Structure

Array

An ordered, random-access collection.

iOS 8.0+ | iPadOS 8.0+ | Mac Catalyst 13.0+ | macOS 10.10+ | tvOS 9.0+ | visionOS 1.0+ | watchOS 2.0+

```
@frozen
struct Array<Element>
```

Overview

Arrays are one of the most commonly used data types in an app. You use arrays to organize your app's data. Specifically, you use the `Array` type to hold elements of a single type, the array's `Element` type. An array can store any kind of elements—from integers to strings to classes.

Swift makes it easy to create arrays in your code using an array literal: simply surround a comma-separated list of values with square brackets. Without any other information, Swift creates an array that includes the specified values, automatically inferring the array's `Element` type. For example:

```
// An array of 'Int' elements
let oddNumbers = [1, 3, 5, 7, 9, 11, 13, 15]

// An array of 'String' elements
let streets = ["Albemarle", "Brandywine", "Chesapeake"]
```

You can create an empty array by specifying the `Element` type of your array in the declaration. For example:

```
// Shortened forms are preferred
var emptyDoubles: [Double] = []
```

```
// The full type name is also allowed
var emptyFloats: Array<Float> = Array()
```

If you need an array that is preinitialized with a fixed number of default values, use the `Array(repeating:count:)` initializer.

```
var digitCounts = Array(repeating: 0, count: 10)
print(digitCounts)
// Prints "[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]"
```

Accessing Array Values

When you need to perform an operation on all of an array's elements, use a `for-in` loop to iterate through the array's contents.

```
for street in streets {
    print("I don't live on \(street).")
}
// Prints "I don't live on Albemarle."
// Prints "I don't live on Brandywine."
// Prints "I don't live on Chesapeake."
```

Use the `isEmpty` property to check quickly whether an array has any elements, or use the `count` property to find the number of elements in the array.

```
if oddNumbers.isEmpty {
    print("I don't know any odd numbers.")
} else {
    print("I know \(oddNumbers.count) odd numbers.")
}
// Prints "I know 8 odd numbers."
```

Use the `first` and `last` properties for safe access to the value of the array's first and last elements. If the array is empty, these properties are `nil`.

```
if let firstElement = oddNumbers.first, let lastElement = oddNumbers.last {
    print(firstElement, lastElement, separator: ", ")
}
```

```
// Prints "1, 15"

print(emptyDoubles.first, emptyDoubles.last, separator: ", ")
// Prints "nil, nil"
```

You can access individual array elements through a subscript. The first element of a nonempty array is always at index zero. You can subscript an array with any integer from zero up to, but not including, the count of the array. Using a negative number or an index equal to or greater than `count` triggers a runtime error. For example:

```
print(oddNumbers[0], oddNumbers[3], separator: ", ")
// Prints "1, 7"

print(emptyDoubles[0])
// Triggers runtime error: Index out of range
```

Adding and Removing Elements

Suppose you need to store a list of the names of students that are signed up for a class you're teaching. During the registration period, you need to add and remove names as students add and drop the class.

```
var students = ["Ben", "Ivy", "Jordell"]
```

To add single elements to the end of an array, use the `append(_:)` method. Add multiple elements at the same time by passing another array or a sequence of any kind to the `append(contentsOf:)` method.

```
students.append("Maxime")
students.append(contentsOf: ["Shakia", "William"])
// ["Ben", "Ivy", "Jordell", "Maxime", "Shakia", "William"]
```

You can add new elements in the middle of an array by using the `insert(_:at:)` method for single elements and by using `insert(contentsOf:at:)` to insert multiple elements from another collection or array literal. The elements at that index and later indices are shifted back to make room.

```
students.insert("Liam", at: 3)
// ["Ben", "Ivy", "Jordell", "Liam", "Maxime", "Shakia", "William"]
```

To remove elements from an array, use the `remove(at:)`, `removeSubrange(_:)`, and `removeLast()` methods.

```
// Ben's family is moving to another state
students.remove(at: 0)
// ["Ivy", "Jordell", "Liam", "Maxime", "Shakia", "William"]

// William is signing up for a different class
students.removeLast()
// ["Ivy", "Jordell", "Liam", "Maxime", "Shakia"]
```

You can replace an existing element with a new value by assigning the new value to the subscript.

```
if let i = students.firstIndex(of: "Maxime") {
    students[i] = "Max"
}
// ["Ivy", "Jordell", "Liam", "Max", "Shakia"]
```

Growing the Size of an Array

Every array reserves a specific amount of memory to hold its contents. When you add elements to an array and that array begins to exceed its reserved capacity, the array allocates a larger region of memory and copies its elements into the new storage. The new storage is a multiple of the old storage's size. This exponential growth strategy means that appending an element happens in constant time, averaging the performance of many append operations. Append operations that trigger reallocation have a performance cost, but they occur less and less often as the array grows larger.

If you know approximately how many elements you will need to store, use the `reserveCapacity(_:)` method before appending to the array to avoid intermediate reallocations. Use the `capacity` and `count` properties to determine how many more elements the array can store without allocating larger storage.

For arrays of most `Element` types, this storage is a contiguous block of memory. For arrays with an `Element` type that is a class or `@objc` protocol type, this storage can be a contiguous block of memory or an instance of `NSArray`. Because any arbitrary subclass of `NSArray` can become an `Array`, there are no guarantees about representation or efficiency in this case.

Modifying Copies of Arrays

Each array has an independent value that includes the values of all of its elements. For simple types such as integers and other structures, this means that when you change a value in one array, the value of that element does not change in any copies of the array. For example:

```
var numbers = [1, 2, 3, 4, 5]
var numbersCopy = numbers
numbers[0] = 100
print(numbers)
// Prints "[100, 2, 3, 4, 5]"
print(numbersCopy)
// Prints "[1, 2, 3, 4, 5]"
```

If the elements in an array are instances of a class, the semantics are the same, though they might appear different at first. In this case, the values stored in the array are references to objects that live outside the array. If you change a reference to an object in one array, only that array has a reference to the new object. However, if two arrays contain references to the same object, you can observe changes to that object's properties from both arrays. For example:

```
// An integer type with reference semantics
class IntegerReference {
    var value = 10
}
var firstIntegers = [IntegerReference(), IntegerReference()]
var secondIntegers = firstIntegers

// Modifications to an instance are visible from either array
firstIntegers[0].value = 100
print(secondIntegers[0].value)
// Prints "100"

// Replacements, additions, and removals are still visible
// only in the modified array
firstIntegers[0] = IntegerReference()
print(firstIntegers[0].value)
// Prints "10"
print(secondIntegers[0].value)
// Prints "100"
```

Arrays, like all variable-size collections in the standard library, use copy-on-write optimization. Multiple copies of an array share the same storage until you modify one of the copies. When that happens, the array being modified replaces its storage with a uniquely owned copy of itself, which is then modified in place. Optimizations are sometimes applied that can reduce the amount of copying.

This means that if an array is sharing storage with other copies, the first mutating operation on that array incurs the cost of copying the array. An array that is the sole owner of its storage can perform mutating operations in place.

In the example below, a `numbers` array is created along with two copies that share the same storage. When the original `numbers` array is modified, it makes a unique copy of its storage before making the modification. Further modifications to `numbers` are made in place, while the two copies continue to share the original storage.

```
var numbers = [1, 2, 3, 4, 5]
var firstCopy = numbers
var secondCopy = numbers

// The storage for 'numbers' is copied here
numbers[0] = 100
numbers[1] = 200
numbers[2] = 300
// 'numbers' is [100, 200, 300, 4, 5]
// 'firstCopy' and 'secondCopy' are [1, 2, 3, 4, 5]
```

Bridging Between Array and NSArray

When you need to access APIs that require data in an `NSArray` instance instead of `Array`, use the type-cast operator (`as`) to bridge your instance. For bridging to be possible, the `Element` type of your array must be a class, an `@objc` protocol (a protocol imported from Objective-C or marked with the `@objc` attribute), or a type that bridges to a Foundation type.

The following example shows how you can bridge an `Array` instance to `NSArray` to use the `write(to:atomically:)` method. In this example, the `colors` array can be bridged to `NSArray` because the `colors` array's `String` elements bridge to `NSString`. The compiler prevents bridging the `moreColors` array, on the other hand, because its `Element` type is `Optional<String>`, which does *not* bridge to a Foundation type.

```
let colors = ["periwinkle", "rose", "moss"]
let moreColors: [String?] = ["ochre", "pine"]
```

```
let url = URL(fileURLWithPath: "names.plist")
(colors as NSArray).write(to: url, atomically: true)
// true

(moreColors as NSArray).write(to: url, atomically: true)
// error: cannot convert value of type '[String?]' to type 'NSArray'
```

Bridging from `Array` to `NSArray` takes O(1) time and O(1) space if the array's elements are already instances of a class or an `@objc` protocol; otherwise, it takes O(*n*) time and space.

When the destination array's element type is a class or an `@objc` protocol, bridging from `NSArray` to `Array` first calls the `copy(with:)` (`- copyWithZone:` in Objective-C) method on the array to get an immutable copy and then performs additional Swift bookkeeping work that takes O(1) time. For instances of `NSArray` that are already immutable, `copy(with:)` usually returns the same array in O(1) time; otherwise, the copying performance is unspecified. If `copy(with:)` returns the same array, the instances of `NSArray` and `Array` share storage using the same copy-on-write optimization that is used when two instances of `Array` share storage.

When the destination array's element type is a nonclass type that bridges to a Foundation type, bridging from `NSArray` to `Array` performs a bridging copy of the elements to contiguous storage in O(*n*) time. For example, bridging from `NSArray` to `Array<Int>` performs such a copy. No further bridging is required when accessing elements of the `Array` instance.

> **Note**
>
> The `ContiguousArray` and `ArraySlice` types are not bridged; instances of those types always have a contiguous block of memory as their storage.

Topics

Creating an Array

In addition to using an array literal, you can also create an array using these initializers.

`init()`

Creates a new, empty array.

`init<S>(S)`

Creates a new instance of a collection containing the elements of a sequence.

`init<S>(S)`

Creates an array containing the elements of a sequence.

`init(repeating: Element, count: Int)`

Creates a new array containing the specified number of a single, repeated value.

`init(unsafeUninitializedCapacity: Int, initializingWith: (inout UnsafeMutableBufferPointer<Element>, inout Int) throws -> Void) rethrows`

Creates an array with the specified capacity, then calls the given closure with a buffer covering the array's uninitialized memory.

Inspecting an Array

`var isEmpty: Bool`

A Boolean value indicating whether the collection is empty.

`var count: Int`

The number of elements in the array.

`var capacity: Int`

The total number of elements that the array can contain without allocating new storage.

Accessing Elements

`subscript(Int) -> Element`

Accesses the element at the specified position.

`var first: Self.Element?`

The first element of the collection.

`var last: Self.Element?`

The last element of the collection.

`subscript(Range<Int>) -> ArraySlice<Element>`

Accesses a contiguous subrange of the array's elements.

`subscript<R>(R) -> Self.SubSequence`

`subscript<R>(R) -> Self.SubSequence`

Accesses the contiguous subrange of the collection's elements specified by a range expression.

`subscript((UnboundedRange_) -> ()) -> Self.SubSequence`

`func randomElement() -> Self.Element?`

Returns a random element of the collection.

`func randomElement<T>(using: inout T) -> Self.Element?`

Returns a random element of the collection, using the given generator as a source for randomness.

Adding Elements

`func append(Element)`

Adds a new element at the end of the array.

`func insert(Element, at: Int)`

Inserts a new element at the specified position.

`func insert<C>(contentsOf: C, at: Self.Index)`

Inserts the elements of a sequence into the collection at the specified position.

`func replaceSubrange<C>(Range<Int>, with: C)`

Replaces a range of elements with the elements in the specified collection.

`func replaceSubrange<C, R>(R, with: C)`

Replaces the specified subrange of elements with the given collection.

`func reserveCapacity(Int)`

Reserves enough space to store the specified number of elements.

Combining Arrays

`func append<S>(contentsOf: S)`

Adds the elements of a sequence to the end of the array.

`func append<S>(contentsOf: S)`

Adds the elements of a sequence or collection to the end of this collection.

`static func + <Other>(Other, Self) -> Self`

Creates a new collection by concatenating the elements of a sequence and a collection.

`static func + <Other>(Self, Other) -> Self`

Creates a new collection by concatenating the elements of a collection and a sequence.

```
static func + (Array<Element>, Array<Element>) -> Array<Element>
```

```
static func + <Other>(Self, Other) -> Self
```

Creates a new collection by concatenating the elements of two collections.

```
static func += <Other>(inout Self, Other)
```

Appends the elements of a sequence to a range-replaceable collection.

```
static func += (inout Array<Element>, Array<Element>)
```

Removing Elements

```
func remove(at: Int) -> Element
```

Removes and returns the element at the specified position.

```
func removeFirst() -> Self.Element
```

Removes and returns the first element of the collection.

```
func removeFirst(Int)
```

Removes the specified number of elements from the beginning of the collection.

```
func removeLast() -> Self.Element
```

Removes and returns the last element of the collection.

```
func removeLast(Int)
```

Removes the specified number of elements from the end of the collection.

```
func removeSubrange(Range<Self.Index>)
```

Removes the elements in the specified subrange from the collection.

```
func removeSubrange<R>(R)
```

Removes the elements in the specified subrange from the collection.

```
func removeAll(where: (Self.Element) throws -> Bool) rethrows
```

Removes all the elements that satisfy the given predicate.

```
func removeAll(keepingCapacity: Bool)
```

Removes all elements from the array.

```
func popLast() -> Self.Element?
```

Removes and returns the last element of the collection.

Finding Elements

`func contains(Self.Element) -> Bool`

Returns a Boolean value indicating whether the sequence contains the given element.

`func contains(where: (Self.Element) throws -> Bool) rethrows -> Bool`

Returns a Boolean value indicating whether the sequence contains an element that satisfies the given predicate.

`func allSatisfy((Self.Element) throws -> Bool) rethrows -> Bool`

Returns a Boolean value indicating whether every element of a sequence satisfies a given predicate.

`func first(where: (Self.Element) throws -> Bool) rethrows -> Self.Element?`

Returns the first element of the sequence that satisfies the given predicate.

`func firstIndex(of: Self.Element) -> Self.Index?`

Returns the first index where the specified value appears in the collection.

`func index(of: Self.Element) -> Self.Index?`

Returns the first index where the specified value appears in the collection.

`func firstIndex(where: (Self.Element) throws -> Bool) rethrows -> Self.Index?`

Returns the first index in which an element of the collection satisfies the given predicate.

`func last(where: (Self.Element) throws -> Bool) rethrows -> Self.Element?`

Returns the last element of the sequence that satisfies the given predicate.

`func lastIndex(of: Self.Element) -> Self.Index?`

Returns the last index where the specified value appears in the collection.

`func lastIndex(where: (Self.Element) throws -> Bool) rethrows -> Self.Index?`

Returns the index of the last element in the collection that matches the given predicate.

`func min() -> Self.Element?`

Returns the minimum element in the sequence.

`func min(by: (Self.Element, Self.Element) throws -> Bool) rethrows -> Self.Element?`

Returns the minimum element in the sequence, using the given predicate as the comparison between elements.

`func max() -> Self.Element?`

Returns the maximum element in the sequence.

`func max(by: (Self.Element, Self.Element) throws -> Bool) rethrows -> Self.Element?`

Returns the maximum element in the sequence, using the given predicate as the comparison between elements.

Selecting Elements

`func prefix(Int) -> Self.SubSequence`

Returns a subsequence, up to the specified maximum length, containing the initial elements of the collection.

`func prefix(through: Self.Index) -> Self.SubSequence`

Returns a subsequence from the start of the collection through the specified position.

`func prefix(upTo: Self.Index) -> Self.SubSequence`

Returns a subsequence from the start of the collection up to, but not including, the specified position.

`func prefix(while: (Self.Element) throws -> Bool) rethrows -> Self.SubSequence`

Returns a subsequence containing the initial elements until `predicate` returns `false` and skipping the remaining elements.

`func suffix(Int) -> Self.SubSequence`

Returns a subsequence, up to the given maximum length, containing the final elements of the collection.

`func suffix(from: Self.Index) -> Self.SubSequence`

Returns a subsequence from the specified position to the end of the collection.

Excluding Elements

`func dropFirst(Int) -> Self.SubSequence`

Returns a subsequence containing all but the given number of initial elements.

```
func dropLast(Int) -> Self.SubSequence
```

Returns a subsequence containing all but the specified number of final elements.

```
func drop(while: (Self.Element) throws -> Bool) rethrows -> Self.Sub
Sequence
```

Returns a subsequence by skipping elements while `predicate` returns `true` and returning the remaining elements.

Transforming an Array

```
func flatMap<SegmentOfResult>((Self.Element) throws -> SegmentOfResult)
rethrows -> [SegmentOfResult.Element]
```

Returns an array containing the concatenated results of calling the given transformation with each element of this sequence.

```
func flatMap<ElementOfResult>((Self.Element) throws -> ElementOfResult
?) rethrows -> [ElementOfResult]
```

```
func compactMap<ElementOfResult>((Self.Element) throws -> ElementOf
Result?) rethrows -> [ElementOfResult]
```

Returns an array containing the non-`nil` results of calling the given transformation with each element of this sequence.

```
func reduce<Result>(Result, (Result, Self.Element) throws -> Result)
rethrows -> Result
```

Returns the result of combining the elements of the sequence using the given closure.

```
func reduce<Result>(into: Result, (inout Result, Self.Element) throws -
> ()) rethrows -> Result
```

Returns the result of combining the elements of the sequence using the given closure.

```
var lazy: LazySequence<Self>
```

A sequence containing the same elements as this sequence, but on which some operations, such as `map` and `filter`, are implemented lazily.

Iterating Over an Array's Elements

```
func forEach((Self.Element) throws -> Void) rethrows
```

Calls the given closure on each element in the sequence in the same order as a `for-in` loop.

```
func enumerated() -> EnumeratedSequence<Self>
```

Returns a sequence of pairs (*n*, *x*), where *n* represents a consecutive integer starting at zero and *x* represents an element of the sequence.

```
func makeIterator() -> IndexingIterator<Self>
```

Returns an iterator over the elements of the collection.

```
var underestimatedCount: Int
```

A value less than or equal to the number of elements in the collection.

Reordering an Array's Elements

```
func sort()
```

Sorts the collection in place.

```
func sort(by: (Self.Element, Self.Element) throws -> Bool) rethrows
```

Sorts the collection in place, using the given predicate as the comparison between elements.

```
func sorted() -> [Self.Element]
```

Returns the elements of the sequence, sorted.

```
func sorted(by: (Self.Element, Self.Element) throws -> Bool) rethrows -> [Self.Element]
```

Returns the elements of the sequence, sorted using the given predicate as the comparison between elements.

```
func reverse()
```

Reverses the elements of the collection in place.

```
func reversed() -> ReversedCollection<Self>
```

Returns a view presenting the elements of the collection in reverse order.

```
func shuffle()
```

Shuffles the collection in place.

```
func shuffle<T>(using: inout T)
```

Shuffles the collection in place, using the given generator as a source for randomness.

```
func shuffled() -> [Self.Element]
```

Returns the elements of the sequence, shuffled.

```
func shuffled<T>(using: inout T) -> [Self.Element]
```

Returns the elements of the sequence, shuffled using the given generator as a source for randomness.

```
func partition(by: (Self.Element) throws -> Bool) rethrows -> Self.Index
```

Reorders the elements of the collection such that all the elements that match the given predicate are after all the elements that don't match.

```
func swapAt(Self.Index, Self.Index)
```

Exchanges the values at the specified indices of the collection.

Splitting and Joining Elements

```
func split(separator: Self.Element, maxSplits: Int, omittingEmptySubsequences: Bool) -> [Self.SubSequence]
```

Returns the longest possible subsequences of the collection, in order, around elements equal to the given element.

```
func split(maxSplits: Int, omittingEmptySubsequences: Bool, whereSeparator: (Self.Element) throws -> Bool) rethrows -> [Self.SubSequence]
```

Returns the longest possible subsequences of the collection, in order, that don't contain elements satisfying the given predicate.

```
func joined() -> FlattenSequence<Self>
```

Returns the elements of this sequence of sequences, concatenated.

```
func joined<Separator>(separator: Separator) -> JoinedSequence<Self>
```

Returns the concatenated elements of this sequence of sequences, inserting the given separator between each element.

```
func joined(separator: String) -> String
```

Returns a new string by concatenating the elements of the sequence, adding the given separator between each element.

```
func joined(separator: String) -> String
```

Returns a new string by concatenating the elements of the sequence, adding the given separator between each element.

Creating and Applying Differences

`func applying(CollectionDifference<Self.Element>) -> Self?`

Applies the given difference to this collection.

`func difference<C>(from: C) -> CollectionDifference<Self.Element>`

Returns the difference needed to produce this collection's ordered elements from the given collection.

`func difference<C>(from: C, by: (C.Element, Self.Element) -> Bool) -> CollectionDifference<Self.Element>`

Returns the difference needed to produce this collection's ordered elements from the given collection, using the given predicate as an equivalence test.

Comparing Arrays

`static func == (Array<Element>, Array<Element>) -> Bool`

Returns a Boolean value indicating whether two arrays contain the same elements in the same order.

`static func != (Self, Self) -> Bool`

Returns a Boolean value indicating whether two values are not equal.

`func elementsEqual<OtherSequence>(OtherSequence) -> Bool`

Returns a Boolean value indicating whether this sequence and another sequence contain the same elements in the same order.

`func elementsEqual<OtherSequence>(OtherSequence, by: (Self.Element, OtherSequence.Element) throws -> Bool) rethrows -> Bool`

Returns a Boolean value indicating whether this sequence and another sequence contain equivalent elements in the same order, using the given predicate as the equivalence test.

`func starts<PossiblePrefix>(with: PossiblePrefix) -> Bool`

Returns a Boolean value indicating whether the initial elements of the sequence are the same as the elements in another sequence.

`func starts<PossiblePrefix>(with: PossiblePrefix, by: (Self.Element, PossiblePrefix.Element) throws -> Bool) rethrows -> Bool`

Returns a Boolean value indicating whether the initial elements of the sequence are equivalent to the elements in another sequence, using the given predicate as the equivalence test.

`func lexicographicallyPrecedes<OtherSequence>(OtherSequence) -> Bool`

Returns a Boolean value indicating whether the sequence precedes another sequence in a lexicographical (dictionary) ordering, using the less-than operator (<) to compare elements.

```
func lexicographicallyPrecedes<OtherSequence>(OtherSequence, by: (Self.Element, Self.Element) throws -> Bool) rethrows -> Bool
```

Returns a Boolean value indicating whether the sequence precedes another sequence in a lexicographical (dictionary) ordering, using the given predicate to compare elements.

Manipulating Indices

```
var startIndex: Int
```

The position of the first element in a nonempty array.

```
var endIndex: Int
```

The array's "past the end" position—that is, the position one greater than the last valid subscript argument.

```
func index(after: Int) -> Int
```

Returns the position immediately after the given index.

```
func formIndex(after: inout Int)
```

Replaces the given index with its successor.

```
func index(before: Int) -> Int
```

Returns the position immediately before the given index.

```
func formIndex(before: inout Int)
```

Replaces the given index with its predecessor.

```
func index(Int, offsetBy: Int) -> Int
```

Returns an index that is the specified distance from the given index.

```
func formIndex(inout Self.Index, offsetBy: Int)
```

Offsets the given index by the specified distance.

```
func index(Int, offsetBy: Int, limitedBy: Int) -> Int?
```

Returns an index that is the specified distance from the given index, unless that distance is beyond a given limiting index.

```
func formIndex(inout Self.Index, offsetBy: Int, limitedBy: Self.Index) -> Bool
```

Offsets the given index by the specified distance, or so that it equals the given limiting index.

`func distance(from: Int, to: Int) -> Int`

Returns the distance between two indices.

Accessing Underlying Storage

`func withUnsafeBufferPointer<R, E>((UnsafeBufferPointer<Element>) throws(E) -> R) throws(E) -> R`

Calls a closure with a pointer to the array's contiguous storage.

`func withUnsafeMutableBufferPointer<R, E>((inout UnsafeMutableBufferPointer<Element>) throws(E) -> R) throws(E) -> R`

Calls the given closure with a pointer to the array's mutable contiguous storage.

`func withUnsafeBytes<R>((UnsafeRawBufferPointer) throws -> R) rethrows -> R`

Calls the given closure with a pointer to the underlying bytes of the array's contiguous storage.

`func withUnsafeMutableBytes<R>((UnsafeMutableRawBufferPointer) throws -> R) rethrows -> R`

Calls the given closure with a pointer to the underlying bytes of the array's mutable contiguous storage.

`func withContiguousStorageIfAvailable<R>((UnsafeBufferPointer<Element>) throws -> R) rethrows -> R?`

Executes a closure on the sequence's contiguous storage.

`func withContiguousMutableStorageIfAvailable<R>((inout UnsafeMutableBufferPointer<Element>) throws -> R) rethrows -> R?`

Executes a closure on the collection's contiguous storage.

Encoding and Decoding

`func encode(to: any Encoder) throws`

Encodes the elements of this array into the given encoder in an unkeyed container.

`init(from: any Decoder) throws`

Creates a new array by decoding from the given decoder.

Describing an Array

`var description: String`

A textual representation of the array and its elements.

`var debugDescription: String`

A textual representation of the array and its elements, suitable for debugging.

`var customMirror: Mirror`

A mirror that reflects the array.

`func hash(into: inout Hasher)`

Hashes the essential components of this value by feeding them into the given hasher.

Converting Between Arrays and Create ML Types

`init(MLDataColumn<Element>)`

Constructs an Array with the elements of a DataColumn.

`init(MLUntypedColumn)`

Constructs an Array with the elements of an MLUntypedColumn.

Related Array Types

`struct ContiguousArray`

A contiguously stored array.

`struct ArraySlice`

A slice of an `Array`, `ContiguousArray`, or `ArraySlice` instance.

Reference Types

Use bridged reference types when you need reference semantics or Foundation-specific behavior.

`class NSArray`

A static ordered collection of objects.

`class NSMutableArray`

A dynamic ordered collection of objects.

Supporting Types

`typealias Index`

The index type for arrays, `Int`.

`typealias Indices`

The type that represents the indices that are valid for subscripting an array, in ascending order.

`typealias Iterator`

The type that allows iteration over an array's elements.

`typealias ArrayLiteralElement`

The type of the elements of an array literal.

`typealias SubSequence`

A collection representing a contiguous subrange of this collection's elements. The subsequence shares indices with the original collection.

Infrequently Used Functionality

`init(arrayLiteral: Element...)`

Creates an array from the given array literal.

`var hashValue: Int`

The hash value.

Initializers

`init(fromSplitComplex: DSPSplitComplex, scale: Float, count: Int)`

Creates a new array of single-precision values from a `DSPSplitComplex` structure.

`init(fromSplitComplex: DSPDoubleSplitComplex, scale: Double, count: Int)`

Creates a new array of single-precision values from a `DSPDoubleSplitComplex` structure.

Instance Properties

`var mutableSpan: MutableSpan<Element>`

`var span: Span<Element>`

Instance Methods

```
func withUnsafeTaggedBuffers<R>(([CMTaggedBuffer]) throws -> sending R) rethrows -> sending R
```

Access the underlying CMTaggedBuffers.

Type Aliases

```
typealias Specification
```

```
typealias UnderlyingSequence
```

```
typealias UnwrappedType
```

```
typealias ValueType
```

Type Properties

```
static var defaultResolverSpecification: EmptyResolverSpecification<Array<Element>>
```

Type Methods

```
static func monoscopicForVideoOutput() -> [CMTag]
```

Creates a collection of CMTags with the required tags to describe monoscopic video, where there is no stereo view, e.g. kCMTagStereoNone.

```
static func stereoscopicForVideoOutput() -> [CMTag]
```

Creates a collection of CMTags with the required tags to describe basic stereoscopic video, where both left and right stereo eyes are present, e.g. kCMTagStereoLeftAndRight.

Default Implementations

- BidirectionalCollection Implementations
- Collection Implementations
- CustomDebugStringConvertible Implementations
- CustomReflectable Implementations
- CustomStringConvertible Implementations

- Decodable Implementations
- Encodable Implementations
- Equatable Implementations
- ExpressibleByArrayLiteral Implementations
- Hashable Implementations
- MutableCollection Implementations
- OperationParameter Implementations
- RandomAccessCollection Implementations
- RangeReplaceableCollection Implementations
- Sequence Implementations

Relationships

Conforms To

`AccelerateBuffer`
`AccelerateMutableBuffer`
`AppExtensionScene`
`Attachable`
`BNNSGraph.Builder.OperationParameter`
`BidirectionalCollection`

Conforms when `Element` conforms to `Copyable` and `Escapable`.

`CKRecordValueProtocol`
`CMSampleBuffer.Content`
`CMSampleBuffer.ContentWithFormatDescription`
`CVarArg`

Conforms when `Element` conforms to `Copyable` and `Escapable`.

`Collection`

Conforms when `Element` conforms to `Copyable` and `Escapable`.

`ContiguousBytes`
`ConvertibleFromGeneratedContent`
`ConvertibleToGeneratedContent`
`Copyable`

Conforms when `Element` conforms to `Copyable` and `Escapable`.

`CustomDebugStringConvertible`

Conforms when `Element` conforms to `Copyable` and `Escapable`.

`CustomReflectable`

Conforms when `Element` conforms to `Copyable` and `Escapable`.

`CustomStringConvertible`

Conforms when `Element` conforms to `Copyable` and `Escapable`.

`DataProtocol`

`Decodable`

Conforms when `Element` conforms to `Decodable`.

`DecodableWithConfiguration`

`Encodable`

Conforms when `Element` conforms to `Encodable`.

`EncodableWithConfiguration`

`Equatable`

Conforms when `Element` conforms to `Equatable`.

`ExpressibleByArrayLiteral`

Conforms when `Element` conforms to `Copyable` and `Escapable`.

`Generable`

`Hashable`

Conforms when `Element` conforms to `Hashable`.

`InstructionsRepresentable`

`MLDataValueConvertible`

`MutableCollection`

Conforms when `Element` conforms to `Copyable` and `Escapable`.

`MutableDataProtocol`

`PositionScaleRange`

`PromptRepresentable`

`RandomAccessCollection`

Conforms when `Element` conforms to `Copyable` and `Escapable`.

`RangeReplaceableCollection`

Conforms when `Element` conforms to `Copyable` and `Escapable`.

`RelationshipCollection`

`ResultsCollection`

`ScaleDomain`

`ScaleRange`

`Sendable`

Conforms when `Element` conforms to `Copyable`, `Escapable`, and `Sendable`.

`SendableMetatype`

Conforms when `Element` conforms to `Copyable`, `Escapable`, and `Sendable`.

`Sequence`

Conforms when `Element` conforms to `Copyable` and `Escapable`.

See Also

Standard Library

`struct Int`

A signed integer value type.

`struct Double`

A double-precision, floating-point value type.

`struct String`

A Unicode string value that is a collection of characters.

`struct Dictionary`

A collection whose elements are key-value pairs.

Swift Standard Library

Solve complex problems and write high-performance, readable code.