

Advanced RxSwift – 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 – Binding, KVO, 2-Way Bindings**

Adding a Reactive Extension to Custom UI Element, Advanced TableView – RxDataSources

- Day 5 – Schedulers (observeOn, subscribeOn),
Unit Test (RxTest, RxBlocking)



Binding

```
OneObject: ObservableType  
  
    .bind(to: TwoObject: ObserverType)  
  
    .disposed(by disposeBag)
```



ObservableType

```
public protocol ControlPropertyType : ObservableType, ObserverType

public struct ControlProperty<PropertyType> : ControlPropertyType

public class Observable<Element> : ObservableType

public final class BehaviorRelay<Element>: ObservableType
  => any Subject and Relay
```



ObserverType

```
public struct Binder<Value>: ObserverType
```

```
public protocol ControlPropertyType : ObservableType, ObserverType
```

```
public struct ControlProperty<PropertyType> : ControlPropertyType
```

```
public final class PublishSubject<Element>: Observable<Element>, SubjectType  
Cancelable, ObserverType => any Subject and Relay
```



Binder? Or ControlProperty?

Binder

```
UIView - .isHidden, .alpha,  
        .isUserInteractionEnabled
```

```
UILabel - .text, .attributedText
```

```
UIControl - isEnabled, .isSelected
```

```
UIButton  
    = .title, .image, .backgroundImage, .attributedTitle
```

```
UIBarButtonItem - isEnabled, .title
```

```
UIImageView = .image
```

```
UIRefreshControl = .isRefreshing
```

```
UIActivityIndicatorView - .isAnimating
```



Binder? Or ControlProperty?

Binder

`NSLayoutConstraint` - `.constant` , `.active`

`UIAlertAction` - `.isEnabled`

`UINavigationController` - `.title`

`UIPageControl` = `.currentPage`,
`.numberOfPage`

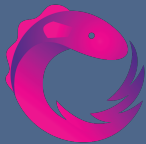
`UIProgressView` - `.progress`

`UIScrollView` - `.isScrollEnabled`

`UIStepper` - `.stepValue`

`UITabBarItem` - `.badgeValue`

`UIViewController` = `.title`



Binder? Or ControlProperty?

ControlProperty

UITextField - .text, .attributedText, .textInput

UITextView - .text, .attributedText, .textInput

UISearchBar - .text, .selectedScopeButtonIndex

UISegementedControl - .selectedSegmentIndex

UIDatePicker - .date, .countDownDuration

UIScrollView - .contentOffset

UISlider - .value

UIStepper - .value

UISwitch - .isOn



Binding

```
cell.textValue.asObservable()  
    .bind(to: self.userInputLabel.rx.text)  
    .disposed(by: cell.disposeBag)
```

```
cell.textValue.asDriver()  
    .drive(self.userInputLabel.rx.text)  
    .disposed(by: cell.disposeBag)
```

```
cell.textValue.asObservable()  
    .subscribe(onNext: { input in  
        self.userInputLabel.text = input  
    })  
    .disposed(by: cell.disposeBag)
```

```
cell.textValue.asDriver()  
    .drive(onNext: { input in  
        self.userInputLabel.text = input  
    })  
    .disposed(by: cell.disposeBag)
```



Binding - KVO

```
@IBOutlet weak var oneLabel: UILabel!  
@IBOutlet weak var twoLabel: UILabel!  
let disposeBag = DisposeBag()  
  
oneLabel.rx.observe(Bool.self, "hidden")  
    .subscribe(onNext: { [unowned self] hidden in  
        if let _hidden = hidden {  
            self.twoLabel.isHidden = _hidden  
        }  
    }).disposed(by: disposeBag)
```



Binding - KVO

```
@IBOutlet weak var oneLabel: UILabel!  
@IBOutlet weak var twoLabel: UILabel!  
let hiddenRelay = BehaviorRelay<Bool>(value: false)  
let disposeBag = DisposeBag()  
  
oneLabel.rx.observe(Bool.self, "hidden")  
    .subscribe(onNext: { [unowned self] hidden in  
        if let _hidden = hidden {  
            self.hiddenRelay.accept(_hidden)  
        }  
    }).disposed(by: disposeBag)  
  
self.hiddenRelay.asObservable()  
    .bind(to: self.twoLabel.rx.isHidden)  
    .disposed(by: disposeBag)
```



Binding - KVO

```
@IBOutlet weak var oneLabel: UILabel!  
@IBOutlet weak var twoLabel: UILabel!  
let disposeBag = DisposeBag()  
  
oneLabel.rx.observe(Bool.self, "hidden")  
    .map { //unwrapping  
        if let hidden = $0 {  
            return hidden  
        } else {  
            return false  
        }  
    }  
    .bind(to: self.twoLabel.rx.isHidden)  
    .disposed(by: disposeBag)
```



KVO

https://developer.apple.com/library/content/documentation/Swift/Conceptual/BuildingCocoaApps/AdoptingCocoaDesignPatterns.html#//apple_ref/doc/uid/TP40014216-CH7-XID_8

Key-Value Observing

Key-value observing is a mechanism that allows objects to be notified of changes to specified properties of other objects. You can use key-value observing with a Swift class, as long as the class inherits from the NSObject class. You can use these two steps to implement key-value observing in Swift.

Add the dynamic modifier and @objc attribute to any property you want to observe. For more information on dynamic, see Requiring Dynamic Dispatch.

```
class MyObjectToObserve: NSObject {  
    @objc dynamic var myDate = NSDate()  
    func updateDate() {  
        myDate = NSDate()  
    }  
}
```



KVO

```
@IBOutlet weak var kvoTestButton: UIButton!
@dynamic var someString = ""
@dynamic var someBoolean = false

let disposeBag = DisposeBag()

//KVO Test
self.rx.observe(String.self, "someString")
    .subscribe(onNext: { some in
        if let _some = some {
            print(_some)
        } else {
            print("")
        }
    }).disposed(by: disposeBag)
```

```
self.rx.observe(Bool.self, "someBoolean")
    .subscribe(onNext: { some in
        if let _some = some {
            print(_some ? "true" : "false")
        } else {
            print("false")
        }
    }).disposed(by: disposeBag)

func kvoTest() {
    self.someBoolean = !self.someBoolean
    self.someString = self.someBoolean ?
        "KVO Test 1" : "KVO Test 2"
}
```



2 Way Bindings

Custom Implementation

<https://github.com/ReactiveX/RxSwift/blob/master/RxExample/RxExample/Operators.swift>

```
func <-> <T>(property: ControlProperty<T>, variable: BehaviorRelay<T>) -> Disposable {  
  
    let bindToUIDisposable = variable.asObservable()  
        .bind(to: property)  
    let bindToVariable = property  
        .subscribe(onNext: { n in  
            variable.accept(n)  
        }, onCompleted: {  
            bindToUIDisposable.dispose()  
        })  
  
    return Disposables.create(bindToUIDisposable, bindToVariable)  
}
```




2 Way Bindings

```
@IBOutlet weak var textField: UITextField!  
let textValue = BehaviorRelay(value: "")  
var disposeBag = DisposeBag()  
  
override func awakeFromNib() {  
    super.awakeFromNib()  
    // Initialization code  
  
    let textDisposable = textField.rx.textInput <-> textValue  
    textDisposable.disposed(by: self.disposeBag)  
}
```



Adding a reactive extension to Custom UI Element

UILabel

```
myObservable
    .map { "new value is \($0)" }
    .bind(to: myLabel.rx.text )
    .disposed(by: bag)
```

```
extension Reactive where Base: UILabel {

    /// Bindable sink for `text` property.
    public var text: Binder<String?> {
        return Binder(self.base) { label, text in
            label.text = text
        }
    }
}
```



Adding a reactive extension to Custom UI Element

SwiftSpinner

```
Observable<Int>.timer(0.0, period: 0.15, scheduler: MainScheduler.instance)
    .bind(to: SwiftSpinner.sharedInstance.rx.progress )
    .disposed(by: bag)
```

```
extension Reactive where Base: SwiftSpinner {
    public var progress: Binder<Int> {
        return Binder(self.base) { spinner, progress in
            let progress = max(0, min(progress, 100))
            SwiftSpinner.show(progress: Double(progress)/100.0, title: "\(progress)% completed")
        }
    }
}
```



RxDataSources

Using **RxDataSources** requires more work to learn its idioms, but offers more powerful, advanced features. Instead of a simple array of data, it requires you to provide contents using objects which conform to the **SectionModelType** protocol.

Each section itself contains the actual objects. For sections with multiple object types, use the enum technique shown above to differentiate the types.

<https://github.com/RxSwiftCommunity/RxDataSources>



Lab

<https://github.com/younghwankim/RxSwiftClass/tree/master/AdvancedRxSwift/day4/AdvancedTableView>