

COLLECTIONS

Collections of things



A collection is a value that contains other values.



A collection has items of the same type.



Examples include arrays, stacks, set, dictionaries, and lists.



We can get far with just using lists (dynamic arrays).

Arrays

- The grandfather of collections
- Most languages start with a built-in array
- Supports fast indexing
- Fast length queries
- Length is fixed upon construction

[Learn](#) / [.NET](#) / [API browser](#) / [System](#) /

Array Class

Reference

Definition

Namespace: [System](#)

Assembly: [System.Runtime.dll](#)

Provides methods for creating, manipulating, searching, and sorting arrays, thereby serving as the foundation for the common language runtime.

C#

```
public abstract class Array : ICloneable, System.Collections.IList,  
    System.Collections.IStructuralComparable, System.Collections.IStructuralEquatable
```

Inheritance [Object](#) → [Array](#)

Implements [ICollection](#) , [IEnumerable](#) , [IList](#) , [IStructuralComparable](#) , [IStructuralEquatable](#) , [IClonable](#)

<https://learn.microsoft.com/en-us/dotnet/api/system.array?view=net-6.0>

SYSTEM.ARRAY

Changing Items

- You can change the values within an array
- You do it by assigning a value to the indexer ([] operator)
- It is the first class that we have seen so far that allows that.
- This is useful, but you must be cautious

Array.Resize() is Misleading

```
[Test]
public static void ArrayResizeDemo()
{
    var xs = new int[] { 1, 2, 3 };
    var ys = xs;
    OutputArray(xs);
    OutputArray(ys);
    xs[0] = 5;
    ys[1] = 4;
    OutputArray(xs);
    OutputArray(ys);
    Array.Resize(ref xs, 4);
    OutputArray(xs);
    OutputArray(ys);
    xs[0] = 1;
    ys[1] = 2;
    OutputArray(xs);
    OutputArray(ys);
}
```

✓ ArrayResizeDemo

Source: [ArrayAndListDemos.cs](#) line 51

Duration: 2 ms

Standard Output:

```
Contents of array is: 1, 2, 3
Contents of array is: 1, 2, 3
Contents of array is: 5, 4, 3
Contents of array is: 5, 4, 3
Contents of array is: 5, 4, 3, 0
Contents of array is: 5, 4, 3
Contents of array is: 1, 4, 3, 0
Contents of array is: 5, 2, 3
```

Only rarely do you need “ref”

- The “ref” keyword is used to pass a variable by reference
- You use it to change what the passed variable points to
- Be nice to your variables, let them live their happy lives they were intended

```
[Test]
public static void CreateArrays()
{
    var array1 = new int[] { 1, 2, 3, 4 };
    var array2 = new[] { 1, 2, 3, 4 };
    Assert.AreEqual(array1, array2);
    Console.WriteLine($"The contents of array are {array1}");
    Console.WriteLine("The compiler says the type is an int[]");
    Console.WriteLine($"The type of array1 is {array1.GetType()}");
}
```

✓ CreateArrays

Source: [ArrayAndListDemos.cs](#) line 12

⌚ Duration: 12 ms

Standard Output:

The contents of array are System.Int32[]
The compiler says the type is an int[]
The type of array1 is System.Int32[]

RUNTIME TYPE OF AN ARRAY

Why don't I see contents?

- Converting arrays to strings can consume a lot of resources
- The "Array.ToString()" just returns the type of the array
- Manually convert to a string using String.Join

```
[Test]
public static void ArrayToString()
{
    var array = new object[] { 'a', 12, "hello", 3.14f };
    Console.WriteLine(string.Join(";", array));
}
```

✓ ArrayToString
Source: [ArrayAndListDemos.cs](#) line 23
Duration: 8 ms
Standard Output:
a;12;hello;3.14

Common Array Operations

- `Array.Length()` – returns the length of the array
- `Array.CopyTo()` – copies the array, or a part of it, into another array
- `Array.IndexOf()` – find the index of a specific element

Array.FindIndex()

Array.FindIndex Method

Reference

 [Feedback](#)

Definition

Namespace: [System](#)

Assembly: [System.Runtime.dll](#)

Searches for an element that matches the conditions defined by a specified predicate, and returns the zero-based index of the first occurrence within an [Array](#) or a portion of it.

A PREDICATE

A function that takes a single value as input
and returns a Boolean.

Array.FindIndex() Demo

```
public static bool EndsWithY(string s)
{
    return s.EndsWith("y");
}

[Test]
public static void TestFindIndex2()
{
    var index = Array.FindIndex(Months, EndsWithY);
    var month = index < 0 ? "no month" : Months[index];
    Console.WriteLine($"First month that ends with y is {month}");
}
```

The ternary conditional operator - ?:

- Takes a Boolean value (Cond) and two other argument (X and Y)
- Returns X if Cond is True
- Returns Y if Cond is False
- Short circuit evaluation

<https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/conditional-operator>

Without the Ternary Operator

```
[Test]
public static void TestFindIndex3()
{
    var index = Array.FindIndex(Months, EndsWithY);
    string month;
    if (index >= 0)
    {
        month = Months[index];
    }
    else
    {
        month = "no month";
    }
    Console.WriteLine($"First month that ends with y is {month}");
}
```

```
[Test]
public static void ShortCircuitEvaluationDemo()
{
    var x = IsWeekend() ? MyFunc("Party") : MyFunc("Work");
    Console.WriteLine(x);
    PrintAOrB(IsWeekend(), MyFunc("Party"), MyFunc("Work"));
}
```

✓ ShortCircuitEvaluationDemo
Source: [ArrayAndListDemos.cs](#) line 66
Duration: 4 ms

Standard Output:

MyFunc is being called with Work
Work
MyFunc is being called with Party
MyFunc is being called with Work
Work

SHORT CIRCUIT EVALUATION

```

[Test]
[TestCase(true, true)]
[TestCase(false, false)]
[TestCase(true, false)]
[TestCase(false, true)]
public static void TestBoolean(bool a, bool b)
{
    var notA =
    var notB =
    var and =
    var or =
    var xor =

    Assert.AreEqual(!a, notA);
    Assert.AreEqual(!b, notB);
    Assert.AreEqual(a && b, and);
    Assert.AreEqual(a || b, or);
    Assert.AreEqual(a ^ b, xor);
}

```


IMPLEMENTING BOOLEAN OPERATORS

It is possible to implement
the Boolean operations with
only the ternary operator.
Try it!

FindIndex<T>(T[], Predicate<T>)

Searches for an element that matches the conditions defined by the specified predicate, and returns the zero-based index of the first occurrence within the entire [Array](#).

C#

 Copy

```
public static int FindIndex<T> (T[] array, Predicate<T> match);
```

Type Parameters

T

The type of the elements of the array.

Parameters

array [T\[\]](#)

The one-dimensional, zero-based [Array](#) to search.

match [Predicate<T>](#)

The [Predicate<T>](#) that defines the conditions of the element to search for.

Returns

[Int32](#)

The zero-based index of the first occurrence of an element that matches the conditions defined by **match**, if found; otherwise, -1.

What is the TypeParameter T?

- `Array.FindIndex()` is a [Generic Method](#)
- This means it has type parameters.
- Constrains the type of the elements in the array to match the type of the predicate parameter
- Results in compile-time errors if misused

125
126
127
128
129
130
131
132
133
134
135
136

```
[Test]
public static void TestFindIndexNoCompile()
{
    var xs = new int[] { 1, 2, 3, 4 };
    var index = Array.FindIndex(xs, EndsWithY);
    Console.WriteLine($"Found the index {index}");
}

[Test]
public static void SpookyEffects()
{
```

%
List
ire Solution
ch Error List

8
7

1 Error
0 of 24 Warnings
0 of 11 Messages

Build + IntelliSer

	Code	Description	Project	File	Line	Suppressi
✖	CS1503	Argument 2: cannot convert from 'method group' to 'Predicate<int>'	DemosAndTests	ArrayAndListDemos.cs	130	Active

Example of a Generic Method

```
public static void OutputArray<T>(T[] array)
{
    var content = string.Join(", ", array);
    Console.WriteLine($"Contents of array is: {content}");
}
```

We actually rarely use “object”

- Upcasting to object loses useful type information
- Compiler becomes less effective at telling us our mistakes
- Always choose generic methods where appropriate

Array.FindIndex() Demo 2

```
static string[] Months = new[]  
{  
    "January", "February", "March", "April",  
    "May", "June", "July", "August",  
    "September", "October", "November", "December"  
};
```

```
[Test]  
public static void TestFindIndex()  
{  
    var index = Array.FindIndex(Months, m => m.EndsWith("y"));  
    var month = index < 0 ? "no month" : Months[index];  
    Console.WriteLine($"First month that ends with y is {month}");  
}
```

My First Lambda

```
m => m.EndsWith("y")
```


A function without a name is as lovely

- Lambdas are also called anonymous functions
- We are just defining a small inline local function
- Notice that the types of the parameter and return type are inferred
- The function we pass it to tells the compiler the type
- More convenient than defining named functions for single-use

Mutating Array Operations

- `Array.Reverse` – reverses order of elements
- `Array.Sort` – sorts elements in ascending order
- `Array.Fill` – fills array with same element
- `Array.ConvertAll()` – calls a converter function to transform each element

What Happens?

```
[Test]
public static void SpookyEffects()
{
    var xs = new int[] { 17, 3, 512, 99, 9 };
    Console.WriteLine($"The third value is {xs[2]}");
    var min = GetMinValue(xs);
    Console.WriteLine($"The minimum value is {min}");
    Console.WriteLine($"The third value is {xs[2]}");
}
```

Did you predict this?

Standard Output:

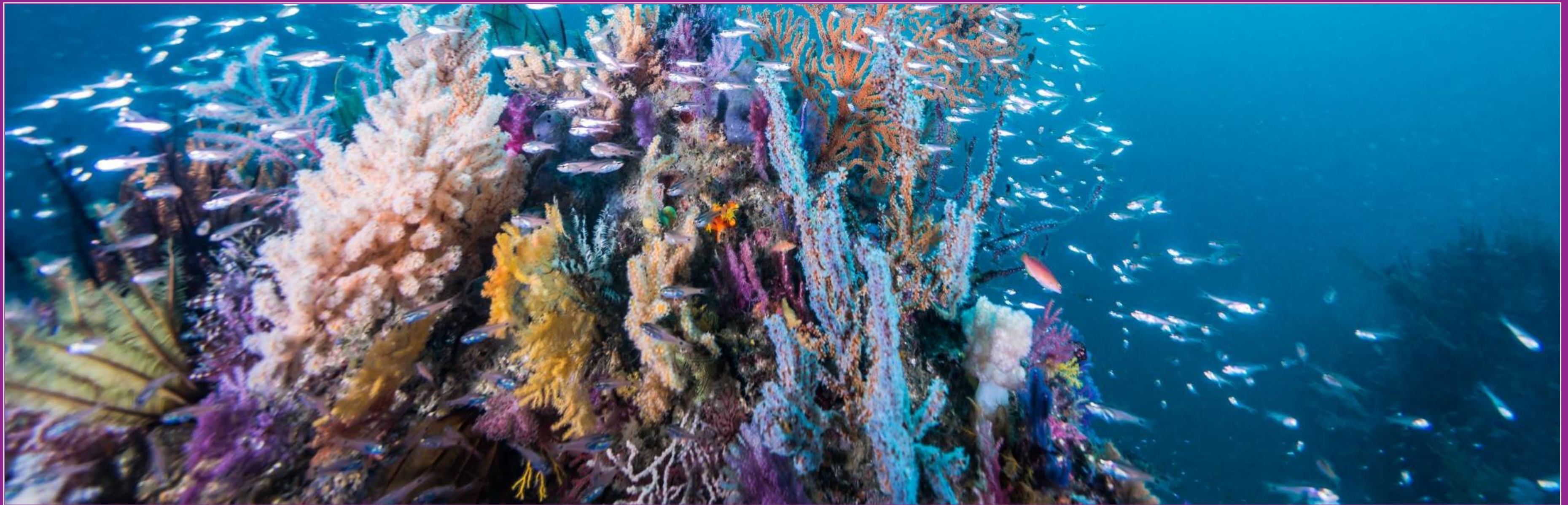
The third value is 512

The minimum value is 3

The third value is 17

The naughty GetMinValue() implementation

```
public static int GetMinValue(int[] array)
{
    Array.Sort(array);
    return array[0];
}
```

THAT WAS BAD ... DON'T DO THAT

Adding to an Array is hard

- You can't change the length of an array
- So adding items to it is hard
- Instead we can use an dynamic array data structure, or what C# calls a "List"

Lists are Generic Classes

[Test]

```
public static void ListDemo()
{
    var xs = new List<int>() { 1, 2, 3 };
    xs.Add(4);
    xs.Add(5);
    var content = string.Join(", ", xs);
    Console.WriteLine($"My list has {xs.Count} items: {content}");
    Console.WriteLine($"My list has the type {xs.GetType()}");
}
```


Output of List Demo

```
My list has 5 items: 1, 2, 3, 4, 5
```

```
My list has the type System.Collections.Generic.List`1[System.Int32]
```

List<T>

Items of the
same type.

```
xs = new  
List<Vector3>();
```

Add, remove,
and insert
items:

```
xs.Add(),  
xs.Remove(),  
xs.Insert()
```

Query the
count:

```
xs.Count
```

Retrieve specific
elements:

```
var v = xs[3];
```

Change specific
items:

```
xs[i] = new  
Vector3(0, 2, 0);
```

List<T> Example

```
> var xs = new List<int>();  
> xs.Add(1);  
> xs.Add(2);  
> xs.Add(4);  
> xs.Add(8);  
> xs.Count  
4  
> xs[0]  
1  
> xs[xs.Count - 1]  
8  
> xs[1] = 13  
13  
> xs  
List<int>(4) { 1, 13, 4, 8 }  
>
```

```
> xs = new List<int> { 5, 6, 7 }  
List<int>(3) { 5, 6, 7 }  
>
```

Indexing



Lists and Arrays use an integer index for each item



Indices start at 0



You can retrieve an item: **myList[index]**



You can modify an item at a position: **myList[index] = 12**



The last item in a list is: **myList[myList.Count - 1]**

What is the Runtime Type of a List

- A list is a generic type
- It is parameterized over the type of an element
- Arrays are too ... but for historical reasons are special

Arrays and Lists support IReadOnlyList

IReadOnlyList<T> Interface

Reference

 [Feedback](#)


Definition

Namespace: [System.Collections.Generic](#)

Assembly: [System.Runtime.dll](#)

Represents a read-only collection of elements that can be accessed by index.

C#

 Copy

```
public interface IReadOnlyList<out T> : System.Collections.Generic.IEnumerable<out T>,  
System.Collections.Generic.IReadOnlyCollection<out T>
```

ReadOnlyList<T> is very Useful

- Provides indexing (without changing items)
- Support a fast count retrieval
- Implements IEnumerable<T>

Arrays are Special

Single-dimensional arrays implement the `System.Collections.Generic.IList<T>`, `System.Collections.Generic ICollection<T>`, `System.Collections.Generic.IEnumerable<T>`, `System.Collections.Generic.IReadOnlyList<T>` and `System.Collections.Generic.IReadOnlyCollection<T>` generic interfaces. The implementations are provided to arrays at run time, and as a result, the generic interfaces do not appear in the declaration syntax for the `Array` class. In addition, there are no reference topics for interface members that are accessible only by casting an array to the generic interface type (explicit interface implementations). The key thing to be aware of when you cast an array to one of these interfaces is that members which add, insert, or remove elements throw `NotSupportedException`.

What if I don't need mutation?



Preventing changes should be your default



If you need efficient indexing and count: use **ReadOnlyList<T>**



Otherwise: use **IEnumerable<T>**

Don't Change a Collection During Foreach

Remarks

The returned `IEnumerator<T>` provides the ability to iterate through the collection by exposing a `Current` property. You can use enumerators to read the data in a collection, but not to modify the collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, `Current` is undefined. Therefore, you must call the `MoveNext` method to advance the enumerator to the first element of the collection before reading the value of `Current`.

`Current` returns the same object until `MoveNext` is called again as `MoveNext` sets `Current` to the next element.

If `MoveNext` passes the end of the collection, the enumerator is positioned after the last element in the collection and `MoveNext` returns `false`. When the enumerator is at this position, subsequent calls to `MoveNext` also return `false`. If the last call to `MoveNext` returned `false`, `Current` is undefined. You cannot set `Current` to the first element of the collection again; you must create a new enumerator instance instead.

If changes are made to the collection, such as adding, modifying, or deleting elements, the behavior of the enumerator is undefined.

<https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.ienumerable-1.getenumerator?view=net-6.0>

That was really BAD

- Arrays can be cast IList
- But it isn't in the documentation
- And not in the source code
- And most of the functions will throw errors

Or as Reddit user tweq says



tweq · 1 yr. ago

Technically speaking, arrays are magic and the compiler and runtime fudge the rules to make generics work on them.

But in general, [types can implement interface members without having public members of the same name.](#)

↑ 32 ↓ Reply Share Report Save Follow



bhjeff · 1 yr. ago

Technically speaking, arrays are magic

I love how a good explanation for it is "technically it's magic"

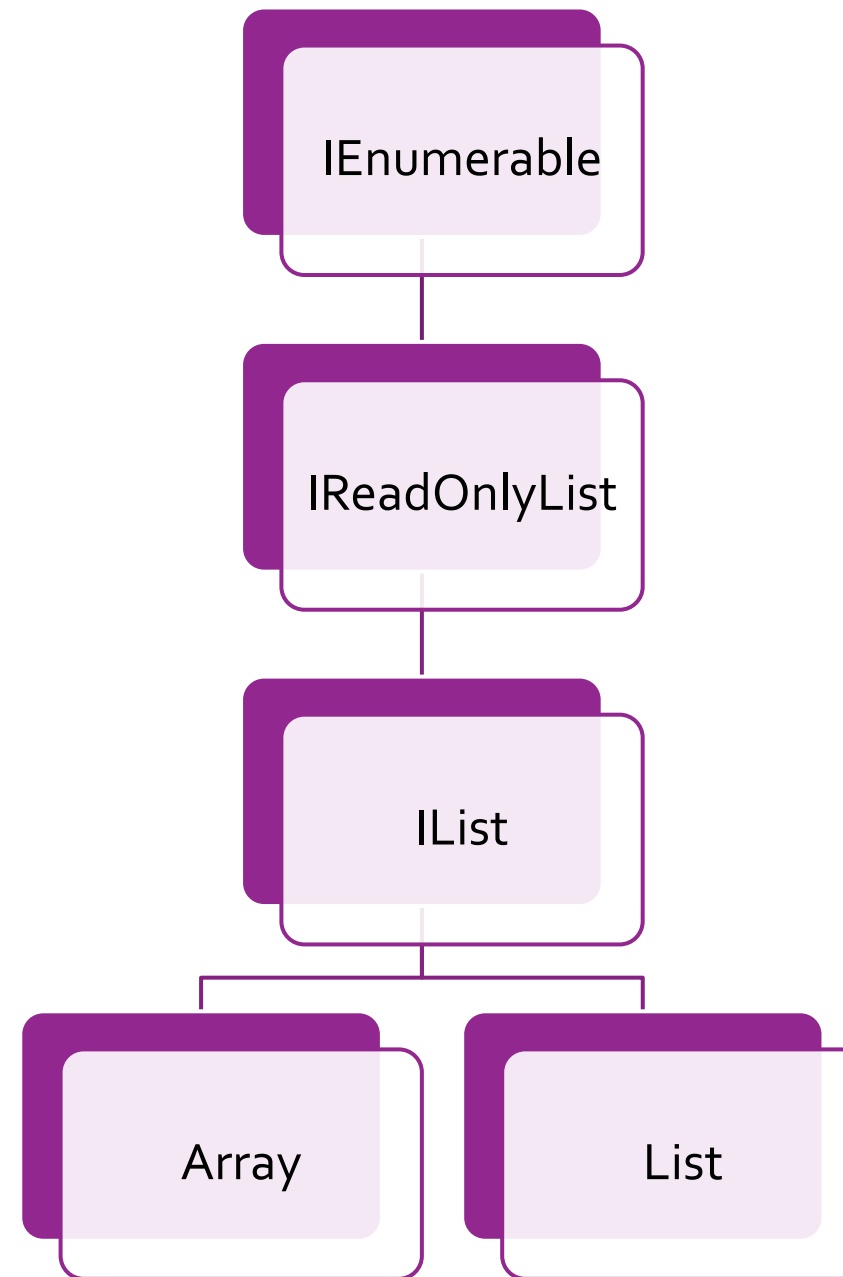
↑ 8 ↓ Reply Share Report Save Follow

https://www.reddit.com/r/csharp/comments/q2sgsa/how_can_string_implement_ireadonlycollection/

Good news

- Arrays and Lists can be cast to a happy list interface called “`ReadOnlyList<T>`”
- Use this collection when you want counts and indexing
- Prevent spooky actions at a distance!

Conceptually this is the model:



Interfaces and Upcasts

- Interfaces that a class implements are higher up on the hierarchy tree
- So casting from a class to an interface it implements is an upcast

When are interfaces useful?

- We use usually “var” for variable declarations
- As function parameters and return types.
- I want a function that enumerates a collection: use IEnumerable
- I want a function that uses the count or indexes a collection: use IReadOnlyList

Robustness Principle

- "be conservative in what you send, be liberal in what you accept".
- Allows you to write functions that are reusable in more scenarios

Using “Foreach”

- A foreach is a loop statement designed for IEnumerable<T>.
- Easier to type and read
- Visits each element in sequence

```
public void DrawList(Mesh mesh, Material material, List<Vector3> positions)
{
    foreach (var pos3D in positions)
    {
        DrawMeshAt(mesh, pos3D, material);
    }
}
```

IEnumerable<T>

An **IEnumerable<T>** is a generalization of collections

Enumerating means visiting each item in order one by one

To access an element, you must walk through items one by one

To get the count, you must visit each item until the end

Be careful: can sneakily increase computational complexity.

Basic IEnumerable<T> Operations (LINQ)



Select - Transform each item into a new one using a function



Where - Filter a collection by checking against a conditional function (predicate)



Aggregate - Combine elements using a function (generalization of sum)

Next Week

- More on LINQ