A Functional Programming Kickstart

DANIEL H STEINBERG

Introducing Functional Programming Fundamentals in Swift

Editors Cut
A Functional Programming Kickstart

Introducing Functional Programming Fundamentals In Swift

by Daniel H Steinberg

Editors Cut
Copyright

"A Functional Programming Kickstart", by Daniel H Steinberg

Copyright © 2020 Dim Sum Thinking, Inc. All rights reserved.


Book Version

This is version 0.9 for Swift 5.3, Xcode 12, macOS Big Sur, and iOS 14 released November 2020.

Legal

Every precaution was taken in the preparation of this book. The publisher and author assume no responsibility for errors and omissions, or for damages resulting from the use of the information contained herein and in the accompanying code downloads.

The sample code is intended to be used to illustrate points made in the text. It is not intended to be used in production code.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks or service marks. Where those designations appear in this book, and Dim Sum Thinking, Inc. was aware of the trademark claim, the designations have been printed with initial capital letters or in all capitals.

This book uses terms that are registered trademarks of Apple Inc. for which the terms of use don’t permit rendering them in all caps or initial caps. You can view a complete list of

The Editor’s Cut name and logo are registered trademarks of Dim Sum Thinking, Inc.
Playing with a Full Deck
Playing With A Full Deck

In this section let's play with a Deck of Cards. We'll some of the higher-order functions we've met in this chapter and create one or two of our own.

Setup

We start with a freshDeck and deal() twenty cards off the top and display them in a Hand using a HandView. You know how to do all of this, so I've given you this starting point in the 08PlayingWithAFullDeck playground page.

var result = freshDeck

// the rest of your code in this section will go here

let hand = result.deal(20).hand
import PlaygroundSupport
PlaygroundPage.current.setLiveView(HandView(of: hand))
Run the playground to see this in the live view.

That's the top twenty cards in order. Let's see if we can shuffle the deck.

**Of course we can**

Array includes two built-in functions `shuffle()` and `shuffled()`.

Like `sort()` and `sorted()`, the difference is that `shuffle()` is mutating and changes the `Array` in place while `shuffled()` is non-mutating and leaves the original `Array` as it is while returning a new `Array` that has been shuffled.

Here's how we use `shuffled()`. 
let shuffled = freshDeck.shuffled()

Run the playground and you'll see that `shuffled` contains all of the `Cards` from `freshDeck` in a random order.

But wait.

Run the playground again. As before, `shuffled` contains all of the `Cards` from `freshDeck` in a random order. But it's a different random order.

This is an external dependency on a random number generator that keeps us from getting predictable results or testing our code. We want a deterministic shuffle. Much later in the book we'll create a pseudo-random number generator using more advanced techniques.

**A Perfect Shuffle**

If you have a complete `Deck` of fifty-two `Cards`, here's how you execute a perfect shuffle.

First, you divide the `Cards` exactly in half. You have the top twenty-six `Cards` in one hand - say it's your left hand, and the bottom twenty-six in your other hand - say it's your right hand.
Next, you alternate cards from each hand into a combined pile on the table in front of you. In a perfect shuffle you alternate exactly one Card at a time. You can imagine the top Card is the former top of the Deck. It is also the top Card in your left hand.

The second Card is the former twenty-seventh Card of the Deck or the top card of the portion of the Deck in your right hand. The third Card is the second Card in your left hand and the fourth is the second Card in your right hand. And so on.

This is just zip().
If you've shuffled a deck of cards then you know that there's that moment when the cards are interleaved but you have to straighten them back into a neat pile.

That's where we are now.

tuples is a Zip2Sequence where each tuple is a pair of Cards. Let's use reduce() to straighten this back into a Deck.

The initial value for reduce() will be an empty Deck and the rule to get from each step to the next is to take our deckSoFar and append the next tuple of Cards.

```swift
extension Deck {
    func perfectShuffle() -> Deck {
        let topHalf = self[0..<26]
        let bottomHalf = self[26...]  
        let tuples = zip(topHalf, bottomHalf)
        return tuples.reduce(Deck()){(deckSoFar, pair) in
                                     deckSoFar + [pair.0, pair.1]}
    }
}
```

Shuffle the freshDeck once and set the result equal to result.

```swift
let perfectlyShuffled = freshDeck.perfectShuffle()

[A♠, A♣, 2♠, 2♣, 3♠, 3♣, 4♠, 4♣, (...)]

result = perfectlyShuffled
```
You can see the **Cards** perfectly intermeshed. It might be easier if you run the playground to see the first ten cards of each of the Spades and Clubs neatly shuffled together.

The problem with the perfect shuffle is that it is perfect. It is predictable. In fact, if you perform a perfect shuffle eight times you end up restoring the deck to its initial state.
```swift
let perfectlyShuffled = freshDeck
    .perfectShuffle()
    .perfectShuffle()
    .perfectShuffle()
    .perfectShuffle()
    .perfectShuffle()
    .perfectShuffle()
    .perfectShuffle()
    .perfectShuffle()

[A♠, 2♠, 3♠, 4♠, 5♠, 6♠, 7♠, (...)]

perfectlyShuffled == freshDeck
true

We need something less perfect.

A less perfect shuffle

What happens if we don't cut the Deck exactly in half? Here's a version of shuffle() that let's us specify how deep the cut is.

Add this method to the playground. I've only highlighted the differences between it and perfectShuffle().```
extension Deck {
    func shuffle(cutDepth index: Int) -> Deck {
        let topHalf = self[0..<index]
        let bottomHalf = self[index...]
        let tuples = zip(topHalf, bottomHalf)
        return tuples.reduce(Deck()){(deckSoFar, pair) in
            deckSoFar + [pair.0, pair.1]
        }
    }
}

Let's use it and make a shallow cut.

let shuffled = freshDeck.shuffle(cutDepth: 3)
ALESPE, 4♣, 2♣, 5♣, 3♠, 6♠

The result is [A♠, 4♣, 2♣, 5♣, 3♠, 6♠]. That's it. Those six Cards. It's because when we zip the three Cards in the topHalf with the forty-nine Cards in the bottomHalf we stop zipping after the topHalf is exhausted.

What happens in a physical shuffle?

Once we exhaust the shorter half, we follow the alternating Cards with the rest of the half that still contains Cards.

Here's a revision to shuffle().
extension Deck {
    func shuffle(cutDepth index: Int) -> Deck {
        let topHalf = self[0..<index]
        let bottomHalf = self[index...]  
        let tuples = zip(topHalf, bottomHalf)
        let shuffledPart
            = tuples.reduce(Deck()){(deckSoFar, pair) in
                deckSoFar + [pair.0, pair.1]
            }  
        return shuffledPart
            + ((index < count/2 )
                ?  self[(index * 2)...]
                :  self[(count - index)..<index])
    }
}

If we try cut depth of 3 again we now see a full deck. After the first six cards we pick up with 7♠, 8♠, 9♠ and so on.

let shuffled = freshDeck.shuffle(cutDepth: 3)

[A♠ , 4♠ , 2♠ , 5♠ , 3♠ , 6♠ , 7♠ , 8♠ , 9♠ , (...)]

Similarly, we can cut so deep as to not leave many cards in the bottomHalf.

let shuffled = freshDeck.shuffle(cutDepth: 45)

[A♠ , 7♥ , 2♠ , 8♥ , 3♠ , 9♥ , 4♠ , 10♥ , 5♠ , J♥ , (...)]
The bottom seven cards in the Deck are shuffled with the first seven followed by cards eight through forty-four.

Other Decks

There's nothing in the shuffle() function that depends on this being a full Deck.

Let's filter the freshDeck to only contain the numbered cards in each suit. Remove the A, J, Q, and K.
let numberedCards = freshDeck.filter { card in
    card.rank < .jack
    && card.rank > .ace
}

[2♠, 3♠, 4♠, 5♠, 6♠, 7♠, 8♠, 9♠, 10♠,
  2♦, 3♦, 4♦, 5♦, 6♦, 7♦, 8♦, 9♦, 10♦,
  2♣, 3♣, 4♣, 5♣, 6♣, 7♣, 8♣, 9♣, 10♣,
  2♥, 3♥, 4♥, 5♥, 6♥, 7♥, 8♥, 9♥, 10♥]

We can shuffle these thirty-two cards thoroughly by choosing different cut depths. Set it equal to `result`.

let shuffledNumbers = numberedCards
  .shuffle(cutDepth: 18)
  .shuffle(cutDepth: 17)
  .shuffle(cutDepth: 19)
  .shuffle(cutDepth: 18)

result = shuffledNumbers
Run the playground to see the top twenty Cards in the Deck.

Organize your hand

Sort the cards in your hand. We have to sort the hand and then display the sorted Hand.

```swift
import PlaygroundSupport
PlaygroundPage.current
    .setLiveView(HandView(of: finalHand))
```

```swift
result = shuffledNumbers

let hand = result.deal(20).hand

let finalHand = hand.sorted{$0.rank < $1.rank}
    .sorted{$0.suit < $1.suit}
```
Run the playground to see our sorted Hand.

In the next chapter we'll look at map() in a variety of settings.