Advanced RxSwift – Day 1

Younghwan Kim

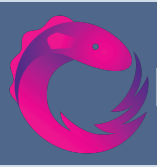
RxSwift Basics

- 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)



Protocol-Oriented Programming in Swift

WWDC 2015 – Session 408

At the heart of Swift's design are two incredibly powerful ideas: protocol-oriented programming and first class value semantics. Each of these concepts benefit predictability, performance, and productivity, but together they can change the way we think about programming.

Find out how you can apply these ideas to improve the code you write.

https://developer.apple.com/videos/play/wwdc2015/408/

Protocol – Example 1

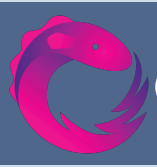
```
protocol Poppable {
    func pop()
}

extension Poppable where Self: UIView {
    func pop() {
        UIView.animate(withDuration: 0.3,
                       delay: 0,
                       options: .curveEaseIn,
                       animations: { self.alpha = 1.0 }) { (animationCompleted) in
                        if animationCompleted == true {
                            UIView.animate(withDuration: 0.3,
                                           delay: 2.0,
                                           options: .curveEaseOut,
                                           animations: { self.alpha = 0.0 },
                                           completion: nil)
                        }
        }
    }
}
```

Protocol – Example 2

```
protocol Buzzable {
    func buzz()
}

extension Buzzable where Self: UIView {
    func buzz() {
        let animation = CABasicAnimation(keyPath: "position")
        animation.duration = 0.05
        animation.repeatCount = 5
        animation.autoreverses = true
        animation.fromValue = NSValue(cgPoint: CGPoint(x: self.center.x - 5.0, y: self.center.y))
        animation.toValue = NSValue(cgPoint: CGPoint(x: self.center.x + 5.0, y: self.center.y))
        layer.add(animation, forKey: "position")
    }
}
```

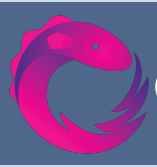
Protocol – Example 3

```
class BuzzableTextField: UITextField, Buzzable {}
class BuzzableButton: UIButton, Buzzable {}
class BuzzableImageView: UIImageView, Buzzable {}
class BuzzablePoppableLabel: UILabel, Buzzable, Poppable {}

class LogInViewController: UIViewController {
    @IBOutlet weak var passcodTextField: BuzzableTextField!
    @IBOutlet weak var loginButton: BuzzableButton!
    @IBOutlet weak var errorMessageLabel: BuzzablePoppableLabel!
    @IBOutlet weak var profileImageView: BuzzableImageView!

    @IBAction func loginButtonClicked(_ sender: UIButton) {
        passcodTextField.buzz()
        loginButton.buzz()
        errorMessageLabel.buzz()
        errorMessageLabel.pop()
        profileImageView.buzz()
    }
```



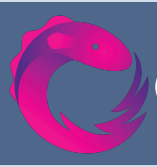
Grouping – Protocol Extension, associatetype

```
public struct Extension<Base> {
    public let base: Base // Generic Type (Base)
    public init(_ base: Base) {
        self.base = base
    }
}
```



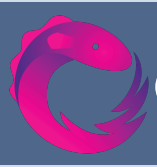
Grouping – Protocol Extension, associatetype

```
public struct Extension<Base> {
    public let base: Base // Generic Type (Base)
    public init(_ base: Base) {
        self.base = base
    }
}

//  Associated Types
//
//  When defining a protocol, it's sometimes useful to declare one or more associated types as part of the protocol's definition.
//  An associated type gives a placeholder name to a type that is used as part of the protocol. The actual type to use for
//  that associated type isn't specified until the protocol is adopted. Associated types are specified with the associatedtype keyword.

public protocol ExtensionCompatible {
    associatedtype Compatible
    var ex: Extension<Compatible> { get }
}
```



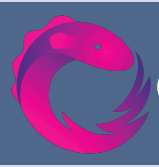
Grouping – Protocol Extension, associatetype

```
//  Protocol Associated Type Declaration
//
//  Protocols declare associated types using the associatedtype keyword. An associated type provides an alias for a type that is used
//    as part of a protocol's declaration. Associated types are similar to type parameters in generic parameter clauses, but they're
//    associated with Self in the protocol in which they're declared. In that context, Self refers to the eventual type that conforms
//    to the protocol. For more information and examples, see Associated Types.

// Generic Conforming
public extension ExtensionCompatible {
    //Self refers to the eventual type that conforms to the protocol
    public var ex: Extension<Self> {
        get {
            return Extension(self)
        }
    }
}
```



Grouping – Protocol Extension, associatetype

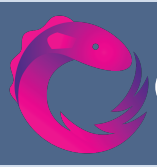
```
extension Int: ExtensionCompatible { }

public extension Extension where Base == Int {
    public var cube: Int {
        return base * base * base
    }
}


 print("\(4.ex.cube)")
```



Grouping – Protocol Extension, associatetype

```
public struct ExtensionTwo<Base> {
    public let base: Base
    public init(_ base: Base) {
        self.base = base
    }
}

public protocol ExtensionTwoCompatible {
    associatedtype Compatible
    var ext: ExtensionTwo<Compatible> { get set }
    static var ext: ExtensionTwo<Compatible>.Type
    { get set }
}
```

```
public extension ExtensionTwoCompatible {
    public var ext: ExtensionTwo<Self> {
        get {
            return ExtensionTwo(self)
        }
        set {
            // this enables using Extension to "mutate" base object
        }
    }
    public static var ext: ExtensionTwo<Self>.Type {
        get {
            return ExtensionTwo<Self>.self
        }
        set {
            // this enables using Extension to "mutate" base type
        }
    }
}
```



Grouping – Protocol Extension, associatetype

```
extension Int: ExtensionTwoCompatible { }

public extension ExtensionTwo where Base == Int {
    public var cube: Int {
        return base * base * base
    }
    public static var almostMax: Int {
        return Int.max - 1
    }
}


print("\(4.ext.cube)")
print("\(Int.ext.almostMax)")
```

RxSwift – Reactive.swift

https://github.com/ReactiveX/RxSwift/blob/master/RxSwift/Reactive.swift

RxSwift – Reactive.swift

```
public struct Reactive<Base> {
    /// Base object to extend.
    public let base: Base

    /// Creates extensions with base object.
    public init(_ base: Base) {
        self.base = base
    }
}
/// A type that has reactive extensions.
public protocol ReactiveCompatible {
    /// Extended type
    associatedtype CompatibleType

    /// Reactive extensions.
    static var rx: Reactive<CompatibleType>.Type { get set }
    /// Reactive extensions.
    var rx: Reactive<CompatibleType> { get set }
}
```

RxSwift – Reactive.swift

```
extension ReactiveCompatible {
    /// Reactive extensions.
    public static var rx: Reactive<Self>.Type {
        get {
            return Reactive<Self>.self
        }
        set {// this enables using Reactive to "mutate" base type }
    }
    /// Reactive extensions.
    public var rx: Reactive<Self> {
        get {
            return Reactive(self)
        }
        set {// this enables using Reactive to "mutate" base object }
    }
}
import class Foundation.NSObject

/// Extend NSObject with `rx` proxy.
extension NSObject: ReactiveCompatible { }
```