A Detailed look at Overriding the Equality Operator

This item in chinese

It is surprisingly easy to make a mistake when overriding the equality operator. Not only does the equality operator bring along with it a lot of baggage, there is a lot of flawed guidance out there, even on the MSDN website. So we are going to try to clear the air by presenting a systematic breakdown of both a reference type and a value type that supports equality.

For the sake of clarity, reference types will be referred to as classes and value types as structures from here on out.

Operator overloading usually makes more sense in structures than in classes, so structures will always be presented first. The main difference between the two is that for classes, nulls have to be checked for, while in structures one should be aware of potential boxing. This will be illustrated latter on.

Class Signature

The signature for structures are straightforward. One merely needs to tag it with the System.IEquatable interface. Note that there is not a non-generic version of this interface, that role is handled by the base class Object.

```
C#
struct PointStruct : System.IEquatable<PointStruct>

VB
Public Structure PointStruct
     Implements IEquatable(Of PointStruct)
```

For classes, the signature is essentially the same. This does pose a problem in that inheritance can break equality. Specifically, a.Equals(b) won't return the same thing as b.Equals(a) if a is a base type and b is a subtype that overrides the Equals method. This will be handled later by sealing the Equals methods.

```
C#
class PointClass : System.IEquatable<PointClass>

VB
Public Class PointClass      Implements IEquatable(Of PointClass)
```

Fields and Properties

Any field that will be used for equality comparisons must be immutable. This usually means that either all properties on the class are read-only or the class has a unique identifier like a database key.

This rule is vitally important when using anything that relies on hashing. Examples include Hashtable, Dictionary, HashSet, and KeyedCollection. For all of these, the hash code is used for searching and storage. If the hash code changes, the object will be in the wrong slot and the collection will not work correctly. The most common symptom is not being able to find objects that were previously placed in the collection.

To help ensure that the fields stay immutable, they are marked as read-only. This is a bit of a misnomer, as they can be set in constructors. But once fully initialized, there is no way to alter them directly.

```
C#
    readonly int _X;
    readonly int _Y;

    public PointStruct (int x, int y)
    {
       _X = x;
       _Y = y;
        }

    int X
    {
       get { return _X; }
```

```
    }

    int Y
    {
       get { return _Y; }
    }
```

VB

```
    Private ReadOnly m_X As Integer
    Private ReadOnly m_Y As Integer

     Public Sub New(ByVal x As Integer, ByVal y As Integer)
       m_X = x
       m_Y = y
    End Sub

    Public ReadOnly Property X() As Integer
       Get
          Return m_X
       End Get
    End Property

    Public ReadOnly Property Y() As Integer
       Get
          Return m_Y
       End Get
    End Property
```

Since the class versions are nearly identical, or completely identical as in the case of VB, they will not be shown here.

Type-Safe Equality

The first method we implement is the type-safe Equals, which is used by the IEquatable interface.

C#

```
    public bool Equals(PointStruct other)
    {
       return (this._X == other._X) && (this._Y == other._Y);
    }
```

VB

```
    Public Overloads Function Equals(ByVal other As PointStruct) As Boolean _
     Implements System.IEquatable(Of PointStruct).Equals
       Return m_X = other.m_X AndAlso m_Y = other.m_Y
        End Function
```

For classes, an extra check needs to be made for null values. By convention, all non-nulls are considered unequal to null.

You will note that we are not using idiomatic C# code when checking for nulls. This is because of a difference between how C# and VB handle equality.

Visual Basic has an explicit distinction between reference equality and value equality. The former uses the Is operator, while the latter uses the = operator.

C# lacks this distinction and uses the == operator for both. Since we specifically don't want to use ==, which we will be overriding, we have to turn to a backdoor. This backdoor is the Object.ReferenceEquals function.

Since classes are always equal to themselves, there is a check for that before the potentially more expensive equality checks. We are comparing the private fields, but there is no reason we could not have used the properties instead.

C#

```
    public bool Equals(PointClass other)
    {
       if (Object.ReferenceEquals(other, null))
```

```
        {
            return false;
        }
        if (Object.ReferenceEquals(other, this))
        {
            return true;
        }
        return (this._X == other._X) && (this._Y == other._Y);
    }
```

VB

```
    Public Overloads Function Equals(ByVal other As PointClass) As Boolean
   Implements System.IEquatable(Of PointClass).Equals
        If other Is Nothing Then Return False
        If other Is Me Then Return True
        Return m_X = other.m_X AndAlso m_Y = other.m_Y
    End Function
```

Hash Codes

The next step is to generate the hash codes. The easiest way to do this is to simply use an exclusive or on the hash codes of all the fields used for equality.

C#

```
    public override int GetHashCode()
    {
    return _X.GetHashCode() ^ _Y.GetHashCode();
    }
```

VB

```
    Public Overrides Function GetHashCode() As Integer
        Return m_X.GetHashCode Xor m_Y.GetHashCode
    End Function
```

If you do decide to write your own hash code from scratch, you must ensure that it always returns the same hash code for a given set of values. Or in other words, if a and b are equal so are their hash codes.

Hash codes do not have to be unique and unequal values can have the same hash code. That said, they should have a good distribution. Simple returning 42 for every hash code is technically legal, but will kill performance on anything that uses it.

Hash codes should also be very fast to calculate. Since they can be the bottleneck themselves, prefer a fast hash code algorithm with reasonably good distribution to a slow, complicated one with perfectly even distribution.

Equals(Object)

Overriding the base class Equals method is essential, as it is called by other methods including function Object.Equals(Object, Object).

You should note there is a slight performance hit in that the casting is performed twice, once to see if it is valid and a second time to actually perform it. Unfortunately, this is unavoidable with structures.

C#

```
    public override bool Equals(object obj)
    {
        if (obj is PointStruct)
        {
        return this.Equals((PointStruct)obj);
        }
        return false;
    }
```

VB

```
    Public Overrides Function Equals(ByVal obj As Object) As Boolean
       If TypeOf obj Is PointStruct Then Return CType(obj, PointStruct) = Me
    End Function
```

For classes there is a way to reduce this down to a single cast attempt. In the process, we can detect nulls early and skip the method call to Equals(PointClass). Again, C# must use the ReferenceEquals function for the null check.

The methods have been sealed to prevent subclasses from breaking equality.

C#

```
    public sealed override bool Equals(object obj)
    {
    var temp = obj as PointClass;
       if (!Object.ReferenceEquals(temp, null))
    {
          return this.Equals(temp);
       }
       return false;
    }
```

VB

```
    Public NotOverridable Overrides Function Equals(ByVal obj As Object) As Boolean
       Dim temp = TryCast(obj, PointClass)
       If temp IsNot Nothing Then Return Me.Equals(temp)
    End Function
```

Operator Overloading

Now that all the hard work is done, we can actually overload the operators. Here it is as simply as calling the type-safe Equals method.

C#

```
    public static bool operator ==(PointStruct point1 PointStruct point2)
    {
       return point1.Equals(point2);
    }

    public static bool operator !=(PointStruct point1, PointStruct point2)
    {
       return !(point1 == point2);
    }
```

VB

```
    Public Shared Operator =(ByVal point1 As PointStruct, ByVal point2 As PointStruct) As Boolean
       Return point1.Equals(point2)
    End Operator

    Public Shared Operator <>(ByVal point1 As PointStruct,
      ByVal point2 As PointStruct) As Boolean
    Return Not (point1 = point2)
    End Operator
```

For classes, nulls need to be checked for. Fortunately, the Object.Equals(object, object) method handles that for you. Then is calls Object.Equals(Object), which has already been overridden.

C#

```
    public static bool operator ==(PointClass point1, PointClass point2)
    {
    return Object.Equals(point1, point2);
    }

    public static bool operator !=(PointClass point1, PointClass point2)
    {
```

```
        return !(point1 == point2);
    }
```

VB

```
    Public Shared Operator =(ByVal point1 As PointClass, ByVal point2 As PointClass) As Boolean
        Return Object.Equals(point1 ,point2)
    End Operator

    Public Shared Operator <>(ByVal point1 As PointClass, ByVal point2 As PointClass) As Boolean
        Return Not (point1 = point2)
    End Operator
```

Performance

You will notice that each call chain is somewhat lengthy, especially the not equals operator. If performance is a concern, one can speed things up by implementing the comparison logic separately in each method. This can be more error prone and make maintenance tricky, so only do it if you can prove it is necessary using profiling tools.

Testing

The following tests were used for developing this article. It is presented in tabular format to make it easier to translate to your favorite unit test framework. Because equality is so easy to break, this is a prime candidate for unit testing.

Please note that the tests are not comprehensive. One should also test cases where the left and right hand values are partially equal, such as PointStruct(1, 2) and PointStruct(1, 5).

Variable	Type	Value
A	PointStruct	new PointStruct(1, 2)
a2	PointStruct	new PointStruct(1, 2)
B	PointStruct	new PointStruct(3, 4)
nullValue	Object	Null

Expression	Expected Value
Equal values	
a == a2	True
a != a2	False
a.Equals(a2)	True
object.Equals(a, a2)	True
Unequal values, a on left	
b == a	False
b != a	True
b.Equals(a)	False
object.Equals(b, a)	False
Unequal values, a on right	
a == b	False
a != b	True
a.Equals(b)	False
object.Equals(a, b)	False
nulls, a on left	
a.Equals(nullValue)	False
object.Equals(a, nullValue)	False
nulls, a on right	
object.Equals(nullValue, a)	False
Hash codes	
a.GetHashCode() == a2.GetHashCode()	True
a.GetHashCode() == b.GetHashCode()	Indeterminate

Variable	Type	Value

a	PointClass	new PointClass (1, 2)
a2	PointClass	new PointClass (1, 2)
b	PointClass	new PointClass (3,4)
nullValue	PointClass	Null
nullValue2	PointClass	Null

Expression	Expected Value
Same Object	
a == a	True
a != a	False
a.Equals(a)	True
object.Equals(a, a)	True
Equal values	
a == a2	True
a != a2	False
a.Equals(a2)	True
object.Equals(a, a2)	True
Unequal values, a on left	
b == a	False
b != a	True
b.Equals(a)	False
object.Equals(b, a)	False
Unequal values, a on right	
a == b	False
a != b	True
a.Equals(b)	False
object.Equals(a, b)	False
nulls, a on left	
a == nullValue	False
a != nullValue	True
a.Equals(nullValue)	False
object.Equals(a, nullValue)	False
nulls, a on right	
nullValue == a	False
nullValue != a	True
object.Equals(nullValue, a)	False
both null	
nullValue == nullValue2	True
object.Equals(nullValue, nullValue2)	True
Hash codes	
a.GetHashCode() == a2.GetHashCode()	True
a.GetHashCode() == b.GetHashCode()	Indeterminate

Related Editorial

- **Preview of C# 8.x**
- **C# 8 Nullable Reference Types Update**
- **C# 8 Pattern Matching Enhancements**
- **C# 8 Nullable Value Type Enhancements**
- **C# Default Interface Methods Update**