Learning Functional Reactive Programming

CollectionType

.filter

.reduce

.map

CollectionType.filter

```
[1, 2, 3, 4].filter { even($0) }
```

```
[2, 4]
```

CollectionType.reduce

```
[1, 2, 3, 4].reduce(0) { $0 + $1 }
```

```
10
```

CollectionType.map

```
[1, 2, 3, 4].map { ordinal($0) }
```

```
["1st", "2nd", "3rd", "4th"]
```

CollectionType.map

```
["First" : 1, "Second" : 2]
    .map { $0 + ":" + ordinal($1) }

["Second : 2nd", "First : 1st"]
```

CollectionType

.flatten

.flatMap

.split

CollectionType.flatten

```
[[1, 2], [3, 4]].flatten().map { ordinal($0) }

["1st", "2nd", "3rd", "4th"]
```

CollectionType.flatMap

```
[1, 2, 3, 4].flatMap { evenOrdinal($0) }
```

```
["2nd", "4th"]
```

CollectionType.flatMap

```
[1, 2, 3, 4]
    .map { evenOrdinal($0) }
    .filter { $0 != nil }
    .map { $0! }

["2nd", "4th"]
```

CollectionType.split

```
[1, 2, 3, 4].split { even($0) }
```

```
[ArraySlice([1]), ArraySlice([3])]
```

Optional

.map

.flatMap

Optional.map

```
evenOrdinal(2).map { $0 + "!" }
```

```
Optional("2nd!")
```

Optional.map

```
evenOrdinal(1).map { $0 + "!" }
```

```
Optional(nil)
```

Optional.map

```
let bar1 = {
    if let value = foo {
        return ordinal(value)
    } else {
        return "-"
    }
}()
// vs
let bar2 = foo.map { ordinal($0) } ?? "-"
```

Optional.flatMap

```
let foo: Int? = 2
foo.map     { evenOrdinal($0) } // "Optional(Optional("2nd"))
// vs
foo.flatMap { evenOrdinal($0) } // "Optional("2nd")
```

.flatMap allows failable transforms[1]

[1] Thanks to Harlan Haskins @harlanhaskins for clarifying

Odd

.zip([1, 1])

zip(Array, Array)

```
zip(["1st", "2nd", "3rd", "4th"], ["!", "?", "¡", "¿"])
    .map { $0 + $1 }

["1st!", "2nd?", "3rd¡", "4th¿"]
```

Result

```
enum Result<T> {
    case Success(T)
    case Failure(ErrorType)
}
```

Optional

```
enum Optional<T> {
    case Some(T)
    case None
}
```

Result

```
enum Result<T> {
    case Success(T)
    case Failure(ErrorType)
}
```

Result

```
extension Result<T> {
    func flatMap<U>(f: T -> Result<U>) -> Result<U> {
        switch self {
        case let .Success(t): return f(t)
        case let .Failure(err): return .Failure(err)
        }
    }
}
```

Result

```
func readFile(name: String) -> Result<Data> {}
func toJson(data: Data) -> Result<Dictionary> {}
func toCar(dict: Dictionary) -> Result<Car> {}

let userResult = readFile("car.json")
    .flatMap(toJson)
    .flatMap(toCar)
```

Result Async

```
extension Result<T> -> Void {
    func flatMap<U>(f: (T, Result<U> -> Void))
                    -> (Result<U> -> Void) {
        return { completion in
            switch self {
            case let .Success(t): f(t, completion))
            case let .Failure(err): return .Failure(err)
            }
        }
    }
}
```

Result Async

```
func readFile(file: String) -> (Result<Data> -> Void) {}
func toJson(data: Data) -> (Result<Dictionary> -> Void) {}
func toCar(dict: Dictionary) -> (Result<Car> -> Void) {}

let userResult = readFile("car.json")
    .flatMap(toJson)
    .flatMap(toCar)
```

Learning Functional Reactive Programming

Observable
Signal
Stream
Channel
Pipe

```
class Observable<T> {
    private var value: Result<T>?
    private var callbacks: [Result<T> -> Void] = []

    func subscribe(f: Result<T> -> Void) -> Observable<T> {
        if let value = value { f(value) }
        callbacks.append(f)
        return self
    }
    func update(result: Result<T>) {
        value = result
        callbacks.forEach { $0(result) }
    }
}
```

Observable Sync

```
extension Observable<T> {
    func flatMap<U>(f: T -> Result<U>) -> Observable<U> {
        let observable = Observable<U>()
        subscribe { result in
            observable(result.flatMap(f))
        }
        return observable
    }
}
```

Observable Async

```
extension Observable<T> {
    func flatMap<U>(f: (T, Result<U> -> Void) -> Void)
        -> Observable<U> {
            let observable = Observable<U>()
            subscribe { result in
                observable.update(result.flatMap(f))
            }
            return observable
        }
}
```

Observable

```
extension Observable<T> {
    func flatMap<U>(f:  T -> Result<U>)                    -> Observable<U>
    func flatMap<U>(f: (T, Result<U>->Void) -> Void) -> Observable<U>
    func flatMap<U>(f: (T -> Observable<U>))               -> Observable<U>

    // Bonus
    func map<U>(f: T -> U)                                 -> Observable<U>
    func flatMap<U>(f: T throws -> U)                      -> Observable<U>
}
```

Observable

```
extension Observable<T> {
    func filter(f: T -> Bool) -> Observable<T>
    func merge<U>(merge: Observable<U>) -> Observable<(T,U)>
}
```

Create an Observable

```
class Button {
    let observable = Observable<Bool>(value: false)

    private var selected: Bool {
        didSet {
            guard oldValue != selected else { return }
            observable.update(value: selected)
        }
    }
}
```

Use Observables

```
class VC: UIViewController {
    let (button0, button1) = (Button(), Button())
    func viewDidLoad() {
        button0.observable
            .merge(button1.observable)).
            .map { ($0.peek() ?? false) && ($1.peek() ?? false) }
            .next { self.valid = $0 } }
    }
}
```

Extend UIKit 😕

```
var SwitchHandle: UInt8 = 0
extension UISwitch {
    private(set) var valueObservable: Observable<Bool> {
        let observer: Observable<Bool>
        if let handle = objc_getAssociatedObject(self, &SwitchHandle) as? Observable<Bool> {
            observer = handle
        } else {
            observer = Observable()
            addTarget(self, action: #selector(UISwitch(_:)), forControlEvents: .ValueChanged)
            objc_setAssociatedObject(self, &SwitchHandle, observer, .OBJC_ASS…_RETAIN_NONATOMIC)
        }
        return observer
    }
    public func didChangeValue(sender: AnyObject) {
        valueObservable.update(on)
    }
}
```

Links

- The Best FRP Resources *by Javi Lorbada*
- Interstellar *by Jens Ravens*
- Async Errors *by Crunchy Development*
- Blending Cultures *by Daniel Steinberg*

Videos

- Functioning as a Functionalist *by Andy Matuschak*
- Controlling Complexity in Swift — or — Making Friends with Value Types *by Andy Matuschak*
- Functional Programming in Swift *by Chris Eidhof*
- Protocol-Oriented Programming Swift *WWDC 2015*
- Building Better Apps with Value Types in Swift *WWDC 2015*

Books

- Functional Swift *by objc.io*

Learning Functional Reactive Programming

Tobias Due Munk

@tobiasdm

github.com/duemunk

developmunk.dk