## **The Array Class**

The System.Array class serves as the implicit base class for all single-dimensional and multi-dimensional arrays in C#. It unifies the behavior of all arrays, providing a common set of methods regardless of their declared type or dimensions.

### **Core Concepts**

* **Implicit Subtyping:** When you declare an array using C#'s syntax (e.g., string[] names), the Common Language Runtime (CLR) implicitly creates a special "pseudotype" that is a subtype of Array. This pseudotype automatically implements generic collection interfaces like IList<T>.
* **Contiguous Memory:** Arrays are allocated as a contiguous block of memory. This design makes indexing into arrays extremely efficient (constant time, O(1)).
* **Fixed Size:** A direct consequence of contiguous memory allocation is that arrays have a fixed size after creation and cannot be resized. While Array.Resize exists, it works by creating a *new* array and copying elements, which is inefficient and doesn't update existing references to the old array. For resizable collections, List<T> is the preferred choice.
* **Element Storage:**
  + **Value-type elements:** Are stored directly within the array's memory block. An array of long (8 bytes each) will occupy 8 \* Length bytes.
  + **Reference-type elements:** The array stores *references* (pointers) to the actual objects, not the objects themselves. Each reference takes up 4 bytes on a 32-bit system or 8 bytes on a 64-bit system. The objects themselves reside elsewhere on the heap.

| StringBuilder[] builders = new StringBuilder[5]; // Array of 5 StringBuilder references builders[0] = new StringBuilder("builder1"); // Object is on heap, array holds reference long[] numbers = new long[3]; // Array of 3 long values (stored directly) numbers[0] = 12345; |
| --- |

* **Arrays are Reference Types:** Despite their element type, arrays *themselves* are always reference types. This means:
  + arrayB = arrayA; makes both arrayB and arrayA reference the *same* array in memory.
  + a1 == a2 will always be false for two distinct array instances, even if their contents are identical, because == for reference types (by default) checks for referential equality.
  + To compare array *contents*, you need a structural equality comparer, such as IStructuralEquatable.

| object[] a1 = { "string", 123, true }; object[] a2 = { "string", 123, true }; Console.WriteLine(a1 == a2); // False (different array instances) Console.WriteLine(a1.Equals(a2)); // False (object.Equals checks referential equality by default)  IStructuralEquatable se1 = a1; // True, using a comparer that checks element-by-element equality Console.WriteLine(se1.Equals(a2, StructuralComparisons.StructuralEqualityComparer)); |
| --- |

* **Cloning Arrays:**
  + Clone(): Creates a **shallow copy** of the array. For value-type elements, the values are copied. For reference-type elements, *only the references are copied*, meaning both the original and cloned arrays will point to the same underlying objects.
  + **Deep Copy:** To create a deep copy (where all nested reference types are also duplicated), you must manually iterate through the array and clone each element.

### **Construction and Indexing**

While C# syntax is the easiest way to create and index arrays:

| int[] myArray = { 1, 2, 3 }; int first = myArray[0]; |
| --- |

You can also create arrays dynamically using Array.CreateInstance. This allows you to specify the element type, number of dimensions (rank), and even a non-zero lower bound at runtime. (However, non-zero-based arrays are generally discouraged as they are not CLS-compliant and can cause interoperability issues with other .NET languages.)

* **GetValue() and SetValue():** These methods allow dynamic access to array elements, especially useful for arrays created with CreateInstance or when dealing with arrays of unknown type or rank. They accept params int[] indices for multi-dimensional arrays.

| Array a = Array.CreateInstance(typeof(string), 2); a.SetValue("hi", 0); string s = (string)a.GetValue(0);  // Casting to a C# array is possible for zero-indexed, compatible arrays string[] cSharpArray = (string[])a; |
| --- |

* SetValue throws an exception if the element type is incompatible.
* When an array is instantiated, its elements are automatically initialized to their default values (null for reference types, bitwise zero for value types). Array.Clear() can also reset a portion of an array to default values.

### **Enumeration**

Arrays inherently support enumeration via the foreach statement:

| int[] myArray = { 1, 2, 3 }; foreach (int val in myArray)  Console.WriteLine(val); |
| --- |

Additionally, Array.ForEach() provides a static helper for iterating and performing an action on each element using an Action<T> delegate or lambda expression:

| Array.ForEach(new[] { 1, 2, 3 }, Console.WriteLine); // Or Array.ForEach([1, 2, 3], Console.WriteLine); with C# 12+ collection expressions |
| --- |

### 

### **Length and Rank**

The Array class provides properties and methods to query an array's dimensions and total elements:

* Length: Total number of elements in all dimensions.
* LongLength: Total number of elements as a long (for very large arrays, though limited by CLR's 2GB object size).
* Rank: Number of dimensions (e.g., 1 for int[], 2 for int[,]).
* GetLength(int dimension): Length of a specific dimension.
* GetLowerBound(int dimension): Lower bound (usually 0).
* GetUpperBound(int dimension): Upper bound.

### **Searching**

The Array class provides static methods for searching one-dimensional arrays:

* **BinarySearch**: For **sorted** arrays. Requires elements to be comparable (via IComparable<T> or a custom IComparer<T>). Very fast (logarithmic time). Returns the index of the item or ~index if not found (where index is the next larger element's position).
* **IndexOf / LastIndexOf**: For **unsorted** arrays. Performs a linear scan, returning the index of the first (or last) matching element, or -1 if not found.
* **Predicate-based methods (e.g., Find, FindAll, Exists, TrueForAll)**: Search unsorted arrays based on a Predicate<T> delegate (a function that takes an element and returns true or false).

| string[] names = { "Rodney", "Jack", "Jill" }; // Find the first name containing 'a' string match = Array.Find(names, n => n.Contains("a")); // Using lambda expression Console.WriteLine(match); // Jack |
| --- |

* + FindAll: Returns a new array of all matching items. (Similar to LINQ's Where but returns an array).
  + Exists: Returns true if *any* element satisfies the predicate. (Similar to LINQ's Any).
  + TrueForAll: Returns true if *all* elements satisfy the predicate. (Similar to LINQ's All).

### **Sorting**

Array.Sort() provides static methods to sort one-dimensional arrays.

* **Single Array Sorting:** Array.Sort(T[] array) sorts the array in place.

| int[] numbers = { 3, 2, 1 }; Array.Sort(numbers); // numbers is now { 1, 2, 3 } |
| --- |

* **Paired Array Sorting:** Array.Sort(TKey[] keys, TValue[] items) sorts two arrays in tandem, basing the order on the keys array.

| int[] numbers = { 3, 2, 1 }; string[] words = { "three", "two", "one" }; Array.Sort(numbers, words); // numbers: { 1, 2, 3 }; words: { "one", "two", "three" } |
| --- |

* **Custom Sorting:** Sort requires elements to be IComparable<T> by default. You can provide custom sorting logic using:
  + An IComparer<T> object.
  + A Comparison<T> delegate (often a lambda expression):

| int[] numbers = { 1, 2, 3, 4, 5 }; // Sort odd numbers first Array.Sort(numbers, (x, y) => x % 2 == y % 2 ? 0 : x % 2 == 1 ? -1 : 1); // numbers is now { 1, 3, 5, 2, 4 } |
| --- |

### **Reversing Elements**

Array.Reverse() static methods can reverse the order of all or a portion of elements in an array.

| int[] data = { 1, 2, 3 }; Array.Reverse(data); // data is now { 3, 2, 1 } |
| --- |

### 

### **Copying**

The Array class offers methods for shallow copying:

* Clone(): Returns a new shallow-copied array.
* CopyTo() (instance method) / Copy() (static method): Copies a contiguous subset of elements from one array to another. Source and destination ranges can overlap.
* ConstrainedCopy(): Similar to Copy, but performs an atomic operation. If any element cannot be successfully copied (e.g., due to type incompatibility), the entire operation is rolled back.
* AsReadOnly(): Returns a wrapper that prevents modification of array elements.

### 

### **Converting and Resizing**

* Array.ConvertAll<TInput, TOutput>(): Creates a new array of a different element type by applying a Converter<TInput, TOutput> delegate (often a lambda) to each element.

| float[] reals = { 1.3f, 1.5f, 1.8f }; int[] wholes = Array.ConvertAll(reals, r => Convert.ToInt32(r)); // wholes is { 1, 2, 2 } |
| --- |

* Array.Resize<T>(): As mentioned, this method creates a *new* array of the specified size, copies elements from the original, and updates the *provided reference variable* to point to the new array. Be mindful that other references to the *original* array will still point to the old (now potentially smaller) array.