Functional Swift

Daniel H Steinberg

Paris

December 2018

http://dimsumthinking.com

Functional Swift

Daniel H Steinberg

Paris

December 2018

http://dimsumthinking.com

XCODE 9 - SWIFT 4 : JANUARY 2018

Functional Swift

map, filter, reduce, and all that

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

PART 1

Partial Application

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 {
    return hours * 15.euros.perHour
}
```

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.

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

payAt12(for: 3.5.hours)
```

That's horrible.

No more currying syntax

We'd like something like this.

```swift
// This doesn't work
func pay(at rateInEuros: Euros.Rate)(for hours: Hours) -> Euros {
    return rateInEuros * hours
}
```

We'd then call it like this.

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

That's not allowed either.

Both were legal in Swift but have been removed.

Solution 1 - Multiple Parameters

The first solution is to replace

```
// This doesn't work
func pay(at rateInEuros: Euros.Rate)(for hours: Hours) -> Euros {
    return rateInEuros * hours
}
```

with

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

In other words, get rid of the separate parentheses enclosing each parameter and create a single parameter list with comma separated parameters.

Instead of calling it like this:

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

Call it like this:

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

But we lose something big

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

```
// can't do this anymore
let payAgainAt15 = 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.

```
// not quite right
func pay(at rateInEuros: Euros.Rate) -> (for hours: Hours) -> Euros {
```

Unfortunately, we've lost the ability to include external labels on the function being returned.

We can either use no label at all or we can use an internal label only.

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

or (we'll use this one with no label at all)

```
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 {
        return rateInEuros * hours
    }
    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
        return rateInEuros * hours
    }
    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)

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

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
}
```

Higher-Order 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)
```

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

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

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 changed<Input, Output>(_ input: Input,
                            using f: (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 changed to an array

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

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

Apply the function to our array.

```
let result1 = changed(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 `changed()`. We use an extension. Note we don't need `Input` anymore.

```
extension Array {
    func changed<Output>(using f: (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.changed{pay15for($0)}
result2.description
```

Move changed to Sequence

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

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

```
extension Sequence {
    func changed<Output>(using f: (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

`changed()` 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

`changed()`. 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.

```
hoursForTheWeek <^> pay15for
```

If the `Input` is `CustomStringConvertible` we could also display this more nicely

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

hoursForTheWeek
    <^> pay15for
    <^> {x in x.description}
```

I'd rather not. I'd rather not restrict `Input`.

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

let result4 = hoursForTheWeek |> pay15for
result4.description
```

PART 4

Mapping Dictionaries

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
    }
}

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{$0.value.value < $1.value.value}
worstToFirst.description
```

```
"[(key: Mon, value: 3.5), (key: Fri, value: 4.5999999999999996),
(key: Wed, value: 7.0), (key: Tue, value: 10.0), (key: Thu, value:
12.0)]"
```

This is terrible. Let's add conformance to `Hours` for `Comparable`.

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

This is nicer.

```
let worstToFirst = hoursForTheWeek.sorted{$0.value.value < $1.value.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.rawValue.first!
}
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 Model

Set up

Here again is 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
    }
}

let hoursForTheWeek = [Weekdays.Mon: 3.5.hours,
                       .Tue: 10.hours, .Wed: 7.hours,
                       .Thu: 12.hours, .Fri: 4.6.hours]
```

A simple type

We used to use dictionaries a lot when writing Cocoa apps in Objective-C. Then Objective-C 2 introduced properties. In some ways properties were key value pairs with the keys being their names and the values being their values.

So let's create a simple type that represents an element of the

`Dictionary<Weekdays: Hours>`.

```
struct WorkRecord {
    let day: Weekdays
    let hours: Hours
}
```

We create an instance like this.

```
let exampleRecord = WorkRecord(day: .Mon, hours: 3.5.hours)
```

It might help to have a description.

```
extension WorkRecord: CustomStringConvertible {
    var description: String {
        return "\(day): \(hours)"
    }
}
```

Model

We can use our `Model` type to create a collection of `WorkRecord`s.

```
let records = Model(WorkRecord(day: .Mon, hours: 3.5.hours),
                    WorkRecord(day: .Tue, hours: 10.hours),
                    WorkRecord(day: .Wed, hours: 7.hours),
                    WorkRecord(day: .Thu, hours: 12.hours),
                    WorkRecord(day: .Fri, hours: 4.6.hours))
```

Sorting

Because

`Model` is a `Sequence` we now get some higher order functions for free. For example, sorting.

```
let alphabeticalDays = records.sorted{(record1, record2) in
    record1.day.rawValue < record2.day.rawValue
    }
alphabeticalDays.description
```

The typing makes the calls clearer. Instead of .key and .value it's .day and .hours.

```
let worstToFirst = records.sorted{(record1, record2) in
    record1.hours < record2.hours}
worstToFirst.description
```

Map

As before, no matter what type of `Sequence` you apply `map()` to, the result is an array. But we do get map for free.

```
let justHours = records.map{record
    in record.hours
}
```

We can also chain operations.

```
let orderedHours
    = records
        .sorted{(record1, record2) in
                 record1.hours < record2.hours
        }
        .map{record in record.hours}

let orderedHours2 = records
    .sorted{$0.hours < $1.hours}
    .map{record in record.hours}
```

More types

Create another struct.

```
struct PayRecord {
    let day: Weekdays
    let euros: Euros
}

extension PayRecord: CustomStringConvertible {
    var description: String {
        return "\(day): \(euros)"
    }
}
```

Use map to produce pay per day.

```
let payRecords = records.map{record in
    PayRecord(day: record.day, euros: pay15for(record.hours))
}

payRecords.description
```

PART 6

Filter

Set up

We start this playground with examples of an array and a dictionary.

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

enum Weekdays: String, CustomStringConvertible {
    case Mon, Tue, Wed, Thu, Fri
    var description: String {
        return rawValue
    }
}

let hoursForTheWeekdays = [Weekdays.Mon: 3.5.hours,
                           .Tue: 10.hours, .Wed: 7.hours,
                           .Thu: 12.hours, .Fri: 4.6.hours]
```

How filter works

We'll implement a version of `filter()` named `keepIfSatisfies()` in `Sequence`. It needs a function that takes elements of the same type as are in the `Sequence` and returns a `Bool`. If the return value is `true` we keep the element in the `Sequence` if it's `false` we remove it.

```
extension Sequence {
    func keepIfSatisfies(_ condition: (Element) -> Bool) -> [Element] {
        var output = [Element]()
        for element in self {
            if condition(element) {
                output.append(element)
            }
        }
        return output
    }
}
```

Use it

Let's filter all of the elements of the array that are `8.hours` or larger.

```
let under8
    = hoursForTheWeek.keepIfSatisfies{hours in
        hours < 8.hours
    }
under8.description
```

```
"[3.5, 7.0, 4.5999999999999996]"
```

Actually, let's use the built-in `filter()` method.

```
let under8a = hoursForTheWeek.filter{hours in
                  hours < 8.hours
              }
under8a.description
```

Filtering Dictionaries

As of Swift 4, when we filter a dictionary we get back a dictionary of the same type. We can filter on the keys.

```
let daysBeginningWithT
    = hoursForTheWeekdays
        .filter{(key, _) in
             key.description.first! == "T"
         }
daysBeginningWithT.description
```

```
"[Thu: 12.0 hours, Tue: 10.0 hours]"
```

Or the values.

```
let under8b = hoursForTheWeekdays.filter{$0.value < 8.hours}
under8b.description
```

```
"[Wed: 7.0, Fri: 4.5999999999999996, Mon: 3.5]"
```

Reduce

Set up

We start our playground with an array and two useful functions.

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

func adjustedHoursForOvertime(_ hours: Hours) -> Hours {
    guard hours > 8.hours else {return hours}
    return hours + (hours - 8.hours) * 0.5
}

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

Like induction

`reduce()` is like proof by induction. We need two things: an initial value, and a way of getting from the current value of the accumulator (which starts out with the initial value) to the next value using the next element of our sequence.

Here's how we might implement it as `transform()`.

```
extension Sequence {
    func transform<Output>(_ initialValue: Output,
                             using f: (Output, Element) -> Output)
                                                              -> Output {
        var accumulator = initialValue
        for element in self {
            accumulator = f(accumulator, element)
        }
        return accumulator
    }
}
```

Note that we know nothing about `Output`. It might even be a collection.

transform is reduce

Here we use transform to total our hours.

```
let totalHours
    = hoursForTheWeek
        .transform(0.hours){(runningTotal, hoursForTheDay) in
            runningTotal + hoursForTheDay
         }
totalHours

let totalHoursa = hoursForTheWeek.transform(0.hours){ $0 + $1}
totalHoursa
```

```
37.1
```

We can use the built-in `reduce()` function instead.

```
let totalHours1
    = hoursForTheWeek
        .reduce(0.hours){(runningTotal, hoursForTheDay) in
            runningTotal + hoursForTheDay
         }
totalHours1

let totalHours1a = hoursForTheWeek.reduce(0.hours){ $0 + $1}
totalHours1a
```

Different Types

We can use `totalHours` to calculate `totalPay` like this.

```
let totalPay = pay15for(totalHours1)
```

```
€ 556.50
```

Or, we can use `pay15For()` in `reduce()`. Note how that changes the types of the parameters.

```
let totalPay1
    = hoursForTheWeek
        .reduce(0.euros){(runningTotal, hoursForTheDay) in
            runningTotal + pay15for(hoursForTheDay)
}
totalPay1
```

There's flexibility here as we can calculate overtime pay for each day.

```
let payWithOvertime
    = hoursForTheWeek
        .reduce(0.euros){(runningTotal, hoursForTheDay) in
            runningTotal
                + pay15for(adjustedHoursForOvertime(hoursForTheDay))}
payWithOvertime
```

```
€ 601.50
```

Map

We can (but shouldn't) implement `map()` with `reduce()`. We can (but shouldn't) replace `map()` with `reduce()`.

```
let dailyPay
    = hoursForTheWeek
        .reduce([Euros]()){(outputArray, hoursForTheDay) in
            outputArray + [pay15for(hoursForTheDay)]}
dailyPay.description
```

We can implement map with reduce this way.

```
extension Sequence {
    func mapWithReduce<Output>(_ f: (Element) -> Output) -> [Output] {
        return reduce([Output]()){(outputArray, element) in
            outputArray + [f(element)]
        }
    }
}
```

Use this mapWithReduce.

```
let dailyPay2 = hoursForTheWeek.mapWithReduce(pay15for)
```

Filter

We can also (but also shouldn't) implement `filter()` with `reduce()`. We can (but shouldn't) replace `filter()` with `reduce()`.

```
let under8
    = hoursForTheWeek
        .reduce([Hours]()){guard $1 < 8 else {return $0}
                                               return $0 + [$1]}
under8.description

extension Sequence {
    func filterWithReduce(_ f: (Element) -> Bool) -> [Element] {
        return reduce([Element]()){(outputArray, element) in
            guard f(element) else {return outputArray}
            return outputArray + [element]
        }
    }
}
let under8a
    = hoursForTheWeek
        .filterWithReduce{hours in hours < 8.hours}
```

PART 8

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()` `changed()` for now.

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

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

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

Optional map()

Of course `changed()` 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 9

Our Own Map

Set up

Let's start with some of the same elements as when we were mapping Model.

```
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
    }
}

struct WorkRecord {
    let day: Weekdays
    let hours: Hours
}

extension WorkRecord: CustomStringConvertible {
    var description: String {
        return "\(day): \(hours)"
    }
}

struct PayRecord {
    let day: Weekdays
    let euros: Euros
}

extension PayRecord: CustomStringConvertible {
    var description: String {
        return "\(day): \(euros)"
    }
}

let workRecord = WorkRecord(day: .Mon, hours: 3.5.hours)

let records = Model(WorkRecord(day: .Mon, hours: 3.5.hours),
                    WorkRecord(day: .Tue, hours: 10.hours),
                    WorkRecord(day: .Wed, hours: 7.hours),
                    WorkRecord(day: .Thu, hours: 12.hours),
                    WorkRecord(day: .Fri, hours: 4.6.hours))
```

A generic type

`WorkRecord` and `PayRecord` look similar. Let's capture that similarity in a new type.

```
struct DailyRecord<Value> {
    let day: Weekdays
    let value: Value
}

extension DailyRecord: CustomStringConvertible {
    var description: String {
        return "\(day): \(value)"
    }
}
```

Using the Generic Type

For the most part we can think of our two specific types like this.

```
struct DailyRecord<Value> {
    let day: Weekdays
    let value: Value
}

extension DailyRecord: CustomStringConvertible {
    var description: String {
        return "\(day): \(value)"
    }
}
```

```
typealias WorkRecord = DailyRecord<Hours>
typealias PayRecord = DailyRecord<Euros>
```

We lose the property labels `hours` and `euros` but we gain the common structure.

```
let workRecord = WorkRecord(day: .Mon, value: 3.5.hours)

let records = Model(WorkRecord(day: .Mon, value: 3.5.hours),
                    WorkRecord(day: .Tue, value: 10.hours),
                    WorkRecord(day: .Wed, value: 7.hours),
                    WorkRecord(day: .Thu, value: 12.hours),
                    WorkRecord(day: .Fri, value: 4.6.hours))
```

Introducing map()

The goal of `map()` is to take a function `f` from `A -> B` and lift it to a function `mapf` from `DailyRecord<A> -> DailyRecord<B>`.

```
extension DailyRecord {
    func map<TargetValue>(_ f: (Value) -> TargetValue) -> DailyRecord<TargetVal
ue> {
        return DailyRecord<TargetValue>(day: day, value: f(value))
    }
}
```

Using map()

Our `map()` now works like this.

```
let workRecord = WorkRecord(day: .Mon, value: 3.5.hours)
let payRecord = workRecord.map(pay15for)
```

Comparing maps

When we had a `Model` filled with `WorkRecords` we used `Model`'s `map()`.

```
let payRecords = records.map{record in
    PayRecord(day: record.day, value: pay15for(record.value))
}
```

We can use the two maps together to simplify the code like this.

```
let payRecords2 = records.map{record in
    record.map(pay15for)
}
```

In the next section we see a solution using a `Result` type.

PART 10

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
    }
}
let hoursForTheWeek = [Weekdays.Mon: 3.5.hours,
                       .Tue: 10.hours, .Thu: 12.hours]
```

Result type

Before Swift had an `Error` type, many people created their own `Result` type. Now we can use the two types together like this.

```
enum Result<Value> {
    case error(Error)
    case success(Value)
}

extension String: Error {}
```

Using Result

We can use `Result` instead of just `Error` but we do get an additional layer.

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

hoursFor(.Mon)
hoursFor(.Wed)
```

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

Using a Result type

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

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

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

```
success(€ 52.50)
error("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 Result {
    func map<TargetValue>(_ f: (Value) -> TargetValue) -> Result<TargetValue> {
        switch self {
        case .error(let errorMessage):
            return Result<TargetValue>.error(errorMessage)
        case .success(let value):
            return Result<TargetValue>.success(f(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)
error("Not scheduled on Wed.")
success(€180.00)
```

Extension

We have two things hard-coded into our calculation: the rate of pay, and the name of the `Dictionary` containing the hours.

Let's fix both of these with an extension.

```
extension Dictionary where Key == Weekdays, Value == Hours  {
    func hoursFor(_ day: Weekdays) -> Result<Hours> {
        guard let hours = self[day]
            else {return Result.error("Not scheduled on \(day).")}
        return Result.success(hours)
    }

    func earningsFor(_ day: Weekdays,
                     at rate: Euros) -> Result<Euros> {
        return self.hoursFor(day).map(pay(at: rate))
    }
}
```

This gives us more flexibility and simplifies the calls.

```
hoursForTheWeek.earningsFor(.Mon, at: 10.euros.perHour)
hoursForTheWeek.earningsFor(.Wed, at: 10.euros.perHour)
```

```
success(€ 35.00)
error("Not scheduled on Wed.")
```

PART 11

Non-Container Map

Set up

Let's start with the mostly the same code we've been using for a couple of sections.

```
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]
```

Generic Function Container

This time our type will abstract functions from `Weekdays` to a generic target type. This example is different from `Array`, `Optional`, or `Result` as it doesn't contain an element of the generic type, it contains a function whose range is this type.

```
struct WeekdaysCalculation<Output> {
    let g: (Weekdays) -> Output
}
```

Using this type

Here's how we create an instance of this type.

```
let hours = WeekdaysCalculation<Hours>{day in
    hoursForTheWeek[day, default: 0]}
```

We can call it differently using a trailing closure - which looks a little different.

```
let hours2 = WeekdaysCalculation<Hours>{day in
    hoursForTheWeek[day, default: 0]}
```

The call to the function defined in `hours` looks a little different as well. Don't forget this is actually a call to `f`.

```
hours.g(.Mon)
```

```
3.5
```

```
hours.g(.Wed)
```

```
0
```

A second example

Suppose this time we want a function from `Weekdays` to `Euros` that calculates `pay`. We can either write this from scratch or we can use our existing `hours` function.

```
let pay = WeekdaysCalculation<Euros>{day in
    pay15for(hours.f(day))
}
```

Here's how we use it.

```
pay.g(.Mon)
```

```
€ 52,59
```

```
pay.g(.Wed)
```

```
€ 0.00
```

Introduce map

This becomes nicer if we introduce `map()`. In effect, `map()` is composition that makes the function diagram commute.

```
extension WeekdaysCalculation {
    func map<TargetOutput>(_ f: @escaping (Output) -> TargetOutput)
                                    -> WeekdaysCalculation<TargetOutput> {
        return WeekdaysCalculation<TargetOutput>{x in f(self.g(x))}
    }
}
```

Using map

We can now use this function to simplify our calls.

```
hours.map(pay15for).g(.Mon)
```

```
€ 52.50
```

```
hours.map(pay15for).g(.Wed)
```

€ 0.00

Here's another example where we map to `WeekdaysCalculation<String>`.

```
hours.map{hoursWorked in
    hoursWorked.value == 0 ? "Didn't work"
                           : "worked \(hoursWorked)"}
    .g(.Mon)
hours.map{hoursWorked in
    hoursWorked.value == 0 ? "Didn't work"
                           : "worked \(hoursWorked)"}
    .g(.Wed)
```

We could first assign the function then apply it.

```
let workedOrDidnt = hours.map{hoursWorked in
    hoursWorked.value == 0 ? "Didn't work"
        : "worked \(hoursWorked)"}

workedOrDidnt.g(.Mon)

workedOrDidnt.g(.Wed)
```

Free function

It might be useful to step back and see that `Weekdays ->` is a functor.

Create a type alias.

```
typealias AltWeekdaysCalculation<A> = (Weekdays) -> A
```

Create a free function that is a `map` for elements of this type.

```
func map<A,B>(_ g: @escaping AltWeekdaysCalculation<A>,
              _ f: @escaping (A) -> B) -> AltWeekdaysCalculation<B> {
    return {x in g(f(x))}
}
```

Use `map()` to create a function from `Weekdays` to `Euros`.

```
let amountPaidOn = map({day in
    hoursForTheWeek[day, default: 0.hours]},
                       pay15for)
```

Use it.

```
amountPaidOn(.Mon)
amountPaidOn(.Wed)
```

PART 12

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 13

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 14

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"]
```

A Third 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.

The third flat map 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 flat map 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{schedule[$0]}
```

The result is an array of optionals.

```
[Optional([Mon, Tue, Fri]), nil, Optional([Tue, Wed, Fri]), nil]
```

Building FlatMap

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`flatMap()` 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{schedule[$0]}
```

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

This is the third flat map. So we can use `flatMap()` instead like this.

```
let daysForTeamFlatMap = team.flatMap{schedule[$0]}
```

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

Filter Map

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

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

Because of this, there's discussion on the Swift Evolution list to rename this third flatMap to filterMap.

Combining flat maps

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

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

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

We can instead use `joined()`.

```
let daysJoined = daysForTeamFlatMap.joined()
```

This doesn't give us the type we want.

```
FlattenBidirectionalCollection<Array<Array<Weekdays>>>(_base: [[Mon, Tue, Fri], [Tue, Wed, Fri]])
```

If we create an Array from the result of `joined()` we get what we want.

```
let daysJoinedArray = Array(daysJoined)
```

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

PART 15

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

extension Result {
    func map<TargetValue>(_ f: (Value) -> TargetValue) -> Result<TargetValue> {
        switch self {
        case .error(let errorMessage):
            return Result<TargetValue>.error(errorMessage)
        case .success(let value):
            return Result<TargetValue>.success(f(value))
        }
    }
}
extension String: Error{}

let hoursForTheWeek = [Weekdays.Mon: 3.5.hours,
                       .Tue: 10.hours, .Thu: 12.hours,
                       .Fri: 0.hours]
```

From Day to Result<Hours>

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

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

```
func pay15For(_ hours: Hours) -> Result<Euros> {
    guard hours > 0 else {
        return Result.error("No hours")
    }
    return Result.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 Result

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

```
extension Result {
    func map<TargetValue>(_ f: (Value) -> TargetValue) -> Result<TargetValue> {
        switch self {
        case .error(let errorMessage):
            return Result<TargetValue>.error(errorMessage)
        case .success(let value):
            return Result<TargetValue>.success(f(value))
        }
    }
```

```
    func flatMap<TargetValue>(_ f: (Value) -> Result<TargetValue>) -> Result<Ta
rgetValue> {
        switch self {
        case .error(let errorMessage):
            return Result<TargetValue>.error(errorMessage)
        case .success(let value):
            return f(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)
```

PART 16

Pure

Set up

Here's the initial state of our playground.

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

let hoursDictionary = [Weekdays.Mon: 3.5.hours,
                       .Tue: 10.hours, .Thu: 12.hours]
```

Wrapping Arrays

The difference between `map()` and `flatMap()` was an extra layer of wrapping.

Here's a function that takes an element and wraps it in an array. It seems silly, but stick with me a minute.

```
extension Array {
    static func pure(_ a: Element) -> [Element] {
        return [a]
    }
}
```

The use of this operator feels odd.

```
let fiveHours = Array.pure(5.hours)
let fiveInts = Array.pure(5)
let fiveStrings = Array.pure("five")
```

We get `5.hours`, `5`, and `"five"` each wrapped in an array of their type.

You may prefer to implement it as an init.

```
extension Array {
    init(pure element: Element) {
        self = [element]
    }
}
```

You can use this as follows.

```
let fiveHoursAlt = Array(pure: 5.hours)
let fiveIntsAlt = Array(pure: 5)
let fiveStringsAlt = Array(pure: "five")
```

map() and flatMap()

`flatMap()` essentially was `map()` with the ability to unwrap a layer. We saw that if we had `map()` there were times when we also needed to create a `flatMap()`. We didn't get `flatMap()` for free. The opposite is not true. If we have `flatMap()` and `pure()` we always get `map()` for free.

```
extension Array {
    func myMap<Output>(_ f: (Element) -> Output) -> [Output] {
        return flatMap{a in Array<Output>(pure: f(a))}
    }
}
```

We can now use `myMap()` the same way we used `map()`.

```
let pay = hoursArray.myMap(pay15for)
```

Pure for Optionals

As you might suspect, we can do the same thing for Optionals.

```
extension Optional {
    static func pure(_ a: Wrapped) -> Wrapped? {
        return .some(a)
    }
}
```

We can use it like this.

```
let optionalFiveHours = Optional(6).pure(5.hours)
```

We can also use the init version.

```
extension Optional {
    init(pure element: Wrapped) {
        self = Optional.some(element)
    }
}
```

We use it like this.

```
let optionalFiveHoursAlt = Optional(pure: 5.hours)
```

Map for Optionals

As with Arrays, we can build a map from `flatMap()` and `pure()` for optionals.

```
extension Optional {
    func myMap<Output>(_ f: (Wrapped) -> Output) -> Output? {
        return flatMap{a in Optional<Output>.pure(f(a))}
    }
}
```

Using myMap

We use `myMap()` for optionals the same way we used `map()`.

```
let mondayPay = hoursDictionary[.Mon].myMap{pay15for($0)}
```

```
Optional(€ 52.50)
```

```
let wednesdayPay = hoursDictionary[.Wed].myMap{pay15for($0)}
```

```
nil
```

Result

As an exercise, create

`pure` and `myMap` for `Result`.

Solution:

```
extension Result {
    func map<TargetValue>(_ f: (Value) -> TargetValue) -> Result<TargetValue> {
        switch self {
        case .error(let errorMessage):
            return Result<TargetValue>.error(errorMessage)
        case .success(let value):
            return Result<TargetValue>.success(f(value))
        }
    }

    func flatMap<TargetValue>(_ f: (Value) -> Result<TargetValue>) -> Result<Ta
rgetValue> {
        switch self {
        case .error(let errorMessage):
            return Result<TargetValue>.error(errorMessage)
        case .success(let value):
            return f(value)
        }
    }

    static func pure(_ a: Value) -> Result<Value> {
        return Result.success(a)
    }
    func myMap<TargetValue>(_ f: (Value) -> TargetValue) -> Result<TargetValue>
 {
        return flatMap{ a in Result<TargetValue>.pure(f(a))}
    }
}
```