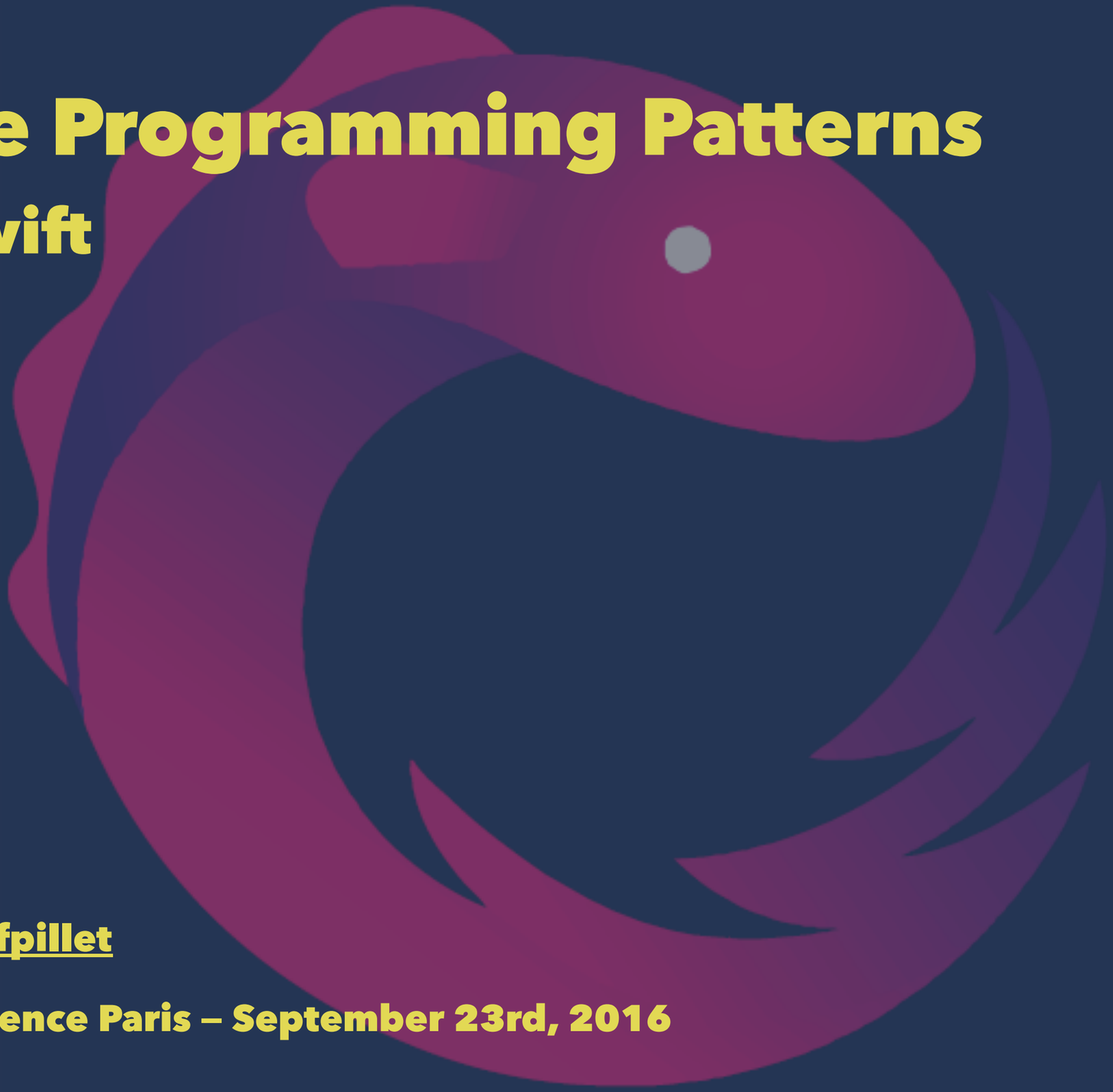
# Reactive Programming Patterns
## with RxSwift



Florent Pillet – @fpillet

FrenchKit Conference Paris – September 23rd, 2016

# Agenda

- Introduction to Rx
- Creating observable sequences
- Basic patterns
- User interface patterns
- Architecture patterns

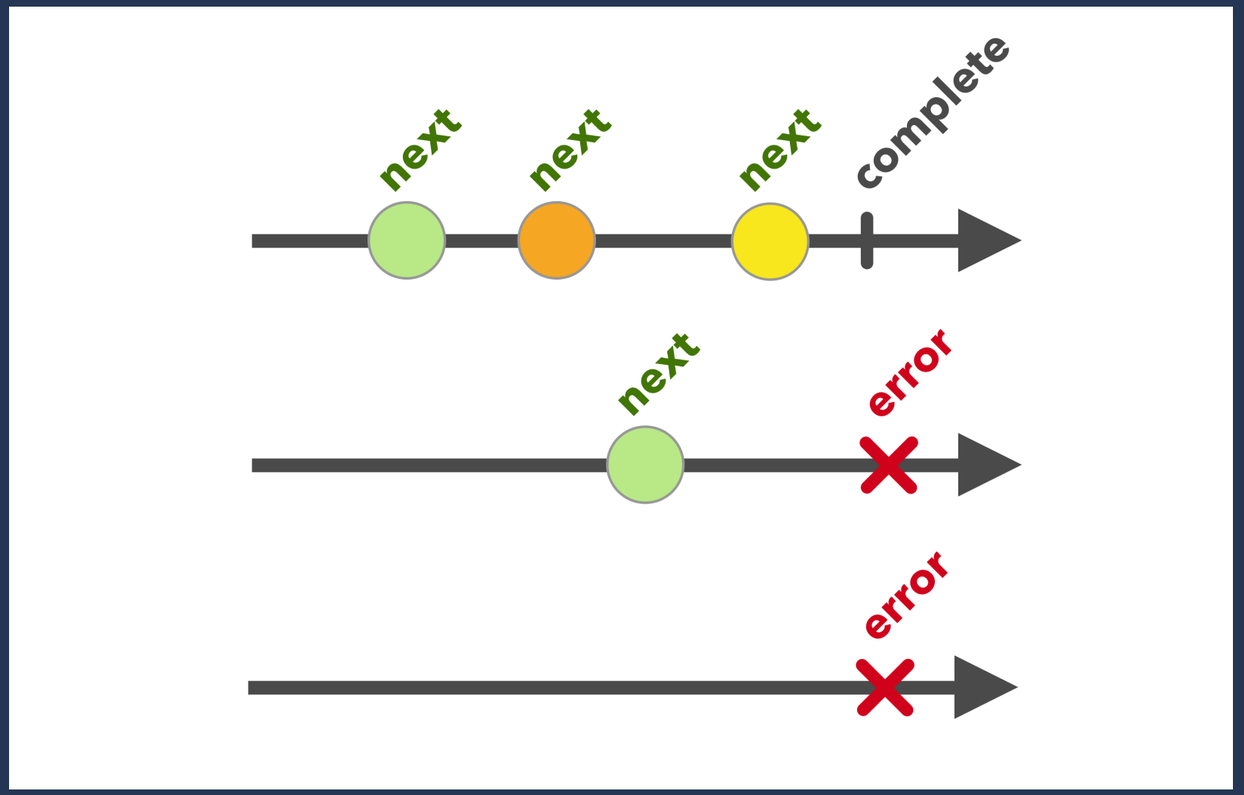
# About Rx

- Microsoft Reactive Extensions (Rx.NET) - 2009
- ReactiveX.io defines a common API for Rx implementations
- RxSwift 2015
- Shares concepts and API with other implementations
- Heavily tested framework


SALES
COSTS
PROFITS
JANUARY 150 90
FEBRUARY 165 99
MARCH 182 109
APRIL 200 120
MAY 220 132
JUNE 242 145
JUNE 758 455
JULY 834 500
AUGUST 917 550
SEPTEMBER 1009 605
OCTOBER 1110 666
NOVEMBER 1221 733
DECEMBER 1343 806
disk II
disk II
apple II

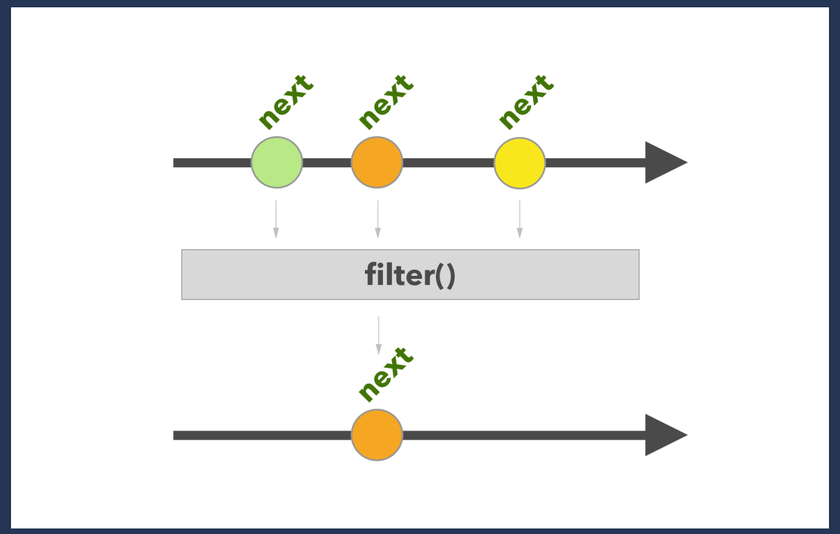
# Base concepts

Asynchronous observable **sequences**



# Base concepts

- Compose and transform observable sequences using **operators**
- *Rx* models the **asynchronous propagation of change**



```
let urlString = "http://api.openweathermap.org/data/2.5/weather?
    units=metric&APPID=\(API_KEY)&q=Paris"
let urlRequest = NSURLRequest(URL: NSURL(string: urlString)!)

NSURLSession.sharedSession()
    .rx_JSON(urlRequest)
    .map {
        let root = $0 as! [String:AnyObject]
        let main = root["main"] as! [String:AnyObject]
        return main["temp"] as! Float
    }
    .subscribeNext { (temp: Float) in
      print("Temperature in Paris is now \(temp)")
    }
```

```
let urlString = "http://api.openweathermap.org/data/2.5/weather?
    units=metric&APPID=\(API_KEY)&q=Paris"
let urlRequest = NSURLRequest(URL: NSURL(string: urlString)!)

NSURLSession.sharedSession()
    .rx_JSON(urlRequest)
    .map {
        let root = $0 as! [String:AnyObject]
        let main = root["main"] as! [String:AnyObject]
        return main["temp"] as! Float
    }
    .subscribeNext { (temp: Float) in
      print("Temperature in Paris is now \(temp)")
    }
```

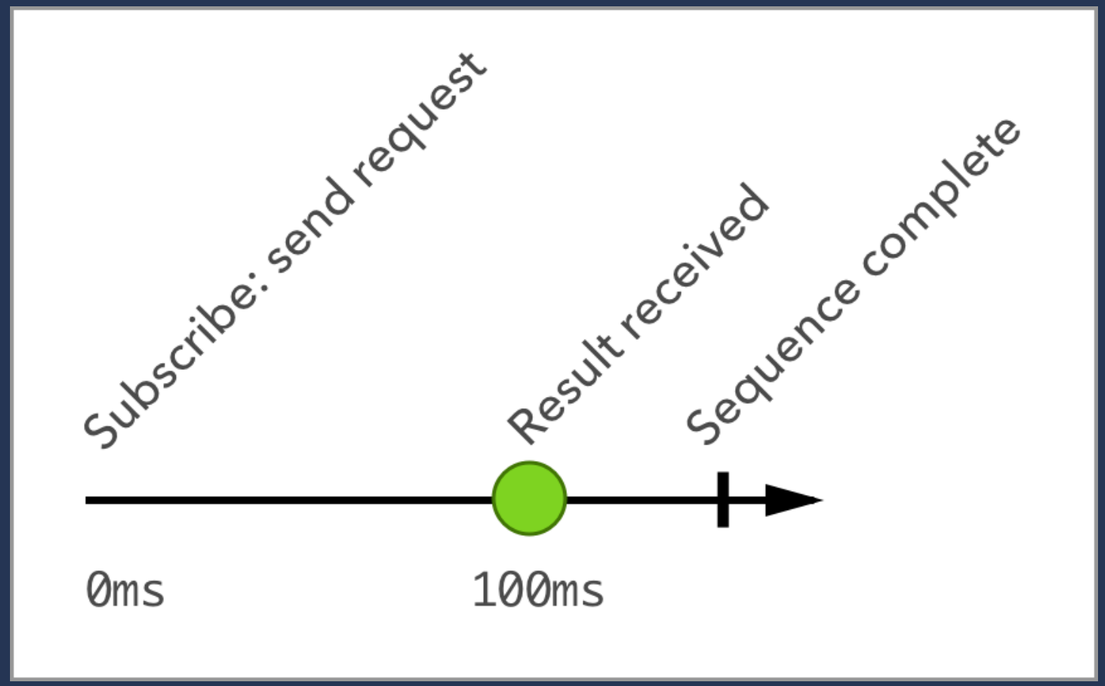
```
let urlString = "http://api.openweathermap.org/data/2.5/weather?
    units=metric&APPID=\(API_KEY)&q=Paris"
let urlRequest = NSURLRequest(URL: NSURL(string: urlString)!)

NSURLSession.sharedSession()
    .rx_JSON(urlRequest)
    .map {
        let root = $0 as! [String:AnyObject]
        let main = root["main"] as! [String:AnyObject]
        return main["temp"] as! Float
    }
    .subscribeNext { (temp: Float) in
      print("Temperature in Paris is now \(temp)")
    }
```

```
let urlString = "http://api.openweathermap.org/data/2.5/weather?
    units=metric&APPID=\(API_KEY)&q=Paris"
let urlRequest = NSURLRequest(URL: NSURL(string: urlString)!)

NSURLSession.sharedSession()
    .rx_JSON(urlRequest)
    .map {
        let root = $0 as! [String:AnyObject]
        let main = root["main"] as! [String:AnyObject]
        return main["temp"] as! Float
    }
    .subscribeNext { (temp: Float) in
      print("Temperature in Paris is now \(temp)")
    }
```

# What did we just see?

- a **sequence** emitted one item then completed,
- the `map` operator **transformed** a sequence of JSON objects into a sequence of Floats,
- similar to Swift sequence mapping but **asynchronous**.

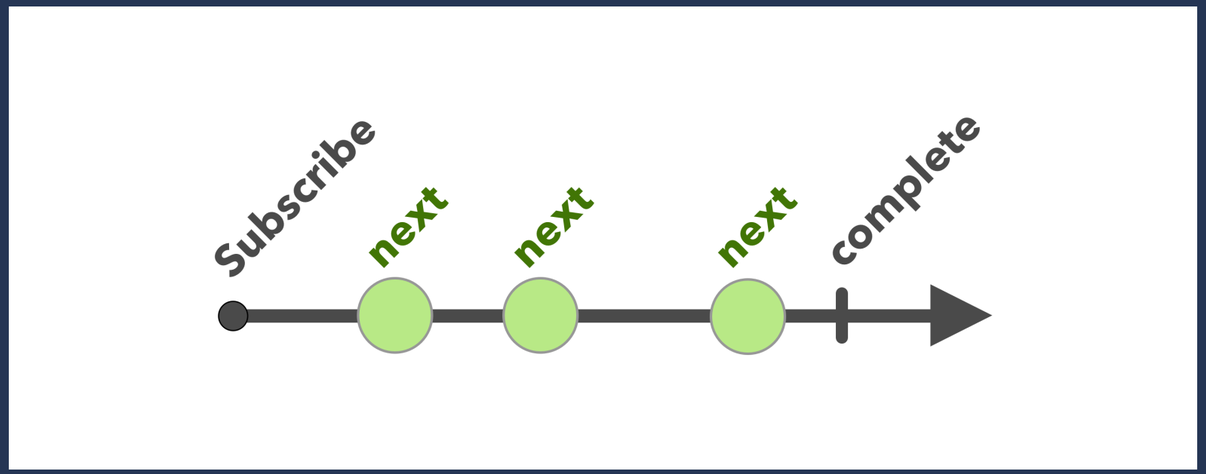
# What did we just see?




Experiment with interactive marble diagrams at rxmarbles.com

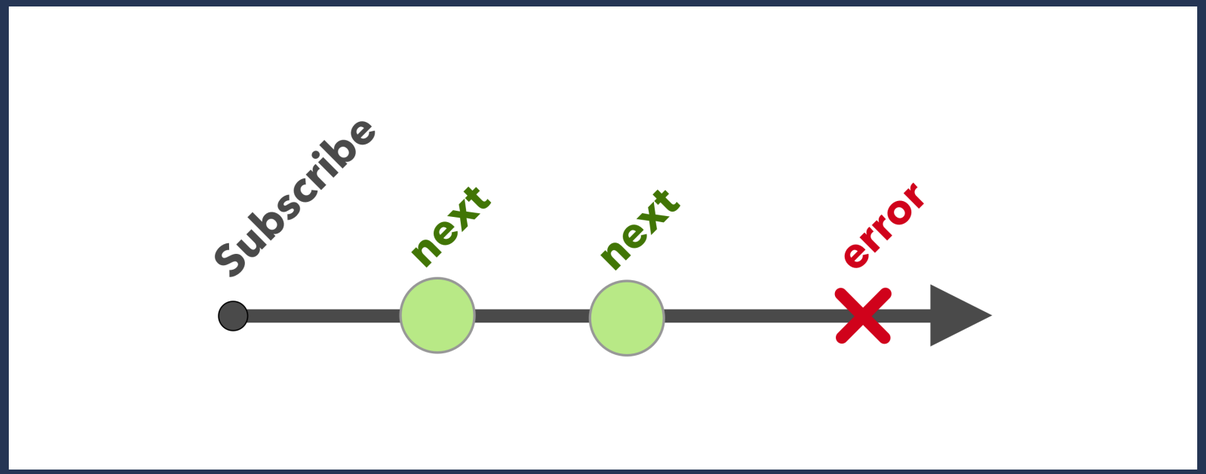
# Observable sequence lifecycle

```
let disposable = someObservable.subscribe(
    onNext:      { print("value: \($0)") },
    onCompleted: { print("completed")    },
    )
```



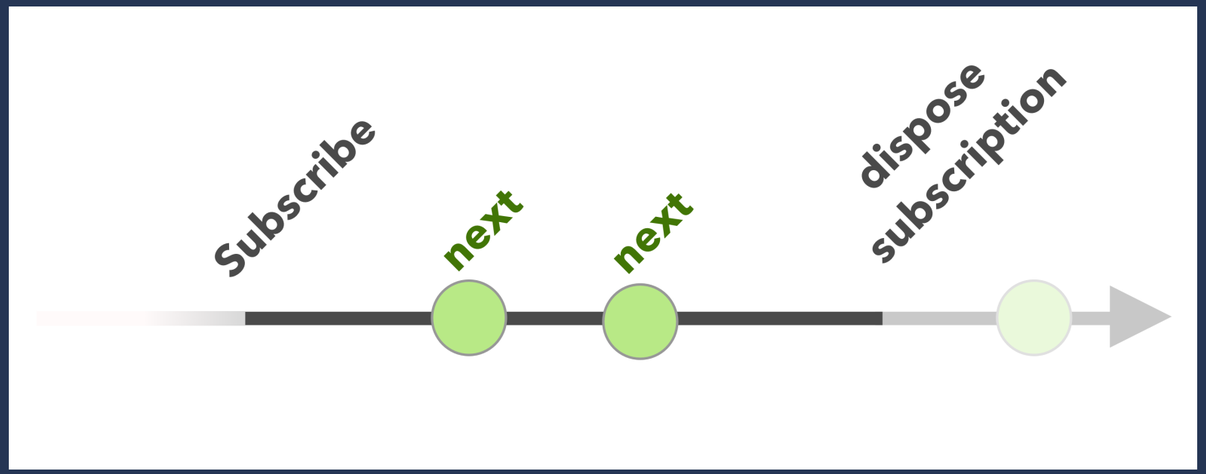
# Observable sequence lifecycle

```
let disposable = someObservable.subscribe(
    onNext:      { print("value: \($0)") },
    onError:     { print("error \($0)")  },
    onCompleted: { print("completed")    }
    )
```



# Observable sequence lifecycle

```
let disposable = someObservable.subscribe(
    onNext:      { print("value: \($0)") },
    onError:     { print("error \($0)")  },
    onCompleted: { print("completed")    },
    onDisposed:  { print("disposed")     }
    )

// at any point, cancel your subscription
// by calling dispose()

disposable.dispose()
```



# The mysterious genesis of the Observable

# The mysterious genesis of the Observable

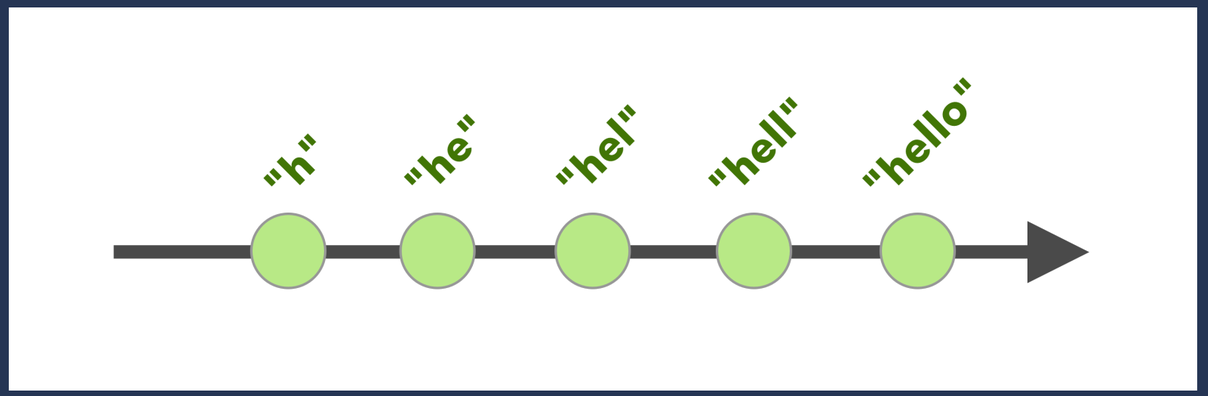
## RxCocoa

```
import RxCocoa


let disposable = NSNotificationCenter.defaultCenter()
    .rx_notification(UIApplicationSignificantTimeChangeNotification)
    .subscribeNext {
        (notification: UINotification) in
        print("Date changed: time to update!")
    }
```

# The mysterious genesis of the Observable

## RxCocoa

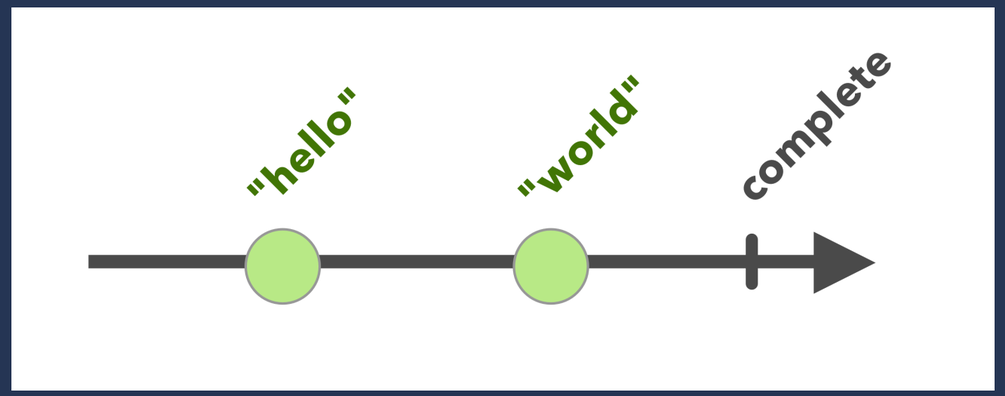
```
import RxCocoa


@IBOutlet var textField : UITextField!

override func viewDidLoad() {
    super.viewDidLoad()
    let _ = textField.rx_text.subscribeNext {
            (text: String) in
            print("text field changed to \(text)")
    }
}
```



# The mysterious genesis of the Observable

## Manual creation

```
let strings : Observable<Int> =
    Observable.create { observer in

        observer.onNext("Hello")
        observer.onNext("World")
        observer.onCompleted()

        // we don't need to release any
        // resource on dispose()
        return NopDisposable.instance
    }
```



# The mysterious genesis of the Observable

## Manual creation

```
let asyncComputation : Observable<Data> =
    Observable.create { observer in
        let task = someAsyncTask()
        task.run(
            success: {
                (result: Data) in
                observer.onNext(result)
                observer.onCompleted()
            }
            error:  {
                (error: ErrorType) in
                observer.onError(error)
            }
        )
        return AnonymousDisposable {
            task.cancel()
        }
    }
```

# The mysterious genesis of the Observable

More ways to obtain observables:

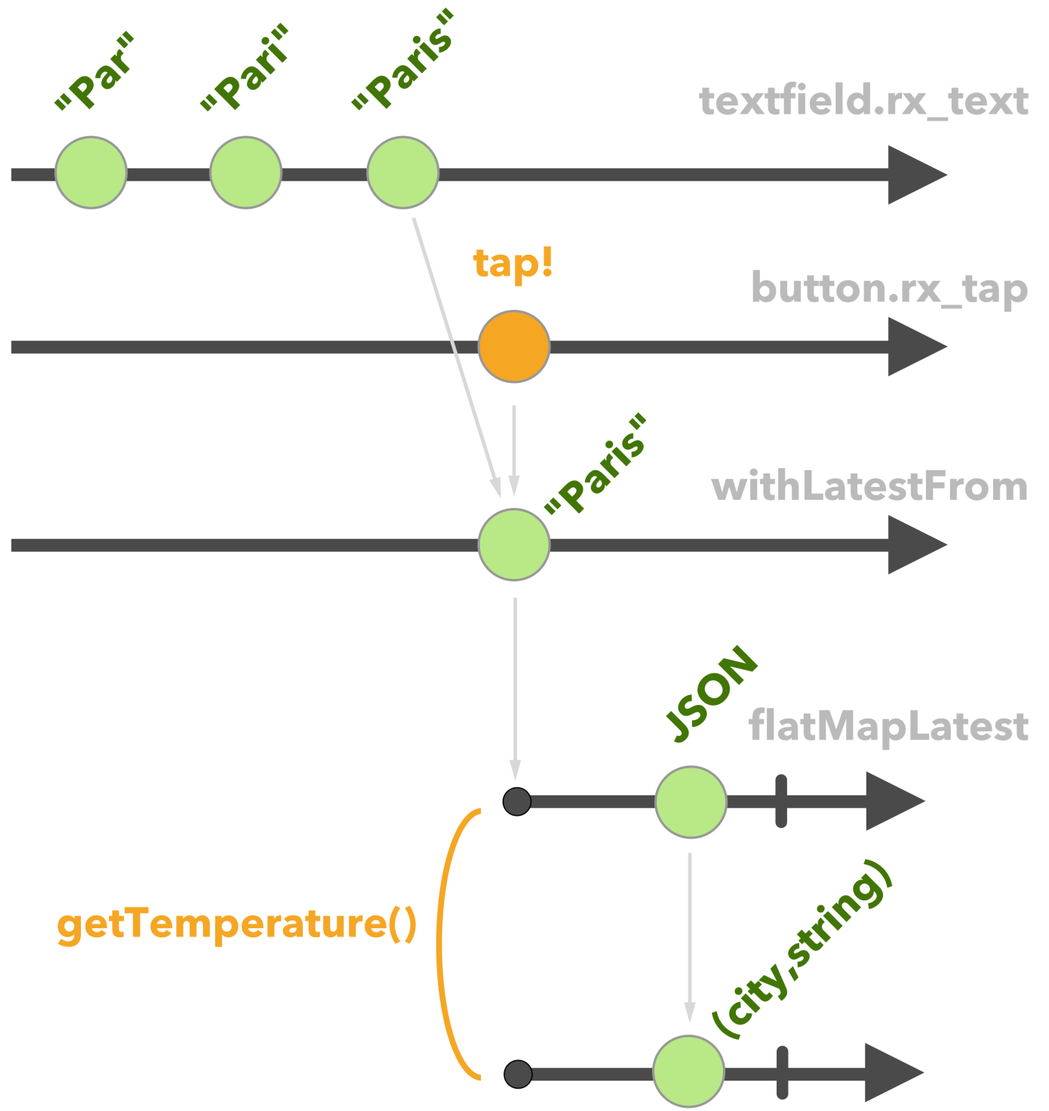
- Items from an array or collection
- `DelegateProxy`
- `rx_observe(type, keypath, options)`
- `rx_sentMessage(#selector)`
- `Subject` (*stateless*) and `Variable` (*stateful*)

# Basic Patterns

# Composition

## Task: update temperature label when button tapped

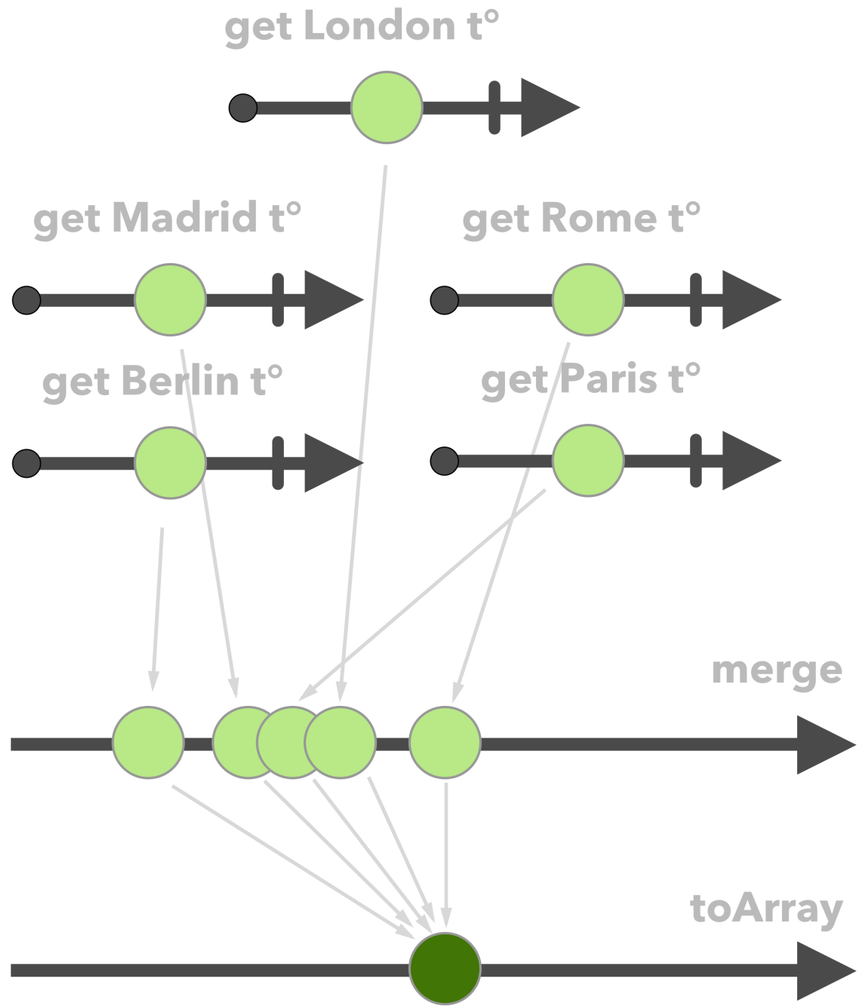
```
func getTemperature(city: String)
    -> Observable<(String,Float)>

func formattedTemperature(temp: Float)
    -> String

let disposable = button.rx_tap
    .withLatestFrom(textField.rx_text)
    .flatMapLatest {
        (city: String) -> Observable<(String,Float)> in
        return getTemperature(city)
    }
    .subscribeNext {
        (temp: (String,Float)) in
        let degrees = formattedTemperature(temp.1)
        label.text = "It's \(degrees) in \(temp.0)"
    }
```



# Aggregation

**Task: obtain the current temperature in multiple cities**

```
let disposable = ["Berlin","London",
                  "Madrid","Paris",
                  "Rome"]
    .map {
        (city: String) -> Observable<(String,Float)> in
        return getTemperature(city)
    }
    .toObservable()
    .merge()
    .toArray()
    .subscribeNext {
        (temperatures: [(String,Float)]) in
        // we get the result of the five requests
        // at once in a nice array!
    }
```



# Cancellation

## Task: update temperature every second until VC disappears

```
var timerDisposable : Disposable!

override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)
    timerDisposable = Observable
        .timer(0.0, period: 60.0, scheduler: MainScheduler.instance)
        .flatMap { _ -> Observable<String> in
            getTemperature("Paris").map {
                (temp: (String,Float)) -> String in
                return String(temp.1)
            }
        }
        .bindTo(label.rx_text)
}

override func viewWillDisappear(animated: Bool) {
    timerDisposable.dispose()
    super.viewWillDisappear(animated)
}
```

# Error handling

**Task: if a temperature network request fails, display "--"**

```
func getTemperatureAsString(city: String) -> Observable<(String,String)> {
    return getTemperature(city)
        .map {
            (temp: (String,Float)) -> String in
            return (city, formattedTemperature(temp.1))
        }
        .catchErrorJustReturn((city, "--"))
}
```

# Error handling

**Task: if a temperature network request fails, display "--"**

```
let disposable = button.rx_tap
    .withLatestFrom(textField.rx_text)
    .flatMapLatest {
        (city: String) -> Observable<(String,String)> in
        return getTemperatureAsString(city)
    }
    .map {
        (temp: (String,String)) -> String in
        return "It's \(temp.1) in \(temp.0)"
    }
    .bindTo(label.rx_text)
```

# User interface patterns

# Driver

```
let disposable = button.rx_tap
    .withLatestFrom(textField.rx_text)
    .flatMapLatest {
        (city: String) -> Driver<String> in
        return getTemperature(city)
            .map { formattedTemperature($0.1) }
            .asDriver(onErrorJustReturn: "--")
    }
    .drive(label.rx_text)
```

# Action

- not technically part of RxSwift
- an important pattern for binding the UI
- `pod Action`
- a very useful pattern for MVVM

# Action

```
import Action

lazy var getTemperatureAction : CocoaAction = CocoaAction {
    [unowned self] in
    return self.getTemperatureAsString(self.textfield.text)
}

button.rx_action = getTemperatureAction
getTemperatureAction.elements.bindTo(label.rx_text)
```

# Architecture patterns

# Architecture patterns

- Expose all data to display as Observable sequences
- Use Action to wire up the UI whenever possible
- MVVM is a perfect fit for Rx

# Architecture patterns

- Decouple application logic from application *infrastructure*
- Storage, geolocation, network requests, image cache etc. are a good fit for insulation
- Makes replacing whole parts of the app easier
- Testing and mocking are easier too

# Summary

# Reactive programming

- Powerful way to express program logic
- Model the asynchronous propagation of change
- Eliminate state from your code
- Code is more testable
- RxSwift is a solid foundation
- Fast growing pool of users, add-ons and contributors (RxSwiftCommunity!)

# Links

- RxSwift source github.com/reactivex/rxswift
- Community projects github.com/RxSwiftCommunity
- Artsy's Eidolon app github.com/artsy/eidolon
- ReactiveX website reactivex.io
- RxMarbles rxmarbles.com

# Q & A