values are lists of `Customers`.

Tuples are similar in that they are lightweight types that you define by creating instances. They are distinct from anonymous types in that they are mutable value types with public fields. The compiler defines the actual type by deriving from a generic `Tuple` type, giving the properties the names you specify.

Furthermore, instantiating a tuple does not generate a new type the way that instantiating a new anonymous type does. Instead, instantiating a tuple creates a new closed generic type of one of the ValueTuple generic classes. (There are multiple ValueTuple generic types for tuples with different numbers of fields.)

You would instantiate a tuple using this syntax:

```
var aPoint = (X: 5, Y: 67);
```

You've written that you want a Tuple containing two integer fields. The C# compiler also keeps track of the semantic names (`X` and `Y`) that you've chosen for these fields. This feature means that you can access these fields using these semantic names:

```
Console.WriteLine(aPoint.X);
```

The `System.ValueTuple` generic structures contain methods for equality tests, comparison tests, and a `ToString()` method that prints the value of each field in the tuple. The instantiated ValueTuple contains fields names `Item1`, `Item2`, and so on for each field used. The compiler and many development tools enable the use of the semantic field names used to define the tuple.

C# type compatibility is generally based on the name of a type, referred to as nominiative typing. Tuples use structural typing rather than nominative typing to determine if different objects are the same type. Tuples rely on their 'shape' rather than a name (even a compiler generated one) to determine the specific type of tuples. Any tuple that contains exactly 2 integers would be the same type as the tuple point above. Both would be an instantiation of `System.ValueTuple<int, int>`. The semantic names for the fields are set when a tuple is initialized, either explicitly when declaring the variable, or implicitly by using the names from the right-hand side of the instantiation. If the field names are specified on both the left and right side, the names from the left-hand side are used.

```
var aPoint = (X: 5, Y: 67);
// another point has fields 'X' and 'Y'
var anotherPoint = aPoint;

// pointThree has fields 'Rise' and 'Run'
(int Rise, int Run) pointThree = aPoint;
```

Tuples and anonymous types are both lightweight types that are defined by the statements that instantiate them. They are both easy to use when you want to define simple storage types that hold data but do not define any behavior.

There are differences between anonymous types and tuples that you must consider when deciding which to use. Tuples are preferred for method return types and method parameters because they following structural typing. Anonymous types are better for composite keys in collections because they are immutable. Tuples have all the advantages associated with value types; anonymous types have all the advantages associated with reference types (Reference another item here). You can also experiment using both. Look back at the first initialization examples. The syntax to instantiate either is similar.

Anonymous types and tuples aren't as exotic as they seem, and they don't harm readability when they are used correctly. If you have interim results that you need to keep track of and if they're modeled well with an immutable type, then you should use anonymous types. If they are modeled better as discrete mutable values, use a tuple.

Item 8: Define Local Functions on Anonymous Types

Tuples don't use nominative typing. Anonymous types have names, but you can't reference them (see Item 7). That means learning techniques to use these lightweight types as method arguments, method returns, and properties. It also means understanding the limitations when you use either of these types in this manner.

Let's start with tuples. You simply define the shape of the tuple as the return type:

```
static (T sought, int index) FindFirstOccurence<T>(
    IEnumerable<T> enumerable, T value)
{
    int index = 0;
    foreach (T element in enumerable)
    {
        if (element.Equals(value))
        {
            return (value, index);
        }
        index++;
    }
    return (default(T), -1);
}
```

You don't need to specify the names for the fields of the returned tuple, but you should to communicate the meaning of the fields to callers.

You can assign the return value either to a tuple, or use deconstruction to assign the returned values to distinct variables:

```
// assign the result to a tuple variable:
var result = FindFirstOccurence(list, 42);
Console.WriteLine($"First {result.sought} is at {result.index}'
// assign the result to different variables:
(int number, int index) = FindFirstOccurence(list, 42);
Console.WriteLine($"First {number} is at {index}");
```

Tuples are easy to use for method return values. Anonymous types are harder to work with because they have names, but you can't type them in source code. The answer is to create generic methods and specify the anonymous type in

place of a type parameter.

As a simple example, this method returns a sequence of all objects in a collection that match a sought value:

```
static IEnumerable<T> FindValue<T>(IEnumerable<T> enumerable,
    T value)
{
    foreach (T element in enumerable)
    {
        if (element.Equals(value))
        {
            yield return element;
        }
    }
}
```

You can use it with anonymous types like this:

```
IDictionary<int, string> numberDescriptionDictionary =
    new Dictionary<int, string>()
{
    {1,"one"},
    {2, "two"},
    {3, "three"},
    {4, "four"},
    {5, "five"},
    {6, "six"},
    {7, "seven"},
    {8, "eight"},
    {9, "nine"},
    {10, "ten"},
};
List<int> numbers = new List<int>()
    { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
var r = from n in numbers
        where n % 2 == 0
        select new
        {
            Number = n,
            Description = numberDescriptionDictionary[n]
        };
r = from n in FindValue(r,
    new { Number = 2, Description = "two" })
    select n;
```

The `FindValue()` method knows nothing about the type; it's simply a generic type.

Of course, such simple functions can do only so much. If you want to write methods that use particular properties in your anonymous types, you need to create and use higher-order functions. A **higher-order function** is one that either takes a function as a parameter or returns a function. Higher-order functions that take functions as parameters are useful when you're working with anonymous types. You can use higher-order functions and generics to work with anonymous methods across multiple methods. Take a look at this query:

```
Random randomNumbers = new Random();
var sequence = (from x in Utilities.Generator(100,
                    () => randomNumbers.NextDouble() * 100)
                let y = randomNumbers.NextDouble() * 100
                select new { x, y }).TakeWhile(
                point => point.x < 75);
```

The query ends in the `TakeWhile()` method, which has this signature:

```
public static IEnumerable<TSource> TakeWhile<TSource>
    (this IEnumerable<TSource> source,
    Func<TSource, bool> predicate);
```

Notice that the signature of `TakeWhile` returns an `IEnumerable<TSource>` and has an `IEnumerable<TSource>` parameter. In our simple example, `TSource` stands in for an anonymous type representing an `X,Y` pair. `Func<TSource, bool>` represents a function that takes a `TSource` as its parameter.

This technique gives you the pathway to creating large libraries and code that works with anonymous types. The query expressions rely on generic methods that can work with anonymous types. The lambda expression, because it's declared in the same scope as the anonymous type, knows all about the anonymous type. The compiler creates the private nested class that passes around instances of the anonymous type to the other methods.

The following code creates an anonymous type and then processes that type in many generic methods:

```
var sequence = (from x in Funcs.Generator(100,
                    () => randomNumbers.NextDouble() * 100)
                let y = randomNumbers.NextDouble() * 100
                select new { x, y }).TakeWhile(
                point => point.x < 75);

var scaled = from p in sequence
             select new {x = p.x * 5, y = p.y * 5};
var translated = from p in scaled
                 select new { x = p.x - 20, y = p.y - 20};

var distances = from p in translated
                let distance = Math.Sqrt(
                    p.x * p.x + p.y * p.y)
                where distance < 500.0
                select new { p.x, p.y, distance };
```

There isn't anything amazing going on here. It's simply the compiler generating delegates and calling them. Every one of those query methods results in a compiler-generated method that takes your anonymous type as a parameter. The compiler creates a delegate that is bound to each of those methods and uses that delegate as the parameter to the query method.

As this first sample continues to grow, it's easy to let algorithms get out of hand, create multiple copies of algorithms, and end up with a large investment in repeated code. So let's look at how to modify this code so that as more capabilities are needed, you can continue to keep the code simple, modular, and extensible.

One approach is to move some of the code around to create a simpler method and yet preserve the reusable blocks. You refactor some of the algorithms into generic methods that will take lambda expressions to perform the specific work needed by the algorithm.

Almost all of the following methods perform a simple mapping from one type to another. Some of them are an even simpler mapping to a different object of the same type.

```
public static IEnumerable<TResult> Map<TSource, TResult>
    (this IEnumerable<TSource> source,
    Func<TSource, TResult> mapFunc)
{
```

```
    foreach (TSource s in source)
        yield return mapFunc(s);
}

// Usage:
var sequence = (from x in Utilities.Generator(100,
                    () => randomNumbers.NextDouble() * 100)
                let y = randomNumbers.NextDouble() * 100
                select new { x, y }).TakeWhile(
                point => point.x < 75);

var scaled = sequence.Map(p =>
new {
    x = p.x * 5,
    y = p.y * 5 }
);
var translated = scaled.Map(p =>
new {
    x = p.x - 20,
    y = p.y - 20
});
var distances = translated.Map(p => new
{
    p.x,
    p.y,
    distance = Math.Sqrt(p.x * p.x + p.y * p.y)
});
var filtered = from location in distances
               where location.distance < 500.0
               select location;
```

The important technique here is to extract those algorithms that can be performed with minimal knowledge of the anonymous type. All anonymous types override Equals() to provide value semantics. So you can assume the existence only of the `System.Object` public members. Nothing has changed here, but you should realize that anonymous types can be passed around to methods only if you also pass around the methods.

In the same vein, you may find that part of the original method will be used in other locations. In those cases, you should factor out those reusable nuggets of code and create a generic method that can be called from both locations.

That points to the need to be careful of taking these techniques too far. Anonymous types should not be used for types that are essential to many of

your algorithms. The more often you find yourself using the same type and the more processing you're doing with that type, the more likely it is that you should convert that anonymous type into a concrete type. Any recommendation is arbitrary, but I suggest that if you're using the same anonymous type in more than three major algorithms, it would be better to convert it into a concrete type. If you find yourself creating longer and more complicated lambda expressions in order to continue to use an anonymous type, that should be a flag to create a concrete type.

Anonymous types are lightweight types that simply contain read and write properties that usually hold simple values. You will build many of your algorithms around these simple types. You can manipulate anonymous types using lambda expressions and generic methods. Just as you can create private nested classes to limit the scope of types, you can exploit the fact that the scope of an anonymous type is limited to a given method. Through generics and higher-order functions, you can create modular code using anonymous types.

Item 9: Understand the Relationships Among the Many Different Concepts of Equality

When you create your own types (either classes or structs), you define what equality means for that type. C# provides four different functions that determine whether two different objects are "equal":

```
public static bool ReferenceEquals
    (object left, object right);
public static bool Equals
    (object left, object right);
public virtual bool Equals(object right);
public static bool operator ==(MyClass left, MyClass right);
```

The language enables you to create your own versions of all four of these methods. But just because you can doesn't mean that you should. You should never redefine the first two static functions. You'll often create your own instance `Equals()` method to define the semantics of your type, and you'll occasionally override `operator==()`, typically for performance reasons in value types. Furthermore, there are relationships among these four functions, so when you change one, you can affect the behavior of the others. Yes, needing

four functions to test equality is complicated. But don't worry—you can simplify it.

Of course, those four methods are not the only options for equality. Types that override `Equals()` should implement `IEquatable<T>`. Types that implement value semantics should implement the `IStructuralEquality` interface. That means six different ways to express equality.

Like so many of the complicated elements in C#, this one follows from the fact that C# enables you to create both value types and reference types. Two values of a reference type are equal if they refer to the same object, referred to as object identity. Two references of a value type are equal if they are the same type and they contain the same contents. That's why equality tests need so many different methods.

Let's start with the two static functions you can't override. `Object.ReferenceEquals()` returns true if two references refer to the same object—that is, the two references have the same object identity. Whether the types being compared are reference types or value types, this method always tests object identity, not object contents. Yes, that means that `ReferenceEquals()` always returns false when you use it to test equality for value types. Even when you compare a value type to itself, `ReferenceEquals()` returns false. This is due to boxing, which is covered in Item 45.

```
int i = 5;
int j = 5;
if (Object.ReferenceEquals(i, j))
    WriteLine("Never happens.");
else
    WriteLine("Always happens.");

if (Object.ReferenceEquals(i, i))
    WriteLine("Never happens.");
else
    WriteLine("Always happens.");
```

You'll never redefine `Object.ReferenceEquals()` because it does exactly what it is supposed to do: tests the object identity of two different references.

The second function you'll never redefine is `static Object.Equals()`. This method tests whether two references are equal when you don't know the runtime type of the two arguments. Remember that `System.Object` is the ultimate base class for everything in C#. Anytime you compare two variables, they are instances of `System.Object`. Instances of value types and reference types are instances of `System.Object`. So how does this method test the equality of two references, without knowing their type, when equality changes its meaning depending on the type? The answer is simple: This method delegates that responsibility to one of the types in question. The static `Object.Equals()` method is implemented something like this:

```
public static bool Equals(object left, object right)
{
    // Check object identity
    if (Object.ReferenceEquals(left, right) )
        return true;
    // both null references handled above
    if (Object.ReferenceEquals(left, null) ||
        Object.ReferenceEquals(right, null))
        return false;
    return left.Equals(right);
}
```

This example code introduces a method I have not discussed yet: namely, the instance `Equals()` method. I'll explain that in detail, but I'm not ready to end my discussion of the `staticEquals()` just yet. For right now, I want you to understand that `static Equals()` uses the instance `Equals()` method of the left argument to determine whether two objects are equal.

As with `ReferenceEquals()`, you'll never overload or redefine your own version of the static `Object.Equals()` method because it already does exactly what it needs to do: determines whether two objects are the same when you don't know the runtime type. Because the `static Equals()` method delegates to the left argument's instance `Equals()`, it uses the rules for that type.

Now you understand why you never need to redefine the `static ReferenceEquals()` and `static Equals()` methods. It's time to discuss the methods you will override. But first, let's briefly discuss the mathematical properties of an equality relation. You need to make sure that your definition and implementation are consistent with other programmers' expectations. Your

unit tests for types that override Equals() should ensure your implementation obeys these properties. This means that you need to keep in mind the mathematical properties of equality: Equality is reflexive, symmetric, and transitive. The reflexive property means that any object is equal to itself. No matter what type is involved, a == a is always true. The symmetric property means that order does not matter: If a == b is true, b == a is also true. If a == b is false, b == a is also false. The last property is that if a == b and b == c are both true, then a == c must also be true. That's the transitive property.

Now it's time to discuss the instance `Object.Equals()` function, including when and how you override it. You create your own instance version of `Equals()` when the default behavior is inconsistent with your type. The `Object.Equals()` method uses object identity to determine whether two references are equal. The default `Object.Equals()` function behaves exactly the same as `Object.ReferenceEquals()`. But wait—value types are different. `System.ValueType` does override `Object.Equals()`. Remember that `ValueType` is the base class for all value types that you create (using the `struct` keyword). Two references of a value type are equal if they are the same type and they have the same contents. `ValueType.Equals()` implements that behavior. Unfortunately, `ValueType.Equals()` does not have an efficient implementation. `ValueType.Equals()` is the base class for all value types. To provide the correct behavior, it must compare all the member fields in any derived type, without knowing the runtime type of the object. In C#, that means using reflection. As you'll see in Item 41, there are many disadvantages to reflection, especially when performance is a goal. Equality is one of those fundamental constructs that gets called frequently in programs, so performance is a worthy goal. Under almost all circumstances, you can write a much faster override of `Equals()` for any value type. The recommendation for value types is simple: Always create an override of `ValueType.Equals()` whenever you create a value type.

You should override the instance `Equals()` function only when you want to change the defined semantics for a reference type. A number of classes in the .NET Framework Class Library use value semantics instead of reference semantics for equality. Two string objects are equal if they contain the same contents. Two `DataRowView` objects are equal if they refer to the same `DataRow`. The point is that if your type should follow value semantics

(comparing contents) instead of reference semantics (comparing object identity), you should write your own override of instance `Object.Equals()`.

Now that you know when to write your own override of `Object.Equals()`, you must understand how you should implement it. The equality relationship for value types has many implications for boxing and is discussed in Item 9 in Effective C#, third edition. For reference types, your instance method needs to follow predefined behavior to avoid strange surprises for users of your class. Whenever you override `Equals()`, you'll want to implement `IEquatable<T>` for that type. I'll explain why a little further into this item. Here is the standard pattern for overriding `System.Object.Equals`. The highlight shows the changes to implement `IEquatable<T>`.

```
public class Foo : IEquatable<Foo>
{
    public override bool Equals(object right)
    {
        // check null:
        // this reference is never null in C# methods.
        if (object.ReferenceEquals(right, null))
            return false;

        if (object.ReferenceEquals(this, right))
            return true;

        // Discussed below.
        if (this.GetType() != right.GetType())
            return false;

        // Compare this type's contents here:
        return this.Equals(right as Foo);
    }

    // IEquatable<Foo> Members
    public bool Equals(Foo other)
    {
        // elided.
        return true;
    }
}
```

First, `Equals()` should never throw exceptions—it doesn't make much sense. Two references are or are not equal; there's not much room for other failures.

Just return false for all failure conditions, such as null references or the wrong argument types. Now, let's go through this method in detail so you understand why each check is there and why some checks can be left out. The first check determines whether the right-side object is null. There is no check on this reference. In C#, this is never null. The CLR throws an exception before calling any instance method through a null reference. The next check determines whether the two object references are the same, testing object identity. It's a very efficient test, and equal object identity guarantees equal contents.

When you look at the first check, where you see if `right` is `null`, you realize you can't satisfy the transitive property. `a.Equals(b)` returns `false` when `a` is non-null and `b` is `null`. However, `b.Equals(a)` will throw a `NullReferenceException` for those values.

The next check determines whether the two objects being compared are the same type. The exact form is important. First, notice that it does not assume that `this` is of type `Foo`; it calls `this.GetType()`. The actual type might be a class derived from Foo. Second, the code checks the exact type of objects being compared. It is not enough to ensure that you can convert the right-side parameter to the current type. That test can cause two subtle bugs. Consider the following example involving a small inheritance hierarchy:

```
public class B : IEquatable<B>
{
    public override bool Equals(object right)
    {
        // check null:
        if (object.ReferenceEquals(right, null))
            return false;

        // Check reference equality:
        if (object.ReferenceEquals(this, right))
            return true;

        // Problems here, discussed below.
        B rightAsB = right as B;
        if (rightAsB == null)
            return false;

        return this.Equals(rightAsB);
    }
```

```
        // IEquatable<B> Members

        public bool Equals(B other)
        {
            // elided
            return true; // or false, based on test
        }
    }

    public class D : B, IEquatable<D>
    {
        // etc.
        public override bool Equals(object right)
        {
            // check null:
            if (object.ReferenceEquals(right, null))
                return false;

            if (object.ReferenceEquals(this, right))
                return true;

            // Problems here.
            D rightAsD = right as D;
            if (rightAsD == null)
                return false;

            if (base.Equals(rightAsD) == false)
                return false;

            return this.Equals(rightAsD);
        }

        // IEquatable<D> Members
        public bool Equals(D other)
        {
            // elided.
            return true; // or false, based on test
        }
    }

//Test:
B baseObject = new B();
D derivedObject = new D();

// Comparison 1.
if (baseObject.Equals(derivedObject))
    WriteLine("Equals");
```

```
else
    WriteLine("Not Equal");

// Comparison 2.
if (derivedObject.Equals(baseObject))
    WriteLine("Equals");
else
    WriteLine("Not Equal");
```

Under any possible circumstances, you would expect to see either Equals or Not Equal printed twice. Because of some errors, this is not the case with the previous code. The second comparison will never return true. The base object, of type `B`, can never be converted into a `D`. However, the first comparison might evaluate to true. The derived object, of type `D`, can be implicitly converted to a type `B`. If the `B` members of the right-side argument match the `B` members of the left-side argument, `B.Equals()` considers the objects equal. Even though the two objects are different types, your method has considered them equal. You've broken the symmetric property of Equals. This construct broke because of the automatic conversions that take place up and down the inheritance hierarchy.

When you write this, the `D` object is explicitly converted to a `B`:

```
baseObject.Equals(derivedObject)
```

If `baseObject.Equals()` determines that the fields defined in its type match, the two objects are equal. On the other hand, when you write this, the `B` object cannot be converted to a `D` object:

```
derivedObject.Equals(baseObject)
```

The `derivedObject.Equals()` method always returns false. If you don't check the object types exactly, you can easily get into this situation, in which the order of the comparison matters.

All of the examples above also showed another important practice when you override `Equals()`. Overriding `Equals()` means that your type should implement `IEquatable<T>`. `IEquatable<T>` contains one method: `Equals(T other)`. Implemented `IEquatable<T>` means that your type also supports a type-safe equality comparison. If you consider that the `Equals()` should return

true only in the case where the right-hand side of the equation is of the same type as the left side, `IEquatable<T>` simply lets the compiler catch numerous occasions where the two objects would be not equal.

There is another practice to follow when you override `Equals()`. You should call the base class only if the base version is not provided by `System.Object` or `System.ValueType`. The previous code provides an example. Class `D` calls the `Equals()` method defined in its base class, Class `B`. However, Class `B` does not call `baseObject.Equals()`. It calls the version defined in `System.Object`, which returns true only when the two arguments refer to the same object. That's not what you want, or you wouldn't have written your own method in the first place.

The guidance is to override `Equals()` when you create a value type, and to override `Equals()` on reference types when you do not want your reference type to obey reference semantics, as defined by `System.Object`. When you write your own `Equals()`, follow the implementation just outlined. Overriding `Equals()` means that you should write an override for `GetHashCode()`. See Item 10 for details.

We're almost done. `operator==()` is simple. When you create a value type, you should probably redefine `operator==()`. The reason is exactly the same as with the instance `Equals()` function. The default version uses reflection to compare the contents of two value types. That's far less efficient than any implementation that you would write, so write your own. Follow the recommendations in Item 9 in Effective C# to avoid boxing when you compare value types.

Notice that I didn't say that you should write `operator==()` whenever you override instance `Equals()`. I said to write `operator==()` when you create value types. You should rarely override `operator==()` when you create reference types. The .NET Framework classes expect `operator==()` to follow reference semantics for all reference types.

Finally, you come to `IStructuralEquality`, which is implemented on `System.Array` and the `Tuple<>` generic classes. It enables those types to implement value semantics without enforcing value semantics for every comparison. It is doubtful that you'll ever create types that implement

`IStructuralEquality`. It is needed only for those lightweight types. Implementing `IStructuralEquality` declares that a type can be composed into a larger object that implements value-based semantics.

C# gives you numerous ways to test equality, but you need to consider providing your own definitions for only two of them, along with supporting the analogous interfaces. You never override the `static Object.ReferenceEquals()` and `static Object.Equals()` because they provide the correct tests, regardless of the runtime type. You always override instance `Equals()` and `operator==()` for value types to provide better performance. You override instance `Equals()` for reference types when you want equality to mean something other than object identity. Anytime you override `Equals()` you implement `IEquatable<T>`. Simple, right?

Item 10: Understand the Pitfalls of GetHashCode()

This is the only item in this book dedicated to one function that you should avoid writing. `GetHashCode()` is used in one place only: to define the hash value for keys in a hash-based collection, typically the `HashSet<T>` or `Dictionary<K,V>` containers. That's good because there are a number of problems with the base class implementation of `GetHashCode()`. For reference types, it works but is inefficient. For value types, the base class version is often incorrect. But it gets worse. It's entirely possible that you cannot write `GetHashCode()` so that it is both efficient and correct. No single function generates more discussion and more confusion than `GetHashCode()`. Read on to remove all that confusion.

If you're defining a type that won't ever be used as the key in a container, this won't matter. Types that represent window controls, Web page controls, or database connections are unlikely to be used as keys in a collection. In those cases, do nothing. All reference types will have a hash code that is correct, even if it is very inefficient. Value types should be immutable (see Item 20), in which case, the default implementation always works, although it is also inefficient. In most types that you create, the best approach is to avoid the existence of `GetHashCode()` entirely.

One day, you'll create a type that is meant to be used as a hash key, and you'll

need to write your own implementation of `GetHashCode()`, so read on. Hash-based containers use hash codes to optimize searches. Every object generates an integer value called a hash code. A hash-based container uses these values internally. A hash-based container separates the objects it stores into "buckets". Each bucket stores objects that match a set of hash values. When an object is stored in a hash container, its hash code is computed. That determines the correct bucket for the object. To retrieve for an object, you request its key and search just that one bucket. The point of all this is to increase the performance of searches. Ideally, each bucket contains a very small number of objects. In .NET, every object has a hash code, determined by System.`Object.GetHashCode()`. Any overload of `GetHashCode()` must follow these three rules:

1. If two objects are equal (as defined by the instance `Equals() method`), they must generate the same hash value. Otherwise, hash codes can't be used to find objects in containers.

2. For any object `A`, `A.GetHashCode()` must be an instance invariant. No matter what methods are called on `A`, `A.GetHashCode()` must always return the same value. That ensures that an object placed in a bucket is always in the right bucket.

3. The hash function should generate a uniform distribution among all integers for all typical input sets. You want to avoid clustering around sets of values. That's how you get efficiency from a hash-based container. You want to create buckets with very few objects in them.

Writing a correct and efficient hash function requires extensive knowledge of the type to ensure that rule 3 is followed. The versions defined in `System.Object` and `System.ValueType` do not have that advantage. These versions must provide the best default behavior with almost no knowledge of your particular type. `Object.GetHashCode()` uses an internal field in the `System.Object` class to generate the hash value.

Now examine `Object.GetHashCode()` in light of those three rules. If two objects are equal, `Object.GetHashCode()` returns the same hash value.. `System.Object`'s version of `operator==()` tests object identity. `GetHashCode()` returns the internal object identity field. It works. However, if

you've supplied your own version of `Equals()`, you must also supply your own version of `GetHashCode()` to ensure that the first rule is followed. See Item 6 for details on equality.

The second rule is followed: After an object is created, its hash code never changes.

The third rule, a uniform distribution among all integers for all inputs, holds reasonably well for `System.Object`. It does the best that it can without specific knowledge of the derived types.

Before covering how to write your own override of `GetHashCode`, this section examines `ValueType.GetHashCode()` with respect to those same three rules. `System.ValueType` overrides `GetHashCode()`, providing the default behavior for all value types. Its version returns the hash code from the first field defined in the type. Consider this example:

```
public struct MyStruct
{
    private string msg;
    private int id;
    private DateTime epoch;
}
```

The hash code returned from a `MyStruct` object is the hash code generated by the msg field. The following code snippet always returns true, assuming msg is not null:

```
MyStruct s = new MyStruct();
s.SetMessage("Hello");
return s.GetHashCode() == s.GetMessage().GetHashCode();
```

The first rule says that two objects that are equal (as defined by `Equals()`) must have the same hash code. This rule is followed for value types under most conditions, but you can break it, just as you could with for reference types. `Equals()` compares the first field in the `struct`, along with every other field. That satisfies rule 1. As long as any override that you define for `operator==` uses the first field, it will work. Any `struct` whose first field does not participate in the equality of the type violates this rule, breaking `GetHashCode()`.

The second rule states that the hash code must be an instance invariant. That rule is followed only when the first field in the `struct` is an immutable field. If the value of the first field can change, so can the hash code. That breaks the rules. Yes, `GetHashCode()` is broken for any `struct` that you create when the first field can be modified during the lifetime of the object. It's yet another reason why immutable value types are your best bet (see Item 20).

The third rule depends on the type of the first field and how it is used. If the first field generates a uniform distribution across all integers, and the first field is distributed across all values of the `struct`, then the `struct` generates an even distribution as well. However, if the first field often has the same value, this rule is violated. Consider a small change to the earlier `struct`:

```
public struct MyStruct
{
    private DateTime epoch;
    private string msg;
    private int id;
}
```

If the `epoch` field is set to the current date (not including the time), all `MyStruct` objects created in a given date will have the same hash code. That prevents an even distribution among all hash code values.

Summarizing the default behavior, `Object.GetHashCode()` works correctly for reference types, although it does not necessarily generate an efficient distribution. (If you have overridden `Object.operator==()`, you can break `GetHashCode()`). `ValueType.GetHashCode()` works only if the first field in your `struct` is read-only. `ValueType.GetHashCode()` generates an efficient hash code only when the first field in your `struct` contains values across a meaningful subset of its inputs.

If you're going to build a better hash code, you need to place some constraints on your type. Ideally, you'd create an immutable value type. The rules for a working `GetHashCode()` are simpler for immutable value types than they are for unconstrained types. Examine the three rules again, this time in the context of building a working implementation of `GetHashCode()`.

First, if two objects are equal, as defined by `Equals()`, they must return the

same hash value. Any property or data value used to generate the hash code must also participate in the equality test for the type. Obviously, this means that the same properties used for equality are used for hash code generation. It's possible to have properties participate in equality that are not used in the hash code computation. The default behavior for `System.ValueType` does just that, but it often means that rule 3 usually gets violated. The same data elements should participate in both computations.

The second rule is that the return value of `GetHashCode()` must be an instance invariant. Imagine that you defined a reference type, Customer:

```
public class Customer
{
    private decimal revenue;

    public Customer(string name) =>
        this.Name = name;

    public string Name { get; set; }

    public override int GetHashCode() =>
        Name.GetHashCode();
}
```

Suppose that you execute the following code snippet:

```
Customer c1 = new Customer("Acme Products");
myHashMap.Add(c1, orders);
// Oops, the name is wrong:
c1.Name = "Acme Software";
```

`c1` is lost somewhere in the hash map. When you placed `c1` in the map, the hash code was generated from the string "Acme Products". After you change the name of the customer to "Acme Software", the hash code value changed. It's now being generated from the new name: "Acme Software". `c1` is stored in the bucket based on the value "Acme Products", but it should be in the bucket based on the value "Acme Software". You've lost that customer in your own collection. It's lost because the hash code is not an object invariant. You've changed the correct bucket after storing the object. It is still in the container, but the code looks in the wrong bucket now.

The earlier situation can occur only if `Customer` is a reference type. Value types misbehave differently, but they still cause problems. If customer is a value type, a copy of `c1` gets stored in the hash map. The last line changing the value of the name has no effect on the copy stored in the hash map. Because boxing and unboxing make copies as well, it's very unlikely that you can change the members of a value type after that object has been added to a collection.

The only way to address rule 2 is to define the hash code function to return a value based on some invariant property or properties of the object. `System.Object` abides by this rule using the object identity, which does not change. `System.ValueType` hopes that the first field in your type does not change. You can't do better without making your type immutable. When you define a value type that is intended for use as a key type in a hash container, it must be an immutable type. If you violate this recommendation, then the users of your type will find a way to break hashtables that use your type as keys. Revisiting the `Customer` class, you can modify it so that the customer name is immutable. The highlight shows the changes to make a customer's name immutable:

```
public class Customer
{
    private decimal revenue;

    public Customer(string name) => this.Name = name;

    public string Name { get; }

    public decimal Revenue { get; set; }

    public override int GetHashCode() =>
        Name.GetHashCode();

    public Customer ChangeName(string newName) =>
        new Customer(newName) { Revenue = revenue };
}
```

`ChangeName()` creates a new `Customer` object, using the constructor and object initialize syntax to set the current revenue. Making the name immutable changes how you must work with customer objects to modify the name:

```
Customer c1 = new Customer("Acme Products");
myDictionary.Add(c1, orders);
// Oops, the name is wrong:
Customer c2 = c1.ChangeName("Acme Software");
Order o = myDictionary[c1];
myDictionary.Remove(c1);
myDictionary.Add(c2, o);
```

You have to remove the original customer, change the name, and add the new `Customer` object to the dictionary. It looks more cumbersome than the first version, but it works. The previous version allowed programmers to write incorrect code. By enforcing the immutability of the properties used to calculate the hash code, you enforce correct behavior. Users of your type can't go wrong. Yes, this version is more work. You're forcing developers to write more code, but only because it's the only way to write the correct code. Make certain that any data members used to calculate the hash value are immutable.

The third rule says that `GetHashCode()` should generate a random distribution among all integers for all inputs. Satisfying this requirement depends on the specifics of the types you create. If a magic formula existed, it would be implemented in `System.Object`, and this item would not exist. A common algorithm is to XOR all the return values from `GetHashCode()` on all fields in a type. If your type contains some mutable fields, exclude those fields from the calculations.This algorithm will only be successful if there is no correlation among the fields in your type. When they do correlate, this algorithm clusters hash codes so that you container would have few buckets, and those buckets would have many items. Two examples from the .NET framework illustrate the importance of this rule. The `GetHashCode()` implementation for `int` returns that `int`. Those are anything but random. They will cluster. The implementation for `DateTime` XORs the high and low 32 bits of the 64 bit internal ticks field. This will cluster less. Putting these facts together, imagine building a hash function for a student class that had names and birthdates as fields. If you use the `GetHashCode()` implementation from `DateTime`, you'll get much better results than if you build something that computes a hash code from the year, month, and date fields: for students, the year component will likely cluster around the most common ages of students. Knowledge of your data set is critical to building a proper `GetHashCode()` method.

`GetHashCode()` has very specific requirements: Equal objects must produce

equal hash codes, and hash codes must be object invariants and must produce an even distribution to be efficient. All three can be satisfied only for immutable types. For other types, rely on the default behavior, but understand the pitfalls.

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 that want to use your types to do so correctly. Robust API design takes into account many aspects of the types you create. In includes how developers can create instances of a type. It includes how you choose to expose its capabilities through methods and properties. In includes how an object reports changes through events or outbound method calls. And 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 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. 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. It works because a circle is a specific type of 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 language also supports a variety of numeric conversions.

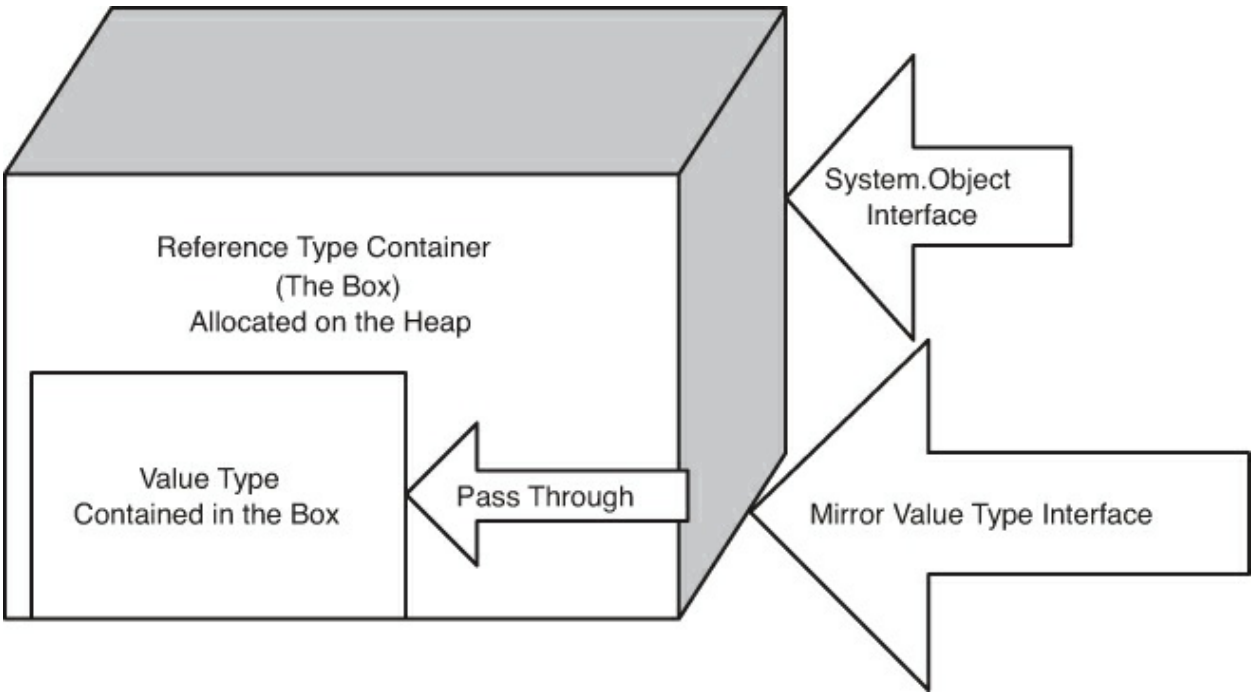
When you define a conversion operator for your type, you tell the compiler that your type may be substituted for the target type. These substitutions often result in subtle errors because your type probably isn't a perfect substitute for the target type. Side effects that modify the state of the target type won't have the same effect on your type. 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, not the runtime type of an object. 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 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 you believe that, although the `Circle` and `Ellipse` are related, you don't want to have nonabstract leaf classes in your hierarchy, and several implementation problems occur when you try to derive the `Circle` class from the `Ellipse` class. However, you realize that every circle could be an ellipse. In addition, some ellipses could be substituted for circles.

That leads you to add two conversion operators. Every `Circle` is an `Ellipse`, so you add an implicit conversion to create a new `Ellipse` from a `Circle`. An implicit conversion operator will be called whenever one type needs to be converted to another type. By contrast, an explicit conversion will be called only when the programmer puts a cast operator in the source code.

Figure 2.1 Basic shape hierarchy.



```
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 sample shows what I mean by substitutability: A circle has been substituted for an ellipse. The `ComputeArea` function works even with the substitution. You got lucky. But examine 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. 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 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 only 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() are 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 have not 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. They have other problems. 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's best avoided for all the reasons discussed in Item 26.

Conversion operators introduce a form of substitutability that causes problems in your code. You're indicating 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, you cause clients to work with temporary objects or internal fields in place of the class you created. You then modify temporary objects and discard the results. These subtle bugs are hard 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. That means 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. That means you should avoid using named parameters in many situations, and also you should avoid changing the names of the formal parameters on 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. This small 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. Optional parameters means that these Office APIs can create default values for all those locations where Type.Missing would be used. That simplifies even this small snippet:

```
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. . .");
```

Even this small change increases the readability of this snippet. Of course, you may not always want to use all the defaults. And yet, you still don't want to add all the `Type.Missing` parameters in the middle. Suppose you wanted to create a new Web page instead of new Word document. That's the last

parameter of four 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 only need to specify those parameters you intend to use. It's simpler than 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 named and optional parameters provide. Remember that some of the Office APIs have as many as 16 parameters, and optional and named parameters are a big help.

I left 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. I did not include the `ref` modifier there, because that would be clearly misleading. In fact, in most production code, I would not include the `ref` modifier on the call to Add() either. I did above so that you could see the actual API signature.

Of course, just because the justification for named and optional parameters was COM and the Office APIs, that doesn't mean you should 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.

This method:

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

```
}
```

Can be called using named parameters to avoid any confusion on the order:

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

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

Changing parameter names manifests itself in an interesting way as a breaking change. The parameter names are stored in the MSIL only at the callsite, not at the calling site. You can change parameter names and release the component without breaking any users of your component in the field. The developers who use your component will see a breaking change when they go to compile 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 continue to run, even if they contain calls to SetName that specify named parameters. However, when client developers went to build updates to their assemblies, any code like this would no longer compile:

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

The parameter names have changed.

Changing the default value also requires callers to recompile in order 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 want to upset the developers who use your components either. For that reason, you must consider the names of your parameters as 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 break at runtime. Optional parameters are implemented in a similar fashion to named parameters. The callsite will contain annotations in the MSIL that reflect the existence of default values, and what those default values are. The calling site substitutes those values for any optional parameters the caller did not explicitly specify.

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

Now, after that explanation, the guidance 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. They are now part of your public interface.

Item 13: Limit Visibility of Your Types

Not everybody needs to see everything. 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 standalone 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 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. The enumerator class being private gives many advantages. First, the `List<T>` class can completely replace the type implementing `IEnumerator<T>`, and you'd be none the wiser. Nothing breaks. You can use the enumerator by knowing 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. By default, most programmers create public classes all the time, without any thought to the alternativesInstead of unthinkingly accepting the default, 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 26). 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, exchange.
        return true;
    }
}
```

Months pass, and this class works fine. Then you get a request to handle international phone numbers. The previous `PhoneValidator` fails. It was coded to handle only U.S. phone numbers. You still need the U.S. Phone Validator, but now you need to use an international version in one installation. Rather than stick the extra functionality in this one class, you're better off reducing 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, exchange.
        return true;
    }
}
```

Finally, you can 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, I prefer the implementation using public interfaces because there is little, if any, shared functionality. Other uses would be better served with public abstract base classes. Either way you implement it, 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 mocks 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 has its place, but it is a different place. Interfaces are 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," and interfaces 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 it behaves.

Interfaces describe a set of functionality, or 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 identify and factor reusable behavior into interfaces. You use interfaces as parameters and return values. You also have more chances 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 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. You are declaring the binary contract that must be supported by all types that implement an interface. However, you can create extension methods on those interfaces to give the illusion of an implementation for interfaces. 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. You saw this in Item 8:

```
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: By adding and implementing functionality in the base class, all derived classes immediately incorporate that behavior. Adding a member to an interface 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. Instead, if you find that you need to add functionality to an interface, create a new one and 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. Base classes can be extended over time. Those extensions 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. However, those additional methods are not formally defined in the `IEnumerable<T>` interface. That means class developers do not need to create their own implementation of all those methods.

Examine this class that 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 direct
    {
        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
}

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();
}
```

The `WeatherStream` class models a sequence of weather observations. To do that it 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 would naturally be drawn to the generic interface over the version typed as `System.Object`.

Implementing 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. The query above translates to the following calls:

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

In the code above, the Where and Select calls look like they belong to IEnumerable<WeatherData>. They do not. Those methods appear to belong to IEnumerable<WeatherData> because they are extension methods. They are actually static methods in System.Linq.Enumerable. The compiler translates those calls into the following static calls:

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

I wrote that last version to show you that interfaces really can't contain implementation. You can emulate that by using extension methods. LINQ does that by creating several extension methods on `IEnumerable<T>` in the class.

That brings me 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 to other developers than coding to base class types. That's important because of the single inheritance hierarchy that

the .NET type system enforces.

These 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 final 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>();
```

That 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.

The second problem 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. That's 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 you intend clients to use. That would mean returning an `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 the 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 26).

Furthermore, unrelated types can implement the same interface. Suppose you're building an application that manages employees, customers, and vendors. Those are unrelated, at least in terms of the class hierarchy. But 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 one routine works for all entities that implement the `IContactInfo` interface. 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));
}
```

The example above makes use of two new features in C# 7. The initial condition is a pattern matching is expression. It tests if `obj` is a `URLInfo`, and if so, assigns it to the variable `other`. The other new feature is a throw expression. In cases where `obj` is not a `URLInfo`, that throws an exception. 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 have fewer times when boxing and unboxing is necessary 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. If you understand the differences, you will 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 seems to be the same as overriding an abstract function. You provide a definition for a member that has been declared in another type. That first glance is very deceiving. 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. They are different concepts with different uses.

But you can implement interfaces in such a manner 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 I had to add the new keyword to the definition of the previous Message method (see Item 33). `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`: 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 if that method is virtual or not. Let's review the language rules for implementing interfaces. When a class declaration contains interfaces in its base types the compiler determines what 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. Remember that virtual and abstract members are considered to be members of the type that declares them, not the type which overrides them.

But you often want to create interfaces, implement them in base classes, and modify the behavior in derived classes. You can. You've got two options. 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 on 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 the 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` this way:

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

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

Any derived class can override `OnMessage()` and add their own work to the `Message()` method declared in `MyClass`. It's a pattern you've seen before when classes implement `IDisposable` (see Item 17).

Explicit interface implementation (see Item 31) 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. You use explicit interface implementation to limit client code from using the interface methods when a more appropriate version is available. The `IComparable` idiom in Item 31 shows this in detail.

There is also one last wrinkle to add to working with interfaces and base classes. A base class can provide a default implementation for methods in an interface. Then, a derived class can declare that it implements an interface and inherit the implementation from that base class, as this 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 allows more options than 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 provide virtual method calls in the methods that implement interfaces. You can decide exactly how and when derived classes can modify the default behavior for members of any interface your class implements. Interface methods are not virtual methods but 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*, Gamma, Helm, Johnson, and Vlissides pp. 293–303.) Events define the notifications for your type. Events 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 developers start thinking that events and delegates are the same things. In Item 24, I showed you 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. 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 gets raised only when there are listeners attached to the event.

I've defined `LoggerEventArgs` to hold 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 C# compiler creates the add and remove accessors for the event. The versions generated by the compiler uses a different add and assign construct that is guaranteed to be thread safe. I find the public event declaration language more concise and easier to read and maintain than the add/remove syntax. When you create events in your class, declare public events and let the compiler 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 contained only one event. There are classes (mostly Windows controls) that have very large numbers of events. In those cases, the idea of using one field per event might be unacceptable. In some cases, only a small number of the defined events is actually used in any one application. If you encounter that situation, you can modify the design to create the event objects only when needed at runtime.

The core framework contains examples of how to do this in the Windows control subsystem. To show you how, add subsystems to the `Logger` class. You create an event for each subsystem. Clients 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 that specifies the subsystem generating the log message. If the subsystem has any listeners, the event gets raised. Also, if an event listener has registered an interest in all messages, its event gets 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);
        }
    }
}
```

This new example stores the individual event handlers in the `EventHandlerList` collection. Sadly, there is no generic version of `EventHandlerList`. Therefore, 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. 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 on 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 all its event fields. Each event field internally accesses a collection of objects to add

and remove the particular handlers. You can find more information that shows 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 exception if 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. I'd 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. 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 dataset. You created properties to hide your internal data structures. 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 a read-write property, where you would consider these issues, but a 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 will innocently misuse your APIs and create bugs they later blame you for. More sinister developers will maliciously probe your libraries for ways to exploit them. Don't provide functionality that you did not intend to provide. It won't be tested, or hardened against malicious use.

You've got four different strategies for protecting 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 the copy as much as necessary to achieve their purposes. This does not affect your internal state.

Immutable types, such as `System.String`, are also safe (see Item 6. You can return strings, or any immutable type, safely knowing that no client of your class can modify the string. Your internal state is safe.

The third option is to define interfaces that allow clients to access a subset of your internal member's functionality (see Item 22). 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 changes 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. The Machiavellian programmers out there can defeat that byusing 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. These are the actions that developers looking to misuse your work will do. You should do what you can to make it harder for them to use your work to exploit end users.

There is one strange twist in the `BindingList` class that may cause some problems. There isn't a generic version of `IBindingList`, so you may want to create two different API methods for accessing the data: one that supports databinding via the `IBindingList` interface, and one that supports programming through the `ICollection<T>`, or 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 take a brief look at how you can respond to changes in your data when you allow public clients to modify it. This is important because you'll often want to export an `IBindingList` to UI controls so that the user can edit the data. You've undoubtedly already 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 anytime you want to expose internal data elements for modification by public clients, but you need to validate and respond to those 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 25.)

Going back 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 by setting various properties on the `BindingList` object (`AddEdit`, `AllowNew`, `AllowRemove`, etc.). The values of these properties are honored by UI controls. The UI controls enable and disable different actions based on the value of these properties. These are public properties, so that you can modify the behavior of your collection. But that also means you should not expose the `BindingList<T>` object as a public property. Clients could modify those properties and circumvent your intent to make 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 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. 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 internals without going through the methods and properties you've defined. That seems counterintuitive, which makes it a common mistake. You need to modify your class's interfaces to take into account 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 call any method that is available in your members. You limit that access by exposing 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. You can attach an event handler, or you can override a virtual function in the base class. Why provide two ways of doing the same thing? Because different situations call for different methods, that's why. Inside derived classes, you should always override the virtual function. Limit your use of the event handlers to responding to events in unrelated objects.

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);
    }
}
```

Or, 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, MouseButtonE
    {
        DoMouseThings(e);
    }
}
```

You should prefer the first solution. This may 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 are called (see Items 24 and 25). Some other ill-formed code prevents the system from calling your event handler. By overriding the protected `virtual` function, your handler gets called first. The `base` class version of the `virtual` function is responsible for calling any event handlers attached to the particular event. That means that if you want the event handlers called (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 none of the event handlers gets 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's not enough for you, examine the first listing in this item again. Which is clearer? Overriding a virtual function has one function to examine and modify if you need to maintain the form. 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.

Okay, I've been giving all these reasons to use the overrides and not use the event handlers. The .NET Framework designers must have added events for a reason, right? Of course they did. Like the rest of us, they're too busy to write code nobody uses. The overrides are 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 this example, your designer may have actions that are supposed to occur on a MouseDown event. The designer will create XAML declarations for those behaviors. Those behaviors will be accessed using events on the form. You could redefine all that behavior in your code, but that's way too much work to handle one event. It only moves the problem from the designer's hands to yours. You clearly want designers doing design work instead of you. The obvious way to handle that is to create an event and access the XAML declarations created by a design tool. So, in the end, you 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 the event mechanism is that events are wired up at runtime. You have more flexibility using events. You can wire up different event handlers, depending on the circumstances of the program. Suppose that you write a drawing program. Depending on the state of the program, a mouse down 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, handle the event 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. You might have multiple event handlers hooked up on the MouseDown event. The first would perform the particular action. The second might update the status bar or update the accessibility of different commands. Multiple actions can take place in response to the same event.

When you have one function that handles one event in a derived class, the

override is the better approach. It is easier to maintain, more likely to be 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. And yet, there are many other reasons why a derived class may want to use the same name. 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 33 covered why using the `new` modifier could 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 I'm not sure anyone could 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. That increases the chance of ambiguity. It increases the chance that your interpretation of the spec is different than the compiler's, 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, and yet people always question if it really should be so strict. Maybe that's because overloading sounds very much

like overriding. Overriding virtual methods is such a core principle of C based object-oriented languages; that's obviously not what I mean. Overloading means creating multiple methods with the same name and different parameter lists. Does overloading base class methods really have that much of an effect on overload resolution? Let's look at the different ways where overloading methods in the base class can cause issues.

There are a lot of permutations to this problem. Let's start simple. The interplay between overloads in base classes has a lot to do with base and derived classes used for parameters. The samples 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");
}
```

Now, 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”. However, even the simple overload rules can be surprising. The reason both calls resolve to `Tiger.Foo` is that when there is a candidate method in the most derived compile-time type, that method is the better method. That’s still true when there is even a better match 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 this does:

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

I chose the words above very carefully because `obj3` has the compile-time type of `Animal` (your Base class), even though the runtime type is `Tiger` (your Derived class). `Foo` isn’t virtual; therefore, `obj3.Foo()` must resolve to `Animal.Foo`.

If your poor users actually want to get the resolution rules they might expect, they 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 users, you’ve failed. 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");
}
```

Hopefully, you've already seen what will happen here. This 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 show the kinds of problems you can get into with one parameter method. The issues become more and 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");
}
```

Then, 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 before:

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

obj2.Baz(sequence);
```

What do you suppose gets printed this time? If you've been paying attention, you'd figure that "In Tiger.Foo2" gets printed. That answer gets you partial credit. That is what happens in C# 4.0. Starting in C# 4.0, 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>`. However, earlier versions of C# do not support generic variance. 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 samples above showed 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 make the method you want, not the method the compiler picks as the "best" method. But the fact that real-world situations are occasionally ugly 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#. It 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 to be able to use your API. Instead, don't overload methods declared in a base class. It doesn't

provide any value, and it will only 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. There are coupling issues 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
    }
}
```

Now, every subscriber to that event is coupled. Suppose you have multiple subscribers on a single event. One subscriber might request a cancel, and the second 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 here, but it might not work in some 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. Or 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, there's another form of coupling 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. (Remember that the contract of `IDisposable` is that no other methods should be called after an object is disposed. See Item 17 in Effective C#.)

As a result, event subscribers need to modify their implementation of the dispose pattern to unhook event handlers as part of the `Dispose()` method. . Otherwise, subscriber objects continue to live on because reachable delegates exist in the event source object. It's another case where runtime coupling can cost you. Even though it appears that there is looser coupling because the

compile-time dependencies are minimized, runtime coupling does have costs.

Event-based communication loosens the static coupling between types, but it 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 it's as easy as declaring any other C# language element as virtual. Unfortunately, because you can declare events using field-like syntax as well as `add` and `remove` syntax, it's not that simple. It's remarkably simple 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 the preceding item 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. You can invoke it only through the publicly accessible event declaration. That restriction, obviously, also applies to derived events. You can't manually write code that accesses the private backing field of the base class. However, the compiler can access its own generated fields, so 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.

Therefore, 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. 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 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 to contain 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, never use field-like syntax. You can't use field-like syntax 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;
    }
}
```

Of course, when you look at this code, you'll see 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 args 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, there is both compile-time and runtime coupling 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. Worse, when you make what seem to be innocent changes to your code, you can cause different methods to be called and therefore 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 greater 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 the 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 chances for 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 chooses 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 and to know which methods will be called. The more possible variations you add, the worse the situation gets, and the more likely it is that you can create ambiguity.

Several conditions change the way the compiler resolves these methods. The 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, you get a compiler error. But those are the easy cases. You can't ship code that has compiler errors. The hard 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.

I begin by noting that any methods having 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 the 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 have avoided introducing any ambiguity. Every numeric type except `decimal` is listed, and therefore the compiler always calls the version that is a correct match. (`Decimal` has been omitted because a conversion from `decimal` to `double` requires an explicit conversion.) If you have a C++ background, you probably wonder why I haven't recommended replacing all those overloads with a single generic

method. That's because 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 2, Chapter 1). Of course, you might think about using delegates to define a method constraint (see Item 6, Chapter 1). But in this case, that technique only 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.

However, 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 the `double` cases. There are implicit conversions from `short` to `float`, and from `short` to `double`. Which one will the compiler pick? And if it can't pick one method, you've forced coders to specify an explicit cast to get their code to compile. Here, 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, gloat 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;
    }
}
```

Now, what happens if you call with `int,float`? Or with `int,long`?

```
Point p = new Point { X = 5, Y = 7 };
// Note that 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 in trying to determine which method gets called.

But wait—method lookup can get a lot 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 method signatures that conflict, 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)`. The third call goes to a generic method for `Max<float>`. That's because for generic methods, one of the types can always be a perfect match, 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 does an accessible member function? Thankfully, extension methods are a last resort. They are only examined if no applicable instance method is found.

As you can see, the compiler examines quite a few places to find candidate methods. 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 for your users. If only one in twenty developers can correctly identify which method overload gets called when he invokes one of a series of overloaded methods, you've clearly made your API too complex. Users should be able to know immediately 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 only 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. This means that you need to provide plenty of hooks for those other developers. You should implement those hooks in the form of partial methods. Partial methods let you provide hooks that another developer may, or may not, need to implement.

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, the compiler 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: 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 as 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 internals.

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 would be called after changing state and allows the other class author to respond to the state change.

Your tool's core code would be 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 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 `UpdateValue()` compiles down to 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}");
    }
}
```

Here, I show a protocol with a cancel flag that lets developers cancel any mutator operation. Your class may prefer a protocol in which the user-defined code can throw an exception to cancel an operation. Throwing the exception is better if the cancel operation should be propagated up to the calling code. Otherwise, the Boolean cancel flag should be used because it's lightweight.

Furthermore, notice that the `RequestChange` object gets created 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 don't remember those, other class authors can't validate or respond to property changes.

You next need to make sure to provide hooks for user-generated code in constructors. Neither the generated code nor the user-written code can control which constructor gets called. Therefore, your code generator must 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 I make `Initialize()` the last method called during construction. That enables the hand-written code to examine the current object state and possibly make any modifications or throw exceptions if the developer finds something invalid for his problem domain. You want to make sure that you don't call `Initialize()` twice, and you must make sure it is called from every constructor defined in the generated code. The human developer must not call his own `Initialize()` routine from any constructor he adds. Instead, he 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 is especially important if the event is one of the events that request 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 I show here, you should never need to modify code generated by a tool. You are probably using code generated by Visual Studio or

other tools. Before you consider modifying any of the code written by the tool, you must examine the interface provided by the generated code in hopes that 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. But 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. Finally, 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 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 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 a simpler class. It's 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 so that it can be coerced into a `System.Object` reference. The caller must 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
}
```

`string` is a special case because this class is immutable. If you assign an error message object, both error message objects 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 16).

The general case of creating a `struct` that contains arbitrary reference fields is more complicated. It's also far rarer. The built-in assignment for the `struct` creates a shallow copy, with both `struct`s 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 supported a deep copy with its Clone() method. Even then, that will only work 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 could 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 only clones 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 23). 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 devived 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`; 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 provides the necessary methods for any derived classes that want `ICloneable` support.

`ICloneable` does have its use, but it is the exception rather than rule. It's significant that the .NET Framework did not add an `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 have strict type checking but fail at runtime. The following small program compiles just fine. It 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;
}
```

The 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 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 examine 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 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());
```

But 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'll end up being 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 those problems by typing parameters as interface types that create a type-safe sequence to use. Input parameters should be typed 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 APIs that support modifying the collection. When you write a method where 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 in your API signature. (See chapter 2 on API design 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 17, Chapter 3). When you want to generate the sequence, return the sequence as an `IEnumerable<T>` for some `T`.

And yet there are times when you want to pass arbitrary options in methods. That's when you can reach for an array of arguments. But 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);
}
```

You can see that 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 won't compile:

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

Trying `null` 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. However, you're less likely to run into the problem. 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. 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 developer using your code needs to write truly pathological constructs. She would need to create an actual array of a different type. Then she 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 universally wrong method parameters, but they can cause two types of errors. The array's covariance behavior causes 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 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. But nothing is ever free. Both iterator methods and async methods delay execution of the code you write in those methods. This initial code is often argument checking and object validation code that should throw

exceptions immediately if a method was called incorrectly, or at the wrong time. That doesn't happen 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. However, 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 the same class. That can make it much more difficult to find and diagnose the problem. The exception is thrown in code that's unrelated to the code that 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 does 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 at 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. You'd want developers using your library incorrectly to see mistakes reported when they are made so that those mistakes are easy to fix. The way to do that 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`. Any method 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 seqeunce 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 does 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 You should limit the accessibility of this method as much as you can. 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 has 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 seqeunce 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. You ensure that there is no way to bypass the validation code and call the implementation method directly. However, there are other advantages. Notice 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 have the async modifier. This wrapper method does all the validation and eagerly reports any errors. The implementation method has the async modifier and does the async 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: 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 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, where local functions can often be implemented as private methods.

High-level constructs like 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 do that, make sure you limit the accessible of the implementation method that does not contain any of the error checking.

3. Task based Asynchronous Programming

More of our programming tasks involve starting and responding to asynchronous work. We work on distributed programs, running on multiple machines or virtual machines. Many applications span threads, processes, containers, virtual machines or physical machines. Asynchronous programming is not synonymous with multi-threaded programming. Modern programming means mastering asynchronous work. That work may include awaiting for the next network packet, or awaiting for user input.

The C# language, along with some classes in the .NET framework provide tools that make asynchronous programming easier. Asynchronous programming can be hard, but by remembering a few important practices, it's easier than it has ever been.

Item 27: Use Async methods for async work

Async methods are an easier way to construct asynchronous algorithms. You write the core logic for an async method as though it were a synchronous method. However, the execution path is not the same as a synchronous method. You write sequences of instructions and you expect those instructions to execute in order, just like you wrote them. That's not necessarily the case with async methods. Async methods may return before executing all the logic you wrote. Then, at some later time in response to a task completing, the method picks up execution where it left off, while your program has continued in its normal flow. Without careful understanding, it's magic. With a little understanding, it becomes very confusing and generates more questions than it answers. Read on to fully understand how the compiler transforms your code into async methods. You'll learn how to analyze async code by appreciating the core algorithms it describes, and have the skills to understand how the code executes as it follows through those instructions and tasks.

Let's start with the simplest example: an async method that actually executes synchronously. Consider this method:

```
static async Task SomeMethodAsync()
{
    Console.WriteLine("Entering SomeMethodAsync");
    Task awaitable = SomeMethodReturningTask();

    Console.WriteLine("In SomeMethodAsync, before the await");
    var result = await awaitable;
    Console.WriteLine("In SomeMethodAsync, after the await");
}
```

In some cases, the asynchronous work may complete before the first task is awaited. The library designer may have designed a cache, and you may be retrieving a value already loaded. When you await the initial task, the task has already completed and execution continues synchronously on the next instruction. The remainder of the method executes to completion, and the result is packaged in a Task object and returned. It all happens synchronously. Now, when this method returns, it returns a completed Task, and the caller will also continue synchronously when the caller awaits this task. So far, this is familiar to any developer.

But what happens in that same method if the result isn't available when the task is awaited? Then, the flow of control gets more complicated. Before `async` and `await` language support, you'd have to configure some callback to process the return from an asynchronous task. This could be either an event handler, or a delegate of some kind. Now, it's much easier. First, let's just look at what happens conceptually, without any concern for how the language implements this behavior.

When the `await` is reached, the method returns. It returns a Task object that indicates the asynchronous work has not yet completed. Here's where the magic happens. When the awaited task completes, this method continues execution on the next instruction after the await. It will continue to do its work, and upon completion of that work, update the Task object that it returned earlier with the completed result. This task now notifies any code awaiting it that it has completed. Those can now also continue where they were interrupted by awaiting for this task. The best way I understand the control flow is to walk through some samples in a debugger. Step through code with await expressions and see how the execution flow proceeds.

You may also find an analogy with real world asynchronous tasks useful.

Consider the tasks for making a homemade pizza. You start by synchronously making the dough. Then, you can start an asynchronous process to let the dough rise. After you've started that task, you can continue to make the sauce. Once you've made the sauce, you can await the completion of the dough rising task. Then, you can start an asynchronous task to heat up the oven. While it starts, you can assembly the pizza. Finally, after awaiting the oven reaching the correct temperature, you put the pizza in the oven to cook.

Now, let's remove the magic by explaining how is implemented. When the compiler processes an async method, it builds mechanisms to start asynchronous work, and continue further instructions when that async work has completed. The interesting changes are in the `await` expression. The compiler builds data structures and delegates so that execution can continue at the next instruction following the await expression. The data structures ensure that all local variables have their values preserved. The compiler configures a continuation on the awaited task such that the continuation jumps back into the method in the same location when the task is completed. Effectively, the compiler generates a delegate for the code that follows the await. The compiler writes state to ensure that when the awaited task completes, the delegate gets invoked.

When the awaited task completes, it raises the event to indicate it has completed. This method is re-entered and the state is restored. The code appears to pick up where it left off. The state was restored and execution jumped to the appropriate location. This is similar to what happens when execution continues after a synchronous call: the state is set for that method, and execution continues at the point following the method call. When the remainder of the method executes, it completes its work, updates the previously returned `Task`, and raises the events that completed.

When the task completes, the notification mechanism calls the async method and it continues execution. The `SynchronizationContext` class is responsible for implementing this behavior. This class is responsible for making sure that when an asynchronous method resumes after an awaited task completes, the environment and the context are compatible with where it was when the awaited task paused. Effectively, the context "brings you back where you were." The compiler generates the code to use the

`SynchronizationContext` to bring you back. Before an async method begins, the compiler caches the current `SynchronizationContext`, using the static `Current` property. When the awaited task resumes, the compiler posts the remaining code as a delegate to that same `SynchronizationContext`. The `SynchronizationContext` schedules the work using the appropriate means for the environment. In a GUI application, the `SynchronizationContext` will use the Dispatcher to schedule the work (See Item 19). In a web context, the `SynchronizationContext` will use the Thread Pool and `QueueUserWorkItem` (See Item 27). In a console application, where there is no `SynchronizationContext`, the work continues on the current thread. Notice that some contexts have multiple threads, and others have single threads and schedule work cooperatively.

If the awaited asynchronous task has faulted, the exception that faulted the task is thrown in the code posted to the `SynchronizationContext`. The exception is thrown when that continuation executes. That means tasks which are not awaited will not have any exceptions observed when they have faulted. Their continuations are not scheduled, and the exception was caught, but never re-thrown in the `SynchronizationContext`. For that reason, it's always important to await any Tasks you start: it's the best way to observe any exceptions that are thrown from the asynchronous work.

This same strategy is extended for methods that have multiple await expressions. Each await may cause the async method to return to the caller with the task still uncompleted. The internal state is updated so that when the routine is continued again, execution begins at the correct spot. As with the single await, the synchronization context determines how the remaining work is scheduled: either on the single thread in the context, or on a different thread.

The language writes the same kind of code that you would write to register for notifications when asynchronous work completes. It does it in a standard manner, and that makes it easier to read the code as though it was synchronous.

The path I've described up to this point assumes that all asynchronous work completes successfully. We know that doesn't always happen. Sometimes, exceptions are thrown. Async methods must handle those conditions as well. That complicates control flow as well, because an async method may have returned to its caller before completing all its work. It must somehow reinject

any exceptions up into the call stack. Inside an async method, the compiler generates a `try` / `catch` block that catches all exceptions. Any and all exceptions are stored in an `AggregateException` that is a member of the Task object, and the Task is set to the Faulted state. When a faulted Task is awaited, the await expression throws the first exception in the aggregate exception object. In the most common case, there is only one, and that gets thrown in the caller's context. In cases where there may be multiple exceptions, the caller would need to unpackage the aggregate exception and examine each. (See Item N).

This asynchronous mechanism can be overridden by using certain `Task` APIs. If you really must wait for a `Task` to complete, you can call the `Task.Wait()` API, or you can examine the `Task<T>.Result` property. Either of those block until all asynchronous work has completed. That can be useful for a `Main()` method in a console application. See Item 27 for how these can cause deadlocks and why they should be avoided.

The compiler doesn't perform magic when you create asynchronous methods using the async and await keywords. It does a lot of work to generate a lot of code to handle continuations, error reporting, and resuming methods. The benefits of all this compiler manipulation is that work appears to pause when asynchronous work is not completed. It resumes when that asynchronous work is ready. This pause can travel as far up the call stack as needed, as long as `Tasks` are awaited. The magic works well, unless you override it.

Item 28: Never Write Async void Methods

The title of this item is strong, and there are a small number of exceptions to its advice as you'll see below. But this advice is stated so strongly because it is so important. When you write an `asyncvoid` method, you defeat the protocol that enables exceptions thrown by async methods to be caught by the methods that started the asynchronous work. Asynchronous methods report exceptions through the `Task` object. When an exception is thrown, the `Task` enters the faulted state. When you await a faulted task, the await expression throws the exception. When you await a task that faults later, the exception is thrown when the method is scheduled to resume.

`Asyncvoid` methods cannot be awaited. There's no way to for the code that calls an `asyncvoid` method to catch or propagate an exception thrown from the async method. Don't write `asyncvoid` methods because errors are hidden from callers.

Code in an `asyncvoid` method may generate an exception. Something must happen with those exceptions. The code generated for async void methods throws any exceptions directly on the `SynchronizationContext` (See Item N) that was active when the `asyncvoid` method started. That makes it much harder for developers using your library to process those exceptions. You must use the `AppDomain.UnhandledException` or some similar out-of-band catch-all handler. Note that `AppDomain.UnhandledException` does not enable you to recover from the exception. You can log it, and possibly save data, but you cannot prevent the uncaught exception from terminating the application. Consider this method:

```
private static async void FireAndForget()
{
    var task = DoAsyncThings();
    await task;
    var task2 = ContinueWork();
    await task2;
}
```

If you wanted to log errors before calling FireAndForget(), you'd need to setup the unhandled exception handler. This example writes the exception information to the console in the Cyan color:

```
AppDomain.CurrentDomain.UnhandledException += (sender, e) =>
{
    Console.ForegroundColor = ConsoleColor.Cyan;
    Console.WriteLine(e.ExceptionObject.ToString());
};
```

There's really no need to set the console's `ForegroundColor` back to its original color, as the application will terminate.

Forcing your developers to use a completely different error handling mechanism from all their other code is bad API design. You know that many developers will neglect this extra work. It's even worse to give them no way to

recover from any errors. If developers don't do the extra work, any exceptions generated from async void methods are unreported. The runtime aborts the thread running in the synchronization context regardless. Developers using your code will get no notifications, no catch handlers are triggered, no exception logging happens. The thread just goes away.

In addition to the exception behavior, `async void` methods have other difficulties. In many async methods, you'll start async work, await that task, and then do more work after the first awaited task has finished. Async tasks compose easily. `Async void` methods cannot be awaited. Therefore, developers using your async methods cannot easily know when an `async void` method has finished all its work. That easy composition is no longer possible. `Async void` methods are essentially 'fire and forget' methods: developers start async work, but do not and cannot easily know when that work finishes.

These same issues complicate the process of testing async methods. Automated tests cannot know when an `async void` method has completed. Therefore, the automated test cannot be written to check for any effects from the `async void` method running to completion. Consider writing an automated unit test for this method:

```
public async void SetSessionState()
{
    var config = await ReadConfigFromNetwork();
    this.CurrentUser = config.User;
}
```

To write a test, you might consider code like the following:

```
var t = new SessionManager();
t.SetSessionState();
// wait a while
await Task.Delay(1000);
Assert.Equal(t.User, "TestLibrary User");
```

There are bad practices here, and in fact, they may not always work. The key is that `Task.Delay` call. You can't write this test resiliently. You don't have any indication exactly when the asynchronous work ends. Maybe one second is enough. Maybe not. Worse, maybe it is in most cases, but on rare events, it is not. Then, your tests fail and provide false feedback.

`Async void` methods are bad. Whenever possible, you should create async methods that return `Tasks` or other awaitable objects (See Item 32). But, `async void` methods are allowed because without `async void` methods, you cannot create async event handlers.

The protocol for event handlers, where event handlers are void returning methods, was established before `async` and `await` support were added to the language. Even if changes were made, you would still require `async void` methods to attach async event methods events that had been defined in earlier versions. In addition, the library author may not know if an event handler requires asynchronous access. Talking all these into consideration, the C# language supports void returning async methods. Also, callers of event handlers are typically not user code. It won't know what to do with a returned Task, so why require it?

Even though I said to never write `async void` methods, you'll one day find that you must write an `async void` event handler. If you must write one, let's write an async event handler as safely as possible.

Let's start with the fact that `async void` methods can't be awaited. The code that raised the event won't know when your event handler has finished running. Event handlers typically do not return data to the caller, so the caller can 'fire and forget' when raising events.

Handling any potential exceptions safely requires more work. If any exceptions are thrown from your `async void` method, the synchronization context will be killed. You must write your `async void` event handler so that no exceptions are thrown from this method. This may go against other recommendations, but this is one idiom where you usually want to catch all exceptions. The pattern for a typical async void event handler becomes something like this:

```
private async void OnCommand(object sender, RoutedEventArgs e)
{
    var viewModel = (DataContext as SampleViewModel);
    try
    {
        await viewModel.Update();
    }
    catch (Exception ex)
    {
```

```
            viewModel.Messages.Add(ex.ToString());
        }
}
```

This assumes you feel safe simply logging any exceptions and continuing normal execution. That may be safe in many scenarios. If that's true for your scenario, you're done.

But what if some exceptions that might be thrown in this event handler are catastrophic conditions that can't be handled? Maybe they can cause serious data corruption. You want to terminate the program immediately, rather than continue blithely along and further corrupt data. That's the case where you do want to throw the exception and have the system abort the thread on that synchronization context.

To do that, you'll likely want to log everything, and throw the exception from the async void method. That's a slight modification to the earlier code:

```
private async void OnCommand(object sender, RoutedEventArgs e)
{
    var viewModel = (DataContext as SampleViewModel);
    try
    {
        await viewModel.Update();
    }
    catch (Exception ex) when (logMessage(viewModel, ex))
    {
    }
}
private bool logMessage(SampleViewModel viewModel, Exception ex
{
    viewModel.Messages.Add(ex.ToString());
    return false;
}
```

This method logs every exception by using an exception filter (see Effective C# Item N) to log information about the exception, and then rethrows the exception to cause the synchronization context to be stopped, possibly stopping the program as well.

Both of these methods can be generalized using a Func argument to represent the async work you in each of these methods. That enables you to reuse the

common elements from these two idioms.

```
public static class Utilities
{
    public static async void FireAndForget(this Task task,
        Action<Exception> onErrors)
    {
        try
        {
            await task;
        }
        catch (Exception ex)
        {
            onErrors(ex);
        }
    }

    public static async void FireAndForget(this Task task,
             Func<Exception, bool> onError)
    {
        try
        {
            await task;
        }
        catch (Exception ex) when (onError(ex))
        {
        }
    }
}
```

The real world may not always be as simple as catching all exceptions or rethrowing all exceptions. In many real-world applications, you may be able to recover from some exceptions but not others. For example, you may be able to recover from a FileNotFoundException, but no others. This can be made more general and reusable by replacing the specific exception type with a generic type:

```
public static async void FireAndForget<TException>
    (this Task task,
    Action<TException> recovery, Func<Exception, bool> onError)
    where TException : Exception
{
    try
    {
        await task;
```

```
    }
    // relies on onError() logging method
    // always returning false:
    catch (Exception ex) when (onError(ex))
    {
    }
    catch (TException ex2)
    {
        recovery(ex2);
    }
}
```

You can extend that same technique to more exception types if you'd like.

These techniques help make `async void` methods a little more robust in terms of error recovery. They haven't helped with testability, or composability. There aren't good techniques to alleviate those issues. That's why you should limit the use of `async void` methods to the locations where you must write them: in event handlers. Everywhere else, never write `async void` methods.

Item 29: Avoid Composing Synchronous and Asynchronous Methods

Declaring a method with the async modifier declares that this method may return before completing all its work. The returned object represents the state of that work: completed, faulted, or still pending. It further declares that any pending work may take long enough that callers are advised to await the result while doing other useful work.

Declaring a synchronous method declares that when it completes, all of its post conditions have bee met. Regardless of the time the method takes to execute, it does all its work using the same resources as the caller. The caller blocks until completion.

Mixing these clear statements represent poor API design and leads to bugs, including deadlock. That leads to two important rules. First, don't create synchronous methods that block waiting for asynchronous work to complete. Second, avoid async methods to offload long running CPU bound work.

Let's start with the first rule. There are three reasons why composing synchronous code on top of asynchronous code causes problems: different exception semantics, possible deadlocks, and resource consumption.

Asynchronous tasks may cause multiple exceptions, so the Task class contains a list of exceptions that have been thrown. When you await a task, the first exception in that list is thrown, if there are any exceptions. However, when you call `Task.Wait()`, or access `Task.Result`, the containing `AggregateException` is thrown for a Faulted task. You have to catch the `AggregateException` and unwrap the exception that was thrown. Compare these two `try` / `catch` clauses:

```
public static async Task<int> ComputeUsageAsync()
{
    try
    {
        var operand = await GetLeftOperandForIndex(19);
        var operand2 = await GetRightOperandForIndex(23);
        return operand + operand2;
    }
    catch (KeyNotFoundException e)
    {
        return 0;
    }
}

public static int ComputeUsage()
{
    try
    {
        var operand = GetLeftOperandForIndex(19).Result;
        var operand2 = GetRightOperandForIndex(23).Result;
        return operand + operand2;
    }
    catch (AggregateException e)
    when (e.InnerExceptions.FirstOrDefault().GetType()
        == typeof(KeyNotFoundException))
    {
        return 0;
    }
}
```

Notice the difference in semantics on handling the exceptions. The version where the task is awaited is much more readable than the version where a

blocking call is used. The awaited version catches the specific type of exception whereas the blocking version must catch an aggregate exception and apply an exception filter to catch only when the first contained exception matches the sought exception types. The idioms needed for the blocking APIs are more complicated, and harder for other developers to understand.

Let's move on to how composing synchronous methods over asynchronous code can lead to deadlocks. Consider this code

```
private static async Task SimulatedWorkAsync()
{
    await Task.Delay(1000);
}

// This method can cause a deadlock in ASP.NET or GUI context.
public static void SyncOverAsyncDeadlock()
{
    // Start the task.
    var delayTask = SimulatedWorkAsync();
    // Wait Synchronously for the delay to complete.
    delayTask.Wait();
}
```

Calling `SyncOverAsyncDeadlock()` works fine in a Console application. However, it will deadlock in either a GUI or web context. That's because these different application types make use of different types of synchronization contexts (See Item 29). The `SynchronizationContext` for console applications contains multiple threads from the thread pool, whereas the `SynchronizationContext` for the GUI and ASP.NET contexts contain a single thread. The awaited task, started by `SimulatedWorkAsync()` cannot continue because the only thread available is blocked waiting for the task to complete. You want your APIs to be useful in as many application types as possible. Composing synchronous code on top of asynchronous APIs defeats that. Instead of waiting synchronously for asynchronous work to finish, perform other work while awaiting for the task to finish.

Notice that the example above uses `Task.Delay` instead of `Thread.Sleep` to yield control and simulate a longer running task. This is preferred because `Thread.Sleep` makes your application pay the cost of that thread's resources the entire time it is idle. You made that thread, keep it busy doing useful work.

`Task.Delay` is asynchronous and will enable callers to compose asynchronous work while your task simulates longer running tasks. This can be useful in unit tests to ensure that your tasks finish asynchronously (See Item 33).

There is one common exception to the rule that you should not compose synchronous methods over asynchronous methods: The `Main()` method in a console application. If the `Main()` method were async, it could return before all the work was complete, terminating the program. It's the one location where sync over async is appropriate. Otherwise, it's async all the way up. There is a proposal to allow the `Main` method to be async and handle that situation. There is also a NuGet package, AsyncEx that enables an async main context.

You probably have synchronous APIs in your libraries today that can be updated to be asynchronous APIs. Removing your synchronous API would be a breaking change. I've just convinced you not to convert to an asynchronous API and redirect the synchronous method to block while calling the asynchronous method. That doesn't mean you're stuck supporting only the synchronous API with no path forward. You can create an asynchronous API that mirrors the synchronous code, and support both. Those users that are ready for async will use the async method, and others can continue using the synchronous method. At some later date, you can deprecate the synchronous method. I mention this because developers are beginning to make assumptions about libraries that support both synchronous and asynchronous versions of the same method: They will assume the synchronous method is a legacy method, and the asynchronous method is the preferred method.

This observation brings me to why an async API that wraps a synchronous CPU bound operation is also a bad idea. Developers are beginning to assume that the asynchronous method would be preferred when there are both synchronous and asynchronous versions of the same method. They'll gravitate toward the asynchronous methods. When that is simply a wrapper, you've misled them. Consider the following two methods:

```
public double ComputeValue()
{
    // Do lots of work.
    double finalAnswer = 0;
    for (int i = 0; i < 10_000_000; i++)
```

```
        finalAnswer += InterimCalculation(i);
    return finalAnswer;
}

public Task<double> ComputeValueAsync()
{
    return Task.Run(() => ComputeValue());
}
```

The synchronous version enables callers to decide if they want to run that CPU bound work synchronously, or asynchronously on another thread. The asynchronous method has taken that choice away from them. They are forced to spin up a new thread or retrieve one from a pool, and run this operation on that new thread. If this CPU bound work is part of a larger operation, it may already be on a separate thread. Or, it may be called from a console application, where running on a background thread just ties up more resources.

I'm not saying there is no place for doing CPU bound work on separate threads. I am saying you want that CPU bound work to be a large-grained as possible. The code that starts the background task should be at the entry point to your application. Consider the Main method to a console application, response handlers in a web application, or UI handlers in a GUI application. These are the points in an application where CPU bound work should be dispatched to other threads. Creating asynchronous methods over CPU bound synchronous work in other locations only misleads other developers.

Offloading work using asynchronous methods begins to worm its way through your application as you compose more asynchronous methods on top of other asynchronous APIs. That's exactly as it should be. You'll keep adding more and more async methods in vertical slices up the call stack. If you are converting or extending an existing library, consider running asynchronous and synchronous versions of your API side by side. But only do that when the work is asynchronous and you are offloading work to another resource. If you are adding asynchronous versions of CPU bound methods, you are only misleading your users.

Item 30: Use async Methods to Avoid Thread Allocations and Context Switches

It's easy to begin thinking that all asynchronous tasks represent work being done on other threads. After all, that is one use for asynchronous work. But many times, asynchronous work doesn't start a new thread. File I/O is asynchronous, but uses IO completion ports rather than threads. Web requests are asynchronous, but uses network interrupts rather than threads. In these instances, using async tasks frees a thread to do useful work.

When you offload work to another thread, you free one thread at the cost of creating and running another. That's a wise design only when the thread you are freeing up is a scarce resource. In a GUI application, the UI thread is a scarce resource: only one thread interacts with any of the visual elements the user sees. However, thread pool threads are not unique or scarce (but they are limited in number). One thread is the same as another. For that reason, you should avoid CPU bound async tasks in non-GUI applications.

Let's start with GUI applications. When the user initiates an action from the UI, she expects the UI to remain responsive. It won't be if the UI thread spends seconds (or more) performing the last action. The answer is to offload that work to another resource so that the UI can remain responsive to other user actions. As you saw in Item 35, these UI event handlers are one of the locations where async over sync composition makes sense.

From here, let's move on to console applications. Console applications that perform only one long running CPU bound task would not benefit from executing that work on a separate thread. The main thread would be synchronously waiting, and the worker thread would be busy. You've tied up two threads to do the work of one.

However, if a Console application performs several long running CPU bound operations, it may make sense to run those on separate threads. Item 25 discusses several options to run CPU bound work on multiple threads.

That brings us to ASP.NET Server applications, where developers seem to have much confusion. On the one hand, you want to keep threads free so that your application can handle a greater number of incoming requests. That leads to a design where you would offload the CPU bound work to different threads in your ASP.NET handlers:

```
public async Task<IActionResult> Compose()
{
    var model = await LongRunningCPUTask();
    return View(model);
}
```

But, let's examine what happens in this situation. By starting another thread for the task, you allocate a second thread pool thread. The first thread has nothing to do and can be recycled and given more work. But there's more overhead. In order to "bring you back where you were", the `SynchronizationContext` must keep track of all the state for this web request, and then when the awaited CPU bound work completes, restore that state. Then, the handler can respond to the client.

You haven't freed any resources, but you've added two context switches to processing a request.

If you do have long running CPU bound work to do in response to web requests, you need to offload that work to another process or another machine in order to free up the thread resource and increase your web application's ability to service requests.

For example, you might have a second web job that receives CPU bound requests and executes them in turn. Or, you might allocate a second machine to the CPU bound work.

Which one is the fastest depends on the characteristics of your application: the amount of traffic, the time to do the CPU bound work, and network latency. You must measure these items to make an informed decision. One of the configurations you should measure is doing all the work in the web application synchronously. It will likely be faster than offloading the work to another thread in the same thread pool and process.

Asynchronous work seems like magic. You offload work to another location, and pick up afterward. But you need to make sure that when you offload work, you free resources rather than simply context switch between similar resources.

Item 31: Avoid Marshalling Context Unnecessarily

We refer to code that can run in any synchronization context as “context-free code”. Conde that must run in a specific context is “context aware code”. Most of the code you write is context free code. Context aware code includes code in a GUI application that interacts with UI controls, and code in a web application that interacts with the `HTTPContext` or related classes. When context aware code is executed after an awaited task has completed, it must be executed on the correct context. You saw this mechanism in Item 33. However, all other code can execute in the default context.

With so few locations where code is context aware, you may wonder why the default behavior is to execute continuations on the captured context. It’s because the consequences of switching contexts unnecessarily are much less than the consequences of not switching contexts when it is necessary. If you execute context free code on the captured context, nothing goes drastically wrong. However, if you execute context aware code on the wrong context, your application probably crashes. Therefore the default behavior is to execute the continuation on the captured context, whether it is necessary or not.

When I said that resuming on the captured context did not cause drastic problems, it still causes problems that compound over time. By running continuations on the captured context, you don’t ever take advantage of any opportunities to offload some continuations to other threads. In GUI applications, this can make the UI unresponsive. In web applications, it can limit how many requests per minute the application can manage. Over time, performance degrades. In the case of GUI applications, you increase the chance of deadlock (See Item 35). In web applications, you don’t fully utilize the thread pool.

The way around this is to use `ConfigureAwait()` to indicate that continuations do not need to run on the captured context. In library code, where the continuation is context free code, you would use it like this:

```
public static async Task<XElement> ReadPacket(string Url)
{
    var result = await DownloadAsync(Url)
        .ConfigureAwait(continueOnCapturedContext: false);
```

```
    return XElement.Parse(result);
}
```

In simple cases, it's easy. You add ConfigureAwait() and your continuation will run on the default context. Consider this method:

```
public static async Task<Config> ReadConfig(string Url)
{
    var result = await DownloadAsync(Url)
        .ConfigureAwait(continueOnCapturedContext: false);
    var items = XElement.Parse(result);
    var userConfig = from node in items.Descendants()
                     where node.Name == "Config"
                     select node.Value;
    var configUrl = userConfig.SingleOrDefault();
    if (configUrl != null)
    {
        result = await DownloadAsync(configUrl)
            .ConfigureAwait(continueOnCapturedContext: false);
        var config = await ParseConfig(result)
            .ConfigureAwait(continueOnCapturedContext: false);
        return config;
    }
    else
        return new Config();
}
```

You might think that once the first await is reached, the continuation runs on the default context, so `ConfigureAwait`() is not needed on any of the subsequent async calls. That assumption may be wrong. What if the first task completes synchronously? Remember from Item 33 that this would mean the work continues synchronously on the captured context. Execution reaches the next await while still on that original context. These subsequent calls don't continue on the default context, so all the work continues on the captured context.

For this reason, whenever you call a `Task` returning async method and the continuation is context free code, you should use `ConfigureAwait(false)` to continue on the default context. Your goal is to isolate the context aware code to code that must manipulate the UI. Consider this method:

```
private async void OnCommand(object sender, RoutedEventArgs e)
{
    var viewModel = (DataContext as SampleViewModel);
```

```
    try
    {
        var userInput = viewModel.webSite;
        var result = await DownloadAsync(userInput);
        var items = XElement.Parse(result);
        var userConfig = from node in items.Descendants()
                         where node.Name == "Config"
                         select node.Value;
        var configUrl = userConfig.SingleOrDefault();
        if (configUrl != null)
        {
            result = await DownloadAsync(configUrl);
            var config = await ParseConfig(result);
            await viewModel.Update(config);
        }
        else
            await viewModel.Update(new Config());
    }
    catch (Exception ex) when (logMessage(viewModel, ex))
    {
    }
}
```

This method is structured so that it is hard to isolate the context aware code. There are several asynchronous methods called from this method. Most are context free. However, the code toward the end of the method updates UI controls and is context aware. You should treat all awaitable code as context-free, unless it will update user interface controls. Only code that updates the user interface is context-aware.

As written, this method must run all its continuations on the captured context. Once any continuations start on the default context, there's no easy way back. The first step is to restructure the code so that all the context free code is moved to a new method. Then, once that restructuring is done, you can add the ConfigureAwait(false) calls on each method to run the asynchronous continuations on the default context:

```
private async void OnCommand(object sender, RoutedEventArgs e)
{
    var viewModel = (DataContext as SampleViewModel);
    try
    {
        Config config = await ReadConfigAsync(viewModel);
        await viewModel.Update(config);
```

```
    }
    catch (Exception ex) when (logMessage(viewModel, ex))
    {
    }
}

private async Task<Config> ReadConfigAsync(SampleViewModel
    viewModel)
{
    var userInput = viewModel.webSite;
    var result = await DownloadAsync(userInput)
        .ConfigureAwait(continueOnCapturedContext: false);
    var items = XElement.Parse(result);
    var userConfig = from node in items.Descendants()
                     where node.Name == "Config"
                     select node.Value;
    var configUrl = userConfig.SingleOrDefault();
    var config = default(Config);
    if (configUrl != null)
    {
        result = await DownloadAsync(configUrl)
            .ConfigureAwait(continueOnCapturedContext: false);
        config = await ParseConfig(result)
            .ConfigureAwait(continueOnCapturedContext: false);
    }
    else
        config = new Config();
    return config;
}
```

It might have been simpler if the default were to continue on the default context. But, doing so would have meant that the getting it wrong would lead to crashes. Using the current strategy, your application will run if all continuations use the captured context, but it will run less efficiently. Your users deserve better. Structure your code to isolate the code that must run on the captured context. Use `ConfigureAwait(false)` to continue on the default context whenever possible.

Item 32: Compose Asynchronous Work Using Task Objects

Tasks are the abstraction for the work you've offloaded to another resource. The Task type and related classes and structs have rich APIs to manipulate

Tasks and the work that has been offloaded. Tasks are also objects that can be manipulated using their methods and properties. Tasks compose to form larger-grained tasks. Tasks can be ordered, or run in parallel. You use `await` expressions to enforce an ordering: code that follows the awaited expression will not execute until the awaited task has completed. You can specify that tasks may only be started in response to another task completing. Overall, there is a rich set of APIs used by Tasks that enable you to write elegant algorithms that work with these objects and the work they represent. The more you learn how to use Tasks as objects, the more elegant your asynchronous code will be.

Let's begin with an asynchronous method that starts a number of tasks and awaits for each to finish. A very naïve implementation would be this:

```
public static async Task<IEnumerable<StockResult>>
    ReadStockTicker(IEnumerable<string> symbols)
{
    var results = new List<StockResult>();
    foreach (var symbol in symbols)
    {
        var result = await ReadSymbol(symbol);
        results.Add(result);
    }
    return results;
}
```

These several tasks are independent, and there's no reason for you to wait for each task to finish before starting the next. One change you could make is to start all tasks, and then await for all of them to finish before executing the continuations:

```
public static async Task<IEnumerable<StockResult>>
    ReadStockTicker(IEnumerable<string> symbols)
{
    var resultTasks = new List<Task<StockResult>>();
    foreach (var symbol in symbols)
    {
        resultTasks.Add(ReadSymbol(symbol));
    }
    var results = await Task.WhenAll(resultTasks);
    return results.OrderBy(s => s.Price);
}
```

This would be the correct implementation if the continuation requires the result

of all the tasks to effectively continue. Using `WhenAll`, you create a new `Task` that completes when all the tasks it is watching is completed. The result from `Task.WhenAll` is an array of all the completed (or faulted) tasks.

Other times, you may start several different tasks that are all generating the same result. Your goal is to try different sources, and continue working with the first task that finishes. The `Task.WhenAny()` method creates a new task that is complete as soon as any of the tasks it is awaiting is complete. Suppose you want to read a single stock symbol from multiple online sources, and return the first result that completes. You could use `WhenAny` to determine which of the started tasks completed first:

```
public static async Task<StockResult>
    ReadStockTicker(string symbol, IEnumerable<string> sources)
{
    var resultTasks = new List<Task<StockResult>>();
    foreach (var source in sources)
    {
        resultTasks.Add(ReadSymbol(symbol, source));
    }
    return await Task.WhenAny(resultTasks);
}
```

Other times, you may want to execute the continuation as each task completes. A naïve implementation might look like this:

```
public static async Task<IEnumerable<StockResult>>
    ReadStockTicker(IEnumerable<string> symbols)
{
    var resultTasks = new List<Task<StockResult>>();
    var results = new List<StockResult>();
    foreach (var symbol in symbols)
    {
        resultTasks.Add(ReadSymbol(symbol));
    }
    foreach(var task in resultTasks)
    {
        var result = await task;
        results.Add(result);
    }
    return results;
}
```

There's no guarantee that the tasks finish in the order you've started them. This could be a very inefficient algorithm. Any number of completed tasks may be in the queue to process behind a task that is simply taking longer.

You might try to improve this using Task.WhenAny. The implementation would look like this:

```
public static async Task<IEnumerable<StockResult>>
    ReadStockTicker(IEnumerable<string> symbols)
{
    var resultTasks = new List<Task<StockResult>>();
    var results = new List<StockResult>();
    foreach (var symbol in symbols)
    {
        resultTasks.Add(ReadSymbol(symbol));
    }
    while (resultTasks.Any())
    {
             // Each time through the loop, this creates a
             // new task. That can be expensive.
        Task<StockResult> finishedTask = await
            Task.WhenAny(resultTasks);
        var result = await finishedTask;
        resultTasks.Remove(finishedTask);
        results.Add(result);
    }
    var first = await Task.WhenAny(resultTasks);
    return await first;
 }
```

The comments indicate that this is not a good way to create this behavior. You create new tasks each time you call `Task.WhenAny()`. As the number of tasks you want to manage grows, this algorithm does more and more allocations, and is more inefficient.

Instead, you can use the `TaskCompletionSource` class `TaskCompletionSource` enables you to return a `Task` that you manipulate to produce the result at a later point in time. Effectively, you can produce the result for any method asynchronously. Its most common use is to provide a conduit between a source `Task` (or `Tasks`) and a destination `Task` (or `Tasks`). You write the code that executes when the source task completes. Your code awaits the source task, and updates the destination task using the

`TaskCompletionSource`. For this example, you have an array of source tasks. You'll create an array of destination `TaskCompletionSource` objects. As each task finishes, you'll update one of the destination tasks using its `TaskCompletionSource`. Here's the code:

```
public static Task<T>[] OrderByCompletion<T>(
    this IEnumerable<Task<T>> tasks)
{
    // Copy to List because it gets enumerated multiple times.
    var sourceTasks = tasks.ToList();

    // Allocate the sources, allocate the output tasks.
    // Each output task is the corresponding task from
    // each completion source.
    var completionSources =
        new TaskCompletionSource<T>[sourceTasks.Count];
    var outputTasks = new Task<T>[completionSources.Length];
    for (int i = 0; i < completionSources.Length; i++)
    {
        completionSources[i] = new TaskCompletionSource<T>();
        outputTasks[i] = completionSources[i].Task;
    }

    // Magic, part one:
    // Each task has a continuation that puts its
    // result in the next open location in the completion
    // sources array.
    int nextTaskIndex = -1;
    Action<Task<T>> continuation = completed =>
    {
        var bucket = completionSources
            [Interlocked.Increment(ref nextTaskIndex)];
        if (completed.IsCompleted)
            bucket.TrySetResult(completed.Result);
        else if (completed.IsFaulted)
            bucket.TrySetException(completed.Exception);
    };

    // Magic, part two:
    // for each input task, configure the
    // continuation to set the output task
    // As each task completes, it uses the next location.
    foreach (var inputTask in sourceTasks)
    {
        inputTask.ContinueWith(continuation, CancellationToken.
            TaskContinuationOptions.ExecuteSynchronously,
```

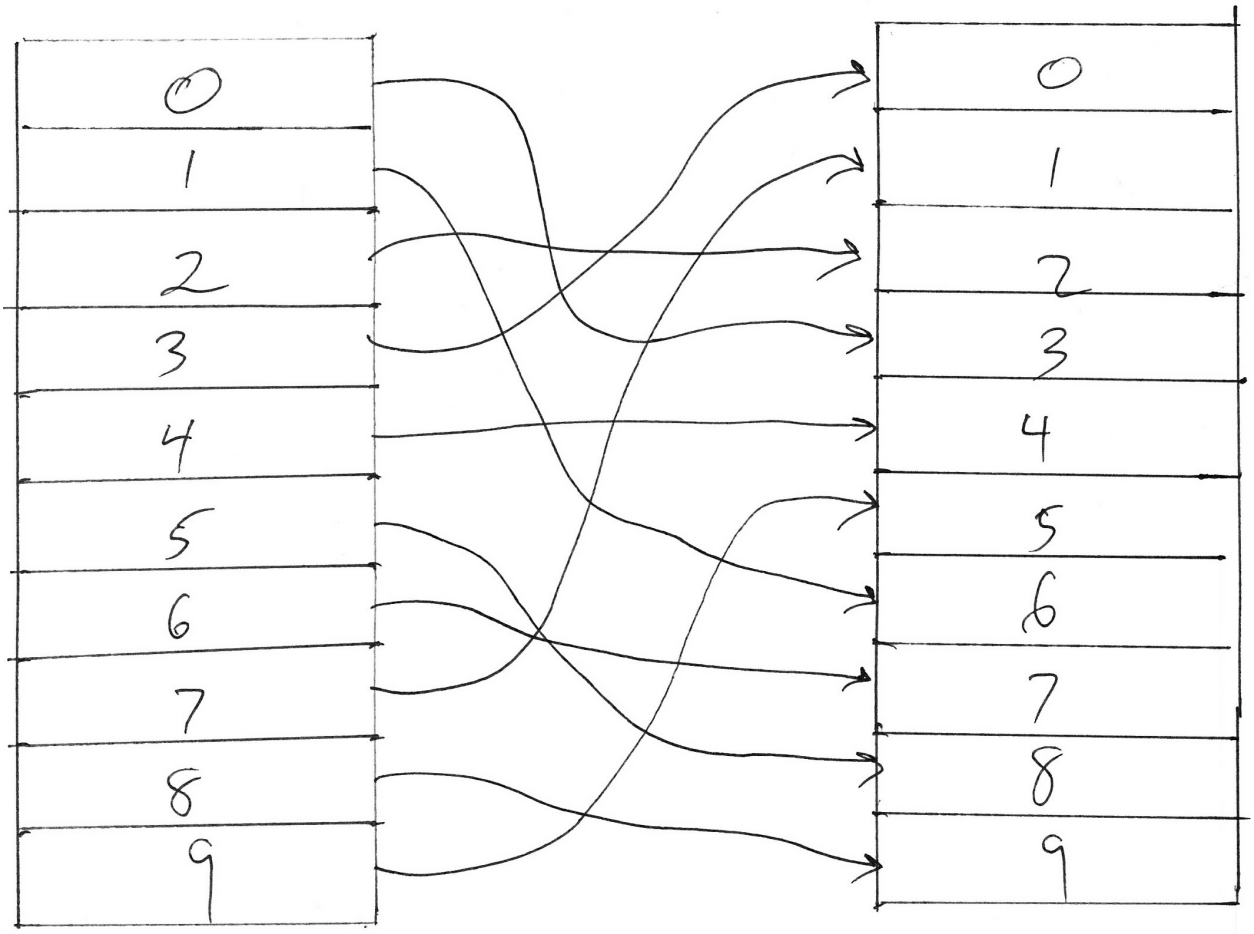
```
            TaskScheduler.Default);
    }

    return outputTasks;
}
```

There's quite a bit going on here, so let's look at it section by section. First, the method allocates an array of `TaskCompletionSource` objects. Then, it defines the continuation code that will run as each source task completes. This continuation code sets the next slot in the destination `TaskCompletionSource` objects to complete. It uses the `InterlockedIncrement()` method to update the next open slot in a thread safe manner. Finally, it sets the continuation for each Task to execute this code.

This method returns the sequence of tasks from the array of `TaskCompletionSources`.

The caller can now enumerate the list of tasks which will be ordered by their completion time. Let's walk through one typical run starting 10 tasks. Let's suppose the tasks finish in this order: 3, 7, 2, 0, 4, 9, 1, 6, 5, 8. When task 3 finishes, its continuation will run, placing its Task result in slot 0 of the destination array. Next, task 7 finishes, placing its result in slot 1. Task 2 places its result in slot 2. This process continues until task 8 finishes, placing its result in slot 9. See Figure 3.1.

Figure 3.1 Ordering Tasks based on their completion



Let's extend the code so that it handles tasks that end up in the faulted state. The only change is in the continuation:

```
// Magic, part one:
// Each task has a continuation that puts its
// result in the next open location in the completion
// sources array.
int nextTaskIndex = -1;
Action<Task<T>> continuation = completed =>
{
    var bucket = completionSources
        [Interlocked.Increment(ref nextTaskIndex)];
    if (completed.IsCompleted)
        bucket.TrySetResult(completed.Result);
    else if (completed.IsFaulted)
        bucket.TrySetException(completed.Exception);
};
```

There are a number of methods and APIs that enable programming with tasks

and enabling actions when tasks complete or fault. Using these methods makes it easier to construct elegant algorithms that process the results of asynchronous code when they are ready. Learn the extensions from the Task library, program actions that take place when tasks finish. You'll ready easily readable code that manipulates tasks as they finish in a very inefficient fashion.

Item 33: Consider Implementing the Task Cancellation Protocol

The Task Asynchronous Programming model includes standard APIs for cancellation and reporting progress. These are optional but should be implemented correctly when the asynchronous work can effectively report progress or be cancelled.

Not every asynchronous task can be cancelled. The underlying mechanism may not support any cancellation protocol. In those cases, your asynchronous API should not support any overloads that indicate cancellation is possible. You don't want callers to do extra work to implement a cancellation protocol when it doesn't have any effect. The same is true for reporting progress. The programming model supports reporting progress. APIs should only implement this protocol when they can report progress. Don't implement the progress reporting overload when you're not able to accurately report how much of the asynchronous work has been done. For example, consider a web request. You won't receive any interim progress from the network stack about delivery of the request, processing of the request, or any other action before receiving the response. Once you've received the response, the task is completed. Progress reporting does not provide any added value. Contrast that with a task that makes a series of 5 web requests to different services to perform a complex operation. Suppose you wrote an api to: process payroll. It might:

1. Call a web service to retrieve the list of employees and their reported hours.

2. Call another web services to calculate tax reporting.

3. Call a third web service to generate paystubs and email them to employees.

4. Call a fourth web service to deposit wages.

5. Close the payroll period.

You might reasonably assume that each of those services represents 20% of the work. You might implement the Progress reporting overload to report progress after each of the 5 steps. Furthermore, you might implement the cancellation API. Up until the fourth step begins, this operation could be cancelled. Once the money has been paid, the operation is no longer cancellable. Let's look at the different overloads you should support. First, let's start with the simplest: running the payroll processing without support for either cancellation or progress reporting:

```
public async Task RunPayroll(DateTime payrollPeriod)
{
    // Step 1: Calculate hours and pay
    var payrollData = await RetieveEmployeePayrollDataFor(
        payrollPeriod);

    // Step 2: Calculate tax
    var taxReporting = new Dictionary<EmployeePayrollData,
        TaxWithholding>();
    foreach(var employee in payrollData)
    {
        var taxWitholding = await RetrieveTaxData(employee);
        taxReporting.Add(employee, taxWitholding);
    }

    // Step 3: generate and email paystub documents
    var paystubs = new List<Task>();
    foreach(var payrollItem in taxReporting)
    {
        var payrollTask = GeneratePayrollDocument(
            payrollItem.Key, payrollItem.Value);
        var emailTask = payrollTask.ContinueWith(
            paystub => EmailPaystub(
                payrollItem.Key.Email, paystub.Result));
        paystubs.Add(emailTask);
    }
    await Task.WhenAll(paystubs);

    // Step 4: Deposit pay
    var depositTasks = new List<Task>();
    foreach(var payrollItem in taxReporting)
    {
        depositTasks.Add(MakeDeposit(payrollItem.Key,
            payrollItem.Value));
```

```
    }
    await Task.WhenAll(depositTasks);
    // Step 5: Close payroll period
    await ClosePayrollPeriod(payrollPeriod);
}
```

Next, let's add the overload that supports progress reporting. Here's what it would look like:

```
public async Task RunPayroll2(DateTime payrollPeriod,
    IProgress<(int, string)> progress)
{
    progress?.Report((0, "Starting Payroll"));
    // Step 1: Calculate hours and pay
    var payrollData = await RetieveEmployeePayrollDataFor(
        payrollPeriod);

    progress?.Report((20, "Retrieved employees and hours"));

    // Step 2: Calculate tax
    var taxReporting = new Dictionary<EmployeePayrollData,
        TaxWithholding>();
    foreach (var employee in payrollData)
    {
        var taxWitholding = await RetrieveTaxData(employee);
        taxReporting.Add(employee, taxWitholding);
    }
    progress?.Report((40, "Calculated Witholding"));

    // Step 3: generate and email paystub documents
    var paystubs = new List<Task>();
    foreach (var payrollItem in taxReporting)
    {
        var payrollTask = GeneratePayrollDocument(
            payrollItem.Key, payrollItem.Value);
        var emailTask = payrollTask.ContinueWith(
            paystub => EmailPaystub(payrollItem.Key.Email,
                paystub.Result));
        paystubs.Add(emailTask);
    }
    await Task.WhenAll(paystubs);
    progress?.Report((60, "Emailed Paystubs"));

    // Step 4: Deposit pay
    var depositTasks = new List<Task>();
    foreach (var payrollItem in taxReporting)
    {
```

```
        depositTasks.Add(MakeDeposit(payrollItem.Key,
            payrollItem.Value));
    }
    await Task.WhenAll(depositTasks);
    progress?.Report((80, "Deposited pay"));

    // Step 5: Close payroll period
    await ClosePayrollPeriod(payrollPeriod);
    progress?.Report((100, "complete"));
}
```

Callers would use this idiom as follows:

```
public class ProgressReporter :
    IProgress<(int percent, string message)>
{
    public void Report((int percent, string message) value)
    {
        WriteLine($"{value.percent} completed: {value.message}'
    }
}

await generator.RunPayroll(DateTime.Now, new ProgressReporter()
```

Now that you've added progress reporting, let's implement cancellation. Here's the implementation that handles cancellation, but not progress reporting:

```
public async Task RunPayroll(DateTime payrollPeriod,
    CancellationToken cancellationToken)
{
    // Step 1: Calculate hours and pay
    var payrollData = await RetieveEmployeePayrollDataFor(
        payrollPeriod);
    cancellationToken.ThrowIfCancellationRequested();

    // Step 2: Calculate tax
    var taxReporting = new Dictionary<EmployeePayrollData,
        TaxWithholding>();
    foreach (var employee in payrollData)
    {
        var taxWitholding = await RetrieveTaxData(employee);
        taxReporting.Add(employee, taxWitholding);
    }
    cancellationToken.ThrowIfCancellationRequested();
```

```
    // Step 3: generate and email paystub documents
    var paystubs = new List<Task>();
    foreach (var payrollItem in taxReporting)
    {
        var payrollTask = GeneratePayrollDocument(
            payrollItem.Key, payrollItem.Value);
        var emailTask = payrollTask.ContinueWith(
            paystub => EmailPaystub(payrollItem.Key.Email,
                paystub.Result));
        paystubs.Add(emailTask);
    }
    await Task.WhenAll(paystubs);
    cancellationToken.ThrowIfCancellationRequested();

    // Step 4: Deposit pay
    var depositTasks = new List<Task>();
    foreach (var payrollItem in taxReporting)
    {
        depositTasks.Add(MakeDeposit(payrollItem.Key,
            payrollItem.Value));
    }
    await Task.WhenAll(depositTasks);

    // Step 5: Close payroll period
    await ClosePayrollPeriod(payrollPeriod);
}
```

A caller would access this method as follows:

```
var cts = new CancellationTokenSource();
generator.RunPayroll(DateTime.Now, cts.Token);
// to cancel:
cts.Cancel();
```

The caller requests cancellation by using the `CancellationTokenSource`. Like the `TaskCompletionSource` you saw in the previous item, this class provides the intermediary between code that requests cancellation and code that supports cancellation.

Next, notice that the idiom reports cancellation by throwing a `TaskCanceledException` to indicate that the work did not complete. Cancelled tasks are faulted tasks. It follows that you should never support cancellation for async void methods (See Item 28). If you try, the cancelled task will call the unhandled exception handler.

Finally, let's combine these combinations into a common implementation:

```
public Task RunPayroll(DateTime payrollPeriod) =>
    RunPayroll(payrollPeriod, new CancellationToken(), null);

public Task RunPayroll(DateTime payrollPeriod,
    CancellationToken cancellationToken) =>
    RunPayroll(payrollPeriod, cancellationToken, null);

public Task RunPayroll(DateTime payrollPeriod,
    IProgress<(int, string)> progress) =>
    RunPayroll(payrollPeriod, new CancellationToken(), progress

public async Task RunPayroll(DateTime payrollPeriod,
    CancellationToken cancelationToken,
    IProgress<(int, string)> progress)
{
    progress?.Report((0, "Starting Payroll"));
    // Step 1: Calculate hours and pay
    var payrollData = await RetieveEmployeePayrollDataFor(
        payrollPeriod);
    cancelationToken.ThrowIfCancellationRequested();
    progress?.Report((20, "Retrieved employees and hours"));

    // Step 2: Calculate tax
    var taxReporting = new Dictionary<EmployeePayrollData,
        TaxWithholding>();
    foreach (var employee in payrollData)
    {
        var taxWitholding = await RetrieveTaxData(employee);
        taxReporting.Add(employee, taxWitholding);
    }
    cancelationToken.ThrowIfCancellationRequested();
    progress?.Report((40, "Calculated Witholding"));

    // Step 3: generate and email paystub documents
    var paystubs = new List<Task>();
    foreach (var payrollItem in taxReporting)
    {
        var payrollTask = GeneratePayrollDocument(
            payrollItem.Key, payrollItem.Value);
        var emailTask = payrollTask.ContinueWith(
            paystub => EmailPaystub(payrollItem.Key.Email,
                paystub.Result));
        paystubs.Add(emailTask);
    }
    await Task.WhenAll(paystubs);
```

```
    cancelationToken.ThrowIfCancellationRequested();
    progress?.Report((60, "Emailed Paystubs"));

    // Step 4: Deposit pay
    var depositTasks = new List<Task>();
    foreach (var payrollItem in taxReporting)
    {
        depositTasks.Add(MakeDeposit(payrollItem.Key,
            payrollItem.Value));
    }
    await Task.WhenAll(depositTasks);
    progress?.Report((80, "Deposited pay"));

    // Step 5: Close payroll period
    await ClosePayrollPeriod(payrollPeriod);
    cancelationToken.ThrowIfCancellationRequested();
    progress?.Report((100, "complete"));
}
```

Note that all the common code is factored into a single method. Progress is reported only when requested. The cancellation token is created for all overloads that don't support cancellation, but these overloads will never request cancellation.

You can see that the Task Asynchronous Programming model supports a rich vocabulary to start, cancel, and monitor asynchronous operations. These protocols enable you to design an asynchronous API that represents the capabilities of the underlying asynchronous work. Support either or both of these optional protocols when you can support them effectively. When you can't, don't implement them and mislead callers.

Item 34: Cache Generalized Async Return types

Every item that discussed the Task Asynchronous Programming model has used the `Task` or `Task<T>` type for the return type of the asynchronous code. These are the most common types you'll use for the return type on asynchronous work. But, sometimes the `Task` types introduce performance bottlenecks in your code. If you make asynchronous calls in a tight loop or in hot code paths, the Task class may be expensive to allocate and use for your asynchronous methods. The C# 7 language does not force you to use `Task` or `Task<T>` as the return types for asynchronous methods. Rather, the language demands that a

method with the async modifier must return a type that conforms to the Awaiter pattern. It must have an accessible `GetAwaiter()` method that rerturns an object that implements the `INotifyCompletion` and `ICriticalNotifyCompletion` interfaces. This accessible `GetAwaiter()` method may be by an extension method.

CThe release of the .NET Framework includes a new `ValueTask<T>` type that can be more efficient to use. This type is a value type, so it does not require an additional allocation. This reduces collection pressure. It is best suited for idioms where your asynchronous method may be retrieving cached results. Consider this method that checks weather data:

```
public async Task<IEnumerable<WeatherData>>
    RetrieveHistoricalData(DateTime start, DateTime end)
{
         var observationDate = this.startDate;
         var results = new List<WeatherData>();
         while (observationDate < this.endDate)
    {
        var observation = await RetrieveObservationData(
            observationDate);
        results.Add(observation);
        observationDate += TimeSpan.FromDays(1);
    }
    return results;
}
```

As implemented, it calls the network every time it is called. If this method is part of a widget in a phone app that displays brief status every minute, that is very inefficient. Weather information doesn't change quite that fast. You decide to cache the results for up to 5 minutes. Using Task, that implementation would look like this:

```
private List<WeatherData> recentObservations =
    new List<WeatherData>();
private DateTime lastReading;
public async Task<IEnumerable<WeatherData>>
    RetrieveHistoricalData()
{
    if (DateTime.Now - lastReading > TimeSpan.FromMinutes(5))
    {
        recentObservations = new List<WeatherData>();
        var observationDate = this.startDate;
```

```
        while (observationDate < this.endDate)
        {
            var observation = await RetrieveObservationData(
                observationDate);
            recentObservations.Add(observation);
            observationDate += TimeSpan.FromDays(1);
        }
        lastReading = DateTime.Now;
    }
    return recentObservations;
}
```

In many cases, that change is probably sufficient. The network latency is the most impactful bottleneck in this code. However, suppose this widget runs in a very memory constrained environment. Now, you want to avoid the object allocations each time the method is called. That's when you would switch to using the `ValueTask` type. Here's how that implementation would look:

```
public ValueTask<IEnumerable<WeatherData>>
    RetrieveHistoricalData()
{
    if (DateTime.Now - lastReading > TimeSpan.FromMinutes(5))
    {
        return new ValueTask<IEnumerable<WeatherData>>
            (recentObservations);
    }
    else
    {
        async Task<IEnumerable<WeatherData>> loadCache()
        {
            recentObservations = new List<WeatherData>();
            var observationDate = this.startDate;
            while (observationDate < this.endDate)
            {
                var observation = await
                    RetrieveObservationData(observationDate);
                recentObservations.Add(observation);
                observationDate += TimeSpan.FromDays(1);
            }
            lastReading = DateTime.Now;
            return recentObservations;
        }
        return new ValueTask<IEnumerable<WeatherData>>
            (loadCache());
    }
}
```

There are important idioms in this method that you should use when you incorporate `ValueTask`. First, the method is not an `async` method, but rather returns a `ValueTask`. The nested function that performs the async work has the `async` modifier. That means your program doesn't do the extra state machine management and allocation if the cache is valid. Second, notice that `ValueTask` has a constructor that takes a `Task` as its argument. It will do the await internally.

The `ValueTask` type enables optimizations that you can make when your performance measurements indicate that memory allocations for Task objects are creating bottlenecks in your code. You'll still likely use the Task types for most of your asynchronous methods. In fact, I'd recommend using Task and `Task<T>` for all asynchronous methods until you've measured and found memory allocations to be a bottleneck. The conversion to a value type is not difficult, and can be made when you discover that will fix your performance issues.

4. Parallel Processing

Writing parallel algorithms is not the same as writing asynchronous algorithms. The challenges are different for working with CPU bound parallel code. The tools are different too. While the Task Asynchronous Programming model can be used with parallel CPU algorithms, there are often better choices.

In this chapter, I'll cover many of the ways you can use different libraries and tools to make parallel programming easier. It's still not easy, but using better tools correctly makes it easier than it was.

Item 35: Learn How PLINQ Implements Parallel Algorithms

This is the item where I wish I could say that parallel programming is now as simple as adding `AsParallel()` to all your loops. It's not, but PLINQ does make it much easier than it was to leverage multiple cores in your programs and still have programs that are correct. It's by no means trivial to create programs that make use of multiple cores, but PLINQ makes it easier.

You still have to understand when data access must be synchronized. You still need to measure the effects of parallel and sequential versions of the methods declared in `ParallelEnumerable`. Some of the methods involved in LINQ queries can execute in parallel very easily. Others force more sequential access to the sequence of elements—or, at least, require the complete sequence (like OrderBy). Let's walk through a few samples using PLINQ and learn what works well, and where some of the pitfalls still exist. All the samples and discussions for this item use LINQ to Objects. The title even calls out "Enumerable," not "Queryable". PLINQ really won't help you parallelize LINQ to SQL, or Entity Framework algorithms. That's not really a limiting feature, because those implementations leverage the parallel database engines to execute queries in parallel.

Here's a simple query using method call syntax that calculates n! for values

less than 150 from a data source consisting of a large set of integers:

```
var nums = data.Where(m => m < 150).
    Select(n => Factorial(n));
```

You can make this a parallel query by simply adding AsParallel() as the first method on the query:

```
var numsParallel = data.AsParallel().
    Where(m => m < 150).Select(n => Factorial(n));
```

Of course, you can do the same kind of work with query syntax.

```
var nums = from n in data
           where n < 150
           select Factorial(n);
```

The Parallel version relies on putting AsParallel() on the data sequence:

```
var numsParallel = from n in data.AsParallel()
                   where n < 150
                   select Factorial(n);
```

The results are the same as with the method call version.

This first sample is very simple yet it does illustrate a few important concepts used throughout PLINQ. AsParallel() is the method you call to opt in to parallel execution of any query expression. Once you call `AsParallel()`, subsequent operations will occur on multiple cores using multiple threads. `AsParallel()` returns an `IParallelEnumerable()` rather than an `IEnumerable()`. PLINQ is implemented as a set of extension methods on `IParallelEnumerable`. They have almost exactly the same signatures as the methods found in the `Enumerable` class that extends `IEnumerable`. Simply substitute `IParallelEnumerable` for `IEnumerable` in both parameters and return values. The advantage of this choice is that PLINQ follows the same patterns that all LINQ providers follow. That makes PLINQ very easy to learn. Everything you know about LINQ, in general, will apply to PLINQ.

Of course, it's not quite that simple. This initial query is very easy to use with PLINQ. It does not have any shared data. The order of the results doesn't matter. That's why it is possible to get a speedup that's in direct proportion to

the number of cores in the machine upon which this code is running. To help you get the best performance out of PLINQ, there are several methods that control how the parallel task library functions are accessible using `IParallelEnumerable`.

Every parallel query begins with a partitioning step. PLINQ needs to partition the input elements and distribute those over the number of tasks created to perform the query. Partitioning is one of the most important aspects of PLINQ, so it is important to understand the different approaches, how PLINQ decides which to use, and how each one works. First, partitioning can't take much time. That would cause the PLINQ library to spend too much time partitioning, and too little time actually processing your data. PLINQ uses four different partitioning algorithms, based on the input source and the type of query you are creating:

• Range Partitioning

• Chunk Partitioning

• Striped Partitioning

• Hash Partitioning

The simplest algorithm is range partitioning. Range partitioning divides the input sequence by the number of tasks and gives each task one set of items. For example, an input sequence with 1,000 items running on a quad core machine would create four ranges of 250 items each. Range partitioning is used only when the query source supports indexing the sequence and reports how many items are in the sequence. That means range partitioning is limited to query sources that are like `List<T>`, arrays, and other sequences that support the `IList<T>` interface. Range partitioning is usually used when the source of the query supports those operations.

The second choice for partitioning is chunk partitioning. This algorithm gives each task a "chunk" of input items anytime it requests more work. The internals of the chunking algorithm will continue to change over time, so I won't cover the current implementation in depth. You can expect that the size of chunks will start small, because an input sequence may be small. That prevents the

situation where one task must process an entire small sequence. You can also expect that as work continues, chunks may grow in size. That minimizes the threading overhead and helps to maximize throughput. Chunks may also change in size depending on the time cost for delegates in the query and the number of elements rejected by `where` clauses. The goal is to have all tasks finish at close to the same time to maximize the overall throughput.

The other two partitioning schemes optimize for certain query operations. First is a striped partition. A striped partition is a special case of range partitioning that optimizes processing the beginning elements of a sequence. Each of the worker threads processes items by skipping N items and then processing the next M. After processing M items, the worker thread will skip the next N items again. The stripe algorithm is easiest to understand if you imagine a stripe of 1 item. In the case of four worker tasks, one task gets the items at indices 0, 4, 8, 12, and so on. The second task gets items at indices 1, 5, 9, 13, and so on. Striped partitions avoid any interthread synchronization to implement `TakeWhile()` and `SkipWhile()` for the entire query. Also, it lets each worker thread move to the next items it should process using simple arithmetic.

The final algorithm is a Hash Partitioning. Hash Partitioning is a special-purpose algorithm designed for queries with the `Join`, `GroupJoin`, `GroupBy`, `Distinct`, `Except`, `Union`, and `Intersect` operations. Those are more expensive operations, and a specific partitioning algorithm can enable greater parallelism on those queries. Hash Partitioning ensures that all items generating the same hash code are processed by the same task. That minimizes the intertask communications for those operations.

Independent of the partitioning algorithm, there are three different algorithms used by PLINQ to parallelize tasks in your code: Pipelining, Stop & Go, and Inverted Enumeration. Pipelining is the default, so I'll explain that one first. In pipelining, one thread handles the enumeration (the `foreach`, or query sequence). Multiple threads are used to process the query on each of the elements in the sequence. As each new item in the sequence is requested, it will be processed by a different thread. The number of threads used by PLINQ in pipelining mode will usually be the number of cores (for most CPU bound queries). In my factorial example, it would work with two threads on my dual core machine. The first item would be retrieved from the sequence and

processed by one thread. Immediately the second item would be requested and processed by a second thread. Then, when one of those items finished, the third item would be requested, and the query expression would be processed by that thread. Throughout the execution of the query for the entire sequence, both threads would be busy with query items. On a machine with more cores, more items would be processed in parallel.

For example, on a 16 core machine, the first 16 items would be processed immediately by 16 different threads (presumably running on 16 different cores). I've simplified a little. There is a thread that handles the enumeration, and that often means Pipelining creates (Number of Cores + 1) threads. In most scenarios, the enumeration thread is waiting most of the time, so it makes sense to create one extra.

Stop and Go means that the thread starting the enumeration will join on all the threads running the query expression. That method is used when you request that immediate execution of a query by using `ToList()` or `ToArray()`, or anytime PLINQ needs the full result set before continuing such as ordering and sorting. Both of the following queries use Stop and Go:

```
var stopAndGoArray = (from n in data.AsParallel()
                      where n < 150
                      select Factorial(n)).ToArray();

var stopAndGoList = (from n in data.AsParallel()
                     where n < 150
                     select Factorial(n)).ToList();
```

Using Stop and Go processing you'll often get slightly better performance at a cost of a higher memory footprint. However, notice that I've still constructed the entire query before executing any of the query expressions. You'll still want to compose the entire query, rather than processing each portion using Stop and Go and then composing the final results using another query. That will often cause the threading overhead to overwhelm performance gains. Processing the entire query expression as one composed operation is almost always preferable.

The final algorithm used by the parallel task library is Inverted Enumeration. Inverted Enumeration doesn't produce a result. Instead, it performs some

action on the result of every query expression. In my earlier samples, I printed the results of the Factorial computation to the console:

```
var numsParallel = from n in data.AsParallel()
                   where n < 150
                   select Factorial(n);
foreach (var item in numsParallel)
    Console.WriteLine(item);
```

LINQ to Objects (nonparallel) queries are evaluated lazily. That means each value is produced only when it is requested. You can opt into the parallel execution model (which is a bit different) while processing the result of the query. That's how you ask for the Inverted Enumeration model:

```
var nums2 = from n in data.AsParallel()
            where n < 150
            select Factorial(n);
nums2.ForAll(item => Console.WriteLine(item));
```

Inverted enumeration uses less memory than the Stop and Go method. Also, it enables parallel actions on your results. Notice that you still need to use `AsParallel()` in your query in order to use `ForAll()`. `ForAll()` has a lower memory footprint than the Stop and Go model. In some situations, depending on the amount of work being done by the action on the result of the query expression, inverted enumeration may often be the fastest enumeration method.

All LINQ queries are executed lazily. You create queries, and those queries are only executed when you ask for the items produced by the query. LINQ to Objects goes a step further. LINQ to Objects executes the query on each item as you ask for that item. PLINQ works differently. Its model is closer to LINQ to SQL, or the Entity Framework. In those models, when you ask for the first item, the entire result sequence is generated. PLINQ is closer to that model, but it's not exactly right. If you misunderstand how PLINQ executes queries, then you'll use more resources than necessary, and you can actually make parallel queries run more slowly than LINQ to Objects queries on multicore machines.

To demonstrate some of the differences, I'll walk through a reasonably simple query. I'll show you how adding `AsParallel()` changes the execution model. Both models are valid. The rules for LINQ focus on what the results are, not how they are generated. You'll see that both models will generate the exact

same results. Differences in how would only manifest themselves if your algorithm has side effects in the query clauses.

Here's the query I used to demonstrate the differences:

```
var answers = from n in Enumerable.Range(0, 300)
                where n.SomeTest()
                select n.SomeProjection();
```

I instrumented the `SomeTest()` and `SomeProjection()` methods to show when each gets called:

```
public static bool SomeTest(this int inputValue)
{
    Console.WriteLine($"testing element: {inputValue}");
    return inputValue % 10 == 0;
}

public static string SomeProjection(this int input)
{
    Console.WriteLine($"projecting an element: {input}");
    return $"Delivered {input} at {DateTime.Now:T}";
}
```

Finally, instead of a simple `foreach` loop, I iterated the results using the `IEnumerator<string>` members so that you can see when different actions take place. This is so that I can more clearly show exactly how the sequence is generated (in parallel) and enumerated (in this enumeration loop). In production code, I prefer a different implementation.

```
var iter = answers.GetEnumerator();

Console.WriteLine("About to start iterating");
while (iter.MoveNext())
{
    Console.WriteLine("called MoveNext");
    Console.WriteLine(iter.Current);
}
```

Using the standard LINQ to Objects implementation, you'll see output that looks like this:

```
About to start iterating
```

```
testing element: 0
projecting an element: 0
called MoveNext
Delivered 0 at 1:46:08 PM
testing element: 1
testing element: 2
testing element: 3
testing element: 4
testing element: 5
testing element: 6
testing element: 7
testing element: 8
testing element: 9
testing element: 10
projecting an element: 10
called MoveNext
Delivered 10 at 1:46:08 PM
testing element: 11
testing element: 12
testing element: 13
testing element: 14
testing element: 15
testing element: 16
testing element: 17
testing element: 18
testing element: 19
testing element: 20
projecting an element: 20
called MoveNext
Delivered 20 at 1:46:08 PM
testing element: 21
testing element: 22
testing element: 23
testing element: 24
testing element: 25
testing element: 26
testing element: 27
testing element: 28
testing element: 29
testing element: 30
projecting an element: 30
```

The query does not begin to execute until the first call to `MoveNext()` on the enumerator. The first call to `MoveNext()` executes the query on enough elements to retrieve the first element on the result sequence (which happens to be one element for this query). The next call to `MoveNext()` processes

elements in the input sequence until the next item in the output sequence has been produced. Using LINQ to Objects, each call to `MoveNext()` executes the query on as many elements are necessary to produce the next output element.

The rules change once you change the query to be a parallel query:

```
var answers = from n in ParallelEnumerable.Range(0, 300)
                where n.SomeTest()
                select n.SomeProjection();
```

The output from this query will look very different. Here's a sample from one run (it will change somewhat for each run):

```
About to start iterating
testing element: 150
projecting an element: 150
testing element: 0
testing element: 151
projecting an element: 0
testing element: 1
testing element: 2
testing element: 3
testing element: 4
testing element: 5
testing element: 6
testing element: 7
testing element: 8
testing element: 9
testing element: 10
projecting an element: 10
testing element: 11
testing element: 12
testing element: 13
testing element: 14
testing element: 15
testing element: 16
testing element: 17
testing element: 18
testing element: 19
testing element: 152
testing element: 153
testing element: 154
testing element: 155
testing element: 156
testing element: 157
```

```
testing element: 20
... Lots more here elided ...
testing element: 286
testing element: 287
testing element: 288
testing element: 289
testing element: 290
Delivered 130 at 1:50:39 PM
called MoveNext
Delivered 140 at 1:50:39 PM
projecting an element: 290
testing element: 291
testing element: 292
testing element: 293
testing element: 294
testing element: 295
testing element: 296
testing element: 297
testing element: 298
testing element: 299
called MoveNext
Delivered 150 at 1:50:39 PM
called MoveNext
Delivered 160 at 1:50:39 PM
called MoveNext
Delivered 170 at 1:50:39 PM
called MoveNext
Delivered 180 at 1:50:39 PM
called MoveNext
Delivered 190 at 1:50:39 PM
called MoveNext
Delivered 200 at 1:50:39 PM
called MoveNext
Delivered 210 at 1:50:39 PM
called MoveNext
Delivered 220 at 1:50:39 PM
called MoveNext
Delivered 230 at 1:50:39 PM
called MoveNext
Delivered 240 at 1:50:39 PM
called MoveNext
Delivered 250 at 1:50:39 PM
called MoveNext
Delivered 260 at 1:50:39 PM
called MoveNext
Delivered 270 at 1:50:39 PM
called MoveNext
Delivered 280 at 1:50:39 PM
```

```
called MoveNext
Delivered 290 at 1:50:39 PM
```

Notice how much it changed. The very first call to `MoveNext()` causes PLINQ to start all the threads involved in generating the results. That causes quite a few (in this case, almost all) result objects to be produced. Each subsequent call to `MoveNext()` will grab the next item from those already produced. You can't predict when a particular input element will be processed. All you know is that the query will begin executing (on several threads) as soon as you ask for the first element of the query.

PLINQ's methods that support query syntax understand how this behavior can affect performance on queries. Suppose you modify the query to select the second page of results using `Skip()` and `Take()`:

```
var answers = (from n in ParallelEnumerable.Range(0, 300)
               where n.SomeTest()
               select n.SomeProjection()).
               Skip(20).Take(20);
```

Executing this query produces output that is identical to that produced by LINQ to Objects. That's because PLINQ knows that it will be faster to produce only 20 elements rather than 300. (I'm simplifying, but PLINQ's implementation of `Skip()` and `Take()` do tend to favor a sequential algorithm more than other algorithms.)

You can modify the query a bit more, and get PLINQ to generate all the elements using the parallel execution model. Just add an `orderby` clause:

```
var answers = (from n in ParallelEnumerable.Range(0, 300)
               where n.SomeTest()
               orderby n.ToString().Length
               select n.SomeProjection()).
               Skip(20).Take(20);
```

The lambda argument for `orderby` must not be something that the compiler can optimize away (that's why I used `n.ToString().Length` rather than just n above, in case there are optimizations around `Enumerable.Range` being ordered.). Now, the query engine must generate all the elements of the output sequence before it can order them properly. Only once the elements are

ordered properly can the `Skip()` and `Take()` methods know which elements should be returned. Of course, it's faster on multicore machines to use multiple threads to generate all the output than it would be to generate the sequence sequentially. PLINQ knows that, too, so it starts multiple threads to create the output.

PLINQ tries to create the best implementation for the queries you write in order to generate the results you need with the least amount of work, and in the least amount of time. Sometimes that means PLINQ queries will execute in a different manner than you would expect. Sometimes it will act more like LINQ to Objects, where asking for the next item in the output sequence executes the code that produces it. Sometimes it will behave more like LINQ to SQL or Entity Framework in that asking for the first item will produce all of them. Sometimes it will behave like a mixture of the two. You should make sure that you don't introduce any side effects in your LINQ queries. Side effects in LINQ queries are wrong in sequential execution queries, and even more so in a PLINQ execution model. You should construct your queries with some care to ensure that you get the most out of the underlying technology. That requires you to understand how they work differently.

Parallel algorithms are limited by Amdahl's law: The speedup of a program using multiple processors is limited by the sequential fraction of the program. The extension methods in `ParallelEnumerable` are no exception to this rule. Many of these methods can operate in parallel, but some of them will affect the degree of parallelism due to their nature. Obviously `OrderBy` and `ThenBy` require some coordination between tasks. `Skip`, `SkipWhile`, `Take`, and `TakeWhile` will affect the degree of parallelism. Parallel tasks running on different cores may finish in different orders. You can use the `AsOrdered()` and `AsUnordered()` methods to instruct PLINQ as to whether or not order matters in the result sequence.

Sometimes your own algorithm relies on side effects and cannot be parallelized. You can force sequential execution using the `ParallelEnumerable.AsSequential()` extension method to interpret a parallel sequence as an IEnumerable and force sequential execution.

Finally, `ParallelEnumerable` contains methods that allow you to control how PLINQ executes parallel queries. You can use `WithExecutionMode()` to

suggest parallel execution, even if that means selecting a high overhead algorithm. By default, PLINQ will parallelize those constructs where it expects parallelism to help. You can use `WithDegreeOfParallelism()` to suggest the number of threads that may be used in your algorithm. Usually, PLINQ will allocate threads based on the number of processors on the current machine. You can also use the `WithMergeOptions()` to request a change in how PLINQ controls buffering results during a query. Usually, PLINQ will buffer some results from each thread before making them available to the consumer thread. You can request no buffering to make results available immediately. You can request full buffering, which will increase performance at a cost of higher latency. Auto Buffering, the default, provides a balance between latency and performance. Buffering is a hint, not a demand. PLINQ may ignore your request.

I'm not providing any specific guidance on which of these settings is best for you because they will be highly dependent on your algorithm. However, you have those settings that you can change, and you should experiment on a variety of target machines to see if these will help your algorithms. If you don't have several different target machines to experiment with, I'd recommend using the defaults.

PLINQ makes parallel computing much easier than it previously was. It's an important time for these additions; parallel computing will continue to become more important as more and more cores become commonplace for desktop and laptop computers. It's still not easy. And poorly designed algorithms may not see performance improvements from parallelization. Your task is to look for loops and other tasks that can be parallelized. Take those algorithms and try the parallel versions. Measure the results. Work on the algorithms to get better results on the performance. Realize that some algorithms aren't easily parallelizable, and keep those serial.

Item 36: Construct Parallel Algorithms with Exceptions in Mind

The previous two items blissfully ignored the possibility of anything going wrong with any of the child threads doing its work. That's clearly not how the

real world works. Exceptions will occur in your child threads, and you'll be left to pick up the pieces somehow. Of course, exceptions in background threads increase the complexity in several ways. Exceptions can't just continue up the call stack across thread boundaries. Instead, if an Exception reaches the thread start method, that thread gets terminated. There's no way for the calling thread to retrieve the error, or do anything about it. Furthermore, if your parallel algorithm must support rollback if there are problems, you'll have to do more work to understand any side effects that have occurred and what you should do to recover from those errors. Every algorithm has different requirements, so there are no universal answers for handling exceptions in a parallel environment. The guidelines I provide here are just that: guidelines that you can use to determine the best strategy for your particular application.

Let's begin with the async download method from the Item 25. That has a very simple strategy in that there are no side effects, and the downloads from all other Web hosts can continue without concern for the one download that is failing. Parallel operations use the `AggregateException` type to handle exceptions in parallel operations. The `AggregateException` is a container that holds all exceptions generated from any of the parallel operations in an `InnerExceptions` property. There are a couple different ways to handle the exception in this process. First, I'll show the more general case, how to handle any errors generated by subtasks in the outer processing.

The `RunAsync()` method shown in the Item 25 uses more than one parallel operation. That means you may have `AggregateExceptions` in the `InnerExceptions` collection that is part of the `AggregateException` you actually catch. The more parallel operations you have, the deeper this nesting can go. Because of the way parallel operations compose with each other, you may end up with multiple copies of the original exception in the final collection of exceptions. I modified the call to `RunAsync()` to process possible errors:

```
try
{
    urls.RunAsync(
        url => startDownload(url),
        task => finishDownload(
            task.AsyncState.ToString(), task.Result));
}
```

```
catch (AggregateException problems)
{
    ReportAggregateError(problems);
}

private static void ReportAggregateError(
    AggregateException aggregate)
{
    foreach (var exception in aggregate.InnerExceptions)
        if (exception is AggregateException agEx)
            ReportAggregateError(agEx);
        else
            Console.WriteLine(exception.Message);
}
```

The `ReportAggregateError` prints out the messages for all exceptions that are not themselves `AggregateExceptions`. Of course, this has the nasty side effect of swallowing all exceptions, whether you anticipated them or not. That's rather dangerous. Instead, what you want to do is handle only those exceptions from which you can recover, and rethrow any other exceptions.

There are enough recursive collections here that a utility method makes sense. The generic method must know which exception types you want to handle, and which are not expected and how you'll handle the ones you are handling. You need to send this method a set of exception types, and the code to handle the exception. That's simply a dictionary of types and `Action<T>` lambda expressions. And, if the handler doesn't process everything in the collection of `InnerExceptions`, clearly something else went wrong. That means it's time to rethrow the original exception. Here's the updated code that calls `RunAsync`:

```
try
{
    urls.RunAsync(
        url => startDownload(url),
        task => finishDownload(task.AsyncState.ToString(),
        task.Result));
}
catch (AggregateException problems)
{
    var handlers = new Dictionary<Type, Action<Exception>>();
    handlers.Add(typeof(WebException),
        ex => Console.WriteLine(ex.Message));

    if (!HandleAggregateError(problems, handlers))
```

```
        throw;
}
```

The `HandleAggregateError` method recursively looks at every exception. If the exception is expected, the handler is called. Otherwise, `HandleAggregateError` returns false, indicating that this set of aggregate exceptions cannot be processed correctly.

```
private static bool HandleAggregateError(
    AggregateException aggregate,
    Dictionary<Type, Action<Exception>> exceptionHandlers)
{
    foreach (var exception in aggregate.InnerExceptions)
    {
        if (exception is AggregateException agEx)
        {
            if (!HandleAggregateError(agEx, exceptionHandlers))
            {
                return false;
            } else
            {
                continue;
            }
        }
        else if (exceptionHandlers.ContainsKey(
            exception.GetType()))
        {
            exceptionHandlers[exception.GetType()](exception);
        }
        else
            return false;
    }
    return true;
}
```

When this code encounters an `AggregateException`, it evaluates that child list recursively. When it encounters any other kind of exception, it looks into the dictionary. If a handler Action<> has been registered, it calls that handler. If not, then it returns false immediately, having found an exception that it should not be handled.

You may be wondering why the original `AggregateException` gets thrown rather than the single exception for which there was no handler. The problem is that throwing one exception out of the collection could lose important

information. The `InnerExceptions` may contain any number of exceptions. More than one may be of a type that is not expected. You must return the entire collection or risk losing much of that information. In many cases, there will be only one exception in the `AggregateException`'s `InnerExceptions` collection. However, you should not code that way because when you do need that extra information, it won't be there.

Of course, this feels a bit ugly. Wouldn't it be better to prevent the exception from leaving the task doing the background work? In almost all cases, that is better. That requires changing the code that runs the background task to ensure that no exceptions can exit the background task. Whenever you use the `TaskCompletionSource<>` class, that means never calling `TrySetException()`, but rather ensuring that every task somehow calls `TrySetResult()` to indicate completeness. That would mean the following changes to `startDownload`. But, just like I said earlier, you should not be catching every single exception. You should catch only those exceptions from which you can recover. In this example, you can reasonably recover from a `WebException`, indicating that the remote host isn't available. Other exception types would indicate more serious problems. Those should continue to generate exceptions and stop all processing. That causes the following changes to the `startDownload` method:

```
private static Task<byte[]> startDownload(string url)
{
    var tcs = new TaskCompletionSource<byte[]>(url);
    var wc = new WebClient();
    wc.DownloadDataCompleted += (sender, e) =>
    {
        if (e.UserState == tcs)
        {
            if (e.Cancelled)
                tcs.TrySetCanceled();
            else if (e.Error != null)
            {
                if (e.Error is WebException)
                    tcs.TrySetResult(new byte[0]);
                else
                    tcs.TrySetException(e.Error);
            }
            else
                tcs.TrySetResult(e.Result);
        }
```

```
    };
    wc.DownloadDataAsync(new Uri(url), tcs);
    return tcs.Task;
}
```

A `WebException` causes a return indicating 0 bytes read, and all other exceptions are thrown through the normal channels. Yes, this does mean that you still need to handle what happens when `AggregateExceptions` are thrown. It's possible that you merely need to treat those as fatal errors, and your background tasks can handle all other errors. But you do need to understand that it's a different kind of exception.

Of course, errors in background tasks create other issues when you use LINQ syntax. Remember from Item 33 that I described three different parallel algorithms. In all cases, using PLINQ makes some changes to the normal lazy evaluation, and these changes are important for how you must handle exceptions in your PLINQ algorithms. Remember that usually, a query executes only as other code requests the items produced by the query. That isn't quite how it works with PLINQ. Background threads generate results as they run, and another task constructs the final result sequence. It's not exactly an eager evaluation. The query results are not produced immediately. However, the background threads that produce the results will start as soon as the scheduler allows. Not immediately, but very soon. Processing any of those items may throw an exception. Now, that means you must change your exception handling code. In a typical LINQ query, you can put `try`/`catch` blocks around the code that uses the query results. It's not needed around the code that defines the LINQ query expression:

```
var nums = from n in data
           where n < 150
           select Factorial(n);

try
{
    foreach (var item in nums)
        Console.WriteLine(item);
}
catch (InvalidOperationException inv)
{
    // elided
}
```

Once PLINQ is involved, you must enclose the definition of the query in the `try/catch` block as well. And, of course, remember that once you use PLINQ, you must catch `AggregateException` instead of whatever exception you had been originally expecting. This is true whether you use Pipelining, Stop and Go, or Inverted Enumeration.

Exceptions are complicated in any algorithm. Parallel tasks create more complications. The Parallel Task Library uses the `AggregateException` class to hold any and all exceptions thrown somewhere in the depths of your parallel algorithms. Once any of the background threads throws an exception, any other background operations are also stopped. Your best plan is to try to ensure that no exceptions can be thrown from the code executing in your parallel tasks. Even so, other exceptions that you don't expect may be thrown elsewhere. That means you must handle any `AggregateException` in the controlling thread that initiated all the background work.

Item 37: Use the Thread Pool Instead of Creating Threads

You can't know the optimum number of threads that should be created for your application. Your application will run on a machine with multiple cores now, but it's almost certain that whatever number of cores you assume today will be wrong six months from now. Furthermore, you can't control for the number of threads that the CLR will create for its own tasks, such as the garbage collector. On a server application, such as ASP.NET or REST services, each new request is handled by a different thread. That makes it very hard for you, as an application or class library developer, to optimize for the proper number of threads on the target system. However, the .NET **thread pool** has all the knowledge necessary to optimize the number of active threads on the target system. Furthermore, if you have created too many tasks and threads for the target machine, the thread pool queues up additional requests until a new background thread is available. Even better, the Task based library utilizes the thread pool to run tasks that you start using `Task.Run`.

The .NET thread pool performs much of the work to handle thread resource management for you. It manages those resources in such a way that you get

better performance when your application starts background tasks repeatedly and doesn't interact with those tasks very closely.

You should not create threads in your application code. You should use libraries that manage threads and the thread pool for you, like the Task Parallel Library.

I don't cover the thread pool implementation in exhaustive detail, because the purpose of using the thread pool is to off-load much of that work and make it the framework's problem. In short, the number of threads in the thread pool grows to provide you the best mix of available threads and the minimum amount of allocated and unused resources. You queue up a worker item, and when a thread is available, it executes your thread procedure. The thread pool's job is to make sure that a thread becomes available quickly. Essentially, you fire the request and forget it.

The thread pool also manages the end-of-task cycle automatically. When a task finishes, the thread is not destroyed; instead, it is returned to a ready state so that it is available for another task. The thread is again available for other work, as needed by the thread pool. This next task need not be the same task; the thread can execute any other long-running method your application has in mind. You simply call `Task.Run` with another target method, and your thread pool will manage the work for that method as well.

The system manages the number of tasks that are active in a thread pool. The thread pool starts tasks based on the amount of system resources available. If the system is currently operating at close to capacity, the thread pool waits to start new tasks. However, if the system is lightly loaded, the thread pool launches additional tasks immediately. You don't need to write your own load-balancing logic. The thread pool manages that for you.

You might think that the optimal number of tasks would be equal to the number of cores in the target machine. That's not the worst strategy to take, but it's simplistic in its analysis, and it's almost certainly not the best answer. Wait time, contention for resources other than the CPU, and other processes outside your control all have an effect on the optimal number of threads for your application. If you create too few threads, you'll end up not getting the best performance for your application as cores sit idle. Having way too many

threads means that your target machine will spend too much time scheduling threads and too little time executing the work performed by them.

To give you some general guidance, I wrote a small application that uses the Hero of Alexandria algorithm to calculate square roots. It's general guidance, because each algorithm has unique characteristics. In this case, the core algorithm is simple and does not communicate with other threads to perform its work.

You start by making a guess at the square root of a number. A simple starting guess is 1. To find the next approximation, you find the average of (1) the current guess and (2) the original number divided by the current guess. For example, to find the square root of 10, you'd make a guess of 1. The next guess is (1 + (10/1)) / 2, or 5.5. You continue to repeat the steps until the guess converges at the answer. Here's the code:

```
public static class Hero
{
    public static double FindRoot(double number)
    {
        double previousError = double.MaxValue;
        double guess = 1;
        double error = Math.Abs(guess * guess - number);

        while (previousError / error > 1.000001)
        {
            guess = (number / guess + guess) / 2.0;
            previousError = error;
            error = Math.Abs(guess * guess - number);
        }
        return guess;
    }
}
```

To examine the performance characteristics of the thread pool against manually created threads and against a single-threaded version of the application, I wrote test harnesses that perform repeated calculations against this algorithm:

```
private static double OneThread()
{
    Stopwatch start = new Stopwatch();
    double answer;
    start.Start();
```

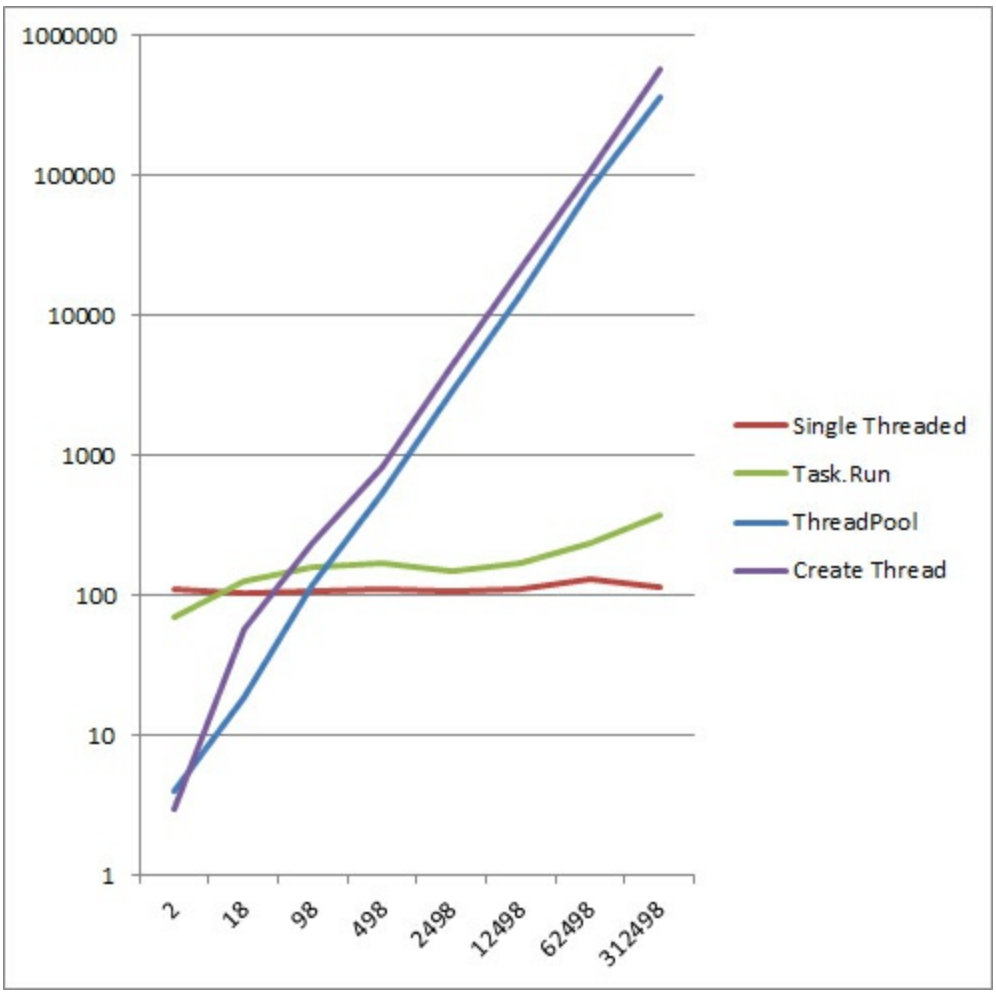
```
    for (int i = LowerBound; i < UpperBound; i++)
        answer = Hero.FindRoot(i);
    start.Stop();
    return start.ElapsedMilliseconds;
}

private static async Task<double> TaskLibrary(int numTasks)
{
    var itemsPerTask = (UpperBound - LowerBound) / numTasks + 1
    double answer;
    List<Task> tasks = new List<Task>(numTasks);
    Stopwatch start = new Stopwatch();
    start.Start();
    for(int i = LowerBound; i < UpperBound; i+= itemsPerTask)
    {
        tasks.Add(Task.Run(() =>
        {
            for (int j = i; j < i + itemsPerTask; j++)
                answer = Hero.FindRoot(j);
        }));
    }
    await Task.WhenAll(tasks);
    start.Stop();
    return start.ElapsedMilliseconds;
}

private static double ThreadPoolThreads(int numThreads)
{
    Stopwatch start = new Stopwatch();
    using (AutoResetEvent e = new AutoResetEvent(false))
    {
        int workerThreads = numThreads;
        double answer;
        start.Start();
        for (int thread = 0; thread < numThreads; thread++)
            System.Threading.ThreadPool.QueueUserWorkItem(
                (x) =>
                {
                    for (int i = LowerBound;
                        i < UpperBound; i++)
                        if (i % numThreads == thread)
                            answer = Hero.FindRoot(i);
                    if (Interlocked.Decrement(
                        ref workerThreads) == 0)
                        e.Set();
                });
        e.WaitOne();
        start.Stop();
```

```
        return start.ElapsedMilliseconds;
    }
}

private static double ManualThreads(int numThreads)
{
    Stopwatch start = new Stopwatch();
    using (AutoResetEvent e = new AutoResetEvent(false))
    {
        int workerThreads = numThreads;
        double answer;
        start.Start();
        for (int thread = 0; thread < numThreads; thread++)
        {
            System.Threading.Thread t = new Thread(
                () =>
                {
                    for (int i = LowerBound;
                        i < UpperBound; i++)
                        if (i % numThreads == thread)
                            answer = Hero.FindRoot(i);
                    if (Interlocked.Decrement(
                         ref workerThreads) == 0)
                        e.Set();
                });
            t.Start();
        }
        e.WaitOne();
        start.Stop();
        return start.ElapsedMilliseconds;
    }
}
```

This main program produces timing for the single-threaded version and both multithreaded versions so that you can see the effect of adding threads using both algorithms. Figure 4.1 shows the resulting graph. There are a few things to learn from this example. First, the manually created threads have much more overhead compared with the thread pool threads or the Task based implementation. If you create more than ten threads, threading overhead becomes the main performance bottleneck. Even with this algorithm, in which there isn't much wait time, that's not good. The Task based library has a constant overhead: it is slower for a small number of threads, but as the number of requested tasks increases, the task based API manages the number of threads better than the other algorithms.

Using the thread pool, you must queue more than 40 items before the overhead dominates the work time. And that's on a dual-core laptop. Server-class machines with more cores would be efficient with more threads. Having more threads than cores is often the smart choice. However, that choice is highly dependent on the application and on the amount of time the application's threads spend waiting for resources.

Figure 4.1 The effects of calculation time for the single-threaded and multi-threaded versions using `System.Threading.Thread` versus `System.Threading.ThreadPool.QueueUserWorkItem`. The Y axis shows the time (in milliseconds) per 100,000 calculations on a quad-core laptop.



Two important factors result in the higher performance of the thread pool compared with creating your own threads manually. First, the thread pool reuses threads as they become available for work. When you manually create

new threads, you must create a new thread for each new task. The creation and destruction of those threads take more time than the .NET thread pool management.

Second, the thread pool manages the active number of threads for you. If you create too many threads, the system queues them up, and they wait to execute until enough resources are available. `QueueUserWorkItem` hands work to the next available thread in the thread pool and does some thread resource management for you. If all the threads in the application's thread pool are busy, it queues tasks to wait for the next available thread.

The farther you move down the road into a world with increasing numbers of cores, the more likely it is that you'll be creating multithreaded applications. If you're creating server-side applications in .NET with WCF, ASP.NET, or .NET remoting, you're already creating multithreaded applications. Those .NET subsystems use the thread pool to manage thread resources, and you should, too. You'll find that the thread pool introduces less overhead, and that leads to better performance. Also, the .NET thread pool does a better job of managing the number of active threads that should be performing work than you can manage at the application level.

Item 38: Use BackgroundWorker for Cross-Thread Communication

Item 11 shows a sample that started various numbers of background tasks using `ThreadPool.QueueUserWorkItem`. Using this API method is simple, because you have off-loaded most of the thread management issues to the framework and the underlying operating system (OS). There's a lot of functionality you can simply reuse, so `QueueUserWorkItem` should be your default tool of choice when you need to create background threads that execute tasks in your application. `QueueUserWorkItem` makes several assumptions about how you should be performing your work. When your design doesn't match those assumptions, you'll have more work to do. Instead of creating your own threads using `System.Threading.Thread`, you should use `System.ComponentModel.BackgroundWorker`. The `BackgroundWorker` class is built on top of `ThreadPool` and adds many features for interthread

communication.

The single most important issue you must deal with is exceptions in your `WaitCallback`, the method that does the work in the background thread. If any exceptions are thrown from that method, the system will terminate your application. It doesn't simply terminate that one background thread; it terminates the entire application. This behavior is consistent with other background thread API methods, but the difference is that `QueueUserWorkItem` doesn't have any built-in capability to handle reporting errors.

In addition, `QueueUserWorkItem` does not give you any built-in methods to communicate between the background threads and the foreground thread. It doesn't provide any built-in means for you to detect completion, track progress, pause tasks, or cancel tasks. When you need those capabilities, you can turn to the `BackgroundWorker` component, which is built on top of the `QueueUserWorkItem` functionality.

The `BackgroundWorker` component was built on top of the `System.ComponentModel.Component` class to facilitate design-level support. However, `BackgroundWorker` is quite useful in code that doesn't include the designer support. In fact, most of the time when I use `BackgroundWorker`, it is not in a form class.

The simplest use of `BackgroundWorker` is to create a method that matches the delegate signature, attach that method to `BackgroundWorker`'s `DoWork` event, and then call the `RunWorkerAsync()` method of `BackgroundWorker`:

```
BackgroundWorker backgroundWorkerExample =
    new BackgroundWorker();
backgroundWorkerExample.DoWork += (sender, doWorkEventArgs) =>
{
    // body of work elided
};
backgroundWorkerExample.RunWorkerAsync();
```

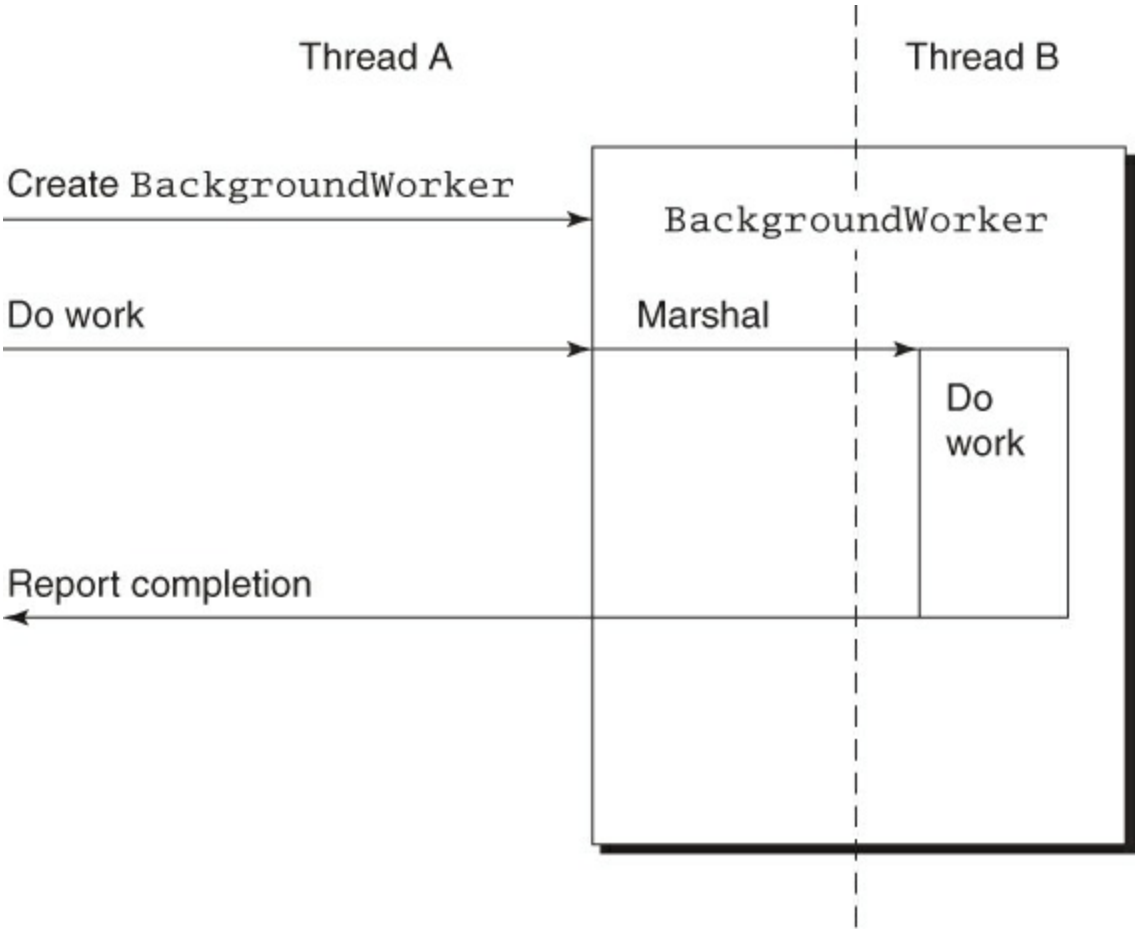
In this pattern, `BackgroundWorker` performs exactly the same function as `ThreadPool.QueueUserWorkItem`. The `BackgroundWorker` class performs its background tasks using `ThreadPool` and by using `QueueUserWorkItem`

internally.

The power of `BackgroundWorker` comes with the framework that is already built in for these other common scenarios. `BackgroundWorker` uses events to communicate between the foreground and background threads. When the foreground thread launches a request, `BackgroundWorker` raises the `DoWork` event on the background thread. The `DoWork` event handler reads any parameters and begins doing the work.

When the background thread procedure has finished (as defined by the exit of the `DoWork` event handler), `BackgroundWorker` raises the `RunWorkerCompleted` event on the foreground thread, as shown in Figure 4.2. The foreground thread can now do any necessary postprocessing after the background thread has completed.

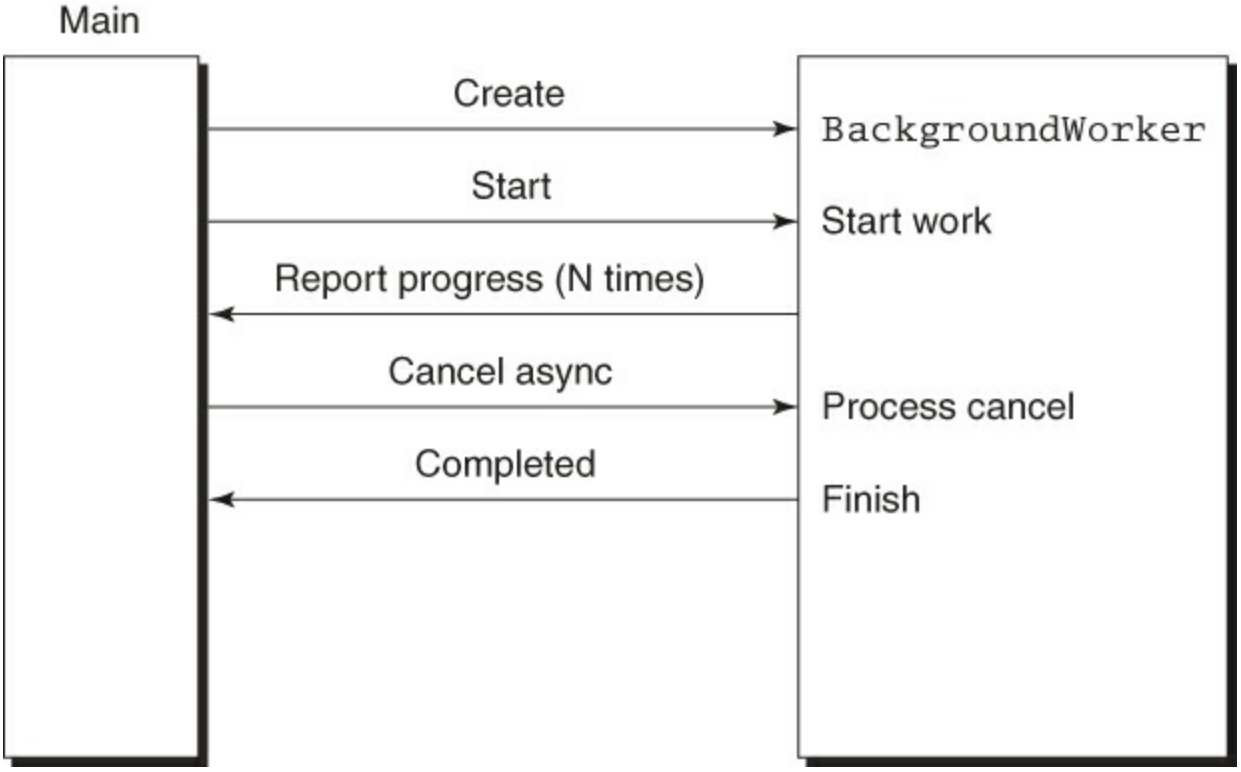
Figure 4.2 The `BackgroundWorker` class can report completion to an event handler defined in the foreground thread. You register the event handler for the completion event, and `BackgroundWorker` raises that event when your `DoWork` delegate has completed execution.



In addition to the events raised by `BackgroundWorker`, properties can be manipulated to control how the foreground and background thread interact. The `WorkerSupportsCancellation` property lets `BackgroundWorker` know that the background thread knows how to interrupt an operation and exit. The `WorkerReportsProgress` property informs `BackgroundWorker` that the worker procedure will report progress to the foreground thread at regular intervals, as shown in Figure 4.3. In addition, `BackgroundWorker` forwards cancellation requests from the foreground thread to the background thread. The background thread procedure can check the `CancellationPending` flag and stop processing if necessary.

Figure 4.3 The `BackgroundWorker` class supports multiple events to request cancellation of the current task, reporting of progress to the foreground task, completion, and error reporting. `BackgroundWorker` defines the protocol and raises the events necessary to support any of these communication mechanisms. To report progress, your background

procedure must raise an event defined on `BackgroundWorker`. Your foreground task code must request that these extra events be raised and must register handlers on these events.



Finally, `BackgroundWorker` has a built-in protocol to report errors that occur in the background thread. In Item 34 (earlier in this chapter) I explain that exceptions cannot be thrown from one thread to another. If an exception is generated in the background thread and is not caught by the thread procedure, the thread will be terminated. Worse, the foreground thread does not receive any notification that the background thread has stopped processing. `BackgroundWorker` solves this problem by adding an `Error` property to `DoWorkEventArgs` and propagating that property to the `Error` property in the result arguments. Your worker procedure catches all excepti

\ons and sets them to the error property. (Note that this is one of the rare occasions when catching all exceptions is recommended.) Simply return from the background thread procedure, and handle the error in the event handler for the foreground results.

Earlier I said that I often use `BackgroundWorker` in classes that aren't the

`Form` class, and even in non-Windows Forms applications, such as services or Web services. This works fine, but it does have some caveats. When `BackgroundWorker` determines that it is running in a Windows Forms application and the form is visible, the `ProgressChanged` and `RunWorkerCompleted` events are marshaled to the graphical user interface (GUI) thread via a marshaling control and `Control.BeginInvoke` (see Item 16 later in this chapter). In other scenarios, those delegates are simply called on a free thread pool thread. As you will see in Item 16, that behavior may affect the order in which events are received.

Finally, because `BackgroundWorker` is built on `QueueUserWorkItem`, you can reuse `BackgroundWorker` for multiple background requests. You need to check the `IsBusy` property of `BackgroundWorker` to see whether `BackgroundWorker` is currently running a task. When you need to have multiple background tasks running, you can create multiple `BackgroundWorker` objects. Each will share the same thread pool, so you have multiple tasks running just as you would with `QueueUserWorkItem`. You need to make sure that your event handlers use the correct sender property. This practice ensures that the background threads and foreground threads are communicating correctly.

`BackgroundWorker` supports many of the common patterns that you will use when you create background tasks. By using it you can reuse that implementation in your code, adding any of those patterns as needed. You don't have to design your own communication protocols between foreground and background threads.

Item 39: Understand Cross-Thread Calls in XAML environments

Windows controls use the Component Object Model (COM) single-threaded apartment (STA) model because those underlying controls are apartment-threaded. Furthermore, many of the controls use the **message pump** for many operations. This model says that all function calls to each control must be on the same thread that created the control. `Invoke` (and `BeginInvoke` and `EndInvoke`) marshals method calls to the proper thread. The underlying code

for both models is similar, so I focus the discussion on the Windows Forms API. When there are differences in the calling conventions, I give both versions. There's quite a bit of complicated code doing this, but we'll get to the bottom of it.

First, let's look at a simple bit of generic code that will make your life much easier when you run into this situation. **Anonymous delegates** provide a shortcut for wrapping small methods that are used only in one context. Unfortunately, anonymous delegates don't work with methods—such as `Control.Invoke`—that use the abstract `System.Delegate` type:

```
private void UpdateTime()
{
    Action action = () => textBlock1.Text = DateTime.Now.ToStri
    if (System.Threading.Thread.CurrentThread !=
        textBlock1.Dispatcher.Thread)
    {
        textBlock1.Dispatcher.Invoke
            (System.Windows.Threading.DispatcherPriority.Norma
            action);
    }
    else
    {
        action();
    }
}
```

In Windows Forms, you use `Control.Invoke` perform the marshaling:

```
private void OnTick(object sender, EventArgs e)
{
    Action action = () =>
        toolStripStatusLabel1.Text =
        DateTime.Now.ToLongTimeString();
    if (this.InvokeRequired)
        this.Invoke(action);
    else
        action();
}
```

That idiom further obscures the actual logic of the event handler, making the code less readable and harder to maintain. It also introduces a delegate definition whose only purpose is to provide a method signature for the abstract

delegate.

A small bit of generic coding can make that much easier. The following XAML`ControlExtensions` static class contains generic methods for any invoke delegate having up to two parameters. You can add more overloads by adding more parameters. Further, it contains methods that use those delegate definitions to call the target, either directly or through the marshaling provided by the dispatcher: .

```
public static class XAMLControlExtensions
{
    public static void InvokeIfNeeded(
        this System.Windows.Threading.DispatcherObject ctl,
        Action doit,
        System.Windows.Threading.DispatcherPriority priority)
    {
        if (System.Threading.Thread.CurrentThread !=
            ctl.Dispatcher.Thread)
        {
            ctl.Dispatcher.Invoke(priority,
                doit);
        }
        else
        {
            doit();
        }
    }
    public static void InvokeIfNeeded<T>(
        this System.Windows.Threading.DispatcherObject ctl,
        Action<T> doit,
        T args,
        System.Windows.Threading.DispatcherPriority priority)
    {
        if (System.Threading.Thread.CurrentThread !=
            ctl.Dispatcher.Thread)
        {
            ctl.Dispatcher.Invoke(priority,
                doit, args);
        }
        else
        {
            doit(args);
        }
    }
}
```

You can create a similar set of extensions for Windows Forms controls:

```
public static class ControlExtensions
{
    public static void InvokeIfNeeded(
        this Control ctl, Action doit)
    {
       if (ctl.IsHandleCreated == false)
            doit();
        else if (ctl.InvokeRequired)
            ctl.Invoke(doit);
        else
            doit();
    }

    public static void InvokeIfNeeded<T>(this Control ctl,
        Action<T> doit, T args)
    {
        if (ctl.IsHandleCreated == false)
            throw new InvalidOperationException(
            "Window handle for ctl has not been created");
        else if (ctl.InvokeRequired)
            ctl.Invoke(doit, args);
        else
            doit(args);
    }
    // versions built on 3,4 parameters elided
    public static void InvokeAsync(
        this Control ctl, Action doit)
    {
        ctl.BeginInvoke(doit);
    }

    public static void InvokeAsync<T>(this Control ctl,
        Action<T> doit, T args)
    {
        ctl.BeginInvoke(doit, args);
    }
}
```

Using `InvokeIfNeeded` greatly simplifies the code that handles events in a (possibly) multithreaded environment:

```
private void OnTick(object sender, EventArgs e)
{
    this.InvokeAsync(() => toolStripStatusLabel1.Text =
```

```
        DateTime.Now.ToLongTimeString());
}
```

The WPF version does not have an `InvokeRequired()` method call. Instead, you examine the identity of the current thread and compare it to the thread on which all control interaction should take place. `DispatcherObject` is the base class for many of the WPF controls. It handles the dispatch operations between threads for WPF controls. Also, notice that in WPF, you can specify the priority for the event handler action. That's because WPF applications use two UI threads. One thread handles the UI rendering pipeline so that the UI can always continue to render any animations or other actions. You can specify the priority to control which actions are more important for your users: either the rendering or the handling of a particular background event.

This code has several advantages. The body of the event handler logic is read inside the event handler, even though it is using an anonymous delegate definition. It's much more readable and easier to maintain than repeated logic using the dispatcher in your application code. Inside the `WPFControlExtensions` class, the generic method handles the check for `InvokeRequired`, or comparing thread identities, meaning that you don't need to remember it each time. I don't use these methods if I know I'm writing code for a single-threaded application, but if I think my code might end up in a multithreaded environment, I use this version for generality.

Before you use this idiom in all your event handlers, let's look closely at the work done by `InvokeRequired`. These aren't free calls, and it's not advisable to simply apply this idiom everywhere.`InvokeRequired` determines whether the current code, on the one hand, is executing on the thread that created the control or, on the other hand, is executing on another thread and therefore needs to be marshaled. In most cases, this property contains a reasonably simple implementation. It checks the current thread ID and compares it to the thread ID for the control in question. If they match, `Invoke` is not required. If they don't match, `Invoke` is required. That comparison doesn't take much time..

But there are some interesting edge cases. Suppose the control in question has not yet been created. That can happen when a parent control has been created and the current control is in the process of being instantiated. The C# object exists, but the underlying window handle is still `null`. In that case, there's

nothing to compare. The framework wants to help you out, and that takes time. The framework walks the tree of parent controls to see whether any of them has been created. If the framework finds a window that has been created, that window is used as the marshaling window. That's a reasonably safe conclusion, because parent controls are responsible for creating child controls. This approach guarantees that the child controls will be created on the same thread as the parent control found by the framework. After finding a suitable parent control, the framework performs the same check mentioned earlier, checking the current thread ID against the control thread ID.

But, of course, if the framework can't find any parent window that's been created, the framework needs to find some kind of window. If none of the windows in the hierarchy exists, the framework looks for the **parking window,** a special window that's used to hide from you some of the strange behavior of the Win32 API. In short, there are some changes to windows that require destroying and re-creating the Win32 windows. (Modifying certain styles requires a window be destroyed and re-created.) The parking window is used to hold child windows whenever a parent window must be destroyed and re-created. During that time, there is a period when the UI thread can be found only from the parking window.

In WPF, some of this has been simplified through the use of the `Dispatcher` class. Each thread has a dispatcher. The first time you ask a control for its dispatcher, the library looks to see whether that thread already has a dispatcher. If it does, the library returns that dispatcher. If not, a new `Dispatcher` object is created and associated with the control and its thread.

But there are still holes and possible failures. It's possible that none of the windows is yet created, even the parking window. In that case, `InvokeRequired` always returns false, indicating that you don't need to marshal the call to another thread. This situation is somewhat dangerous, because it has a chance of being wrong, but it's the best that the framework can do. Any method call you make that requires the window handle to exist will fail. There's no window, so trying to use it will fail. On the other hand, marshaling will certainly fail. If the framework can't find any marshaling control, then there's no way for the framework to marshal the current call to the UI thread. The framework chooses a possible later failure instead of a certain

immediate failure. Luckily, this situation is rather rare in practice. In WPF, the `Dispatcher` contains extra code to protect against this situation.

Let's summarize what you've learned about `InvokeRequired`. Once your controls are created, `InvokeRequired` is reasonably fast and always safe. However, if the target control has not been created, `InvokeRequired` can take much longer, and if none of the controls has been created, `InvokeRequired` takes a long time to give you an answer that's probably not even correct. Even though `Control.InvokeRequired` can be a bit expensive, it's still quite a bit cheaper than a call to `Control.Invoke` when it's not necessary. In WPF, some of the edge cases have been optimized and work better than they do in the Windows Forms implementation.

Now let's look at `Control.Invoke` and what it does. (`Control.Invoke` can do quite a bit of work, so this discussion is greatly simplified.) First, there's the special case when you've called `Invoke` even though you're running on the same thread as the control. It's a short-circuit path, and the framework simply calls your delegate. Calling `Control.Invoke()` when `InvokeRequired()` returns false means that your code does a bit of extra work, but it is safe.

The interesting case happens when you actually need to call `Invoke`. `ControlInvoke` handles the cross-thread calls by posting a message to the target control's message queue. `Control.Invoke` creates a private structure that contains everything needed to call the delegate. That includes all parameters, a reference to the call stack, and the delegate target. The parameters are copied to avoid any modification of the value of the parameters before the target delegate is called (remember that this is a multithreaded world.)

After this structure is created and added to a queue, a message is posted to the target control. `Control.Invoke` then does a combination of spin-wait and sleep while it waits for the UI thread to process the message and invoke the delegate. This part of the process contains an important timing issue. When the target control processes the `Invoke` message, it doesn't simply process one delegate. It processes all `Invoke` delegates in the queue. If you always use the synchronous version of `Control.Invoke`, you won't see any effects. However, if you mix `Control.Invoke` and `Control.BeginInvoke()`, you'll see different behavior. I return to this toward the end of this item, but for now,

understand that the control's `WndProc` processes every waiting `Invoke` message whenever it processes any `Invoke` messages. You have a little more control in WPF, because you can control the priority of the asynchronous operation. You can instruct the dispatcher to queue the message for processing (1) based on system or application conditions, (2) in the normal order, or (3) as a high-priority message.

Of course, these delegates can throw exceptions, and exceptions can't cross thread boundaries. The control wraps the call to your delegate in a `try` / `catch` block and catches all exceptions. Any exceptions are copied into a structure that is examined in the worker thread after the UI thread has finished its processing.

After the UI thread process finishes, `Control.Invoke` looks for any exceptions that were thrown from the delegate on the UI thread. If there are any exceptions, `Invoke` rethrows them on the background thread. If there aren't any exceptions, normal processing continues. As you can see, that's quite a bit of processing to call a method.

`Control.Invoke` blocks the background thread while the marshaled call is being processed. It gives the impression of synchronous behavior even though multiple threads are involved.

But that may not be what you need for your application. Many times, a progress event is raised by a worker thread and you want the worker thread to continue processing rather than wait for a synchronous update to the UI. That's when you use `BeginInvoke`. This method does much of the same processing as `Control.Invoke`. However, after posting the messages to the target control, `BeginInvoke` returns immediately rather than wait for the target delegate to finish. `BeginInvoke` allows you to post a message for future processing and immediately unblock the calling thread. You can add corresponding generic asynchronous methods to the `ControlExtensions` class to make it easier to process cross-thread UI calls asynchronously. You gain less benefit from these methods than the earlier ones, but for consistency, let's add them to the `ControlExtensions` class:

```
public static void QueueInvoke(this Control ctl, Action doit)
{
```

```
    ctl.BeginInvoke(doit);
}

public static void QueueInvoke<T>(this Control ctl,
    Action<T> doit, T args)
{
    ctl.BeginInvoke(doit, args);
}
```

The `QueueInvoke` method does not test `InvokeRequired` first. That's because you may want to invoke a method asynchronously even if you are currently executing on the UI thread. `BeginInvoke()` does that for you. `Control.Invoke` posts the message to the control and returns. The target control processes that message when it next checks its message queue. It's not really asynchronous if `BeginInvoke` is called on the UI thread. Rather, the processing is still synchronous; you just perform the action some time after the current operation.

I'm ignoring the `Asynch` result returned from `BeginInvoke`. In practice, UI updates rarely have return values. That makes it much easier to process those messages asynchronously. Simply call `BeginInvoke` and expect the delegate methods to be invoked at some later time. You need to code these delegate methods defensively, because any exceptions are swallowed in the cross-thread marshaling.

Before we finish this item, let's clean up a loose end inside the control's `WndProc`. Recall that when `WndProc` receives the `Invoke` message `WndProc` processes every delegate on `InvokeQueue`. You can run into timing problems if you expect events to be processed in a certain order and you are using a mixture of `Invoke` and `BeginInvoke`. You can guarantee that the delegates called by `Control.BeginInvoke` (or `Control.Invoke`) are processed in the order they are received. `BeginInvoke` adds a delegate to the queue. Any later calls to `Control.Invoke` process all messages on the queue, including those previously added with a call to `BeginInvoke()`. Processing a delegate "some time later" means that you can't control when "some time later" actually happens. Processing a delegate "now" means that the application processes all waiting asynchronous delegates and then processes this one. It is possible that one of the waiting delegate targets for `BeginInvoke` will change program state before your `Invoke` delegate is called. You need to code defensively and

ensure that you recheck program state inside the delegate rather than rely on the state passed from some time in the past when `Control.Invoke` was called.

Very simply, this version of the original handler rarely displays the extra text:

```
private void OnTick(object sender, EventArgs e)
{
    this.InvokeAsync(() => toolStripStatusLabel1.Text =
        DateTime.Now.ToLongTimeString());
    toolStripStatusLabel1.Text += "  And set more stuff";
}
```

That's because the code invokes the first change by queuing up the message, and the change is made when the next messages are handled. That's after the next statement adding more text to the label.

`Invoke` and `InvokeRequired` do quite a bit of work on your behalf. All this work is required because Windows Forms controls are built on the single-threaded apartment model. That legacy behavior continues under the new WPF libraries. Underneath all the new .NET Framework code, the Win32 API and window messages are still lurking. That message passing and thread marshaling can lead to unexpected behavior. You need to understand what those methods do and work with their behavior.

Item 40: Use lock() as Your First Choice for Synchronization

Threads need to communicate with each other. Somehow, you need to provide a safe way for various threads in your application to send and receive data. However, sharing data between threads introduces the potential for data integrity errors in the form of synchronization issues. Therefore, you need to be certain that the current state of every shared data item is consistent. You achieve this safety by using **synchronization primitives** to protect access to the shared data. Synchronization primitives ensure that the current thread is not interrupted until a critical set of operations is completed.

There are many primitives available in the .NET BCL that you can use to safely ensure that access to shared data is synchronized. Only one pair of them

—`Monitor.Enter()` and `Monitor.Exit()`—was given special status in the C# language. `Monitor.Enter()` and `Monitor.Exit()` implement a **critical section** block. Critical sections are such a common synchronization technique that the language designers added support for them using the `lock()` statement. You should follow that example and make `lock()` your primary tool for synchronization.

The reason is simple: The compiler generates consistent code, but you may make mistakes some of the time. The C# language introduces the *lock* keyword to control synchronization for multithreaded programs. The lock statement generates exactly the same code as if you used `Monitor.Enter()` and `Monitor.Exit()` correctly. Furthermore, it's easier and it automatically generates all the exception-safe code you need.

However, under two conditions `Monitor` gives you necessary control that you can't get when you use `lock()`. First, be aware that `lock` is lexically scoped. This means that you can't enter a `Monitor` in one lexical scope and exit it in another when using the `lock` statement. Thus, you can't enter a `Monitor` in a method and exit it inside a lambda expression defined in that method (see Item 41, Chapter 5). The second reason is that `Monitor.Enter` supports a time-out, which I cover later in this item.

You can lock any reference type by using the `lock` statement:

```
public int TotalValue
{
    get
    {
        lock (syncHandle) { return total; }
    }
}

public void IncrementTotal()
{
    lock (syncHandle) { total++; }
}
```

The `lock` statement gets the exclusive monitor for an object and ensures that no other thread can access the object until the lock is released. The preceding sample code, using `lock()`, generates code that has the same behavior as the

following version, using `Monitor.Enter()` and `Monitor.Exit()`:

```
public void IncrementTotal()
{
    object tmpObject = syncHandle;
    System.Threading.Monitor.Enter(tmpObject);
    try
    {
        total++;
    }
    finally
    {
        System.Threading.Monitor.Exit(tmpObject);
    }
}
```

The `lock` statement provides many checks that help you avoid common mistakes. It checks that the type being locked is a reference type, as opposed to a value type. The `Monitor.Enter` method does not include such safeguards. This routine, using `lock()`, doesn't compile:

```
public void IncrementTotal()
{
    lock (total) // compiler error: can't lock value type
    {
        total++;
    }
}
```

But this does:

```
public void IncrementTotal()
{
    // really doesn't lock total.
    // locks a box containing total.
    Monitor.Enter(total);
    try
    {
        total++;
    }
    finally
    {
        // Might throw exception
        // unlocks a different box containing total
        Monitor.Exit(total);
```

```
    }
}
```

`Monitor.Enter()` compiles because its official signature takes a `System.Object`. You can coerce `total` into an object by boxing it. `Monitor.Enter()` actually locks the box containing `total`. That's where the first bug lurks. Imagine that thread 1 enters `IncrementTotal()` and acquires a lock. Then, while incrementing `total`, the second thread calls `IncrementTotal()`. Thread 2 now enters `IncrementTotal()` and acquires the lock. It succeeds in acquiring a different lock, because `total` gets put into a different box. Thread 1 has a lock on one box containing the value of `total`. Thread 2 has a lock on another box containing the value of `total`. You've got extra code in place, and no synchronization.

Then you get bitten by the second bug: When either thread tries to release the lock on `total`, the `Monitor.Exit()` method throws a `SynchronizationLockException`. That's because `total` goes into yet another box to coerce it into the method signature for `Monitor.Exit`, which also expects a `System.Object` type. When you release the lock on this box, you unlock a resource that is different from the resource that was used for the lock. `Monitor.Exit()` fails and throws an exception.

Of course, some bright soul might try this:

```
public void IncrementTotal()
{
    // doesn't work either:
    object lockHandle = total;
    Monitor.Enter(lockHandle);
    try
    {
        total++;
    }
    finally
    {
        Monitor.Exit(lockHandle);
    }
}
```

This version doesn't throw any exceptions, but neither does it provide any synchronization protection. Each call to `IncrementTotal()` creates a new box

and acquires a lock on that object. Every thread succeeds in immediately acquiring the lock, but it's not a lock on a shared resource. Every thread wins, and `total` is not consistent.

There are subtler errors that `lock` also prevents. `Enter()` and `Exit()` are two separate calls, so you can easily make the mistake of acquiring and releasing different objects. This action may cause a `SynchronizationLockException`. But if you happen to have a type that locks more than one synchronization object, it's possible to acquire two different locks in a thread and release the wrong one at the end of a critical section.

The `lock` statement automatically generates exception-safe code, something many of us humans forget to do. Also, it generates more-efficient code than `Monitor.Enter()` and `Monitor.Exit()`, because it needs to evaluate the target object only once. So, by default, you should use the `lock` statement to handle the synchronization needs in your C# programs.

However, there is one limitation to the fact that `lock` generates the same MSIL as `Monitor.Enter()`. The problem is that `Monitor.Enter()` waits forever to acquire the lock. You have introduced a possible deadlock condition. In large enterprise systems, you may need to be more defensive in how you attempt to access critical resources. `Monitor.TryEnter()` lets you specify a time-out for an operation and attempt a workaround when you can't access a critical resource.

```
public void IncrementTotal()
{
    if (!Monitor.TryEnter(syncHandle, 1000)) // wait 1 second
        throw new PreciousResourceException
            ("Could not enter critical section");
    try
    {
        total++;
    }
    finally
    {
        Monitor.Exit(syncHandle);
    }
}
```

You can wrap this technique in a handy little generic class:

```
public sealed class LockHolder<T> : IDisposable
    where T : class
{
    private T handle;
    private bool holdsLock;

    public LockHolder(T handle, int milliSecondTimeout)
    {
        this.handle = handle;
        holdsLock = System.Threading.Monitor.TryEnter(
            handle, milliSecondTimeout);
    }

    public bool LockSuccessful
    {
        get { return holdsLock; }
    }

    public void Dispose()
    {
        if (holdsLock)
            System.Threading.Monitor.Exit(handle);
        // Don't unlock twice
        holdsLock = false;
    }
}
```

You would use this class in the following manner:

```
object lockHandle = new object();

using (LockHolder<object> lockObj = new LockHolder<object>
    (lockHandle, 1000))
{
    if (lockObj.LockSuccessful)
    {
        // work elided
    }
}
// Dispose called here.
```

The C# team added implicit language support for `Monitor.Enter()` and `Monitor.Exit()` pairs in the form of the `lock` statement because it is the most common synchronization technique that you will use. The extra checks that the compiler can make on your behalf make it easier to create synchronization code in your application. The lock statement is the only primitive that is

guaranteed by the language specification to produce ordering of side effects. Therefore, `lock()` is the best choice for most synchronization between threads in your C# applications.

However, `lock` is not the only choice for synchronization. In fact, when you are synchronizing access to numeric types or are replacing a reference, the `System.Threading.Interlocked` class supports synchronizing single operations on objects. `System.Threading.Interlocked` has a number of methods that you can use to access shared data so that a given operation completes before any other thread can access that location. It also gives you a healthy respect for the kinds of synchronization issues that arise when you work with shared data.

Consider this method:

```
public void IncrementTotal() =>
    total++;
```

As written, interleaved access could lead to an inconsistent representation of the data. An increment operation is not a single machine instruction. The value of `total` must be fetched from main memory and stored in a register. Then the value of the register must be incremented, and the new value from the register must be stored back into the proper location in main memory. If another thread reads the value after the first thread, the second thread grabs the value from main memory but before storing the new value, thereby causing data inconsistency.

Suppose two threads interleave calls to `IncrementTotal`. Thread A reads the value of 5 from `total`. At that moment, the active thread switches to thread B. Thread B reads the value of 5 from `total`, increments it, and stores 6 in the value of `total`. At this moment, the active thread switches back to thread A. Thread A now increments the register value to 6 and stores that value in `total`. As a result, `IncrementTotal()` has been called twice—once by thread A, and once by thread B—but because of untimely interleaved access, the end effect is that only one update has occurred. These errors are hard to find, because they result from interleaved access at exactly the wrong moment.

You could use `lock()` to synchronize this operation, but there is a simpler

way. The `Interlocked` class has a simple method that fixes the problem: `InterlockedIncrement`. If you rewrite `IncrementTotal` as follows, the increment operation cannot be interrupted and both increment operations will always be recorded:

```
public void IncrementTotal() =>
    System.Threading.Interlocked.Increment(ref total);
```

The `Interlocked` class contains other methods to work with built-in data types. `Interlocked.Decrement()` decrements a value. `Interlocked.Exchange()` switches a value with a new value and returns the current value. You'd use `Interlocked.Exchange()` to set new state and return the preceding state. For example, suppose you want to store the user ID of the last user to access a resource. You can call `Interlocked.Exchange()` to store the current user ID while at the same time retrieving the previous user ID.

Finally, there is the `CompareExchange()` method, which reads the value of a piece of shared data and, if the value matches a sought value, updates it. Otherwise, nothing happens. In either case, `CompareExchange` returns the preceding value stored at that location. In the next section, Item 14 shows how to use `CompareExchange` to create a private lock object inside a class.

The `Interlocked` class and `lock()` are not the only synchronization primitives available. The `Monitor` class also includes the `Pulse` and `Wait` methods, which you can use to implement a consumer/producer design. You can also use the `ReaderWriterLockSlim` class for those designs in which many threads are accessing a value that few threads are modifying. `ReaderWriterLockSlim` contains several improvements over the earlier version of `ReaderWriterLock`. You should use `ReaderWriterLockSlim` for all new development.

For most common synchronization problems, examine the `Interlocked` class to see whether you can use it to provide the capabilities you need. With many single operations, you can. Otherwise, your first choice is the `lock()` statement. Look beyond those only when you need special-purpose locking capability..

Item 41: Use the Smallest Possible Scope for Lock Handles

When you write concurrent programs, you want to localize the synchronization primitives to the best of your ability. The more places there are in an application where you can use a synchronization primitive, the more difficult it will be to avoid deadlocks, missing locks, or other concurrent programming issues. It's a matter of scale: The more places you have to look, the harder it will be to find a particular issue.

In object-oriented programming, you use private member variables to minimize (not remove, but minimize) the number of locations you need to search for state changes. In concurrent programs, you want to do the same thing by localizing the object that you use to provide synchronization.

Two of the most widely used locking techniques are just plain wrong when seen from that viewpoint. `lock(this)` and `lock(typeof(MyType))` have the nasty effect of creating your lock object based on a publicly accessible instance.

Suppose you write code like this:

```
public class LockingExample
{
    public void MyMethod()
    {
        lock (this)
        {
            // elided
        }
    }
    // elided
}
```

Now suppose that one of your clients—let's call him Alexander—figures he needs to lock something. Alexander writes this:

```
LockingExample x = new LockingExample();
lock (x)
    x.MyMethod();
```

That type of locking strategy can easily cause deadlock. Client code has acquired a lock on the `LockingExample` object. Inside `MyMethod`, your code acquires another lock on the same object. That's all fine and good, but one day soon, different threads will lock the `LockingExample` object from somewhere in the program. Deadlock issues happen, and there's no good way to find where the lock was acquired. It could be anywhere.

You need to change your locking strategy. There are three strategies to avoid this problem.

First, if you are protecting an entire method, you can use `MethodImplAttribute` to specify that a method is synchronized:

```
[MethodImpl(MethodImplOptions.Synchronized)]
public void IncrementTotal()
{
    total++;
}
```

Of course, that's not the most common practice.

Second, you can mandate that a developer can create a lock only on the current type or the current object. Namely, you recommend that everyone use `lock(this)` or `lock(MyType)`. That would work—if everyone followed your recommendation. It relies on all clients in the entire world knowing that they can never lock on anything except the current object or the current type. It will fail, because it can't be enforced.

The best answer is the third choice. In general cases, you can create a handle that can be used to protect access to the shared resources of an object. That handle is a private member variable and therefore cannot be accessed outside the type being used. You can ensure that the object used to synchronize access is private and is not accessible by any nonprivate properties. That policy ensures that any lock primitives on a given object are local to a given location.

In practice, you create a variable of `System.Object` to use as a synch handle. Then you lock that handle when you need to protect access to any of the members of the class. But you need to be a bit careful when you create the synch handle. You want to make sure that you do not end up with extra copies

of the synch handle because you've had threads interleave memory access at the wrong time. The `Interlocked` class's `CompareExchange` method tests a value and replaces it if necessary. You can use that method to ensure that you allocate exactly one synch handle object in your type.

Here's the simplest code:

```
private object syncHandle = new object();

public void IncrementTotal()
{
    lock (syncHandle)
    {
        // code elided
    }
}
```

You may find that you don't often need the lock and you want to create the synch object only when you need it. In those cases, you can get a little fancier with the synch handle creation:

```
private object syncHandle;

private object GetSyncHandle()
{
    System.Threading.Interlocked.CompareExchange(
        ref syncHandle, new object(), null);
    return syncHandle;
}

public void AnotherMethod()
{
    lock (GetSyncHandle())
    {
        // ... code elided
    }
}
```

The `syncHandle` object is used to control access to any of the shared resources in your class. The private method `GetSyncHandle()` returns the single object that acts as the synch target. The `CompareExchange` call, which can't be interrupted, ensures that you create only one copy of the synch handle. It compares the value of `syncHandle` with `null`, and, if `syncHandle` is `null`,

then `CompareExchange` creates a new object and assigns that object to `syncHandle`.

That handles any locking that you might do for instance methods, but what about static methods? The same technique works, but you create a static synch handle so that there is one synch handle that is shared by all instances of the class.

Of course, you can lock sections of code that are smaller than a single method. You can create synchronization blocks around any section of code inside a method (or, for that matter, a property accessor or indexer). However, whatever the scope is, you need to do what you can to minimize the scope of locked code.

```
public void YetAnotherMethod()
{
    DoStuffThatIsNotSynchronized();
    int val = RetrieveValue();
    lock (GetSyncHandle())
    {
        // ... code elided
    }
    DoSomeFinalStuff();
}
```

If you create or use a lock inside a lambda expression, however, you must be careful. The C# compiler creates a closure around lambda expressions. This, combined with the deferred execution model supported by C# 3.0 constructs, means that it will be difficult for developers to determine when the lexical scope of the lock ends. That makes this approach more prone to deadlock issues, because developers may not be able to determine whether code is inside a locked scope.

I close with a couple of other recommendations on locking. If you find that you want to create different locking handles for different values in your class, that is a strong indication that you should break the current class into multiple classes. The class is trying to do too many things. If you need to protect access to some variables and use other locks to protect other variables in the class, that's a strong indication that you should split the class into different types having different responsibilities. It will be much easier to control the

synchronization if you view each type as a single unit. Each class that holds shared data—data that must be accessed or updated by different threads—should use a single synchronization handle to protect access to those shared resources.

When you decide what to lock, pick a private field that's not visible to any callers. Do not lock a publicly visible object. Locking publicly visible objects requires that all developers always and forever follow the same practice, and it enables client code to easily introduce deadlock issues.

Item 42: Avoid Calling Unknown Code in Locked Sections

At one end of the scale are problems that are caused by not locking enough. Then when you begin creating locks, the next most likely problem is that you may create deadlocks. Deadlocks occur when a thread blocks waiting for another thread complete while that thread is waiting on the first thread to complete. In the .NET Framework, you can have a special case in which cross-thread calls are marshaled in such a way that they emulate synchronous calls. It's possible to have two threads deadlocked when only one resource is locked. (Item 16, in the next section, demonstrates one such situation.)

You've already learned one of the simplest ways to avoid this problem: Item 38 discusses how using a private nonvisible data member as the target of the lock localizes the locking code in your application. But there are other ways to introduce a deadlock. If you invoke unknown code from inside a synchronized region of code, you introduce the possibility that another thread will deadlock your application.

For example, suppose you write code like this to handle a background operation:

```
public class WorkerClass
{
    public event EventHandler<EventArgs> RaiseProgress;
    private object syncHandle = new object();

    public void DoWork()
```

```
    {
        for (int count = 0; count < 100; count++)
        {
            lock (syncHandle)
            {
                System.Threading.Thread.Sleep(100);
                progressCounter++;
                RaiseProgress?.Invoke(this, EventArgs.Empty);
            }
        }
    }

    private int progressCounter = 0;
    public int Progress

    {
        get
        {
            lock (syncHandle)
                return progressCounter;
        }
    }
}
```

The `raiseProgress()` method notifies all listeners of updated progress. Note that any listeners can be registered to handle that event. In a multithreaded program a typical event handler might look like this:

```
static void engine_RaiseProgress(object sender, EventArgs e)
{
    WorkerClass engine = sender as WorkerClass;
    if (engine != null)
        Console.WriteLine(engine.Progress);
}
```

Everything runs fine, but only because you got lucky. It works because the event handler runs in the context of the background thread.

However, suppose this application were a Windows Forms application, and you needed to marshal the event handler back to your UI thread (see Item 16). `Control.Invoke` marshals the call to the UI thread, if necessary. Furthermore, `Control.Invoke` blocks the original thread until the target delegate has completed. That sounds innocent enough. You're operating on a different thread now, but that should be just fine.

The second important action causes the whole process to deadlock. Your event handler makes a callback into the engine object in order to get the status details. The `Progress` accessor, now running on a different thread, can't acquire the same lock.

The `Progress` accessor locks the synchronization handle. That looks correct from the local context of this object, but it's not. The UI thread is trying to lock the same handle already locked in the background thread. But the background thread is suspended waiting for the event handler to return, and the background thread already has the synch handle locked. You're deadlocked.

Table 4.1 shows the call stack. The table shows why it's difficult to debug these problems. This scenario has eight methods on the call stack between the first lock and the second attempted lock. Worse, the thread interleaving happens inside the framework code. You may not even see it.

The root problem is that you've tried to reacquire a lock. Because you cannot know what actions may be taken by code outside your control, you should try to avoid invoking the callback from inside the locked region. In this example, this means that you must raise the progress-reporting event from outside the locked section:

```
public void DoWork()
{
    for (int count = 0; count < 100; count++)
    {
        lock (syncHandle)
        {
            System.Threading.Thread.Sleep(100);
            progressCounter++;
        }
        RaiseProgress?.Invoke(this, EventArgs.Empty);
    }
}
```

Now that you've seen the problem, it's time to make sure you understand the various ways that calls to unknown code might creep into your applications. Obviously, raising any publicly accessible event is a callback. Invoking a delegate that was passed as a parameter, or set through a public API, is a callback. Invoking a lambda expression that's passed in as a parameter might

also be calling unknown code (see Item 7 in *Effective C#, Third Edition*).

Table 4.1 Call Stack for Code That Marshals Execution Between a Background Thread and a Foreground Thread That Updates a Window Display

Method	Thread
`DoWork`	BackgroundThread
`raiseProgress`	BackgroundThread
`OnUpdateProgress`	BackgroundThread
`engine_OnUpdateProgress`	BackgroundThread
`Control.Invoke`	BackgroundThread
`UpdateUI`	UIThread
`Progress (property access)`	UIThread (deadlock)

Those sources of unknown code are rather easy to spot. But there is another possible location lurking in most classes: virtual methods. Any virtual method you invoke can be overridden by a derived class. That derived class, in turn, can invoke any method (public or protected) in your class. Any of those invocations can try to lock a shared resource again.

No matter how it happens, the pattern is similar. Your class acquires a lock. Then, while still in the synchronized section, it invokes a method that calls code beyond your control. That client code is an open-ended set of code that may eventually trace back into your class, even on another thread. You can't do anything to prevent that open-ended set of code from doing something that might be evil. So instead, you must prevent the situation: Don't call unknown code from inside locked sections of your code.

5. Dynamic Proramming

There are advantages to both static typing and dynamic typing. Dynamic typing can enable quicker development times and easier interoperability with dissimilar systems. Static typing enables the compiler to find classes of errors. Because the compiler can make those checks, runtime checks can be streamlined, which results in better performance. C# is a statically typed language and will remain one. However, for those times when dynamic languages provide more efficient solutions, C# contains dynamic features. Those features enable you to switch between static typing and dynamic typing when the need arises. The wealth of features that you have in static typing means that most of your C# code will be statically typed. This chapter shows you the problems suited for dynamic programming and the techniques you will use to solve those problems most efficiently.

Item 43: Understand the Pros and Cons of Dynamic

C#'s support for dynamic typing is meant to provide a bridge to other locations. It's not meant to encourage general dynamic language programming, but rather to provide a smoother transition between the strong, static typing associated with C# and those environments that use a dynamic typing model.

However, that doesn't mean you should restrict your use of dynamic to interoperating with other environments. C# types can be coerced into dynamic objects and treated as dynamic objects. Like everything else in this world, there's good and bad in treating C# objects as dynamic objects. Let's look at one example and go over what happens.

One of the limitations of C# generics is that in order to access methods beyond those defined in `System.Object`, you need to specify constraints. Furthermore, constraints must be in the form of a base class, a set of interfaces, or the special constraints for reference type, value type, and the existence of a public parameterless constructor. You can't specify that some known method is available. This can be especially limiting when you want to create a general method that relies on some operator, like +. Dynamic invocation can fix that.

As long as a member is available at runtime, it can be used. Here's a method that adds two dynamic objects, as long as there is an available operator + at runtime:

```
public static dynamic Add(dynamic left,
    dynamic right)
{
    return left + right;
}
```

This is my first discussion of dynamic, so let's look into what it's doing. Dynamic can be thought of as "`System.Object` with runtime binding." At compile time, dynamic variables have only those methods defined in `System.Object`. However, the compiler adds code so that every member access is implemented as a dynamic call site. At runtime, code executes to examine the object and determine if the requested method is available. (See Item 43 on implementing dynamic objects.) This is often referred to as "duck typing": If it walks like a duck and talks like a duck, it may as well be a duck. You don't need to declare a particular interface, or provide any compile-time type operations. As long as the members needed are available at runtime, it will work.

For this method above, the dynamic call site will determine if there is an accessible + operator for the actual runtime types of the two objects listed. All of these calls will provide a correct answer:

```
dynamic answer = Add(5, 5);
answer = Add(5.5, 7.3);
answer = Add(5, 12.3);
```

Notice that the answer must be declared as a dynamic object. Because the call is dynamic, the compiler can't know the type of the return value. That must be resolved at runtime. The only way to resolve the type of the return code at runtime is to make it a dynamic object. The static type of the return value is dynamic. Its runtime type is resolved at runtime.

Of course, this dynamic Add method is not limited to numeric type. You can add strings (because string does have an operator + defined):

```
dynamic label = Add("Here is ", "a label");
```

You can add a timespan to a date:

```
dynamic tomorrow = Add(DateTime.Now, TimeSpan.FromDays(1));
```

As long as there is an accessible operator +, the dynamic version of Add will work.

This opening explanation of dynamic might lead you to overuse dynamic programming. I've only discussed the pros of dynamic programming. It's time to consider the cons as well. You've left the safety of the type system behind, and with that, you've limited how the compiler can help you. Any mistakes in interpreting the type will only be discovered at runtime.

The result of any operation where one of the operands (including a possible this reference) is dynamic is itself dynamic. At some point, you'll want to bring those dynamic objects back into the static type system used by most of your C# code. That's going to require either a cast or a conversion operation:

```
dynamic answer = Add(5, 12.3);
int value = (int)answer;
string stringLabel = System.Convert.ToString(answer);
```

The cast operation will work when the actual type of the dynamic object is the target type, or can be cast to the target type. You'll need to know the correct type of the result of any dynamic operation to give it a strong type. Otherwise, the conversion will fail at runtime, throwing an exception.

Using dynamic typing is the right tool when you have to resolve methods at runtime without knowledge of the types involved. When you do have compile-time knowledge, you should use lambda expressions and functional programming constructs to create the solution you need. You could rewrite the Add method using lambdas like this:

```
public static TResult Add<T1, T2, TResult>(T1 left, T2 right,
    Func<T1, T2, TResult> AddMethod)
{
    return AddMethod(left, right);
}
```

Every caller would be required to supply the specific method. All the previous

examples could be implemented using this strategy:

```
var lambdaAnswer = Add(5, 5, (a, b) => a + b);
var lambdaAnswer2 = Add(5.5, 7.3, (a, b) => a + b);
var lambdaAnswer3 = Add(5, 12.3, (a, b) => a + b);
var lambdaLabel = Add("Here is ", "a label",
    (a, b) => a + b);
dynamic tomorrow = Add(DateTime.Now, TimeSpan.FromDays(1));
var finalLabel = Add("something", 3,
    (a, b) => a + b.ToString());
```

You can see that the last method requires you to specify the conversion from int to string. It also has a slightly ugly feel in that all those lambdas look like they could be turned into a common method. Unfortunately, that's just how this solution works. You have to supply the lambda at a location where the types can be inferred. That means a fair amount of code that looks the same to humans must be repeated because the code isn't the same to the compiler. Of course, defining the Add method to implement Add seems silly. In practice, you'd use this technique for methods that used the lambda but weren't simply executing it. It's the technique used in the .NET library `Enumerable.Aggregate()`. `Aggregate()` enumerates an entire sequence and produces a single result by adding (or performing some other operation):

```
var accumulatedTotal = Enumerable.Aggregate(sequence,
    (a, b) => a + b);
```

It still feels like you are repeating code. One way to avoid this repeated code is to use Expression Trees. It's another way to build code at runtime. The `System.Linq.Expression` class and its derived classes provide APIs for you to build expression trees. Once you've built the expression tree, you convert it to a lambda expression and compile the resulting lambda expression into a delegate. For example, this code builds and executes Add on three values of the same type:

```
// Naive Implementation. Read on for a better version
public static T AddExpression<T>(T left, T right)
{
    ParameterExpression leftOperand = Expression.Parameter(
        typeof(T), "left");
    ParameterExpression rightOperand = Expression.Parameter(
        typeof(T), "right");
    BinaryExpression body = Expression.Add(
```

```
        leftOperand, rightOperand);
    Expression<Func<T, T, T>> adder =
        Expression.Lambda<Func<T, T, T>>(
        body, leftOperand, rightOperand);
    Func<T, T, T> theDelegate = adder.Compile();
    return theDelegate(left, right);
}
```

Most of the interesting work involves type information, so rather than using `var` as I would in production code for clarity, I've specifically named all the types.

The first two lines create parameter expressions for variables named "left" and "right," both of type `T`. The next line creates an `Add` expression using those two parameters. The `Add` expression is derived from `BinaryExpression`. You should be able to create similar expressions for other binary operators.

Next, you need to build a lambda expression from the expression body and the two parameters. Finally, you create the `Func<T,T,T>` delegate by compiling the expression. Once compiled, you can execute it and return the result. Of course, you can call it just like any other generic method:

```
int sum = AddExpression(5, 7);
```

I added the comment above the last example indicating that this was a naïve implementation. DO NOT copy this code into your working application. This version has two problems. First, there are a lot of situations where it doesn't work but `Add()` should work. There are several examples of valid `Add()` methods that take dissimilar parameters: `int` and `double`, `DateTime` and `TimeSpan`, etc. Those won't work with this method. Let's fix that. You must add two more generic parameters to the method. Then, you can specify different operands on the left and the right side of the operation. While at it, I replaced some of the local variable names with `var` declarations. This obscures the type information, but it does help make the logic of the method a little more clear.

```
// A little better.
public static TResult AddExpression<T1, T2, TResult>
    (T1 left, T2 right)
{
    var leftOperand = Expression.Parameter(typeof(T1),
```

```
        "left");
    var rightOperand = Expression.Parameter(typeof(T2),
        "right");
    var body = Expression.Add(leftOperand, rightOperand);
    var adder = Expression.Lambda<Func<T1, T2, TResult>>(
        body, leftOperand, rightOperand);
    return adder.Compile()(left, right);
}
```

This method looks very similar to the previous version; it just enables you to call it with different types for the left and the right operand. The only downside is that you need to specify all three parameter types whenever you call this version:

```
int sum2 = AddExpression<int, int, int>(5, 7);
```

However, because you specify all three parameters, expressions with dissimilar parameters work:

```
DateTime nextWeek = AddExpression<DateTime, TimeSpan,
    DateTime>(
    DateTime.Now, TimeSpan.FromDays(7));
```

It's time to address the other nagging issue. The code, as I have shown so far, compiles the expression into a delegate every time the `AddExpression()` method is called. That's quite inefficient, especially if you end up executing the same expression repeatedly. Compiling the expression is expensive, so you should cache the compiled delegate for future invocations. Here's a first pass at that class:

```
// working with many limitations
public static class BinaryOperator<T1, T2, TResult>
{
    static Func<T1, T2, TResult> compiledExpression;

    public static TResult Add(T1 left, T2 right)
    {
        if (compiledExpression == null)
            createFunc();

        return compiledExpression(left, right);
    }

    private static void createFunc()
```

```
    {
        var leftOperand = Expression.Parameter(typeof(T1),
            "left");
        var rightOperand = Expression.Parameter(typeof(T2),
            "right");
        var body = Expression.Add(leftOperand, rightOperand);
        var adder = Expression.Lambda<Func<T1, T2, TResult>>(
            body, leftOperand, rightOperand);
        compiledExpression = adder.Compile();
    }
}
```

Let's discuss how to decide between using expressions and dynamic. That decision depends on the situation. The Expression version uses a slightly simpler set of runtime computations. That might make it faster in many circumstances. However, expressions are a little less dynamic than dynamic invocation. Remember that with dynamic invocation, you could add many different types successfully: `int` and `double`, `short` and `float`, whatever. As long as it was legal in C# code, it was legal in the compiled version. You could even add a string and number. If you try those same scenarios using the expression version, any of those legal dynamic versions will throw an `InvalidOperationException`. Even though there are conversion operations that work, the Expressions you've built don't build those conversions into the lambda expression. Dynamic invocation does more work and therefore supports more different types of operations. For instance, suppose you want to update the `AddExpression` to add different types and perform the proper conversions. Well, you just have to update the code that builds the expression to include the conversions from the parameter types to the result type yourself. Here's what it looks like:

```
// A fix for one problem causes another
public static TResult AddExpressionWithConversion
    <T1, T2, TResult>(T1 left, T2 right)
{
    var leftOperand = Expression.Parameter(typeof(T1),
        "left");
    Expression convertedLeft = leftOperand;
    if (typeof(T1) != typeof(TResult))
    {
        convertedLeft = Expression.Convert(leftOperand,
            typeof(TResult));
    }
    var rightOperand = Expression.Parameter(typeof(T2),
```

```
            "right");
        Expression convertedRight = rightOperand;
        if (typeof(T2) != typeof(TResult))
        {
            convertedRight = Expression.Convert(rightOperand,
            typeof(TResult));
        }
        var body = Expression.Add(convertedLeft, convertedRight);
        var adder = Expression.Lambda<Func<T1, T2, TResult>>(
            body, leftOperand, rightOperand);
        return adder.Compile()(left, right);
    }
```

That will fix all the problems with any addition that needs a conversion, like adding `doubles` and `ints`, or adding a `double` to `string` with the result being a string. However, it breaks valid usages where the parameters should not be the same as the result. In particular, this version would not work with the example above adding a `TimeSpan` to a `DateTime`. With a lot more code, you could solve this. However, at that point, you've pretty much reimplemented the code that handles dynamic dispatch for C# (see Item 43). Instead of all that work, just use dynamic.

You should use the expression version for those times when the operands and the result are the same. That gives you generic type parameter inference and fewer permutations when the code fails at runtime. Here's the version I would recommend to use Expression for implementing runtime dispatch:

```
public static class BinaryOperators<T>
{
    static Func<T, T, T> compiledExpression;

    public static T Add(T left, T right)
    {
        if (compiledExpression == null)
            createFunc();

        return compiledExpression(left, right);
    }

    private static void createFunc()
    {
        var leftOperand = Expression.Parameter(typeof(T),
            "left");
        var rightOperand = Expression.Parameter(typeof(T),
```

```
            "right");
        var body = Expression.Add(leftOperand, rightOperand);
        var adder = Expression.Lambda<Func<T, T, T>>(
            body, leftOperand, rightOperand);
        compiledExpression = adder.Compile();
    }
}
```

You still need to specify the one type parameter when you call Add. Doing so does give you the advantage of being able to leverage the compiler to create any conversions at the callsite. The compiler can promote `ints` to `doubles` and so on.

There are also performance costs with using dynamic and with building expressions at runtime. Just like any dynamic type system, your program has more work to do at runtime because the compiler did not perform any of its usual type checking. The compiler must generate instructions to perform all those checks at runtime. I don't want to overstate this, because the C# compiler does produce efficient code for doing the runtime checking. In most cases, using dynamic will be faster than writing your own code to use reflection and produce your own version of late binding. However, the amount of runtime work is nonzero; the time it takes is also nonzero. If you can solve a problem using static typing, it will undoubtedly be more efficient than using dynamic types.

When you control all the types involved, and you can create an interface instead of using dynamic programming, that's the better solution. You can define the interface, program against the interface, and implement the interface in all your types that should exhibit the behavior defined by the interface. The C# type system will make it harder to introduce type errors in your code, and the compiler will produce more efficient code because it can assume that certain classes of errors are not possible.

In many cases, you can create the generic API using lambdas and force callers to define the code you would execute in the dynamic algorithm.

The next choice would be using expressions. That's the right choice if you have a relatively small number of permutations for different types, and a small number of possible conversions. You can control what expressions get created

and therefore how much work happens at runtime.

When you use dynamic, the underlying dynamic infrastructure will work to make any possible legal construct work, no matter how expensive the work is at runtime.

However, for the `Add()` method I demonstrated at the beginning of this item, that's not possible. `Add()` should work on a number of types that are already defined in the .NET class library. You can't go back and add an `IAdd` interface to those types. You also can't guarantee that all third-party libraries you want to work with will conform to some new interface. The best way to build methods based on the presence of a particular member is to write a dynamic method that defers that choice to the runtime. The dynamic implementation will find a proper implementation, use it, and cache for better performance. It's more expensive than a purely statically typed solution, and it's much simpler than parsing expression trees.

Item 44: Use Dynamic to Leverage the Runtime Type of Generic Type Parameters

`System.Linq.Enumerable.Cast<T>` coerces every object in a sequence to the target type of T. It's part of the framework so that LINQ queries can be used with sequences of `IEnumerable` (as opposed to `IEnumerable<T>`). `Cast<T>` is a generic method, with no constraints on T. That limits the types of conversions available to it. If you use `Cast<T>` without understanding its limitations, you'll find yourself thinking it doesn't work. In reality, it's working exactly as it should, just not the way you expect. Let's examine its inner workings and limitations. Then, it will be easy to create a different version that does what you expect.

The root of the problem lies with the fact that `Cast<T>` is compiled into MSIL without any knowledge of T beyond the fact that T must be a managed type that derives from `System.Object`. Therefore, it does its work only using the functionality defined in `System.Object`. Examine this class:

```
public class MyType
{
```

```
    public String StringMember { get; set; }

    public static implicit operator String(MyType aString)
        => aString.StringMember;

    public static implicit operator MyType(String aString)
        => new MyType { StringMember = aString };
}
```

See Item 11 for why conversion operators are bad; however, a user-defined conversion operator is key to this issue. Consider this code (assume that `GetSomeStrings()` returns a sequence of strings):

```
var answer1 = GetSomeStrings().Cast<MyType>();
try
{
    foreach (var v in answer1)
        WriteLine(v);
}
catch (InvalidCastException)
{
    WriteLine("Cast Failed!");
}
```

Before starting this item, you may have expected that `GetSomeStrings().Cast<MyType>()` would correctly convert each string to a `MyType` using the implicit conversion operator defined in `MyType`. Now you know it doesn't; it throws an `InvalidCastException`.

The above code is equivalent to this construct, using a query expression:

```
var answer2 = from MyType v in GetSomeStrings()
                select v;
try
{
    foreach (var v in answer2)
        WriteLine(v);
}
catch (InvalidCastException)
{
    WriteLine("Cast failed again");
}
```

The type declaration on the range variable is converted to a call to

`Cast<MyType>` by the compiler. Again, it throws an `InvalidCastException`. Here's one way to restructure the code so that it works:

```
var answer3 = from v in GetSomeStrings()
                select (MyType)v;
foreach (var v in answer3)
    WriteLine(v);
```

What's the difference? The two versions that don't work use `Cast<T>()`, and the version that works includes the cast in the lambda used as the argument to `Select()`. `Cast<T>` cannot access any user-defined conversions on the runtime type of its argument. The only conversions it can make are reference conversions and boxing conversions. A reference conversion succeeds when the `is` operator succeeds (see Item 3). A boxing conversion converts a value type to a reference type and vice versa (see Item 9 in Effective C#, Third Edition). `Cast<T>` cannot access any user-defined conversions because it can only assume that `T` contains the members defined in `System.Object`. `System.Object` does not contain any user-defined conversions, so those are not eligible. The version using `Select<T>` succeeds because the lambda used by `Select()` takes an input parameter of `string`. That means the conversion operation defined on `MyType`.

As I've pointed out before, I usually view conversion operators as a code smell. On occasion, they are useful, but often they'll cause more problems than they are worth. Here, without the conversion operators, no developer would be tempted to write the example code that didn't work.

Of course, if I'm recommending not using conversion operators, I should offer an alternative. `MyType` already contains a read/write property to store the `string` property, so you can just remove the conversion operators and write either of these constructs:

```
var answer4 = GetSomeStrings().
    Select(n => new MyType { StringMember = n });
var answer5 = from v in GetSomeStrings()
                select new MyType { StringMember = v };
```

Also, if you needed to, you could create a different constructor for `MyType`. Of course, that is just working around a limitation in `Cast<T>()`. Now that you understand why those limitations exist, it's time to write a different method that

gets around those limitations. The trick is to write the generic method in such a way that it leverages runtime information to perform any conversions.

You could write pages and pages of reflection-based code to see what conversions are available, perform any of those conversions, and return the proper type. You could do that, but it's a waste. Instead, C# 4.0 dynamic can do all the heavy lifting. You're left with a simple `Convert<T>` that does what you expect:

```
public static IEnumerable<TResult> Convert<TResult>(
    this System.Collections.IEnumerable sequence)
{
    foreach (object item in sequence)
    {
        dynamic coercion = (dynamic)item;
        yield return (TResult)coercion;
    }
}
```

Now, as long as there is a conversion (either implicit or explicit) from the source type to the target type, the conversion works. There are still casts involved, so the possibility for runtime failure exists. That's just part of the game when you are coercing the type system. `Convert<T>` does work in more situations than `Cast<T>()`, but it also does more work. As developers, we should be more concerned about what code our users need to create than we are about our own code. `Convert<T>` passes this test:

```
var convertedSequence = GetSomeStrings().Convert<MyType>();
```

`Cast<T>`, like all generic methods, compiles with only limited knowledge of its type parameters. That can lead to generic methods not working the way you'd expect. The root cause is almost always that the generic method could not be made aware of particular functionality in the type representing the type parameters. When that happens, a little application of dynamic can enable runtime reflection to make matters right.

Item 45: Use DynamicObject or IDynamicMetaObjectProvider for Data-Driven Dynamic Types

One great advantage of dynamic programming is the ability to build types whose public interface changes at runtime, based on how you use them. C# provides that ability through `dynamic`, the `System.Dynamic.DynamicObject` base class, and the `System.Dynamic.IDynamicMetaObjectProvider` interface. Using these tools, you can create your own types that have dynamic capabilities.

The simplest way to create a type with dynamic capabilities is to derive from `System.Dynamic.DynamicObject`. That type implements the `IDynamicMetaObjectProvider` interface using a private nested class. This private nested class does the hard work of parsing expressions and forwarding those to one of a number of virtual methods in the `DynamicObject` class. That makes it a relatively simple exercise to create a dynamic class, if you can derive from `DynamicObject`.

For example, consider a class that implemented a dynamic property bag. This example is similar to Razor property bags, ExpandoObject, and the Clay and Gemini projects. See those examples for production-hardened implementations. When you first create the `DynamicPropertyBag`, it doesn't have any items, and therefore it doesn't have any properties. When you try to retrieve any property, it will throw an exception. You can add any property to the bag by calling the setter on that property. After adding the property, you can call the getter and access any of the properties.

```
dynamic dynamicProperties = new DynamicPropertyBag();

try
{
    Console.WriteLine(dynamicProperties.Marker);
}
catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException)
{
    Console.WriteLine("There are no properties");
}

dynamicProperties.Date = DateTime.Now;
dynamicProperties.Name = "Bill Wagner";
dynamicProperties.Title = "Effective C#";
dynamicProperties.Content = "Building a dynamic dictionary";
```

The implementation of the dynamic property bag requires overriding the

TrySetMember and TryGetMember methods in the DynamicObject base class.

```
class DynamicPropertyBag : DynamicObject
{
    private Dictionary<string, object> storage =
        new Dictionary<string, object>();

    public override bool TryGetMember(GetMemberBinder binder,
        out object result)
    {
        if (storage.ContainsKey(binder.Name))
        {
            result = storage[binder.Name];
            return true;
        }
        result = null;
        return false;
    }

    public override bool TrySetMember(SetMemberBinder binder,
        object value)
    {
        string key = binder.Name;
        if (storage.ContainsKey(key))
            storage[key] = value;
        else
            storage.Add(key, value);
        return true;
    }

    public override string ToString()
    {
        StringWriter message = new StringWriter();
        foreach (var item in storage)
            message.WriteLine($"{item.Key}:\t{item.Value}");
        return message.ToString();
    }
}
```

The dynamic property bag contains a dictionary that stores the property names and their values. The work is done in `TryGetMember` and `TrySetMember`.

`TryGetMember` examines the requested name (`binder.Name`), and if that property has been stored in the Dictionary, TryGetMember will return its value. If the value has not been stored, the dynamic call fails.

`TrySetMember` accomplishes its work in a similar fashion. It examines the requested name (`binder.Name`) and either updates or creates an entry for that item in the internal `Dictionary`. Because you can create any property, the `TrySetMember` method always returns true, indicating that the dynamic call succeeded.

`DynamicObject` contains similar methods to handle dynamic invocation of indexers, methods, constructors, and unary and binary operators. You can override any of those members to create your own dynamic members. In all cases, you must examine the `Binder` object to see what member was requested and perform whatever operation is needed. Where there are return values, you'll need to set those (in the specified `out` parameter) and return whether or not your overload handled the member.

If you're going to create a type that enables dynamic behavior, using `DynamicObject` as the base class is the easiest way to do it. Of course, a dynamic property bag is okay, but let's look at one more sample that shows when a dynamic type is more useful.

LINQ to XML made some great improvements to working with XML, but it still left something to be desired. Consider this snippet of XML that contains some information about our solar system:

```
<Planets>
  <Planet>
    <Name>Mercury</Name>
  </Planet>
  <Planet>
    <Name>Venus</Name>
  </Planet>
  <Planet>
    <Name>Earth</Name>
    <Moons>
      <Moon>Moon</Moon>
    </Moons>
  </Planet>
  <Planet>
    <Name>Mars</Name>
    <Moons>
      <Moon>Phobos</Moon>
      <Moon>Deimos</Moon>
    </Moons>
```

```
  </Planet>
  <!-- other data elided -->
</Planets>
```

To get the first planet, you would write something like this:

```
// Create an XElement document containing
// solar system data:
var xml = createXML();

var firstPlanet = xml.Element("Planet");
```

That's not too bad, but the farther you get into the file, the more complicated the code gets. Getting Earth (the third planet) looks like this:

```
var earth = xml.Elements("Planet").Skip(2).First();
```

Getting the name of the third planet is more code:

```
var earthName = xml.Elements("Planet").Skip(2).
    First().Element("Name");
```

Once you're getting moons, it's really long code:

```
var moon = xml.Elements("Planet").Skip(2).First().
        Elements("Moons").First().Element("Moon");
```

Furthermore, the above code only works if the XML contains the nodes you're seeking. If there was a problem in the XML file, and some of the nodes were missing, the above code would throw an exception. Adding the code to handle missing nodes adds quite a bit more code, just to handle potential errors. At that point, it's harder to discern the original intent.

Instead, suppose you had a data-driven type that could give you dot notation on XML elements, using the element name. Finding the first planet could be as simple as:

```
// Create an XElement document containing
// solar system data:
var xml = createXML();

Console.WriteLine(xml);
```

```
dynamic dynamicXML = new DynamicXElement(xml);

// old way:
var firstPlanet = xml.Element("Planet");
Console.WriteLine(firstPlanet);
// new way:
dynamic test2 = dynamicXML.Planet; // returns the first planet.
Console.WriteLine(test2);
```

Getting the third planet would be simply using an indexer:

```
// gets the third planet (Earth)
dynamic test3 = dynamicXML["Planet", 2];
```

Reaching the moons becomes two chained indexers:

```
dynamic earthMoon = dynamicXML["Planet", 2]["Moons", 0].Moon;
```

Finally, because it's dynamic, you can define the semantics so any missing node returns an empty element. That means all of these would return empty dynamic XElement nodes:

```
dynamic test6 = dynamicXML["Planet", 2]
    ["Moons", 3].Moon; // earth doesn't have 4 moons
dynamic fail = dynamicXML.NotAppearingInThisFile;
dynamic fail2 = dynamicXML.Not.Appearing.In.This.File;
```

Because missing elements will return a missing dynamic element, you can continue to dereference it and know that if any element in the composed XML navigation is missing, the final result will be a missing element. Building this is another class derived from `DynamicObject`. You have to override `TryGetMember`, and `TryGetIndex` to return dynamic elements with the appropriate nodes.

```
public class DynamicXElement : DynamicObject
{
    private readonly XElement xmlSource;

    public DynamicXElement(XElement source)
    {
        xmlSource = source;
    }

    public override bool TryGetMember(GetMemberBinder binder,
```

```
        out object result)
    {
        result = new DynamicXElement(null);
        if (binder.Name == "Value")
        {
            result = (xmlSource != null) ? xmlSource.Value : "'
            return true;
        }
        if (xmlSource != null)
            result = new DynamicXElement(xmlSource
                .Element(XName.Get(binder.Name)));
        return true;
    }

    public override bool TryGetIndex(GetIndexBinder binder,
        object[] indexes, out object result)
    {
        result = null;
        // This only supports [string, int] indexers
        if (indexes.Length != 2)
            return false;
        if (!(indexes[0] is string))
            return false;
        if (!(indexes[1] is int))
            return false;

        var allNodes = xmlSource.Elements(indexes[0].ToString()
        int index = (int)indexes[1];
        if (index < allNodes.Count())
            result = new DynamicXElement(allNodes
                .ElementAt(index));
        else
            result = new DynamicXElement(null);
        return true;
    }

    public override string ToString() =>
        xmlSource?.ToString() ?? string.Empty;
}
```

Most of the code uses similar concepts to the code you have seen earlier in this item. The `TryGetIndex` method is new. It must implement the dynamic behavior when client code invokes an indexer to retrieve an `XElement`.

Using `DynamicObject` makes it much easier to implement a type that behaves dynamically. `DynamicObject` hides much of the complexity of creating

dynamic types. It has quite a bit of implementation to handle dynamic dispatch for you. Also, sometimes you will want to create a dynamic type and you won't be able to use `DynamicObject` because you need a different base class. For that reason, I'm going to show you how to create the dynamic dictionary by implementing `IDynamicMetaObjectProvider` yourself, instead of relying on `DynamicObject` to do the heavy lifting for you.

Implementing `IDynamicMetaObjectProvider` means implementing one method: `GetMetaObject`. Here's a second version of `DynamicDictionary` that implements `IDynamicMetaObjectProvider`, instead of deriving from `DynamicObject`:

```
class DynamicDictionary2 : IDynamicMetaObjectProvider
{
    DynamicMetaObject IDynamicMetaObjectProvider.GetMetaObject
        System.Linq.Expressions.Expression parameter)
    {
        return new DynamicDictionaryMetaObject(parameter, this)
    }

    private Dictionary<string, object> storage =
        new Dictionary<string, object>();

    public object SetDictionaryEntry(string key, object value)
    {
        if (storage.ContainsKey(key))
            storage[key] = value;
        else
            storage.Add(key, value);
        return value;
    }

    public object GetDictionaryEntry(string key)
    {
        object result = null;
        if (storage.ContainsKey(key))
        {
            result = storage[key];
        }
        return result;
    }

    public override string ToString()
    {
        StringWriter message = new StringWriter();
```

```
        foreach (var item in storage)
            message.WriteLine($"{item.Key}:\t{item.Value}");
        return message.ToString();
    }
}
```

`GetMetaObject()` returns a new `DynamicDictionaryMetaObject` whenever it is called. Here's where the first complexity enters the picture. `GetMetaObject()` is called every time any member of the `DynamicDictionary` is invoked. If you call the same member ten times, `GetMetaObject()` gets called ten times. Even if methods are statically defined in `DynamicDictionary2`, `GetMetaObject()` will be called and can intercept those methods to invoke possible dynamic behavior. Remember that dynamic objects are statically typed as dynamic, and therefore have no compile-time behavior defined. Every member access is dynamically dispatched.

The `DynamicMetaObject` is responsible for building an Expression Tree that executes whatever code is necessary to handle the dynamic invocation. Its constructor takes the expression and the dynamic object as parameters. After being constructed, one of the Bind methods will be called. Its responsibility is to construct a `DynamicMetaObject` that contains the expression to execute the dynamic invocation. Let's walk through the two `Bind` methods necessary to implement the `DynamicDictionary`: `BindSetMember` and `BindGetMember`.

`BindSetMember` constructs an expression tree that will call `DynamicDictionary2.SetDictionaryEntry()` to set a value in the dictionary. Here's its implementation:

```
public override DynamicMetaObject BindSetMember(
    SetMemberBinder binder,
    DynamicMetaObject value)
{
    // Method to call in the containing class:
    string methodName = "SetDictionaryEntry";

    // setup the binding restrictions.
    BindingRestrictions restrictions =
        BindingRestrictions.GetTypeRestriction(
        Expression, LimitType);

    // setup the parameters:
    Expression[] args = new Expression[2];
```

```
    // First parameter is the name of the property to Set
    args[0] = Expression.Constant(binder.Name);
    // Second parameter is the value
    args[1] = Expression.Convert(value.Expression,
        typeof(object));

    // Setup the 'this' reference
    Expression self = Expression.Convert(Expression, LimitType)

    // Setup the method call expression
    Expression methodCall = Expression.Call(self,
            typeof(DynamicDictionary2).GetMethod(methodName),
            args);

    // Create a meta object to invoke Set later:
    DynamicMetaObject setDictionaryEntry = new DynamicMetaObje
        methodCall,
        restrictions);
    // return that dynamic object
    return setDictionaryEntry;
}
```

Metaprogramming quickly gets confusing, so let's walk through this slowly. The first line sets the name of the method called in the `DynamicDictionary`, "SetDictionaryEntry". Notice that `SetDictionary` returns the right-hand side of the property assignment. That's important because this construct must work:

```
DateTime current = propertyBag2.Date = DateTime.Now;
```

Without setting the return value correctly, that construct won't work.

Next, this method initializes a set of `BindingRestrictions`. Most of the time, you'll use restrictions like this one, restrictions given in the source expression and for the type used as the target of the dynamic invocation.

The rest of the method constructs the method call expression that will invoke `SetDictionaryEntry()` with the property name and the value used. The property name is a constant expression, but the value is a `Conversion` expression that will be evaluated lazily. Remember that the right-hand side of the setter may be a method call or expression with side effects. Those must be evaluated at the proper time. Otherwise, setting properties using the return value of methods won't work:

```
propertyBag2.MagicNumber = GetMagicNumber();
```

Of course, to implement the dictionary, you have to implement `BindGetMember` as well. `BindGetMember` works almost exactly the same way. It constructs an expression to retrieve the value of a property from the dictionary.

```
public override DynamicMetaObject BindGetMember(
    GetMemberBinder binder)
{
    // Method call in the containing class:
    string methodName = "GetDictionaryEntry";

    // One parameter
    Expression[] parameters = new Expression[]
    {
    Expression.Constant(binder.Name)
    };

    DynamicMetaObject getDictionaryEntry = new DynamicMetaObje
        Expression.Call(
            Expression.Convert(Expression, LimitType),
            typeof(DynamicDictionary2).GetMethod(methodName),
            parameters),
        BindingRestrictions.GetTypeRestriction(Expression,
            LimitType));
    return getDictionaryEntry;
}
```

Before you go off and think this isn't that hard, let me leave you with some thoughts from the experience of writing this code. This is about as simple as a dynamic object can get. You have two APIs: property `get`, property `set`. The semantics are very easy to implement. Even with this very simple behavior, it was rather difficult to get right. Expression trees are hard to debug More sophisticated dynamic types would have much more code. That would mean much more difficulty getting the expressions correct.

Furthermore, keep in mind one of the opening remarks I made on this section: Every invocation on your dynamic object will create a new `DynamicMetaObject` and invoke one of the Bind members. You'll need to write these methods with an eye toward efficiency and performance. They will be called a lot, and they have much work to do.

Implementing dynamic behavior can be a great way to approach some of your

programming challenges. When you look at creating dynamic types, your first choice should be to derive from `System.Dynamic.DynamicObject`. On those occasions where you must use a different base class, you can implement `IDynamicMetaObjectProvider` yourself, but remember that this is a complicated problem to take on. Furthermore, any dynamic types involve some performance costs, and implementing them yourself may make those costs greater.

Item 46: Understand How to Make Use of the Expression API

.NET has had APIs that enable you to reflect on types or to create code at runtime. The ability to examine code or create code at runtime is very powerful. There are many different problems that are best solved by inspecting code or dynamically generating code. The problem with these APIs is that they are very low level and quite difficult to work with. As developers, we crave an easier way to dynamically solve problems.

Now that C# has added LINQ and dynamic support, you have a better way than the classic Reflection APIs: expressions and expression trees. Expressions look like code. And, in many uses, expressions do compile down to delegates. However, you can ask for expressions in an Expression format. When you do that, you have an object that represents the code you want to execute. You can examine that expression, much like you can examine a class using the Reflection APIs. In the other direction, you can build an expression to create code at runtime. Once you create the expression tree you can compile and execute the expression. The possibilities are endless. After all, you are creating code at runtime. I'll describe two common tasks where expressions can make your life much easier.

The first solves a common problem in communication frameworks. The typical workflow for using WCF, remoting, or Web services is to use some code generation tool to generate a client-side proxy for a particular service. It works, but it is a somewhat heavyweight solution. You'll generate hundreds of lines of code. You'll need to update the proxy whenever the server gets a new method, or changes parameter lists. Instead, suppose you could write

something like this:

```
var client = new ClientProxy<IService>();
var result = client.CallInterface<string>(
    srver => srver.DoWork(172));
```

Here, the `ClientProxy<T>` knows how to put each argument and method call on the wire. However, it doesn't know anything about the service you're actually accessing. Rather than relying on some out of band code generator, it will use expression trees and generics to figure out what method you called, and what parameters you used.

The `CallInterface()` method takes one parameter, which is an `Expression<Func<T, TResult>>`. The input parameter (of type T) represents an object that implements `IService`. `TResult`, of course, is whatever the particular method returns. The parameter is an expression, and you don't even need an instance of an object that implements `IService` to write this code. The core algorithm is in the `CallInterface()` method.

```
public TResult CallInterface<TResult>(Expression<
    Func<T, TResult>> op)
{
    var exp = op.Body as MethodCallExpression;
    var methodName = exp.Method.Name;
    var methodInfo = exp.Method;
    var allParameters = from element in exp.Arguments
                        select processArgument(element);
    Console.WriteLine($"Calling {methodName}");

    foreach (var parm in allParameters)
        Console.WriteLine(@$"\tParameter type = {parm.ParmType}
            Value = {parm.ParmValue}");
    return default(TResult);
}

private (Type ParmType, object ParmValue) processArgument(
    Expression element)
{
    object argument = default(object);
    LambdaExpression expression = Expression.Lambda(
        Expression.Convert(element, element.Type));
    Type parmType = expression.ReturnType;
    argument = expression.Compile().DynamicInvoke();
    return (parmType, argument);
```

```
}
```

Starting from the beginning of `CallInterface`, the first thing this code does is look at the body of the expression tree. That's the part on the right side of the lambda operator. Look back at the example where I used `CallInterface()`. That example called it with `srver.DoWork(172)`. It is a `MethodCallExpression`, and that `MethodCallExpression` contains all the information you need to understand all the parameters and the method name invoked. The method name is pretty simple: It's stored in the Name property of the Method property. In this example, that would be 'DoWork'. The LINQ query processes any and all parameters to this method. The interesting work in is `processArgument`.

`processArgument` evaluates each parameter expression. In the example above, there is only one argument, and it happens to be a constant, the value 172. However, that's not very robust, so this code takes a different strategy. It's not robust, because any of the parameters could be method calls, property or indexer accessors, or even field accessors. Any of the method calls could also contain parameters of any of those types. Instead of trying to parse everything, this method does that hard work by leveraging the `LambdaExpression` type and evaluating each parameter expression. Every parameter expression, even the `ConstantExpression`, could be expressed as the return value from a lambda expression. `ProcessArgument()` converts the parameter to a `LambdaExpression`. In the case of the constant expression, it would convert to a lambda that is the equivalent of `() => 172`. This method converts each parameter to a lambda expression because a lambda expression can be compiled into a delegate and that delegate can be invoked. In the case of the parameter expression, it creates a delegate that returns the constant value 172. More complicated expressions would create more complicated lambda expressions.

Once the lambda expression has been created, you can retrieve the type of the parameter from the lambda. Notice that this method does not perform any processing on the parameters. The code to evaluate the parameters in the lambda expression would be executed when the lambda expression is invoked. The beauty of this is that it could even contain other calls to `CallInterface()`. Constructs like this just work:

```
client.CallInterface(srver => srver.DoWork(
    client.CallInterface(srv => srv.GetANumber())));
```

This technique shows you how you can use expression trees to determine at runtime what code the user wishes to execute. It's hard to show in a book, but because `ClientProxy<T>` is a generic class that uses the service interface as a type parameter, the `CallInterface` method is strongly typed. The method call in the lambda expression must be a member method defined on the server.

The first example showed you how to parse expressions to convert code (or at least expressions that define code) into data elements you can use to implement runtime algorithms. The second example shows the opposite direction: Sometimes you want to generate code at runtime. One common problem in large systems is to create an object of some destination type from some related source type. For example, your large enterprise may contain systems from different vendors each of which has a different type defined for a contact (among other types). Sure, you could type methods by hand, but that's tedious. It would be much better to create some kind of type that "figures out" the obvious implementation. You'd like to just write this code:

```
var converter = new Converter<SourceContact,
    DestinationContact>();
DestinationContact dest2 = converter.ConvertFrom(source);
```

You'd expect the converter to copy every property from the source to the destination where the properties have the same name and the source object has a public `get` accessor and the destination type has a public `set` accessor. This kind of runtime code generation can be best handled by creating an expression, and then compiling and executing it. You want to generate code that does something like this:

```
// Not legal C#, explanation only
TDest ConvertFromImaginary(TSource source)
{
    TDest destination = new TDest();
    foreach (var prop in sharedProperties)
        destination.prop = source.prop;
    return destination;
}
```

You need to create an expression that creates code that executes the pseudo

code written above. Here's the full method to create that expression and compile it to a function. Immediately following the listing, I'll explain all the parts of this method in detail. You'll see that while it's a bit thorny at first, it's nothing you can't handle.

```
private void createConverterIfNeeded()
{
    if (converter == null)
    {
        var source = Expression.Parameter(typeof(TSource), "sou
        var dest = Expression.Variable(typeof(TDest), "dest");

        var assignments = from srcProp in
            typeof(TSource).GetProperties(
            BindingFlags.Public | BindingFlags.Instance)
               where srcProp.CanRead
               let destProp = typeof(TDest).GetProperty(
                   srcProp.Name,
                   BindingFlags.Public | BindingFlags.Instance)
               where (destProp != null) && (destProp.CanWrite)
               select Expression.Assign(
                   Expression.Property(dest, destProp),
                   Expression.Property(source, srcProp));

        // put together the body:
        var body = new List<Expression>();
        body.Add(Expression.Assign(dest,
            Expression.New(typeof(TDest))));
        body.AddRange(assignments);
        body.Add(dest);

        var expr =
            Expression.Lambda<Func<TSource, TDest>>(
                Expression.Block(
                new[] { dest }, // expression parameters
                body.ToArray() // body
                ),
                source  // lambda expression
            );

        var func = expr.Compile();
        converter = func;
    }
}
```

This method creates code that mimics the pseudo code shown before. First,

you declare the parameter:

```
var source = Expression.Parameter(typeof(TSource), "source");
```

Then, you have to declare a local variable to hold the destination:

```
var dest = Expression.Variable(typeof(TDest), "dest");
```

The bulk of the method is the code that assigns properties from the source object to the destination object. I wrote this code as a LINQ query. The source sequence of the LINQ query is the set of all public instance properties in the source object where there is a `get` accessor:

```
from srcProp in typeof(TSource).GetProperties(
                BindingFlags.Public | BindingFlags.Instance)
                where srcProp.CanRead
```

The `let` declares a local variable that holds the property of the same name in the destination type. It may be null, if the destination type does not have a property of the correct type:

```
let destProp = typeof(TDest).GetProperty(
               srcProp.Name,
               BindingFlags.Public | BindingFlags.Instance)
        where (destProp != null) && (destProp.CanWrite)
```

The projection of the query is a sequence of assignment statements that assigns the property of the destination object to the value of the same property name in the source object:

```
select Expression.Assign(
       Expression.Property(dest, destProp),
       Expression.Property(source, srcProp));
```

The rest of the method builds the body of the lambda expression. The Block() method of the Expression class needs all the statements in an array of Expression. The next step is to create a List<Expression> where you can add all the statements. The list can be easily converted to an array.

```
var body = new List<Expression>();
body.Add(Expression.Assign(dest,
    Expression.New(typeof(TDest))));
```

```
body.AddRange(assignments);
body.Add(dest);
```

Finally, it's time to build a lambda that returns the destination object and contains all the statements built so far:

```
var expr =
    Expression.Lambda<Func<TSource, TDest>>(
        Expression.Block(
        new[] { dest }, // expression parameters
        body.ToArray() // body
        ),
        source  // lambda expression
    );
```

That's all the code you need. Time to compile it and turn it into a delegate that you can call:

```
var func = expr.Compile();
converter = func;
```

That is complicated, and it's not the easiest to write. You'll often find compiler-like errors at runtime until you get the expressions built correctly. It's also clearly not the best way to approach simple problems. But even so, the Expression APIs are much simpler than their predecessors in the Reflection APIs that emit IL. That's when you should use the Expression APIs: When you think you want to use reflection, try to solve the problem using the Expression APIs instead.

The Expression APIs can be used in two very different ways: You can create methods that take expressions as parameters, which enables you to parse those expressions and create code based on the concepts behind the expressions that were called. Also, the Expression APIs enable you to create code at runtime. You can create classes that write code, and then execute the code they've written. It's a very powerful way to solve some of the more difficult general purpose problems you'll encounter.

Item 47: Minimize Dynamic Objects in Public APIs

Dynamic objects just don't behave that well in a statically typed system. The

type system sees them as though they were instances of `System.Object`. But they are special instances. You can ask them to do work above and beyond what's defined in `System.Object`. The compiler generates code that tries to find and execute whatever members you try to access.

But dynamic objects are pushy. Everything they touch becomes dynamic. Perform an operation where any one of the parameters is dynamic, and the result is dynamic. Return a dynamic object from a method, and everywhere that dynamic is used becomes a dynamic object. It's like watching bread mold grow in a petri dish. Pretty soon, everything is dynamic, and there's no type safety left anywhere.

Biologists grow cultures in petri dishes, restricting where they can grow. You need to do the same with dynamic: Do the work with dynamic objects in an isolated environment and return objects that are statically typed as something other than dynamic. Otherwise, dynamic becomes a bad influence, and slowly, everything involved in your application will be dynamic.

This is not to imply that dynamic is universally bad. Other items in this chapter have shown you some of the techniques where dynamic programming is an excellent solution. However, dynamic typing and static typing are very different, with different practices, different idioms, and different strategies. Mixing the two without regard will lead to numerous errors and inefficiencies. C# is a statically typed language, enabling dynamic typing in some areas. Therefore, if you're using C#, you should spend most of your time using static typing and minimize the scope of the dynamic features. If you want to write programs that are dynamic through and through, you should consider a language that is dynamic rather than a static typed language.

If you're going to use dynamic features in your program, try to keep them out of the public interface to your types. That way, you can use dynamic typing in a single object (or type) petri dish without having them escape into the rest of your program, or into all the code developed by developers who use your objects.

One scenario where you will use dynamic typing is to interact with objects created in dynamic environments, such as IronPython. When your design makes use of dynamic objects created using dynamic languages, you should wrap

them in C# objects that enable the rest of the C# world to blissfully ignore the fact that dynamic typing is even happening.

You may want to pick a different solution for those situations where you use dynamic to produce duck typing. Look at the usages of the duck typing sample from Item 43. In every case, the result of the calculation was dynamic. That might not look too bad. But, the compiler is doing quite a bit of work to make this work. These two lines of code (see Item 41):

```
dynamic answer = Add(5, 5);
Console.WriteLine(answer);
```

turn into this to handle dynamic objects:

```
// Compiler generated, not legal user C# code
object answer = Add(5, 5);
if (<Main>o__SiteContainer0.<>p__Site1 == null)
{
    <Main>o__SiteContainer0.<>p__Site1 =
        CallSite<Action<CallSite, Type, object>>.Create(
        new CSharpInvokeMemberBinder(
        CSharpCallFlags.None, "WriteLine",
        typeof(Program), null, new CSharpArgumentInfo[]
        {
            new CSharpArgumentInfo(
            CSharpArgumentInfoFlags.IsStaticType |
            CSharpArgumentInfoFlags.UseCompileTimeType,
            null),
            new CSharpArgumentInfo(
                CSharpArgumentInfoFlags.None,
            null)
        }));
}
<Main>o__SiteContainer0.<>p__Site1.Target.Invoke(
   <Main>o__SiteContainer0.<>p__Site1,
   typeof(Console), answer);
```

Dynamic is not free. There's quite a bit of code generated by the compiler to make dynamic invocation work in C#. Worse, this code will be repeated everywhere that you invoke the dynamic `Add()` method. That's going to have size and performance implications on your application. You can wrap the `Add()` method shown in Item 41 in a bit of generic syntax to create a version that keeps the dynamic types in a constrained location. The same code will be

generated but in fewer places:

```
    private static dynamic DynamicAdd(dynamic left,
        dynamic right) =>
        left + right;

// Wrap it:
public static T1 Add<T1, T2>(T1 left, T2 right)
{
    dynamic result = DynamicAdd(left, right);
    return (T1)result;
}
```

The compiler generates all the dynamic callsite code in the generic `Add()` method. That isolates it into one location. Furthermore, the callsites become quite a bit simpler. Where previously every result was dynamic, now the result is statically typed to match the type of the first argument. Of course, you can create an overload to control the result type:

```
public static TResult Add<T1, T2, TResult>
    (T1 left, T2 right)
{
    dynamic result = DynamicAdd(left, right);
    return (TResult)result;
}
```

In either case, the callsites live completely in the strongly typed world:

```
int answer = Add(5, 5);
Console.WriteLine(answer);

double answer2 = Add(5.5, 7.3);
Console.WriteLine(answer2);

// Type arguments needed because
// args are not the same type
answer2 = Add<int, double, double>(5, 12.3);
Console.WriteLine(answer);

string stringLabel = System.Convert.ToString(answer);

string label = Add("Here is ", "a label");
Console.WriteLine(label);

DateTime tomorrow = Add(DateTime.Now, TimeSpan.FromDays(1));
```

```
Console.WriteLine(tomorrow);

label = "something" + 3;
Console.WriteLine(label);
label = Add("something", 3);
Console.WriteLine(label);
```

The above code is the same example from Item 41. Notice that this version has static types that are not dynamic as the return values. That means the caller does not need to work with dynamically typed objects. The caller works with static types, safely ignoring the machinations you needed to perform to make the operation work. In fact, they don't need to know that your algorithm ever left the safety of the type system.

Throughout the samples in this chapter, you saw that dynamic types are kept isolated to the smallest scope possible. When the code needs to use dynamic features, the samples show a local variable that is dynamic. The methods would convert that dynamic object into a strongly typed object and the dynamic object never left the scope of the method. When you use a dynamic object to implement an algorithm, you can avoid having that dynamic object be part of your interface. Other times, the very nature of the problem requires that a dynamic object be part of the interface. That is still not an excuse to make everything dynamic. Only the members that rely on dynamic behavior should use dynamic objects. You can mix dynamic and static typing in the same API. You want to create code that is statically typed when you can. Use dynamic only when you must.

We all have to work with CSV data in different forms. A production-hardened library is available at https://github.com/JoshClose/CsvHelper. Let's look at a simplified implementation. This snippet of code reads two different CSV files with different headers and displays the items in each row:

```
var data = new CSVDataContainer(
    new System.IO.StringReader(myCSV));
    foreach (var item in data.Rows)
        Console.WriteLine(@$"{item.Name}, {item.PhoneNumber},
{item.Label}");

data = new CSVDataContainer(
    new System.IO.StringReader(myCSV2));
foreach (var item in data.Rows)
```

```
    Console.WriteLine(@$"{item.Date}, {item.high},
{item.low}");
```

That's the API style I want for a general CSV reader class. The rows returned from enumerating the data contain properties for every row header name. Obviously, the row header names are not known at compile time. Those properties must be dynamic. But nothing else in the `CSVDataContainer` needs to be dynamic. The `CSVDataContainer` does not support dynamic typing. However, the `CSVDataContainer` does contain APIs that return a dynamic object that represents a row:

```
public class CSVDataContainer
{
    private class CSVRow : DynamicObject
    {
        private List<(string, string)> values =
            new List<(string, string)>();
        public CSVRow(IEnumerable<string> headers,
            IEnumerable<string> items)
        {
            values.AddRange(headers.Zip(items,
                (header, value) => (header,
                        value)));
        }

        public override bool TryGetMember(
            GetMemberBinder binder,
            out object result)
        {
            var answer = values.FirstOrDefault(n =>
                n.Item1 == binder.Name);
            result = answer.Item2;
            return result != null;
        }
    }
    private List<string> columnNames = new List<string>();
    private List<CSVRow> data = new List<CSVRow>();

    public CSVDataContainer(System.IO.TextReader stream)
    {
        // read headers:
        var headers = stream.ReadLine();
        columnNames =
            (from header in headers.Split(',')
                select header.Trim()).ToList();
```

```
        var line = stream.ReadLine();
        while (line != null)
        {
            var items = line.Split(',');
            data.Add(new CSVRow(columnNames, items));
            line = stream.ReadLine();
        }
    }
    public dynamic this[int index] => data[index];

    public IEnumerable<dynamic> Rows => data;
}
```

Even though you need to expose a dynamic type as part of your interface, it's only where necessary. Those APIs are dynamic. They must be. You can't support any possible CSV format without having dynamic support for column names. You could have chosen to expose everything using dynamic. Instead, dynamic appears in the interface only where the functionality demands dynamic.

For space purposes, I omitted other features in the `CSVDataContainer`. Think about how you would implement `RowCount`, `ColumnCount`, `GetAt(row, column)`, and other APIs. The implementation you have in your head would not use dynamic objects in the API, or even in the implementation. You can meet those requirements with static typing. You should. You'd only use dynamic in the public interface when it is needed.

Dynamic types are a useful feature, even in a statically typed language like C#. However, C# is still a statically typed language. The majority of C# programs should make the most out of the type system provided by the language. Dynamic programming is still useful, but it's most useful in C# when you keep it confined to those locations where it's needed and convert dynamic objects into a different static type immediately. When your code relies on a dynamic type created in another environment, wrap those dynamic objects and provide a public interface using different static types.

6. Participate in the global C# Community

The C# language is used by millions of developers world wide. That community has created a body of knowledge and conventional wisdom about the language. C# questions and answers are consistently in the top 10 categories on StackOverflow. The language team contributes as well, discussing language design on GitHub. The compiler is Open Source on GitHub as well. You need to get involved as well. Be part of the community.

Item 48: Seek the best answer, not the most popular answer

One challenge for a popular programming language community is to evolve practices as newer features are added to the language. The C# team has continued to add features to the language that address idioms that were hard to write correctly. The C# community would like to see more people adopt these newer practices. However, there is a significant body of work that recommends the previous patterns. There is a large amount of code in production that was written with earlier versions of the language. The examples in those products represent the best practices from the time when it was written. It also takes time for a new and better techniques to become the most popular in search engines and other sites. For that reason, you'll often find that the most popular recommendations are using older techniques instead of more modern and better choices.

The C# community is a large and diverse set of developers. The good news about that is the wealth of information available to learn C# and improve your technique. Plug your question into any search engine and you'll instantly get hundreds or thousands of answers. It takes time for great answers to generate the popularity to rise to the top of the rankings. Once they have arrived on the first page of search results, the most popular answers will often crowd out newer and better answers. The size of the C# community means that change in online results happens at a glacial pace. Newer developers search and find good answers that show the best ideas from two or three versions back of the

language. The popularity of these outdated answers was often the motivation for adding new language features. The language design team evaluates new features and adds those that can have the best impact on developers' daily work. The workarounds and extra code previously needed are precisely why these new features have been added. The popularity of the workarounds reflects the importance of the new features. Now, it's up to us as professional developers to help popularize the best features.

One example is covered in Item 8 in the latest edition of Effective C#. It points out that using the `?.` operator to invoke a delegate or raise an event provides a thread safe way to check for null and invoke those methods when not null. There is a long history using a practice that initialized a local variable and then checking it before raising events. This outdated practice is still the most popular in many sites.

A huge majority of the sample code and production code that works with text output uses the archaic positional syntax to perform string formatting and substitution. Items 4 and 5 in Effective C# covers the new interpolated string syntax.

Professional developers should do what they can to promote the best modern techniques. First and foremost, when you search for answers, look beyond the most popular answer to find the best answer for modern C# development. Do enough research to determine what answer is best for your environment and the platforms and versions you must support.

Second, once you find that best answer, support it. That's how these newer answers will slowly rise to be the most popular. Third, where possible, update the pages with the more popular answers so that they reference the newer, better answer. Finally, when you work on your own codebases, with each update, make that code better. Look at the code you're updating and if there are opportunities to refactor it to use better techniques, make some of those fixes. You won't update an entire code base in one sprint, but small amounts of additional care will add up.

Follow those recommendations to be a good citizen, and you'll help others in the C# community find the best, most relevant answers to modern questions.

It takes time for large communities to adopt newer and better practices. Some may move immediately, others many years later. Where ever you and your team are on the adoption curve, you can help by promoting the best answers for modern development.

Item 49: Participate in Specs and Code

The Open Source nature of C# extends beyond the compiler code. The language design process is an open process sas well. There is a wealth of resources available to you to continue learning and growing your C# knowledge. It is one of the best ways for you to learn, keep up to date, and participate in the ongoing evolution of C#. You should visit https://github.com/dotnet/roslyn and https://github.com/dotnet/csharplang on a regular basis.

This represents a major change for C#, and is a new way for you to learn and participate. There are many ways for you to get involved and grow.

It is open source, so you can build the compiler yourself, and run your own C# compiler. You can submit changes, and even suggest your own features and enhancements. If you have great ideas, you can prototype them in your fork and submit the pull request for your new feature.

That may be more than you want to be involved. There are plenty of other ways for you to become involved. If you find any issues, report them on the Roslyn repo at GitHub. The open issues there represent the work that the team is doing or has planned. The issues encompass suspected bugs, spec issues, new feature requests and ongoing updates to the language.

You can also find specifications for any new features on the CSharpLang repository. It’s your way to learn about upcoming features, and to participate in the design and evolution of the language. All the specifications are open for comment and discussion. Read the thoughts of other community members, participate in the discussions, and take an interest in the evolution of your favorite language. New specifications are posted as they are ready for review, and the cadence varies along the release cycle. More are published early in the planning phases for a new release, and fewer as a release nears its final stages.

Even during the early phases, you can keep up by reading one or two feature specs a week. References to the specifications are found as in the "Proposals" folder on the CSharpLang repository. Each proposal has a "champion" issue that tracks its status. These issues are also where you can comment on active proposals.

In addition to the specifications, the minutes from the language design meetings are also posted to the CSharpLang repository. Reading those will give you a deeper understanding of the considerations behind any new features in the language. You'll learn the constraints involved in adding any new feature to a mature language. The language design team discusses the positive and negative impact of any new feature, weighs the benefits, the schedule, and the impact. You'll learn about the scenarios that are anticipated for each new feature. These notes are also open for comment and discussion. Add your voice, and take an interest in the evolution of the language. The language design team meets roughly once a month. The notes are often published shortly thereafter. This is not a huge impact on your time, and is well worth the investment. These are published in the "Meetings" folder on the CSharpLang repository.

The C# Language specification has been converted to markdown, and is also stored on the CSharpLang repository. Comment, write issues, or even issue a PR if you find mistakes.

If you feel more adventurous, clone the Roslyn repository and look at the unit tests for the compiler. You'll learn more depth on the rules that govern any of the features in the language.

It's a big change for the C# community to move from a closed source implementation where only a small percentage of the community were given early access and could participate in the discussions. Now, they are open to the entire C# community. You should participate. Learn more about the features coming in the next releases. Participate where you have greatest interest.

Item 50: Consider automating practices with Analyzers

Effective C# and this book contain recommendations for writing better code. Implementing many of these recommendations can be supported by analyzers

and code fixes built using the Roslyn APIs. These APIs enable addins to analyze code at a semantic level and modify that code to introduce better practices.

The best part is that you may not need to write those analyzers yourself. There are several Open Source projects containing analyzers and code fixes for a variety of practices.

One very popular project is built by the Roslyn compiler team. The Roslyn Analyzers project (https://github.com/dotnet/roslyn-analyzers) started as a way for the team to validate the static analysis APIs. It has since grown and provides automatic validation for many common guidelines used by the Roslyn teams.

Another popular library is the Code Cracker project (https://github.com/code-cracker/code-cracker). This project is created by members of the .NET community, and reflects their thoughts on best practices for writing code. There are both C# and VB.NET versions of the analyzers available.

Another group of community members created a GitHub organization and wrote a set of analyzers to implement different recommendations. Visit the .NET Analyzers organization page (https://github.com/DotNetAnalyzers) to learn more. They have created analyzers for different application types and libraries.

Before you install any of these, research what rules they enforce and how strident they are about enforcing the rules. Analyzers can report rule violations as information, warnings, or even errors. In addition, different analyzers may have different opinions about different rules. Those rules may conflict with each other. You may find that fixing one violation causes a violation from another analyzer. (For example, some analyzers prefer using implicitly typed variables, declared with var and other analyzers prefer explicitly naming each variable's type.) Many of the analyzers enable configuration options that configure which rules are reported on, and which are ignored. That can help create a configuration that works for you. You'll also learn more about the rules and practices of each of the analyzers.

If you have practices that you want to follow that aren't represented by

analyzers you can find, consider building your own. The analyzers in the Roslyn Analyzer repository are Open Source, and provide a great template for building analyzers. Building an analyzer is an advanced topic. It requires a deep understanding of C# syntax and semantic analysis. However, working on a simple analyzer can give you much deeper insight into the C# language. As an example, you can explore a repository I've used to explain the techniques. This repository on GitHub https://github.com/BillWagner/NonVirtualEventAnalyzer shows how to build an analyzer that finds and replaces virtual events (See Item 21). Each number branch shows one step in analyzing and fixing the code.

The Roslyn APIs for analyzers and code fixes enable automated validation for any code practice you want to enforce. There are already a rich set of analyzers available to use that have been built by the team and the community. If none of those fulfill your needs, you can work to build your own. That is an advanced technique, but it will help you gain a much deeper knowledge of the C# language rules.