*Effective* SOFTWARE DEVELOPMENT SERIES
**Scott Meyers, Consulting Editor**

# *Effective* C#

*Third Edition*

COVERS C# 6.0

## *50 Specific Ways to Improve Your C#*

**Content Update Program**
***FREE***…See Details Inside

Bill Wagner

# Praise for *Effective C#, Second Edition*

"Being an effective .NET developer requires one to have a deep understanding of the language of their choice. Wagner's book provides the reader with that knowledge via well-reasoned arguments and insight. Whether you're new to C# or you've been using it for years, you'll learn something new when you read this book."

—Jason Bock, Principal Consultant, Magenic

"If you're at all like me, you have collected a handful of C# language pearls that have immediately transformed your abilities as a professional developer. What you hold in your hands is quite possibly the best collection of these tips that have ever been assembled. Bill has managed to exceed my wildest expectations with the latest edition in his eponymous *Effective C#*."

—Bill Craun, Principal Consultant, Ambassador Solutions

"*Effective C#, Second Edition*, is a must-read for anyone building high performance and/or highly scalable applications. Bill has that rare and awesome ability to take an amazingly complex problem and break it down into human, digestible, and understandable chunks."

—Josh Holmes, Architect Evangelist, Microsoft

"Bill has done it again. This book is a concise collection of invaluable tips for any C# developer. Learn one tip every day, and you'll become a much better C# developer after fifty days!"

—Claudio Lassala, Lead Developer, EPS Software/CODE Magazine

"A fountain of knowledge and understanding of the C# language. Bill gives insight to what happens under the covers of the .NET runtime based on what you write in your code and teaches pragmatic practices that lead to cleaner, easier to write, and more understandable code. A great mix of tips, tricks, and deep understanding . . . that every C# developer should read."

—Brian Noyes, Chief Architect, IDesign Inc. (www.idesign.net)

"*Effective C#* is a must-have for every C# developer. Period. Its pragmatic advice on code design is invaluable."

—Shawn Wildermuth, Microsoft MVP (C#), Author, Trainer, and Speaker

"In this book Bill Wagner provides practical explanations of how to use the most important features in the C# language. His deep knowledge and sophisticated communication skills illuminate the new features in C# so that you can use them to write programs that are more concise and easier to maintain."

—Charlie Calvert, Microsoft C# Community Program Manager

*This page intentionally left blank*

# Effective C#

## Third Edition

# Effective C#

## 50 Specific Ways to Improve Your C#

## Third Edition

Bill Wagner

Addison-Wesley

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

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

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

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

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2016953545

ISBN-13: 978-0-672-33787-1
ISBN-10: 0-672-33787-8

1 16

*To Marlene, who continues to provide inspiration and support for everything we do together.*

# Contents at a Glance

# Contents

## Chapter 4 Working with LINQ 133

## Chapter 5 Exception Practices 221

# Introduction

The C# community is very different in 2016 from what it was in 2004 when the first edition of *Effective C#* was published. There are many more developers using C#. A large contingent of the C# community is now seeing C# as their first professional language. They aren't approaching C# with a set of ingrained habits formed using a different language. The community has a much broader range of experience. New graduates all the way to professionals with decades of experience are using C#. C# now runs on multiple platforms. You can build server applications, Web sites, desktop applications, and mobile applications for multiple platforms in the C# language.

I organized this third edition of *Effective C#* by taking into account both the changes in the language and the changes in the C# community. *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 first edition of *More Effective C#* is included in this edition, and a larger number of items have been removed. With this edition, I'm reorganizing both books, and a new edition of *More Effective C#* will cover other concepts. 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# 6.0, but it is not an exhaustive treatment of the new language features. Like all books in the Effective Software Development Series, it offers practical advice on how to use these features to solve problems you're likely to encounter every day. I specifically cover C# 6.0 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/EffectiveCSharpAnalyzers. If you have ideas or want to contribute, write an issue or send me a pull request.

## Who Should Read This Book?

*Effective C#* was written for professional developers who use C# as part of their daily toolset. It assumes you are familiar with the C# syntax and the language's features. This book does not include tutorial instruction on language features. Instead, it discusses how you can integrate all the features of the current version of the C# language into your everyday development.

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

## About The Content

There are language constructs you'll use every day in almost every C# program you write. Chapter 1, "C# Language Idioms," covers those language idioms you'll use so often they should feel like well-worn tools in your hands. These are the building blocks of every type you create and every algorithm you implement.

Working in a managed environment doesn't mean the environment absolves you of all your responsibilities. You still must work with the environment to create correct programs that satisfy the stated performance requirements. It's not just about performance testing and performance tuning. Chapter 2, ".NET Resource Management," teaches you the design idioms that enable you to work with the environment to achieve those goals before detailed optimization begins.

Generics are the enabling technology for everything else added to the C# language since C# 2.0. Chapter 3, "Working with Generics," covers generics as a replacement for `System.Object` and casts and then moves on to discuss advanced techniques such as constraints, generic specialization, method constraints, and backward compatibility. You'll learn several techniques in which generics will make it easier to express your design intent.

Chapter 4, "Working with LINQ," explains LINQ, query syntax, and related features. You'll see when to use extension methods to separate contracts from implementation, how to use C# closures effectively, and how to program with anonymous types. You'll learn how the compiler maps query keywords to method calls, how to distinguish between delegates and expression trees (and convert between them when needed), and how to escape queries when you're looking for scalar results.

Chapter 5, "Exception Practices," provides guidance on managing exceptions and errors in modern C# programs. You'll learn how to ensure that errors are reported properly and how to leave program state consistent and ideally unchanged when errors occur. You'll learn how to provide a better debugging experience for developers who use your code.

## 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 ones. 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/EffectiveCS. 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.

Register your copy of *Effective C#, Third Edition*, at informit.com for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780672337871) and click Submit. Once the process is complete, you will find any available bonus content under “Registered Products.”

## 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-Bisson, Jared Parsons, Scott Allen, and, most importantly, Mads Torgersen are the basis for many new ideas in this edition.

I had a wonderful team of technical reviewers for this edition. Jason Bock, Mark Michaelis, and Eric Lippert pored over the text and the 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 Renfro 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 of this book you chose, Curt has had something to do with its existence.

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.

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.

*This page intentionally left blank*

# 2 .NET Resource Management

The simple fact that .NET programs run in a managed environment has a big impact on the kinds of designs that create effective C#. Taking advantage of that environment requires changing your thinking from other environments to the .NET Common Language Runtime (CLR). It means understanding the .NET garbage collector (GC). It means understanding object lifetimes. It means understanding how to control unmanaged resources. This chapter covers the practices that help you create software that makes the best use of the environment and its features.

## Item 11: Understand .NET Resource Management

You can't be an effective developer without understanding how the environment handles memory and other important resources. In .NET, that means understanding memory management and the garbage collector.

The GC controls managed memory for you. Unlike in native environments, you are not responsible for most memory leaks, dangling pointers, uninitialized pointers, or a host of other memory-management issues. But the garbage collector works better when you need to clean up after yourself. You are responsible for unmanaged resources such as database connections, GDI+ objects, COM objects, and other system objects. In addition, you can cause objects to stay in memory longer than you'd like because you've created links between them using event handlers or delegates. Queries, which execute when results are requested, can also cause objects to remain referenced longer than you would expect (see Item 41).

Here's the good news: Because the GC controls memory, certain design idioms are much easier to implement than when you must manage all memory yourself. Circular references, both simple relationships and complex webs of objects, are much easier to implement correctly than in environments where you must manage memory. The GC's Mark and Compact algorithm efficiently detects these relationships and removes unreachable webs of objects in their entirety. The GC determines whether an object is reachable by walking the object tree from the application's root object instead of forcing each object to keep track of references to it, as in COM. The `EntitySet` class provides an example of how this algorithm simplifies object ownership decisions. An entity is a collection of objects loaded from a database. Each entity may contain references to other entity objects. Any of these entities may also contain links to other entities. Just like the relational database entity sets model, these links and references may be circular.

There are references all through the web of objects represented by different entity sets. Releasing memory is the GC's responsibility. Because the .NET Framework designers did not need to free these objects, the complicated web of object references did not pose a problem. No decision needed to be made regarding the proper sequence of freeing this web of objects; it's the GC's job. The GC's design simplifies the problem of identifying this kind of web of objects as garbage. The application can stop referencing any entity when it's done. The garbage collector will know if the entity is still reachable from live objects in the application. Any objects that cannot be reached from the application are garbage.

The garbage collector compacts the managed heap each time it runs. Compacting the heap moves each live object in the managed heap so that the free space is located in one contiguous block of memory. Figure 2.1 shows two snapshots of the heap before and after a garbage collection. All free memory is placed in one contiguous block after each GC operation.

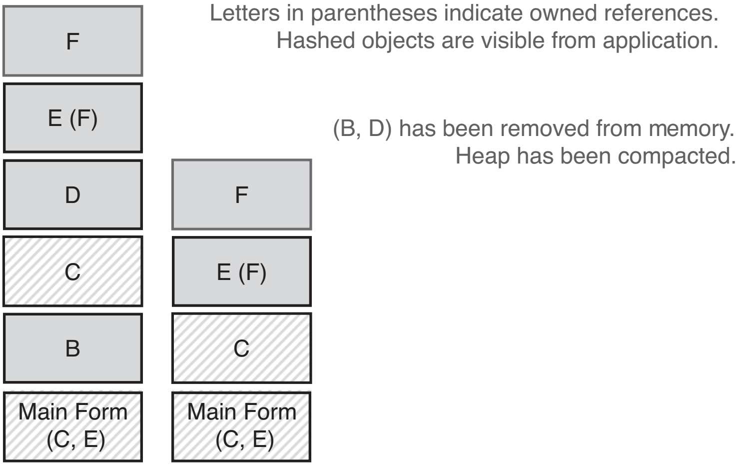

**Figure 2.1** The garbage collector not only removes unused memory, but it also moves other objects in memory to compact used memory and maximize free space.

As you've just learned, memory management (for the managed heap) is completely the responsibility of the garbage collector. Other system resources must be managed by developers: you and the users of your classes. Two mechanisms help developers control the lifetimes of unmanaged resources: finalizers and the `IDisposable` interface. A finalizer is a defensive mechanism that ensures that your objects always have a way to release unmanaged resources. Finalizers have many drawbacks, so you also have the `IDisposable` interface that provides a less intrusive way to return resources to the system in a timely manner.

Finalizers are called by the garbage collector at some time after an object becomes garbage. You don't know when that happens. All you know is that in most environments it happens sometime after your object cannot be reached. That is a big change from C++, and it has important ramifications for your designs. Experienced C++ programmers wrote classes that allocated a critical resource in its constructor and released it in its destructor:

```
// Good C++, bad C#:
class CriticalSection
{
    // Constructor acquires the system resource.
    public CriticalSection()
    {
        EnterCriticalSection();
    }

    // Destructor releases system resource.
    ~CriticalSection()
    {
        ExitCriticalSection();
    }

    private void ExitCriticalSection()
    {
    }
    private void EnterCriticalSection()
    {
    }
}

// usage:
void Func()
```

```
{
    // The lifetime of s controls access to
    // the system resource.
    CriticalSection s = new CriticalSection();
    // Do work.

    //...

    // compiler generates call to destructor.
    // code exits critical section.
}
```

This common C++ idiom ensures that resource deallocation is exception proof. This doesn't work in C#, however—at least not in the same way. Deterministic finalization is not part of the .NET environment or the C# language. Trying to force the C++ idiom of deterministic finalization into the C# language won't work well. In C#, the finalizer eventually executes in most environments, but it doesn't execute in a timely fashion. In the previous example, the code eventually exits the critical section, but in C# it doesn't exit the critical section when the function exits. That happens at some unknown time later. You don't know when. You can't know when. Finalizers are the only way to guarantee that unmanaged resources allocated by an object of a given type are eventually released. But finalizers execute at nondeterministic times, so your design and coding practices should minimize the need for creating finalizers, and also minimize the need for executing the finalizers that do exist. Throughout this chapter you'll learn techniques to avoid creating your own finalizer, and how to minimize the negative impact of having one when it must be present.

Relying on finalizers also introduces performance penalties. Objects that require finalization put a performance drag on the garbage collector. When the GC finds that an object is garbage but also requires finalization, it cannot remove that item from memory just yet. First, it calls the finalizer. Finalizers are not executed by the same thread that collects garbage. Instead, the GC places each object that is ready for finalization in a queue and executes all the finalizers for those objects. It continues with its business, removing other garbage from memory. On the next GC cycle, those objects that have been finalized are removed from memory. Figure 2.2 shows three different GC operations and the difference in memory usage. Notice that the objects that require finalizers stay in memory for extra cycles.

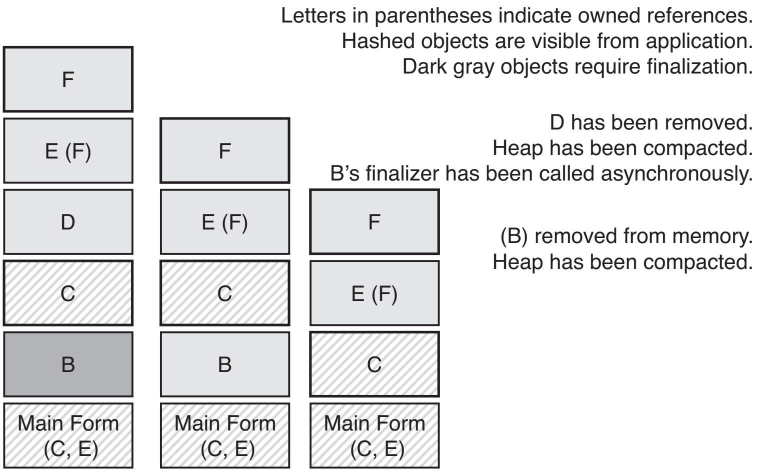

**Figure 2.2** This sequence shows the effect of finalizers on the garbage collector. Objects stay in memory longer, and an extra thread needs to be spawned to run the garbage collector.

This might lead you to believe that an object that requires finalization lives in memory for one GC cycle more than necessary. But I simplified things. It’s more complicated than that because of another GC design decision. The .NET garbage collector defines generations to optimize its work. Generations help the GC identify the likeliest garbage candidates more quickly. Any object created since the last garbage collection operation is a generation 0 object. Any object that has survived one GC operation is a generation 1 object. Any object that has survived two or more GC operations is a generation 2 object. The purpose of generations is to separate short-lived objects from objects that stay around for the life of the application. Generation 0 objects are mostly those short-lived object variables. Member variables and global variables quickly enter generation 1 and eventually enter generation 2.

The GC optimizes its work by limiting how often it examines first- and second-generation objects. Every GC cycle examines generation 0 objects. Roughly one GC out of ten examines the generation 0 and 1 objects. Roughly one GC cycle out of 100 examines all objects. Think about finalization and its cost again: An object that requires finalization might stay in memory for nine GC cycles more than it would if it did not require finalization. If it still has not been finalized, it moves to generation 2. In generation 2, an object lives for an extra 100 GC cycles until the next generation 2 collection.

I've spent some time explaining why finalizers are not a good solution. Yet you still need to free resources. You address these issues using the `IDisposable` interface and the standard dispose pattern (see Item 17 later in this chapter).

To close, remember that a managed environment, where the garbage collector takes the responsibility for memory management, is a big plus: Memory leaks and a host of other pointer-related problems are no longer your problem. Nonmemory resources force you to create finalizers to ensure proper cleanup of those nonmemory resources. Finalizers can have a serious impact on the performance of your program, but you must write them to avoid resource leaks. Implementing and using the `IDisposable` interface avoids the performance drain on the garbage collector that finalizers introduce. The next item describes the specific techniques that will help you create programs that use this environment more effectively.

---

## Item 12: Prefer Member Initializers to Assignment Statements

Classes often have more than one constructor. Over time, it's easy for the member variables and the constructors to get out of sync. The best way to make sure this doesn't happen is to initialize variables where you declare them instead of in the body of every constructor. You should use the initializer syntax for both static and instance variables.

Constructing a member variable when you declare that variable is natural in C#. Just initialize the variable when you declare it:

```
public class MyClass
{
    // declare the collection, and initialize it.
    private List<string> labels = new List<string>();

}
```

Regardless of the number of constructors you eventually add to the `MyClass` type, labels will be initialized properly. The compiler generates code at the beginning of each constructor to execute all the initializers you have defined for your instance member variables. When you add a new constructor, labels get initialized. Similarly, if you add a new member variable, you do not need to add initialization code to every constructor; initializing the variable where you define it is sufficient. Equally important, the initializers are added to the compiler-generated default constructor. The C# compiler creates a default constructor for your types whenever you don't explicitly define any constructors.

Initializers are more than a convenient shortcut for statements in a constructor body. The statements generated by initializers are placed in object code before the body of your constructors. Initializers execute before the base class constructor for your type executes, and they are executed in the order in which the variables are declared in your class.

Using initializers is the simplest way to avoid uninitialized variables in your types, but it's not perfect. In three cases, you should not use the initializer syntax. The first is when you are initializing the object to 0, or `null`. The default system initialization sets everything to 0 for you before any of your code executes. The system-generated 0 initialization is done at a very low level using the CPU instructions to set the entire block of memory to 0. Any extra 0 initialization on your part is superfluous. The C# compiler dutifully adds the extra instructions to set memory to 0 again. It's not wrong—but it can create brittle code.

```
public struct MyValType
{
    // elided
}

MyValType myVal1;  // initialized to 0
MyValType myVal2 = new MyValType(); // also 0
```

Both statements initialize the variable to all 0s. The first does so by setting the memory containing `myVal1` to 0. The second uses the IL instruction `initobj`, which causes both a box and an unbox operation on the `myVal2` variable. This takes quite a bit of extra time (see Item 9).

The second inefficiency comes when you create multiple initializations for the same object. You should use the initializer syntax only for variables that receive the same initialization in all constructors. This version of `MyClass` has a path that creates two different `List` objects as part of its construction:

```
public class MyClass2
 {
     // declare the collection, and initialize it.
     private List<string> labels = new List<string>();
```

```
        MyClass2()
        {
        }

        MyClass2(int size)
        {
            labels = new List<string>(size);
        }
    }
```

When you create a new `MyClass2`, specifying the size of the collection, you create two array lists. One is immediately garbage. The variable initializer executes before every constructor. The constructor body creates the second array list. The compiler creates this version of `MyClass2`, which you would never code by hand. (For the proper way to handle this situation, see Item 14 later in this chapter.)

```
public class MyClass2
{
    // declare the collection, and initialize it.
    private List<string> labels;

    MyClass2()
    {
        labels = new List<string>();
    }

    MyClass2(int size)
    {
        labels = new List<string>();
        labels = new List<string>(size);
    }
}
```

You can run into the same situation whenever you use implicit properties. For those data elements where implicit properties are the right choice, Item 14 shows how to minimize any duplication when you initialize data held in implicit properties.

The final reason to move initialization into the body of a constructor is to facilitate exception handling. You cannot wrap the initializers in a `try` block. Any exceptions that might be generated during the construction of your member variables get propagated outside your object. You cannot attempt any recovery inside your class. You should move that initialization code into the body of your constructors so that you implement the proper recovery code to create your type and gracefully handle the exception (see Item 47).

Member initializers are the simplest way to ensure that the member variables in your type are initialized regardless of which constructor is called. The initializers are executed before each constructor you make for your type. Using this syntax means that you cannot forget to add the proper initialization when you add new constructors for a future release. Use initializers when all constructors create the member variable the same way; it's simpler to read and easier to maintain.

---

## Item 13: Use Proper Initialization for Static Class Members

You know that you should initialize static member variables in a type before you create any instances of that type. C# lets you use static initializers and a static constructor for this purpose. A static constructor is a special function that executes before any other methods, variables, or properties defined in that class are accessed for the first time. You use this function to initialize static variables, enforce the singleton pattern, or perform any other necessary work before a class is usable. You should not use your instance constructors, some special private function, or any other idiom to initialize static variables. For static fields that require complex or expensive initialization, consider using `Lazy<T>` to execute the initialization when a field is first accessed.

As with instance initialization, you can use the initializer syntax as an alternative to the static constructor. If you simply need to allocate a static member, use the initializer syntax. When you have more complicated logic to initialize static member variables, create a static constructor.

Implementing the singleton pattern in C# is the most frequent use of a static constructor. Make your instance constructor private, and add an initializer:

```
public class MySingleton
{
    private static readonly MySingleton theOneAndOnly =
    new MySingleton();
```

```
    public static MySingleton TheOnly
    {
        get { return theOneAndOnly; }
    }

    private MySingleton()
    {
    }

    // remainder elided
}
```

The singleton pattern can just as easily be written this way, in case you have more complicated logic to initialize the singleton:

```
public class MySingleton2
{
    private static readonly MySingleton2 theOneAndOnly;

    static MySingleton2()
    {
        theOneAndOnly = new MySingleton2();
    }
    public static MySingleton2 TheOnly
    {
        get { return theOneAndOnly; }
    }

    private MySingleton2()
    {
    }

    // remainder elided
}
```

Like instance initializers, the static initializers are executed before any static constructors are called. And, yes, your static initializers may execute before the base class's static constructor.

The CLR calls your static constructor automatically before your type is first accessed in an application space (an `AppDomain`). You can define only one static constructor, and it must not take any arguments. Because static constructors are called by the CLR, you must be careful about exceptions generated in them. If you let an exception escape a static constructor, the CLR will terminate your program by throwing a `TypeInitializationException`. The situation where the caller catches the exception is even more insidious. Code that tries to create the type will fail until that `AppDomain` is unloaded. The CLR could not initialize the type by executing the static constructor. It won't try again, and yet the type did not get initialized correctly. An object of that type (or any type derived from it) would not be well defined. Therefore, it is not allowed.

Exceptions are the most common reason to use the static constructor instead of static initializers. If you use static initializers, you cannot catch the exceptions yourself. With a static constructor, you can (see Item 47):

```
static MySingleton2()
{
    try
    {
        theOneAndOnly = new MySingleton2();
    }
    catch
    {
        // Attempt recovery here.
    }
}
```

Static initializers and static constructors provide the cleanest, clearest way to initialize static members of your class. They are easy to read and easy to get correct. They were added to the language to specifically address the difficulties involved with initializing static members in other languages.

## Item 14: Minimize Duplicate Initialization Logic

Writing constructors is often a repetitive task. Many developers write the first constructor and then copy and paste the code into other constructors to satisfy the multiple overrides defined in the class interface. Ideally, you're not one of those. If you are, stop it. Veteran C++ programmers would factor the common algorithms into a private helper method. Stop that, too. When you find that multiple constructors contain the same logic, factor that logic into a common constructor instead. You'll get the benefits of avoiding code duplication, and constructor initializers generate much more efficient object code. The C# compiler recognizes the constructor initializer as special syntax and removes the duplicated variable initializers and the duplicated base class constructor calls. The result is that your final object executes the minimum amount of code to properly initialize the object. You also write the least amount of code by delegating responsibilities to a common constructor.

Constructor initializers allow one constructor to call another constructor. This example shows a simple usage:

```
public class MyClass
{
    // collection of data
    private List<ImportantData> coll;
    // Name of the instance:
    private string name;

    public MyClass() :
        this(0, "")
    {
    }

    public MyClass(int initialCount) :
        this(initialCount, string.Empty)
    {
    }

    public MyClass(int initialCount, string name)
    {
        coll = (initialCount > 0) ?
        new List<ImportantData>(initialCount) :
        new List<ImportantData>();
        this.name = name;
    }
}
```

C# 4.0 added default parameters, which you can use to minimize the duplicated code in constructors. You could replace all the different constructors for `MyClass` with one constructor that specifies default values for all or many of the values:

```
public class MyClass
{
    // collection of data
    private List<ImportantData> coll;
```

```
    // Name of the instance:
    private string name;

    // Needed to satisfy the new() constraint.
    public MyClass() :
        this(0, string.Empty)
    {
    }

    public MyClass(int initialCount = 0, string name = "")
    {
        coll = (initialCount > 0) ?
        new List<ImportantData>(initialCount) :
        new List<ImportantData>();
        this.name = name;
    }
}
```

There are tradeoffs in choosing default parameters over using multiple overloads. Default parameters create more options for your users. This version of `MyClass` specifies the default value for both parameters. Users could specify different values for either or both parameters. Producing all the permutations using overloaded constructors would require four different constructor overloads: a parameterless constructor, one that asks for the initial count, one that asks for the name, and one that asks for both parameters. Add more members to your class, and the number of potential overloads grows as the number of permutations of all the parameters grows. That complexity makes default parameters a very powerful mechanism to minimize the number of potential overloads that you need to create.

Defining default values for all parameters to your type's constructor means that user code will be valid when you call the new `MyClass()`. When you intend to support this concept, you should create an explicit parameterless constructor in that type, as shown in the example code above. While most code would default all parameters, generic classes that use the `new()` constraint will not accept a constructor with parameters that have default values. To satisfy the `new()` constraint, a class must have an explicit parameterless constructor. Therefore, you should create one so that clients can use your type in generic classes or methods that enforce the `new()` constraint. That's not to say that every type needs a parameterless constructor. However, if you support one, make sure to add the code so that the parameterless constructor works in all cases, even when called from a generic class with a `new()` constraint.

You'll note that the second constructor specifies "" for the default value on the name parameter, rather than the more customary `string.Empty`. That's because `string.Empty` is not a compile-time constant. It is a static property defined in the `string` class. Because it is not a compile-time constant, you cannot use it for the default value for a parameter.

However, using default parameters instead of overloads creates tighter coupling between your class and all the clients that use it. In particular, the formal parameter name becomes part of the public interface, as does the current default value. Changing parameter values requires a recompile of all client code in order to pick up those changes. That makes overloaded constructors more resilient in the face of potential future changes. You can add new constructors, or change the default behavior for those constructors that don't specify values, without breaking client code.

Default parameters are the preferred solution to this problem. However, some APIs use reflection to create objects and rely on a parameterless constructor. A constructor with defaults supplied for all arguments is not the same as a parameterless constructor. You may need to write separate constructors that you support as a separate function. With constructors, that can mean a lot of duplicated code. Use constructor chaining, by having one constructor invoke another constructor declared in the same class, instead of creating a common utility routine. Several inefficiencies are present in this alternative method of factoring out common constructor logic:

```
public class MyClass
{
    private List<ImportantData> coll;
    private string name;

    public MyClass()
    {
        commonConstructor(0, "");
    }
```

```
    public MyClass(int initialCount)
    {
        commonConstructor(initialCount, "");
    }

    public MyClass(int initialCount, string Name)
    {
        commonConstructor(initialCount, Name);
    }

    private void commonConstructor(int count,
    string name)
    {
        coll = (count > 0) ?
        new List<ImportantData>(count) :
        new List<ImportantData>();
        this.name = name;
    }
}
```

That version looks the same, but it generates far-less-efficient object code. The compiler adds code to perform several functions on your behalf in constructors. It adds statements for all variable initializers (see Item 12 earlier in this chapter). It calls the base class constructor. When you write your own common utility function, the compiler cannot factor out this duplicated code. The IL for the second version is the same as if you'd written this:

```
public class MyClass
{
    private List<ImportantData> coll;
    private string name;

    public MyClass()
    {
        // Instance Initializers would go here.
        object(); // Not legal, illustrative only.
        commonConstructor(0, "");
    }
```

```
    public MyClass(int initialCount)
    {
        // Instance Initializers would go here.
        object(); // Not legal, illustrative only.
        commonConstructor(initialCount, "");
    }

    public MyClass(int initialCount, string Name)
    {
        // Instance Initializers would go here.
        object(); // Not legal, illustrative only.
        commonConstructor(initialCount, Name);
    }

    private void commonConstructor(int count,
    string name)
    {
        coll = (count > 0) ?
        new List<ImportantData>(count) :
        new List<ImportantData>();
        this.name = name;
    }
}
```

If you could write the construction code for the first version the way the compiler sees it, you'd write this:

```
// Not legal, illustrates IL generated:
public class MyClass
{
    private List<ImportantData> coll;
    private string name;

    public MyClass()
    {
        // No variable initializers here.
        // Call the third constructor, shown below.
        this(0, ""); // Not legal, illustrative only.
    }
```

```
    public MyClass(int initialCount)
    {
        // No variable initializers here.
        // Call the third constructor, shown below.
        this(initialCount, "");
    }

    public MyClass(int initialCount, string Name)
    {
        // Instance Initializers would go here.
        //object(); // Not legal, illustrative only.
        coll = (initialCount > 0) ?
        new List<ImportantData>(initialCount) :
        new List<ImportantData>();
        name = Name;
    }
}
```

The difference is that the compiler does not generate multiple calls to the base class constructor, nor does it copy the instance variable initializers into each constructor body. The fact that the base class constructor is called only from the last constructor is also significant: You cannot include more than one constructor initializer in a constructor definition. You can delegate to another constructor in this class using `this()`, or you can call a base class constructor using `base()`. You cannot do both.

Still don't buy the case for constructor initializers? Then think about read-only constants. In this example, the name of the object should not change during its lifetime. This means that you should make it read-only. That causes the common utility function to generate compiler errors:

```
public class MyClass
{
    // collection of data
    private List<ImportantData> coll;
    // Number for this instance
    private int counter;
    // Name of the instance:
    private readonly string name;
```

```
    public MyClass()
    {
        commonConstructor(0, string.Empty);
    }

    public MyClass(int initialCount)
    {
        commonConstructor(initialCount, string.Empty);
    }

    public MyClass(int initialCount, string Name)
    {
        commonConstructor(initialCount, Name);
    }

    private void commonConstructor(int count,
    string name)
    {
        coll = (count > 0) ?
        new List<ImportantData>(count) :
        new List<ImportantData>();
        // ERROR changing the name outside of a constructor.
        //this.name = name;
    }
}
```

The compiler enforces the read-only nature of `this.name` and will not allow any code not in a constructor to modify it. C#'s constructor initializers provide the alternative. All but the most trivial classes contain more than one constructor. Their job is to initialize all the members of an object. By their very nature, these functions have similar or, ideally, shared logic. Use the C# constructor initializer to factor out those common algorithms so that you write them once and they execute once.

Both default parameters and overloads have their place. In general, you should prefer default values to overloaded constructors. After all, if you are letting client developers specify parameter values at all, your constructor must be capable of handling any values that users specify. Your original default values should always be reasonable and shouldn't generate exceptions. Therefore, even though changing the default parameter values is technically a breaking change, it shouldn't be observable to your clients. Their code will still use the original values, and those original values should still produce reasonable behavior. That minimizes the potential hazards of using default values.

This is the last item about object initialization in C#. That makes it a good time to review the entire sequence of events for constructing an instance of a type. You should understand both the order of operations and the default initialization of an object. You should strive to initialize every member variable exactly once during construction. The best way for you to accomplish this is to initialize values as early as possible. Here is the order of operations for constructing the first instance of a type:

1. Static variable storage is set to 0.
2. Static variable initializers execute.
3. Static constructors for the base class execute.
4. The static constructor executes.
5. Instance variable storage is set to 0.
6. Instance variable initializers execute.
7. The appropriate base class instance constructor executes.
8. The instance constructor executes.

Subsequent instances of the same type start at step 5 because the class initializers execute only once. Also, steps 6 and 7 are optimized so that constructor initializers cause the compiler to remove duplicate instructions.

The C# language compiler guarantees that everything gets initialized in some way when an object is created. At a minimum, you are guaranteed that all memory your object uses has been set to 0 when an instance is created. This is true for both static members and instance members. Your goal is to make sure that you initialize all the values the way you want and execute that initialization code only once. Use initializers to initialize simple resources. Use constructors to initialize members that require more sophisticated logic. Also factor calls to other constructors to minimize duplication.

## Item 15: Avoid Creating Unnecessary Objects

The garbage collector does an excellent job of managing memory for you, and it removes unused objects in a very efficient manner. But no matter how you look at it, allocating and destroying a heap-based object takes more processor time than not allocating and not destroying a heap-based object. You can introduce serious performance drains on your program by creating an excessive number of reference objects that are local to your methods.

So don't overwork the garbage collector. You can follow some simple techniques to minimize the amount of work that the GC needs to do on your program's behalf. All reference types, even local variables, create memory allocations. These objects become garbage when no root is keeping them alive. For local variables, that is typically when the method in which they are declared is no longer active. One very common bad practice is to allocate GDI objects in a Windows paint handler:

```
protected override void OnPaint(PaintEventArgs e)
{
    // Bad. Created the same font every paint event.
    using (Font MyFont = new Font("Arial", 10.0f))
    {
        e.Graphics.DrawString(DateTime.Now.ToString(),
            MyFont, Brushes.Black, new PointF(0, 0));
    }
    base.OnPaint(e);
}
```

`OnPaint()` gets called frequently. Every time it gets called, you create another `Font` object that contains the exact same settings. The garbage collector needs to clean those up for you. Among the conditions that the GC uses to determine when to run are the amount of memory allocated and the frequency of memory allocations. More allocations mean more pressure on the GC, causing it to run more often. That's incredibly inefficient.

Instead, promote the `Font` object from a local variable to a member variable. Reuse the same font each time you paint the window:

```
private readonly Font myFont =
    new Font("Arial", 10.0f);

protected override void OnPaint(PaintEventArgs e)
{
    e.Graphics.DrawString(DateTime.Now.ToString(),
        myFont, Brushes.Black, new PointF(0, 0));
    base.OnPaint(e);
}
```

Your program no longer creates garbage with every paint event. The garbage collector does less work. Your program runs just a little faster. When you elevate a local variable that implements `IDisposable` to a member variable, such as the font in this example, you need to implement `IDisposable` in your class. Item 17 explains how to properly do just that.

You should promote local variables to member variables when they are reference types (value types don't matter) and they will be used in routines that are called frequently. The font in the paint routine is an excellent example. Only local variables in routines that are frequently accessed are good candidates. Infrequently called routines are not. You're trying to avoid creating the same objects repeatedly, not turn every local variable into a member variable.

The static property `Brushes.Black` used earlier illustrates another technique that you should use to avoid repeatedly allocating similar objects. Create static member variables for commonly used instances of the reference types you need. Consider the black brush used earlier as an example. Every time you need to draw something in your window using the color black, you need a black brush. If you allocate a new one every time you draw anything, you create and destroy a huge number of black brushes during the course of a program. The first approach of creating a black brush as a member of each of your types helps, but it doesn't go far enough. Programs might create dozens of windows and controls and would create dozens of black brushes. The .NET Framework designers anticipated this and created a single black brush for you to reuse whenever you need it. The `Brushes` class contains a number of static `Brush` objects, each with a different common color. Internally, the `Brushes` class uses a lazy evaluation algorithm to create only those brushes you request. A simplified implementation looks like this:

```
private static Brush blackBrush;
public static Brush Black
{
    get
    {
        if (blackBrush == null)
            blackBrush = new SolidBrush(Color.Black);
        return blackBrush;
    }
}
```

The first time you request a black brush, the `Brushes` class creates it. The `Brushes` class keeps a reference to the single black brush and returns that same handle whenever you request it again. The end result is that you create one black brush and reuse it forever. Furthermore, if your application does not need a particular resource—say, the lime green brush—it never gets created. The framework provides a way to limit the objects created to the minimum set you need to accomplish your goals. Consider that technique in your programs. On the positive side, you create fewer objects. On the minus side, this may cause objects to be in memory for longer than necessary. It can even mean not being able to dispose of unmanaged resources because you can't know when to call the `Dispose()` method.

You've learned two techniques to minimize the number of allocations your program performs as it goes about its business. You can promote often-used local variables to member variables. You can use dependency injection to create and reuse objects that represent common instances of a given type. The last technique involves building the final value for immutable types. The `System.String` class is immutable: After you construct a string, the contents of that string cannot be modified. Whenever you write code that appears to modify the contents of a string, you are actually creating a new string object and leaving the old string object as garbage. This seemingly innocent practice:

```
string msg = "Hello, ";
msg += thisUser.Name;
msg += ". Today is ";
msg += System.DateTime.Now.ToString();
```

is just as inefficient as if you had written this:

```
string msg = "Hello, ";
// Not legal, for illustration only:
string tmp1 = new String(msg + thisUser.Name);
msg = tmp1; // "Hello " is garbage.
string tmp2 = new String(msg + ". Today is ");
msg = tmp2; // "Hello <user>" is garbage.
string tmp3 = new String(msg + DateTime.Now.ToString());
msg = tmp3; // "Hello <user>. Today is " is garbage.
```

The strings `tmp1`, `tmp2`, and `tmp3` and the originally constructed `msg ("Hello")` are all garbage. The += operator on the `string` class creates a new string object and returns that string. It does not modify the existing string by concatenating the characters to the original storage. For simple constructs such as the previous one, you should use interpolated strings:

```
string msg = string.Format("Hello, {0}. Today is {1}",
        thisUser.Name, DateTime.Now.ToString());
```

For more complicated string operations, you can use the `StringBuilder` class:

```
StringBuilder msg = new StringBuilder("Hello, ");
msg.Append(thisUser.Name);
msg.Append(". Today is ");
msg.Append(DateTime.Now.ToString());
string finalMsg = msg.ToString();
```

The example above is simple enough that you'd use string interpolation (see Item 4). Use `StringBuilder` when the logic needed to build the final string is too complex for string interpolation. `StringBuilder` is the mutable string class used to build an immutable string object. It provides facilities for mutable strings that let you create and modify text data before you construct an immutable string object. Use `StringBuilder` to create the final version of a string object. More importantly, learn from that design idiom. When your designs call for immutable types, consider creating builder objects to facilitate the multiphase construction of the final object. That provides a way for users of your class to construct an object in steps, yet maintain the immutability of your type.

The garbage collector does an efficient job of managing the memory that your application uses. But remember that creating and destroying heap objects still takes time. Avoid creating excessive objects; don't create what you don't need. Also avoid creating multiple objects of reference types in local functions. Instead, consider promoting local variables to member variables, or create static objects of the most common instances of your types. Finally, consider creating mutable builder classes for immutable types.

## Item 16: Never Call Virtual Functions in Constructors

Virtual functions exhibit strange behaviors during the construction of an object. An object is not completely created until all constructors have executed. In the meantime, virtual functions may not behave the way you'd like or expect. Examine the following simple program:

```
class B
{
    protected B()
    {
        VFunc();
    }
```

```
    protected virtual void VFunc()
    {
        Console.WriteLine("VFunc in B");
    }
}

class Derived : B
{
    private readonly string msg = "Set by initializer";

    public Derived(string msg)
    {
        this.msg = msg;
    }

    protected override void VFunc()
    {
        Console.WriteLine(msg);
    }

    public static void Main()
    {
        var d = new Derived("Constructed in main");
    }
}
```

What do you suppose gets printed—“Constructed in main,” “VFunc in B,” or “Set by initializer”? Experienced C++ programmers would say, “VFunc in B.” Some C# programmers would say, “Constructed in main.” But the correct answer is “Set by initializer.”

The base class constructor calls a virtual function that is defined in its class but overridden in the derived class. At runtime, the derived class version gets called. After all, the object’s runtime type is `Derived`. The C# language definition considers the derived object completely available, because all the member variables have been initialized by the time any constructor body is entered. After all, all the variable initializers have executed. You had your chance to initialize all variables. But this doesn’t mean that you have necessarily initialized all your member variables to the value you want. Only the variable initializers have executed; none of the code in any derived class constructor body has had the chance to do its work.

No matter what, some inconsistency occurs when you call virtual functions while constructing an object. The C++ language designers decided that virtual functions should resolve to the runtime type of the object being constructed. They decided that an object's runtime type should be determined as soon as the object is created.

There is logic behind this. For one thing, the object being created is a `Derived` object; every function should call the correct override for a `Derived` object. The rules for C++ are different here: The runtime type of an object changes as each class's constructor begins execution. Second, this C# language feature avoids the problem of having a null method pointer in the underlying implementation of virtual methods when the current type is an abstract base class. Consider this variant base class:

```
abstract class B
{
    protected B()
    {
        VFunc();
    }

    protected abstract void VFunc();
}

class Derived : B
{
    private readonly string msg = "Set by initializer";

    public Derived(string msg)
    {
        this.msg = msg;
    }

    protected override void VFunc()
    {
        Console.WriteLine(msg);
    }
    public static void Main()
    {
        var d = new Derived("Constructed in main");
    }
}
```

The sample compiles, because `B` objects aren't created, and any concrete derived object must supply an implementation for `VFunc()`. The C# strategy of calling the version of `VFunc()` matching the actual runtime type is the only possibility of getting anything except a runtime exception when an abstract function is called in a constructor. Experienced C++ programmers will recognize the potential runtime error if you use the same construct in that language. In C++, the call to `VFunc()` in the `B` constructor would crash.

Still, this simple example shows the pitfalls of the C# strategy. The `msg` variable is immutable. It should have the same value for the entire life of the object. Because of the small window of opportunity when the constructor has not yet finished its work, you can have different values for this variable: one set in the initializer, and one set in the body of the constructor. In the general case, any number of derived class variables may remain in the default state, as set by the initializer or by the system. They certainly don't have the values you thought, because your derived class's constructor has not executed.

Calling virtual functions in constructors makes your code extremely sensitive to the implementation details in derived classes. You can't control what derived classes do. Code that calls virtual functions in constructors is very brittle. The derived class must initialize all instance variables properly in variable initializers. That rules out quite a few objects: Most constructors take some parameters that are used to set the internal state properly. So you could say that calling a virtual function in a constructor mandates that all derived classes define a default constructor, and no other constructor. But that's a heavy burden to place on all derived classes. Do you really expect everyone who ever uses your code to play by those rules? I didn't think so. There is very little gain, and lots of possible future pain, from playing this game. In fact, this situation will work so rarely that it's included in the FxCop and Static Code Analyzer tools bundled with Visual Studio.

## Item 17: Implement the Standard Dispose Pattern

We've discussed the importance of disposing of objects that hold unmanaged resources. Now it's time to cover how to write your own resource management code when you create types that contain resources other than memory. A standard pattern is used throughout the .NET Framework for disposing of unmanaged resources. The users of your type will expect you to follow this standard pattern. The standard dispose idiom frees your unmanaged resources using the `IDisposable` interface when clients remember, and it uses the finalizer defensively when clients forget. It works with the garbage collector to ensure that your objects pay the performance penalty associated with finalizers only when necessary. This is the right way to handle unmanaged resources, so it pays to understand it thoroughly. In practice, unmanaged resources in .NET can be accessed through a class derived from `System.Runtime.Interop.SafeHandle`, which implements the pattern described here correctly.

The root base class in the class hierarchy should do the following:

- It should implement the `IDisposable` interface to free resources.
- It should add a finalizer as a defensive mechanism if and only if your class directly contains an unmanaged resource.
- Both `Dispose` and the finalizer (if present) delegate the work of freeing resources to a virtual method that derived classes can override for their own resource management needs.

The derived classes need to

- Override the virtual method only when the derived class must free its own resources
- Implement a finalizer if and only if one of its direct member fields is an unmanaged resource
- Remember to call the base class version of the function

To begin, your class must have a finalizer if and only if it directly contains unmanaged resources. You should not rely on clients to always call the `Dispose()` method. You'll leak resources when they forget. It's their fault for not calling `Dispose`, but you'll get the blame. The only way you can guarantee that unmanaged resources get freed properly is to create a finalizer. So if and only if your type contains an unmanaged resource, create a finalizer.

When the garbage collector runs, it immediately removes from memory any garbage objects that do not have finalizers. All objects that have finalizers remain in memory. These objects are added to a finalization queue, and the GC runs the finalizers on those objects. After the finalizer thread has finished its work, the garbage objects can usually be removed from memory. They are bumped up a generation because they survived collection. They are also marked as not needing finalization because the finalizers have run. They will be removed from memory on the next collection of that higher generation. Objects that need finalization stay in memory for far longer than objects without a finalizer. But you have no choice. If you're going to be defensive, you must write a finalizer when your type holds unmanaged resources. But don't worry about performance just yet. The next steps ensure that it's easier for clients to avoid the performance penalty associated with finalization.

Implementing `IDisposable` is the standard way to inform users and the runtime system that your objects hold resources that must be released in a timely manner. The `IDisposable` interface contains just one method:

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

The implementation of your `IDisposable.Dispose()` method is responsible for four tasks:

1. Freeing all unmanaged resources.
2. Freeing all managed resources (this includes unhooking events).
3. Setting a state flag to indicate that the object has been disposed of. You need to check this state and throw `ObjectDisposed` exceptions in your public members if any get called after disposing of an object.
4. Suppressing finalization. You call `GC.SuppressFinalize(this)` to accomplish this task.

You accomplish two things by implementing `IDisposable`: You provide the mechanism for clients to release all managed resources that you hold in a timely fashion, and you give clients a standard way to release all unmanaged resources. That's quite an improvement. After you've implemented `IDisposable` in your type, clients can avoid the finalization cost. Your class is a reasonably well-behaved member of the .NET community.

But there are still holes in the mechanism you've created. How does a derived class clean up its resources and still let a base class clean up as well? If derived classes override finalize or add their own implementation of `IDisposable`, those methods must call the base class; otherwise, the base class doesn't clean up properly. Also, finalize and `Dispose` share some of the same responsibilities; you have almost certainly duplicated code between the finalize method and the `Dispose` method. Overriding interface functions does not always work the way you'd expect. Interface functions are not virtual by default. We need to do a little more work to address these concerns. The third method in the standard dispose pattern, a protected virtual helper function, factors out these common tasks and adds a hook for derived classes to free resources they allocate. The base class contains the code for the core interface. The virtual function provides the hook for derived classes to clean up resources in response to `Dispose()` or finalization:

```
protected virtual void Dispose(bool isDisposing)
```

This overloaded method does the work necessary to support both finalize and `Dispose`, and because it is virtual, it provides an entry point for all derived classes. Derived classes can override this method, provide the proper implementation to clean up their resources, and call the base class version. You clean up managed and unmanaged resources when `isDisposing` is `true`, and you clean up only unmanaged resources when `isDisposing` is `false`. In both cases, call the base class's `Dispose(bool)` method to let it clean up its own resources.

Here is a short sample that shows the framework of code you supply when you implement this pattern. The `MyResourceHog` class shows the code to implement `IDisposable` and create the virtual `Dispose` method:

```
public class MyResourceHog : IDisposable
{
    // Flag for already disposed
    private bool alreadyDisposed = false;

    // Implementation of IDisposable.
    // Call the virtual Dispose method.
    // Suppress Finalization.
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    // Virtual Dispose method
    protected virtual void Dispose(bool isDisposing)
    {
        // Don't dispose more than once.
        if (alreadyDisposed)
            return;
        if (isDisposing)
        {
            // elided: free managed resources here.
        }
```

```
            // elided: free unmanaged resources here.
            // Set disposed flag:
            alreadyDisposed = true;
        }

        public void ExampleMethod()
        {
            if (alreadyDisposed)
                throw new
                    ObjectDisposedException("MyResourceHog",
                    "Called Example Method on Disposed object");
            // remainder elided.
        }
    }
```

If a derived class needs to perform additional cleanup, it implements the protected `Dispose` method:

```
public class DerivedResourceHog : MyResourceHog
{
    // Have its own disposed flag.
    private bool disposed = false;

    protected override void Dispose(bool isDisposing)
    {
        // Don't dispose more than once.
        if (disposed)
            return;
        if (isDisposing)
        {
            // TODO: free managed resources here.
        }
        // TODO: free unmanaged resources here.

        // Let the base class free its resources.
        // Base class is responsible for calling
        // GC.SuppressFinalize( )
        base.Dispose(isDisposing);

        // Set derived class disposed flag:
        disposed = true;
    }
}
```

Notice that both the base class and the derived class contain a flag for the disposed state of the object. This is purely defensive. Duplicating the flag encapsulates any possible mistakes made while disposing of an object to only the one type, not all types that make up an object.

You need to write `Dispose` and finalizers defensively. They must be idempotent. `Dispose()` may be called more than once, and the effect should be the same as calling them exactly once. Disposing of objects can happen in any order. You will encounter cases in which one of the member objects in your type is already disposed of before your `Dispose()` method gets called. You should not view that as a problem because the `Dispose()` method can be called multiple times. Note that `Dispose()` is the exception to the rule of throwing an `ObjectDisposedException` when public methods are called on an object that has been disposed of. If it's called on an object that has already been disposed of, it does nothing. Finalizers may run when references have been disposed of, or have never been initialized. Any object that you reference is still in memory, so you don't need to check null references. However, any object that you reference might be disposed of. It might also have already been finalized.

You'll notice that neither `MyResourceHog` nor `DerivedResourceHog` contains a finalizer. The example code I wrote does not directly contain any unmanaged resources. Therefore, a finalizer is not needed. That means the example code never calls `Dispose(false)`. That's the correct pattern. Unless your class directly contains unmanaged resources, you should not implement a finalizer. Only those classes that directly contain an unmanaged resource should implement the finalizer and add that overhead. Even if it's never called, the presence of a finalizer does introduce a rather large performance penalty for your types. Unless your type needs the finalizer, don't add it. However, you should still implement the pattern correctly so that if any derived classes do add unmanaged resources, they can add the finalizer and implement `Dispose(bool)` in such a way that unmanaged resources are handled correctly.

This brings me to the most important recommendation for any method associated with disposal or cleanup: You should be releasing resources only. Do not perform any other processing during a dispose method. You can introduce serious complications to object lifetimes by performing other processing in your `Dispose` or finalize methods. Objects are born when you construct them, and they die when the garbage collector reclaims them. You can consider them comatose when your program can no longer access them. If you can't reach an object, you can't call any of its methods. For all intents and purposes, it is dead. But objects that have finalizers get to breathe a last breath before they are declared dead. Finalizers should do nothing but clean up unmanaged resources. If a finalizer somehow makes an object reachable again, it has been resurrected. It's alive and not well, even though it has awoken from a comatose state. Here's an obvious example:

```
public class BadClass
{
    // Store a reference to a global object:
    private static readonly List<BadClass> finalizedList =
        new List<BadClass>();
    private string msg;

    public BadClass(string msg)
    {
        // cache the reference:
        msg = (string)msg.Clone();
    }

    ~BadClass()
    {
        // Add this object to the list.
        // This object is reachable, no
        // longer garbage. It's Back!
        finalizedList.Add(this);
    }
}
```

When a `BadClass` object executes its finalizer, it puts a reference to itself on a global list. It has just made itself reachable. It's alive again! The number of problems you've just introduced would make anyone cringe. The object has been finalized, so the garbage collector now believes there is no need to call its finalizer again. If you actually need to finalize a resurrected object, it won't happen. Second, some of your resources might not be available. The GC will not remove from memory any objects that are reachable only by objects in the finalizer queue, but it might have already finalized them. If so, they are almost certainly no longer usable. Although the members that `BadClass` owns are still in memory, they will have likely been disposed of or finalized. There is no way in the language that you can control the order of finalization. You cannot make this kind of construct work reliably. Don't try.

I've never seen code that has resurrected objects in such an obvious fashion, except as an academic exercise. But I have seen code in which the finalizer attempts to do some real work and ends up bringing itself back to life when some function that the finalizer calls saves a reference to the object. The moral is to look very carefully at any code in a finalizer and, by extension, both `Dispose` methods. If that code is doing anything other than releasing resources, look again. Those actions likely will cause bugs in your program in the future. Remove those actions, and make sure that finalizers and `Dispose()` methods release resources and do nothing else.

In a managed environment, you do not need to write a finalizer for every type you create; you do it only for types that store unmanaged types or when your type contains members that implement `IDisposable`. Even if you need only the `IDisposable` interface, not a finalizer, implement the entire pattern. Otherwise, you limit your derived classes by complicating their implementation of the standard dispose idiom. Follow the standard dispose idiom I've described. That will make life easier for you, for the users of your class, and for those who create derived classes from your types.

# Index