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#

50 Specific Ways to Improve Your C#

Third Edition

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

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

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

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

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

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2016953545

Copyright © 2017 Pearson Education, Inc.

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

ISBN-10: 0-672-33787-8
To Marlene, who continues to provide inspiration and support for everything we do together.
This page intentionally left blank
Contents at a Glance

Introduction xiii
Chapter 1 C# Language Idioms 1
Chapter 2 .NET Resource Management 43
Chapter 3 Working with Generics 77
Chapter 4 Working with LINQ 133
Chapter 5 Exception Practices 221
Index 253
This page intentionally left blank
Contents

Introduction  xiii

Chapter 1  C# Language Idioms  1
  Item 1: Prefer Implicitly Typed Local Variables  1
  Item 2: Prefer readonly to const  7
  Item 3: Prefer the is or as Operators to Casts  12
  Item 4: Replace string.Format() with Interpolated Strings  19
  Item 5: Prefer FormattableString for Culture-Specific Strings  23
  Item 6: Avoid String-ly Typed APIs  26
  Item 7: Express Callbacks with Delegates  28
  Item 8: Use the Null Conditional Operator for Event Invocations  31
  Item 9: Minimize Boxing and Unboxing  34
  Item 10: Use the new Modifier Only to React to Base Class Updates  38

Chapter 2  .NET Resource Management  43
  Item 11: Understand .NET Resource Management  43
  Item 12: Prefer Member Initializers to Assignment Statements  48
  Item 13: Use Proper Initialization for Static Class Members  51
  Item 14: Minimize Duplicate Initialization Logic  53
  Item 15: Avoid Creating Unnecessary Objects  61
  Item 16: Never Call Virtual Functions in Constructors  65
  Item 17: Implement the Standard Dispose Pattern  68

Chapter 3  Working with Generics  77
  Item 18: Always Define Constraints That Are Minimal and Sufficient  79
  Item 19: Specialize Generic Algorithms Using Runtime Type Checking  85
  Item 20: Implement Ordering Relations with IComparable<T> and IComparer<T>  92
  Item 21: Always Create Generic Classes That Support Disposable Type Parameters  98
  Item 22: Support Generic Covariance and Contravariance  101
  Item 23: Use Delegates to Define Method Constraints on Type Parameters  107
  Item 24: Do Not Create Generic Specialization on Base Classes or Interfaces  112
Item 25: Prefer Generic Methods Unless Type Parameters Are Instance Fields 116
Item 26: Implement Classic Interfaces in Addition to Generic Interfaces 120
Item 27: Augment Minimal Interface Contracts with Extension Methods 126
Item 28: Consider Enhancing Constructed Types with Extension Methods 130

Chapter 4 Working with LINQ 133
Item 29: Prefer Iterator Methods to Returning Collections 133
Item 30: Prefer Query Syntax to Loops 139
Item 31: Create Composable APIs for Sequences 144
Item 32: Decouple Iterations from Actions, Predicates, and Functions 151
Item 33: Generate Sequence Items as Requested 154
Item 34: Loosen Coupling by Using Function Parameters 157
Item 35: Never Overload Extension Methods 163
Item 36: Understand How Query Expressions Map to Method Calls 167
Item 37: Prefer Lazy Evaluation to Eager Evaluation in Queries 179
Item 38: Prefer Lambda Expressions to Methods 184
Item 39: Avoid Throwing Exceptions in Functions and Actions 188
Item 40: Distinguish Early from Deferred Execution 191
Item 41: Avoid Capturing Expensive Resources 195
Item 42: Distinguish between IEnumerable and IQueryable Data Sources 208
Item 43: Use Single() and First() to Enforce Semantic Expectations on Queries 212
Item 44: Avoid Modifying Bound Variables 215

Chapter 5 Exception Practices 221
Item 45: Use Exceptions to Report Method Contract Failures 221
Item 46: Utilize using and try/finally for Resource Cleanup 225
Item 47: Create Complete Application-Specific Exception Classes 232
Item 48: Prefer the Strong Exception Guarantee 237
Item 49: Prefer Exception Filters to catch and re-throw 245
Item 50: Leverage Side Effects in Exception Filters 249

Index 253
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:

```csharp
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
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.

Letters in parentheses indicate owned references. Hashed objects are visible from application.

(B, D) has been removed from memory. Heap has been compacted.

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 \texttt{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 \texttt{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#:
\begin{verbatim}
class CriticalSection
{
   // Constructor acquires the system resource.
   public CriticalSection()
   {
      EnterCriticalSection();
   }

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

   private void ExitCriticalSection()
   {
   }
   private void EnterCriticalSection()
   {
   }
}
\end{verbatim}

// usage:
void Func()
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.
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.

**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.
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:

```csharp
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.

```csharp
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:

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

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
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:

```csharp
public class MySingleton
{
    private static readonly MySingleton theOneAndOnly = new MySingleton();
```
The singleton pattern can just as easily be written this way, in case you have more complicated logic to initialize the singleton:

```csharp
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):

```csharp
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 construc-
tors 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:

```csharp
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:

```csharp
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:

```csharp
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 ini-
tializers into each constructor body. The fact that the base class con-
structor 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 can-
not 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;
}
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:

```csharp
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:

```csharp
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
Avoid Creating Unnecessary Objects

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:

```csharp
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:

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

is just as inefficient as if you had written this:

```csharp
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:

```csharp
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:

```csharp
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:

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

```csharp
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.InteropServices.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:

```csharp
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:

```csharp
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:

```csharp
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 reclaim 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 final-
izer 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:

```csharp
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 num-
ber 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 resur-
rected 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.
This page intentionally left blank
Symbols
$ (dollar sign), interpolated strings, 20
? (question mark) operator, null conditional operator, 33–34
{} (curly brackets), readability of interpolated strings, 20
< (less-than) operator, ordering relations with IComparable, 124

Numbers
0 initialization, avoid initializer syntax in, 49

A
Abrahams, Dave, 238
Action<>, delegate form, 28
Action methods
called for every item in collection, 152
naming, 222
writing to ensure no exceptions, 189
Actions
avoid throwing exceptions in, 188–190
create new exception classes for different, 234–235
decouple iterations from, 151–157
Add() generic method, 108
AddFunc() method, generic classes, 107–108
Algorithms
create with delegate-based contracts, 109
loosen coupling with function parameters, 161–163
use runtime type checking to specialize generic, 85–92
Allocations, minimize number of program, 61–64
Anonymous types
implicitly typed local variables supporting, 1
in queries with SelectMany, 177
API signatures
define method constraints on type parameters, 107
distinguish between IEnumerable/IQueryable data sources, 208
APIs
avoid string-ly typed, 26–27
create composable (for sequences), 144–151
AppDomain, initializing static class members, 52–53
Application-specific exception classes, 232
AreEqual() method, minimizing constraints, 80–83
Arguments
generator method using, 135–139
nameof() operator for, 26–27
As operator
checking for equality on Name types, 123
prefer to casts, 12–19
.AsParallel() method, query syntax, 144
AsQueryable() method, 211–212
Assignment statements
prefer member initializers to, 48–51
support generic covariance/contravariance, 103
Backward compatibility, `IComparable` for, 93

Base classes
- calling constructor using `base()`, 59
- define minimal/sufficient constraints, 80, 83
- define with function parameters/generic methods, 160–161
- do not create generic specialization on, 112–116
- execute static initializers before static constructor on, 49, 52
- force client code to derive from, 158
- implement standard dispose pattern, 69–73
- loosen coupling using, 157–160, 163
- use `new` modifier only to react to updates of, 38–41

`BaseWidget` class, 40–41

Basic guarantee, exceptions, 238

BCL. See .NET Base Class Library (BCL)

Behavior
- compile-time vs. runtime constants, 8
- define in interfaces with extension methods, 126–130
- `IEnumerable` vs `IQueryable`, 208–212
- `nameof()` operator and consistent, 26–27
- when extension methods cause strange, 128–129

`BindingList<T>` constructor, 155–156

Bound variables
- avoid capturing expensive resources, 195–197, 204–205
- avoid modifying, 215–220
- lifetime of, 195

Boxing operations
- implement `IComparable` and, 92–93
- minimize, 34–38

`Brushes` class, minimizing number of programs, 63–64

C

C# language idioms
- avoid string-ly typed APIs, 25–27
- express callbacks with delegates, 28–31

minimize boxing and unboxing, 34–38
overview of, 1
prefer `FormattableString` for culture-specific strings, 23–25
prefer implicitly typed local variables, 1–7
prefer `is` or `as` operators to casts, 12–19
prefer `readonly` to `const`, 7–11
replace `string.format()` with interpolated strings, 19–23
use `new` modifier only to react to base class updates, 38–41
use `null` operator for event invocations, 31–34

Callbacks, express with delegates, 28–31

Captured variables
- avoid capturing expensive resources, 195–196
- avoid modifying, 215–220

Cargill, Tom, 238

Casts
- as alternative to constraints, 80–81
- `GetEnumerator()`, `ReverseEnumerator<T>` and, 89–90
- `pref er is` or `as` operators to, 12–19
- specifying constraints vs., 79
- `T` implementing/not implementing `IDisposable`, 99

Cast<T>` method, converting elements, 18–19

Catch clauses
- create application-specific exception classes, 232–237
- exception filters with side effects and, 250–251
- prefer exception filters to, 245–249

CheckEquality() method, 122–123

Circular memory, with garbage collector, 43–44

Classes
- avoid extension methods for, 163–167
- constraints on, 112
- use generic methods for nongeneric, 116–120

Close() method, `SqlConnection`, 230–231
Closed generic type, 77–79

Closures
  captured variables inside, 196–197
  compiler converting lambda expressions into, 215, 218–220
  extended lifetime of captured variables in, 195

CLR (Common Language Runtime), generics and, 77

Code conventions, used in this book, xv

Collections
  avoid creating nongeneric class/generic methods for, 120
  create set of extension methods on specific, 130
  inefficiencies of operating on entire, 144
  prefer iterator methods to returning, 133–139
  treating as covariant, 103

COMException class, exception filters for, 248

Common Language Runtime (CLR), generics and, 77

CompareTo() method, IComparable<T>, 92–95, 98

Comparison<T> delegate, ordering relations, 95

Compile-time constants
  declaring with const keyword, 8
  limited to numbers, strings, and null, 9
  prefer runtime constants to, 7–8

Compiler, 3
  adding generics and, 77
  emitting errors on anything not defined in System.Object, 80
  using implicitly typed variables with, 1–2

Components, decouple with function parameters, 157–163

Conditional expressions, string interpolation and, 21–22

Const keyword, 7–11

Constants, types of C#, 7–8

Constraints
  documenting for users of your class, 98
  on generic type parameters, 19
  must be valid for entire class, 116–117
  specifying minimal/sufficient, 79–84
  as too restrictive at first glance, 107
  transforming runtime errors into compile-time errors, 98
  type parameters and, 98
  use delegates to define method, 107–112

Constructed generic types, extension methods for, 130–132

Constructor initializers, minimize duplicate initialization logic, 53–54, 59–61

Constructors
  Exception class, 235–236
  minimize duplicated code in, 53–61
  minimize duplicated code with parameterless, 55–56
  never call virtual functions in, 65–68
  parameterless, 55–56
  static, 51–53, 61

Continuable methods, 148

Continuations, in query expressions, 173–174

Contract failures, report using exceptions, 221–225

Contravariance, generic, 101–102, 106–107

Conversions
  built-in numeric types and implicit, 3–5
  casts with generics not using operators for, 19
  foreach loops and, 16–17
  as and is vs. casts in user-defined, 13–15

Costs
  of decoupling components, 158
  extension methods and performance, 164
  of generic type definitions, 77
  memory footprint runtime, 79
  throwing exceptions and performance, 224
  use exception filters to avoid additional, 245

Coupling, loosen with function parameters, 157–163

Covariance, generic, 101–107

CreateSequence() method, 155–157, 161

Customer struct, 94, 96–98
D

Data
distinguish early from deferred execution, 190–195
throw exceptions for integrity errors, 234
treating code as, 179

Data sources, 169
IEnumerable vs. IQueryable, 208–212
lambda expressions for reusable library and, 186

Data stores, LINQ to Objects queries on, 186

Debugger, exception filters and, 251–252

Declarative code, 191

Declarative model
distinguish early from deferred execution, 192
query syntax moving program logic to, 139

Default constructor
constraint, 83–84
defined, 49

Default parameters
minimize duplicate initialization logic, 60–61
minimize duplicated code in constructors, 54–56

Defensive copy mechanism
meet strong exception guarantee with, 239
no-throw guarantee, delegate invocations and, 244–245
problem of swapping reference types, 240–241

Deferred execution
avoid capturing expensive resources, 200
composability of multiple iterator methods, 148–149
defined, 145
distinguish early from, 191–195
writing iterator methods, 145–146

Delegate signatures
define method constraints with, 107–108
loosen coupling with, 159–163

Delegate targets, no-throw guarantee for, 244–245

Delegates
captured variables inside closure and, 195, 196–197
cause objects to stay in memory longer, 43
compiler converting lambda expressions into, 215–216
define method constraints on type parameters using, 107–110
define method constraints with, 112
express callbacks with, 28–31
generic covariance/contravariance in, 105–107
in IEnumerable<T> extension methods, 209

Dependency injection, create/reuse objects, 64

Derived classes
calling virtual functions in constructors and, 66–68
implement standard dispose pattern, 69, 70–71

Deterministic finalization, not part of .NET environment, 46

Disposabe type parameters, create generic classes supporting, 98–101

Dispose() method
no-throw guarantee for exceptions, 244
resource cleanup, 225–227, 229–231
standard dispose pattern, 69–73, 75
T implementing IDisposable, 99, 100

Documentation, of constraints, 98

Duplication, minimize in initialization logic, 53–61

Dynamic typing, implicitly typed local variables vs., 2

E

Eager evaluation, 179–184

Early evaluation, 191–195

EntitySet class, GC's Mark and Compact algorithm, 44

Enumerable.Range() iterator method, 138
Enumerable.Reverse() method, 7
Enumerators, functional programming in classes with, 192
Envelope-letter pattern, 241–243
Equality relations
  classic and generic interfaces for, 122–124, 126
  ordering relations vs., 98
Equality tests, getting exact runtime type for, 18
Equals() method
  checking for equality by overriding, 123
  minimizing constraints, 82
  not needed for ordering relations, 98
Errors
  exceptions vs. return codes and, 222
  failure-reporting mechanism vs., 222
  from modifying bound variables between queries, 215–220
  use exceptions for errors causing long-lasting problems, 234
Event handlers
  causing objects to stay in memory longer, 43
  event invocation traditionally and, 31–33
  event invocation with null conditional operator and, 33–34
Events
  use null conditional operator for invocation of, 31–34
  use of callbacks for, 28
Exception filters
  leverage side effects in, 249–252
  no-throw guarantee for, 244
  prefer to catch and re-throw, 245–249
  with side effects, 251
“Exception Handling: A False Sense of Security” (Cargill), 238
Exception, new exception class must end in, 235
Exception-safe guarantees, 238
Exception translation, 237
Exceptional C++ (Sutter), 238
Exceptions
  avoid throwing in functions and actions, 188–190
  best practices, 238
  create application-specific exception classes, 232–237
  for errors causing long-lasting problems, 234
  initialize static class members and, 52–53
  leverage side effects in exception filters, 249–252
  move initialization into body of constructors for, 50–51
  nameof() operator and types of, 27
  overview of, 221
  prefer exception filters to catch and re-throw, 245–249
  prefer strong exception guarantee, 237–245
  report method contract failures with, 221–225
  resource cleanup with using and try/finally, 225–232
  thrown by Single(), 212–213
Execution semantics, 169
Expensive resources, avoid capturing, 195–208
Expression trees
  defined, 209
  IQueryable<T> using, 209
  LINQ to Objects using, 186
Expression.MethodCall node, LINQ to SQL, 186
Expressions
  conditional, 21–22
  describing code for replacement strings, 20–21
Extension methods
  augment minimal interface contracts with, 126–130
  define interface behavior with, 126
  enhance constructed generic types with, 130–132
  IEnumerable<T>, 209
  implicitly typed local variables and, 6–7
  never use same signature for multiple, 167
Extension methods (continued)
query expression pattern, 169
reuse lambda expressions in complicated queries, 187

F
Failures, report method contract, 221–225
False, exception filter returning, 249–250
Feedback, server-to-client callbacks, 28–31
Finalizers
avoid resource leaks with, 48
control unmanaged resources with, 45–46
effect on garbage collector, 46–47
implement standard dispose pattern with,
69–70, 73–75
minimize need for, 46–47
no-throw guarantee for exceptions, 244
use IDisposable interface instead of, 48
Find() method, List<T> class, 29
First() method, 212–214
FirstOrDefault() method, 213–214
Flexible, const vs. read-only, 11
Font object, 62–63
Foreach loop, conversions with casts, 16–17
FormattableString, culture-specific strings, 23–25
Func<>, delegate form, 28
Function parameters
define interfaces or creating base classes,
160–163
IEnumerable<T> extension methods using, 209
loosen coupling with, 157–158
Functional exception programming style, strong exception guarantee, 239
Functions
avoid throwing exceptions in, 188–190
decouple iterations from, 151–157
use lambda expressions, type inference and enumerators with, 192

G
Garbage collector (GC)
avoid overworking, 61–62
control managed memory with, 43–46
effect of finalizers on, 46–47
eligibility of local variables when out of scope for, 195
implement standard dispose pattern with,
69–72, 74
notify that object does not need finalization, 231
optimize using generations, 47
Generate-as-needed strategy, iterator methods, 138
Generations, garbage collector
finalizers and, 69–70
optimizing, 47
Generic contravariance
in delegates, 105–107
for generic interfaces, 105
overview of, 101–102
use of in modifier for, 107
Generic covariance
array problems, 102–103
in delegates, 105–107
in generic interfaces, 103–105
overview of, 101–102
use of out modifier for, 107
Generic interfaces, treating covariantly/contravariantly, 103–104
Generic methods
compiler difficulty resolving overloads of, 112–115
define interfaces or create base classes,
160–161
prefer unless type parameters are instance fields, 116–120
vs. base class, 115
Generic type definitions, 77
Generics
augment minimal interface contracts with extension methods, 126–130
avoid boxing and unboxing with, 35
avoid generic specialization on base class INTERFACE, 112–116
create generic classes supporting disposable type parameters, 98–101
define minimal/sufficient constraints, 79–84
enhance constructed types with extension methods, 130–132
implement classic and generic interfaces, 120–126
ordering relations with IComparable<T>/ IComparer<T>, 92–98
overview of, 77–79
prefer generic methods unless type parameters are instance fields, 116–120
specialize generic algorithms with runtime type checking, 85–92
support generic covariance/contravariance, 107–112
GetEnumerator() method, 88–90
GetHashCode() method, overriding, 123
GetHttpCode() method, exception filters for, 248–249
GetType() method, get runtime of object, 18
Greater-than (>) operator, ordering relations with IComparable, 124
GroupBy method, query expression pattern, 174
GroupJoin method, query expression pattern, 178

H

HttpException class, use exception filters for, 248–249

I

IEnumerable<T> interface
classic IEnumerable support for, 126
Enumerable.Reverse() and, 7
incompatible with ICollection, 126
specialize generic algorithms using runtime type checking, 88–89, 91

IComparable interface
courage calling code to use new version with, 126
implement IComparable<T> with, 92–95
natural ordering using, 98

IComparable<T> interface
define extension methods for, 127
implement ordering relations, 92–95, 123–124
specify constraints on generic types, 81, 83
use class constraints with, 112

IComparer<T> interface
forcing extra runtime checks, 118
implement ordering relations, 96–98

IDisposable interface
avoid creating unnecessary objects, 62–63
avoid performance drain of finalizers, 48
captured variables inside closure and, 197–198
control unmanaged resources with, 45
create generic classes supporting disposable type parameters, 98–101
implement standard dispose pattern, 69–71, 75
leak resources due to exceptions, 238
resource cleanup with using and try/finally, 225, 227–232
variable types holding onto expensive resources implementing, 196
variables implementing, 201

IEnumerable<T> interface
create stored collection, 139
define extension methods for, 127
enhance constructed types with extension methods, 130–132
generic covariance in, 104
inherits from IEnumerable, 126
IQueryable<T> data sources vs., 208–212
performance of implicitly typed locals, 5–6
prefer query syntax to loops, 140–141
query expression pattern, 169
reverse-order enumeration and, 85–87
specify constraints with, 112
use implicitly typed local variables, 1
writing iterator methods, 145–146
IEnumerator<T> interface
generic covariance in, 104
specialize generic algorithms with runtime
type checking, 85–86, 88–91

IEquatable<T> interface
minimize constraints, 82–83
use class constraints, 112

IL, or MSIL (Microsoft Intermediate
Language) types, 8–9, 77–79

IList<T> interface
classic IEnumerable support for, 126
incompatible with IList, 126
specialize generic algorithms, 87–89

Immutable types
build final value for, 64–65
strong exception guarantee for, 239–240

Imperative code
defined, 191
lose original intent of actions in, 142

Imperative model
methods in, 192
query syntax moves program logic from,
139–144

Implicit properties, avoid initializer syntax
for, 50–51

Implicitly typed local variables
declare using var, 1–2, 7
extension methods and, 6–7
numeric type problems, 3–4
readability problem, 2–3
reasons for using, 1

In (contravariant) modifier, 107

Inheritance relationships
array covariance in, 103
runtime coupling switching to use delegates
from, 163

Initialization
assignment statements vs. variable, 48–49
local variable type in statement of, 2
minimize duplication of logic in, 53–61
order of operations for object, 61
of static class members, 51–53

InnerException property, lower-level
errors and, 236–237

INotifyPropertyChanged interface,
nameof() expression, 26

Instance constants, readonly values for, 9

Interface pointer, boxing/unboxing and, 35

Interfaces
augment with extension methods, 126–130
constraints on, 112
implement generic and classic, 120–126
loosen coupling by creating/coding against,
158–159, 163
loosen coupling with delegate signatures
vs., 160
nameof() operator for, 26–27
use function parameters/generic methods
to define, 160–161

Interfaces, generic
avoid creating nongeneric class/generic
methods for, 120
avoid generic specialization for, 112–116
implement classic interfaces and, 120–126

Internationalization, prefer
FormattableString for, 25

Interpolated strings
avoid creating unnecessary objects, 64–65
boxing/unboxing of value types and, 35–36
converting to string or formattable string,
23
prefer FormattableString for culture-
specific strings, 23–25
replace string.Format() with, 19–23

InvalidCastException, caused by
foreach loops, 17

InvalidOperationException,
Single(), 213

Invoke() method, use “?” operator with,
33–34

IQueryable enumerators, 187

IQueryable<T> interface
do not parse any arbitrary method, 209
IEnumerable<T> data sources vs.,
208–212
implement query expression pattern, 169
set of operators/methods and, 209
use implicitly typed local variables, 1, 5–6
**IQueryProvider**
prefer lambda expressions to methods, 187
translating queries to T-SQL, 210
use implicitly typed local variables, 1

**Is operator**
following rules of polymorphism, 17–18
prefer to casts, 12–19

**Iterations**
  decouple from actions, predicates, and functions, 151–157
  inefficiencies for entire collections, 144
  produce final collection in one, 144

**Iterator methods**
create composable APIs for sequences, 145–151
defined, 133
not necessarily taking sequence as input parameter, 154
prefer to returning collections, 133–139
when not recommended, 139

**Iterators, defined, 145**

**J**

**Join method, query expression pattern, 178**

**Just-In-Time (JIT) compiler, generics and,** 77–79

**L**

**Lambda expressions**
  compiler converting into delegates or closures, 215–220
deferred execution using, 191–195
define methods for generic classes with, 108
express delegates with, 28–29
IEnumerable<T> using delegates for, 209
not all creating same code, 215–216
prefer to methods, 184–188
reusable queries expressed as, 132

**Language**
idioms. See C# language idioms

prefer FormattableString for culture-specific strings, 23–25
string interpolation embedded into, 20–23

**Late evaluation, 191–195**

**Lazy evaluation, 179–184, 192**

**Less-than (<) operator, order relations with IComparable, 124**

**Libraries. See also .NET Base Class Library (BCL)**
  exceptions generated from, 236–237
  string interpolation executing code from, 20–21

**LINQ**
avoid capturing expensive resources, 195–208
avoid modifying bound variables, 215–220
avoid throwing exceptions in functions/actions, 188–190
built on delegates, 29
create composable APIs for sequences, 144–151
decouple iterations from actions, predicates, and functions, 151–154
distinguish early from deferred execution, 190–195
generate sequence items as requested, 154–157
how query expressions map to method calls, 167–179
IEnumerable vs. IQueryable, 208–212
loosen coupling by using function parameters, 157–163
never overload extension methods, 163–167
overview of, 133
prefer iterator methods to returning collections, 133–139
prefer lambda expressions to methods, 184–188
prefer lazy vs. eager evaluation in queries, 179–184
prefer query syntax to loops, 139–144
use queries in interpolated strings, 22
use Single() and First() to enforce semantic expectations on queries, 212–214

**LINQ to Objects, 186–187, 209**
LINQ to SQL

distinguish early from deferred execution, 194–195
IEnumerable vs. IQueryable, 208
IQueryable<T> implementation of, 210
prefer lambda expressions to methods, 186–187
string.LastIndexOf() parsed by, 211–212
List.ForEach() method, List<T> class, 29
List.RemoveAll() method signature, 159
List<T> class, methods using callbacks, 29
Local type inference
can create difficulties for developers, 5
compiler making best decision in, 4
static typing unaffected by, 2
Local variables. See also Implicitly typed local variables
avoid capturing expensive resources, 204–207
avoid string-ly typed APIs, 27
eligibility for garbage collection, 195–196
prefer exception filters to catch and re-throw, 246–247
promoting to member variables, 62–65
use null conditional operator for event invocation, 32–33
when lambda expressions access, 218–220
when lambda expressions do not access, 216–218
writing and disposing of, 99–101
Localizations, prefer FormattableString for, 25
Logging, of exceptions, 250–251
Logic, minimize duplicate initialization, 53–61
Loops, prefer query syntax to, 139–144

M
Managed environment, 75
copying heap-allocated objects in, 240
memory management with garbage collector in, 43–44, 48
Managed heap, memory management for, 44–45
Mapping query expressions to method calls, 167–178
Mark and Compact algorithm, garbage collector, 43–44
Max() method, 183
Member initializers, prefer to assignment statements, 48–51
Member variables
avoid creating unnecessary objects, 62–65
call virtual functions in constructors, 66
garbage collector generations for, 47
generic classes using instance of type parameters as, 99–100
initialize once during construction, 61
initialize where declared, 48–51
never call virtual functions in constructors, 66
static, 63–65
when lambda expressions access, 218
Memory management, .NET, 43–48
Method calls, mapping query expressions to, 167–178
Method parameters
covariant type parameters as, 105
how compiler treats in lambda expressions, 220
Method signatures
augment minimal interface contracts with extension methods, 127, 129, 131
decouple iterations from actions, predicates, and functions, 152
implement classic and generic interfaces, 126
loosen coupling using function parameters, 159
map query expressions to method calls, 167–169, 171
prefer implicitly typed local variables, 2
prefer is or as operators to casts, 12
return codes as part of, 222
Methods
culture-specific strings with FormattableString, 24–25
declare compile-time vs. runtime constants, 8
distinguish early from deferred execution, 190–195
extension. See Extension methods
generic. See Generic methods
iterator. See Iterator methods
prefer lambda expressions to, 184–188
in query expression pattern. See Query
expression pattern
readability of implicitly typed local
variables and names of, 2–3
use exceptions to report contract failures of,
221–225
use new modifier to incorporate new
version of base class, 39–41
Min() method, 183
MSIL, or IL (Microsoft Intermediate
Language) types, 8–9, 77–79
Multicast delegates
all delegates as, 29–30
event invocation with event handlers and,
31–32
no-throw guarantee in delegate targets and,
244

N
Named parameters, 11
nameof() operator, avoid string-ly typed
APIs, 26
Names
checking for equality on, 123
importance of method, 222–223
variable type safety vs. writing full type, 1–2
Namespaces
nameof() operator for, 27
never overload extension methods in, 164–167
Nested loops, prefer query syntax to, 141
.NET 1.x collections, avoid boxing/unboxing
in, 36–37
.NET Base Class Library (BCL)
convert elements in sequence, 18–19
delegate definition updates, 105
ForAll implementation, 140
implement constraints, 112
loosen coupling with function parameters,
162–163
use generic collections in 2.0 version of, 36
.NET resource management
avoid creating unnecessary objects, 61–65
implement standard dispose pattern, 68–75
minimize duplicate initialization logic,
53–61
never call virtual functions in constructors,
65–68
overview of, 43
prefer member initializers to assignment
statements, 48–51
understanding, 43–48
use proper initialization for static class
members, 51–53
New() constraint, 83–84
implement IDisposable, 100–101
requires explicit parameterless constructors,
55–56
New modifier, use only to react to base class
updates, 38–41
No-throw guarantee, exceptions, 238, 244
Nonvirtual functions, avoid new modifier to
redefine, 39
NormalizeValues() method,
BaseWidget class, 40–41
Null operator
avoid initializer syntax in, 49
compile-time constants limited to, 9
event invocation with, 31–34
for queries returning zero or one element, 213
use with as operator vs. casts, 13
NullReferenceException, 31–32
Numbers
compile-time constants limited to, 9
generate sequence items as requested,
154–157
Numeric types
explicitly declaring, 7
problems with implicitly declaring, 3–5
provide generic specialization for, 115
Objects
- Avoid creating unnecessary, 61–65
- Avoid initializer syntax for multiple initializations of same, 49–50
- Manage resource usage/lifetimes of, 195
- Never call virtual functions in construction of, 65–68
- Order of operations for initialization of, 61
- Ownership decisions, 44

OnPaint() method, 62–63

Open generic type, 77

Optional parameters, 11

OrderBy method
  - Needs entire sequence for operation, 183
  - Query expression pattern, 172–173

OrderByDescending method, query expression pattern, 172–173

Ordering relations
  - With IComparable<T>/IComparer<T>, 92–98
  - Implement classic and generic interfaces for, 123–124, 126
  - As independent from equality relations, 124

Out (covariance) modifier, 107

Overloads
- Avoid extension method, 163–167
- Compiler difficulty in resolving generic method, 112–115
- Minimize duplicated code with constructor, 55–56, 60–61

Performance
  - Const vs. read-only trade-offs, 11
  - Cost of boxing and unboxing, 34
  - Exception filter effects on program, 248
  - IEnumerable vs. IQueryable, 208–212
  - Penalties of relying on finalizers, 46
  - Produce final collection in one iteration for, 144

Polymorphism, is operator following rules of, 17–18

Predicates, decouple iterations from, 151–157

Predicate<T>, delegate form, 28

Private methods, use of exceptions, 222–223

Public methods, use of exceptions, 222–223

Queries. See also LINQ
  - Cause objects to stay in memory longer, 43
  - Compiler converting into delegates or closures, 215–216
  - Designed to return one scalar value, 214
  - Execute in parallel using query syntax, 144
  - Generate next value only, 198
  - IEnumerable vs. IQueryable, 209
  - Implement as extension methods, 131–132
  - Prefer lazy evaluation to eager evaluation, 179–184

Query expression pattern
  - Eleven methods of, 169–170
  - GroupBy method, 174
  - GroupJoin method, 178
  - Join method, 178
  - OrderBy method, 172–173
  - OrderByDescending method, 172–173
  - Select method, 171–172
  - SelectMany method, 174–177
  - ThenBy method, 172–173
  - ThenByDescending method, 172–173
  - Where method, 169–170

Query expressions
  - Deferred execution using, 191–195

P

Parameterless constructors, minimize duplicated code with, 55–56

Parameters. See also Type parameters
  - Default, 54–56, 60–61
  - Function, 157–163, 209
  - Method, 105, 220
  - Optional, 11

Params array, 20–21
as lazy, 181
mapping to method calls, 167–178
Query syntax, prefer to loops, 139–144

R
Re-throw exceptions, exception filters vs., 245–249
Readability
implicitly typed local variables and, 2–3, 7
interpolated strings improving, 20, 23
query syntax improving, 140
ReadNumbersFromStream() method, 198–199
Readonly values
assigned/resolved at runtime, 9
avoid creating unnecessary objects, 62
avoid modifying bound variables, 216–219
implement ordering relations, 92, 94, 96
implement standard dispose pattern, 74
initialization for static class members, 51–52
never call virtual functions in constructors, 66–67
prefer iterator methods to returning collections, 136
prefer to const, 7–11

ReadNumbersFromStream() method
Refactored lambda expressions, unusability of, 185
Reference types
boxing converting value types to, 34–35
create memory allocations, 62
define constraints that are minimal/sufficient, 83
foreach each statement using casts to support, 17
iterator methods for sequences containing, 150
in local functions, avoid creating multiple objects of, 65
program errors from swapping, 240–241, 243–245
promote local variables to member variables when they are, 63
Replacement strings, using expressions for, 20
Resources, 225–232
avoid capturing expensive, 195–208
avoid leaking in face of exceptions, 238
management of. See .NET resource management
Return values, multicast delegates and, 30
Reusability, produce final collection in one iteration as sacrifice of, 144
RevenueComparer class, 96–98
ReverseEnumerable constructor, 85–86
ReverseEnumerator<T> class, 87–90
ReverseStringEnumerator class, 89–90
Revisions, tracking with compile-time constants, 10–11
Runtime
catch clauses for types of exceptions at, 233–234
constants, 7–9
define minimal constraints at, 79–80
delegates enable use of callbacks at, 31
evaluate compatibility at, 9–10
get type of object at, 18
readonly values resolved at, 9
testing, vs. using constraints, 80
type checking at, 12
working with generics, 77–79
Runtime checks
to determine whether type implements IComparer<T>, 118
for generic methods, 115–116
specialize generic algorithms using, 85–92

S
SafeUpdate() method, strong exception guarantee, 242, 244
Sealed keyword, adding to IDisposable, 99, 100
Select clause, query expression pattern, 171, 174–175
Select method, query expression pattern, 171–172
SelectMany method, query expression pattern, 174–177
Semantic expectations, enforce on queries, 212–214
Sequences
- create composable APIs for, 144–151
- generate as requested, 154–157
- generate using iterator methods, 133–139
- when not to use iterator methods for, 139
Servers, callbacks providing feedback to clients from, 28–31
Side effects, in exception filters, 249–252
Single() method, enforce semantic expectations on queries, 212–214
SingleOrDefault() method, queries returning zero/one element, 213–214
Singleton pattern, initialize static class members, 51–53
SqlConnection class, freeing resources, 230–231
Square() iterator method, 149–150
Standard dispose pattern, 68–75
State, ensuring validity of object, 238
Static analysis, nameof() operator for, 27
Static class members, proper initialization for, 51–53
Static constants, as compile-time constants, 9
Static constructors, 51–53, 61
Static initializers, 51–53
Static member variables, 63–65
Static typing
- local type inference not affecting, 2
- overview of, 12
String class, ReverseEnumerator<T>, 89–90
String interpolation. See Interpolated strings
StringBuilder class, 65
String.Format() method, 19–23
Stringly-typed APIs, avoid, 26–27
Strings
- compile-time constants limited to, 9
- FormattableString for culture-specific, 23–25
- nesting, 22
- replace string.Format() with interpolated, 19–23
- specifying for attribute argument with nameof, 27
- use string interpolation to construct, 23–25
Strong exception guarantee, 238–240
Strongly typed public overload, implement IComparable, 94
Sutter, Herb, 238
Symbols, nameof() operator for, 26–27
System.Exception class, derive new exceptions, 234–235
System.Linq.Enumerable class
- extension methods, 126–127, 130–132
- prefer lazy evaluation to eager in queries, 183
- query expression pattern, 169
System.Linq.Queryable class, query expression pattern, 169
System.Object
- avoid substituting value types for, 36, 38
- boxing/unboxing of value types and, 34–36
- check for equality using, 122
- IComparable taking parameters of, 92–93
- type parameter constraints and, 80
T
T local variable, implement IDisposable, 99–101
T-SQL, IQueryProvider translating queries into, 210
Task-based asynchronous programming, exception filters in, 248
Templates, generics vs. C++, 77
Test methods, naming, 222–223

Text
prefer `FormattableString` for culture-specific strings, 23–25
replace `string.Format()` with interpolated strings, 19–23

`ThenBy()` method, query expression pattern, 172–173

`ThenByDescending()` method, query expression pattern, 172–173

Throw statement, exception classes, 234

`ToArray()` method, 139, 184

`ToList()` method, 139, 184

Translation, from query expressions to method calls, 170–171

`TrueForAll()` method, `List<T>` class, 29

Try/catch blocks, no-throw guarantees, 244

Try/finally blocks, resource cleanup, 225–232

Type inference
define methods for generic classes, 108
functional programming in classes, 192

Type parameters
closed generic type for, 26, 77
create generic classes supporting disposable, 98–101
create generic classes vs. set of generic methods, 117–118
define method constraints on, 107–112
generic classes using instance of, 99–100
minimal/sufficient constraints for, 79–84, 98
reuse generics by specifying new, 85
weigh necessity for class constraints, 19
when not to prefer generic methods over, 116–120
wrap local instances in `using` statement, 99

Type variance, covariance and contravariance, 101–102

`TypeInitializationException`, initialize static class members, 53

Types, `nameof()` operator, 26–27

**U**

Unboxing operations
`IComparable` interface and, 92–93
minimize, 34–38

Unique() iterator method
composability of multiple iterator methods, 149–150
as continuation method, 148
create composable APIs for sequences, 146–148

Unmanaged resources
control, 43
control with finalizers, 45–46
explicitly release types that use, 225–232
implement standard dispose pattern for, 68–75
use `IDisposable` interface to free, 48, 69

Updates, use `new` modifier in base class, 38–41

UseCollection() function, 17

User-defined conversion operators, 13–16

User-defined types, casting, 13–14

Using statement
ensure `Dispose()` is called, 225–227
never overload extension methods, 166–167
resource cleanup utilizing, 225–232
wrap local instances of type parameters in, 99

Utility class, use generic methods vs. generic class, 116–120

**V**

Value types
avoid substituting for `System.Object`, 36, 38
cannot be set to `null`, 100
cost of boxing and unboxing, 38
create immutable, 37
minimize boxing and unboxing of, 34–38

`Var` declaration, 1–3, 7

Variables
avoid modifying bound, 215–220
captured, 195–197, 215–220
Variables (continued)
hold onto expensive resources, 196
implicitly typed local. See Implicitly typed
local variables
lifetime of bound, 195
local. See Local variables
member. See Member variables
nameof() operator for, 26–27
static member, 63–65

Virtual functions
implement standard dispose pattern, 70–71
never call in constructors, 65–68

W
When keyword, exception filters, 244–246
Where method
needs entire sequence for operation, 183
query expression pattern, 169–170

Windows Forms, cross-thread marshalling in, 29
Windows paint handler, avoid allocating
GDI objects in, 62–63
Windows Presentation Foundation (WPF),
cross-thread marshalling in, 29
WriteMessage(MyBase b), 113–115
WriteMessage<T>(T obj), 113–115

Y
Yield return statement
create composable APIs for sequences, 145–150
generate sequence with, 154–157
write iterator methods, 133–134, 136, 138

Z
Zip
create composable APIs for sequences, 149–150
delegates defining method constraints on
type parameters, 109–110
loosen coupling with function parameters,
160–161