

Re: changing access modifier of base method

Source:

<http://www.tech-archive.net/Archive/DotNet/microsoft.public.dotnet.languages.csharp/2008-01/msg03098.html>

- *From:* "Jon Skeet [C# MVP]" <skeet@xxxxxxxx>
 - *Date:* Wed, 23 Jan 2008 01:33:12 -0800 (PST)
-

On Jan 23, 9:11 am, "Peter Duniho" <NpOeStPe...@xxxxxxxxxxxxxxxxxxxx> wrote:

It's got 3 meanings that I can think of, two of which are very similar:

- 1) Equivalent to readonly for variables, including local variables
- 2) Equivalent to sealed on a class
- 3) Equivalent to sealed on a method

I have seen it used as an equivalent to "const". That is, according to at least some Java articles I've read, variables declared with an constant assignment don't even wind up allocating space. The compiler just hard-codes the value in the code, like a const declaration in other languages. I'm not sure if this counts as a different meaning than your #1...I'm not even sure if it's even an actual meaning (I've found varying quality in the Java resources I've seen :)).

That's a specialist meaning of #1 really. And the space is still there at runtime, but any references to it are hardcoded. So if I have:

```
public class Foo
{
public static final int SOME_SHOUTY_CONSTANT = 1;
}
```

and then:

```
public class Bar
{
void X()
{
int x = Foo.SOME_SHOUTY_CONSTANT;
}
```

Re: changing access modifier of base method

```
}
```

then that's exactly equivalent (in Bar.X) to
int x = 1;

The Foo class still has SOME_SHOUTY_CONSTANT available at execution time, but references to it are resolved at compile-time. This is exactly the same as "const" in C# – with the attendant issues, too.

I saw a page on the Java web site that had a long list of the different contexts in which "final" could be used. It's possible (probable?) that the number of different semantic meanings was much less, given the overlap between inheritance behaviors in different members. But I think there might be more than 3 different meanings. :)

Hmm... I *suspect* all of those extra meanings fit into my main 3, just in different ways – but I'd have to see the list to check.

<snip>

It would be a pain to write in such a language, but probably instructive.

Well, even though I know it doesn't really matter from a performance perspective, I still prefer non-virtual methods over virtual because of the slight overhead of virtual methods.

It definitely matters from a performance perspective on .NET. It doesn't matter in Java with HotSpot, because that is capable of inlining a virtual method until it sees something overriding it, and then undoing the optimisation.

It's hard to know for sure until you're in that position, but I believe that I'd opt for making everything sealed by default, and only virtual where I specifically want it.

I'd agree with that, but for design reasons rather than performance reasons.

In addition to the performance issue, I also think there's a legitimate design reason for making sealed the default. Most of my classes contain many more private members than public. Lots of little helper functions, for example, that will never be overridden. Sure, I could explicitly make

Re: changing access modifier of base method

those all "final" in Java, but what a pain that would be.

C# did the right thing when it came to methods. Shame about classes :(

(I think private methods are final by default in Java though – you certainly can't make private methods virtual in C#.)

Likewise, even for most of the public methods, I don't intend for the methods to be overridden. I suppose there, there's less reason to be picky about that, but I think it can still be dangerous.

I'd say there's much **more** reason to be picky about public methods – you're defining a contract. You can change your private stuff later without fear of breaking things, which isn't true of public methods.

IMHO, allowing a member to be overridden requires some forethought as to how that member might be overridden.

Exactly. See <http://msmvps.com/blogs/jon.skeet/archive/2006/03/04/inheritancetax.aspx> for more on my view of this.

So to some extent, on top of my performance bias, for me it comes down to the frequency of one or the other. Since my virtual members are almost always outnumbered by a large proportion by the sealed members, making virtual the default doesn't make sense. Either I wind up with stuff that's virtual and really shouldn't be (which is what's happening to me in Java right now), or I waste a lot of time explicitly making things sealed.

For me, the most appropriate default is the one which can do the least damage. Even if I were to want more public methods than private methods, I like the default being private in C#. If you make a method more private than it should be, you tend to find out about it quickly because people can't access it. If you make a method more **public** than it should be, you'll only find out about it when you want to change it later and find that it's part of a contract.

Even in my so-far-brief foray into Java, I can see some value in making everything virtual by default. It leads to very flexible inheritance. On the other hand, it leads to very flexible and sometimes unpredictable inheritance too. :) I'm not sure that encouraging developers to make everything virtual is really the way to go. It might be fine for people who live and breathe OOP and love the infinite possibilities. But for someone like me who basically uses OOP, or any language for that matter,

Re: changing access modifier of base method

just as a means to some other end (like getting my program to work :)), it just seems to get in the way and make things riskier.

When methods are virtual, you bleed implementation details. If I have two virtual methods, X and Y, and X calls Y, I have to document that – because otherwise someone might override Y to just call X. That may look legitimate and **be** legitimate with an alternative implementation, but in that particular implementation it would lead to a stack overflow.

I don't like bleeding implementation details, thus I don't like having too many virtual methods.

The one benefit I can see of making everything virtual is that it makes testing easier. I don't think the cost is worth the benefit though. (Many TDDers see the benefit, but I don't believe they all consider the cost.)

Jon

.