Functional Swift

Daniel H Steinberg

Singapore

January, 2019

http://dimsumthinking.com

Functional Swift

Daniel H Steinberg

Singapore

January, 2019

http://dimsumthinking.com

XCODE 10 - SWIFT 4 : JANUARY 2019

Functional Swift

An introduction to higher-order functions, map, and flatMap

Recommended Settings

The ePub is best viewed in scrolling mode using the original fonts. The ePub and Mobi versions of this book are best read in single column view.

Contact

Dim Sum Thinking at http://dimsumthinking.com

View the complete Course List.

email: inquiries@dimsumthinking.com.

Legal

TABLE OF CONTENTS

PART 1

Functions that return other functions

Getting Started

We have types for various currencies in our ***Sources*** directory. Create this method for calculating earnings at 15 Euros per hour.

```
func payAt15(for hours: Hours) -> Euros {
    let pay = hours * 15.euros.perHour
    return pay
}
```

We use the method like this.

```
payAt15(for: 3.5.hours)
```

A different rate

In the last section we created a function that calculated someone's pay at a fixed rate of € 15 per hour.

What if we now have someone being paid € 12 an hour. We don't want to have to add this function.

```
func payAt12(for hours: Hours) -> Euros {
    let pay = hours * 12.euros.perHour
    return pay
}

payAt12(for: 3.5.hours)
```

That's horrible.

We want to be able to first create the `payAt15` or `payAt12` functions and then provide them with the hours

Imagine a table view where you select an employee. A function is passed to the next view controller with the partially applied function that includes the employees pay rate.

Imagine splitting the calculation up either like this

```
pay(at: 15.euros.perHour)(for: 3.5.hours)
```

or like this

```
let employeePay = pay(at: 15.euros.perHour)
```

Where `employeePay` is a function from `Hours` to `Euros`.

You can then calculate

```
employeePay(for: 3.5.hours)
```

The familiar way using Multiple Parameters

```
func pay(at rateInEuros: Euros.Rate, for hours: Hours) -> Euros {
    let pay = rateInEuros * hours
    return pay
}
```

Call it like this:

```
pay(at: 15.euros.perHour,
    for: 3.5.hours)
```

We can't just apply the first parameter and get a function.

```
// can't do this in Swift
let employeePay = pay(at: 15.euros.perHour)
```

Higher Order Functions

Although we lose some of the flexibility in expression, we can get the desired result by creating and returning a function. We'll take a few steps to get where we want.

First let's figure out the signature.

We want to replace this.

```
func pay(at rateInEuros: Euros.Rate,
         for hours: Hours) -> Euros {
```

With this.

```
func pay(at rateInEuros: Euros.Rate) -> (Hours) -> Euros {
}
```

Build and return a function

We're going to build a function and then return it.

```
func pay(at rateInEuros: Euros.Rate) -> (Hours) -> Euros {
    func payCalculation(hours: Hours) -> Euros {
        let pay = rateInEuros * hours
        return pay
    }
    return payCalculation
}
```

A function is a closure

We can instead assign `payCalculation` to the closure.

```
func pay(at rateInEuros: Euros.Rate) -> (Hours) -> Euros {
    let payCalculation = {(hours: Hours) -> Euros in
        let pay = rateInEuros * hours
        return pay
    }
    return payCalculation
}
```

Return the closure

We can remove the explaining variable.

```
func pay(at rateInEuros: Euros.Rate) -> (Hours) -> Euros {
```

```
    return {(hours: Hours) -> Euros in
        return rateInEuros * hours
    }
}
```

Use it

Use our new function.

```
pay(at: 15.euros.perHour)(10.hours)

let pay15 = pay(at: 15.euros.perHour)
pay15(3.5.hours)
```

Clean up

Swift can infer a lot of this information.

```
func pay(at rateInEuros: Euros.Rate) -> (Hours) -> Euros {
    return {hours in rateInEuros * hours}
}
```

Take a moment to compare this to the two parameter version.

```
func pay(at rateInEuros: Euros.Rate,
         for hours: Hours) -> Euros {
    return rateInEuros * hours
}
```

Nested Closures

Take another look at

```
func pay(at rateInEuros: Euros.Rate) -> (Hours) -> Euros {
    return {hours in rateInEuros * hours}
}
```

We can write it even more compactly.

```
let altPay: (Euros.Rate) -> (Hours) -> Euros
     = {rate in {hours in rate * hours}}
```

Call it the same way you called `pay`.

```
altPay(12.euros.perHour)(10.hours)
```

We can capture the function type in a `typealias` if you like.

```
typealias PayCalculator = (Euros.Rate) -> (Hours) -> Euros

let altPay: PayCalculator  = {rate in {hours in rate * hours}}
```

Currying

We can also take advantage of this existing function

```
func pay(at rateInEuros: Euros.Rate,
         for hours: Hours) -> Euros {
    return rateInEuros * hours
}
```

And write our closure using it so that Swift can infer the types.

```
let altPay2 = {rate in {hours in pay(at: rate, for: hours)}}
```

We use it like this.

```
altPay2(10.euros.perHour)(7.5.hours)
```

Find the syntax that you and your team are comfortable with.

Challenge

Can you write a function that takes a function with two parameters and returns a curried version of that function? Can you write a second function that goes the other way?

A solution is below - no peeking.

Solution

```
func curry<A, B, C>(_ f: @escaping (A,  B) -> C) -> (A) -> (B) -> C {
    return {a in {b in f(a, b)}}
}

func uncurry<A, B, C>(_ f: @escaping (A) -> (B) -> C) -> (A, B) -> C {
    return { (a, b) in f(a)(b)}
}

let curriedPay = curry(pay)
curriedPay(5.euros.perHour)(7.hours)

let uncurriedAltPay = uncurry(altPay)
uncurriedAltPay(4.euros.perHour, 12.hours)
```

PART 2

Functions that consume other functions

Set up

We have this in our playground.

```
func pay(at rateInEuros: Euros.Rate,
         for hours: Hours) -> Euros {
    return rateInEuros * hours
}

func pay(at rateInEuros: Euros.Rate) -> (Hours) -> Euros {
    return {hours in rateInEuros * hours}
}
```

First class functions

The return value of the second `pay()` function is a function. This `pay()` is one form of what is called a higher-order function.

```
let pay15for = pay(at: 15.euros.perHour)
```

We can (and did) assign this function to a variable. We lose the labeled parameter but

`pay15for` is a reference value that points to a function.

Accepting Functions

We also call a function a higher-order function if it accepts another function. Here's an example.

```
func apply(_ f: (Hours) -> Euros,
           to startingValue: Hours) -> Euros {

}
```

We can implement it like this.

```
func apply(f: (Hours) -> Euros,
           to startingValue: Hours) -> Euros {
    return f(startingValue)
}
```

And we call it like this.

```
apply(f: pay15for, to: 3.5.hours)
```

We can also pass in a closure for the first argument.

```
apply(f: {hours in pay15for(hours)},
      to: 3.5.hours)
```

Actually, now that we call it, you can see that we might prefer not to have a label for the first parameter.

```
func apply(
```

```
_ f: (Hours) -> Euros,
            to startingValue: Hours) -> Euros {
    return f(startingValue)
}
```

This changes our calls.

```
apply(pay15for,
      to: 3.5.hours)

apply({hours in pay15for(hours)},
      to: 3.5.hours)
```

Trailing Closures

We prefer to create methods/functions that take no more than one function/closure and when we do we prefer that the function/closure be the last parameter.

```
func change(startingValue: Hours,
            using f: (Hours) -> Euros) -> Euros {
    return f(startingValue)
}

change(startingValue: 3.5.hours, using: pay15for)
```

Again, let's clean things up by getting rid of the first label.

```
func change(_ startingValue: Hours,
            using f: (Hours) -> Euros) -> Euros {
    return f(startingValue)
}
```

```
change(3.5.hours, using: pay15for)

change(3.5.hours, using: {hours in pay15for(hours)})
```

We can go further. If the last parameter is a closure, we can move it outside the parameter parentheses using a trailing closure.

```
change(3.5.hours){hours in pay15for(hours)}

change(3.5.hours){hours in
    pay(at: 15.euros.perHour)(hours)
}
```

In a playground, if we want to see the result we need to create a temp variable or use QuickLook.

```
let result = change(3.5.hours, using: {hours in pay15for(hours)})

result
```

Generic version

There are two more changes I'd like to make to `change()`. First, I want to follow the Swift naming convention and rename it `changed()`. More importantly, there's nothing special about `Hours` and `Euros`. Let's create a generic version.

```
func changed<Input, Output>(_ input: Input,
                            using f: (Input) -> Output) -> Output {
    return f(input)
}
```

We can use this exactly as before.

```
changed(3.5.hours){hours in
    pay(at: 15.euros.perHour)(hours)
}
```

$0

Actually, we can use `$0` to refer to the first parameter of a closure, `$1` for the second, and so on.

```
changed(3.5.hours){pay(at: 15.euros.perHour)($0)}
```

You'll have to decide what you consider more readable in your situation.

Many prefer

```
changed(3.5.hours){hours in
    pay(at: 15.euros.perHour)(hours)
}
```

Custom Operators

Often the custom operator `|>` is used for `changed()`.

Look in ***Sources*** > ***CustomOperators.swift*** and you'll see it defined like this>

```
public func |> <Input, Output>(x: Input, f: (Input) -> Output) -> Output {
    return f(x)
}
```

We can use it in the playground like this.

```
3.5.hours |> pay(at:15.euros.perHour)
```

We could also do the following if you like.

```
3.5.hours |> (15.euros.perHour |> pay)
```

Note, we have to use parentheses because `|>` is left associative.

At this point `changed()` seems silly. You've seen we could get the same result more simply. We could even do this

We're setting up for something.

PART 3

Mapping Arrays

Set up

Here's the starting point for the playground page. These are the functions from the last section along with an array of values to play with.

```
func myMap<Input, Output>(_ input: Input,
                          _ transform: (Input) -> Output) -> Output {
    return f(input)
}

func pay15for(_ hours: Hours) -> Euros {
    return hours * 15.euros.perHour
}

let hoursForTheWeek = [3.5.hours, 10.hours,
                       7.hours, 12.hours,
                       4.6.hours]
```

Applying myMap to an array

What needs to change if we operate on an array instead of a single value.

```
func myMap<Input, Output>(_ input: [Input],
                          _ transform: (Input) -> Output) -> [Output] {
    var output = [Output]()
    for element in input {
        output.append(f(element))
    }
    return output
}
```

Apply the function to our array.

```
let result1 = myMap(hoursForTheWeek){pay15for($0)}
result1.description
```

Arrays should be able to change themselves

We want to clean up this call so that an array can call `myMap()`. We use an extension. Note we don't need `Input` anymore.

```
extension Array {
    func myMap<Output>(_ transform: (Element) -> Output) -> [Output] {
        var output = [Output]()
        for element in self {
            output.append(f(element))
        }
        return output
    }
}
```

Now we can call it like this.

```
let result2 = hoursForTheWeek.myMap{pay15for($0)}
result2.description
```

Move myMap to Sequence

Actually, let's move `myMap()` up into the `Sequence` protocol.

The only thing that changes is the name of the type.

```
extension Sequence {
    func myMap<Output>(_ transform: (Element) -> Output) -> [Output] {
        var output = [Output]()
        for element in self {
            output.append(f(element))
        }
        return output
    }
}
```

In Swift 3 we would have had to specify that we are using `Sequence`'s `Iterator`'s `Element`. Swift 4 knows that that's what we mean by `Element`.

The map() function

`myMap()` is essentially Swift's `map()` function. The `map()` function also `rethrows`. We aren't taking care of that detail.

```
let result3 = hoursForTheWeek.map{pay15for($0)}
result3
```

Custom Operator

We had a custom operator for `myMap()`. We can't use the <$> operator that is popular in other languages. You'll often see <^> for map.

```
infix operator <^>: Compose

public func <^> <Input, Output>(xs: [Input], f: (Input) -> Output) -> [Output] {
    return xs.map(f)
}
```

We can now use it like this.

```
let result4 = hoursForTheWeek <^> pay15for
result4.description
```

PART 4

Mapping Dictionaries (optional)

Set up

Here's our method, a simple enum representing weekdays, and a simple dictionary with `Weekdays` as keys and `Hours` as values.

```
func pay15for(_ hours: Hours) -> Euros {
    return hours * 15.euros.perHour
}

enum Weekdays: String, CustomStringConvertible {
    case mon, tue, wed, thu, fri
    var description: String {
        return rawValue
    }
    var firstLetter: Character {
        return self.rawValue.first!
    }
}
```

```
let hoursForTheWeek = [Weekdays.mon: 3.5.hours,
                       .tue: 10.hours, .wed: 7.hours,
                       .thu: 12.hours, .fri: 4.6.hours]
```

Sorting dictionaries

We can sort the dictionary like this.

```
let worstToFirst = hoursForTheWeek.sorted{(entry1, entry2) in
    entry1.value < $1.entry1.value}
worstToFirst.description
```

```
"[(key: mon, value: 3.5), (key: fri, value: 4.599999999999996),
(key: wed, value: 7.0), (key: tue, value: 10.0), (key: thu, value: 12.0)]"
```

Note I've added conformance to `Hours` for `Comparable`.

```
extension Hours: Comparable {
    public static func <(hours1: Hours, hours2: Hours) -> Bool {
        return hours1.value < hours2.value
    }
}
```

`sorted()` is just a demo of another higher-order function in the Swift Standard Library.

Grouping

New in Swift 4, we can group elements in key-value pairs. For instance let's group the dictionary keys based on the first letter in their key's `rawValue`.

```
let alphabeticalDays
    = Dictionary(grouping: hoursForTheWeek.keys){day in
        day.firstLetter
}
alphabeticalDays.description
```

```
"["w": [wed], "m": [mon], "f": [fri], "t": [tue, thu]]"
```

Map returns an Array

No matter what type of `Sequence` you apply `map()` to, the result is an array.

```
let result1 = hoursForTheWeek.map{(day, hours) in
    pay15for(hours)
}
result1.description

"[€ 69.00, € 105.00, € 52.50, € 150.00, € 180.00]"
```

Tuples?

You can preserve the key value pairs using tuples.

```
let result2 = hoursForTheWeek.map{(day, hours) in
    (day, pay15for(hours))
}
result2.description

"[(fri, € 69.00), (wed, € 105.00), ( mon, € 52.50),
(tue, € 150.00), (thu, € 180.00)]"
```

MapValue

New in Swift 4 you can do what you often want to in a Dictionary - just map the values. Note that

`$0` refers to the value.

```
let result3 = hoursForTheWeek.mapValues{hours in
    pay15for(hours)
}
result3.description

"[fri: € 69.00, wed: € 105.00, mon: € 52.50,
tue: € 150.00, thu: € 180.00]"
```

PART 5

Mapping Optionals

Set up

Let's start with some of the same elements as when we were mapping Dictionaries - except let's not have hours scheduled for every weekday in our dictionary.

```
func pay15for(_ hours: Hours) -> Euros {
    return hours * 15.euros.perHour
}

enum Weekdays: String, CustomStringConvertible {
    case mon, tue, wed, thu, fri
    var description: String {
        return rawValue
    }
}

let hoursForTheWeek = [Weekdays.mon: 3.5.hours,
                       .tue: 10.hours, .thu: 12.hours]
```

Calculate earnings

We're going to calculate earnings for a specific day. If there is no entry for that day we'll return `nil`.

```
func earningsFor(_ day: Weekdays) -> Euros? {
    guard let hours = hoursForTheWeek[day] else {return nil}
    return pay15for(hours)
}
```

Call it for valid days and invalid days. `.Mon` will give us a wrapped value of 52,50 `Euros` while `.Wed` gives us `nil`.

```
earningsFor(.mon)
earningsFor(.wed)
```

Building map()

We can create our own map from one optional type to another. If the input is `nil` then the `map()` results in `nil`. If the input is not `nil` then `map()` results in a wrapped value of `f(input)`. We'll call our `map()` `myMap()` for now.

```
extension Optional {
    func myMap<Output>(_ transform: (Wrapped) -> Output) -> Output? {
        switch self {
        case .none:
            return .none
        case .some(let value):
            return .some(transform(value))
        }
    }
}
```

We use it like this in `earningsFor()`.

```
func earningsFor(_ day: Weekdays) -> Euros? {
    return hoursForTheWeek[day].myMap{pay15for($0)}
}
```

Optional map()

Of course `myMap()` is just our version of the `map()` function for Optionals from the Swift Standard Library that takes a function `f`.

```
func earningsFor(_ day: Weekdays) -> Euros? {
    return hoursForTheWeek[day].map{pay15for($0)}
}
```

Nil Coalescing Operator

We may prefer to not get an optional returned. We can instead use `map()` together with `??` to get 0 `Euros` if no work was done on a given day.

```
func earningsFor(_ day: Weekdays) -> Euros {
    return hoursForTheWeek[day].map{pay15for($0)} ?? Euros(0)
}
```

This time the result is `€ 52.50` and `€ 0.00`.

Flip the order

We can use the nil coalescing operator inside the call to the function.

```
func earningsFor(_ day: Weekdays) -> Euros {
    return pay15for(hoursForTheWeek[day, default: 0.hours])
}
```

Default values

This has nothing to do with `map()` but new in Swift 4 we have default values when looking up entries in dictionaries. We can use this concept like this.

```
func earningsFor(_ day: Weekdays) -> Euros {
    return pay15for(hoursForTheWeek[day, default: 0.hours])
}
```

In the next section we create our own map.

PART 6

Result Type

Set up

Let's start with our dictionary, our enum, our `pay15for()` function and our higher-order `pay()` function.

```
func pay15for(_ hours: Hours) -> Euros {
    return hours * 15.euros.perHour
}
func pay(at rateInEuros: Euros.Rate) -> (Hours) -> Euros {
    return {hours in rateInEuros * hours}
}
enum Weekdays: String, CustomStringConvertible {
    case mon, tue, wed, thu, fri
    var description: String {
        return rawValue
    }
    var firstLetter: Character {
        return self.rawValue.first!
    }
}
let hoursForTheWeek = [Weekdays.mon: 3.5.hours,
                       .tue: 10.hours, .thu: 12.hours]
```

ResultLite type

Before Swift 5 now has a Result type so we'll create a light version.

```
enum ResultLite<Value> {
    case failure(String)
    case success(Value)
}
```

Using Result

We can use `ResultLite` instead of just `Optional` but we do get an additional layer.

```
func hoursFor(_ day: Weekdays) -> ResultLite<Hours> {
    guard let hours
        = hoursForTheWeek[day]
        else {return Result.failure("Not scheduled on \(day).")}
    return ResultLite.success(hours)
}

hoursFor(.mon)
hoursFor(.wed)

success(3.5 hours)
failure("Not scheduled on Wed.")
```

Using a ResultLite type

Sometimes we want to use the result of a function that returns a `ResultLite`. Here's one way.

```
func earningsFor(_ hours: ResultLite<Hours>) -> ResultLite<Euros> {
    switch hours {
    case .failure(let errorMessage):
        return ResultLite.failure(errorMessage)
    case .success(let value):
        return ResultLite.success(pay15for(value))
    }
}


earningsFor(hoursFor(.Mon))
earningsFor(hoursFor(.Wed))

success(€ 52.50)
failure("Not scheduled on Wed.")
```

Introduce map

This becomes nicer if we introduce `map()`. Note we just need a function from the `Value` type in one to the `Value` type in the other.

```
extension ResultLite {
    func map<TargetValue>(_ transform: (Value) -> TargetValue) -> ResultLite<TargetV
alue> {
        switch self {
        case .failure(let errorMessage):
            return ResultLite<TargetValue>.failure(errorMessage)
        case .success(let value):
            return ResultLite<TargetValue>.success(transforms(value))
        }
    }
}
```

Using map

We can now use this function to simplify our calls.

```
let monPay = hoursFor(.Mon).map{hours in pay15for(hours)}
monPay
let wedPay = hoursFor(.Wed).map{pay15for($0)}
wedPay
let thursPay = hoursFor(.Thu).map(pay15for)
thursPay

success(€ 52.50)
failure("Not scheduled on Wed.")
success(€180.00)
```

Custom Operator

In *Sources* > *CustomOperators.swift* we've declared this operator which we'll use for map.

```
infix operator <^>: Apply //map
```

We can implement our operator to perform map in many ways. Here's one.

```
extension ResultLite {
    static func <^> <TargetValue>(v: ResultLite,
                                  transform: (Value) -> TargetValue) -> ResultLite<Targe
tValue> {
        return v.map(transform)
    }
}
```

We can use our operator like this.

```
Weekdays.mon |> hoursFor <^> pay15for
```

We can also remove the hard-coding of our rate by changing our call to

```
Weekdays.mon |> hoursFor ^lt;^> pay(at: 15.euros.perHour)
```

or

```
Weekdays.mon |> hoursFor <^> (15.euros.perHour |> pay)
```

PART 7

FlatMapping Sequences

Set up

Here's the starting point for the playground page. There's the familiar `Weekdays` enum, four arrays of `Weekdays` corresponding to days worked by different employees. Finally, there's a dictionary of employees and their days worked.

```
enum Weekdays: String, CustomStringConvertible {
    case mon, tue, wed, thu, fri
    var description: String {
        return rawValue
    }
}

let joansDays = [Weekdays.mon, .tue, .fri]
let davesDays = [Weekdays.tue, .wed, .fri]
let marysDays = [Weekdays.mon, .wed, .fri]
let fredsDays = [Weekdays.mon, .tue, .wed]

let schedule = ["Joan": joansDays, "Dave": davesDays,
                "Mary": marysDays, "Fred": fredsDays]
```

map() the Dictionary

We can easily get an array of total days worked by using

`map()` to build an array of each person's days worked.

```
let scheduleMap = schedule.map{ (key, days) in
    days
}
```

If you prefer, we can use the `$0` notation like this.

```
let scheduleMap = schedule.map{$0.value}
```

The result

Unfortunately, we get an array of arrays.

```
[[tue, wed, fri], [mon, wed, fri],
[mon, tue, wed], [mon, tue, fri]]
```

We don't want an array of arrays, we want an array of elements.

Building flatMap()

The issue with map is that we build up our resulting array like this.

```
ouput.append(f(element))
```

In the case that

`f(element)` is of type `[Output]`, our resulting array must have the type `[[Output]]`. Instead we have to append the contents of the created inner array like this.

```
output.append(contentsOf: f(element))
```

Given that, here's our implementation of `flatMap()` for sequences.

```
extension Sequence {
    func changed<Output>(by f: (Element) -> [Output]) -> [Output] {
        var output = [Output]()
        for element in self {
            output.append(contentsOf: f(element))
        }
        return output
    }
}
```

Using flatMap()

Let's use our new function.

```
let scheduleChanged = schedule.changed{$0.value}
```

Check out the difference in the results.

```
[tue, wed, fri, mon, wed, fri, mon, tue, wed, mon, tue, fri]
```

This function, is, of course `flatMap()` so let's use `flatMap()` instead.

```
let scheduleFlatMap = schedule.flatMap{$0.value}
```

```
[tue, wed, fri, mon, wed, fri, mon, tue, wed, mon, tue, fri]
```

What you get

Now that you have a flattened `Array` you can apply another function to it.

We can also find unique days by creating a `Set` from it.

```
let coveredDays = Array(Set(scheduleFlatMap))
```

Now we can easily see that no one is working on Thursday.

```
[wed, fri, mon, tue]
```

PART 8

FlatMapping Optionals

Set up

Here's the starting point for the playground page. This is the same setup we used for the preceding section.

```
enum Weekdays: String, CustomStringConvertible {
    case mon, tue, wed, thu, fri
    var description: String {
        return rawValue
    }
}

let joansDays = [Weekdays.mon, .tue, .fri]
let davesDays = [Weekdays.tue, .wed, .fri]
let marysDays = [Weekdays.mon, .wed, .fri]
let fredsDays = [Weekdays.mon, .tue, .wed]

let schedule = ["Joan": joansDays, "Dave": davesDays,
                "Mary": marysDays, "Fred": fredsDays]
```

An optional

Read an entry from the `schedule` Dictionary. This will be an optional.

```
let joan = schedule["Joan"]
```

`joan` is an Optional Array. It is just `joansDays` wrapped in an Optional.

Map

Suppose we want to find the first element of the array represented by `joan` We can't call this because `joan` is an Optional.

```
joan.first
```

We could use Optional chaining - but let's use `map()`. Remember, `map()` takes one Optional to another.

```
let joansFirstMap = joan.map{$0.first}
```

The result

Unfortunately, we get an Optional Optional.

This is difficult to see in the playground unless we print the value.

```
joansFirstMap
                                                                mon
print(joansFirstMap)
                                                "Optional(Optional(mon))\n"
```

We don't want an optional optional, we want an optional. Again, we could use Optional Chaining, but we're going to use flatMap() which is what underlies optional chaining.

Building flatMap()

The issue with map for optionalsis that we return from our .some case like this.

```
return .some(f(value))
```

In the case that `f(value)` is of type `Output?`, placing it in the `.some` case gives us `Output??`. Instead we have to return `f(value)` itself.

Here's our implementation of `flatMap` for Optionals.

```
extension Optional {
    func changed<Output>(_ f: (Wrapped) -> Output?) -> Output? {
        switch self {
        case .none:
            return .none
        case .some(let value):
            return f(value)
        }
    }
}
```

Using flatMap()

Let's use our new function.

```
let joansFirstChanged = joan.changed{$0.first}
```

Check out the difference in the results.

```
"Optional(mon)\n"
```

This function, is, of course `flatMap()` so let's use `flatMap()` instead.

```
let joansFirstFlatMap = joan.flatMap{$0.first}
```

```
"Optional(mon)\n"
```

Optional Chaining

Try optional chaining and your results will be the same.

```
let joansFirstChained = joan?.first
```

Now we can easily see that no one is working on Thursday.

```
"Optional(mon)\n"
```

A longer chain

Introduce an array and a silly method on our particular dictionary type.

```
let team = ["Joan", "Mike", "Dave", "Anna"]

extension String {
    func hasValueIn(_ dictionary: [String: [Weekdays]]) -> [Weekdays]? {
        return dictionary[self]
    }
}
```

Walk this longer chain with optional chaining.

```
let teamsFirstChained  = team.first?.hasValueIn(schedule)?.first
```

Do the same with `flatMap()`

```
let teamsFirstFlatMap = team.first
    .flatMap{name in name.hasValueIn(schedule)}
    .flatMap{days in days.first}
```

PART 9

FlatMapping Sequences of Optionals

Note: this name will change.

Set up

We need some items to play with.

```
enum Weekdays: String, CustomStringConvertible {
    case mon, tue, wed, thu, fri
    var description: String {
        return rawValue
    }
}

let joansDays = [Weekdays.mon, .tue, .fri]
let davesDays = [Weekdays.tue, .wed, .fri]
let marysDays = [Weekdays.mon, .wed, .fri]
let fredsDays = [Weekdays.mon, .tue, .wed]

let schedule = ["Joan": joansDays, "Dave": davesDays,
                "Mary": marysDays, "Fred": fredsDays]

let team = ["Joan", "Mike", "Dave", "Anna"]
```

Not really a Flat Map

The first flat map was used when we were mapping a sequence using a function whose target was an array to avoid ending up with an array of arrays.

The second flat map was used when we were mapping an optional using a function whose target was an optional to avoid ending up with an optional optional.

This third function used to be called flatMap but it doesn't fit the pattern. It is used when we are mapping a sequence using a function whose target is an optional to avoid ending up with an array of optionals. This compactMap eliminates nils and unwraps optionals.

The Problem

The `team` includes two people whose names are keys in `schedule` and two who aren't. So when we use map to find the days they worked

```
let daysForTeamMap = team.map{name in schedule[name]}
```

The result is an array of optionals.

```
[Optional([mon, tue, fri]), nil, Optional([tue, wed, fri]), nil]
```

Building CompactMap

This is what `map()` looked like.

```
func changed<Input, Output>(_ input: [Input],
                            using f: (Input) -> Output) -> [Output] {
    var output = [Output]()
    for element in input {
        output.append(f(element))
    }
    return output
}
```

The key is that`compactMap()` doesn't rewrap the values. If `f: (Input) -> Output?` instead of `-> Output` then we have to unwrap the optional.

We need to replace

```
output.append(f(element))
```

with

```
if let newElement = f(element) {
    output.append(newElement)
}
```

If `f(element)` is `nil` we ignore it. If it is not `nil`, then we unwrap it and append it. Add this code for `changed` to the playground.

```
extension Sequence {
    func changed<Output>(_ f: (Element) -> Output?) -> [Output] {
        var output = [Output]()
        for element in self {
            if let newElement = f(element) {
                output.append(newElement)
            }
        }
        return output
    }
}
```

```
let daysForTeamChanged = team.changed{name in schedule[name]}
```

```
[[Mon, Tue, Fri], [Tue, Wed, Fri]]
```

This is compact map. So we can use `compactMap()` instead like this.

```
let daysForTeamCompactMap = team.compactMap{name in schedule[name]}
```

```
[[mon, tue, fri], [tue, wed, fri]]
```

Compact Map

We can replace compactMap with map followed by filtering out the nils and mapping to force unwrap the rest.

```
let daysForTeamCompactMap2
    = team.map{name in schedule[name]}
        .filter{days in days != nil}
        .map{value in value!}
```

Combining with flat map

Notice the results of that `compactMap()` is an array of arrays. We can use `flatMap()` on this result to further flatten it.

```
let daysForTeam = team.compactMap{name in schedule[name]}.flatMap{$0}
```

```
[mon, tue, fri, tue, wed, fri]
print(Set(daysForTeam))
```

PART 10

Our Own FlatMap

Set up

Here's the initial state of our playground.

```
func pay(at rateInEuros: Euros.Rate) -> (Hours) -> Euros {
    return {hours in rateInEuros * hours}
}
enum Weekdays: String, CustomStringConvertible {
    case mon, tue, wed, thu, fri
    var description: String {
        return rawValue
    }
}

enum ResultLite<Value> {
    case failure(Error)
    case success(Value)
}
extension ResultLite: CustomStringConvertible {
    var description: String {
        switch self {
        case .failure(let errorMessage):
            return "(Error: \(errorMessage))"
        case .success(let value):
            return "(Success: \(value))"
        }
    }
}

extension ResultLite {
    func map<TargetValue>>(_ transform: (Value) -> TargetValue) -> ResultLite<Target
Value> {
    switch self {
    case .failure(let errorMessage):
    return ResultLite<TargetValue>.failure(errorMessage)
    case .success(let value):
    return ResultLite<TargetValue>.success(transform(value))
    }
    }
}

let hoursForTheWeek = [Weekdays.mon: 3.5.hours,
                       .tue: 10.hours, .thu: 12.hours,
                       .fri: 0.hours]
```

From Day to ResultLite<Hours>

Here's a silly function that reads from the `hoursForTheWeek` dictionary and returns a `ResultLite<Hours>`.

```
func hoursWorkedOn(_ day: Weekdays) -> ResultLite<Hours> {
    guard let hours = hoursForTheWeek[day] else  {
        return ResultLite.failure("Didn't work on \(day)")
    }
    return ResultLite.success(hours)
}
```

Here we use it.

```
let monHours = hoursWorkedOn(.mon)
let wedHours = hoursWorkedOn(.wed)
let friHours = hoursWorkedOn(.fri)

(Success: 3.5 hours)
(Error: Didn't work on Wed)
(Success: 0.0 hours)
```

map() of the result type

Here's a function from `Hours` to `ResultLite<Euros>`.

```
func pay15For(_ hours: Hours) -> ResultLite<Euros> {
    guard hours > 0 else {
        return ResultLite.failure("No hours")
    }
    return ResultLite.success(hours * 15.euros.perHour)
}
```

Use `map()` and you will see we've double wrapped our result.

```
let wrongMonPay = monHours.map(pay15For)
let wrongWedPay = wedHours.map(pay15For)
let wrongFriPay = friHours.map(pay15For)

(Success: (Success: €52.50))
(Error: Didn't work on Wed)
(Success: (Error: No hours))
```

FlatMap for ResultLite

Our flatmap follows this same shape as the other flat maps.

```
extension ResultLite {
    func map<TargetValue>(_ transform: (Value) -> TargetValue) -> ResultLite<TargetV
alue> {
        switch self {
        case .failure(let errorMessage):
            return ResultLite<TargetValue>.failure(errorMessage)
        case .success(let value):
            return ResultLite<TargetValue>.success(transform(value))
        }
    }
```

```
    func flatMap<TargetValue>(_ transform: (Value) -> ResultLite<TargetValue>) -> Re
sultLite<TargetValue> {
        switch self {
        case .failure(let errorMessage):
            return ResultLite<TargetValue>.failure(errorMessage)
        case .success(let value):
            return transform(value)
        }
    }
}
```

Using flatMap

Use flatMap instead.

```
let monPay = monHours.flatMap(pay15For)
let wedPay = wedHours.flatMap(pay15For)
let friPay = friHours.flatMap(pay15For)

(Success: €52.50)
(Error: Didn't work on Wed)
(Error: No hours)
```