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 trademarks and registered trademarks of Apple Inc at

The Editor’s Cut name and logo are registered trademarks of Dim Sum Thinking, Inc.
Result
Optionals and Errors

Since an Optional either has a value or is empty, we often use an Optional to handle errors.

For example, in our function `cardFromFreshDeck()` we did not check for valid input and risked crashing if the function received non-valid input.

```swift
public func cardFromFreshDeck(at index: Int) -> Card {
    freshDeck[index]
}
```

If we called this with a value outside of the Deck's indices this crashed the playground. For example,

```
cardFromFreshDeck(at: 100)
// this crashes playground
```

So earlier in this chapter we created a version of this function named `safeCardFromFreshDeck()` that throws an error.

```swift
public func safeCardFromFreshDeck(at index: Int) throws -> Card {
    guard (0...51).contains(index) else {
        throw ThatsJustStupid(yourNumber: index)
    }
    return freshDeck[index]
}
```
This was much safer but required that we use `do - try - catch`.

Often we use an `Optional` to say "go ahead and give me a `Card` if you can. If you can't, just return `nil`."

For instance, we could add this function named `optionalCardFromFreshDeck()` to `BagOfTricks.swift`. I've highlighted the main difference between it and `safeCardFromFreshDeck()`.

```swift
05/Map.playground/Sources/BagOfTricks.swift
public func optionalCardFromFreshDeck(at index: Int) -> Card? {
    guard 0...51.contains(index) else {
        return nil
    }
    return freshDeck[index]
}
```

Use it in the playground and you'll see how easy it is. As always, I've reported the result properly as opposed to how you will actually see it in the playground.

```swift
05/Map.playground/05Result
optionalCardFromFreshDeck(at: 17)  // Optional(5♦)
optionalCardFromFreshDeck(at: 100) // nil
```

This is nice and easy to work with. But what if we do want to know what went wrong if there's an error without the pain of having to try and catch at every step?

**The Result type**
The Result type is just like Optional but it replaces nil with a Failure that we can use to express what went wrong.

Result is also implemented as an enum, something like this. Note it is generic in two types, one for Success and one for Failure. In fact, Failure is constrained to conform to Error.

```swift
// no need to type this code in this is in the Standard Library
denum Result<Success, Failure> where Failure: Error {
    case success(Success)
    case failure(Failure)
}
```

Add this function to BagOfTricks.swift named resultCardFromFreshDeck() that creates a success case with associated value equal to index if index is valid and a failure case with associated value equal to an error of type ThatsJustStupid.

Again, although this entire function is to be added, I'm just highlighting what is different about it and the optional and throwing versions.

```
05/Map.playground/Sources/BagOfTricks.swift

public func resultCardFromFreshDeck(at index: Int) -> Result<Card, ThatsJustStupid> {
    guard (0...51).contains(index) else {
        return .failure(ThatsJustStupid(yourNumber: index))
    }
    return .success(freshDeck[index])
}
```

Note that in the success case we have to wrap our Card in our Result type, the same way a non-nil Optional is wrapped.

Call this function from the 05Result playground page.
let card17 = resultCardFromFreshDeck(at: 17)  
  success(5♦)
let card100 = resultCardFromFreshDeck(at: 100)
  failure(You can't choose 100. It's not between 0 and 51)

We now have a success and a failure. It's time for map().

map() for Result

Take another look at Result and ask yourself, "what should map() for Result do?"

enum Result<Success, Failure> where Failure: Error {
    case success(Success)
    case failure(Failure)
}

Compare Result with what we saw with Optional.

enum Optional<Wrapped> {
    case some(Wrapped)
    case none
}

In Result, success is playing the role of some and failure is a more information rich version of none (nil).

Remember, what the map() did in Optional depended on whether the Optional instance was nil or some(value). map() returned none if it was given none, and map() took some(value) and returned some(transform(value)).

The map() for Result is similar to map() for Optional. map() is generic in type Output. It transforms the Success type to Output and leaves the Failure type
alone.

In other words, the signature of `map()` looks like this.

```swift
// don't type this as it exists already
extension Result {
    func map<Output>(_ transform: (Success) -> Output)
        -> Result<Output, Failure> {
        // not implemented yet
    }
}
```

The transform can only apply in the `success` case. So we treat the `success` case exactly as we treated the `some` case for `Optional`.

The `failure` case corresponds to the `none` case for `Optional`. We just change the type of the `Result` from `Result<Success, Failure>` to `Result<Output, Failure>`. The `Failure` type didn't change so we forward the error.

```swift
// don't type this as it exists already
extension Result {
    func map<Output>(_ transform: (Success) -> Output)
        -> Result<Output, Failure> {
        switch self {
        case .success(let success):
            return .success(transform(success))
        case .failure(let failure):
            return .failure(failure)
        }
    }
}
```

Take a moment to implement the custom operator for `map()` in `Sources/CustomOperators.swift`. The only real difference between implementing `<>^` for `Result` and `Optional` is that we must specify that the beginning and end `Failure` types are the same.
public func <^> <Input, Output, Failure: Error>(xs: Result<Input, Failure>,
    f: (Input) -> Output) -> Result<Output, Failure> {
    xs.map(f)
}

Let's use map() on card17 and card100.

Using map()

Both card17 and card100 are of type Result<Card, ThatsJustStupid>.

To map() them we need a function that takes a Card as its input. It's not very exciting, but let's use increment().

We expect card17, which is success(5♦) to be transformed to success(6♣). The map() function reaches inside of the Result, extracts the associated value of the success, transforms that value using the transform function, and then rewraps it in success.

05/Map.playground/05Result

card17
  .map(increment)

success(6♣)

Here comes the cool part. card100 is a failure. It's an Error. But we don't have to try catch it as long as it remains inside of a Result. What map() does to a failure is it just carries it forward. If transform changes the type of the Success then map() changes the Result type holding this failure so that the Success type matches.
I love that we don't have to stop and deal with the *Failure* until we're ready to.

**One more trunk**

For fun, introduce an *Error* type to use in an example that will use our functions named `emphasize()` and `numberOfCharacters`. Name the error type `IsEmpty`.

```swift
struct IsEmpty: Error, CustomDebugStringConvertible {
    var debugDescription: String {
        "This is empty"
    }
}
```

Let's use `IsEmpty` to create a *Result* type that works like our trunks did. Our trunks either contained an element of some generic type or they were empty.

Use a *typedef* to declare that `Trunk` is a *Result* type where the *Success* can be any type but the *Failure* is `IsEmpty`.

```swift
typealias Trunk<Success> = Result<Success, IsEmpty>
```
We can create `init()`s similar to the ones we used previously.

```
05/Map.playground/05Result

extension Trunk where Failure == IsEmpty {
    init() {
        self = .failure(IsEmpty())
    }
    init(_ whatsInside: Success) {
        self = .success(whatsInside)
    }
}
```

Create an empty `Trunk` and a `Trunk` that contains the `String "Hello"`.

```
05/Map.playground/05Result

let emptyTrunk = Trunk<String>()

let helloTrunk = Trunk("Hello")
```

Let's use `Result`'s map on these.

**Mapping Trunk**

No matter what we try to do with our `emptyTrunk`, it will always be an `emptyTrunk`. 
 emptyTrunk.map(emphasize) failure(This is empty)

 emptyTrunk.map(emphasize).map(numberOfCharacters) failure(This is empty)

The type of the first expression is Trunk<String> and the type of the second expression is Trunk<Int>.

Similarly, we can map() helloTrunk. Our helloTrunk remains a success although the contents change.

 helloTrunk.map(emphasize) success("HELLO!")

 helloTrunk.map(emphasize).map(numberOfCharacters) success(6)

In the chained calls to map() we don't have to worry about reaching into success to access the value HELLO! before moving on to the second map(). The map() method reaches inside to access the associated value, transforms it, then rewraps the transformed value in success.

Before we move on, remember we defined our custom operator <> for Result. Let's use it.
This chain of calls looks identical to what we did for `Array` and `Optional`. The only difference is the type of our starting value is a `Result<String, IsEmpty<>

I don't know about you, but I find that pretty cool. It helps me see `map()` as a design pattern and not just as a one-off.

So ...

We've seen three versions of a trunk. `DustyTrunk` used a struct to model a container that may or may not have a single element inside. `ShinyTrunk` used `Optionals` and `Trunk` used the `Result` type.

The `map()` function was different in all three cases but once we had an instance of one of our types of trunks, we could map the same transform functions.

The key is that the container has a `map()` that knows how to reach inside and retrieve values. We only need to pass the `map()` a function that knows how to transform one of these values. The `map()` knows what to do if there is a value inside and it knows what to do if there isn't.

There's one more type I want to look at before we get to `Array` and that is the `Writer`. 