

Re: C# Language Proposal for 'out' Parameters

Source:

<http://www.tech-archive.net/Archive/DotNet/microsoft.public.dotnet.languages.csharp/2004-04/3614.html>

From: Daniel O'Connell [C# MVP] (*onyxkirx_at_--NOSPAM--comcast.net*)

Date: 04/14/04

Date: Wed, 14 Apr 2004 18:54:56 -0500

"Bruno Jouhier [MVP]" <bjouhier@club-internet.fr> wrote in message news:%237sLKSIIIEHA.3968@TK2MSFTNGP12.phx.gbl...

>> *If the assignment isn't made because an exception has been thrown, I*

>> *think that's fine – that's what copyout means (as I understand it) –*

>> *not that every time the assignment is made, the property is written.*

>

> *I agree, and I don't fully understand Daniel's point.*

>

> *a call like*

> *Foo(inout expression) // copyin/copyout semantics*

> *behaves "exactly" like:*

> *expression = Foo(expression)*

>

> *and a call like*

> *Bar(out expression) // copyout semantics*

> *behaves "exactly" like:*

> *expression = Bar()*

>

> *We write this all the time, and we don't have any real issue with these*

> *constructs. So, I don't see why copyin/copyout argument passing would*

> *create*

> *a new problem (Daniel, I don't understand if your issue is on the caller*

> *or*

> *callee's side but I don't see any real issue on the callee's side either).*

>

> *On the other hand, I think that copyin/copyout would be slightly simpler*

> *to*

> *explain than "ref" semantics and we would not get all these posts from*

> *people who get confused between "passing by reference" and "passing the*

> *reference" (fortunately Jon has a very good page on this one).*

>

Sometimes I hate trying to explain these issues. I can see neither of you understand what I am trying to say since your responses have been addressed at issues slightly to the left of what I'm trying to explain. I will try again, hopefully with more success.

The earliest syntax example I saw was something akin to

```
int value = obj.Prop;
if (GetNewValue(ref value))
{
    obj.Prop = value;
    //do some stuff
}
```

the suggested approach of

```
if (GetNewValue(inout obj.Prop))
{
    //do some stuff
}
```

results in `obj.Prop` is assigned no matter what happens with the exception of an exception (pun not intended). The `inout` code is actually closer to

```
int value = obj.Prop
bool returnValue = GetNewValue(ref value)
obj.Prop = value;
if (returnValue)
{
    //do some stuff.
}
```

That is, frankly, less than desirable often and IMHO reduces the utility of `inout` semantics, not to mention reducing overall clarity of the code. It of course also results in subtly different semantics, the two examples above are different. You could not use code checks to ensure that a non-changing variable doesn't get reassigned to the property because properties are code backed, a value not changing doesn't mean an assignment won't change things. Not assigning the property would be terribly unpredictable.

However, I think the above code pattern is *far* less common than the original example, it is in my experience anyway. The mere fact that an `out` or `ref` parameter is being used suggests that more data has to come out of the method than the return type can handle. In most cases this is going to be either a success/failure code or another piece of data that may be used to decide how to proceed. I can't think of any reasonable examples where this isn't the case. The fundamental problem is that `GetNewValue` would *have* to throw an exception if it doesn't want to cause a property reassignment. Now to answer the question of where the problem is, IMHO it is in three places: 1) `GetNewValue` itself, 2) the code calling `GetNewValue`, and 3) the property get/set accessors. You have to consider the ramifications of using `inout` in all three of these pieces of code. You can admittedly get around 3 by writing the code that calls `GetNewValue` without using `inout` when possible, but that forces you to consider the effects of the property setter in ways that you wouldn't using the original pattern of setting the property within the `if` block. If you can't write the value of a get method into a set and still get the same value from get then you can't pass the property to an

inout parameter(
theres a mouthful, to be clear, I mean

```
int value = obj.Prop;  
obj.Prop = value;  
if (obj.Prop == value || obj.underlyingField1 == old obj.underlyingField1 ||  
... obj.underlyingFieldN == old obj.underlyingFieldN)  
    //all is well  
else  
    //we have big problems  
)
```

However, unless you wrote the property or have very clear documentation on the property, you probably shouldn't assume that the assignment has no side effects. In the normal flow of a program, an explicit assignment clearly states that the value will be set in most circumstances, again with the exception of an exception being thrown. In that situation you are clearly recognizing that there are sideeffects and that the assignment definitely needs to happen. With an inout parameter there is the question of if assignment should happen or not. Because of this the original example is moot, the proposal doesn't actually solve the problem. TO me that makes it less useful than it seems. It forces exceptions on you to achieve code that uses a simple if in the current method, in code that should be the more common pattern in any case.

My original point refuting "successful returns" comes from that **any** return must perform the copy. You cannot design a feature that relies on a return value to determine its behaviour, it is just **way** too complicated and unpredictable. Nor could you rely on an underlying block to contain the copyout code. There is no such thing as a successful return, just a return or an exception. a return means copy, an exception behaves as an exception. The semantics are identical to out currently, with the exception that inout would permit the use of properties and would therefore add quite a bit of unpredictable behaviour to the mix. If you remove access to properties, and allow just fields and locals as is the behaviour of out, and I have no problem. And I actually think that inout would have been a better choice than ref with the exception of overly large structs(which you shouldn't really have anyway, eh?).