

Re: No Equals on interfaces

Source:

<http://www.tech-archive.net/Archive/DotNet/microsoft.public.dotnet.languages.vb/2004-09/1146.html>

From: Herfried K. Wagner [MVP] (hurf-spam-me-here_at_gmx.at)

Date: 09/07/04

Date: 07 Sep 2004 03:35:11 +0200

* Michi Henning <michi@zeroc.com> scripsit:

>>>> *The documentation is in contradiction to the implementation:*

>>>>

>>> <URL:<http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemtypeclassbas>

>>> [etypetopic.asp](#)>

>>>> *says:*

>>>> *" Interfaces inherit from 'Object' and from zero or more base*

>>>> *interfaces; therefore, the base type of an interface is*

>>>> *considered to be 'Object'. The base interfaces can be determined with 'GetInterfaces' or 'FindInterfaces'. "*

>>> *Huh? This quote supports exactly what I've been saying all along:*

>>> *an interface type "is-a" Object. That's how it should be, and*

>>> *that's perfectly sensible. The compiler simply has a bug, that's all.*

>> *You snipped the code sample.*

>> *Why is the 'BaseType' property 'null' for interfaces in C# too? Why*

>> *does this property behave differently from its*

>> *documentation?*

>

> *Interesting. Your are right: GetType(I).BaseType returns a null reference*

> *in both languages. (Considering that both languages call into the same*

> *library, it's not surprising that the result is the same in both languages.)*

> *But the result itself is surprising.*

>

> *So, I did a bit more digging and found a few interesting snippets.*

>

> *- From the C# language spec:*

>

> *"For purposes of member lookup, a type T is considered to have the following base types:*

>

> *[...]*

>

> *- If T is an interface-type, the base types of T are the base interfaces of T*

> *and the class type object.*

>

> *- If I write*

>

```
> interface I {}
>
> // ...
>
> Type t = typeof(I);
> MethodInfo[] mi = t.GetMethods(BindingFlags.Public |
> BindingFlags.NonPublic |
> BindingFlags.Instance |
> BindingFlags.FlattenHierarchy |
> BindingFlags.Static);
>
> The returned method array is empty. This is in direct conflict with what is stated in the spec.
>
> – I can write
>
> iref.Equals(null)
>
> and the compiler is perfectly happy to let me do that. In other words, the compiler
> allows access to methods on System.Object via an interface reference, and those methods
> do exactly what you'd expect them to do, yet reflection claims that those methods do not
> exist.
>
> – If I write
>
> I iref = null;
> Console.WriteLine(iref is object);
>
> The compiler omits the following warning:
>
> warning CS0183: The given expression is always of the provided ('object') type
>
> The warning is in agreement with the language specification.
>
> However, when I run that program, it prints false! In other words, the compiler says
> that iref "is-a" object, but reflection says iref "is-not-a" object. That's rather sick...
>
> Going through this exercise in VB, we get:
>
> – GetMethods() does not return the methods on Object. Considering that both languages use
> the same reflection library, that's not really a surprise.
>
> – If I write
>
> Dim iref As I = Nothing
> Console.WriteLine(InstanceOf iref Is Object)
>
> the output is "False", which is in accordance with the compiler's idea that interface
> types do not inherit from Object.
>
> So, in summary, here is what I can see:
>
```

microsoft.public.dotnet.languages.vb: Re: No Equals on interfaces

> – *The C# compiler implements the behavior of the C# language specification, but the behavior of reflection at run time is in conflict with that specification.*

ACK.

> – *The VB compiler disagrees with its own language specification, as does the behavior of reflection at run time.*

ACK.

> – *The VB compiler is schizophrenic: it allows me to write `oref = iref` (permitting an implicit widening from an interface type to `Object`), but it doesn't allow me to write `iref.Equals()`. In other words, the compiler applies implicit widening in one situation, but not in the other. The compiler should at least be consistent.*

That's what I don't agree with you and what I am very unsure about.
Let's again take a look at this document:

Visual Basic Language Concepts — Widening and Narrowing Conversions

<URL:<http://msdn.microsoft.com/library/en-us/vbcn7/html/vaconwidenarrowconversions.asp>>

```
"
+-----+-----+
| Data type | Widens to data types |
+-----+-----+
| 'Byte' | 'Byte', 'Short', 'Integer', 'Long', |
| | 'Decimal', 'Single', 'Double' |
+-----+-----+
| ... | ... |
+-----+-----+
| 'Char' | 'Char', 'String' |
+-----+-----+
| ... | ... |
```

[...]

Widening conversions always succeed and can always be performed implicitly.

"

So, why does this code not compile?

```
\\
```

```
' Does not compile.
```

```
Dim c As Char = "a"c
```

```
Dim n As Integer = c.Length
```

```
///
```

'Length' is a method of 'String', and 'Char' widens to 'String'.

I think it is a similar problem for interfaces and widening to 'Object'. Widening doesn't take place when calling a method on a variable of a type where the method is not defined. Widening is defined for assigning objects to others (using the '=' operator in VB.NET), for example, which is a widening conversion:

```
\\
```

```
' Compiles.  
Dim c As Char = "a"  
Dim s As String = c  
///
```

Assume there is a type 'A' that widens to 'B', 'C', 'D', ..., 'Z' (there is no inheritance or implementation taking place between 'A' and 'B', 'A' and 'C', ...

Should I be able to call all methods defined for 'B', 'C', 'D', ..., 'Z' on a variable of type 'A'? I don't think so. Instead, I need a cast, or directly assign the variable to a variable of one of the types the type widens to:

Sample 1:

```
\\  
Dim i As IFoo = Nothing  
Dim o As Object = i ' Compiles, widening conversion.  
Dim t As Type = o.GetType() ' OK.  
///
```

vs.

```
\\  
Dim i As IFoo = Nothing  
Dim t As Type = i.GetType() ' Does not compile, no conversion in code.  
///
```

Sample 2:

```
\\  
Dim c As Char = "a"  
Dim s As String = c ' Compiles, widening conversion.  
Dim n As Integer = s.Length ' Compiles.  
///
```

vs.

```
\\  
Dim c As Char = "a"  
Dim n As Integer = c.Length ' Does not compile, no conversion in code.  
///
```

- > Now, what would happen if that were fixed? I see two issues:
- >
- > Issue 1: What happens if VB is changed to permit the expression *iref.Equals(...)*?

Then it would have to allow the code in the 2nd listing of sample 2 too. Otherwise there would be a new inconsistency. As shown above this will lead to confusing code (for example, a character /does not/ have a length, an /interface/ does not have an 'Equals' method, ...).

- > I don't see a problem here:
- >
- > All VB programs that currently invoke a method on Object via an interface reference
- > can do so only by either using an explicit cast (CType) to Object, or by assigning
- > (or passing) the interface reference to an object reference. If the change is
- > existing programs will continue to work exactly as they do now.

I am against this change. For reasons, see above.

- > Semantically, it makes sense to allow access to the methods on Object via an interface
- > reference: at run time, an interface reference can either be Nothing, or point at
- > an instance. Anything that is instantiated does indeed provide the methods on Object,
- > so it cannot happen that the compiler would permit a method call to an object that, at
- > run time, does not provide the method (i.e., the Smalltalk behavior of calling a
- > non-existent method at run time cannot arise).

ACK. I agree. But currently I don't see a reason for a change. The change would affect more than interfaces only and would change the behavior of the programming language. The proposed change would not be a change that breaks existing code, but when allowing implicit widening conversion on variables directly, then code would be much harder to understand, because nobody knows all widening conversions of type 'A' without taking a look at the documentation.

- > Issue 2: What happens if System.Reflection is changed to return the methods on Object
- > for a reference to an interface?
- >
- > That one is more hairy, because it is difficult to see the ramifications. It may well
- > be that existing code gets unpleasant surprises.

I would not change the current implementation, but instead update the documentation at the relevant places..

- > Note that the alternative of changing C# to behave like VB is not viable, because that
- > would definitely break a lot of existing code.

I think that a change of what C# does would (theoretically) be the best solution, but there is no reason to change what C# is currently doing. VB.NET sticks to the rules given by the implementation in the .NET Framework. The documentation is partly out of date, and the C# specs are based on this documentation.

microsoft.public.dotnet.languages.vb: Re: No Equals on interfaces

(Please excuse bugs in this post, it's about 3:00 AM in Austria and I am very tired!)

--

M S Herfried K. Wagner
M V P <URL:<http://dotnet.mvps.org/>>
V B <URL:<http://dotnet.mvps.org/dotnet/faq/>>