orphan:

Sequences And Collections in Swift

Unlike many languages, Swift provides a rich taxonomy of abstractions for processing series of elements. This document explains why that taxonomy exists and how it is structured.

Sequences

It all begins with Swift's `for`...`in` loop:

```
for x in s {
  doSomethingWith(x)
}
```

Because this construct is generic, `s` could be

- an array
- a set
- a linked list
- a series of UI events
- a file on disk
- a stream of incoming network packets
- an infinite series of random numbers
- a user-defined data structure
- etc.

In Swift, all of the above are called **sequences**, an abstraction represented by the `SequenceType` protocol:

```
protocol SequenceType {
  typealias Iterator : IteratorProtocol
  func makeIterator() -> Iterator
}
```

As you can see, sequence does nothing more than deliver an iterator. To understand the need for iterators, it's important to distinguish the two kinds of sequences.

- **Volatile** sequences like "stream of network packets," carry their own traversal state, and are expected to be "consumed" as they are traversed.
- **Stable** sequences, like arrays, should *not* be mutated by `for`...`in`, and thus require *separate traversal state*.

To get an initial traversal state for an arbitrary sequence `x`, Swift calls `x.makeIterator()`. The sequence delivers that state, along with traversal logic, in the form of an **iterator**.

Hiding Iterator Type Details

A sequence's iterator is an associated type--rather than something like `AnyIterator<T>` that depends only on the element type--for performance reasons. Although the alternative design has significant usability benefits, it requires one dynamic allocation/deallocation pair and *N* dynamic dispatches to traverse a sequence of length *N*. That said, our optimizer has improved to the point where it can sometimes remove these overheads completely, and we are considering changing the design accordingly.

Iterators

`for`...`in` needs three operations from the iterator:

- get the current element
- advance to the next element
- detect whether there are more elements

If we literally translate the above into protocol requirements, we get something like this:

```
protocol NaiveIteratorProtocol {
  typealias Element
  var current() -> Element      // get the current element
  mutating func advance()       // advance to the next element
  var isExhausted: Bool         // detect whether there are more elements
}
```

Such a protocol, though, places a burden on implementors of volatile sequences: either the iterator must buffer the current element internally so that `current` can repeatedly return the same value, or it must trap when `current` is called twice without an intervening call to `moveToNext`. Both semantics have a performance cost, and the latter unnecessarily adds the possibility of incorrect usage.

Therefore, Swift's `IteratorProtocol` merges the three operations into one, returning `nil` when the iterator is exhausted:

```
protocol IteratorProtocol {
  typealias Element
  mutating func next() -> Element?
}
```

`NSEnumerator`

You might recognize the influence on iterators of the `NSEnumerator` API:

```
class NSEnumerator : NSObject {
  func nextObject() -> AnyObject?
```

Combined with `SequenceType`, we now have everything we need to implement a generic `for`...`in` loop.

```
  }
```

Adding a Buffer

The use-cases for singly-buffered iterators are rare enough that it is not worth complicating `IteratorProtocol`, [1] but support for buffering would fit nicely into the scheme, should it prove important:

```
public protocol BufferedIteratorProtoco
  : IteratorProtocol {
  var latest: Element? {get}
}
```

The library could easily offer a generic wrapper that adapts any `IteratorProtocol` to create a `BufferedIteratorProtocol`:

```
/// Add buffering to any IteratorProtoc
struct BufferedIterator<I : IteratorPro
  : BufferedIteratorProtocol {

  public init(_ baseIterator: I) {
    self._baseIterator = baseIterator
  }
  public func next() -> Element? {
    latest = _baseIterator.next() ?? la
    return latest
  }
  public private(set) var latest: I.Ele
  private var _baseIterator: I
}
```

Operating on Sequences Generically

Given an arbitrary `SequenceType`, aside from a simple `for`...`in` loop, you can do anything that requires reading elements from beginning to end. For example:

```
// Return an array containing the elements of `source`, with
// `separator` interposed between each consecutive pair.
func array<S: SequenceType>(
  _ source: S,
  withSeparator separator: S.Iterator.Element
) -> [S.Iterator.Element] {
  var result: [S.Iterator.Element] = []
  var iterator = source.makeIterator()
  if let start = iterator.next() {
    result.append(start)
    while let next = iterator.next() {
      result.append(separator)
      result.append(next)
    }
  }
  return result
}

let s = String(array("Swift", withSeparator: "|"))
print(s)        // "S|w|i|f|t"
```

Because sequences may be volatile, though, you can--in general--only make a single traversal. This capability is quite enough for many languages: the iteration abstractions of Java, C#, Python, and Ruby all go about as far as `SequenceType`, and no further. In Swift, though, we want to do much more generically. All of the following depend on stability that an arbitrary sequence can't provide:

- Finding a sub-sequence
- Finding the element that occurs most often
- Meaningful in-place element mutation (including sorting, partitioning, rotations, etc.)

Fortunately, many real sequences *are* stable. To take advantage of that stability in generic code, we'll need another protocol.

Iterators Should Be Sequences

In principle, every iterator is a volatile sequence containing the elements it has yet to return from `next()`. Therefore, every iterator *could* satisfy the requirements of `SequenceType` by simply declaring conformance, and returning `self` from its

`makeIterator()` method. In fact, if it weren't for current language limitations, `IteratorProtocol` would refine `SequenceType`, as follows:

```
protocol IteratorProtocol : SequenceTyp
  typealias Element
  mutating func next() -> Element?
}
```

Though we may not currently be able to *require* that every `IteratorProtocol` refines `SequenceType`, most iterators in the standard library do conform to `SequenceType`.

Collections

A **collection** is a stable sequence with addressable "positions," represented by an associated `Index` type:

```
protocol CollectionType : SequenceType {
  typealias Index : ForwardIndexType             // a position
  subscript(i: Index) -> Iterator.Element {get}

  var startIndex: Index {get}
  var endIndex: Index {get}
}
```

The way we address positions in a collection is a generalization of how we interact with arrays: we subscript the collection using its `Index` type:

```
let ith = c[i]
```

An **index**--which must model `ForwardIndexType`--is a type with a linear series of discrete values that can be compared for equality:

```
protocol ForwardIndexType : Equatable {
  typealias Distance : SignedIntegerType
  func successor() -> Self
}
```

Dictionary Keys

Although dictionaries overload `subscript` to also operate on keys, a `Dictionary`'s `Key` type is distinct from its `Index` type. Subscripting on an index is expected to offer direct access, without introducing overheads like searching or hashing.

While one can use `successor()` to create an incremented index value, indices are more commonly advanced using an in-place increment operator, just as one would when traversing an array: `++i` or `i++`. These operators are defined generically, for all models of `ForwardIndexType`, in terms of the `successor()` method.

Every collection has two special indices: a `startIndex` and an `endIndex`. In an empty collection, `startIndex == endIndex`. Otherwise, `startIndex` addresses the collection's first element, and `endIndex` is the successor of an index addressing the collection's last element. A collection's `startIndex` and `endIndex` form a half-open range containing its elements: while a collection's `endIndex` is a valid index value for comparison, it is not a valid index for subscripting the collection:

```
if c.startIndex != c.endIndex { } // OK
c[c.endIndex]                     // Oops! (index out-of-range)
```

Mutable Collections

A **mutable collection** is a collection that supports in-place element mutation. The protocol is a simple refinement of `CollectionType` that adds a subscript setter:

```
protocol MutableCollectionType : CollectionType {
  subscript(i: Index) -> Iterator.Element { get set }
}
```

The `CollectionType` protocol does not require collection to support mutation, so it is not possible to tell from the protocol itself whether the order of elements in an instance of a type that conforms to `CollectionType` has a domain-specific meaning or not. (Note that since elements in collections have stable indices, the element order within the collection itself is stable; the order sometimes does not have a meaning and is not chosen by the code that uses the collection, but by the implementation details of the collection itself.)

`MutableCollectionType` protocol allows the caller to replace a specific element, identified by an index, with another one in the same position. This capability essentially allows to rearrange the elements inside the collection in any order, thus types that conform to `MutableCollectionType` can represent collections with a domain-specific element order (not every instance of a `MutableCollectionType` has an interesting order, though).

Range Replaceable Collections

The `MutableCollectionType` protocol implies only mutation of content, not of structure (for example, changing the number of elements). The `RangeReplaceableCollectionType` protocol adds the capability to perform structural mutation, which in its most general form is expressed as replacing a range of elements, denoted by two indices, by elements from a collection with a **different** length.

```
public protocol RangeReplaceableCollectionType : MutableCollectionType {
  mutating func replaceSubrange<
    C: CollectionType where C.Iterator.Element == Self.Iterator.Element
  >(
    _ subRange: Range<Index>, with newElements: C
  )
}
```

Index Protocols

As a generalization designed to cover diverse data structures, `CollectionType` provides weaker guarantees than arrays do. In particular, an arbitrary collection does not necessarily offer efficient random access; that property is determined by the protocol conformances of its `Index` type.

Forward indices are the simplest and most general, capturing the capabilities of indices into a singly-linked list:

1. advance to the next position
2. detect the end position

Bidirectional indices are a refinement of forward indices that additionally support reverse traversal:

```
protocol BidirectionalIndexType : ForwardIndexType {
  func predecessor() -> Self
}
```

Indices into a doubly-linked list would be bidirectional, as are the indices that address `Character`s and `UnicodeScalar`s in a `String`. Reversing the order of a collection's elements is a simple example of a generic algorithm that depends on bidirectional traversal.

Random access indices have two more requirements: the ability to efficiently measure the number of steps between arbitrary indices addressing the same collection, and the ability to advance an index by a (possibly negative) number of steps:

```
public protocol RandomAccessIndexType : BidirectionalIndexType {
  func distance(to other: Self) -> Distance
  func advanced(by n: Distance) -> Self
}
```

From these methods, the standard library derives several other features such as `Comparable` conformance, index subtraction, and addition/subtraction of integers to/from indices.

The indices of a deque can provide random access, as do the indices into `String.UTF16View` (when Foundation is loaded) and, of course, array indices. Many common sorting and selection algorithms, among others, depend on these capabilities.

All direct operations on indices are intended to be lightweight, with amortized O(1) complexity. In fact, indices into `Dictionary` and `Set` *could* be bidirectional, but are limited to modeling `ForwardIndexType` because the APIs of `NSDictionary` and `NSSet`--which can act as backing stores of `Dictionary` and `Set`--do not efficiently support reverse traversal.

Conclusion

Swift's sequence, collection, and index protocols allow us to write general algorithms that apply to a wide variety of series and data structures. The system has been both easy to extend, and predictably performant. Thanks for taking the tour!

---

[1] This trade-off is not as obvious as it might seem. For example, the C# and C++ analogues for `IteratorProtocol` (`IEnumerable` and `input iterator`) are saddled with the obligation to provide buffering.