

Advanced RxSwift – Day 5

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)



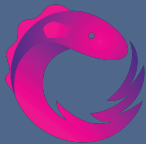
Advanced RxSwift

- Day 1 – Protocol-Oriented Programming, Protocol Extension, AssociateType
- Day 2 – Network Call, Generic Enum
- Day 3 – Binding Track Activity (show / hide 'Loading')
- 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)**



Scheduler – observeOn, subscribeOn

```
Observable<Int>.create { observer in
    print(Thread.currentThread())
    observer.onNext(1)
    sleep(1)
    observer.onNext(2)
    return Disposables.create()
}
.observeOn(MainScheduler.instance)
.subscribeOn(ConcurrentDispatchQueueScheduler(qos: .background))
    .subscribe(onNext: { el in
        print(Thread.currentThread())
    }).disposed(by: disposeBag)
```



Scheduler – observeOn, subscribeOn

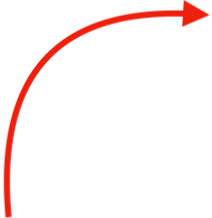
```
Observable.create { observer in  
    subscription code  
}
```

operators (map, filter, etc)

```
subscribe(  
    observing code  
)
```



Scheduler – observeOn, subscribeOn



```
Observable.create { observer in  
    subscription code  
}
```

```
operators (map, filter, etc)
```

```
subscribe(  
  
)
```



Scheduler – observeOn, subscribeOn

```
Observable.create { observer in
```

```
}
```

```
operators (map, filter, etc)
```

```
subscribe(
```

```
  onNext: {
```

```
    observing
```

```
  }
```

```
  onComplete: {
```

```
    observing
```

```
  }
```

```
)
```



RxTest

- `TestableObserver<ElementType>` - an observer, which records all emitted events so you can inspect them and run your asserts on those events
- `TestScheduler` - a scheduler which let's you control values and time, and let's you create testable observers
- `TestObservable` - `Observable`, where you can pass what events should it send at given schedule
- `== (lhs: Event<Element>, rhs: Event<Element>)` adds `Equatable` implementation to Rx events so you can easily check recorded events



```
struct CornSorter {  
    let barnStream: Observable<String>  
  
    init(tractorStream: Observable<String>) {  
        barnStream = tractorStream  
            .filter { $0 == "🌽" }  
    }  
}
```



RxTest

```
func testCornSorter() {  
    var scheduler: TestScheduler!  
    let disposeBag = DisposeBag()  
  
    scheduler = TestScheduler(initialClock: 0)  
    let testObserver = scheduler.createObserver(String.self)  
    // Given  
    let observableInput = scheduler.createHotObservable([  
        // 2  
        Recorded.next(100, "🌽"),  
        Recorded.next(200, "🐛"),  
        Recorded.next(300, "🐭"),  
        Recorded.next(400, "🌽"),  
        Recorded.next(500, "🐝"),  
        Recorded.next(600, "🐞")  
    ])  
    let cornSorter = CornSorter(tractorStream: observableInput.asObservable())
```



RxTest

```
// When
cornSorter.barnStream
    .subscribe(testObserver)
    .disposed(by: disposeBag)

scheduler.start()

// Then
let results = testObserver.events.map {
    $0.value.element!
}
_ = XCTAssertEqual(results, ["🌽", "🌽"])
}
```



RxBlocking

- RxBlocking on the other hand is handy in case you need to test some asynchronous functionality where you can't control the source of asynchronicity. Often times this means you're stepping up from unit tests to integration test.
- What RxBlocking is great to is to allow you to consume an observable sequence in batches or even wait on a single element to be emitted.



RxBlocking

```
func testElements() {  
    let items = Observable.of(1, 5, 10, 15, 20)  
  
    let elements = try! items.toBlocking().toArray()  
    XCTAssertEqual([1, 5, 10, 15, 20], elements)  
  
    let results = try! items.skip(3).take(2).toBlocking().toArray()  
    XCTAssertEqual([15, 20], results)  
}
```



RxBlocking

```
func testCountryInfoFlow() {
    let scheduler = ConcurrentDispatchQueueScheduler(qos: .default)
    do {
        let myArray = try BordersBusinessLogic.shared.countryInfoFlow(code: "FRA")
        .subscribeOn(scheduler)
        .toBlocking()
        .toArray()
        if let countryInfo = myArray.first {
            switch countryInfo {
            case .success(_):
                XCTAssert(true)
                break
            case .failure(_):
                XCTAssert(false)
                break
            }
        }
    } catch(let e) { XCTAssert(false, e.localizedDescription)
    }
}
```