Reactive Programing in the Real World

Vincent Pradeilles - equensWorldline

Reactive?

Reactive ≠ Passive

Passive Programing

```
class ViewController: UIViewController {
    var label = UILabel()
    var model = Model()

    override func viewDidLoad() {
        model.fetchData(destination: label)
    }
}

class Model {
    func fetchData(destination: UILabel) {
        let data = "Foo"

        destination.text = data
    }
}
```

Passive Programing

```
class ViewController: UIViewController {
    var label = UILabel()
    var model = Model()

    override func viewDidLoad() {
        model.fetchData(destination: label)
    }
}

class Model {
    func fetchData(destination: UILabel) {
        let data = "Foo"

        destination.text = data
    }
}
```

Reactive Programing

```
class ViewController: UIViewController {
    var label = UILabel()
    var model = Model()

    override func viewDidLoad() {
        model.fetchData(completionHandler: { data in
            self.label.text = data
        })
    }
}

class Model {
    func fetchData(completionHandler: @escaping (String) -> Void) {
        let data = "Foo"

        completionHandler(data)
    }
}
```

Reactive Programing

```
class ViewController: UIViewController {
    var label = UILabel()
    var model = Model()

    override func viewDidLoad() {
        model.fetchData(completionHandler: { data in
            self.label.text = data
        })
    }
}

class Model {
    func fetchData(completionHandler: @escaping (String) -> Void) {
        let data = "Foo"

        completionHandler(data)
    }
}
```

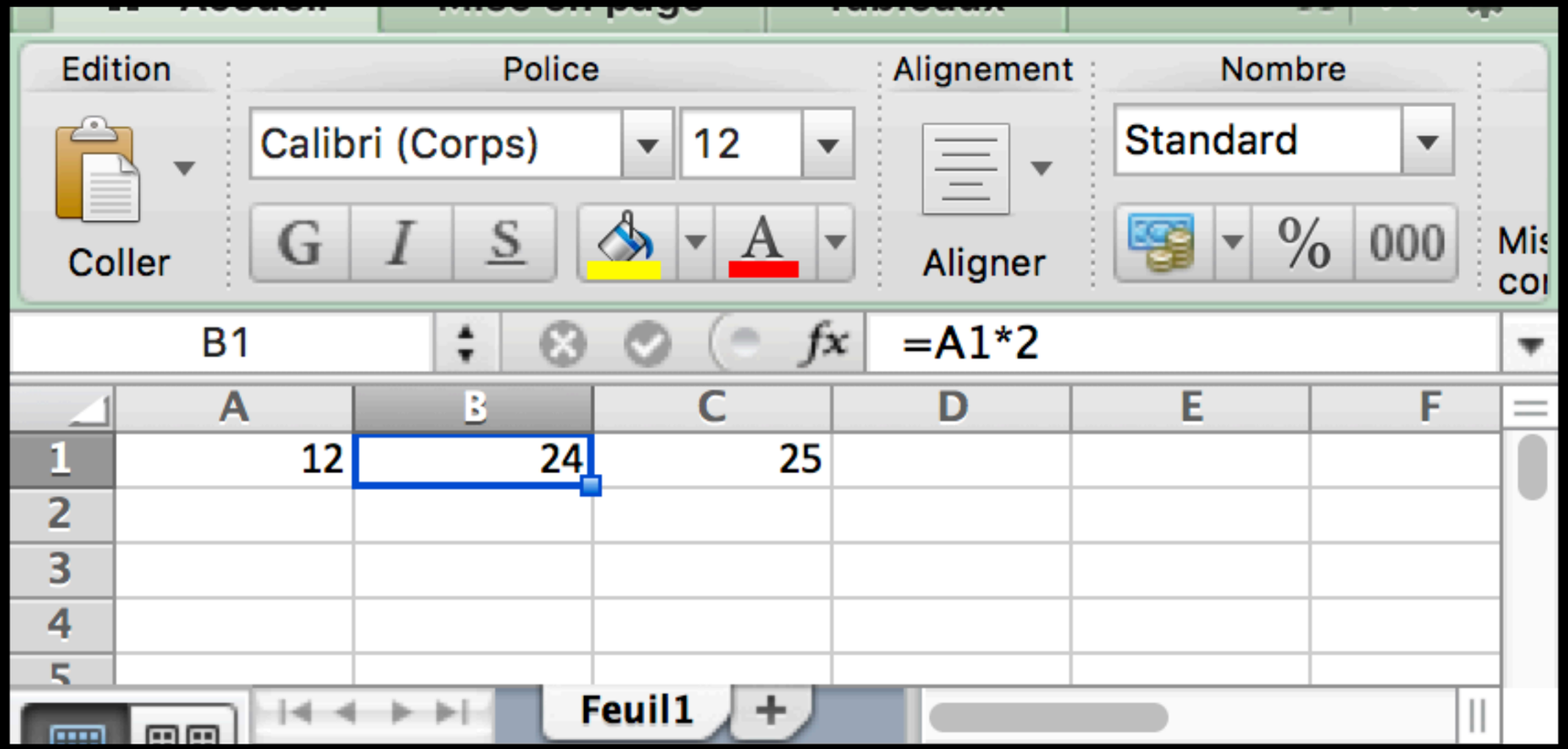
A program is said to be reactive if it ***explicitly subscribes*** *to future events, in order to* ***process them*** *once they have been emitted.*

Great news:
you're already doing it!

Callbacks are reactive!

```
$("#btn_1").click(function() {
  alert("Btn 1 Clicked");
});
```

Excel is reactive!



Advantages of Reactive style

Advantages of Reactive style

- Architecturally sound

Advantages of Reactive style

- Architecturally sound
  - improves module coherence

Advantages of Reactive style

- Architecturally sound
  - improves module coherence
  - strengthens separation of concerns

Advantages of Reactive style

- Architecturally sound
  - improves module coherence
  - strengthens separation of concerns
- Reliable way to handle asynchronous events

Advantages of Reactive style

- Architecturally sound
  - improves module coherence
  - strengthens separation of concerns
- Reliable way to handle asynchronous events
  - network calls

Advantages of Reactive style

- Architecturally sound
  - improves module coherence
  - strengthens separation of concerns
- Reliable way to handle asynchronous events
  - network calls
  - costly computations

Advantages of Reactive style

- Architecturally sound
  - improves module coherence
  - strengthens separation of concerns
- Reliable way to handle asynchronous events
  - network calls
  - costly computations
  - UI interaction

Drawbacks of naive Reactive style?

Callback Hell

```
const verifyUser = function(username, password, callback){
   dataBase.verifyUser(username, password, (error, userInfo) => {
       if (error) {
           callback(error)
       } else {
           dataBase.getRoles(username, (error, roles) => {
               if (error) {
                   callback(error)
               } else {
                   dataBase.logAccess(username, (error) => {
                       if (error) {
                           callback(error);
                       } else {
                           callback(null, userInfo, roles);
                       }
                   })
               }
           })
       }
   })
};
```

Callback Hell

```
const verifyUser = function(username, password, callback){
   dataBase.verifyUser(username, password, (error, userInfo) => {
       if (error) {
           callback(error)
       } else {
           dataBase.getRoles(username, (error, roles) => {
               if (error) {
                   callback(error)
               } else {
                   dataBase.logAccess(username, (error) => {
                       if (error) {
                           callback(error);
                       } else {
                           callback(null, userInfo, roles);
                       }
                   })
               }
           })
       }
   })
};
```

Callback Hell

```
const verifyUser = function(username, password, callback){
   dataBase.verifyUser(username, password, (error, userInfo) => {
       if (error) {
           callback(error)
       } else {
           dataBase.getRoles(username, (error, roles) => {
               if (error) {
                   callback(error)
               } else {
                   dataBase.logAccess(username, (error) => {
                       if (error) {
                           callback(error);
                       } else {
                           callback(null, userInfo, roles);
                       }
                   })
               }
           })
       }
   })
};
```

Callback Hell

```
const verifyUser = function(username, password, callback){
   dataBase.verifyUser(username, password, (error, userInfo) => {
       if (error) {
           callback(error)
       } else {
           dataBase.getRoles(username, (error, roles) => {
               if (error) {
                   callback(error)
               } else {
                   dataBase.logAccess(username, (error) => {
                       if (error) {
                           callback(error);
                       } else {
                           callback(null, userInfo, roles);
                       }
                   })
               }
           })
       }
   })
};
```

Callbacks considered harmful[1]

Callbacks are pieces of code called at arbitrary times.

That makes them powerful tools.

But it also makes them very similar to a goto.

---

[1] Callbacks as our Generations' Go To Statement: http://tirania.org/blog/archive/2013/Aug-15.html

Structured Programing

	Naive	Structured
Passive Programing	`goto`	`while`, `for`, `if`

Structured Programing

	Naive	Structured
Passive Programing	`goto`	`while`, `for`, `if`
Reactive Programing	raw callbacks	**?**

A program is said to be reactive if it explicitly subscribes to future events, in order to process them once they have been emitted, ***in a scalable and maintainable way****.*

ReactiveX (Rx)

***Standardized API**, with several **language-specific implementations**: RxJS, RxJava, RxSwift, RxKotlin, etc.*

Combines the best ideas from the ***Observer pattern****, the* ***Iterator pattern****, and* ***functional programing***

Fundamental API

Observer on steroids

```
public class Observable<Element> {
    public func subscribe<Observer: ObserverType>(_ observer: Observer) -> Disposable
        where Observer.Element == Element
}

public protocol ObserverType {
    associatedtype Element

    func on(_ event: Event<Element>)
}

public enum Event<Element> {
    case next(Element)
    case error(Swift.Error)
    case completed
}
```

Observer on steroids

```
public class Observable<Element> {
    public func subscribe<Observer: ObserverType>(_ observer: Observer) -> Disposable
        where Observer.Element == Element
}

public protocol ObserverType {
    associatedtype Element

    func on(_ event: Event<Element>)
}

public enum Event<Element> {
    case next(Element)
    case error(Swift.Error)
    case completed
}
```

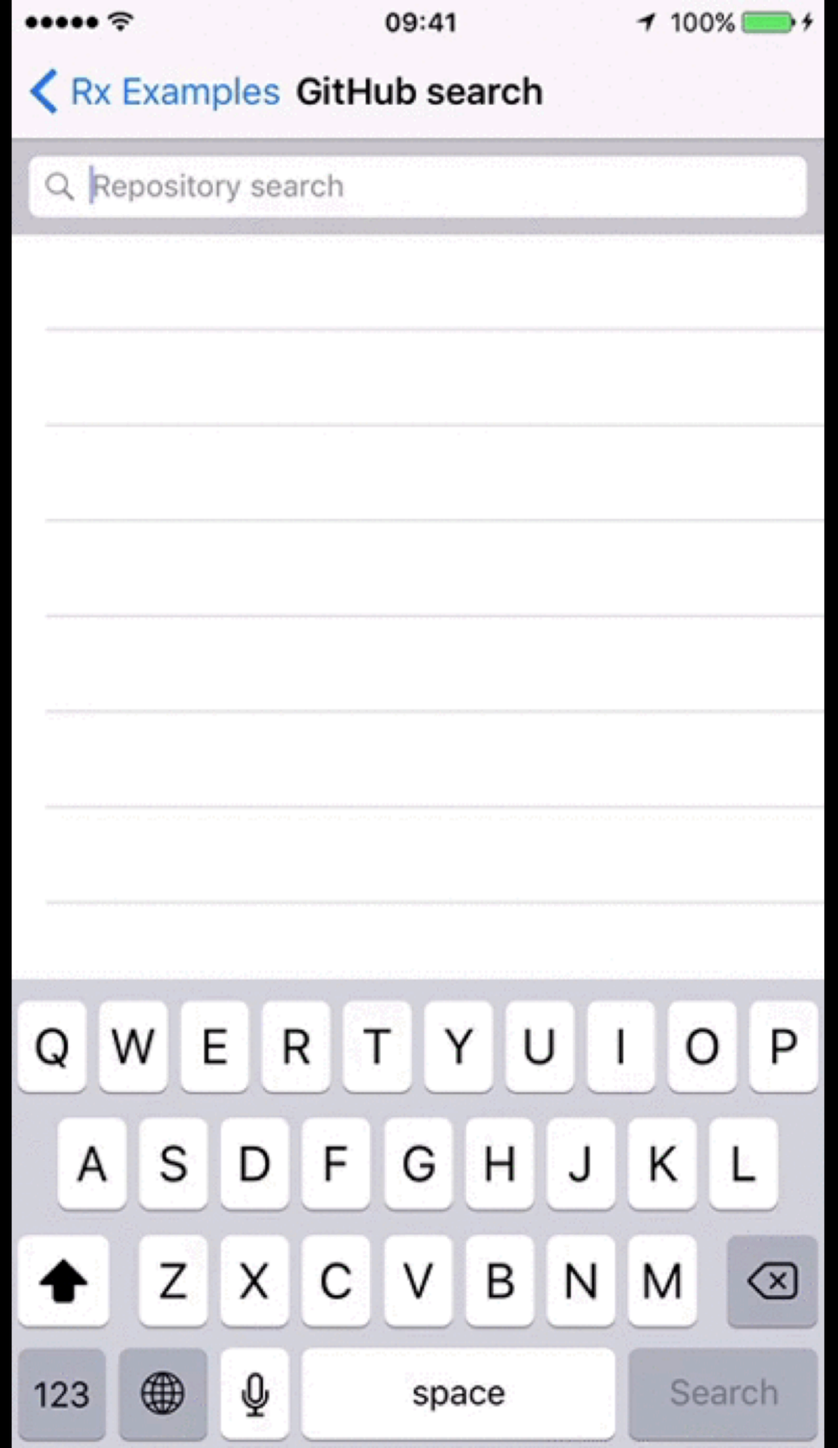
Observer on steroids

```
public class Observable<Element> {
    public func subscribe<Observer: ObserverType>(_ observer: Observer) -> Disposable
        where Observer.Element == Element
}

public protocol ObserverType {
    associatedtype Element

    func on(_ event: Event<Element>)
}

public enum Event<Element> {
    case next(Element)
    case error(Swift.Error)
    case completed
}
```

Observer on steroids

```
public class Observable<Element> {
    public func subscribe<Observer: ObserverType>(_ observer: Observer) -> Disposable
        where Observer.Element == Element
}

public protocol ObserverType {
    associatedtype Element

    func on(_ event: Event<Element>)
}

public enum Event<Element> {
    case next(Element)
    case error(Swift.Error)
    case completed
}
```

Example of use


09:41
100%
Rx Examples
GitHub search
Repository search
Q W E R T Y U I O P
A S D F G H J K L
Z X C V B N M
123
space
Search

```
let searchResults = searchBar.rx.text.orEmpty
    .throttle(0.3, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .flatMapLatest { query -> Observable<[Repository]> in
        if query.isEmpty {
            return .just([])
        }
        return searchGitHub(query)
            .catchErrorJustReturn([])
    }
    .observeOn(MainScheduler.instance)

searchResults
    .bind(to: tableView.rx.items(cellIdentifier: "Cell")) {
        (index, repository: Repository, cell) in
        cell.textLabel?.text = repository.name
        cell.detailTextLabel?.text = repository.url
    }
    .disposed(by: disposeBag)
```

```swift
let searchResults = searchBar.rx.text.orEmpty
    .throttle(0.3, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .flatMapLatest { query -> Observable<[Repository]> in
        if query.isEmpty {
            return .just([])
        }
        return searchGitHub(query)
            .catchErrorJustReturn([])
    }
    .observeOn(MainScheduler.instance)

searchResults
    .bind(to: tableView.rx.items(cellIdentifier: "Cell")) {
        (index, repository: Repository, cell) in
        cell.textLabel?.text = repository.name
        cell.detailTextLabel?.text = repository.url
    }
    .disposed(by: disposeBag)
```

```
let searchResults = searchBar.rx.text.orEmpty
    .throttle(0.3, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .flatMapLatest { query -> Observable<[Repository]> in
        if query.isEmpty {
            return .just([])
        }
        return searchGitHub(query)
            .catchErrorJustReturn([])
    }
    .observeOn(MainScheduler.instance)

searchResults
    .bind(to: tableView.rx.items(cellIdentifier: "Cell")) {
        (index, repository: Repository, cell) in
        cell.textLabel?.text = repository.name
        cell.detailTextLabel?.text = repository.url
    }
    .disposed(by: disposeBag)
```

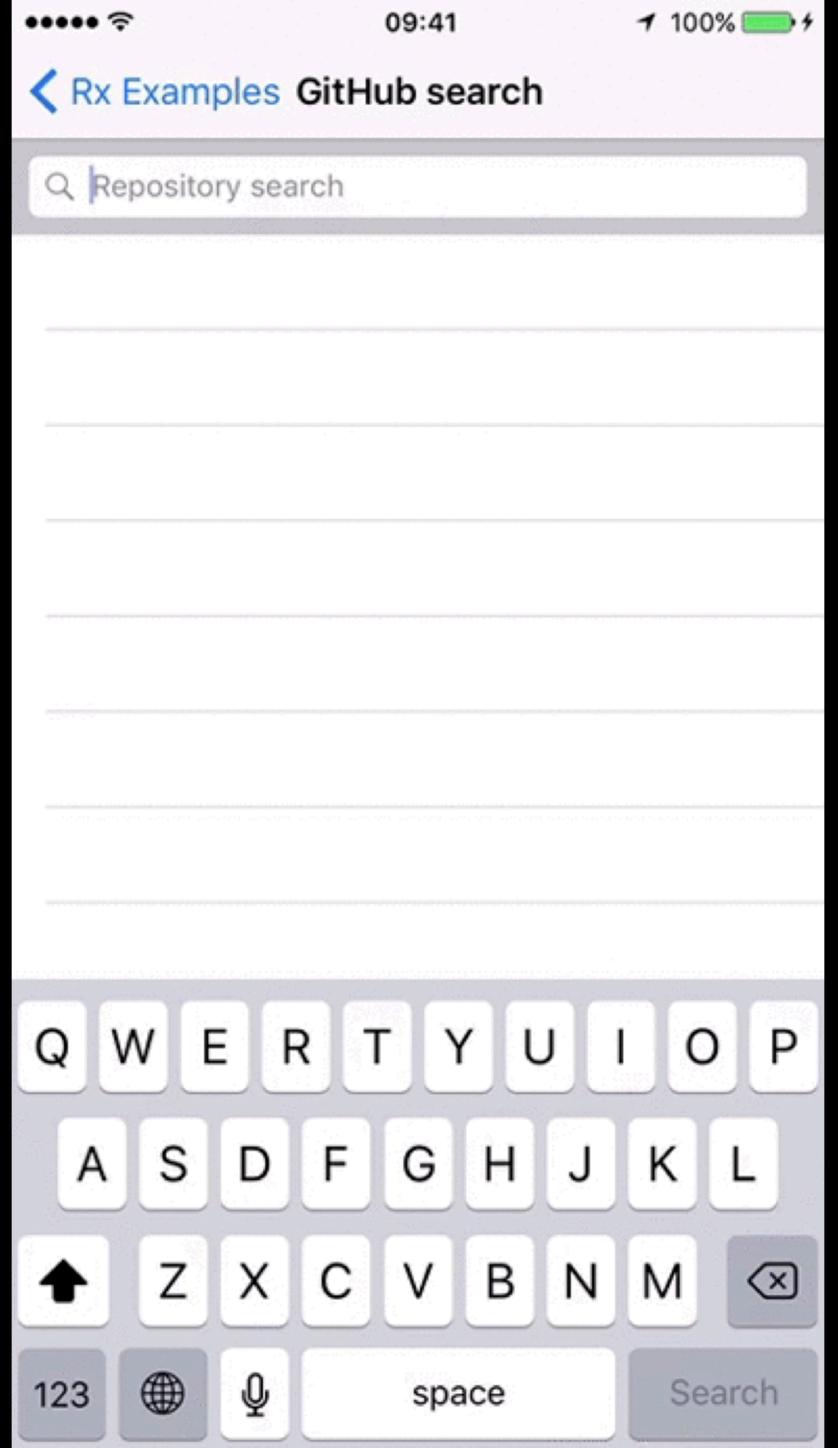
```
let searchResults = searchBar.rx.text.orEmpty
    .throttle(0.3, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .flatMapLatest { query -> Observable<[Repository]> in
        if query.isEmpty {
            return .just([])
        }
        return searchGitHub(query)
            .catchErrorJustReturn([])
    }
    .observeOn(MainScheduler.instance)

searchResults
    .bind(to: tableView.rx.items(cellIdentifier: "Cell")) {
        (index, repository: Repository, cell) in
        cell.textLabel?.text = repository.name
        cell.detailTextLabel?.text = repository.url
    }
    .disposed(by: disposeBag)
```

```
let searchResults = searchBar.rx.text.orEmpty
    .throttle(0.3, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .flatMapLatest { query -> Observable<[Repository]> in
        if query.isEmpty {
            return .just([])
        }
        return searchGitHub(query)
            .catchErrorJustReturn([])
    }
    .observeOn(MainScheduler.instance)

searchResults
    .bind(to: tableView.rx.items(cellIdentifier: "Cell")) {
        (index, repository: Repository, cell) in
        cell.textLabel?.text = repository.name
        cell.detailTextLabel?.text = repository.url
    }
    .disposed(by: disposeBag)
```

```
let searchResults = searchBar.rx.text.orEmpty
    .throttle(0.3, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .flatMapLatest { query -> Observable<[Repository]> in
        if query.isEmpty {
            return .just([])
        }
        return searchGitHub(query)
            .catchErrorJustReturn([])
    }
    .observeOn(MainScheduler.instance)

searchResults
    .bind(to: tableView.rx.items(cellIdentifier: "Cell")) {
        (index, repository: Repository, cell) in
        cell.textLabel?.text = repository.name
        cell.detailTextLabel?.text = repository.url
    }
    .disposed(by: disposeBag)
```

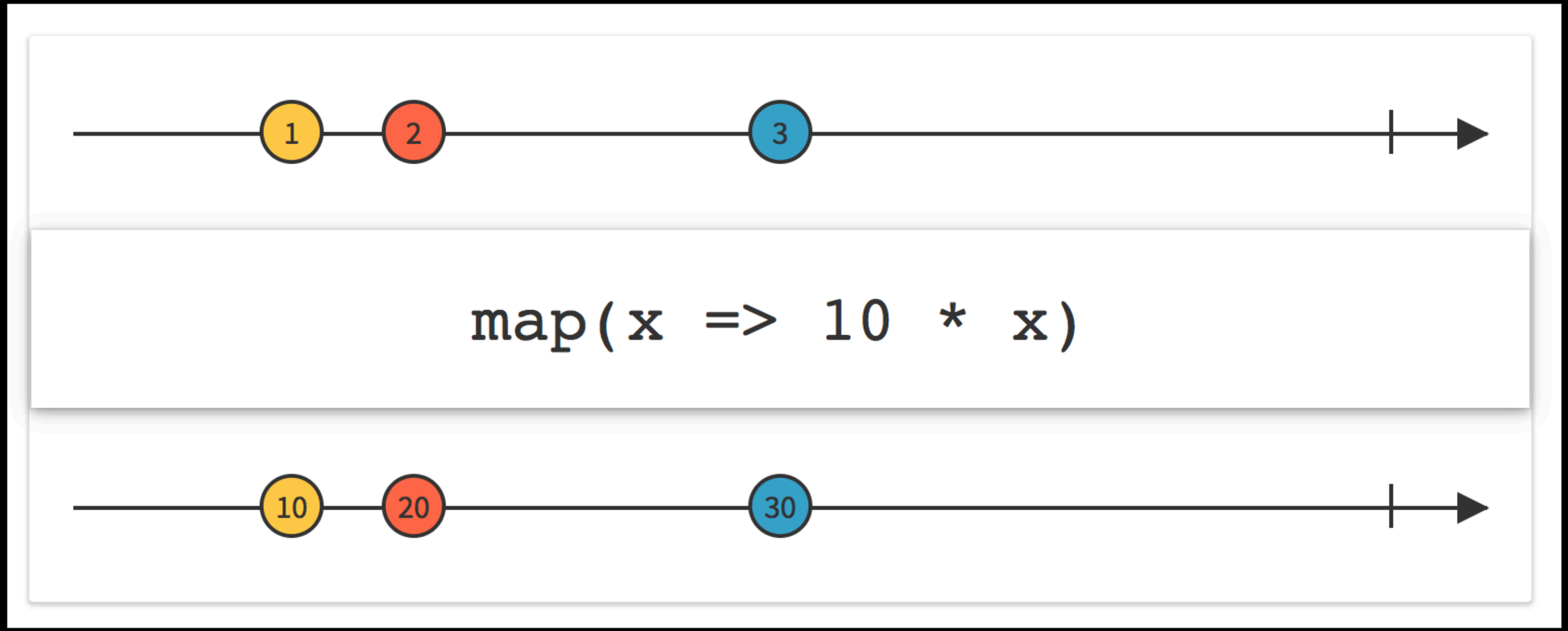
```
let searchResults = searchBar.rx.text.orEmpty
    .throttle(0.3, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .flatMapLatest { query -> Observable<[Repository]> in
        if query.isEmpty {
            return .just([])
        }
        return searchGitHub(query)
            .catchErrorJustReturn([])
    }
    .observeOn(MainScheduler.instance)

searchResults
    .bind(to: tableView.rx.items(cellIdentifier: "Cell")) {
        (index, repository: Repository, cell) in
        cell.textLabel?.text = repository.name
        cell.detailTextLabel?.text = repository.url
    }
    .disposed(by: disposeBag)
```

```
let searchResults = searchBar.rx.text.orEmpty
    .throttle(0.3, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .flatMapLatest { query -> Observable<[Repository]> in
        if query.isEmpty {
            return .just([])
        }
        return searchGitHub(query)
            .catchErrorJustReturn([])
    }
    .observeOn(MainScheduler.instance)

searchResults
    .bind(to: tableView.rx.items(cellIdentifier: "Cell")) {
        (index, repository: Repository, cell) in
        cell.textLabel?.text = repository.name
        cell.detailTextLabel?.text = repository.url
    }
    .disposed(by: disposeBag)
```


09:41
100%
Rx Examples
GitHub search
Repository search
Q W E R T Y U I O P
A S D F G H J K L
Z X C V B N M
123
space
Search

Let's focus on some Rx operators

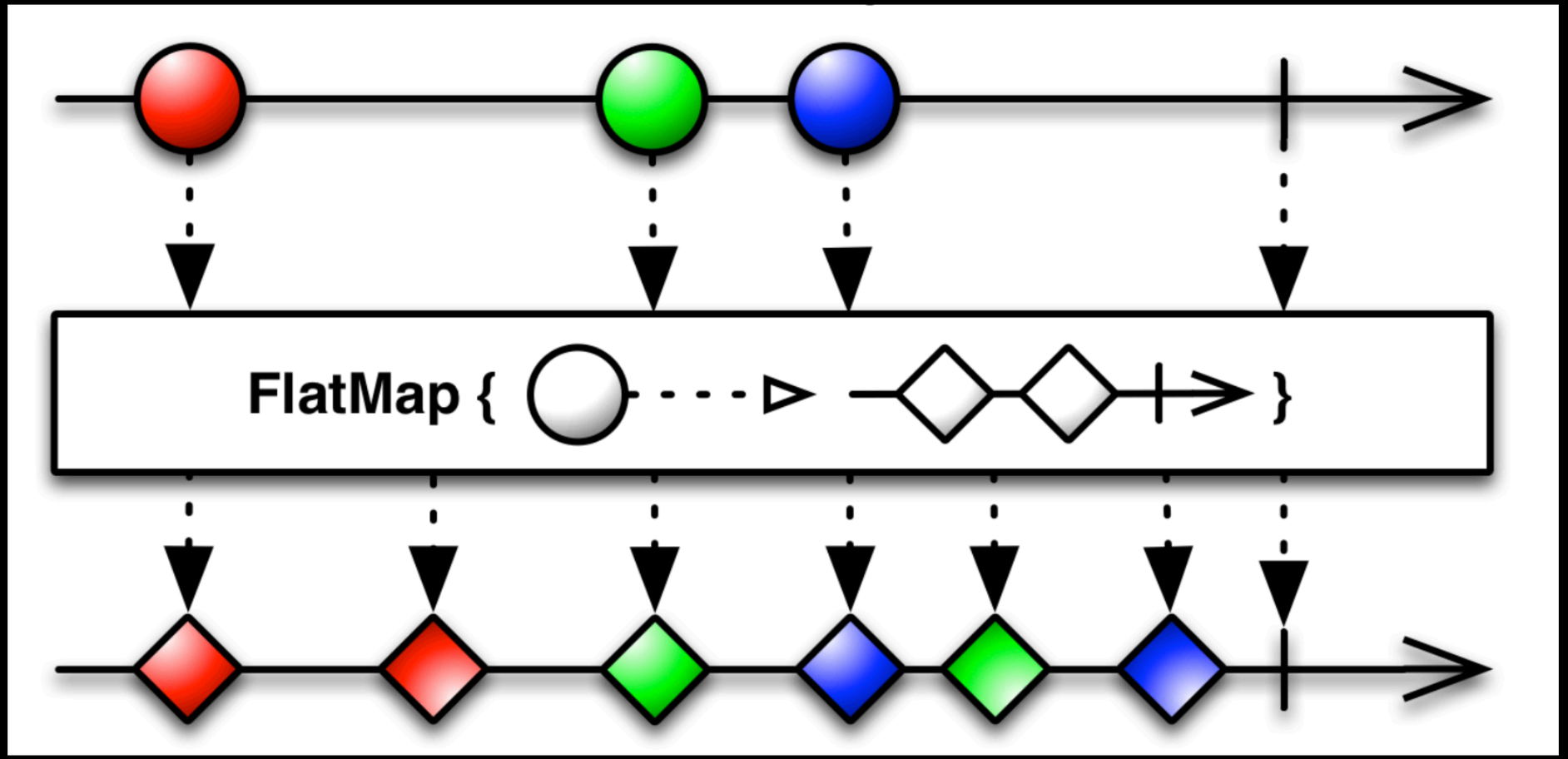
map



map

```
let observable = Observable.of(1, 2, 3, 4)

observable
    .map { value in return value * value }
    .subscribe(onNext: { transformedValue
        print(transformedValue)
    })
```

flatMap



flatMap

```
let data = [1, 2, 3]

data.map({ int in return [int, int + 1, int + 2] })

// [[1, 2, 3], [2, 3, 4], [3, 4, 5]]


data.flatMap({ int in return [int, int + 1, int + 2] })

// [1, 2, 3, 2, 3, 4, 3, 4, 5]
```

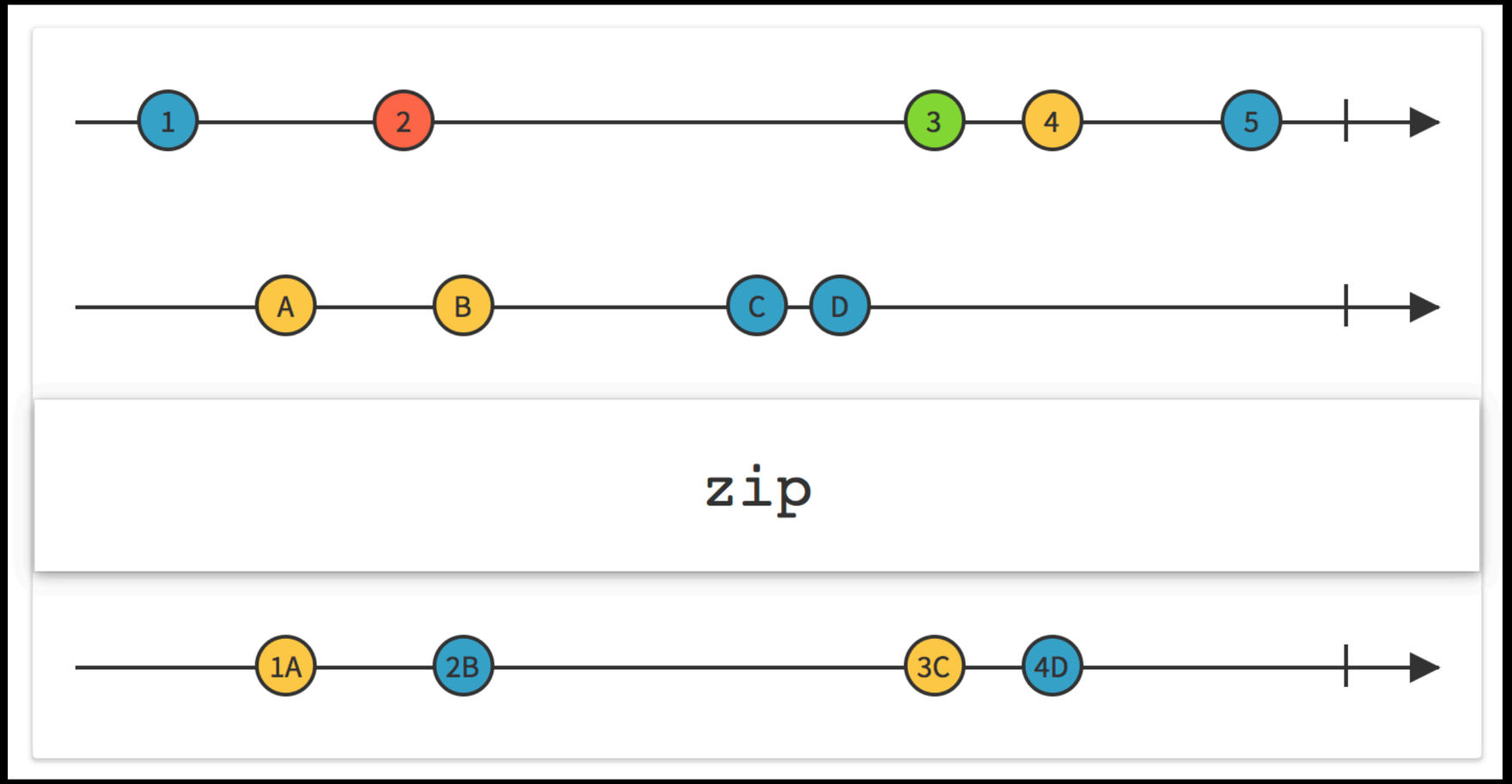
flatMap

```
func service1() -> Observable<Int>
func service2(_ arg: Int) -> Observable<String>

service1()
    .flatMap { service1Result in return service2(service1Result) }
    .subscribe(onNext: { service2Result in
        print(service2Result)
    })
```

flatMap

```
func service1() -> Observable<Int>
func service2(_ arg: Int) -> Observable<String>

service1()
    .flatMap { service1Result in return service2(service1Result) }
    .subscribe(onNext: { service2Result in
        print(service2Result)
    }, onError: { error in
        // deal with `error`
    })
```

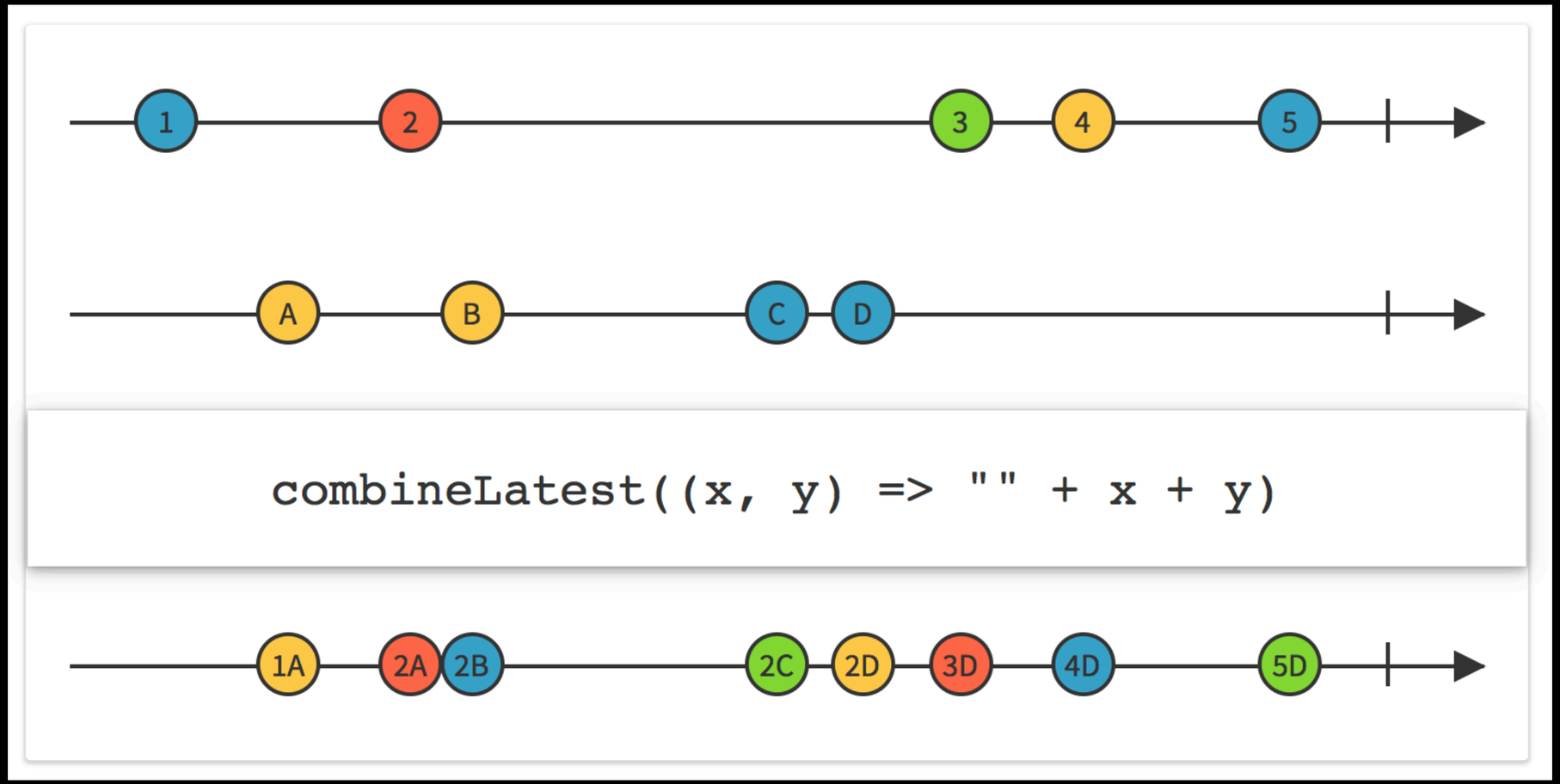
zip



zip

```
func networkCall1() -> Observable<Int>
func networkCall2() -> Observable<String>

Observable.zip(networkCall1(), networkCall2())
    .suscribe(onNext: { int, string in
        print(int)
        print(string)
    })
```

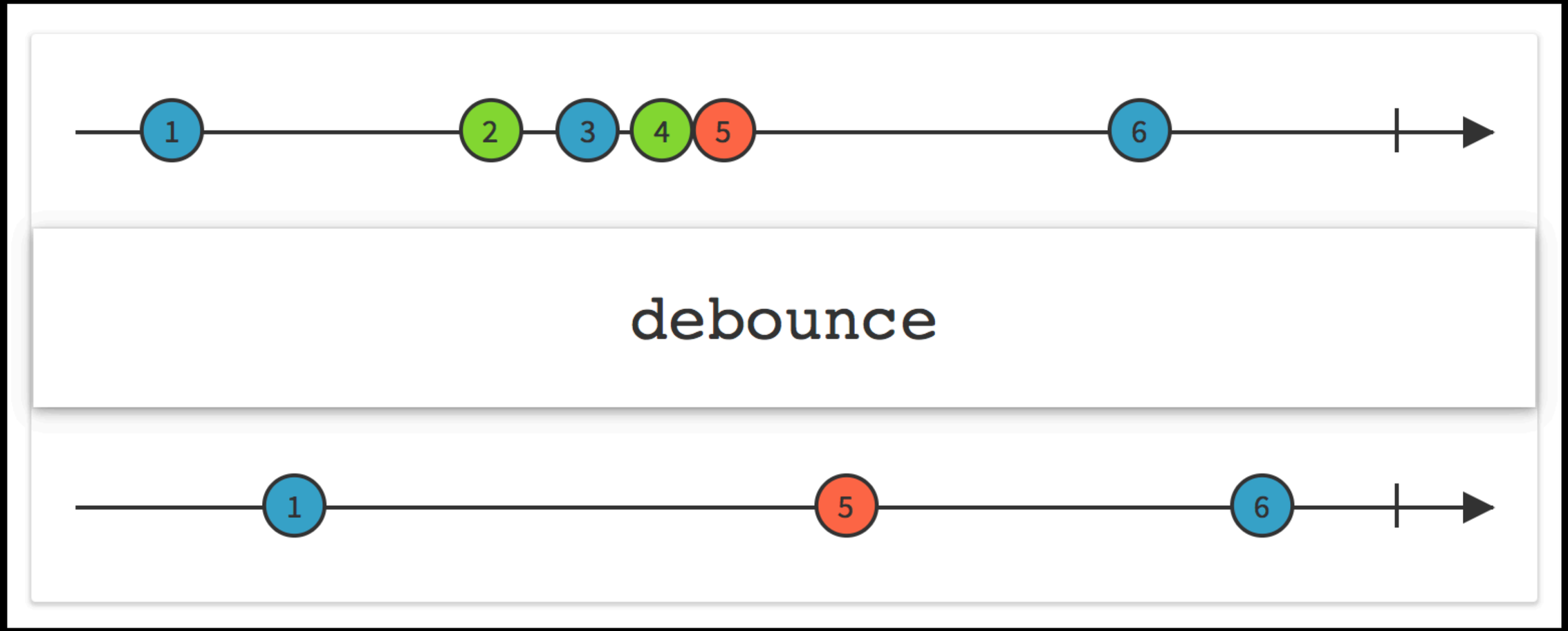
combineLatest



combineLatest

```
let passwordField = UITextField()
let passwordConfirmationField = UITextField()

Observable.combineLatest(passwordField.rx.text, passwordConfirmationField.rx.text)
    .map { password, confirmation in return password == confirmation }
    .subscribe(onNext: { inputsMatch in
        if inputsMatch {
            // ...
        else {
            // ...
        }
    })
```

debounce



debounce

```
let scrollView = UIScrollView()

scrollView
    .rx
    .contentOffset
    .debounce(0.3)
    .subscribe(onNext: { contentOffset in
        // send analytics
    })
```

subscribeOn & observeOn

subscribeOn & observeOn

```
func heavyComputation() -> Observable<Int>

let computationScheduler = ConcurrentDispatchQueueScheduler(qos: .userInitiated)

heavyComputation()
    .subscribeOn(computationScheduler)
    .observeOn(MainScheduler.instance)
    .subscribe(onNext: { result in
        // use `result`
    })
```

Recap

Recap

Recap

- Rx is a set of standardized APIs

Recap

- Rx is a set of standardized APIs
- Built on the Observer pattern

Recap

- Rx is a set of standardized APIs
- Built on the Observer pattern
- Provides operators to:

Recap

- Rx is a set of standardized APIs
- Built on the Observer pattern
- Provides operators to:
  - transform data, in a scalable fashion

Recap

- Rx is a set of standardized APIs
- Built on the Observer pattern
- Provides operators to:
  - transform data, in a scalable fashion
  - synchronize streams, without dedicated logic

Recap

- Rx is a set of standardized APIs
- Built on the Observer pattern
- Provides operators to:
  - transform data, in a scalable fashion
  - synchronize streams, without dedicated logic
  - peform multi-threading, with minimum boilerplate

Let's draft an architecture

Drafting an architecture

Drafting an architecture

- Consider the following layers

Drafting an architecture

- Consider the following layers
  - Networking

Drafting an architecture

- Consider the following layers
  - Networking
  - Service

Drafting an architecture

- Consider the following layers
  - Networking
  - Service
  - ViewModel

Drafting an architecture

- Consider the following layers
  - Networking
  - Service
  - ViewModel
  - UI

How and where does Rx fit?

Networking

Networking

```
class Client {
    func requestJSON(_ request: URLRequest) -> Single<JSON> {
        return URLSession
            .shared
            .rx
            .json(request: request)
            .asSingle()
    }
}
```

Service

Service

```
class MyService {
    struct MyServiceParameters: Encodable { /* ... */ }
    struct MyServiceResponse: Decodable { /* ... */ }

    private let client = Client()

    func call(with parameters: MyServiceParameters) -> Single<MyServiceResponse> {
        return client.requestJSON(parameters.encode())
            .map { json in json.decode(MyServiceResponse.self) }
    }
}
```

ViewModel

ViewModel

```
class MyViewModel {
    let presentableData = Variable<String>

    private let service = MyService()
    private let disposeBag = DisposeBag()

    private func format(_ response: MyServiceResponse) -> String { /* ... */ }

    func fetchData() {
        let parameters = MyServiceParameters()

        service.call(with: parameters)
            .map { response in return self.format(response) }
            .subscribe(onNext: { presentableData in
                self.presentableData.value = presentableData
            })
            .disposed(by: self.disposeBag)
    }
}
```

UI

UI

```
class MyViewController: UIViewController {
    let label = UILabel()

    private let viewModel = MyViewModel()
    private let disposeBag = DisposeBag()

    override func viewDidLoad() {
        super.viewDidLoad()

        self.bindViewModel()
        self.viewModel.fetchData()
    }

    private func bindViewModel() {
        self.viewModel.presentableData
            .asObservable()
            .subscribe(onNext: { presentableData in
                self.label.text = presentableData
            })
            .disposed(by: self.disposeBag)
    }
}
```

Takeways

Takeways

- Rx is an **architectural** pattern

Takeways

- Rx is an **architectural** pattern
- It takes concepts you **already use**, and makes them **scale**

Takeways

- Rx is an **architectural** pattern
- It takes concepts you **already use**, and makes them **scale**
- It aims at reducing **boilerplate** and **hard to maintain** code

Takeways

- Rx is an **architectural** pattern
- It takes concepts you **already use**, and makes them **scale**
- It aims at reducing **boilerplate** and **hard to maintain** code
- It **does not** need to be ubiquitous to be effective

Questions?

Contact

Email: vincent.pradeilles@equensworldline.com
Twitter: @v_pradeilles
GitHub: https://github.com/vincent-pradeilles