Sequences

IN SWIFT

@jeffboek

Opal Labs

What are Sequences?

What are Sequences?

Swift defines the `SequenceType` as

What are Sequences?

Swift defines the `SequenceType` as

A type that can be iterated with a for...in loop.

What are Sequences?

Swift defines the `SequenceType` as

A type that can be iterated with a for...in loop.

→ Array

→ Dictionary

→ Set

```
for element in mySequence {
    ...
}
```

Under the hood

```
let generator = mySequence.generate()
while let element = generator.next() {
    ...
}
```

```
mySequence.generate()
```

```
protocol SequenceType {
    typealias Generator: GeneratorType
    func generate() -> Generator
}
```

`SequenceType's have a GeneratorType`

`GenoratorType`'s are responsible for Encapsulating iteration state and interface for iteration over a sequence.

Generators

```
protocol GeneratorType {
    typealias Element
    mutating func next() -> Self.Element?
}
```

Example

```
struct Deck {
    private var cards: [PlayingCard]

    static func standard52CardDeck() -> Deck {
        let suits: [Suit] = [.Spades, .Hearts, .Diamonds, .Clubs]
        let ranks: [Rank] = [.Ace, .Two, .Three, .Four, .Five, .Six, .Seven, .Eight, .Nine, .Ten, .Jack, .Queen, .King]

        var cards: [PlayingCard] = []
        for suit in suits {
            for rank in ranks {
                cards.append(PlayingCard(rank: rank, suit: suit))
            }
        }

        return Deck(cards)
    }

    init(_ cards: [PlayingCard]) {
        self.cards = cards
    }

    mutating func shuffle() {
        cards.shuffleInPlace()
    }

    mutating func deal() -> PlayingCard? {
        guard !cards.isEmpty else { return nil }

        return cards.removeLast()
    }
}
```

https://github.com/apple/example-package-deckofplayingcards

```
let numberOfCards = 10
var deck = Deck.standard52CardDeck()

for _ in 1...numberOfCards {
    guard let card = deck.deal() else {
        print("No More Cards!")
        break
    }
    print(card)
}
```

```
struct Deck: SequenceType {
    ...
}
```

```
struct DeckGenerator: GeneratorType {
    private var cards: [PlayingCard]
    private var index = 0

    init(cards: [PlayingCard]) {
        self.cards = cards
    }

    mutating func next() -> PlayingCard? {
        guard index < cards.count else { return .None }
        return cards[index++]
    }
}
```

```
extension Deck: SequenceType {
    func generate() -> DeckGenerator {
        return DeckGenerator(cards: cards)
    }
}
```

usage

```
let deck = Deck.standard52CardDeck()
deck.prefix(10).forEach { card in
    ...
}
```

With `SequenceType` you get all of the powerful methods you get with `Array` and `Dicitonary`

`map, filter, forEach, dropFirst, dropLast, prefix, suffix, split...`

CollectionType

```
extension Deck: CollectionType {
    var startIndex: Int { return 0 }
    var endIndex: Int { return cards.count }

    subscript(i: Int) -> PlayingCard {
        return cards[i]
    }
}
```

a collection is multi-pass: any element may be revisited merely by saving its index.

```
// is the same as `for card in deck {}:`
for i in deck.startIndex..<deck.endIndex {
    let card = deck[i]
    ...
}
```

Grabbing the 11th card

```
let card = deck[10]
```

Like `SequenceType` Swift gives you a handful of functions.

`find, indices, partition, reverse, sort...`

AnyGenerator<T>

```
struct Deck: SequenceType {
    func generate() -> DeckGenerator {
        return DeckGenerator(cards: cards)
    }
}
```

```
struct Deck: SequenceType {
    func generate() -> AnyGenerator<PlayingCard> {
        var index = 0

        // pass it a "next" closure
        return anyGenerator {
            guard index < self.cards.count else { return .None }
            return self.cards[index++]
        }
    }
}
```

To infinity, and beyond

```swift
var positiveNumbers: AnyGenerator<Int> {
    var i = 0
    return anyGenerator { return i++ }
}

for i in positiveNumbers {
    print(i)
}
// 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15....
```

```
var positiveNumbers: AnyGenerator<Int> {
    var i = 0
    return anyGenerator { return i++ }
}

let evenNumbers = positiveNumbers.filter { $0 % 2 == 0 } // hangs...
let firstTenEvenNumbers = evenNumbers.prefix(10)
```

Lazy

```
var positiveNumbers: AnyGenerator<Int> {
    var i = 0
    return anyGenerator { return i++ }
}

let evenNumbers = positiveNumbers.lazy.filter { $0 % 2 == 0 }
let firstTenEvenNumbers = evenNumbers.prefix(10)
//[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

```
func fibonacciGenerator() -> AnyGenerator<Int> {
    var n1 = 0
    var n2 = 1

    return anyGenerator {
        (n1, n2) = (n2, n1 + n2)
        return n1
    }
}

for i in fibonacciGenerator() {
    print(i)
}
```

```
// 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
// 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229
// 832040 1346269 2178309 3524578 5702887 9227465 14930352 24157817
// 39088169 63245986 102334155 165580141 267914296 433494437
// 701408733 1134903170 1836311903 2971215073 4807526976 7778742049
// 12586269025...
```

Only calculates 10 times.

```
let sumOfFirstTen = fibonacciGenerator().prefix(10).reduce(0, prefix: +)
```

Beyond

@krzyzanowskim

http://blog.krzyzanowskim.com/2015/06/26/paging/

```swift
protocol AsyncGeneratorType {
    typealias Element
    typealias Fetch
    mutating func next(fetchNextBatch: Fetch, onFinish: ((Element) -> Void)?)
}

/// Generator is the class because struct is captured in asynchronous operations so offset won't update.
class PagingGenerator<T>: AsyncGeneratorType {
    typealias Element = Array<T>
    typealias Fetch = (offset: Int, limit: Int, completion: (result: Element) -> Void) -> Void

    var offset:Int
    let limit: Int

    init(startOffset: Int = 0, limit: Int = 25) {
        self.offset = startOffset
        self.limit = limit
    }

    func next(fetchNextBatch: Fetch, onFinish: ((Element) -> Void)? = nil) {
        fetchNextBatch(offset: offset, limit: limit) { [unowned self] (items) in
            onFinish?(items)
            self.offset += items.count
        }
    }
}
```

```
var paging = PagingGenerator<Contact>(startOffset: 0, limit: 25)
paging.next(...)
```

```
class ViewController: UIViewController {
    @IBOutlet var tableView: UITableView!
    private var paging = PagingGenerator<Contact>(startOffset: 0, limit: 25)

    private var contacts = [Contact]() {
        didSet {
            tableView.reloadData()
        }
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        paging.next(fetchNextBatch, onFinish: updateDataSource) // first page
    }
}
```

```
//MARK: Paging

extension ViewController {
    private func fetchNextBatch(offset: Int, limit: Int, completion: (Array<Contact>) -> Void) -> Void {
        if let remotelyFetched = downloadGithubUsersPage(offset) {
            completion(remotelyFetched)
        }
    }

    private func updateDataSource(items: Array<Contact>) {
        self.contacts += items
    }
}

//MARK: UITableViewDelegate

extension ViewController: UITableViewDelegate {
    func tableView(tableView: UITableView, willDisplayCell cell: UITableViewCell, forRowAtIndexPath indexPath: NSIndexPath) {
        if indexPath.row == tableView.dataSource!.tableView(tableView, numberOfRowsInSection: indexPath.section) - 1 {
            paging.next(fetchNextBatch, onFinish: updateDataSource)
        }
    }
}
```

Thank You