RxSwift

A Prescription for Async Programming

Ragunath Jawahar • @ragunathjawahar

REACTIVE

FUNCTIONAL

FUNCTIONAL

- Functions are first-class citizens
- Immutability
- No side-effects
- Referential transparency
- Declarative
- Favours concurrency

Example 1

```swift
func isDivisibleBy2(number: Int) -> Bool {
  return number % 2 == 0
}
```

Example 1

```
func isDivisibleBy2(number: Int) -> Bool {
  return number % 2 == 0
}
```

Example 2

```
integers
    .map { number in isDivisibleBy2(number) ? "Even" : "Odd‟ }
```

Example 2

```
integers
    .map { number in isDivisibleBy2(number) ? "Even" : "Odd" }
```

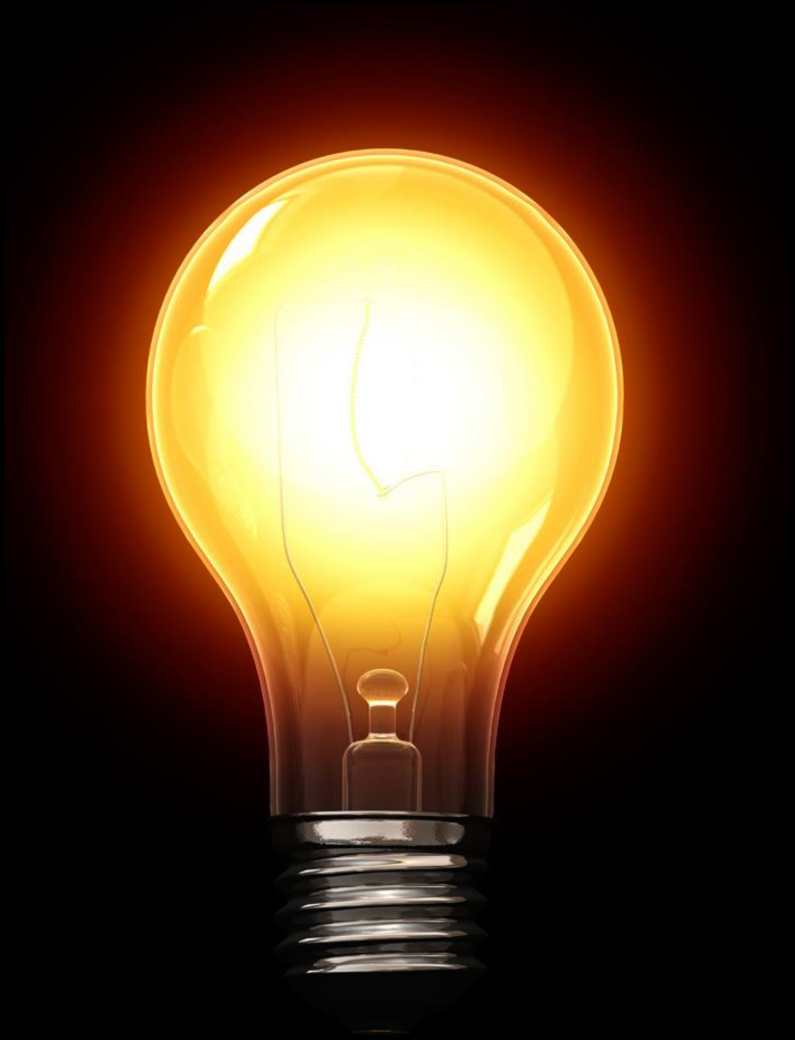
Example 2

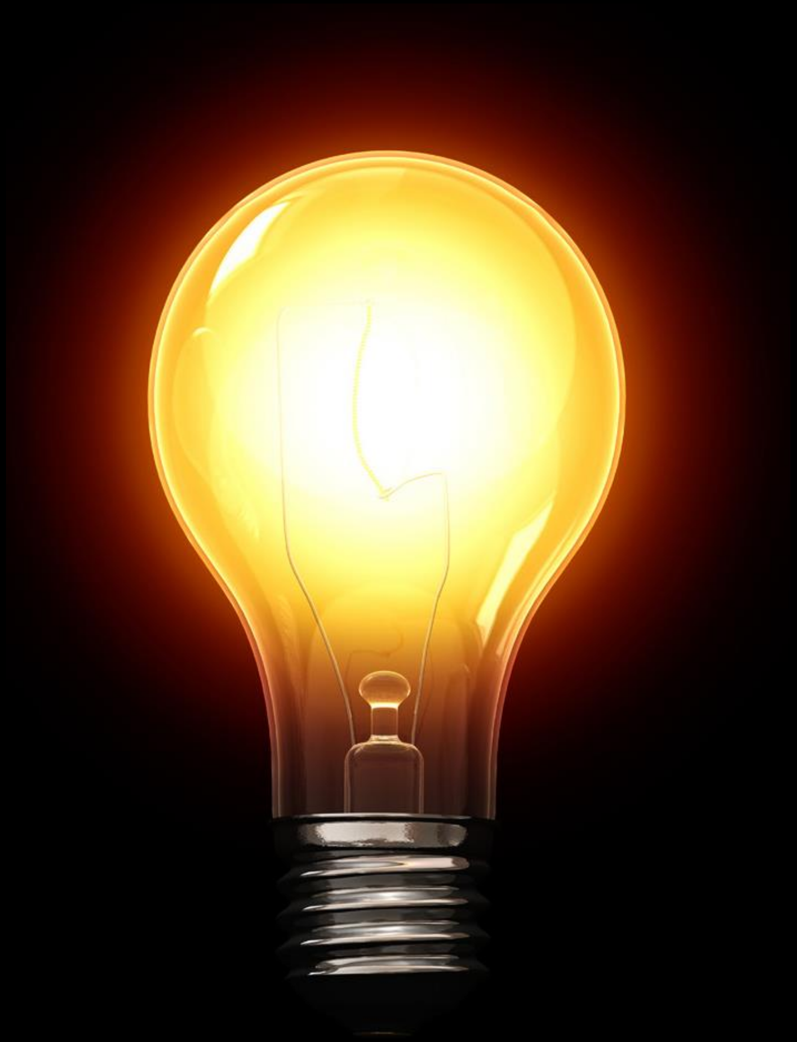
```
integers
    .map { number in isDivisibleBy2(number) ? "Even" : "Odd‟ }
```

Example 2

```
integers
    .map { number in isDivisibleBy2(number) ? "Even" : "Odd" }
```

REACTIVE





REACTIVE

- Responsive
- Resilient
- Elastic
- Message Driven

REACTIVE

Real World Examples

- Speed Traps
- Fire Alarms
- Airbags
- Autonomous Vehicles
- Excel
- GUI Systems

~ Callbacks ~

An imperative way to build reactive systems.

```swift
func login(_ username: String,
           _ password: String,
           success: (String) -> ()) {
    // Login logic...
}
```

```
login(“username”, “password”) { authToken in
    // Save token
}
```

```
login(“username”, “password”) { authToken in
    // Save token
}
```

```
login("username", "password",
    success: { authToken in
        // Save token
    },
    failure: { error in
        // Notify user
    }
);
```

```
login(“username”, “password”,
    success: { authToken in
        // Save token
    },
    failure: { error in
        // Notify user
    }
);
```

```
login(“username”, “password”,
    success: { authToken in
        // Save token
    },
    failure: { error in
        // Notify user
    }
);
```

RxSwift

A library for composing asynchronous and event-based code by using observable sequences and functional style operators, allowing for parameterized execution via schedulers.

RxSwift

- Primitives
- Operators

Primitive - Observable

- Produces events
- Usually lazy
- Can produce one, many or zero events
- Serialized access

Callback

```
button.addTarget(self,
    action: #selector(ViewController.doSomething),
    for: .touchUpInside
)
```

RxSwift

```
button.rx.tap.asObservable()
    .subscribe { _ in doSomething() }
```

Callback

```
button.addTarget(self,
    action: #selector(ViewController.doSomething),
    for: .touchUpInside
)
```

RxSwift

```
button.rx.tap.asObservable()
    .subscribe { _ in doSomething() }
```

Callback

```
button.addTarget(self,
    action: #selector(ViewController.buttonTap),
    for: .touchUpInside)

func buttonTap() {
    api.authenticate(username, password,
        success: { accessToken in
            // Save Token
        },
        failure: { error in
            // Show Error
        }
    )
}
```

Callback

```
button.addTarget(self,
    action: #selector(ViewController.buttonTap),
    for: .touchUpInside)

func buttonTap() {
    api.authenticate(username, password,
        success: { accessToken in
            // Save Token
        },
        failure: { error in
            // Show Error
        }
    )
}
```

Callback

```
button.addTarget(self,
    action: #selector(ViewController.buttonTap),
    for: .touchUpInside)

func buttonTap() {
    api.authenticate(username, password,
        success: { accessToken in
            // Save Token
        },
        failure: { error in
            // Show Error
        }
    )
}
```

Callback

```
button.addTarget(self,
    action: #selector(ViewController.buttonTap),
    for: .touchUpInside)

func buttonTap() {
    api.authenticate(username, password,
        success: { accessToken in
            // Save Token
        },
        failure: { error in
            // Show Error
        }
    )
}
```

Callback

```
button.addTarget(self,
    action: #selector(ViewController.buttonTap),
    for: .touchUpInside)

func buttonTap() {
    api.authenticate(username, password,
        success: { accessToken in
            // Save Token
        },
        failure: { error in
            // Show Error
        }
    )
}
```

Callback

```
button.addTarget(self,
    action: #selector(ViewController.buttonTap),
    for: .touchUpInside)

func buttonTap() {
    api.authenticate(username, password,
        success: { accessToken in
            // Save Token
        },
        failure: { error in
            // Show Error
        }
    )
}
```

Callback

```
button.addTarget(self,
    action: #selector(ViewController.buttonTap),
    for: .touchUpInside)

func buttonTap() {
    api.authenticate(username, password,
        success: { accessToken in
            // Save Token
        },
        failure: { error in
            // Show Error
        }
    )
}
```

Callback

```
button.addTarget(self,
    action: #selector(ViewController.buttonTap),
    for: .touchUpInside)

func buttonTap() {
    api.authenticate(username, password,
        success: { accessToken in
            // Save Token
        },
        failure: { error in
            // Show Error
        }
    )
}
```

Callback

```
button.addTarget(self,
    action: #selector(ViewController.buttonTap),
    for: .touchUpInside)

func buttonTap() {
    api.authenticate(username, password,
        success: { accessToken in
            // Save Token
        },
        failure: { error in
            // Show Error
        }
    )
}
```

RxSwift

```
button.rx.tap.asObservable()
    .flatMap { _ in api.authenticate(username, password) }
    .catchErrorJustReturn(nil)
    .subscribe(onNext: { authToken in
      if (authToken != nil) saveToken(authToken) else showError()
    })
```

RxSwift

```
button.rx.tap.asObservable()
    .flatMap { _ in api.authenticate(username, password) }
    .catchErrorJustReturn(nil)
    .subscribe(onNext: { authToken in
      if (authToken != nil) saveToken(authToken) else showError()
    })
```

RxSwift

```
button.rx.tap.asObservable()
    .flatMap { _ in api.authenticate(username, password) }
    .catchErrorJustReturn(nil)
    .subscribe(onNext: { authToken in
      if (authToken != nil) saveToken(authToken) else showError()
    })
```

RxSwift

```
button.rx.tap.asObservable()
    .flatMap { _ in api.authenticate(username, password) }
    .catchErrorJustReturn(nil)
    .subscribe(onNext: { authToken in
      if (authToken != nil) saveToken(authToken) else showError()
    })
```

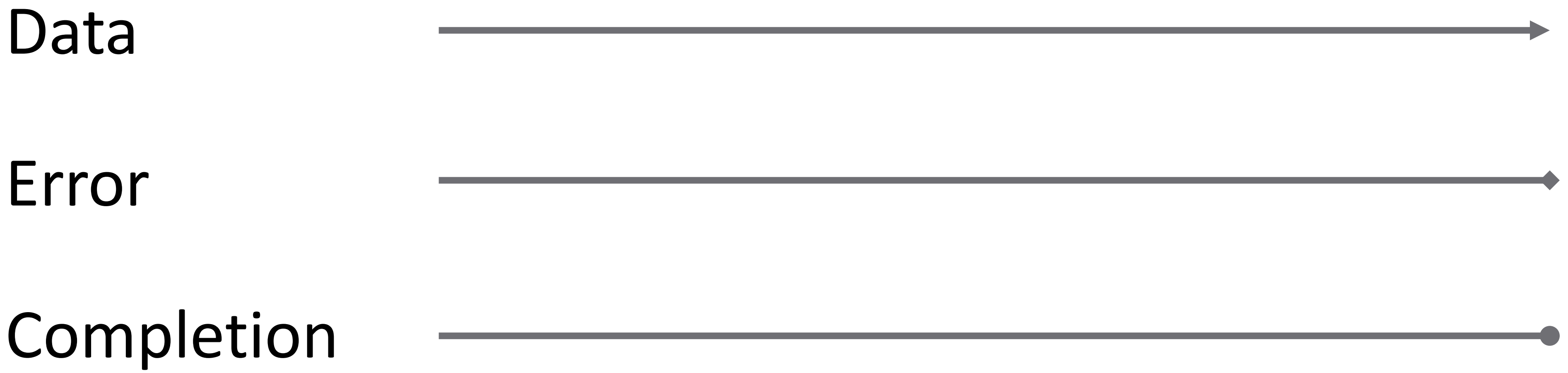
RxSwift

```
button.rx.tap.asObservable()
    .flatMap { _ in api.authenticate(username, password) }
    .catchErrorJustReturn(nil)
    .subscribe(onNext: { authToken in
      if (authToken != nil) saveToken(authToken) else showError()
    })
```

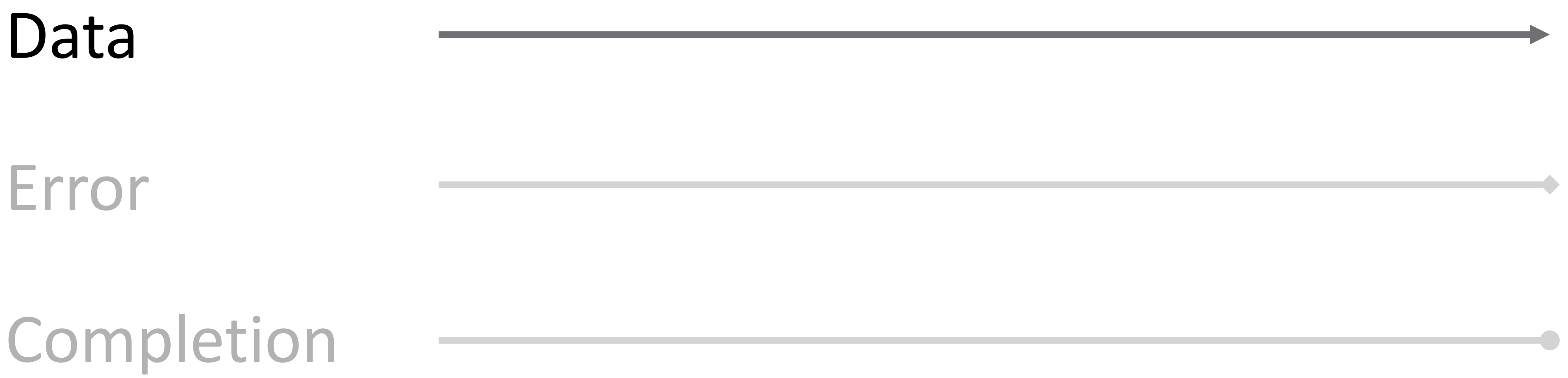
RxSwift

```
button.rx.tap.asObservable()
    .flatMap { _ in api.authenticate(username, password) }
    .catchErrorJustReturn(nil)
    .subscribeOn(backgroundScheduler)
    .observeOn(MainScheduler.instance)
    .subscribe(onNext: { authToken in
      if (authToken != nil) saveToken(authToken) else showError()
    })
```

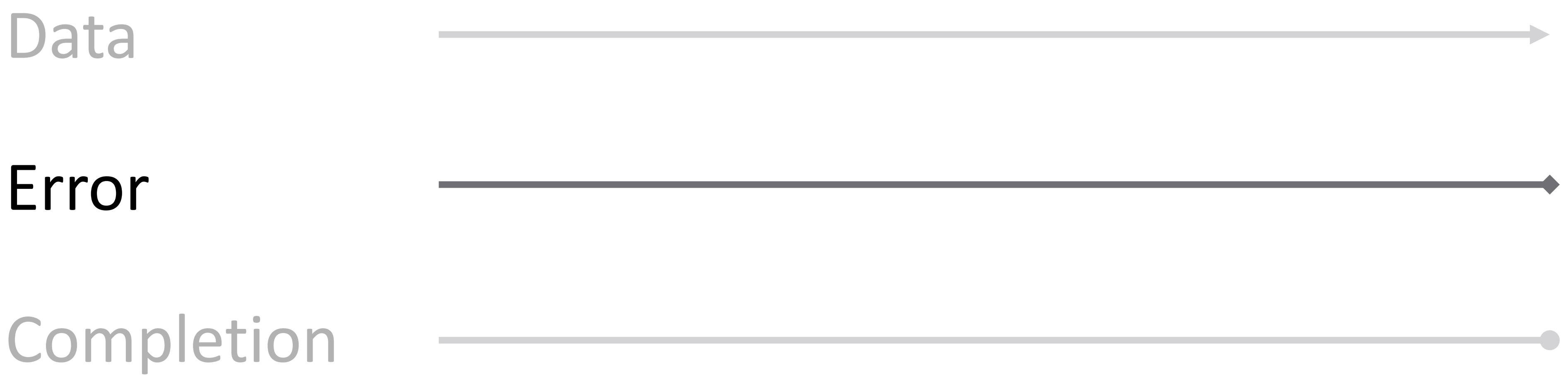
Observable - Anatomy



Observable - Anatomy



Observable - Anatomy

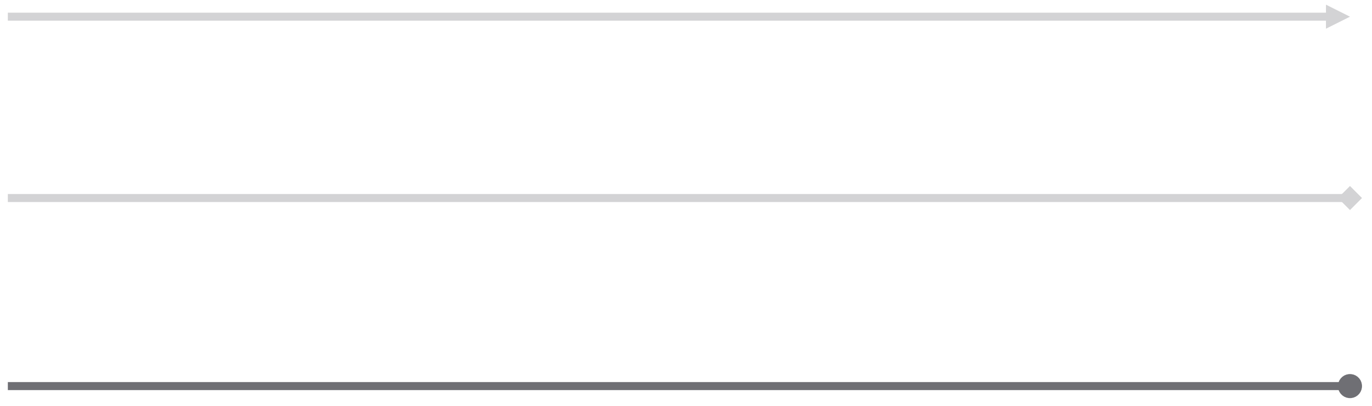


Observable - Anatomy

Data

Error

Completion



Primitive - Observer

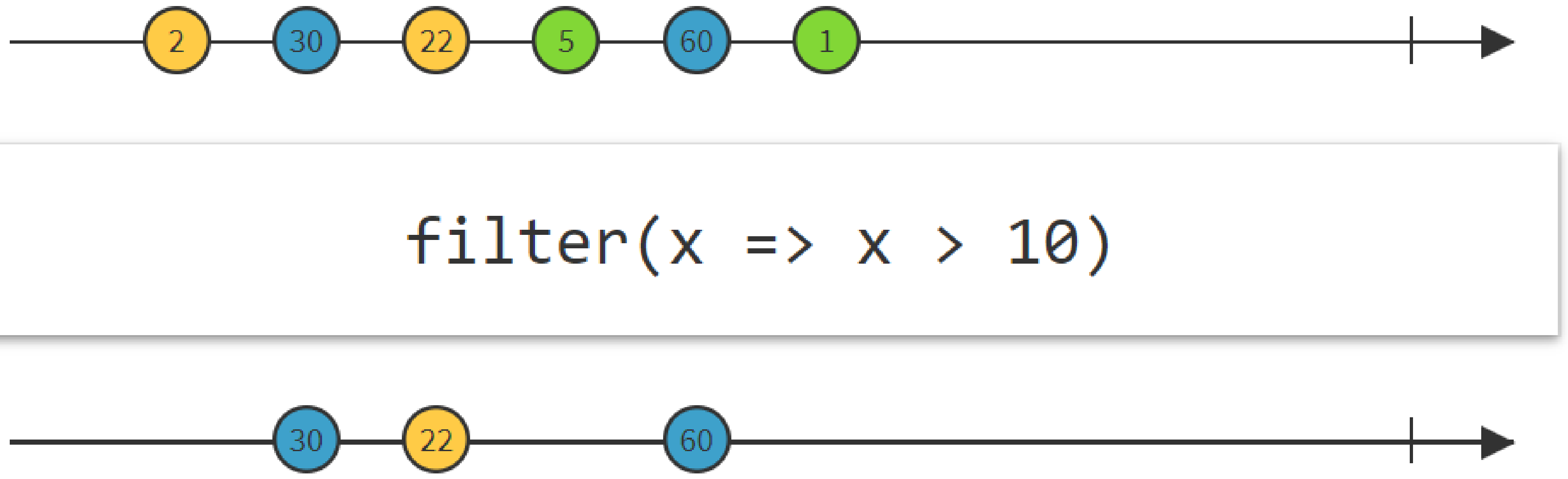
- `onNext(T)`
- `onError(Error)`
- `onComplete()`
- `onDisposed()`

RxSwift - Operators

- Single Observable
- Multiple Observable
- Higher-Order Observables
- Finite and infinite Observables (any of the above)

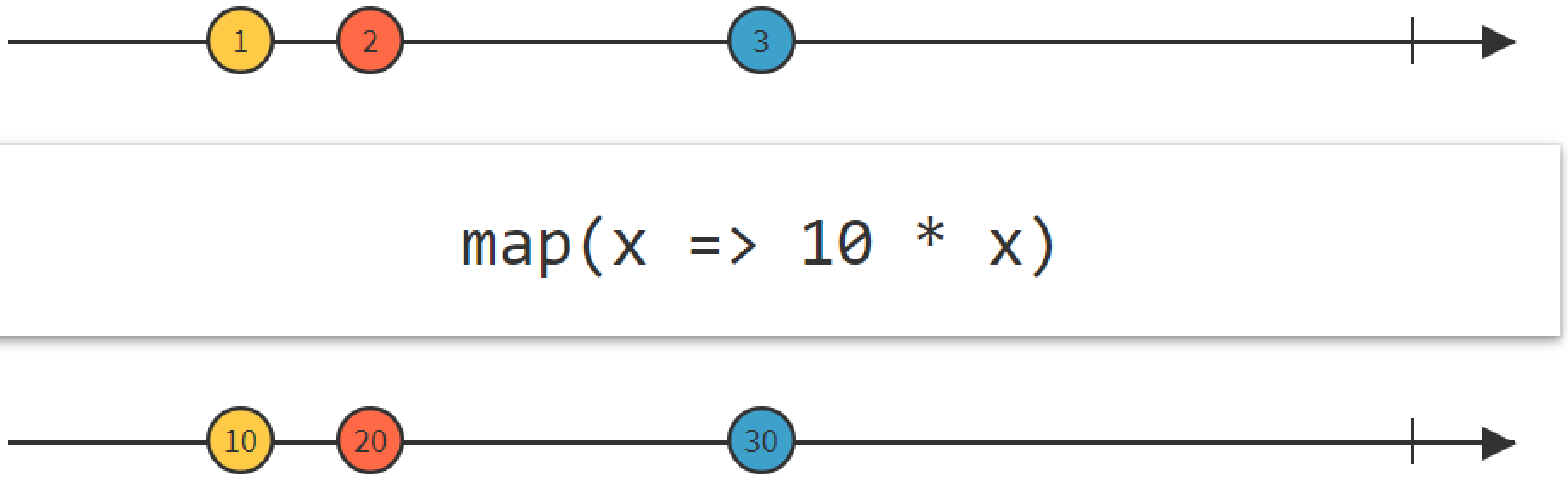
Marble Diagrams

filter



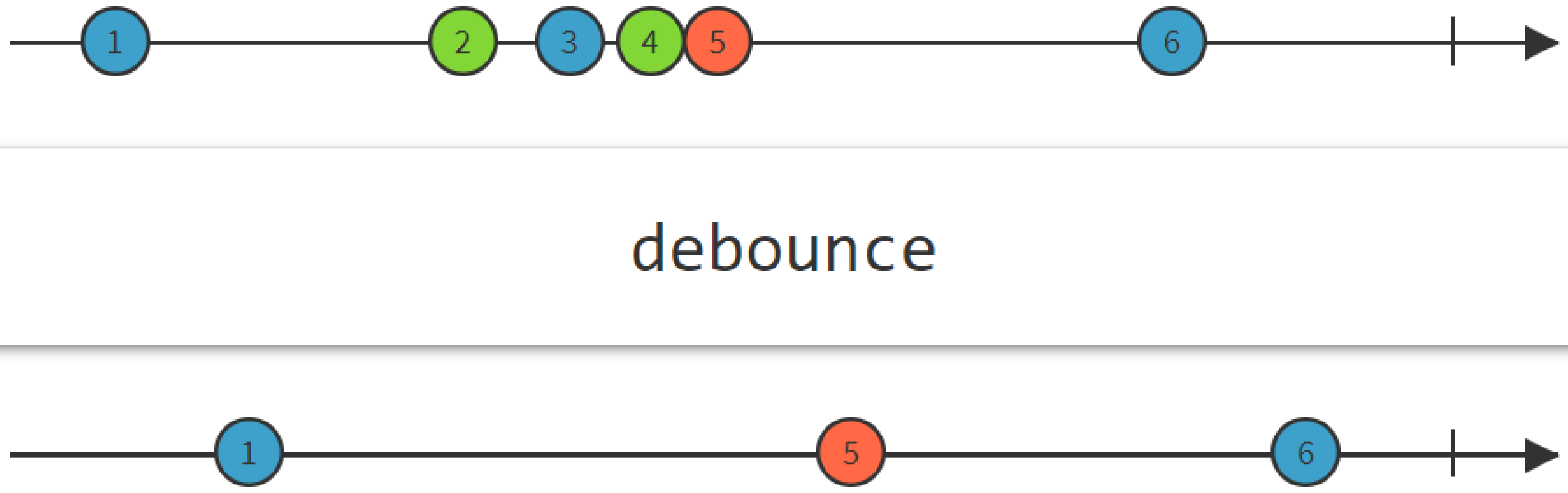
```
filter(x => x > 10)
```

map

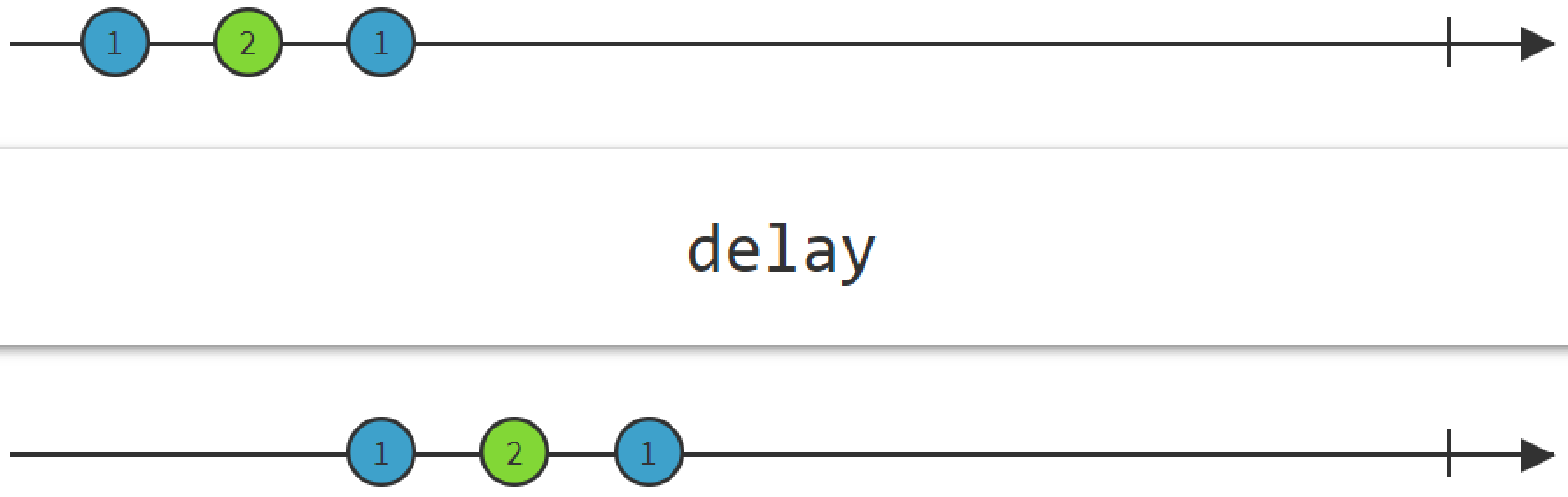


```
map(x => 10 * x)
```

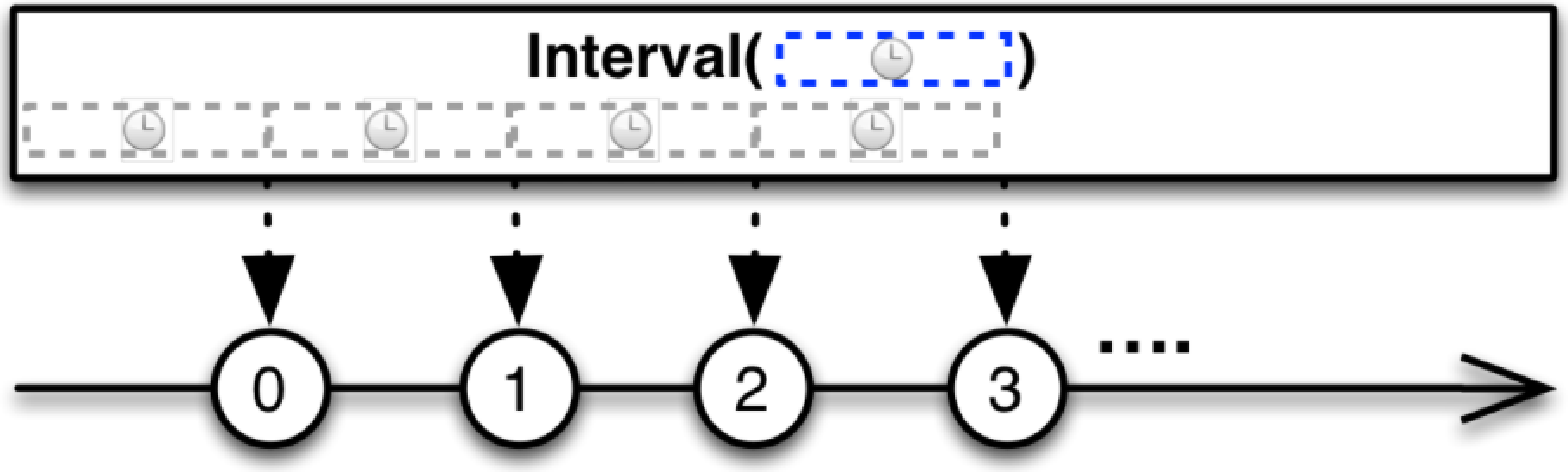
debounce



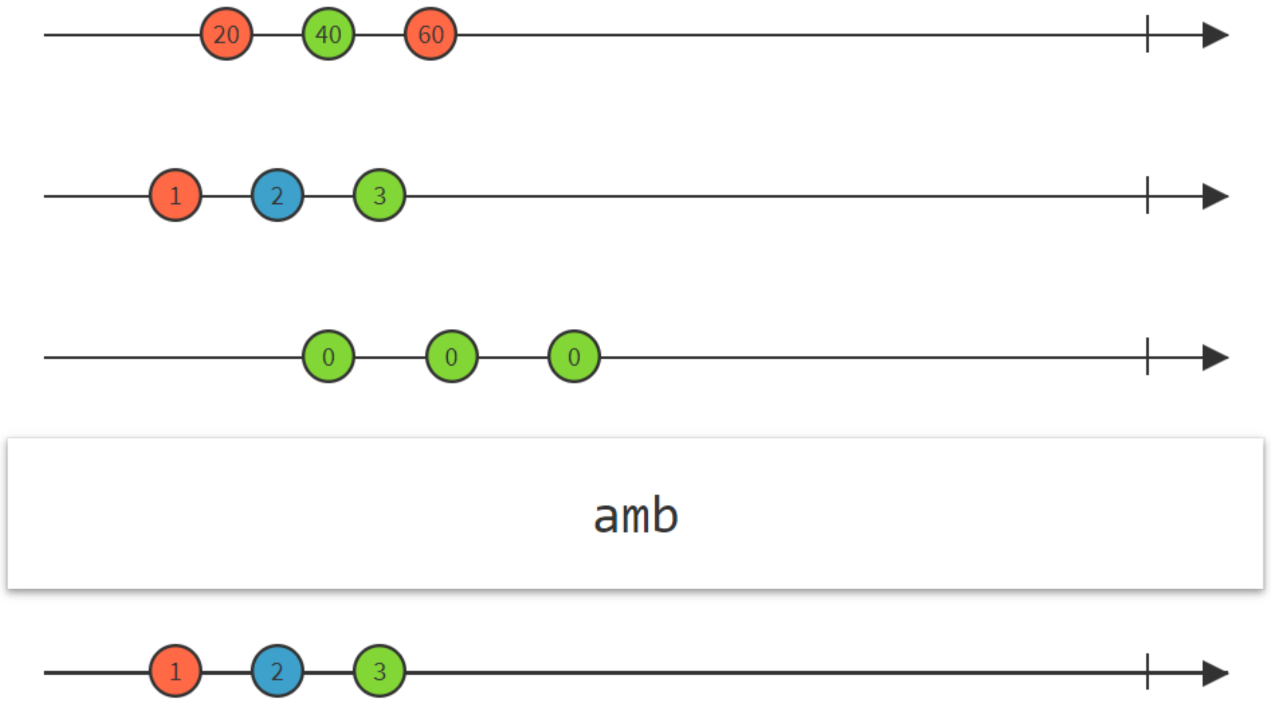
delay



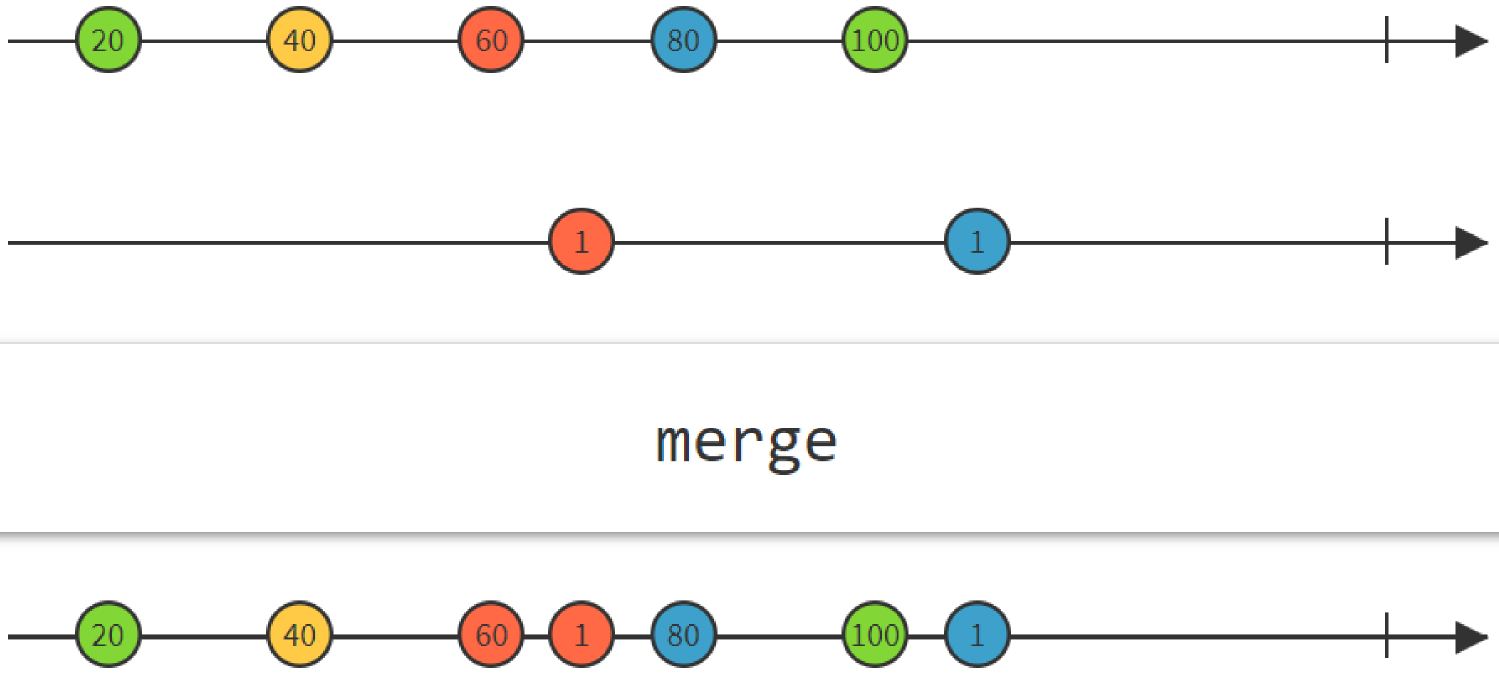
interval



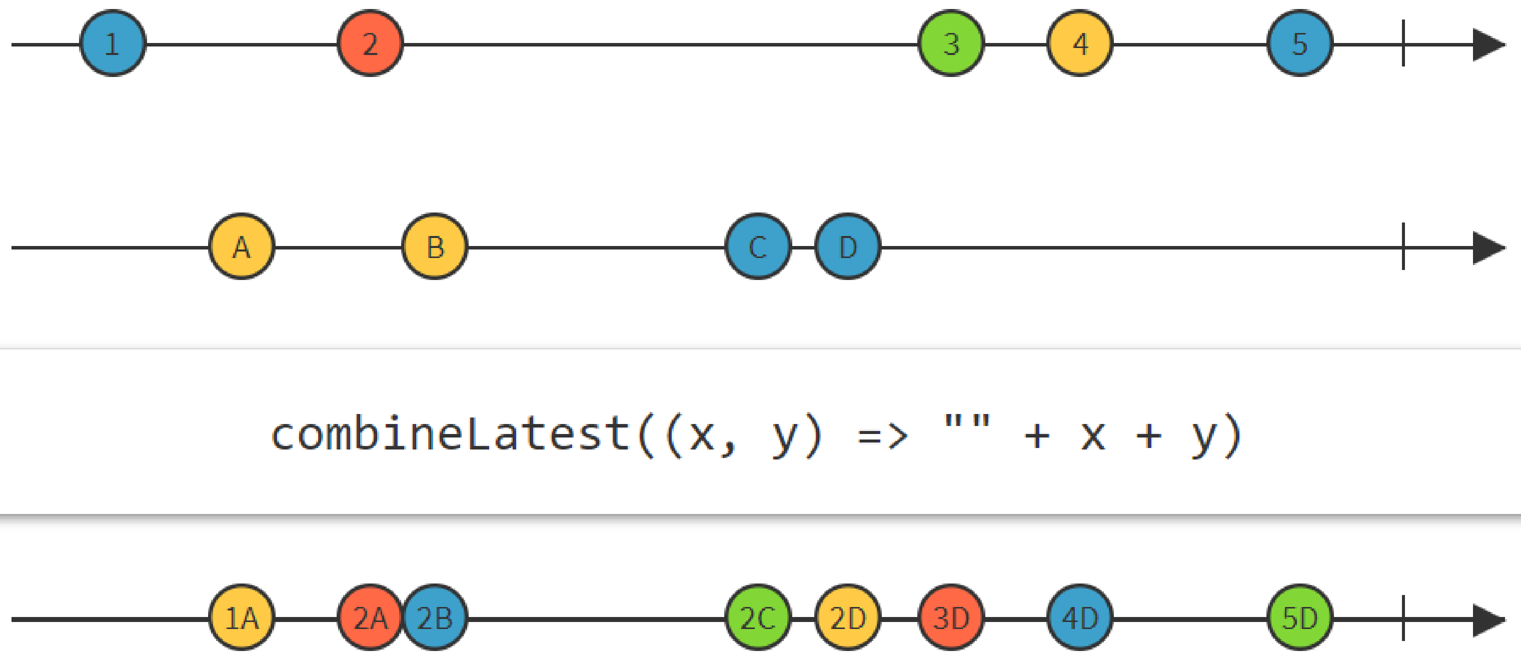
amb



merge

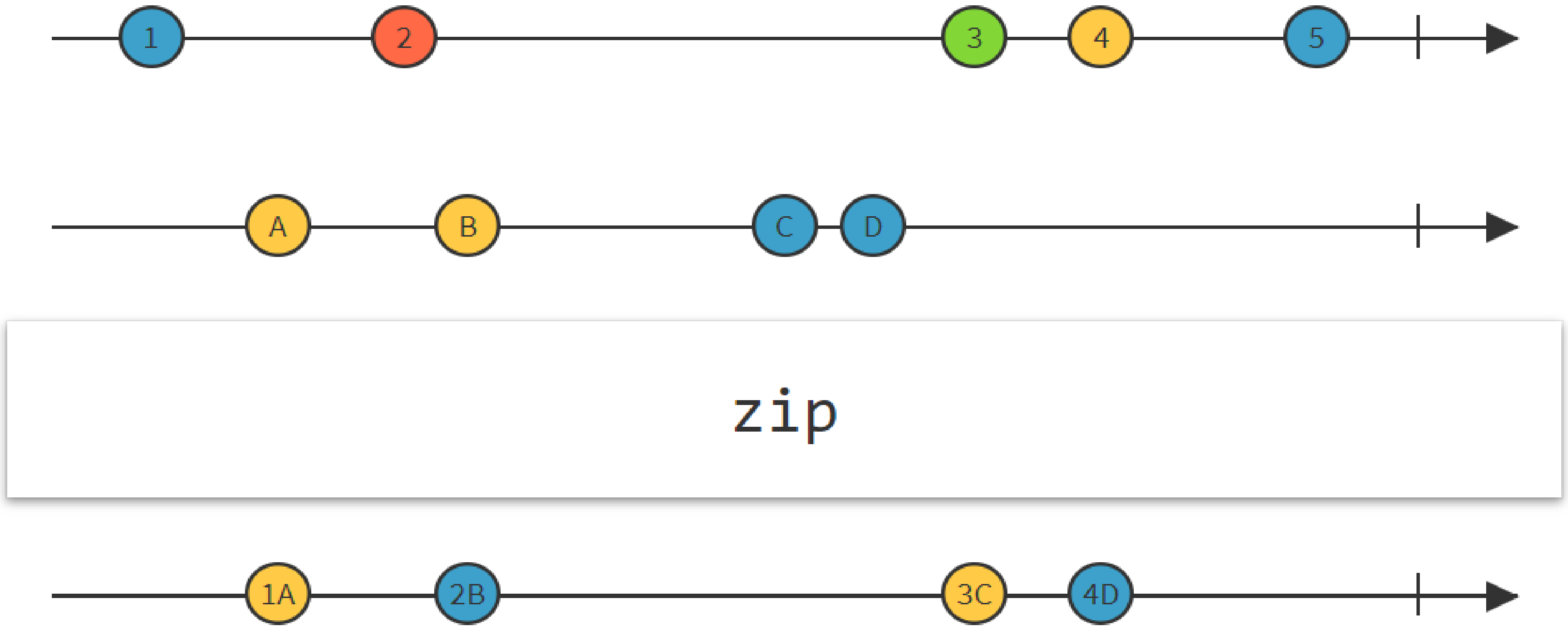


combineLatest

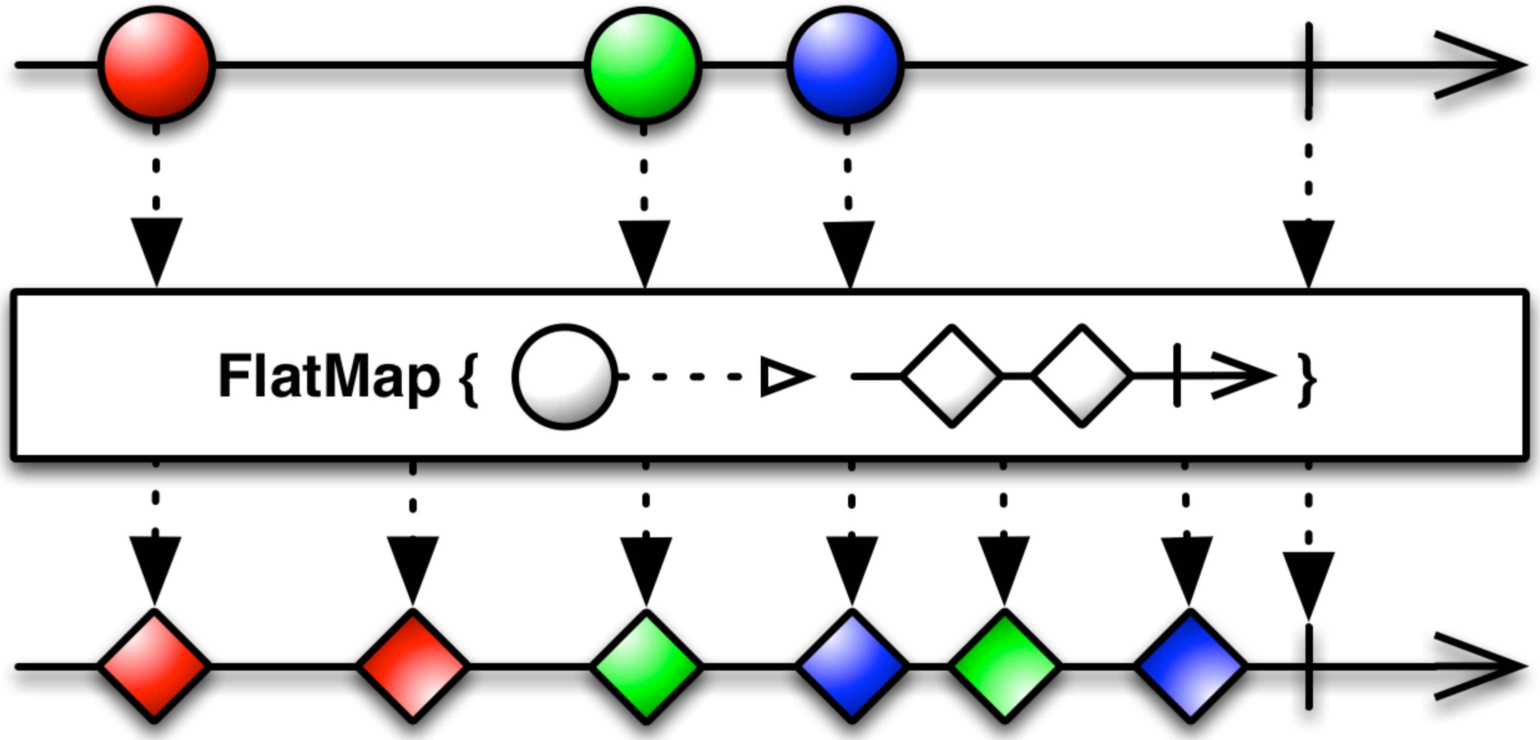


```
combineLatest((x, y) => "" + x + y)
```

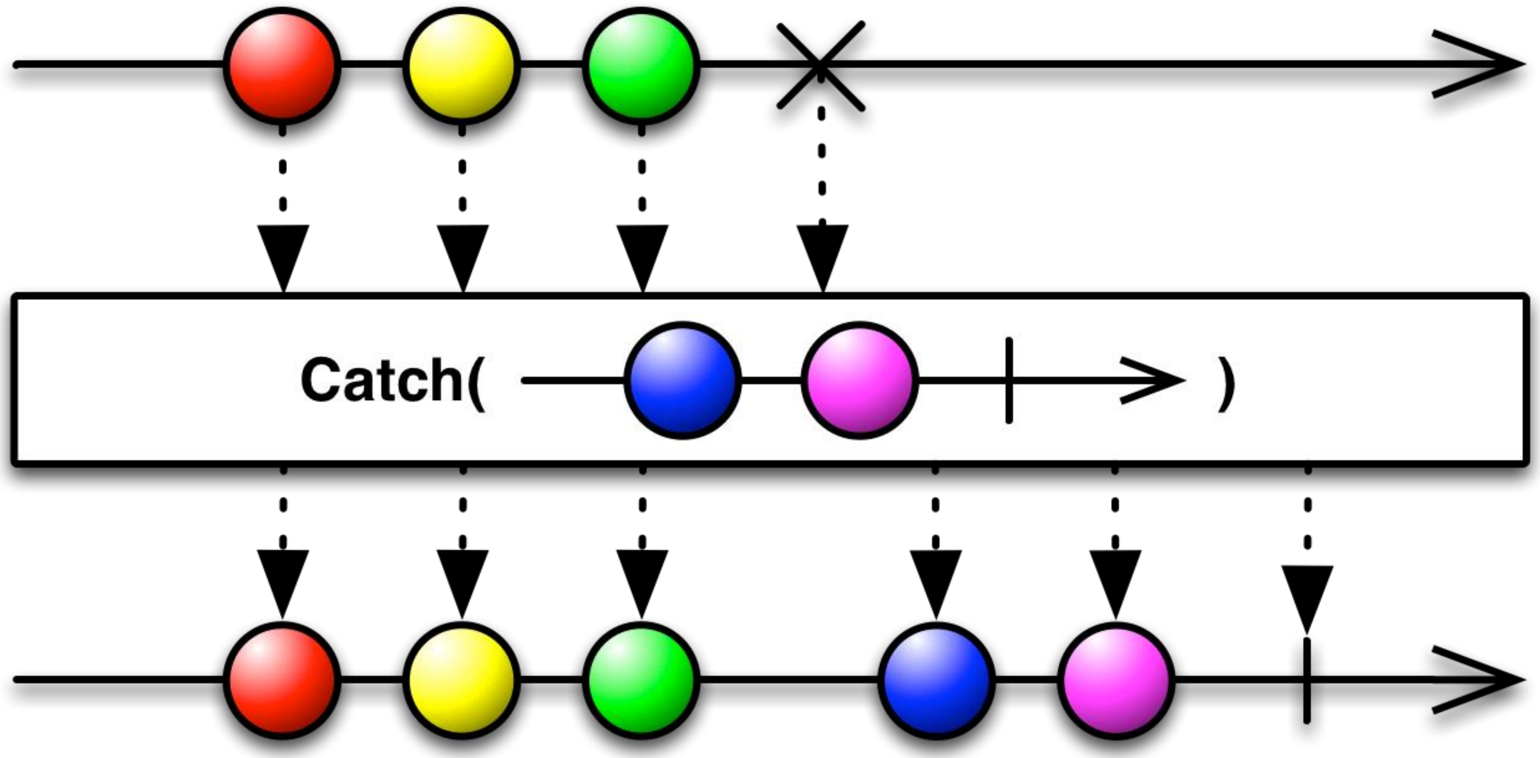
zip



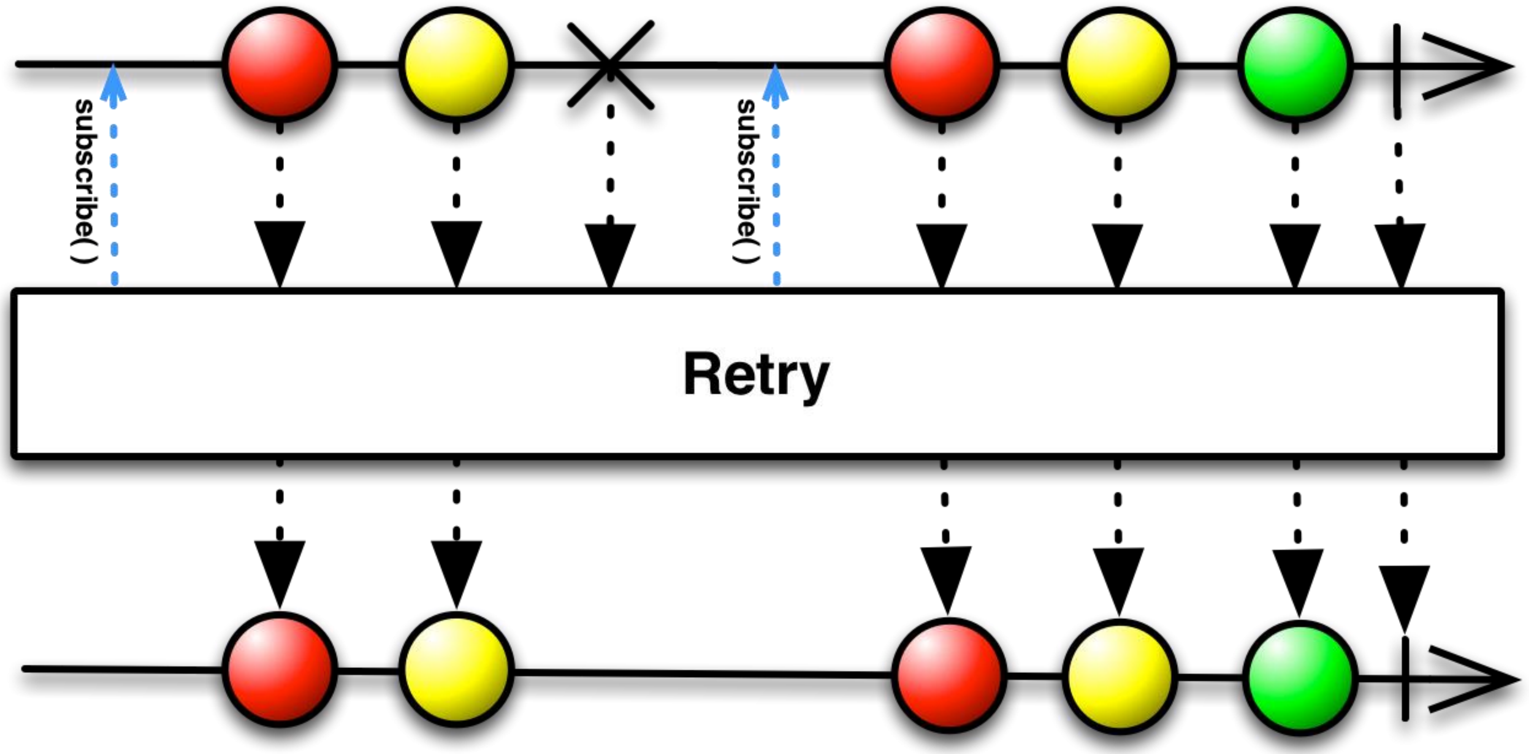
flatMap



catch*



retry*



Error Handling

- Exceptions are normal
- Error handling operators
- Retry operators

Declarative Threading

- Using Schedulers
- `subscribeOn(Scheduler)`
- `observeOn(Scheduler)`

Debugging

- Could be a tricky
- Use `doOn*` operators to inspect the pipeline

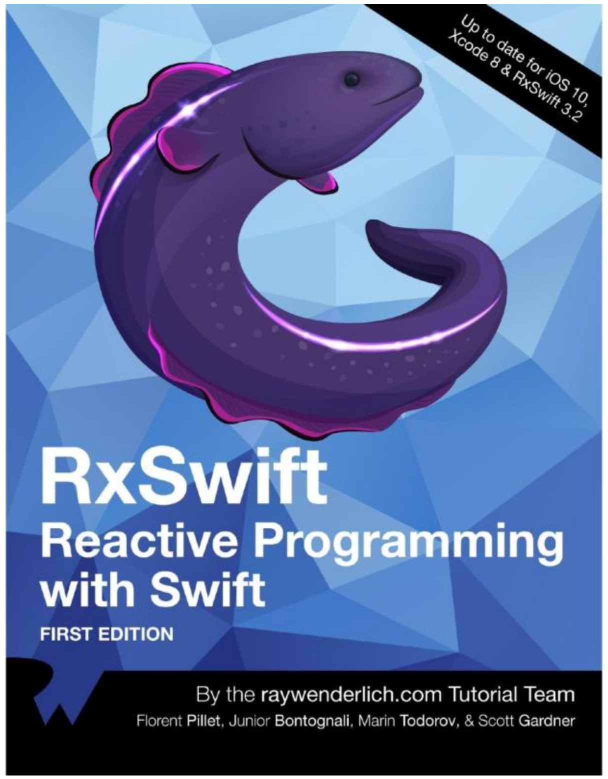
Testing

- Cuckoo
- Subjects / Relays

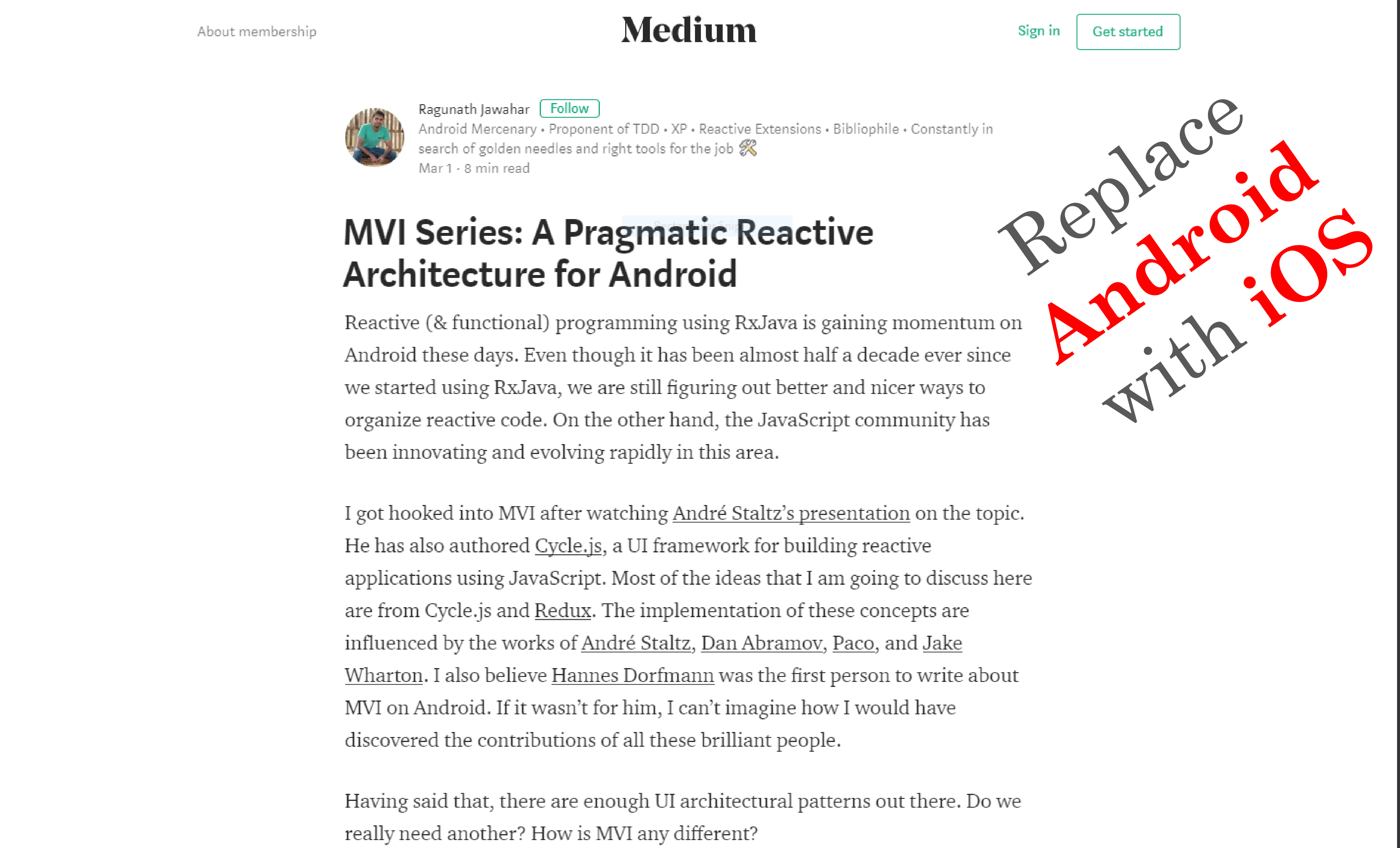
Summary

- Reactive Functional Programming
- Callbacks vs. Streams
- Observable & Observer Contract
- Operators

Further Reading


Up to date for iOS 10, Xcode 8 & RxSwift 3.2
RxSwift
Reactive Programming with Swift
FIRST EDITION
By the raywenderlich.com Tutorial Team
Florent Pillet, Junior Bontognali, Marin Todorov, & Scott Gardner

André Staltz



About membership

Medium

Sign in Get started

Ragunath Jawahar Follow

Android Mercenary • Proponent of TDD • XP • Reactive Extensions • Bibliophile • Constantly in search of golden needles and right tools for the job

Mar 1 · 8 min read

MVI Series: A Pragmatic Reactive Architecture for Android

Reactive (& functional) programming using RxJava is gaining momentum on Android these days. Even though it has been almost half a decade ever since we started using RxJava, we are still figuring out better and nicer ways to organize reactive code. On the other hand, the JavaScript community has been innovating and evolving rapidly in this area.

I got hooked into MVI after watching André Staltz's presentation on the topic. He has also authored Cycle.js, a UI framework for building reactive applications using JavaScript. Most of the ideas that I am going to discuss here are from Cycle.js and Redux. The implementation of these concepts are influenced by the works of André Staltz, Dan Abramov, Paco, and Jake Wharton. I also believe Hannes Dorfmann was the first person to write about MVI on Android. If it wasn't for him, I can't imagine how I would have discovered the contributions of all these brilliant people.

Having said that, there are enough UI architectural patterns out there. Do we really need another? How is MVI any different?

Replace **Android** with **iOS**

@ragunathjawahar

Twitter / Medium / GitHub