Arrays considered somewhat harmful

SEPT. 22, 2008

Eric Lippert

I got a moral question from an author of programming language textbooks the other day requesting my opinions on whether or not beginner programmers should be taught how to use arrays.

Rather than actually answer that question, I gave him a long list of my opinions about arrays, how I use arrays, how we expect arrays to be used in the future, and so on. This gets a bit long, but like Pascal, I didn't have time to make it shorter.

Let me start by saying when you definitely should not use arrays, and then wax more philosophical about the future of modern programming and the role of the array in the coming world.

You probably should not return an array as the value of a public method or property, particularly when the information content of the array is logically immutable. Let me give you an example of where we got that horridly wrong in a very visible way in the framework. If you take a look at the documentation for System.Type, you'll find that just looking at the method descriptions gives one a sense of existential dread. One sees a whole lot of sentences like *"Returns an array of Type objects that represent the constraints on the current generic type parameter."* Almost every method on System.Type returns an array it seems.

Now think about how that must be implemented. When you call, say, GetConstructors() on typeof(string), the implementation cannot possibly do this, as sensible as it seems.

```
public class Type {
  private ConstructorInfo[] ctorInfos;
  public ConstructorInfo[] GetConstructors()
  {
    if (ctorInfos == null) ctorInfos = GoGetConstructorInfosFromMetadata();
    return ctorInfos;
  }
```

Why? Because now the caller can take that array and *replace the contents of it with whatever they please*. Returning an array means that you have to make a fresh copy of the array every time you return it. You get called a hundred times, you'd better make a hundred array instances, no matter how large they are. It's a performance nightmare – particularly if, like me, you are considering using reflection to *build a compiler*. Do you have any idea how many times a second I try to get type information out of reflection? Not nearly as many times as I could; every time I do it's another freakin' array allocation!

The frameworks designers were not foolish people; unfortunately, we did not have generic types in .NET 1.0. clearly the sensible thing now for GetConstructors() to return is IList<ConstructorInfo>. You can build yourself a nice read-only collection object once, and then just pass out references to it as much as you want.

What is the root cause of this malaise? It is simple to state: The caller is requesting *values*. The callee fulfills the request by handing back *variables*.

An array is a collection of variables. The caller doesn't *want* variables, but it'll take them if that's the only way to get the values. But in this case, as in most cases, *neither the callee nor the caller wants those variables to ever vary*. Why on earth is the callee passing back variables then? Variables vary. Therefore, a fresh, different variable must be passed back every time, so that if it does vary, nothing bad happens to anyone else who has requested the same values.

If you are writing such an API, wrap the array in a ReadOnlyCollection<T> and return an IEnumerable<T> or an IList<T> or something, but not an array. (And of course, do not simply cast the array to IEnumerable<T> and think you're done! That is still passing out variables; the caller can simply cast back to array! Only pass out an array if it is wrapped up by a read-only object.)

That's the situation at present. What are the implications of array characteristics for the future of programming and programming languages?

Parallelism Problems

The physics aspects of Moore's so-called "Law" are failing, as they eventually must. Clock speeds have stopped increasing, transistor density has stopped increasing. The laws of thermodynamics and the Uncertainty Principle are seeing to that. But manufacturing costs per chip are still falling, which means that our only hope of Moore's "Law" continuing to hold over the coming decades is to cram more and more processors into each box.

We're going to need programming languages that allow mere mortals to write code that is parallelizable to multiple cores.

Side-effecting change is the enemy of parallelization. Parallelizing in a world with observable side effects means locks, and locks means choosing between implementing lock ordering and dealing with random crashes or deadlocks. Lock ordering requires global knowledge of the program. Programs are becoming increasingly complex, to the point where one person cannot reasonably and confidently have global knowledge. Indeed, we prefer programming languages to have the property that programs in them can be understood by understanding one part at a time, not having to swallow the whole thing in one gulp.

Therefore we tools providers need to create ways for people to program effectively *without causing observable side effects*.

Of all the sort of "basic" types, arrays most strongly work against this goal. An array's whole purpose is to be a mass of mutable state. Mutable state is hard for both humans and compilers to reason about. It will be hard for us to write compilers in the future that generate performant multi-core programs if developers use a lot of arrays.

Now, one might reasonably point out that List<T> is a mass of mutable state too. But at least one could create a threadsafe list class, or an immutable list class, or a list class that has transactional integrity, or uses some form of isolation or whatever. We have an extensibility model for lists because *lists are classes*. We have no ability to make an "immutable array". Arrays are what they are and they're never going to change.

Conceptual Problems

We want C# to be a language in which one can draw a line between code that implements a mechanism and code that implements a policy.

The "C" programming language is all about mechanisms. It lays bare almost exactly what the processor is actually doing, providing only the thinnest abstraction over the memory model. And though we want you to be able to write programs like that in C#, most of the time people should be writing code in the "policy" realm. That is, code that emphasizes *what the code is supposed to do,* not *how it does it*.

Coding which is more declarative than imperative, coding which avoids side effects, coding which emphasizes algorithms and purposes over mechanisms, that kind of coding is the future in a world of parallelism. (And you'll note that LINQ is designed to be declarative, strongly abstract away from mechanisms, and be free of side effects.)

Arrays work against all of these factors. Arrays demand imperative code, arrays are all about side effects, arrays make you write code which emphasizes how the code works, not what the code is doing or why it is doing it. Arrays make optimizing for things like "swapping two values" easy, but destroy the larger ability to optimize for parallelism.

Practical Problems

And finally, given that arrays are mutable by design, the way an array restricts that mutability is deeply weird. All the *contents* of the collection are mutable, but the *size* is fixed. What is up with that? Does that solve a problem anyone actually has?

For this reason alone I do almost no programming with arrays anymore. Arrays simply do not model any problem that I have at all well – I rarely need a collection which has the rather contradictory properties of being *completely mutable*, and at the same time, *fixed in size*. If I want to mutate a collection it is almost always to add something to it or remove something from it, not to change what value an index maps to.

We have a class or interface for everything I need. If I need a sequence I'll use IEnumerable<T>, if I need a mapping from contiguous numbers to data I'll use a List<T>, if I need a mapping across arbitrary data I'll use a Dictionary<K,V>, if I need a set I'll use a HashSet<T>. I simply don't need arrays for anything, so I almost never use them. They don't solve a problem I have better than the other tools at my disposal.

Pedagogic Problems

It is important that beginning programmers understand arrays; it is an important and widely used concept. But it is also important to me that they understand the weaknesses and shortcomings of arrays. In almost every case, there is a better tool to use than an array.

The difficulty is, pedagogically, that it is hard to discuss the merits of those tools without already having down concepts like classes, interfaces, generics, asymptotic performance, query expressions, and so on. It's a hard problem for the writer and for the teacher. Fortunately, for me, it's not a problem that I personally have to solve.

Georgi

Very interesting point of view!

Your arguments against arrays sound very strong and I personally agree with them.

It will be interesting if there is anyone who can advocate poor arrays :)

JaredPar MSFT

Instead of GetConstructors() returning IList<ConstructorInfo>, I would prefer the framework actually define a readonly list interface such is IReadOnlyList<T> and return an instance of that. Returning IList<T> for a

readonly list is a bad idea because you're not actually returning an IList<T>. You're returning an object that is kind of an IList<T> since it can't fulfull several of the methods.

Jonathan Allen

Though probably not what want. IEnumerable<T> is effectively a read-only list.

Jeff

How strange, I was tackling this very issue myself just this afternoon. I wanted to use the array because the code DOM serializer can persist them in nice ways, however, I was aware of the mutable nature of their contents. I managed to solve my serializer problem by having a constructor take IEnumerable<string>, provide the array to consumers as a read-only IList<string> property, and have my type converter wrap the IList in a List<string> and use ToArray with an instance descriptor to my IEnumerable<string> constructor. This way the code DOM serializer persists nicely (rather than persist an IEnumerable which it insists on handling as a resource blob), but my constructor supports more collections and my array contents are immutable for the lifetime of the constructed object.

Of course, I could've worked on writing a nicer serializer, but this route was simpler.

Having been caught out by arrays in many of the ways you mention, I whole-heartedly concur with you.

JaredPar MSFT

@Jonathan,

It's more of a sequence than a list though. List's have several properties that set them apart from sequences. Namely O(1) random access and O(1) size calculation. In this situation we're converting from an array to a new data structure. It seems more natural to go with a list since we already have all of the elements grouped together.

Then again, if reflection could be done more efficiently "on demand" then a sequence would potentially be better.

Peter Thatcher

I like how you bring the topic of arrays back into the more general topic of mutable data. I've been writing lots of Python the last two years, and over that time my style has morphed into almost exclusivly using immutable types. Most of the time, I even use a tuple as a collection instead of an array simple because it's immutable. Clearly, a library writer cannot safely pass out references to internal mutable data.

I also like how you steer the topic of mutability into the topic of concurrency since, as you describe, they are so closely related. Again, I've found that over the last few years my style of programming has morphed, not just from mutable data to immutable data, but from thread-based concurrnecy to actor-based concurrency.

You describe a good solution for C# to address readonly/immutable data, but I've yet to see a good solution in C# for concurrency. Of course, if you addressed THAT in this post, it would have become quite a long post! Still, I'm very curious to see what future versions of C# do to address concurrency. It's an incredibly important topic and no one except the Erlang people seem to have taken it head-on, and everyone seems to think they are crazy for one reason or anothoer.

Will we be seeing a "threads considered somewhat harmful" post soon?

danyel

arrays are good for image processing and science applications: each cell represents a physical object or an atom or a pixel. for images, the size really is fixed but the pixels change.

that said, I don't know of other places where they are a good model, and your article is generally spot on.

Thomas

The problem with returning arrays from methods is actually a subset of a more general problem: returning references to private data. The very same argument you make against returning arrays can also be made against returning another object.

Also, I don't think that it is bad to return an array casted to IEnumerable<T>. You argue that the caller can simply cast it back -- true. But even if we wrap it in another object, the caller can still access it; if not through reflection, then through unsafe memory operations or whatever. If the caller insists on breaking your code, he will be able to do so regardless.

Jon Davis

I like arrays for some things. For example, String.Split(). I use that frequently, and then I modify the contents of individual indexes frequently. If Split() returned ReadOnlyCollection<string> I'd end

up converting that to a string array, frequently.

Likewise, any time you're working with fixed-length linear data, arrays are appropriate.

However, I fully agree with the occasional scenario where arrays were used only because generics weren't available, such as in the scenario you demonstrated with, which quite honestly System.Reflection is one of few places I've ever been really annoyed that .net gave me an array. That said, my reasons were quite the opposite from yours. I wanted *more* mutability. For example, GetProperties returns an array and say I wanted to loop through the list and remove the items that didn't match certain criteria, such as checking for certain custom attributes. In both cases of ReadOnllyCollection<PropertyInfo> and PropertyInfo[], I have to convert to a List<PropertyInfo> to do that. Quite a pain.

So in other words, no one can be happy.

Andrey Shchekin

I completely agree with Jared.

Having an interface that is essentially IEnumerable<T>+this[int] {get;} (IIndexedEnumerable?) woud be the best solution for this problem. Also this interface would solve the future problem of result variance.

Chris Nahr

Surely this issue would be best addressed by adding a read-only array at the CLR level, or even better a recursive read-only tag for arbitrary types? Then we could have C++ style const-ness as well...

Anders Borum

I second the request for an expanded interface for read only collections (but perhaps we should take this to the BCL team instead).

The more I read on this blog, the more I'm looking forward to PDC. Keep them coming Eric!

Frank Quednau

Superb post. Thanks for that. Story of Engineering. The very foundations and thought asssumptions can crumble at any time. It is ironic how unsuited Arrays are for anything. Most of the time I use arrays in situations where I mean "Block of information for you to look at, not touch it". Indeed that construct is utterly unsuited.

Will it not be possible in the future to state that an array should be read-only? Or is it impossible for some known unknown ? Or is it even a silly question because I still don't get it? ;)

Konstantin Triger

To Jon Davis:

If you need to apply a certain criteria on the list content, you should apply a predicate on it (i.e. myList.Where(...)).

In general, if you need to change the [object] stream content, produce a new stream with correct content. This will comply with goals of this article. Basically that's what LINQ does.

Kosta

Paul Hill

My fingers are crossed that CLR embraces Const at some point in the future. Const arrays the enforce immutability at compile-time would be wonderful.

Original URL:
http://blogs.msdn.com/b/ericlippert/archive/2008/09/22/arrays-considered-somewhat-harmful.aspx