TURING 图灵程序设计丛书 微软技术系列

Addison Wesley

More Effective C#

改善C#程序的50个具体办法

（英文版）

[美] Bill Wagner 著

人民邮电出版社
POSTS & TELECOM PRESS

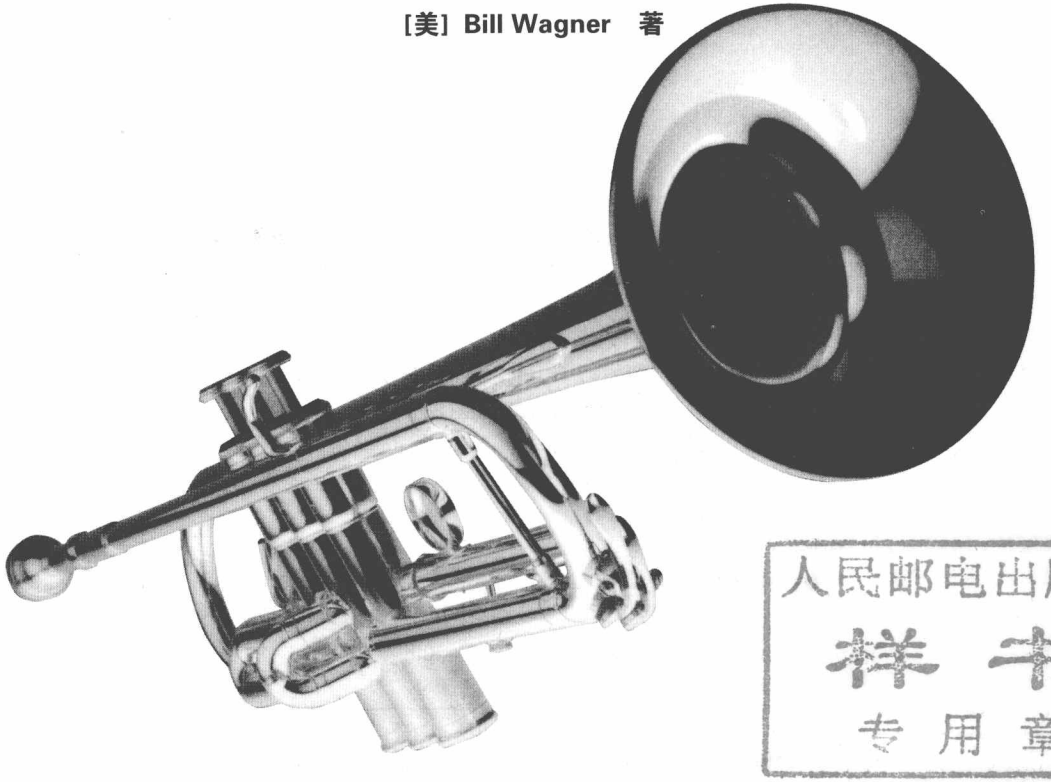
TURING 图灵程序设计丛书 微软技术系列

More Effective C#

改善C#程序的50个具体办法

（英文版）

[美] Bill Wagner 著



人民邮电出版社
北京

图书在版编目（CIP）数据

More Effective C#：改善C#程序的50个具体办法 = More Effective C#: 50 Specific Ways to Improve Your C#：英文/（美）瓦格纳（Wagner, B.）著. —北京：人民邮电出版社，2009.11
（图灵程序设计丛书）
ISBN 978-7-115-21510-9

Ⅰ. ①M… Ⅱ. ①瓦… Ⅲ. ①C语言－程序设计－英文 Ⅳ. ①TP312

中国版本图书馆CIP数据核字（2009）第171898号

内 容 提 要

本书针对C# 2.0和3.0中添加的新特性给出了改善C#代码的50条实用建议，其中着重介绍了泛型技术，这是C# 2.0和3.0中众多新特性的基石。本书按照建议的主题进行分类，其中每个建议针对某个特定问题进行展开，分析了问题的原因，给出解决的办法。

本书适合各层次.NET开发人员阅读。

图灵程序设计丛书

More Effective C#：改善C#程序的50个具体办法（英文版）

◆ 著 [美] Bill Wagner
责任编辑 傅志红

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷

◆ 开本：800×1000 1/16
印张：19.25
字数：310千字 2009年11月第1版
印数：1－3 000册 2009年11月北京第1次印刷

著作权合同登记号 图字：01-2009-5717号

ISBN 978-7-115-21510-9

定价：55.00元

读者服务热线：(010)51095186 印装质量热线：(010)67129223
反盗版热线：(010)67171154

版 权 声 明

对本书的赞誉

“本书就像一盏明灯，照亮了 C# 3.0 中很多不为人知的角落。它不仅介绍了如何做，还解释了这样做的原因，让读者学习到很多经过实践检验的语言新特性用法，包括 LINQ、泛型以及多线程等。若你确实需要使用 C# 语言开发程序，那么本书是必不可少的。”

——Bill Craun，Ambassador Solutions 公司首席咨询师

“本书创造了一个让你能够和 Bill Wagner 并肩思考、工作的机会。Bill 在本书中充分展示了他在 C# 上的造诣，给出了很多编程方面的有效建议，值得每个 Visual C# 开发者去学习。本书并没有停留在泛泛描述 C# 语法上，而是真正教会你使用 C# 语言。”

——Peter Ritchie，微软公司 MVP: Visual C#

“本书是 Bill Wagner 前一本书的很好续作。其中对 C# 3.0 和 LINQ 的介绍非常及时！”

——Tomas Restrepo，微软公司 MVP: Visual C++, .NET 和 Biztalk Server

“作为 C# 设计组的成员，很少有书能够让我从中学到什么新东西。本书则是个例外。它很好地将特定的代码和深入的分析结合了起来。本书提供了一系列非常有用的建议。当你通读全书之后，会发现不只得到了一条条独立的建议，还学到了如何能够以优雅的方式用 C# 进行程序设计。虽然你可以根据需要挑选某个条目阅读，但我仍强烈建议你通读全书——至少不要跳过每一章前面的介绍部分。这些富有洞察力的、充满远见的内容会对你日后学习 C# 提供很大的启发和帮助。”

——Mads Torgersen，微软公司 Visual C# 项目经理

“Bill Wagner 为 C# 开发人员撰写了一本精彩绝伦的图书，其中介绍了大量 C# 最佳实践。本书再次确立了他在 C# 社区中的重要地位。我们大都知道如何使用 C#，同时也期待有人能给出提高的建议，让我们更上一层楼。若想成为 C# 开发的顶级高手，那么没有什么资料要比 Bill Wagner 的这本书更好了。Bill 非常智慧、深刻，富有经验和技巧。若能将这本书中给出的建议应用到你的代码中，定会大大提高你的工作质量。”

——Charlie Calvert，微软公司 Visual C# 社区项目经理

前　言

自从 Anders Hejlsberg 在 2005 年专业开发者大会上第一次演示 LINQ（Language-Integrated Query，语言集成查询）以来，C# 编程世界被彻底地改变了。LINQ 的出现为 C# 语言带来了几个令人着迷的新特性：扩展方法、局部变量类型推断、lambda 表达式、匿名类型、对象初始化器以及集合初始化器。C# 2.0 也为 LINQ 的出现打下了坚实的基础，添加了包括泛型、迭代器、静态类、可空类型、属性访问器权限以及匿名委托等新功能。但即使在非 LINQ 的使用环境中，这些语言特性也有大显身手之处——毕竟还有很多非数据访问的编程任务。

本书针对 C# 2.0 和 C# 3.0 中添加的新特性给出了实用的建议，也包含了在我的上本图书 *Effective C#: 50 Specific Ways to Improve Your C#*（Addison-Wesley，2004）中没有提到的高级特性。本书中的条目主要针对那些正在使用 C# 3.0 编写程序的开发人员。书中着重介绍了泛型技术，这是 C# 2.0 和 C# 3.0 中众多新特性的基石。本书并没有将条目按照语言特性组织起来，而是根据新特性最善于解决的编程问题来编排条目的。

与 *Effective Software Development* 丛书中的其他图书一样，本书中的每个条目的建议都自成一体，针对使用 C# 时的某个特定问题 。这些条目能够帮助你以最佳的方式从 C# 1.*x* 切换至 C# 3.0。

泛型是 C# 3.0 中所有新特性的基础。虽然只有第 1 章专门介绍了泛型，但你会发现泛型技术也是几乎每个条目中的不可分割的一部分。在阅读完本书之后，你定会熟悉并喜欢上泛型以及元编程（metaprogramming）概念。

当然，本书中的很大一部分篇幅都用来讨论了如何使用 C# 3.0 以及 LINQ 查询语法。不过不管你是否将其用在查询数据源上，C# 3.0 所添加的众多语言新特性均非常有用。语言上的改变非常巨大，LINQ 又是引起改变的主要原因，它们都需要专门的章进行介绍。LINQ 和 C# 3.0 将深刻影响你编写 C# 代码的方式，而本书则会让这个过渡更加平稳简单。

读者对象

本书是为那些使用 C# 进行软件设计的专业开发人员所编写的。本书假定你已对 C# 2.0 和 C# 3.0 有了一定的了解。Scott Meyers 告诉我说，*Effective*

系列图书应该作为开发人员针对某一主题学习的进阶参考资料。因此，本书并没有泛泛介绍任何有关语言的新特性，而是着重阐述如何将这些新特性应用到正在开发的软件中。你将会学到，何时该在开发中使用这些新语言特性，以及如何避免误用所造成的问题。

除了对 C# 语言新特性有一定了解之外，你还应该对组成 .NET Framework 的主要组件有所了解，包括 .NET CLR（Common Language Runtime）、.NET BCL（Base Class Library）以及 JIT（Just In Time）编译器等。本书并没有涉及 .NET 3.0 组件，例如 WCF（Windows Communication Foundation）、WPF（Windows Presentation Foundation）以及 WF（Windows Workflow Foundation）等。不过其中介绍的各种用法同样适用于上述各个组件以及其他的 .NET Framework 组件。

内容介绍

泛型是自 C# 1.1 以来所有 C# 语言新功能的基础。第 1 章首先介绍了如何用泛型替代 System. Object 和类型强制转换，随后讨论了一些高级主题，包括约束、泛型的特化、方法约束以及向后兼容性等。其中介绍的几种技术都使用泛型让你更清晰地表达出设计意图。

多核处理器已经普及，同时计算机的核心数量也在不停增加。这也就意味着每个 C# 开发人员都需要对 C# 多线程编程有着足够的理解。即便一章的篇幅不足以让你变成专家，但第 2 章中的建议仍旧会对你开发多线程应用程序有所帮助。

第 3 章介绍了如何用 C# 语言实现常用的设计。其中将介绍用 C# 语言特性表达意图的最好方法。你将学到如何使用延迟求值，如何创建可组合的接口，以及如何避免由于公共接口中各种语言元素所带来的混乱。

第 4 章讨论了如何借助 C# 3.0 的语言增强来解决编程中遇到的困难，包括如何使用扩展方法来分离契约和实现，如何有效地使用 C# 闭包，以及如何使用匿名类型等内容。

第 5 章介绍了 LINQ 及其查询语法，包含了编译器如何将查询关键字映射到方法调用的方法，如何区分委托和表达式树（以及需要时在二者之间进行转换），以及如何在需要单一值（scalar value）时处理查询等。

第 6 章介绍了如何定义部分类，使用可空类型，以及在使用数组参数时避免协变和逆变问题等内容。

示例代码

本书中给出的示例代码不是完整的程序，而是小块的代码片断，但已足够说明问题。在有些示例中，方法名称就表明了该方法要完成的任务，例如

`AllocateExpensiveResource()`。这样你无需阅读整篇的代码即可快速领悟到其表达的含义，从而应用到你的开发中。在省略了方法实现时，方法名称就表明了该方法的用途。

在所有的代码片断中，均假设引入了如下命名空间：

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

若是使用到了其他命名空间中的类型，我将在类型中显式给出命名空间。

在本书的前 3 章中，我会尽可能地使用 C# 2.0 和 C# 3.0 中的新语法，即使这些新语法并不是必需的。在第 4 章和第 5 章中，则假设你已经熟悉了 3.0 的新语法。

建议和反馈

我已经仔细审校过本书，但若是你确信找到了错误，请通过电子邮件联系我：bill.wagner@srtsolutions.com。本书勘误将发布至 http://srtsolutions.com/blogs/MoreEffectiveCSharp 之上。

致谢

最近，一位同事问起我写完一本书之后的感觉。我说有一种类似发布了一个软件产品一样的满足和轻松。尽管写书工作量很大，但它确实也带来很多的成就感。与成功发布软件产品一样，完成一本书需要很多人的共同努力，这些人理应得到感谢。

在 2004 年撰写 *Effective C#* 时，我很荣幸成为 *Effective Software Development* 丛书的一名作者。现在这本涉及大量 C# 语言变化的 *More Effective C#* 继续成为该丛书中的一员，则更让我受宠若惊。撰写本书的想法源于我在 2005 年的 PDC 大会上与 Curt Johnson 和 Joan Murray 的一顿晚餐，那时我已被 Hejlsberg 和其他 C# 团队成员的演示介绍深深震撼。从那时开始，我就开始关注 C# 语言的变化，并仔细思考这些变化将为 C# 开发者带来怎样的影响。

当然，在我有能力和自信给出关于所有这些新特性的建议之前，仍旧需要花费时间来使用这些功能，并与同事、客户以及社区中的其他开发者讨论各种不同的使用方式。在万事俱备之后，我便正式开始撰写本书了。

我非常幸运拥有一支优秀的技术审校队伍。他们向我介绍新的主题，修改现有的建议，并挑出了早期草稿中出现的很多错误。Bill Craun、Wes Dyer、Nick Paldino、Tomas Restrepo 和 Peter Ritchie 都提供了详细的技术反馈意见，让本书更加实用。Pavin Podila 还审校了本书的 WPF 部分，保证了其正确性。

在写作的过程中，我和社区以及 C# 开发团队的成员讨论了很多想法。安

阿伯 .NET 开发组、大湖区 .NET 用户组、大兰辛地区用户组、西密歇根 .NET 用户组和托莱多 .NET 用户组中的成员都充当了本书各个条目的早期审校者。此外，CodeMash 的参会者也帮我决定了各个条目的取舍。特别是 Dustin Campbell、Jay Wren 和 Mike Woelmer，他们和我讨论了很多想法。Mads Torgersen、Charlie Calvert 和 Eric Lippert 也曾帮我澄清了本书中的很多条目。特别值得一提的是，Charlie Calvert 帮我更好地将工程师的思维用写作者的表达方法变成文字。若是没有这些讨论，本书将远远无法清晰明了，且可能会错过很多重要的概念。

从头到尾两次经历了 Scott Meyers 的完整审校流程之后，我可以闭着眼推荐他经手的每一本书。Scott Meyers 虽不是 C# 专家，但他的天分和对图书质量的认真负责让我非常佩服。他对本书的审读意见非常多，逐一解答确认也花费了很长时间，但这保证了书稿的质量。

在本书出版的整个过程中，Joan Murray 的作用无可替代。作为编辑，她总能帮我想到所有的一切。在我需要监督的时候给出提醒，为我找到优秀的审校者，并帮助我把本书的想法变为大纲、再到草稿、直到现在你手中的成书。还有 Curt Johnson，他们让我与 Addison-Wesley 出版社的合作非常愉快。

最后一步是与文字编辑的配合。Betsy Hardinger 帮助将工程师写出的枯燥文字转变成了生动的书面英语，而并没有损害技术上的准确性。在她编辑之后，本书的语言变得更加的流畅。

当然，撰写图书也要花费大量的时间。在这段时间中，Dianne Marsh（SRT Solutions 的另一个所有人）让公司保持正常运转。而最大的牺牲则来自于我的家庭，在撰写本书时，我无暇悉心顾及他们的感受。最真诚的感谢送给 Marlene、Lara、Sarah 和 Scott，感谢你们再次全力支持我的投入。

Contents

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.