Effective SOFTWARE DEVELOPMENT SERIES

Scott Meyers, Consulting Editor

MORE *Effective* C#

50 Specific Ways to Improve Your C#

Bill Wagner

Praise for *More Effective C#*

"Shining a bright light into many of the dark corners of C# 3.0, this book not only covers the 'how,' but also the 'why,' arming the reader with many field-tested methods for wringing the most from the new language features, such as LINQ, generics, and multithreading. If you are serious about developing with the C# language, you need this book."

—Bill Craun, Principal Consultant, Ambassador Solutions, Inc.

"*More Effective C#* is an opportunity to work beside Bill Wagner. Bill leverages his knowledge of C# and distills his expertise down to some very real advice about programming and designing applications that every serious Visual C# user should know. *More Effective C#* is one of those rare books that doesn't just regurgitate syntax, but teaches you *how* to use the C# language."

—Peter Ritchie, Microsoft MVP: Visual C#

"*More Effective C#* is a great follow-up to Bill Wagner's previous book. The extensive C# 3.0 and LINQ coverage is extremely timely!"

—Tomas Restrepo, Microsoft MVP: Visual C++, .NET, and Biztalk Server

"As one of the current designers of C#, it is rare that I learn something new about the language by reading a book. *More Effective C#* is a notable exception. Gently blending concrete code and deep insights, Bill Wagner frequently makes me look at C# in a fresh light—one that really makes it shine. *More Effective C#* is at the surface a collection of very useful guidelines. Look again. As you read through it, you'll find that you acquire more than just the individual pieces of advice; gradually you'll pick up on an approach to programming in C# that is thoughtful, beautiful, and deeply pleasant. While you can make your way willy-nilly through the individual guidelines, I do recommend reading the whole book—or at least not skipping over the chapter introductions before you dive into specific nuggets of advice. There's perspective and insight to be found there that in itself can be an important guide and inspiration for your future adventures in C#."

—Mads Torgersen, Program Manager, Visual C#, Microsoft

"Bill Wagner has written an excellent book outlining the best practices for developers who work with the C# language. By authoring *More Effective C#*, he has again established himself as one of the most important voices in the C# community. Many of us already know how to use C#. What we need is advice on how to hone our skills so that we can become wiser programmers. There is no more sophisticated source of information on how to become a first-class C# developer than Bill Wagner's book. Bill is intelligent, thoughtful, experienced, and skillful. By applying the lessons from this book to your own code, you will find many ways to polish and improve the work that you produce."

—Charlie Calvert, Community Program Manager, Visual C#, Microsoft

This page intentionally left blank

More Effective C#

Effective SOFTWARE DEVELOPMENT SERIES

Scott Meyers, Consulting Editor

The **Effective Software Development Series** provides expert advice on all aspects of modern software development. Books in the series are well written, technically sound, of lasting value, and tractable length. Each describes the critical things the experts almost always do—or almost always avoid doing—to produce outstanding software.

Scott Meyers (author of the *Effective C++* books and CD) conceived of the series and acts as its consulting editor. Authors in the series work with Meyers and with Addison-Wesley Professional's editorial staff to create essential reading for software developers of every stripe.

TITLES IN THE SERIES

Elliotte Rusty Harold, *Effective XML: 50 Specific Ways to Improve Your XML* 0321150406

Ted Neward, *Effective Enterprise Java* 0321130006

Diomidis Spinellis, *Code Reading: The Open Source Perspective* 0201799405

Diomidis Spinellis, *Code Quality: The Open Source Perspective* 0321166078

Bill Wagner, *Effective C#: 50 Specific Ways to Improve Your C#* 0321245660

More Effective C#

50 Specific Ways to Improve Your C#

Bill Wagner

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Wagner, Bill.
More effective C# : 50 specific ways to improve your C# / Bill Wagner.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-321-48589-2 (pbk. : alk. paper)
1. C# (Computer program language) 2. Database management. 3. Microsoft .NET. I. Title.

QA76.73.C154W343 2008
005.13'3—dc22

2008030878

ISBN-13: 978-0-321-48589-2
ISBN-10: 0-321-48589-0
Text printed in the United States on recycled paper at Donnelley in Crawfordsville, IN.
First printing, October 2008

For Marlene, who puts up with me writing,
time and time again.

This page intentionally left blank

Contents at a Glance

This page intentionally left blank

Contents

Introduction

When Anders Hejlsberg first showed Language-Integrated Query (LINQ) to the world at the 2005 Professional Developers Conference (PDC), the C# programming world changed. LINQ justified several new features in the C# language: extension methods, local variable type inference, lambda expressions, anonymous types, object initializers, and collection initializers. C# 2.0 set the stage for LINQ by adding generics, iterators, static classes, nullable types, property accessor accessibility, and anonymous delegates. But all these features are useful outside LINQ: They are handy for many programming tasks that have nothing to do with querying data sources.

This book provides practical advice about the features added to the C# programming language in the 2.0 and 3.0 releases, along with advanced features that were not covered in my earlier *Effective C#: 50 Specific Ways to Improve Your C#* (Addison-Wesley, 2004). The items in *More Effective C#* reflect the advice I give developers who are adopting C# 3.0 in their professional work. There's a heavy emphasis on generics, an enabling technology for everything in C# 2.0 and 3.0. I discuss the new features in C# 3.0; rather than organize the topics by language feature, I present these tips from the perspective of recommendations about the programming problems that developers can best solve by using these new features.

Consistent with the other books in the Effective Software Development Series, this book contains self-contained items detailing specific advice about how to use C#. The items are organized to guide you from using C# 1.x to using C# 3.0 in the best way.

Generics are an enabling technology for all new idioms that are part of C# 3.0. Although only the first chapter specifically addresses generics, you'll find that they are an integral part of almost every item. After reading this book, you'll be much more comfortable with generics and metaprogramming.

Of course, much of the book discusses how to use C# 3.0 and the LINQ query syntax in your code. The features added in C# 3.0 are very useful in

their own right, whether or not you are querying data sources. These changes in the language are so extensive, and LINQ is such a large part of the justification for those changes, that each warrants its own chapter. LINQ and C# 3.0 will have a profound impact on how you write code in C#. This book will make that transition easier.

Who Should Read This Book?

This book was written for professional software developers who use C#. It assumes that you have some familiarity with C# 2.0 and C# 3.0. Scott Meyers counseled me that an *Effective* book should be a developer's second book on a subject. This book does not include tutorial information on the new language features added as the language has evolved. Instead, I explain how you can integrate these features into your ongoing development activities. You'll learn when to leverage the new language features in your development activities, and when to avoid certain practices that will lead to brittle code.

In addition to some familiarity with the newer features of the C# language, you should have an understanding of the major components that make up the .NET Framework: the .NET CLR (Common Language Runtime), the .NET BCL (Base Class Library), and the JIT (Just In Time) compiler. This book doesn't cover .NET 3.0 components, such as WCF (Windows Communication Foundation), WPF (Windows Presentation Foundation), and WF (Windows Workflow Foundation). However, all the idioms presented apply to those components as well as any other .NET Framework components you happen to prefer.

About the Content

Generics are the enabling technology for everything else added to the C# language since C# 1.1. Chapter 1 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.

Multicore processors are already ubiquitous, with more cores being added seemingly every day. This means that every C# developer needs to have a solid understanding of the support provided by the C# language for multi-

threaded programming. Although one chapter can't cover everything you need to be an expert, Chapter 2 discusses the techniques you'll need every day when you write multithreaded applications.

Chapter 3 explains how to express modern design idioms in C#. You'll learn the best way to express your intent using the rich palette of C# language features. You'll see how to leverage lazy evaluation, create composable interfaces, and avoid confusion among the various language elements in your public interfaces.

Chapter 4 discusses how to use the enhancements in C# 3.0 to solve the programming challenges you face every day. 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.

Chapter 5 explains LINQ and query syntax. 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 6 covers those items that defy classification. You'll learn how to define partial classes, work with nullable types, and avoid covariance and contravariance problems with array parameters.

Regarding the Sample Code

The samples in this book are not complete programs. They are the smallest snippets of code possible that illustrate the point. In several samples the method names substitute for a concept, such as `AllocateExpensiveResource()`. Rather than read pages of code, you can grasp the concept and quickly apply it to your professional development. Where methods are elided, the name implies what's important about the missing method.

In all cases, you can assume that the following namespaces are specified:

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

Where types are used from other namespaces, I've explicitly included the namespace in the type.

In the first three chapters, I often show C# 2.0 and C# 3.0 syntax where newer syntax is preferred but not required. In Chapters 4 and 5 I assume that you would use the 3.0 syntax.

Making Suggestions and Providing Feedback

I've made every effort to remove all errors from this book, but if you believe you have found an error, please contact me at bill.wagner@srtsolutions.com. Errata will be posted to http://srtsolutions.com/blogs/MoreEffectiveCSharp.

Acknowledgments

A colleague recently asked me to describe what it feels like to finish a book. I replied that it gives you that same feeling of satisfaction and relief that shipping a software product gives you. It's very satisfying, and yet it's an incredible amount of work. Like shipping a software product, completing a book requires collaboration among many people, and all those people deserve thanks.

I was honored to be part of the Effective Software Development Series when I wrote *Effective C#* in 2004. To follow that up with *More Effective C#* and cover the numerous and far-reaching changes in the language since then is an even greater honor. The genesis of this book was a dinner I shared with Curt Johnson and Joan Murray at PDC 2005, when I expressed my excitement about the direction Hejlsberg and the rest of the C# team were presenting there. I was already taking notes about the changes and learning how they would affect the daily lives of C# developers.

Of course, it was some time before I felt comfortable in offering advice on all these new features. I needed to spend time using them and discussing different idioms with coworkers, customers, and other developers in the community. Once I felt comfortable with the new features, I began working on the new manuscript.

I was lucky enough to have an excellent team of technical reviewers. These people suggested new topics, modified the recommendations, and found scores of technical errors in earlier drafts. Bill Craun, Wes Dyer, Nick Paldino, Tomas Restrepo, and Peter Ritchie provided detailed technical feedback that made this book as useful as it is now. Pavin Podila reviewed those areas that mention WPF to ensure correctness.

Throughout the writing process, I discussed many ideas with members of the community and the C# team. The regulars at the Ann Arbor .NET Developers Group, the Great Lakes Area .NET User Group, the Greater Lansing User Group, the West Michigan .NET User Group, and the Toledo .NET User Group acted as prototype audiences for much of the advice presented here. In addition, CodeMash attendees helped me decide what to leave in and what to leave out. In particular, I want to single out Dustin Campbell, Jay Wren, and Mike Woelmer for letting me discuss ideas with them. In addition, Mads Torgersen, Charlie Calvert, and Eric Lippert joined me in several conversations that helped clarify the advice detailed here. In particular, Charlie Calvert has the great skill of mixing an engineer's understanding with a writer's gift of clarity. Without all those discussions, this manuscript would be far less clear, and it would be missing a number of key concepts.

Having been through Scott Meyers's thorough review process twice now, I'd recommend any book in his series sight unseen. Although he's not a C# expert, he's highly gifted and clearly cares about the books in his series. Responding to his comments takes quite a bit of time, but it results in a much better book.

Throughout the whole process, Joan Murray has been an incredible asset. As editor, she's always on top of everything. She prodded me when I needed prodding, she provided a great team of reviewers, and she helped shepherd the book from inception through outlines, manuscript drafts, and finally into the version you hold now. Along with Curt Johnson, she makes working with Addison-Wesley a joy.

The last step is working with a copy editor. Betsy Hardinger was somehow able to translate an engineer's jargon into English without sacrificing technical correctness. The book you're holding is much easier to read after her edits.

Of course, writing a book takes a large investment of time. During that time, Dianne Marsh, the other owner of SRT Solutions, kept the company moving forward. The greatest sacrifice was from my family, who saw much less of me than they'd like while I was writing this book. The biggest thanks go to Marlene, Lara, Sarah, and Scott, who put up with me as I did this again.

This page intentionally left blank

1 Working with Generics

Without a doubt, C# 2.0 added a feature that continues to have a big impact on how you write C# code: generics. Many articles and papers have been written about the advantages of using generics over the previous versions of the C# collections classes, and those articles are correct. You gain compile-time type safety and improve your applications' performance by using generic types rather than weakly typed collections that rely on `System.Object`.

Some articles and papers might lead you to believe that generics are useful only in the context of collections. That's not true. There are many other ways to use generics. You can use them to create interfaces, event handlers, common algorithms, and more.

Many other discussions compare C# generics to C++ templates, usually to advocate one as better than the other. Comparing C# generics to C++ templates is useful to help you understand the syntax, but that's where the comparison should end. Certain idioms are more natural to C++ templates, and others are more natural to C# generics. But, as you'll see in Item 2 a bit later in this chapter, trying to decide which is "better" will only hurt your understanding of both of them. Adding generics required changes to the C# compiler, the Just In Time (JIT) compiler, and the Common Language Runtime (CLR). The C# compiler takes your C# code and creates the Microsoft Intermediate Language (MSIL, or IL) definition for the generic type. In contrast, the JIT compiler combines a generic type definition with a set of type parameters to create a closed generic type. The CLR supports both those concepts at runtime.

There are costs and benefits associated with generic type definitions. Sometimes, replacing specific code with a generic equivalent makes your program smaller. At other times, it makes it larger. Whether or not you encounter this generic code bloat depends on the specific type parameters you use and the number of closed generic types you create.

Generic class definitions are fully compiled MSIL types. The code they contain must be completely valid for any type parameters that satisfy the

constraints. The generic definition is called a **generic type definition.** A specific instance of a generic type, in which all the type parameters have been specified, is called a **closed generic type.** (If only some of the parameters are specified, it's called an **open generic type.**)

Generics in IL are a partial definition of a real type. The IL contains the placeholder for an instantiation of a specific completed generic type. The JIT compiler completes that definition when it creates the machine code to instantiate a closed generic type at runtime. This practice introduces a tradeoff between paying the increased code cost for multiple closed generic types and gaining the decreased time and space required in order to store data.

Different closed generic types may or may not produce different runtime representations of the code. When you create multiple closed generic types, the JIT compiler and the CLR perform some optimizations to minimize the memory pressure. Assemblies, in IL form, are loaded into data pages. As the JIT compiler translates the IL into machine instructions, the resulting machine code is stored in read-only code pages.

This process happens for every type you create, generic or not. With nongeneric types, there is a 1:1 correspondence between the IL for a class and the machine code created. Generics introduce some new wrinkles to that translation. When a generic class is JIT-compiled, the JIT compiler examines the type parameters and emits specific instructions depending on the type parameters. The JIT compiler performs a number of optimizations to fold different type parameters into the same machine code. First and foremost, the JIT compiler creates one machine version of a generic class for all reference types.

All these instantiations share the same code at runtime:

```
List <string> stringList = new List<string>();
List<Stream> OpenFiles = new List<Stream>();
List<MyClassType> anotherList = new List<MyClassType>();
```

The C# compiler enforces type safety at compile time, and the JIT compiler can produce a more optimized version of the machine code by assuming that the types are correct.

Different rules apply to closed generic types that have at least one value type used as a type parameter. The JIT compiler creates a different set of machine instructions for different type parameters. Therefore, the following three closed generic types have different machine code pages:

```
List<double> doubleList = new List<double>();
List<int> markers = new List<int>();
List<MyStruct> values = new List<MyStruct>();
```

This may be interesting, but why should you care? Generic types that will be used with multiple different reference types do not affect the memory footprint. All JIT-compiled code is shared. However, when closed generic types contain value types as parameters, that JIT-compiled code is not shared. Let's dig a little deeper into that process to see how it will be affected.

When the runtime needs to JIT-compile a generic definition (either a method or a class) and at least one of the type parameters is a value type, it goes through a two-step process. First, it creates a new IL class that represents the closed generic type. I'm simplifying, but essentially the runtime replaces `T` with `int`, or the appropriate value type, in all locations in the generic definition. After that replacement, it JIT-compiles the necessary code into x86 instructions. This two-step process is necessary because the JIT compiler does not create the x86 code for an entire class when loaded; instead, each method is JIT-compiled only when first called. Therefore, it makes sense to do a block substitution in the IL and then JIT-compile the resulting IL on demand, as is done with normal class definitions.

This means that the runtime costs of memory footprint add up in this way: one extra copy of the IL definition for each closed generic type that uses a value type, and a second extra copy of machine code for each method called in each different value type parameter used in a closed generic type.

There is, however, a plus side to using generics with value type parameters: You avoid all boxing and unboxing of value types, thereby reducing the size of both code and data for value types. Furthermore, type safety is ensured by the compiler; thus, fewer runtime checks are needed, and that reduces the size of the codebase and improves performance. Furthermore, as discussed in Item 8, creating generic methods instead of generic classes can limit the amount of extra IL code created for each separate instantiation. Only those methods actually referenced will be instantiated. Generic methods defined in a nongeneric class are not JIT-compiled.

This chapter discusses many of the ways you can use generics and explains how to create generic types and methods that will save you time and help you create usable components. I also cover when and how to migrate .NET 1.x types (in which you use `System.Object`) to .NET 2.0 types, in which you specify type parameters.

Item 1: Use Generic Replacements of 1.x Framework API Classes

The first two releases of the .NET platform did not support generics. Your only choice was to code against `System.Object` and add appropriate runtime checks to ensure that the runtime type of the object was what you expected, usually a specific type derived from `System.Object`. This practice was even more widespread in the .NET Framework, because the framework designers were creating a library of lower-level components that would be used by everyone.

`System.Object` is the ultimate base class for every type you or anyone else creates. That led to the obvious decision to use `System.Object` as a substitute for "whatever type you want to use in this space." Unfortunately, that's all the compiler knows about your types. This means that you must code everything very defensively—and so must everyone who uses your types. Whenever you have `System.Object` as a parameter or a return type, you have the potential to substitute the wrong type. That's a cause for runtime errors in your code.

With the addition of generics, those days are gone. If you've been using .NET for any period of time, you've probably adopted the habit of using many classes and interfaces that now should be cast aside in favor of an updated generic version. You can improve the quality of your code by replacing `System.Object` with generic type parameters. Why? It's because it's much harder to misuse generic types by supplying arguments of the wrong type.

If correctness isn't enough to motivate you to replace your old `System.Object` code with generic equivalents, maybe performance will get you interested. .NET 1.1 forced you to use the ultimate base class of `System.Object` and dynamically cast objects to the expected type before using them. The 1.1 versions of any class or interface require that you box and unbox value types every time you coerce between the value type and the `System.Object` type. Depending on your usage, that requirement may have a significant impact on performance. Of course, it applies only with value types. But, as I said earlier, the weakly typed systems from the 1.1 days require both you and your users to author defensive code to test the runtime type of your parameters and return types. Even when that code functions correctly, it adds runtime performance costs. And it's worse when it fails; the runtime costs probably include stack walks and unwinding when casts throw exceptions and the runtime searches for the proper `catch` clause. You run the risk of everything from costly application slowdown to abnormal application termination.

A good look at the .NET Framework 2.0 shows you how much you can transform your code by using generics. The obvious starting point is the `System.Collections.Generics` namespace, followed by the `System.Collections.ObjectModel` namespace. Every class that is part of the `System.Collections` namespace has a new, improved counterpart in `System.Collections.Generics`. For example, `ArrayList` has been superseded by `List<T>`, `Stack` has been replaced by `Stack<T>`, `Hashtable` has been replaced by `Dictionary<K,V>`, and `Queue` has been replaced by `Queue<T>`. In addition, there are a few new collections, such as `SortedList<T>` and `LinkedList<T>`.

The addition of these classes meant the addition of generic interfaces. Again, the `System.Collections.Generics` namespace points to the obvious examples. The original `IList` interface has been extended with `IList<T>`. All the collections-based interfaces have been similarly upgraded: `IDictionary<K,V>` replaces `IDictionary`, `IEnumerable<T>` extends `IEnumerable`, `IComparer<T>` replaces `IComparer`, and `ICollection<T>` replaces `ICollection`.

I say "extends" and "replaces" deliberately. Many of the generic interfaces derive from their nongeneric counterparts, extending the classic capability with upgraded, type-specific versions. Other classic interfaces are not part of the signature of the newer interfaces. For a variety of reasons, the newer interface method signatures aren't consistent with the classic interfaces. When that happened, the framework designers chose not to tie the new interface definitions to an outdated interface.

The .NET 2.0 Framework has added an `IEquatable<T>` interface to minimize the potential errors involved in overriding `System.Object.Equals`:

```
public interface IEquatable<T>
{
    bool Equals(T other);
}
```

You should add support for this interface wherever you would have overwritten `System.Object.Equals`.

If you need to perform comparisons on a type defined in another library, the .NET 2.0 Framework has also added a new equality interface in the generic collections namespace: `IEqualityComparer<T>`. This interface has two methods: `Equals` and `GetHashCode`.

```
public interface IEqualityComparer<T>
{
    int Equals( T x, T y);
    int GetHashCode(T obj);
}
```

You can create a helper class that implements `IEqualityComparer<T>` for any third-party type you use today. This class works like any class that implements the 1.1 version of `IHashCodeProvider`. It enables you to create type-safe equality comparisons for your types, deprecating the old versions based on `System.Object`. You'll almost never need to write a full implementation of `IEqualityComparer<T>` yourself. Instead, you can use the `EqualityComparer<T>` class and its `Default` property. For example, you would write the following `EmployeeComparer` class, derived from `EqualityComparer<T>`, to test the equality of `Employee` objects created in another library:

```
public class EmployeeComparer : EqualityComparer<Employee>
{
    public override bool Equals(Employee x, Employee y)
    {
        return EqualityComparer<Employee>.Default.Equals(x, y);
    }

    public override int GetHashCode(Employee obj)
    {
        return EqualityComparer<Employee>.Default.
            GetHashCode(obj);
    }
}
```

The `Default` property examines the type argument, `T`. If the type implements `IEquatable<T>`, then `Default` returns an `IEqualityComparer<T>` that uses the generic interface. If not, `Default` returns an `IEqualityComparer<T>` that uses the `System.Object` virtual methods `Equals()` and `GetHashCode()`. In this way, `EqualityComparer<T>` guarantees the best implementation for you.

These methods illustrate one essential fact to remember about generic types: The more fundamental the algorithm, such as equality, the more likely it is that you will want a generic type definition. When you create fundamental algorithms that have several variations, you'll want the compile-time checking you get with generic type definitions.

To show you what I mean, let's browse through the `System` namespace to learn from the other generic classes that are available in the .NET 2.0 Framework. I'm taking you on this tour for two reasons. First, if you've been using C# for a while, you've already developed 1.1 habits that don't include these classes. You should change those habits to incorporate the improvements offered by the generic versions. Second, the framework classes provide a great set of examples for the kinds of problems you can solve using generics.

I mentioned that the `System.Collections.Generic` namespace contains an `IComparer<T>` interface to improve on the classic `IComparer` interface. Well, the `System` namespace also contains a generic compare interface: `IComparable<T>`.

```
public interface IComparable<T>
{
    int CompareTo(T other);
}
```

The obvious analog is the old-style `IComparable` interface:

```
// 1.1 Comparable signature
public interface IComparable
{
    int CompareTo(object other);
}
```

The gains provided by the typical implementation show exactly when the generic version is superior. Here's a snippet of code from Item 9 in my earlier book *Effective C#: 50 Specific Ways to Improve Your C#* (Addison-Wesley, 2004):

```
// Code from the Customer struct (a value type)
public int CompareTo(object right)
{
    if (!(right is Customer))
        throw new ArgumentException("Argument not a customer",
            "right");
    Customer rightCustomer = (Customer)right;
    return Name.CompareTo(rightCustomer.Name);
}
```

The `IComparable<T>` version is much simpler than the `IComparable` version, which relies on `System.Object`:

```
public int CompareTo(Customer right)
{
    return Name.CompareTo(right.Name);
}
```

You gain four advantages by creating and using generic interfaces, because generic interfaces are type-safe. Notice the difference between the implementation of `IComparable` and `IComparable<T>`. The `IComparable<T>` version is much smaller, because the runtime type checking needed for the `IComparable` version is enforced by the compiler when you implement `IComparable<T>`. Therefore, you create less code, and that code does the same work. The generic version is also faster; you've removed some error checking. In addition, the generic version avoids any boxing and unboxing, along with the type conversions. Finally, the generic version does not emit any runtime errors. All those runtime errors that are generated by the nongeneric version are caught by the compiler in the generic version.

Of course, there are times when using `System.Object` makes sense. Your design may include comparing objects that are not related by type (or by inheritance) but may have similar properties. In that situation, you should implement the classic interfaces in addition to the newer generic versions (see Item 10 later in this chapter). Suppose you have used a third-party e-commerce system that must integrate with your own legacy shipping system. Both systems have the concept of an `Order`. Those two concepts are not related in any way by inheritance:

```
namespace ThirdPartyECommerceSystem
{
    public class Order
    {
        // details elided
    }
}
```

You would modify your own order system to include the classic interfaces so that it would support equality for both kinds of `Order` objects:

```
namespace InternalShippingSystem
{
    public class Order : IEquatable<Order>,
        IComparable<Order>
    {
```

```
        #region IEquatable<Order> Members
        public bool Equals(Order other)
        {
            // elided
            return true;
        }
        #endregion

        #region IComparable<Order> Members
        public int CompareTo(Order other)
        {
            // elided
            return 0;
        }
        #endregion

        public override bool Equals(object obj)
        {
            if (obj is Order)
                return this.Equals((Order)obj);
            else if (obj is ThirdPartyECommerceSystem.Order)
                return this.Equals
                    ((ThirdPartyECommerceSystem.Order)obj);
                throw new ArgumentException(
                    "Object type not supported", "obj");
        }
        public bool Equals(ThirdPartyECommerceSystem.Order
            other)
        {
            bool equal = true;
            // tests elided
            return equal;
        }
    }
}
```

The classic interfaces can support this kind of relationship between unrelated types that represent the same concept. This isn't a design I would intentionally add to my system, but when you must get libraries from different vendors to work together with similar concepts, it is the best option.

Examine the code carefully, and notice that the code overriding `System.Object.Equals()` reports problems using runtime errors instead of compile-time errors.

Whenever you can replace `System.Object` with the correct type, you'll get compile-time type safety, along with some performance gains. Clearly, `IComparable<T>` is preferred to `IComparable` in almost all cases. In fact, almost any 1.x interface that used `System.Object` has been updated and replaced with a generic equivalent. The only obvious counterexample is `ICloneable`, which has not been updated with a generic equivalent. Implementing `ICloneable` is discouraged. See Krzysztof Cwalina and Brad Abrams, *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries* (Addison-Wesley, 2005), pp. 221–222.

One of the most commonly used generic additions in the .NET 2.0 Framework is the **nullable generic type.** Nullable types are implemented using two complementary type definitions: the `Nullable<T>` struct, and the static `Nullable` class. The `Nullable<T>` struct is the wrapper for any value type that needs to also represent a null value (see Item 45, Chapter 6). For the moment, it's enough to know that one generic type handles the relationship between a nullable type and the underlying value type instance that it should represent.

To support nullable value types, the base class library added a static `Nullable` class and a `Nullable<T>` generic structure. The static `Nullable` class contains some generic methods that you can use to work with nullable types. `Nullable.GetUnderlyingType(Type t)` returns the type of the underlying object in a `Nullable<T>`; in other words, the following actual runtime type compares two nullables of the same type and determines whether they are equal:

```
Nullable.Compare<T> (Nullable<T> left, Nullable<T> right);
```

Together, these classes provide the functionality that the C# compiler uses to implement your requests for nullable types.

The .NET Framework designers also added some utility delegates using generics that you can use (and they use) to handle common patterns. For example, suppose you want to create a callback that visits every element of a generic collection. Using `IEnumerable<T>`, you could visit each element using a simple `foreach` loop:

```
List<MyType> theList = new List<MyType>();
foreach (MyType thing in theList)
    thing.DoSomething();
```

You can replace `DoSomething` with any function that matches the `System.Action` delegate:

```
public delegate void Action<T>(T obj);
```

A simple generic function can now visit every item in a collection and perform some action on each element:

```
public static void EnumerateAll<T>(IEnumerable<T>
theCollection,
    Action<T> doIt)
{
    foreach (T thing in theCollection)
        doIt(thing);
}
```

Of course, in C# 3.0, you should prefer to create this as an extension method:

```
public static void EnumerateAll<T>(this IEnumerable<T>
theCollection,
    Action<T> doIt)
{
    foreach (T thing in theCollection)
        doIt(thing);
}
```

When you want to sort an array of objects that doesn't implement `IComparable<T>`, you can supply a delegate that matches the `System.Comparison` delegate:

```
delegate int Comparison<T>( T x, T y);
```

This delegate is used in `List.Sort<T>(Comparison<T> comparison)`, an overload of `Sort()` that allows you to specify your own comparison method.

There are also a couple of delegates that you can use to transform objects contained in a collection in a type-safe manner. The `System.Converter` delegate converts one input object into a corresponding output:

```
public delegate TOutput Converter<TInput, TOutput>(TInput
    input);
```

This delegate lets you write a generic method that transforms a sequence of one type into a sequence of another type:

```
public IEnumerable<TOutput> Transform<TInput, TOutput>(
    IEnumerable<TInput> theCollection,
    Converter<TInput, TOutput> transformer)
{
    foreach (TInput source in theCollection)
        yield return transformer(source);
}
```

`Transform` contains two type parameters: `TInput` and `TOutput`. They represent the input and output types for the transform. `System.Converter` uses the same convention to describe the two types.

Sometimes you want to perform a test on every object of a collection. To do that, you create a method that uses the `System.Predicate` delegate:

```
delegate bool Predicate<T>(T obj)
```

Another simple modification lets you create a sequence of all the elements of a sequence that pass a test:

```
public IEnumerable<T> Test<T> (IEnumerable<T> theCollection,
Predicate<T> test)
{
    foreach (T source in theCollection)
        if ( test( source ) )
            yield return source;
}
```

Generics also can help you implement events with a lot less hand-written code. In .NET 1.1, you had to create a class derived from `EventArgs`, then create the delegate definition, and then create an event definition to match the delegate. It's not difficult, but it's terribly repetitive. Instead, the .NET 2.0 Framework includes a generic definition for an event handler:

```
public delegate void EventHandler<TEventArgs>(
    object sender, TEventArgs args)
    where TEventArgs: EventArgs
```

This replaces the most repetitive of the custom code you needed to create the delegate. The separate delegate and event definitions might look like this:

```
public delegate void MyEventHandler(object sender,
    MyEventArgs args);
public event MyEventHandler OnRaiseMyEvent;
```

Instead, you now have this:

```
public event EventHandler<MyEventArgs> OnRaiseMyEvent;
```

It's a small saving, but it adds up when you're defining numerous events.

That covers the major generic additions to the `System` namespace, but there is one more important addition to mention: `System.ComponentModel.BindingList<T>`. Creating a type that implemented `IBindingList` in the .NET 1.x Framework was a boring and painful task. Also, most of that code was very similar, and it had little to do with the actual type being placed in the list. But `IBindingList`, `ICancelAddNew`, and `CurrencyManager` work together, so there were many opportunities to make mistakes. For example, if you wanted to create a class that implemented `IBindingList` for a typical employee class, it would take you several pages of code. In .NET 2.0, it's simply this:

```
System.ComponentModel.BindingList<Employee>
```

What could be easier?

That concludes the tour of the major generic classes in the .NET 2.0 Framework. You can see that using generics means that you write less error-checking code, because now the compiler validates the types of parameters and return values in those places where you had to do it before. Generics are a big win, so get to know the generic definitions in the framework, and use them.

The final question is what to do with your current 1.1 code assets that have better generic equivalents. I recommend that you replace that code with its generic equivalent at your earliest convenience, but don't create a separate task simply to use generics. Code that already works is fine as it is. But as soon as you update or modify it, you encounter new opportunities to create bugs that the compiler would find for you. When you undertake a large task on any nongeneric class, start by replacing the pertinent definitions with the generic equivalent, and let the compiler help you find the remaining issues.

This item presents the case for changing any habits you have of building code using the classic nongeneric application programming interface (API) in favor of using the newer generic versions. It will be easier to use your libraries correctly, and harder to use them incorrectly. You'll leverage the compiler to do more type checking for you. By using the generic delegate definitions, you'll be better able to leverage the C# 3.0 enhancements

that let the compiler infer types, freeing you to think about algorithms and logic. The sooner your code uses generics instead of nongeneric counterparts wherever possible, the sooner you'll be able to leverage the newest language features.

Item 2: Define Constraints That Are Minimal and Sufficient

The constraints you declare on your type parameters specify the must-have behaviors your class needs in order to accomplish its work. A type that doesn't satisfy all your constraints simply won't work. Balance that against the fact that every constraint you impose could mean more work for developers who want to use your type. The right choice varies from task to task, but either extreme is wrong. If you don't specify any constraints, you must perform more checks at runtime: You'll perform more casts, possibly using reflection and generating more runtime errors if your users misuse your type. Specifying unneeded constraints means making too much work for the users of your class. Your goal is to find the middle ground, where you specify what you need but not everything you want.

Constraints enable the compiler to expect capabilities in a type parameter beyond those in the public interface defined in `System.Object`. When you create a generic type, the C# compiler must generate valid IL for the generic type definition. While doing so, the compiler must create a valid assembly even though the compiler has only limited knowledge of the actual type that may be used to substitute for any type parameters. Without any guidance from you, the compiler can only assume the most basic capabilities about those types: the methods exposed by `System.Object`. The compiler cannot enforce any assumptions you have made about your types. All the compiler knows is that your types must derive from `System.Object`. (This means that you cannot create unsafe generics using pointers as type parameters.) Assuming only the capabilities of `System.Object` is very limiting. The compiler will emit errors on anything not defined in `System.Object`. This includes even such fundamental operations as `new T()`, which is hidden if you define a constructor that has parameters.

You use constraints to communicate (to both the compiler and users) any assumptions you've made about the generic types. Constraints communicate to the compiler that your generic type expects functionality not included in `System.Object`'s public interface. This communication helps the compiler in two ways. First, it helps when you create your generic type: