Why do arrays support IList?

Asked 15 years, 3 months ago Modified 12 years, 2 months ago Viewed 265 times



6

The IList interface requires an Add method. Arrays implement this function but it simply throws a NotImplementedException. This seems like very bad design to me.



What were the designers thinking when they did this?



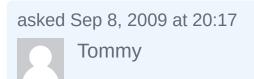


c# arrays ilist



Improve this question

Follow



2 Answers

Sorted by:

Highest score (default)





ILists can be readonly - if in doubt the caller can test the IsFixedSize property before attempting to add or remove an element, or the IsReadOnly property before attempting to modify an element.



An array is a fixed-size IList.



1

It can be convenient to be able to treat an array as a list. One example is mocking a data access method that returns an IList - it can be mocked to simply return an array cast as an IList.

Share Improve this answer Follow

edited Sep 9, 2009 at 4:58

answered Sep 8, 2009 at 20:19



It's actually the IsFixedSize property that you'd need to check. IsReadOnly will be false for arrays because the existing elements can be modified. IsFixedSize will be true because elements can't be added or removed.

- LukeH Sep 8, 2009 at 21:26

Just to clarify the problem with fixed and read only: blogs.msdn.com/ericlippert/archive/2009/08/27/... – Oliver Sep 9, 2009 at 7:02

@Oliver: I'm a big fan of Eric Lippert's blog, but that article has absolutely nothing to do with the IsFixedSize or IsReadOnly properties. – LukeH Sep 9, 2009 at 10:33



0



Because it is common for different objects to have different abilities, some interfaces include members which will be implemented by some but not all implementations. If some, but not all, implementations of a hypothetical interface Ivehicle would be able to attach a trailer, a typical pattern would be to define the function of

AttachTrailer as "Attempt to attach a trailer, if CanAttachTrailer is true, or else throw NotSupportedException". All implementations of Ivehicle would be able to conform to the above specification, whether or not they can handle trailers.

An advantage of this approach is that it is possible for implementations of an interface to offer many different combinations of features, without having to define different types for different combinations. Another advantage of this approach is that it is possible for a method Foo which receives an object that includes capabilities that Foo doesn't need, to pass that object along to method Bar, which does need those capabilities, without requiring any typecasts anywhere, and without Foo knowing what capabilities Bar will need. Yet another advantage is that this makes it easy to write code which doesn't need certain capabilities, but can take advantage of them when they exist.

There are some disadvantages to this approach, however. There is as yet no way for an interface to specify default implementations for any properties or methods. Consequently, even Ivehicle implementations which cannot attach trailers will need to include code to return false to the CanAttachTrailer property, and throw an exception in their AttachTrailer method. Further, since the interface imposes no requirement that many of its methods be implemented, there is no way a compiler can know to reject at attempt to call to a function

that requires a capability with an object of a type which cannot provide it.

When designing IList<T>, Microsoft apparently thought that the advantages of the "optional-capabilities interface" approach outweighed the disadvantages. Indeed, if .net provided a means for classes implementing interfaces to defer to default implementations of members they don't wish to provide, there would be little reason not to include many optional capabilities in base-level interfaces; to allow compile-time enforcement of necessary capabilities, one could have a number of interfaces derive from a base which includes all the needed members, and specify that classes implementing the latter interfaces must implement certain members in ways that are actually useful. For example, Microsoft could have defined IResizableList<T> to inherit IList<T> without adding any members, but with the expectation that IList<T> implementations that allowed resizing would implement the latter interface, while those which did not allow resizing would not implement it. Had they done that, code which needed to be able to resize a list could demand an IResizableList<T> (in which case it would not accept an array), while code which did not need to resize a list could demand an IList<T>). Unfortunately, Microsoft didn't do anything like that, so it's not possible for code to demand a resizable list at compile time--all it can do is squawk if a passed-in list reports itself as fixed-size.