

RxSwift Basics – Day 4

Younghwan Kim



RxSwift Basics

- Day 1 – Observable, Operator (Filter, Transform, Combine)
- Day 2 – Subject (flatMap, flatMapFirst, flatMapLatest)
- Day 3 – Two VCs communications with Subject, RxCocoa (Button)
- **Day 4 – Sequential, Merged Observable Calls**
- Day 5 – RxCocoa, UI Binding (Button, TextField, Label, TableView)



Advanced RxSwift

- Day 1 – Protocol-Oriented Programming, Protocol Extension, Associatetype
- Day 2 – Network Call, Generic Enum
- Day 3 – Binding Track Activity (show / hide ‘Loading’), Scan Operator
- Day 4 – Adding a Reactive Extension to Custom UI Element,
2 Way Binding, Advanced TableView – RxDataSources
- Day 5 – Schedulers (observeOn, subscribeOn),
Unit Test (RxTest, RxBlocking)



Operation

```
let backgroundOperation = Operation()  
backgroundOperation.queuePriority = .low  
backgroundOperation.qualityOfService = .background
```

```
let operationQueue = OperationQueue.main  
operationQueue.addOperation(backgroundOperation)
```

```
let networkingOperation: Operation = ...  
let resizingOperation: Operation = ...  
resizingOperation.addDependency(networkingOperation)
```

```
let operationQueue = OperationQueue.main  
operationQueue.addOperations([networkingOperation, resizingOperation],  
                             waitUntilFinished: false)
```



Operation

```
let operation = Operation()  
operation.completionBlock = {  
    print("Completed")  
}
```

```
OperationQueue.main.addOperation(operation)
```



DispatchGroup

```
let dispatchGroup = DispatchGroup()

dispatchGroup.enter()
longRunningFunction { dispatchGroup.leave() }

dispatchGroup.enter()
longRunningFunctionTwo { dispatchGroup.leave() }

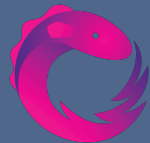
//Case 1. Sequential
dispatch_group_wait(dispatchGroup,
DISPATCH_TIME_FOREVER)

//Case 2. All Completion
dispatchGroup.notify(queue: .main) {
    print("Both functions complete")
}
```



Enum - merge return values

```
enum EnumReturnValues {  
    case success(Int)  
    case fail(String)  
}  
  
func enumTest(seed: Int) -> EnumReturnValues {  
    if seed == 1 {  
        return .success(1)  
    } else {  
        return .fail("Failure")  
    }  
}
```



Sequential Call

```
func seqControlTest() {  
    let fruitObservable = Observable.of("apple", "pineapple", "strawberry")  
    let coffeeObservable = Observable.of("McCafe", "TimHorton", "StarBucks")  
    let carObservable = Observable.of("Toyota", "Nissan", "Ford", "GM")  
  
    fruitObservable  
        .filter{  
            if $0 == "apple" {  
                print($0)  
                return true  
            } else {  
                return false  
            }  
        }  
        .flatMap { _ -> Observable<String> in  
            coffeeObservable  
  
                .filter{  
                    if $0 == "TimHorton" {  
                        print($0)  
                        return true  
                    } else {  
                        return false  
                    }  
                }  
                .flatMap { _ -> Observable<String> in  
                    carObservable  
                }  
                .subscribe(onNext: { car in  
                    print(car)  
                })  
                .disposed(by: self.disposeBag)  
            }  
        }  
}
```




Multi- Sequential Call

```
func multiSeqControlTest() {  
    let fruitObservable = Observable.of("apple","pineapple","strawberry")  
    let coffeeObservable = Observable.of("McCafe", "TimHorton", "StarBucks")  
    let carObservable = Observable.of("Toyota", "Nissan", "Ford","GM")  
  
    Observable<Bool>.create { observer in  
        Observable.merge([fruitObservable,  
                           coffeeObservable])  
            .subscribe(onNext: { thing in  
                print(thing)  
            }, onCompleted: {  
                observer.onNext(true)  
                observer.onCompleted()  
            })  
            .disposed(by: self.disposeBag)  
        return Disposables.create()  
    }  
}
```

```
.filter{  
    $0  
}  
.flatMap { _ -> Observable<String> in  
    carObservable  
}  
.subscribe(onNext: { car in  
    print(car)  
}).disposed(by: self.disposeBag)
```



Traditional

```
Alamofire.request("https://fake/login", method: .post)
    .responseJSON { (response) in
        Alamofire.request("https://fake/creaditToken", method: .post, parameters: ["creditToken": response.id])
            .responseJSON { (response) in
                Alamofire.request("https://fake/creaditCard", method: .post, parameters: ["creditCard": response.token])
                    .responseJSON {
                        // get credit card info
                    }
            }
    }
```



Traditional

```
func fetchInformation() {  
    networking.fetchAppVersion()  
        .responseJSON(completionHandler: { (responseJSON) in  
        self.networking.fetchConfig()  
            .responseJSON(completionHandler: { (responseJSON2) in  
                self.networking.fetchProfile()  
                    .responseJSON(completionHandler: { (responseJSON3) in  
                        print("\(responseJSON)")  
                        print("\(responseJSON2)")  
                        print("\(responseJSON3)")  
                    })  
                })  
            })  
        })  
}
```



Reactive

```
class CreditCardService {  
    func getCreditCards() {  
        let service = [] as! Service  
        service.rxLogin(username: "admin@gmail.com", password: "12345")  
            .flatMap { (authResponse) -> Observable<CreditCardAccount> in  
                return service.rxCredidCardAccount(userId: authResponse.userId)  
            }  
            .flatMap { (creditCardAccount) -> Observable<[CreditCardInfo]> in  
                return service.rxAllCreditCards(userId: creditCardAccount.cardsId)  
            }  
            .subscribe { (creditCardInfo) in  
                print(creditCardInfo)  
            }  
    }  
}
```



- Serial, Multi Calls
- Different Type Observable