

RWSWift

workshops

WI-FI

 EL_Passion_Guest

 passionateguest

URL do repozytorium

<https://github.com/elpassion/RxSwift-workshops>

Agenda

1. Moduł I - wprowadzenie do Functional Reactive Programming i RxSwifta
2. Coffee break ☕
3. Moduł II - UI bindings in RxCocoa
4. Pizza 🍕
5. Moduł III - Reactive API calls
6. Nasze spostrzeżenia na temat RxSwifta

***Każdy moduł składa się z części teoretycznej i praktycznej.**

Moduł

Wprowadzenie do FRP i RxSwifta

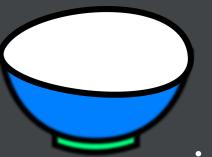
Paradygmaty programowania

1. Imperatywne
2. Deklaratywne
 - funkcyjne

**Programowanie
imperatywne -
opis instrukcji **JAK** coś
wykonać.**

Opis instrukcji **JAK** coś wykonać

1. Weź miskę.



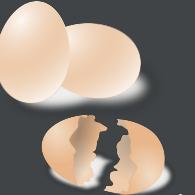
2. Dosyp mąkę.



3. Dolej mleka.



4. Dodaj jajka.



5. Wymieszaj wszystko i smaż ciasto na patelni.



**Programowanie
deklaratywne -
opis **co** zrobić, nie jak
to zrobić.**

Opis CO zrobić, nie jak to zrobić

[ ,  , ]

. reduce( , +)

. mix()

. fryPancakes()

= >



Programowanie funkcyjne

```
func sum(a: Int, b: Int) -> Int {  
    return a + b  
}
```

Co nie jest funkcyjne?

```
var value = 0
```

```
func increment() {  
    value += 1  
}
```

Funkcyjnie

```
func increment(value: Int) -> Int {  
    return value + 1  
}
```

Nie modyfikujemy stanu - zmienna value jest zdefiniowana jako parametr funkcji.

Mając poniższą liste aut:

```
let cars = [  
    Car(brand: "BMW", model: "i3", year: 2016, horsePower: 170, engineType: .electric),  
    Car(brand: "BMW", model: "428xi", year: 2014, horsePower: 170, engineType: .gasoline),  
    Car(brand: "Toyota", model: "Auris", year: 2010, horsePower: 170, engineType: .diesel),  
    Car(brand: "BMW", model: "M135i", year: 2015, horsePower: 326, engineType: .gasoline),  
    Car(brand: "Volkswagen", model: "Golf R", year: 2015, horsePower: 300, engineType: .gasoline),  
    Car(brand: "Honda", model: "Civic Type R", year: 2016, horsePower: 310, engineType: .gasoline),  
    Car(brand: "Mercedes", model: "A 45 AMG", year: 2014, horsePower: 360, engineType: .gasoline),  
    Car(brand: "Audi", model: "A3", year: 2017, horsePower: 110, engineType: .gasoline)  
]
```

chcemy znaleźć takie, które zostały wyprodukowane po 2015
oraz mają moc conajmniej 300km.

Imperatywnie

```
var filteredCars: [Car] = []

for car in cars {
    if car.year >= 2015 && car.horsePower >= 300 {
        filteredCars.append(car)
    }
}

print(filteredCars)

// Car(brand: "BMW", model: "M135i", year: 2015, horsePower: 326, engineType: .gasoline)
// Car(brand: "Volkswagen", model: "Golf R", year: 2015, horsePower: 300, engineType: .gasoline)
// Car(brand: "Honda", model: "Civic Type R", year: 2016, horsePower: 310, engineType: .gasoline)
```

Funkcjnie

```
let filteredCars = cars.filter { car -> Bool in
    return car.year >= 2015 && car.horsePower >= 300
}

print(filteredCars)

// Car(brand: "BMW", model: "M135i", year: 2015, horsePower: 326, engineType: .gasoline)
// Car(brand: "Volkswagen", model: "Golf R", year: 2015, horsePower: 300, engineType: .gasoline)
// Car(brand: "Honda", model: "Civic Type R", year: 2016, horsePower: 310, engineType: .gasoline)
```

Funkcyjnie

```
let filteredCars = cars.filter { $0.year >= 2015 && $0.horsePower >= 300 }
```

```
print(filteredCars)
```

```
// Car(brand: "BMW", model: "M135i", year: 2015, horsePower: 326, engineType: .gasoline)
// Car(brand: "Volkswagen", model: "Golf R", year: 2015, horsePower: 300, engineType: .gasoline)
// Car(brand: "Honda", model: "Civic Type R", year: 2016, horsePower: 310, engineType: .gasoline)
```

Mając daną listę samochodów:

```
let cars = [  
    Car(brand: "BMW", model: "i3", year: 2016, horsePower: 170, engineType: .electric),  
    Car(brand: "BMW", model: "428xi", year: 2014, horsePower: 170, engineType: .gasoline),  
    Car(brand: "Toyota", model: "Auris", year: 2010, horsePower: 170, engineType: .diesel),  
    Car(brand: "BMW", model: "M135i", year: 2015, horsePower: 326, engineType: .gasoline),  
    Car(brand: "Volkswagen", model: "Golf R", year: 2015, horsePower: 300, engineType: .gasoline),  
    Car(brand: "Honda", model: "Civic Type R", year: 2016, horsePower: 310, engineType: .gasoline),  
    Car(brand: "Mercedes", model: "A 45 AMG", year: 2014, horsePower: 360, engineType: .gasoline),  
    Car(brand: "Audi", model: "A3", year: 2017, horsePower: 110, engineType: .gasoline)  
]
```

chcemy policzyć średni rok produkcji dla wszystkich aut.

Imperatywnie

```
var sum: Int = 0

for car in cars {
    sum += car.year
}

let averageYear = sum / cars.count

print(averageYear)

// 2014
```

Funkcyjnie

```
let averageYear = cars.map { $0.year }.reduce(0, +) / cars.count  
print(averageYear)  
  
// 2014
```

Funkcje wyższego rzędu czyli **Higher Order Functions**

Funkcje wyższego rzędu

- funkcje, które przyjmują inną funkcję jako parametr

```
func map<T>(_ transform: (Element) throws -> T) rethrows -> [T]
```

Funkcje wyższego rzędu

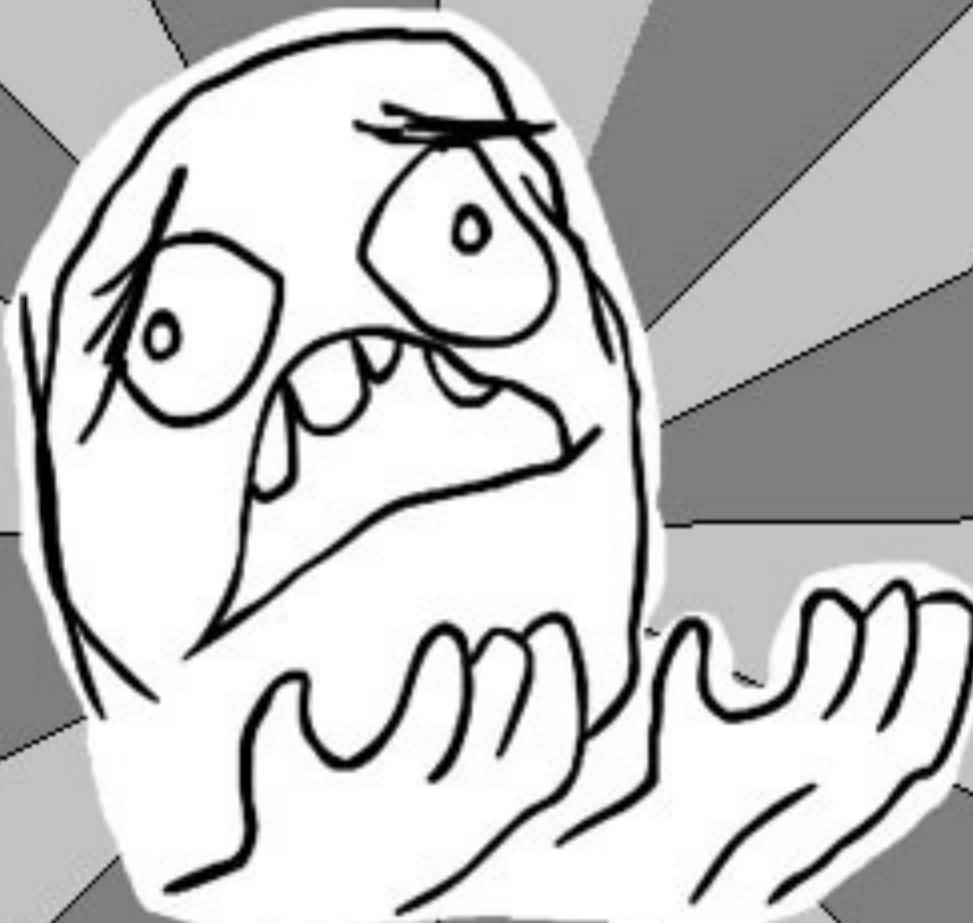
- funkcje, które przyjmują inną funkcję jako parametr

```
func map<T>(_ transform: (Element) throws -> T) rethrows -> [T]
```

- funkcje, które zwracają inne funkcje

```
func add(a: Int) -> (Int) -> Int {  
    return { b in a + b }  
}
```

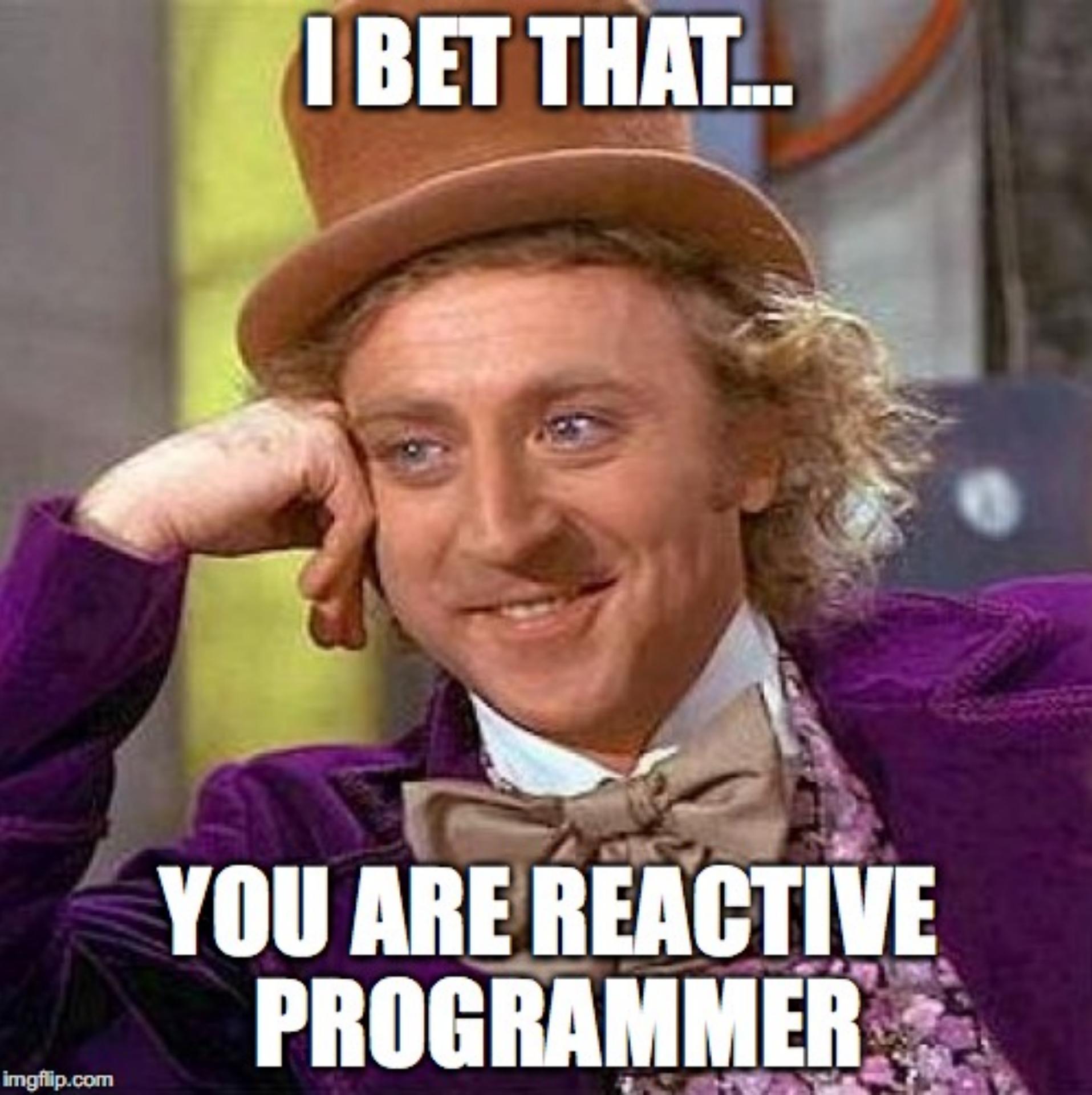
DECLARATIVE?



memy.pl

WHAT ABOUT REACTIVE?

Programowanie reaktywne



I BET THAT...

**YOU ARE REACTIVE
PROGRAMMER**

Programowanie reaktywne

	A	B
1	x:	2
2	y:	3
3	z:	6

Imperatywnie

x = 2

y = 3

z = x * y // 6

x = 3

z // 6

Reaktywnie

x = 2

y = 3

z = x * y // 6

x = 3

z // 9

RWStarfit



Reactive **e**x^tension for **S**wift

Functional + Reactive = FRP ❤

Rx.NET

RxJS

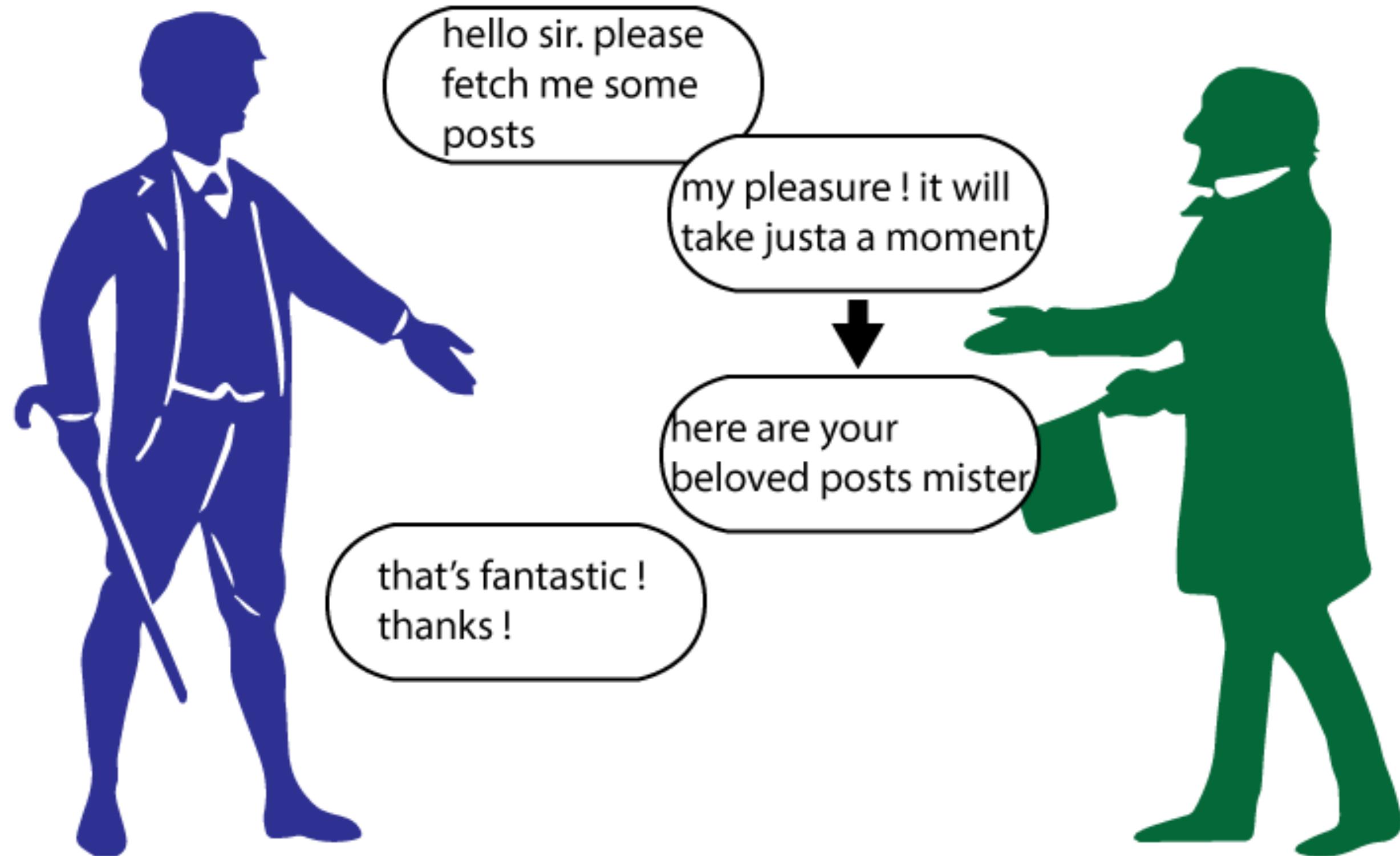
RxKotlin

RxScala

RxJava

"Baron" OBSERVER

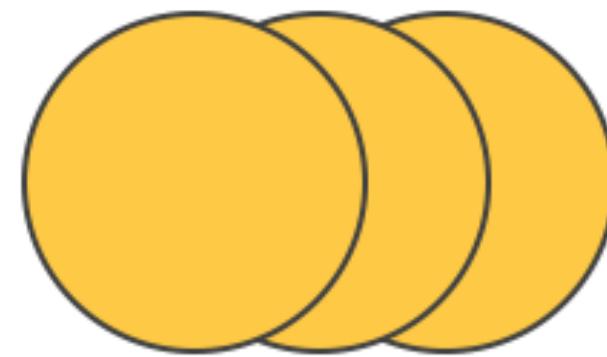
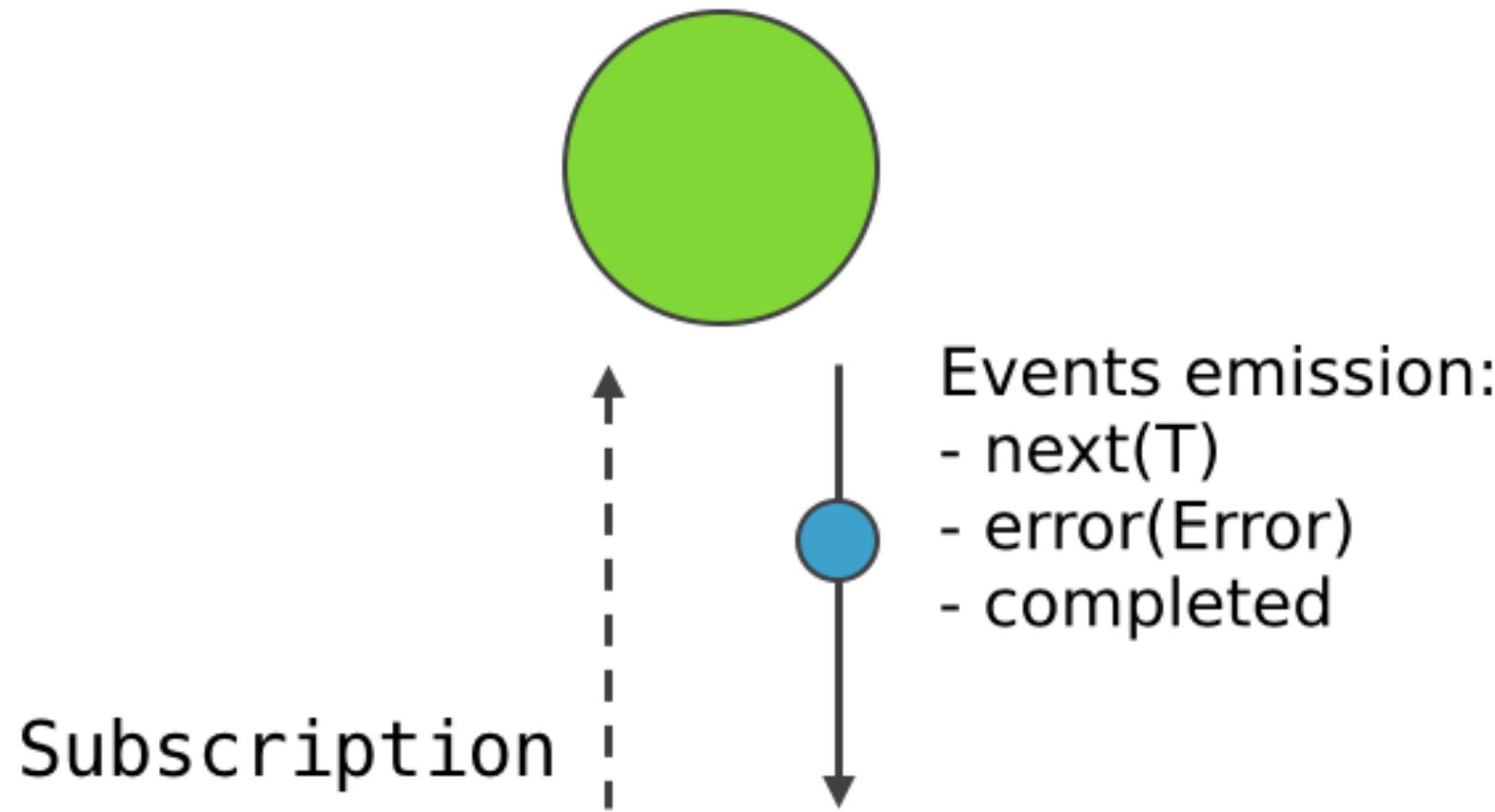
"Lord" OBSERVABLE



RxSwift

- **Observable** - emituje eventy
- **Observer** - odbiera eventy
- **Subscription** - łączy Observable z Observer

Observable



Observer

RxSwift

Observable<T> - emituje eventy

RxSwift

Observable<T> - emituje eventy

- **next**

RxSwift

Observable<T> - emituje eventy

- **next**
- **completed**

RxSwift

Observable<T> - emituje eventy

- **next**
- **completed**
- **error**

RxSwift

Observer<T> - odbiera eventy

RxSwift

Subscription - Łączy Observable z Observer

```
let observable = Observable<String>.of("A", "B", "C")
let observer = AnyObserver<String> { event in
    switch event {
        case .next(let element): print("Next: \(element)")
        case .error(let error): print("Error: \(error)")
        case .completed: print("Completed")
    }
}
let subscription: Disposable = observable.subscribe(observer)

// Next: A
// Next: B
// Next: C
// Completed
```

```
let observable = Observable<String>.of("A", "B", "C")
let observer = AnyObserver<String> { event in
    switch event {
        case .next(let element): print("Next: \(element)")
        case .error(let error): print("Error: \(error)")
        case .completed: print("Completed")
    }
}
let subscription: Disposable = observable.subscribe(observer)

subscription.dispose()
```

DisposeBag

```
let disposeBag = DisposeBag()
```

```
Observable.of("A", "B", "C")
    .subscribe(onNext: { value in print(value) })
    .disposed(by: disposeBag)
```

Rx~~coco~~coca

RxSwift + UIKit = ❤

```
searchTextField.rx.text // Observable<String?>
    .orEmpty // Observable<String>
    .filter { text in
        !text.isEmpty
    } // Observable<String>
    .flatMapLatest { text in
        apiClient.currentWeather(city: text)
    } // Observable<UIImage>
    .bind(to: weatherImageView.rx.image)
```

Operatory w RxSwift

filter

```
Observable.of(1, 2, 3, 4, 5, 6)
    .filter { $0 > 4 }
    .subscribe(onNext: { print($0) })
```

// 5

// 6

\$0 > 4

.filter

map

```
Observable.of(1, 2, 3)
    .map { "Twój numer to: \($0)" }
    .subscribe(onNext: { print($0) })
```

```
// Twój numer to: 1
// Twój numer to: 2
// Twój numer to: 3
```

“Twój numer to: \(\$0)”

.map

takeWhile

```
Observable.of(1, 2, 3, 4, 3, 2, 1)
    .takeWhile { $0 < 3 }
    .subscribe(onNext: { print($0) },
               onCompleted: { print("Completed") })
// 1
// 2
// Completed
```

```
$0 < 3
```

```
.takeWhile
```

distinctUntilChanged

```
Observable.of(1, 1, 1, 1, 2, 3, 3, 5, 1, 5)
    .distinctUntilChanged()
    .subscribe(onNext: { print($0) })

// 1
// 2
// 3
// 5
// 1
// 5
```



\$0

.distinctUntilChanged()

toArray

```
Observable.of(1, 2, 3, 4, 5) // Observable<Int>
    .toArray() // Observable<[Int]>
    .subscribe(onNext: { print($0) },
               onCompleted: { print("Completed") })
// [1, 2, 3, 4, 5]
// Completed
```

\$0

.toArray()

skip

```
Observable.of("a", "b", "c", "d", "e")
    .skip(3)
    .subscribe(onNext: { print($0) })
```

```
// "d"
// "e"
```

\$0

.skip(3)

reduce

Observable

```
.of(1, 2, 3, 4, 5)
.reduce(0, accumulator: { (result, element) -> Int in
    return result + element
})
.subscribe(onNext: { print($0) },
onCompleted: { print("Completed") })
```

// 15

// Completed

\$0

```
.reduce(0, accumulator: { (result, element) ... })
```

merge

```
let first = PublishSubject<String>()
let second = PublishSubject<String>()
```

```
Observable.merge(first, second)
    .subscribe(onNext: { print($0) })
```

```
first.onNext("A") // A
first.onNext("B") // B
second.onNext("1") // 1
second.onNext("2") // 2
first.onNext("AB") // AB
second.onNext("3") // 3
```

first

second

\$0

.merge()

combineLatest

```
let first = PublishSubject<String>()
let second = PublishSubject<String>()
```

```
Observable.combineLatest(first.asObservable(), second.asObservable())
    .subscribe(onNext: { firstValue, secondValue in
        print("First: \(firstValue) Second: \(secondValue)")
    })
```

```
first.onNext("1")
first.onNext("2")
second.onNext("3") // First: 2 Second: 3
first.onNext("4")  // First: 4 Second: 3
first.onNext("5")  // First: 5 Second: 3
second.onNext("6") // First: 5 Second: 6
```

first

second

firstValue secondValue

.combineLatest(first, second)

zip

```
let first = PublishSubject<String>()
let second = PublishSubject<String>()

Observable.zip(first.asObservable(), second.asObservable())
    .subscribe(onNext: { firstValue, secondValue in
        print("First: \(firstValue) Second: \(secondValue)")
    })

first.onNext("1")
first.onNext("2")
second.onNext("3") // First: 1 Second: 3
first.onNext("4")
first.onNext("5")
second.onNext("6") // First: 2 Second: 6
first.onNext("7")
```

first

second

firstValue

secondValue

`.zip(first.asObservable(), second.asObservable())`

catchErrorJustReturn

```
let subject = PublishSubject<Int>()

subject
    .catchErrorJustReturn(7)
    .subscribe(onNext: { print($0) },
               onCompleted: { print("Completed") })

subject.onNext(1) // 1
subject.onNext(2) // 2
subject.onError(NSError(domain: "", code: 0, userInfo: nil)) // 7
// Completed
```

\$0

.catchErrorJustReturn(7)

catchError

```
let first = PublishSubject<Int>()
let second = PublishSubject<Int>()
```

```
first
    .catchError({ _ -> Observable<Int> in
        return second
    })
    .subscribe(onNext: { print($0) })
```

```
first.onNext(1) // 1
first.onNext(2) // 2
first.onError(NSError(domain: "", code: 0, userInfo: nil))
second.onNext(4) // 4
second.onNext(5) // 5
```

first

\$0

.catchError()

do

```
Observable.of(1, 2, 3)
    .do(onNext: { print("🌴 + \"\$0\") })
    .subscribe(onNext: { print(\$0) })
```

```
// 🌴 + 1
// 1
// 🌴 + 2
// 2
// 🌴 + 3
// 3
```

\$0

```
.do(print("🌴 + \($0)"))
```

zadania

Modular

UI bindings w RxSwift i RxCocoa

RxSwift + UIKit => RxCocoa

4 typy subjectów

- PublishSubject
- PublishRelay
- BehaviorSubject
- BehaviorRelay

	Subject	Relay
Publish	No state, errors	No state, no errors
Behavior	State, errors	State, no errors

Bindings

```
let relay = BehaviorRelay(value: "")
```

```
let label = UILabel(frame: .zero)
```

```
relay.asObservable() // Observable<String>
```

```
.bind(to: label.rx.text)
```

```
relay.accept("Hello")
```

```
print(label.text) // "Hello"
```

Błędy

```
let subject = PublishSubject<String>()  
let label = UILabel(frame: .zero)  
  
subject.asObservable() // Observable<String>  
.bind(to: label.rx.text)  
  
subject.onError(APIError.notLoggedIn) // crash
```

Obsługa błędów

```
let subject = PublishSubject<String>()  
let label = UILabel(frame: .zero)  
  
subject.asObservable() // Observable<String>  
.catchErrorJustReturn("")  
.bind(to: label.rx.text)  
  
subject.onError(APIError.notLoggedIn)  
print(label.text) // ""
```

Główny wątek

```
let strings: PublishSubject<String> = // ...
```

```
let label = UILabel(frame: .zero)
```

```
strings  
.observeOn(MainScheduler.instance)  
.bind(to: label.rx.text)
```

Driver

- Subskrypcja zawsze na głównym wątku
- Nie może propagować błędów
- Subskrypcje są współdzielone

Driver

```
let strings: Observable<String> = // ...  
let label = UILabel(frame: .zero)  
  
let stringsDriver: Driver<String> = strings  
.asDriver(onErrorJustReturn: "")  
  
stringsDriver  
.drive(label.rx.text)
```

UILabel

- rx.text

```
userInput.asObservable() // Observable<String>
    .bind(to: label.rx.text)
```

UITextField

- rx.text

```
textField.rx.text // ControlProperty<String?>
    .subscribe(onNext: { print($0) } )
```

UIButton

- rx.tap

```
rx.tap // ControlEvent<Void>
    .map { _ in "Tapped" } // Observable<String>
    .bind(to: label.rx.text)
```

UIImageView

- rx.image

```
button.rx.tap // ControlEvent<Void>
    .map { _ in UIImage(named: "our_image") } // Observable<UIImage?>
    .bind(to: imageView.rx.image)
```

UISegmentedControl

- rx.selectedIndex

```
segmentedControl.rx.selectedIndex // ControlProperty<Int>
    .map { index in UIImage(named: "asset_\($0)") } // Observable<UIImage?>
    .bind(to: imageView.rx.image)
```

UITableView

- rx.items

```
items.asObservable() // Observable<Element>
    .bind(to: tableView.rx.items(cellIdentifier: "Identifier")) { index, element, cell in
        cell.titleLabel.text = "\(element) at \(index)"
    }
```

- rx.itemSelected

```
tableView.rx.itemSelected // Observable<IndexPath>
    .map { "Selected item at \"\($0.item)" } // Observable<String>
    .bind(to: label.rx.text)
```

- rx.modelSelected

```
tableView.rx.modelSelected // Observable<ElementType>
    .map { "Selected item \"\$0\"" } // Observable<String>
    .bind(to: label.rx.text)
```

UICollectionView

- rx.items
- rx.itemSelected
- rx.modelSelected

DisposeBag

```
class Controller: UIViewController {  
    private let disposeBag = DisposeBag()  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
  
        textField.rx.text.orEmpty  
            .bind(to: label.rx.text)  
            .disposed(by: disposeBag)  
    }  
}
```

UITableViewCell

```
class Cell: UITableViewCell {  
    private(set) var disposeBag: DisposeBag = DisposeBag()  
  
    override func prepareForReuse() {  
        super.prepareForReuse()  
  
        disposeBag = DisposeBag()  
    }  
}
```

Setup projektu

1. Folder: Module-2.
2. Budowanie aplikacji na symulatorze.

Aplikacja

- Służy do wyświetlania rozkładu jazdy pociągów.
- Użytkownik może filtrować listę na podstawie przedziałów czasu (**0-30, 30-60, 60+**).
- Użytkownik może wykonać “check-in” w dowolnym z pociągów podając swoje imię.

Implementacja

Aplikacja jest napisana za pomocą natywnych interface'ów
UIKit, czyli:

- UISegmentedControl: `addTarget(_:action:for:)`
`(TimetableViewController)`,
- UITableView: `UITableViewDataSource` i
`UITableViewDelegate` (`TimetableViewController`).

zadania

Zadanie #1

W klasie `TimetableViewController` zastąp użycie funkcji
`addTarget(_:action:for:)` bindigiem rx'owym.

Zadanie #2

W klasie `TimetableViewController` zamień klasyczną implementację data source'a w table view na wersje rx'ową.

Wykorzystaj poniższe property z klasy `TimetableService`

```
var timetableEntries: Observable<[TimetableEntry]>
```

Zadanie #3

W klasie `TimetableViewController` zrefaktoruj akcje tapnięcia
– zamiast ustawiania closure'a skorzystaj z extension `rx.tap`
na `UIButton`.

Module 1 Networking

Dwa typy strumieni

- Hot observable
- Cold observable

Hot observable

Emituje zdarzenia niezależnie od subskrypcji. Przykłady:

- Tapnięcia przycisku i inne zdarzenia UI-owe (**ControlEvent**),
- Czas systemowy,
- m. in. **PublishSubject**, **BehaviorRelay**.

Hot observable

```
let subject = PublishSubject<Int>()

subject
    .subscribe(onNext: { value in
        print("First subscription: \(value)")
    })

subject.onNext(800) // "First subscription: 800"

subject
    .subscribe(onNext: { value in
        print("Second subscription: \(value)")
    })

subject.onNext(801) // "First subscription: 801
                  // Second subscription: 801"
```

Cold observable

Rozpoczyna działanie (a co za tym idzie, emisję zdarzeń) dopiero po subskrypcji. Przykłady:

- Asynchroniczne pobieranie danych (z serwera, bazy danych, . . .),
- Asynchroniczne operacje (operation queue, dispatch, . . .).

Cold observable

```
let observable: Observable<Int> = Observable.create { observer in
    print("Something subscribed...")
    observer.onNext(800)
    observer.onNext(801)
    observer.onCompleted()
    return Disposables.create()
}
```

```
observable
    .subscribe(onNext: { value in          // <-- 1
        print("First subscription: \(value)")
    })
```

```
observable
    .subscribe(onNext: { value in          // <-- 2
        print("Second subscription: \(value)")
    })
```

Cold observable

```
// <-- 1
// "Something subscribed..."
// "First subscription: 800"
// "First subscription: 801"
// <-- 2
// "Something subscribed..."
// "Second subscription: 800"
// "Second subscription: 801"
```

Wyzwoływanie requestów

.rx.data

```
let request: URLRequest = // ...

URLSession.shared.rx.data(request: request)
    .map { try JSONDecoder().decode(APIResponse.self, from: $0) }
    .subscribe(onNext: { response in
        print("API response: \(response)")
    })
    .disposed(by: disposeBag)
```

.rx.response

```
typealias URLSessionResponse = (response: HTTPURLResponse, data: Data)

let request: URLRequest = // ...

URLSession.shared.rx.response(request: request)
    .subscribe(onNext: { (result: URLSessionResponse) in
        print("Status code: \(result.response.statusCode)")
        print("Body: \(result.data)")
    })
    .disposed(by: disposeBag)
```

ActivityIndicator

Kod utrzymywany w projekcie **RxExample** na oficjalnym repo **RxSwift**. URL: tiny.cc/rxindicator.

```
let activityIndicator = ActivityIndicator()

override func viewDidLoad() {
    super.viewDidLoad()

    requestObservable
        .trackActivity(activityIndicator)
        .subscribe()
        .disposed(by: disposeBag)

    activityIndicator.asDriver()
        .drive(onNext: { $0 ? Progress.show() : Progress.dismiss() })
        .disposed(by: disposeBag)
}
```

Funkcja flatMap

flatMap:

1. Działa jak `map`, ale wynikiem mapowania jest kontener, a nie wartość.
2. "Spłaszcza" wynikowy kontener.

flatMap - przykład (1)

```
let listOfLists: Array<Array<Int>> = [[1, 9, 8], [4, 3, 2]]  
let mappedList: Array<Array<Int>> = listOfLists.map { $0 }  
let flatMappedList: Array<Int> = listOfLists.flatMap { $0 }  
  
// let listOfLists: [[Int]] = ...  
// let mappedList: [[Int]] = ...  
// let flatMappedList: [Int] = ...  
  
print("\(mappedList)") // [[1, 9, 8], [4, 3, 2]]  
print("\(flatMappedList)") // [1, 9, 8, 4, 3, 2]
```

flatMap - przykład (2)

```
let address: Optional<String> = .some("https://google.com")
let mappedAddress: Optional<Optional<URL>> = address.map { URL(string: $0) }
let flatMappedAddress: Optional<URL> = address.flatMap { URL(string: $0) }

// let address: String? = ...
// let mappedAddress: URL?? = ...
// let flatMappedAddress: URL? = ...

print("\(mappedAddress)") // Optional(Optional(https://google.com))
print("\(flatMappedAddress)") // Optional(https://google.com)
```

flatMap w RxSwift

1. Przekształca elementy w obserwalne strumienie (**map** z wynikiem typu **Observable<?>**).
2. Złącza emisję zdarzeń **.next** oraz **.error** z wynikowych **Observable** w jeden strumień.

flatMap w RxSwift - przykład

```
Observable.of(82, 88, 33) // Observable<Int>
    .flatMap { number in
        Observable.of("\(number)", String(UnicodeScalar(UInt8(number))))
    } // Observable<String>
    .subscribe(onNext: { print($0) })

// 82
// R
// 88
// X
// 33
// !
```

Zagnieżdżone Observable

```
func logIn() -> Observable<User> { // ... }
```

```
let button: UIButton = ...
```

```
button.rx.tap
    .subscribe(onNext: { _ in
        logIn()
            .subscribe(onNext: { user in
                print("\(user)")
            })
    })
}
```

Zagnieżdżone Observable

```
func logIn() -> Observable<User> { // ... }
```

```
let button: UIButton = ...
```

```
button.rx.tap
    .subscribe(onNext: { _ in
        logIn()
            .subscribe(onNext: { user in
                print("\(user)")
            })
    })
}
```

flatMap

```
func logIn() -> Observable<User> { // ... }

let button: UIButton = ...

button.rx.tap // ControlEvent<Void>
    .flatMap { logIn() } // Observable<User>
    .subscribe(onNext: { user in
        print("\(user)")
    })
}
```

flatMapFirst

```
func logIn() -> Observable<User> { // ... }

let button: UIButton = ...

button.rx.tap // ControlEvent<Void>
    .flatMapFirst { logIn() } // Observable<User>
    .subscribe(onNext: { user in
        print("\(user)")
    })
}
```

flatMapLatest

```
func logIn() -> Observable<User> { // ... }

let button: UIButton = ...

button.rx.tap // ControlEvent<Void>
    .flatMapLatest { logIn() } // Observable<User>
    .subscribe(onNext: { user in
        print("\(user)")
    })
}
```

Tap count:
0

flatMap

flatMap - porównanie

```
let button: UIButton = ...  
  
button.rx.tap  
    .scan(0) { acc, _ in acc + 1 }  
    .flatMap { count -> Observable<Data> in  
        let url = "https://jsonplaceholder.typicode.com/todos/\\" + String(count) + "\"  
        let request = URLRequest(url: URL(string: url)!)  
  
        return URLSession.shared.rx.data(request: request)  
    }  
    .subscribe()
```

Tap count:
0

flatMap

Tap count:
0

flatMap
First

Tap count:
0

flatMap
Latest

Retry

```
ApiClient  
    .fetchUsers() // Observable<User>  
    .retry(3)  
    .subscribe(onNext: { users in  
        print("Users: \(users)")  
    })
```

Wielokrotne subskrypcje

```
let nameLabel: UILabel = ...
```

```
let surnameLabel: UILabel = ...
```

```
let request: Observable<User> = ...
```

```
request
```

```
    .map { $0.firstName }  
    .bind(to: nameLabel.rx.text)
```

```
request
```

```
    .map { $0.lastName }  
    .bind(to: surnameLabel.rx.text)
```

Share replay

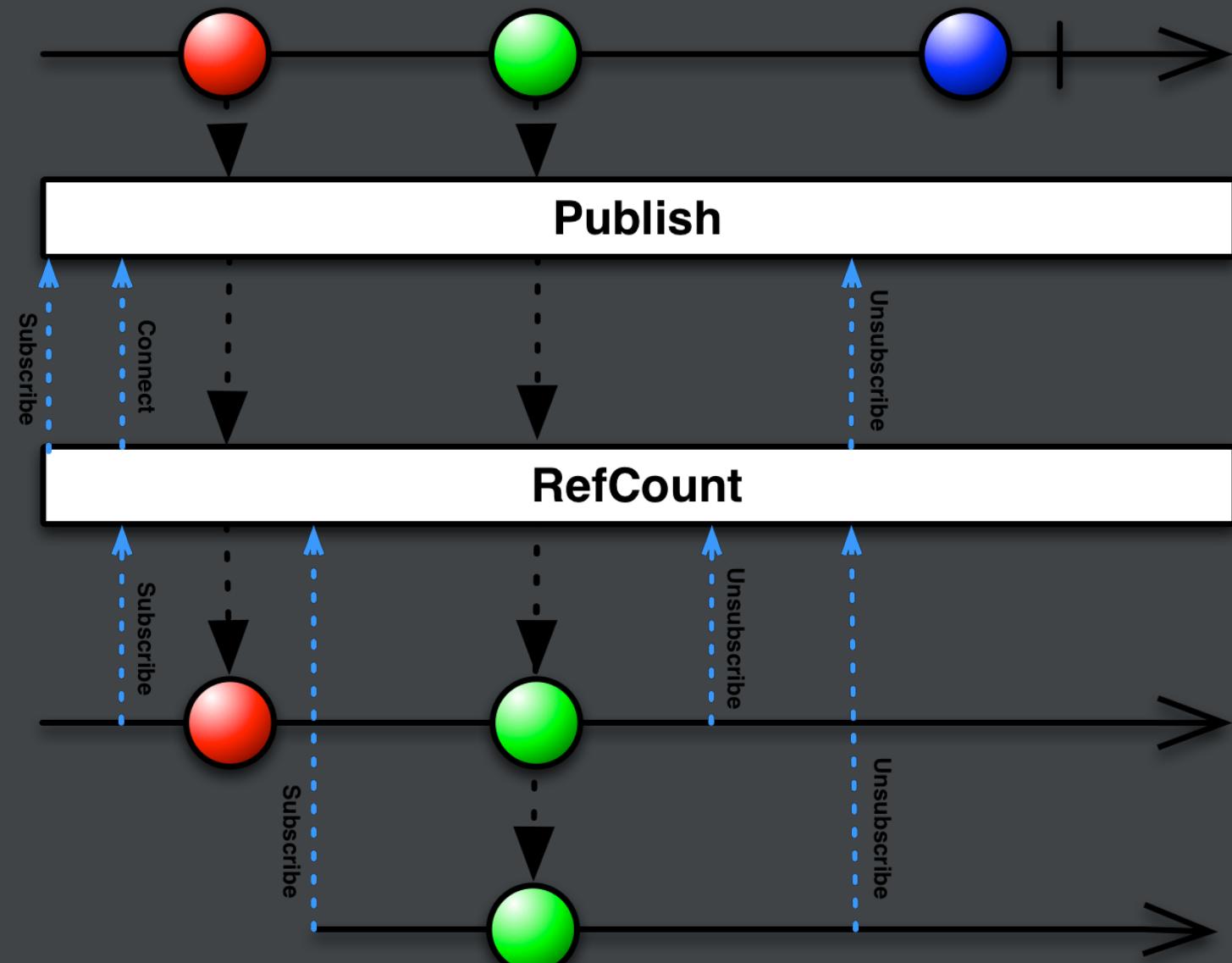
```
let nameLabel: UILabel = ...  
let surnameLabel: UILabel = ...
```

```
let request: Observable<User> = ...  
let sharedRequest = request.share(replay: 1)
```

```
sharedRequest  
.map { $0.firstName }  
.bind(to: nameLabel.rx.text)
```

```
sharedRequest  
.map { $0.lastName }  
.bind(to: surnameLabel.rx.text)
```

share(replay: 0)



Networking Traits

- Single
- Completable

Single

- Emitemy dokładnie jeden **success** lub **error**.
- Idealny do obsługi requestów do API, które zwracają wartość.

```
func fetchUsername() -> Single<String> { /* ... */ }

fetchUsername()
    .subscribe(onSuccess: { username in
        print("Received username: \(username)")
    }, onError: { error in
        print("Received error: \(error)")
    })
}
```

Completable

- Emittuje **complete** lub **error**.
- Idealny do obsługi requestów do API, które nie zwracają wartości.

```
func notifyUser(_ username: String) -> Completable { /* ... */ }

notifyUser("username")
    .subscribe(onCompleted: {
        print("Completed")
    }, onError: { error in
        print("Received error: \(error)")
    })
}
```

zadania

Zadanie #1

W klasie `TimetableViewController` zamienić `timetableService` na zależność typu `HTTPTimetableService` i uruchomić aplikację. Zastanowić się:

- Ile zmian w interfejsie wymagało wczytywanie danych z API względem pliku lokalnego?
- Kiedy wczytywane są dane z API?

Zadanie #2

W klasie `TimetableViewController` zamienić
`timetableService` na zależność typu
`WebSocketTimetableService` i uruchomić aplikację.

Zastanowić się:

- Kiedy wczytywane są dane z API?

Zadanie #3

W klasie `HTTPTimetableService` zaimplementowano pobieranie danych z wykorzystaniem `Observable.create`. Uprość kod za pomocą metody `.rx.response` na `URLSession` i wykorzystaniu funkcji `flatMap`.

Pamiętaj o zweryfikowaniu poprawności za pomocą testów jednostkowych.

Zadanie #4

Dodaj plik

`TimetableViewController+ActivityIndicatorSpec.swift` do
targetu `Workshops-Module2Tests`.

Zadanie #4

W klasie `TimetableViewController` za pomocą

`ActivityIndicator` zaimplementować progress HUD w trakcie pobierania danych.

Wykorzystaj `UIApplication.shared.rx.progress`.

Pamiętaj o zweryfikowaniu poprawności za pomocą testów jednostkowych.

Zadanie #5

Dodaj plik

`TimetableViewController+PullToRefreshSpec.swift` do
targetu `Workshops-Module2Tests`.

Zadanie #5

W klasie `TimetableViewController` zaimplementuj mechanizm pull to refresh. Dane mają być załadowane po wejściu na ekran i za każdym pociągnięciem kontrolki.

Podpowiedź: wykorzystaj statyczny strumień (`Observable.just(/*...*/)`), operator `merge` i metodę `UIRefreshControl.rx.controlEvent(.valueChanged)`.

Pamiętaj o zweryfikowaniu poprawności za pomocą testów jednostkowych.

Zadanie #6

Dodaj plik `CheckInViewControllerSpec.swift` do targetu
`Workshops-Module2Tests`.

Zadanie #6

W klasie `CheckInViewController` zaimplementować wywołanie metody `checkInService.checkIn` jako parametr `username` podając wartość z pola `checkInView.nameTextField`.

Wykorzystać metodę `UIButton.rx.tap` oraz operatory `withLatestFrom` oraz `flatMapLatest`.

Pamiętaj o zweryfikowaniu poprawności za pomocą testów jednostkowych.

Spostirzezenia

RxSwift pozwala mi skupić się na poprawnym przepływie danych. Nie muszę ciągle martwić się o poprawny stan widoków.

Nauczenie się RxSwift zajmuje czas, ale deklaratywny kod jest czystszy. RxSwift to nie tylko framework - to podejście, które pozwala skupić się na logice biznesowej, czyli tym co jest ważne w aplikacji.

Zalety

- Spójna obsługa asynchronicznego kodu.
- Upraszczanie problemów zarządzania stanem.
- Prosta obsługa zależności czasowych (opóźnienia, interwały, throttling).
- Automatyczne zarządzanie cyklem życia zasobów.
- Abstrakcja dostępna dla wielu języków programowania i platform.

Wady

- Szeroki interfejs **Observable**.
- Nieoczywiste problemy z zarządzaniem strumieniami/
subskrypcjami:
 - mnożenie zdarzeń,
 - nieoczekiwane przerwanie subskrypcji przy błędach.
- Nieoczywiste zależności pomiędzy zdarzeniami.
- Debugowanie.

Dziękujemy!