Protocol

Sequence

A type that provides sequential, iterated access to its elements.

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

```
protocol Sequence<Element>
```

Overview

A sequence is a list of values that you can step through one at a time. The most common way to iterate over the elements of a sequence is to use a `for-in` loop:

```
let oneTwoThree = 1...3
for number in oneTwoThree {
    print(number)
}
// Prints "1"
// Prints "2"
// Prints "3"
```

While seemingly simple, this capability gives you access to a large number of operations that you can perform on any sequence. As an example, to check whether a sequence includes a particular value, you can test each value sequentially until you've found a match or reached the end of the sequence. This example checks to see whether a particular insect is in an array.

```
let bugs = ["Aphid", "Bumblebee", "Cicada", "Damselfly", "Earwig"]
var hasMosquito = false
for bug in bugs {
    if bug == "Mosquito" {
        hasMosquito = true
        break
    }
}
print("'bugs' has a mosquito: \(hasMosquito)")
// Prints "'bugs' has a mosquito: false"
```

The `Sequence` protocol provides default implementations for many common operations that depend on sequential access to a sequence's values. For clearer, more concise code, the example above could use the array's `contains(_:)` method, which every sequence inherits from `Sequence`, instead of iterating manually:

```
if bugs.contains("Mosquito") {
    print("Break out the bug spray.")
} else {
    print("Whew, no mosquitos!")
}
// Prints "Whew, no mosquitos!"
```

Repeated Access

The `Sequence` protocol makes no requirement on conforming types regarding whether they will be destructively consumed by iteration. As a consequence, don't assume that multiple `for-in` loops on a sequence will either resume iteration or restart from the beginning:

```
for element in sequence {
    if ... some condition { break }
}

for element in sequence {
    // No defined behavior
}
```

In this case, you cannot assume either that a sequence will be consumable and will resume iteration, or that a sequence is a collection and will restart iteration from the first element. A conforming sequence that is not a collection is allowed to produce an arbitrary sequence of elements in the second `for-in` loop.

To establish that a type you've created supports nondestructive iteration, add conformance to the `Collection` protocol.

Conforming to the Sequence Protocol

Making your own custom types conform to `Sequence` enables many useful operations, like `for-in` looping and the `contains` method, without much effort. To add `Sequence` conformance to your own custom type, add a `makeIterator()` method that returns an iterator.

Alternatively, if your type can act as its own iterator, implementing the requirements of the `IteratorProtocol` protocol and declaring conformance to both `Sequence` and `IteratorProtocol` are sufficient.

Here's a definition of a `Countdown` sequence that serves as its own iterator. The `makeIterator()` method is provided as a default implementation.

```swift
struct Countdown: Sequence, IteratorProtocol {
    var count: Int

    mutating func next() -> Int? {
        if count == 0 {
            return nil
        } else {
            defer { count -= 1 }
            return count
        }
    }
}

let threeToGo = Countdown(count: 3)
for i in threeToGo {
    print(i)
}
// Prints "3"
// Prints "2"
// Prints "1"
```

Expected Performance

A sequence should provide its iterator in O(1). The `Sequence` protocol makes no other requirements about element access, so routines that traverse a sequence should be considered O(*n*) unless documented otherwise.

Topics

Creating an Iterator

`func makeIterator() -> Self.Iterator`

Returns an iterator over the elements of this sequence.

Required Default implementations provided.

`associatedtype Iterator : IteratorProtocol`

A type that provides the sequence's iteration interface and encapsulates its iteration state.

Required

`associatedtype Element`

A type representing the sequence's elements.

Required

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 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) -> PrefixSequence<Self>
```

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

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

Returns a sequence containing the initial, consecutive elements that satisfy the given predicate.

```
func suffix(Int) -> [Self.Element]
```

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

Excluding Elements

```
func dropFirst(Int) -> DropFirstSequence<Self>
```

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

```
func dropLast(Int) -> [Self.Element]
```

Returns a sequence containing all but the given number of final elements.

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

Returns a sequence by skipping the initial, consecutive elements that satisfy the given predicate.

```
func filter((Self.Element) throws -> Bool) rethrows -> [Self.Element]
```

Returns an array containing, in order, the elements of the sequence that satisfy the given predicate.

Transforming a Sequence

`func map<T, E>((Self.Element) throws(E) -> T) throws(E) -> [T]`

Returns an array containing the results of mapping the given closure over the sequence's elements.

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

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

`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 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.

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

Iterating Over a Sequence'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.

```
var underestimatedCount: Int
```

A value less than or equal to the number of elements in the sequence, calculated nondestructively.

Required Default implementations provided.

Sorting 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 reversed() -> [Self.Element]
```

Returns an array containing the elements of this sequence in reverse order.

Reordering a Sequence's Elements

```
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.

Formatting a Sequence

```
func formatted() -> String
```

```
func formatted<S>(S) -> S.FormatOutput
```

```
struct ListFormatStyle<Style, Base> where Style : FormatStyle, Base :
Sequence, Style.FormatInput == Base.Element, Style.FormatOutput ==
String
```

A type that formats lists of items with a separator and conjunction appropriate for a given locale.

Splitting and Joining Elements

`func split(maxSplits: Int, omittingEmptySubsequences: Bool, where Separator: (Self.Element) throws -> Bool) rethrows -> [ArraySlice<Self.Element>]`

Returns the longest possible subsequences of the sequence, in order, that don't contain elements satisfying the given predicate. Elements that are used to split the sequence are not returned as part of any subsequence.

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

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

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

Returns the elements of this sequence of sequences, concatenated.

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

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

Comparing Sequences

`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.

Accessing Underlying Storage

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

Executes a closure on the sequence's contiguous storage.

Required Default implementation provided.

Publishing a Sequence

`var publisher: Publishers.Sequence<Self, Never>`

Applying AppKit Graphic Operations

Use a sequence of rectangles and other types to perform operations on an AppKit graphic context.

`func fill(using: NSCompositingOperation)`

Fills this list of rects in the current NSGraphicsContext in the context's fill color. The compositing operation of the fill defaults to the context's compositing operation, not necessarily using `.copy` like `NSRectFill()`.

`func fill(using: NSCompositingOperation)`

Fills this list of rects in the current NSGraphicsContext with that rect's associated color The compositing operation of the fill defaults to the context's compositing operation, not necessarily using `.copy` like `NSRectFill()`.

`func clip()`

Modifies the current graphics context clipping path by intersecting it with the graphical union of this list of rects This permanently modifies the graphics state, so the current state should be saved beforehand and restored afterwards.

Instance Methods

```
func compare<Comparator>(Comparator.Compared, Comparator.Compared) -> ComparisonResult
```

If `lhs` is ordered before `rhs` in the ordering described by the given sequence of `SortComparators`

```
func count<E>(where: (Self.Element) throws(E) -> Bool) throws(E) -> Int
```

Returns the number of elements in the sequence that satisfy the given predicate.

```
func donatedWithin<DonationInfo>(Tips.DonationTimeRange) -> [Self.Element]
```

```
func fill(using: NSCompositingOperation)
```

Fills this list of rects in the current NSGraphicsContext with that rect's associated gray component value in the DeviceGray color space. The compositing operation of the fill defaults to the context's compositing operation, not necessarily using `.copy` like `NSRectFillListWithGrays()`.

```
func filter(Predicate<Self.Element>) throws -> [Self.Element]
```

```
func filter<T>(matchingCategory: CMTypedTag<T>.Category) -> [CMTypedTag<T>]
```

Filters a sequence of tags based on matching the specified category. Returns the tags that match the specified category.

```
func first<T>(matchingCategory: CMTypedTag<T>.Category) -> CMTypedTag<T>?
```

Finds and returns the first tag matching the specified category.

```
func firstValue<T>(matchingCategory: CMTypedTag<T>.Category) -> T?
```

Finds the first tag matching the specified category and returns the value of the matching tag.

```
func largestSubset<DonationInfo, Value>(groupedBy: KeyPath<DonationInfo, Value>) -> [Self.Element]
```

```
func mapAnnotations<Feature, Input, Output>((Input) async throws -> Output) async rethrows -> [AnnotatedFeature<Feature, Output>]
```

Returns an array containing the results of mapping the given async closure over the sequence's annotations.

```
func mapAnnotations<Feature, Input, Output>((Input) throws -> Output)
rethrows -> [AnnotatedFeature<Feature, Output>]
```

Returns an array containing the results of mapping the given closure over the sequence's annotations.

```
func mapFeatures<Input, Output, Annotation>((Input) async throws ->
Output) async rethrows -> [AnnotatedFeature<Output, Annotation>]
```

Returns an array containing the results of mapping the given async closure over the sequence's features.

```
func mapFeatures<Input, Output, Annotation>((Input) throws -> Output)
rethrows -> [AnnotatedFeature<Output, Annotation>]
```

Returns an array containing the results of mapping the given closure over the sequence's features.

```
func randomSplit<Feature, Annotation>(by: Double, seed: Int?) -> ([
AnnotatedFeature<Feature, Annotation>], [AnnotatedFeature<Feature,
Annotation>])
```

Generates two AnnotatedFeatures by randomly splitting the elements of the sequence, at the same proportion within each unique Annotation.

```
func randomSplit<T>(by: Double, seed: Int?) -> (ArraySlice<T>, Array
Slice<T>)
```

Generates two generic arrays by randomly splitting the elements of the sequence.

```
func randomSplit<T, Generator>(by: Double, using: inout Generator) -> (
ArraySlice<T>, ArraySlice<T>)
```

Generates two generic arrays by randomly splitting the elements of the sequence.

```
func randomSplit<Feature, Annotation, Generator>(by: Double, using:
inout Generator) -> ([AnnotatedFeature<Feature, Annotation>], [
AnnotatedFeature<Feature, Annotation>])
```

Generates two AnnotatedFeatures by randomly splitting the elements of the sequence, at the same proportion within each unique Annotation.

```
func smallestSubset<DonationInfo, Value>(groupedBy: KeyPath<Donation
Info, Value>) -> [Self.Element]
```

```
func sorted<S, Comparator>(using: S) -> [Self.Element]
```

Returns the elements of the sequence, sorted using the given array of `SortComparators` to compare elements.

```
func sorted<Comparator>(using: Comparator) -> [Self.Element]
```

Returns the elements of the sequence, sorted using the given comparator to compare elements.

Type Methods

```
static func reparentEquipment(some Sequence<any Equipment>, childrenOf: some Equipment, order: MoveEquipmentAction.Order?, context: UInt64) -> Self
```

```
static func reparentEquipment(matching: some Sequence<EquipmentIdentifier>, childrenOf: EquipmentIdentifier, order: MoveEquipmentAction.Order?, context: UInt64) -> Self
```

Relationships

Inherited By

`BidirectionalCollection`
`Collection`
`LazyCollectionProtocol`
`LazySequenceProtocol`
`MutableCollection`
`RandomAccessCollection`
`RangeReplaceableCollection`
`StringProtocol`

Conforming Types

`AnyBidirectionalCollection`

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

`AnyCollection`

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

`AnyIterator`

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

`AnyRandomAccessCollection`

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

`AnyRegexOutput`
`AnySequence`

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

`Array`

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

`ArraySlice`

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

`ClosedRange`

Conforms when `Bound` conforms to `Strideable` and `Bound.Stride` conforms to `SignedInteger`.

`CollectionDifference`

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

`CollectionOfOne`

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

`ContiguousArray`

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

`DefaultIndices`

Conforms when `Elements` conforms to `Collection`.

`Dictionary`

Conforms when `Key` conforms to `Hashable`, `Value` conforms to `Copyable`, and `Value` conforms to `Escapable`.

`Dictionary.Keys`

Conforms when `Key` conforms to `Hashable`.

`Dictionary.Values`

Conforms when `Key` conforms to `Hashable`.

`DiscontiguousSlice`

Conforms when `Base` conforms to `Collection`.

`DropFirstSequence`

Conforms when `Base` conforms to `Sequence`.

`DropWhileSequence`

Conforms when `Base` conforms to `Sequence`.

`EmptyCollection`

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

`EmptyCollection.Iterator`

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

`EnumeratedSequence`

Conforms when `Base` conforms to `Sequence`.

`EnumeratedSequence.Iterator`

Conforms when `Base` conforms to `Sequence`.

`FlattenSequence`

Conforms when `Base` conforms to `Sequence` and `Base.Element` conforms to `Sequence`.

`FlattenSequence.Iterator`

Conforms when `Base` conforms to `Sequence` and `Base.Element` conforms to `Sequence`.

`IndexingIterator`

Conforms when `Elements` conforms to `Collection`.

`Int.Words`
`Int16.Words`
`Int32.Words`
`Int64.Words`
`Int8.Words`
`IteratorSequence`

Conforms when `Base` conforms to `IteratorProtocol`.

`JoinedSequence`

Conforms when `Base` conforms to `Sequence` and `Base.Element` conforms to `Sequence`.

`KeyValuePairs`

Conforms when `Key` conforms to `Copyable`, `Key` conforms to `Escapable`, `Value` conforms to `Copyable`, and `Value` conforms to `Escapable`.

`LazyDropWhileSequence`

Conforms when `Base` conforms to `Sequence`.

`LazyFilterSequence`

Conforms when `Base` conforms to `Sequence`.

`LazyFilterSequence.Iterator`

Conforms when `Base` conforms to `Sequence`.

`LazyMapSequence`

Conforms when `Base` conforms to `Sequence`, `Element` conforms to `Copyable`, and `Element` conforms to `Escapable`.

`LazyMapSequence.Iterator`

Conforms when `Base` conforms to `Sequence`, `Element` conforms to `Copyable`, and `Element` conforms to `Escapable`.

`LazyPrefixWhileSequence`

Conforms when `Base` conforms to `Sequence`.

`LazyPrefixWhileSequence.Iterator`

Conforms when `Base` conforms to `Sequence`.

`LazySequence`

Conforms when `Base` conforms to `Collection`.

`PartialRangeFrom`

Conforms when `Bound` conforms to `Strideable` and `Bound.Stride` conforms to `SignedInteger`.

`PrefixSequence`

Conforms when `Base` conforms to `Sequence`.

`Range`

Conforms when `Bound` conforms to `Strideable` and `Bound.Stride` conforms to `SignedInteger`.

`RangeSet.Ranges`

Conforms when `Bound` conforms to `Comparable`.

`Repeated`

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

`ReversedCollection`

Conforms when `Base` conforms to `BidirectionalCollection` and `LazySequenceProtocol`.

`ReversedCollection.Iterator`

Conforms when `Base` conforms to `BidirectionalCollection`.

`Set`

Conforms when `Element` conforms to `Hashable`.

`Slice`

Conforms when `Base` conforms to `Collection` and `LazySequenceProtocol`.

`StrideThrough`

Conforms when `Element` conforms to `Strideable`.

`StrideTo`

Conforms when `Element` conforms to `Strideable`.

`String`
`String.UTF16View`
`String.UTF8View`
`String.UnicodeScalarView`
`Substring`
`Substring.UTF16View`
`Substring.UTF8View`
`Substring.UnicodeScalarView`
`UInt.Words`
`UInt128.Words`
`UInt16.Words`
`UInt32.Words`
`UInt64.Words`
`UInt8.Words`
`UnfoldSequence`
`Unicode.Scalar.UTF16View`
`Unicode.Scalar.UTF8View`
`UnsafeBufferPointer`

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

`UnsafeMutableBufferPointer`

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

`UnsafeMutableRawBufferPointer`

`UnsafeRawBufferPointer`

`UnsafeRawBufferPointer.Iterator`

`Zip2Sequence`

Conforms when `Sequence1` conforms to `Sequence` and `Sequence2` conforms to `Sequence`.

See Also

First Steps

`protocol Collection`

A sequence whose elements can be traversed multiple times, nondestructively, and accessed by an indexed subscript.