BEN WATSON

C# 4.0

HOW-TO

Real Solutions for C# 4.0 Programmers

SAMS

C# 4.0 How-To

ISBN-13: 978-0-672-33063-6
ISBN-10: 0-672-33063-6

Library of Congress Cataloging-in-Publication Data

Watson, Ben, 1980–
C# 4.0 how-to / Ben Watson.
p. cm.
Includes index.
ISBN 978-0-672-33063-6 (pbk. : alk. paper) 1. C# (Computer program language) I. Title.
QA76.73.C154W38 2010
005.13'3—dc22

2010002735

Printed in the United States of America

First Printing March 2010

Trademarks

Warning and Disclaimer

Bulk Sales
Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales
international@pearson.com

Editor-in-Chief
Karen Gettman

Executive Editor
Neil Rowe

Acquisitions Editor
Brook Farling

Development Editor
Mark Renfrow

Managing Editor
Kristy Hart

Project Editor
Lori Lyons

Copy Editor
Bart Reed

Indexer
Brad Herriman

Proofreader
Sheri Cain

Technical Editor
Mark Strawmyer

Publishing Coordinator
Cindy Teeters

Designer
Gary Adair

Compositor
Nonie Ratcliff

INTRODUCTION

Overview of *C# 4.0 How-To*

This book is very different from a typical "bible" approach to a topic. By structuring the book as a "how-to," it presents the material by scenario in steps that are easily followed. Throughout, I have tried to keep the explanatory text to the minimum necessary and keep the focus on the code itself. Often, you will find comments embedded in the code to explain non-obvious bits.

This book is not strictly a language/library book. Besides covering the language features themselves, it dives into practical examples of application patterns, useful algorithms, and handy tips that are applicable in many situations.

Developers, both beginner and advanced, will find hundreds of useful topics in this book. Whether it's a section on lesser-known C# operators, how to sort strings that contain numbers in them, or how to implement Undo, this book contains recipes that are useful in a wide variety of situations, regardless of skill level.

In short, this is the book I wish I had on my desk when I was first learning programming and C# as well as now, whenever I need a quick reference or reminder about how to do something.

How-To Benefit from This Book

We designed this book to be easy to read from cover to cover. The goal is to gain a full understanding of C# 4.0. The subject matter is divided into four parts with easy-to-navigate and easy-to-use chapters.

Part I, "C# Fundamentals," covers the common C# functionality that you will use in every type of programming. While it may seem basic, there are a lot of tips to help you get the most of these fundamental topics.

- Chapter 1, "Type Fundamentals"
- Chapter 2, "Creating Versatile Types"
- Chapter 3, "General Coding"
- Chapter 4, "Exceptions"
- Chapter 5, "Numbers"

- Chapter 6, “Enumerations”
- Chapter 7, “Strings”
- Chapter 8, “Regular Expressions”
- Chapter 9, “Generics”

Part II, “Handling Data,” discusses how to store and manipulate data, including Internet-based data.

- Chapter 10, “Collections”
- Chapter 11, “Files and Serialization”
- Chapter 12, “Networking and the Web”
- Chapter 13, “Databases”
- Chapter 14, “XML”

Part III “User Interaction,” covers the most popular user interface paradigms in .Net, whether you work on the desktop, the Web, or both.

- Chapter 15, “Delegates, Events, and Anonymous Methods”
- Chapter 16, “Windows Forms”
- Chapter 17, “Graphics with Windows Forms and GDI+”
- Chapter 18, “WPF”
- Chapter 19, “ASP.NET”
- Chapter 20, “Silverlight”

Part IV, “Advanced C#,” has the advanced stuff to really take your applications to the next level in terms of performance, design patterns, useful algorithms, and more.

- Chapter 21, “LINQ”
- Chapter 22, “Memory Management”
- Chapter 23, “Threads, Asynchronous, and Parallel Programming”
- Chapter 24, “Reflection and Creating Plugins”
- Chapter 25, “Application Patterns and Tips”
- Chapter 26, “Interacting with the OS and Hardware”
- Chapter 27, “Fun Stuff and Loose Ends”
- Appendix A, “Essential Tools”

All of the code was developed using prerelease versions of Visual Studio 2010, but you can use earlier versions in many cases, especially for code that does not require .NET 4. If you do not have Visual Studio, you can download the Express edition from www.microsoft.com/express/default.aspx. This version will enable you to build nearly all the code samples in this book.

You can access the code samples used in this book by registering on the book's website at **informit.com/register**. Go to this URL, sign in, and enter the ISBN to register (free site registration required). After you register, look on your Account page, under Registered Products, for a link to Access Bonus Content.

How-To Continue Expanding Your Knowledge

No book can completely cover C#, the .NET Framework, or probably even hope to cover a small topic within that world. And if there were, you probably couldn't lift it, let alone read it in your lifetime.

Once you've mastered the essentials, there are plenty of resources to get your questions answered and dive deeply into .NET.

Thankfully, the MSDN documentation for .NET (located at http://msdn.microsoft.com/en-us/library/aa139615.aspx) is top-notch. Most topics have code samples and an explanation use. An added bonus is the ability at the bottom of every topic for anyone to add useful content. There are many good tips found here from other .NET developers.

The .NET Development forums (http://social.msdn.microsoft.com/Forums/en-US/category/netdevelopment) are an excellent place to get your questions answered by the experts, who, in many cases, were involved in the development and testing of .NET.

I have also found StackOverflow.com a good place to get questions answered.

The best advice I can give on how to continue expanding your knowledge is to just write software. Keep at it, think of new projects, use new technologies, go beyond your abilities. This and other books are very useful, to a point. After that, you just need to dive in and start coding, using the book as a faithful reference when you don't know how to approach a topic.

Happy coding!

CHAPTER 2

Creating Versatile Types

IN THIS CHAPTER

- Format a Type with `ToString()`
- Make Types Equatable
- Make Types Hashable with `GetHashCode()`
- Make Types Sortable
- Give Types an Index
- Notify Clients when Changes Happen
- Overload Appropriate Operators
- Convert One Type to Another
- Prevent Inheritance
- Allow Value Type to Be Null

Whenever you create your own classes, you need to consider the circumstances under which they could be used. For example, will two instances of your `Item` struct ever be compared for equality? Will your `Person` class need to be serializable, or sortable?

> **NOTE** Versatility means being able to do many things well. When you're creating your own types, it means outfitting your objects with enough "extra" stuff that they can easily be used in a wide variety of situations.

This chapter is all about making your own objects as useful and versatile as possible. In many cases, this means implementing the standard interfaces that .NET provides or simply overriding base class methods.

Format a Type with `ToString()`

> **Scenario/Problem:** You need to provide a string representation of an object for output and debugging purposes.

Solution: By default, `ToString()` will display the type's name. To show your own values, you must override the method with one of your own. To illustrate this, let's continue our `Vertex3d` class example from the previous chapter.

Assume the class initially looks like this:

```
struct Vertex3d
{
    private double _x;
    private double _y;
    private double _z;

    public double X
    {
        get { return _x; }
        set { _x = value; }
    }

    public double Y
    {
        get { return _y; }
        set { _y = value; }
    }
```

```
    public double Z
    {
        get { return _z; }
        set { _z = value; }
    }

    public Vertex3d(double x, double y, double z)
    {
        this._x = x;
        this._y = y;
        this._z = z;
    }
}
```

Override `ToString()` for Simple Output

To get a simple string representation of the vertex, override `ToString()` to return a string of your choosing.

```
public override string ToString()
{
    return string.Format("({0}, {1}, {2})", X, Y, Z);
}
```

The code

```
Vertex3d v = new Vertex3d(1.0, 2.0, 3.0);
Trace.WriteLine(v.ToString());
```

produces the following output:

```
(1, 2, 3)
```

Implement Custom Formatting for Fine Control

> **Scenario/Problem:** You need to provide consumers of your class fine-grained control over how string representations of your class look.

Solution: Although the `ToString()`implementation gets the job done, and is especially handy for debugging (Visual Studio will automatically call `ToString()` on objects in the debugger windows), it is not very flexible. By implementing `IFormattable` on your type, you can create a version of `ToString()` that is as flexible as you need.

Let's create a simple format syntax that allows us to specify which of the three values to print. To do this, we'll define the following format string:

```
"X, Y"
```

This tells `Vertex3d` to print out X and Y. The comma and space (and any other character) will be output as-is.

The struct definition will now be as follows:

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

namespace VertexDemo
{
struct Vertex3d : IFormattable
{
    ...
    public string ToString(string format, IFormatProvider formatProvider)
    {
        //"G" is .Net's standard for general formatting--all
        //types should support it
        if (format == null) format = "G";

        // is the user providing their own format provider?
        if (formatProvider != null)
        {
            ICustomFormatter formatter =
                formatProvider.GetFormat(this.GetType())
                        as ICustomFormatter;
            if (formatter != null)
            {
                return formatter.Format(format, this, formatProvider);
            }
        }

        //formatting is up to us, so let's do it
        if (format == "G")
        {
            return string.Format("({0}, {1}, {2})", X, Y, Z);
        }

        StringBuilder sb = new StringBuilder();
        int sourceIndex = 0;
```

```
        while (sourceIndex < format.Length)
        {
            switch (format[sourceIndex])
            {
                case 'X':
                    sb.Append(X.ToString());
                    break;
                case 'Y':
                    sb.Append(Y.ToString());
                    break;
                case 'Z':
                    sb.Append(Z.ToString());
                    break;
                default:
                    sb.Append(format[sourceIndex]);
                    break;
            }
            sourceIndex++;
        }
        return sb.ToString();
    }
}
}
```

The `formatProvider` argument allows you to pass in a formatter that does something different from the type's own formatting (say, if you can't change the implementation of `ToString()` on `Vertex3d` for some reason, or you need to apply different formatting in specific situations). You'll see how to define a custom formatter in the next section.

Formatting with `ICustomFormatter` and `StringBuilder`

Scenario/Problem: You need a general-purpose formatter than can apply custom formats to many types of objects.

Solution: Use `ICustomFormatter` and `StringBuilder`. This example prints out type information, as well as whatever the custom format string specifies for the given types.

```
class TypeFormatter : IFormatProvider, ICustomFormatter
{
    public object GetFormat(Type formatType)
    {
```

```
        if (formatType == typeof(ICustomFormatter)) return this;
        return Thread.CurrentThread.CurrentCulture.GetFormat(formatType);
    }

    public string Format(string format, object arg, IFormatProvider
formatProvider)
    {
        string value;
        IFormattable formattable = arg as IFormattable;
        if (formattable == null)
        {
            value = arg.ToString();
        }
        else
        {
            value = formattable.ToString(format, formatProvider);
        }
        return string.Format("Type: {0}, Value: {1}", arg.GetType(),
value);
    }
}
```

The class can be used like this:

```
Vertex3d v = new Vertex3d(1.0, 2.0, 3.0);
Vertex3d v2 = new Vertex3d(4.0, 5.0, 6.0);
TypeFormatter formatter = new TypeFormatter();
StringBuilder sb = new StringBuilder();
sb.AppendFormat(formatter, "{0:(X Y)}; {1:[X, Y, Z]}", v, v2);
Console.WriteLine(sb.ToString());
```

The following output is produced:

```
Type: ch02.Vertex3d, Value: (1 2); Type: ch02.Vertex3d, Value: [4, 5, 6]
```

Make Types Equatable

Scenario/Problem: You need to determine if two objects are equal.

Solution: You should override `Object.Equals()` and also implement the `IEquatable<T>` interface.

By default, `Equals()` on a reference type checks to see if the objects refer to the same location in memory. This may be acceptable in some circumstances, but often, you'll want to provide your own behavior. With value types, the default behavior is to reflect over each field and do a bit-by-bit comparison. This can have a very negative impact on performance, and in nearly every case you should provide your own implementation of `Equals()`.

```
struct Vertex3d : IFormattable, IEquatable<Vertex3d>
{
    ...
    public override bool Equals(object obj)
    {
        if (obj == null)
            return false;
        if (obj.GetType() != this.GetType())
            return false;
        return Equals((Vertex3d)obj);
    }

    public bool Equals(Vertex3d other)
    {
        /* If Vertex3d were a reference type you would also need:
         * if ((object)other == null)
         *  return false;
         *
         * if (!base.Equals(other))
         *  return false;
         */

        return this._x == other._x
            && this._y == other._y
            && this._z == other._z;
    }
}
```

NOTE Pay special attention to the note in `Equals(Vertex3d other)`. If `Vertex3d` was a reference type and `other` was null, the type-safe version of the function would be called, not the `Object` version. You also need to call all the base classes in the hierarchy so they have an opportunity to check their own fields.

There's nothing stopping you from also implementing `IEquatable<string>` (or any other type) on your type—you can define it however you want. Use with caution, however, because this may confuse people who have to use your code.

Make Types Hashable with GetHashCode()

Scenario/Problem: You want to use your class as the key part in a collection that indexes values by unique keys. To do this, your class must be able to convert the “essence” of its values into a semi-unique integer ID.

Solution: You almost always want to override GetHashCode(), especially with value types, for performance reasons. Generating a hash value is generally done by somehow distilling the data values in your class to an integer representation that is different for every value your class can have. You should override GetHashCode() whenever you override Equals().

```
public override int GetHashCode()
{
    //note: This is just a sample hash algorithm.
    //picking a good algorithm can require some
    //research and experimentation
    return (((int)_x ^ (int)_z) << 16) |
           (((int)_y ^ (int)_z) & 0x0000FFFF);
}
```

NOTE Hash codes are not supposed to be unique for every possible set of values your type can have. This is actually impossible, as you can deduce from the previous code sample. For this reason, comparing hash values is not a good way to compute equality.

Make Types Sortable

Scenario/Problem: Objects of your type will be sorted in a collection or otherwise compared to each other.

Solution: Because you often don’t know how your type will be used, making the objects sortable is highly recommended whenever possible.

In the Vector3d class example, in order to make the objects comparable, we’ll add an _id field and implement the IComparable<Vertex3d> interface.

The _id field will be what determines the order (it doesn’t make much sense to sort on coordinates, generally).

The sorting function is simple. It takes an object of `Vertex3d` and returns one of three values:

< 0	`this` is less than `other`
`0`	`this` is same as `other`
> 0	`this` is greater than `other`

Within the `CompareTo` function, you can do anything you want to arrive at those values. In our case, we can do the comparison ourself or just call the same function on the `_id` field.

```
struct Vertex3d : IFormattable, IEquatable<Vertex3d>,
                    IComparable<Vertex3d>
{
    private int _id;

    public int Id
    {
        get
        {
            return _id;
        }
        set
        {
            _id = value;
        }
    }

    public Vertex3d(double x, double y, double z)
    {
        _x = x;
        _y = y;
        _z = z;

        _id = 0;
    }
    ...
    public int CompareTo(Vertex3d other)
    {
        if (_id < other._id)
            return -1;
        if (_id == other._id)
            return 0;
        return 1;
        /* We could also just do this:
         * return _id.CompareTo(other._id);
```

```
        * */
    }
}
```

Give Types an Index

Scenario/Problem: Your type has data values that can be accessed by some kind of index, either numerical or string based.

Solution: You can index by any type. The most common index types are `int` and `string`.

Implement a Numerical Index

You use the array access brackets to define an index on the `this` object, like this sample:

```
public double this[int index]
{
    get
    {
        switch (index)
        {
            case 0: return _x;
            case 1: return _y;
            case 2: return _z;
            default: throw new ArgumentOutOfRangeException("index",
                "Only indexes 0-2 valid!");
        }
    }
    set
    {
        switch (index)
        {
            case 0: _x = value; break;
            case 1: _y = value; break;
            case 2: _z = value; break;
            default: throw new ArgumentOutOfRangeException("index",
                "Only indexes 0-2 valid!");
        }

    }
}
```

Implement a String Index

Unlike regular arrays, however, you are not limited to integer indices. You can use any type at all, most commonly strings, as in this example:

```
public double this[string dimension]
{
    get
    {
        switch (dimension)
        {
            case "x":
            case "X": return _x;
            case "y":
            case "Y": return _y;
            case "z":
            case "Z": return _z;
            default: throw new ArgumentOutOfRangeException("dimension",
                "Only dimensions X, Y, and Z are valid!");
        }
    }
    set
    {
        switch (dimension)
        {
            case "x":
            case "X": _x = value; break;
            case "y":
            case "Y": _y = value; break;
            case "z":
            case "Z": _z = value; break;
            default: throw new ArgumentOutOfRangeException("dimension",
                "Only dimensions X, Y, and Z are valid!");
        }
    }
}
```

Sample usage:

```
Vertex3d v = new Vertex3d(1, 2, 3);
Console.WriteLine(v[0]);
Console.WriteLine(v["Z"]);
```

Output:

```
1
3
```

Notify Clients when Changes Happen

Scenario/Problem: You want users of your class to know when data inside the class changes.

Solution: Implement the `INotifyPropertyChanged` interface (located in `System.ComponentModel`).

```
using System.ComponentModel;
...
class MyDataClass : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new
PropertyChangedEventArgs(propertyName));
        }
    }

    private int _tag = 0;
    public int Tag
    {
        get
        { return _tag; }
        set
        {
            _tag = value;
            OnPropertyChanged("Tag");
        }
    }
}
```

The Windows Presentation Foundation (WPF) makes extensive use of this interface for data binding, but you can use it for your own purposes as well.

To consume such a class, use code similar to this:

```
void WatchObject(object obj)
{
    INotifyPropertyChanged watchableObj = obj as INotifyPropertyChanged;
```

```
    if (watchableObj != null)
    {
        watchableObj.PropertyChanged += new
                PropertyChangedEventHandler(data_PropertyChanged);
    }
}

void data_PropertyChanged(object sender, PropertyChangedEventArgs e)
{
    //do something when data changes
}
```

Overload Appropriate Operators

Scenario/Problem: You want to define what the +, *, ==, and != operators do when called on your type.

Solution: Operator overloading is like sugar: a little is sweet, but a lot will make you sick. Ensure that you only use this technique for situations that make sense.

Implement `operator +`

Notice that the method is `public static` and takes both operators as arguments.

```
public static Vertex3d operator +(Vertex3d a, Vertex3d b)
{
    return new Vertex3d(a.X + b.X, a.Y + b.Y, a.Z + b.Z);
}
```

The same principal can be applied to the -, *, /, %, &, |, <<, >>, !, ~, ++, and -- operators as well.

Implement `operator ==` and `operator !=`

These should always be implemented as a pair. Because we've already implemented a useful `Equals()` method, just call that instead.

```
public static bool operator ==(Vertex3d a, Vertex3d b)
{
    return a.Equals(b);
}

public static bool operator !=(Vertex3d a, Vertex3d b)
{
```

```
    return !(a==b);
}
```

What if the type is a reference type? In this case, you have to handle null values for both `a` and `b`, as in this example:

```
public static bool operator ==(CatalogItem a, CatalogItem b)
{
    if ((object)a == null && (object)b == null)
        return true;
    if ((object)a == null || (object)b == null)
        return false;
    return a.Equals(b);
}

public static bool operator !=(CatalogItem a, CatalogItem b)
{
    return !(a == b);
}
```

Convert One Type to Another

Scenario/Problem: You need to convert one type to another, either automatically or by requiring an explicit cast.

Solution: Implement a conversion operator. There are two types of conversion operators: `implicit` and `explicit`. To understand the difference, we'll implement a new `struct` called `Vertex3i` that is the same as `Vertex3d`, except the dimensions are integers instead of doubles.

Explicit Conversion (Loss of Precision)

Explicit conversion is encouraged when the conversion will result in a loss of precision. When you're converting from `System.Double` to `System.Int32`, for example, all of the decimal precision is lost. You don't (necessarily) want the compiler to allow this conversion automatically, so you make it explicit. This code goes in the `Vertex3d` class:

```
public static explicit operator Vertex3i(Vertex3d vertex)
{
    return new Vertex3i((Int32)vertex._x, (Int32)vertex._y,
➥                      (Int32)vertex._z);
}
```

To convert from `Vertex3d` to `Vertex3i` then, you would do the following:

```
Vertex3d vd = new Vertex3d(1.5, 2.5, 3.5);
Vertex3i vi = (Vertex3i)vd;
```

If you tried it without the cast, you would get the following:

```
//Vertex3i vi = vd;
Error: Cannot implicitly convert type 'Vertex3d' to 'Vertex3i'.
       An explicit conversion exists (are you missing a cast?)
```

Implicit Conversion (No Loss of Precision)

If there will not be any loss in precision, then the conversion can be implicit, meaning the compiler will allow you to assign the type with no explicit conversion. We can implement this type of conversion in the `Vertex3i` class because it can convert up to a double with no loss of precision.

```
public static implicit operator Vertex3d(Vertex3i vertex)
{
    return new Vertex3d(vertex._x, vertex._y, vertex._z);
}
```

Now we can assign without casting:

```
Vertex3i vi = new Vertex3i(1, 2, 3);
Vertex3d vd = vi;
```

Prevent Inheritance

Scenario/Problem: You want to prevent users of your class from inheriting from it.

Solution: Mark the class as `sealed`.

```
sealed class MyClass
{
...
}
```

Structs are inherently sealed.

Prevent Overriding of a Single Method

Scenario/Problem: You don't want to ban inheritance on your type, but you do want to prevent certain methods or properties from being overridden.

Solution: Put `sealed` as part of the method or property definition.

```
class ParentClass
{
    public virtual void MyFunc() { }
}

class ChildClass : ParentClass
{
    //seal base class function into this class
    public sealed override void MyFunc() { }
}

class GrandChildClass : ChildClass
{
    //yields compile error
    public override void MyFunc() { }
}
```

Allow Value Type to Be Null

Scenario/Problem: You need to assign null to a value type to indicate the lack of a value. This scenario often occurs when working with databases, which allow any data type to be null.

Solution: This isn't technically something you need to implement in your class. .NET 2.0 introduced the `Nullable<T>` type, which wraps any value type into something that can be null. It's useful enough that there is a special C# syntax shortcut to do this. The following two lines of code are semantically equivalent:

```
Nullable<int> _id;
int? _id;
```

Let's make the _id field in our `Vertex3d` class `Nullable<T>` to indicate the lack of a valid value. The following code snippet demonstrates how it works:

```
struct Vertex3d : IFormattable, IEquatable<Vertex3d>,
                  IComparable<Vertex3d>
{
    private int? _id;

    public int? Id
    {
        get
        {
            return _id;
        }
        set
        {
            _id = value;
        }
    }
    ...
}
...
Vertex3d vn = new Vertex3d(1, 2, 3);
vn.Id = 3;      //ok
vn.Id = null;   //ok
try
{
    Console.WriteLine("ID: {0}", vn.Id.Value);//throws
}
catch (InvalidOperationException)
{
    Console.WriteLine("Oops--you can't get a null value!");
}

if (vn.Id.HasValue)
{
    Console.WriteLine("ID: {0}", vn.Id.Value);
}
```

Symbols

A

F

G

H

I

J–K

L

M

N

Q

R

T

U

V

W

X–Z