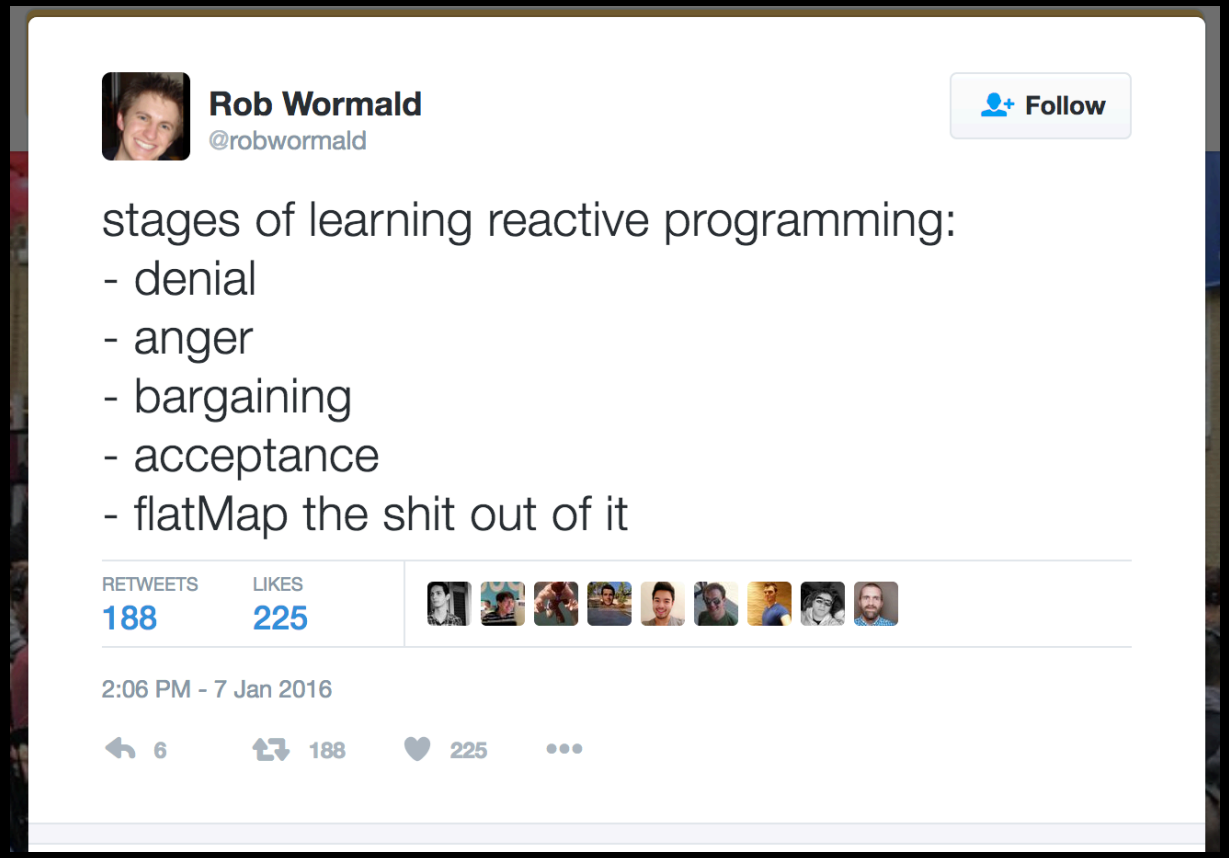
PROGRAMOWANIE REAKTYWNE

wprowadzenie

JAK SIĘ REKLAMUJE

- **Pozwala łatwo łączyć asynchroniczne zdarzenia**
- **… w szczególności zdarzenia UI**
- **Ułatwia wielowątkowość**
- **Jest używane komercyjnie**
- **Jest dostępne na wiele platform (Rx.Net, RxJava, RxJs, RxSwift, RxCpp …)**
- **Bla bla bla, fajne, bądź fajny używaj, bla bla bla, wszystko tym zamodelujesz, bla bla bla, czytelne bla bla, no callback hell bla bla, utrzymywalny kod bla bla bla bla bla bla**

NAUKA


Rob Wormald
@robwormald
Follow
stages of learning reactive programming:
- denial
- anger
- bargaining
- acceptance
- flatMap the shit out of it
RETWEETS
188
LIKES
225
2:06 PM - 7 Jan 2016
6
188
225

NO OK. ALE CZYM TO JEST?

- Generyczna implementacja wzorca projektowego Obserwator
- Mamy strumienie danych (Observable), które za pomocą wbudowanych operatorów przekształcamy w strumień o interesujących nas własnościach (łączenie, map, flatMap, fold, filter, opóźnienie, wątek …)
- Ktoś kiedyś zasubskrybuje się na nasz strumień i będzie dostawać powiadomienia o pojawieniu się nowych wartości (w sumie za bardzo nas ten ziomek nie obchodzi)
- Dużo sznureczków z kropeczkami

OBSERVABLE - PODSTAWY

- Ciąg kolejnych zdarzeń na przestrzeni czasu
- Zdarzenia:
  - `next(wartość)` – pojawiła się nowa wartość
  - `completed` – strumień zakończył się sukcesem
  - `error(błąd)` – strumień zakończył się błędem

OBSERVABLE - SUBSKRYBCJA

- `func subscribe(calback: (event: Event) -> void) -> Subscription`
- `callback` zostanie wykonany dla kolejnych zdarzeń emitowanych przez strumień
- Zwracana wartość reprezentuje subskrypcję.
  Tak długo jak istnieje, będziemy dostawać powiadomienia.
- Jest to dość częste miejsce wycieków pamięci.
  Subskrypcja trzyma silną referencję na `callback`.
  Czasem intuicyjne jest aby `callback` trzymał silną referencję na `self`

OBSERVABLE - PRZYKŁAD

```
class GPSLabel: UILabel {
    let subscription: Disposable

    init(gpsStream: Observable<Location>) {
        super.init()
        subscription = gpsStream.subscribe {
            [weak self] (event: Event<Location>) -> () in
            switch event {
                case Event<Location>.next(let location):
                    self?.text = "\(location.n)N, \(location.e)E”
                case Event<Location>.completed:
                    self?.text = "zakończono GPS”
                case Event<Location>.error(let error):
                    self?.text = "błąd GPS"
            }
        }
        subscription.dispose()
    }
}
```

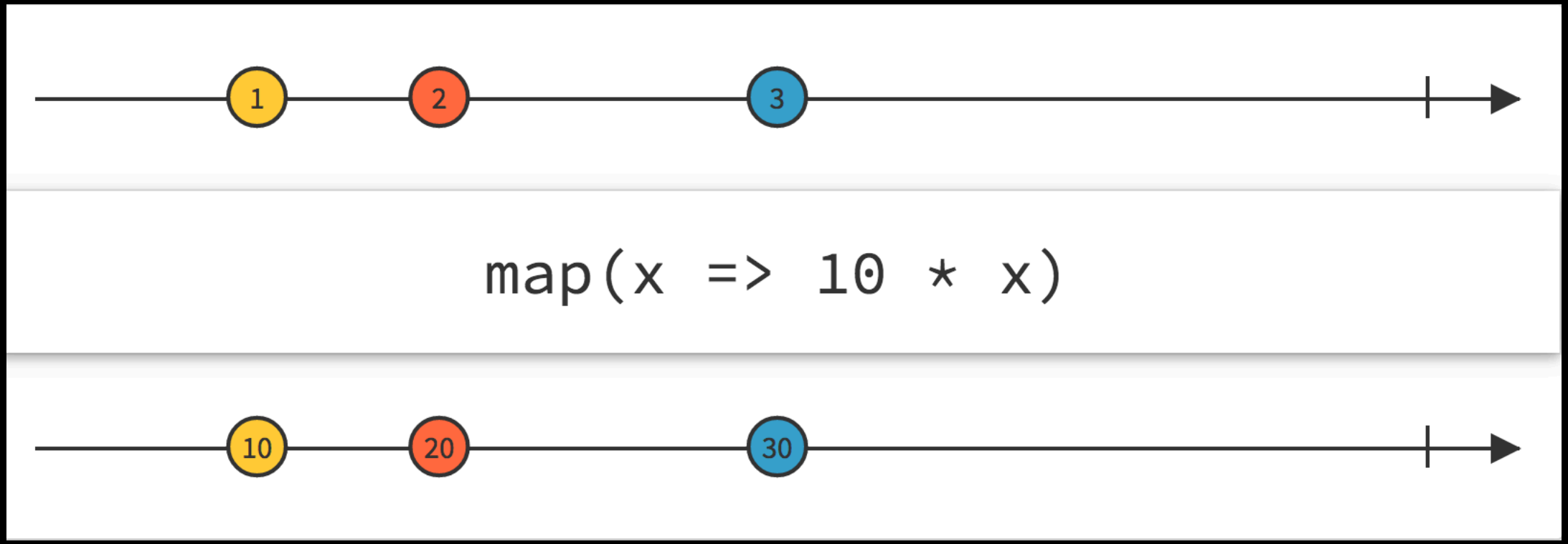
OBSERVABLE

- Są dwa typy strumieni:
  - Zimne (cold) – generują zdarzenia dopiero po tym jak ktoś się zasubskrybuje na dany strumień
  - Gorące (hot) – generują zdarzenia od momentu powstania
- Można je zamieniać za pomocą wielu gotowych operatorów

OPERATORY

- http://rxmarbles.com/
- http://reactivex.io/documentation/operators.html
- Wiekszość sensownych operatorów list jest zaimplementowana (map, flatMap, fold, filter itp.)
- Niektóre implementacje dodają własne operatory
- Niektóre implementacje trochę inaczej je nazywają
- Omówię jedynie najbardziej podstawowe
- Uwaga, wszystkie operatory zakładają że przekazywane im funkcje nie zależą od stanu zewnętrznego.

OPERATORY-MAP



OPERATORY-MAP

- `Observable<A>.map((A) -> B) -> Observable<B>`
- Nowy strumień powstaje poprzez aplikacji funkcji do każdego elementu bazowego strumienia
- Błąd na strumieniu bazowym powoduje pojawienie się błędu na nowym strumieniu
- Rozpoczęcie subskrypcji na strumieniu wynikowym jest przekazywane do strumienia wynikowego
  Czyli jeśli bazowy strumień był zimny to wynikowy również będzie
- Propaguje błędy bazowego strumienia
- Kończy się gdy bazowy strumień się zakończy

MAP - PRZYKŁAD

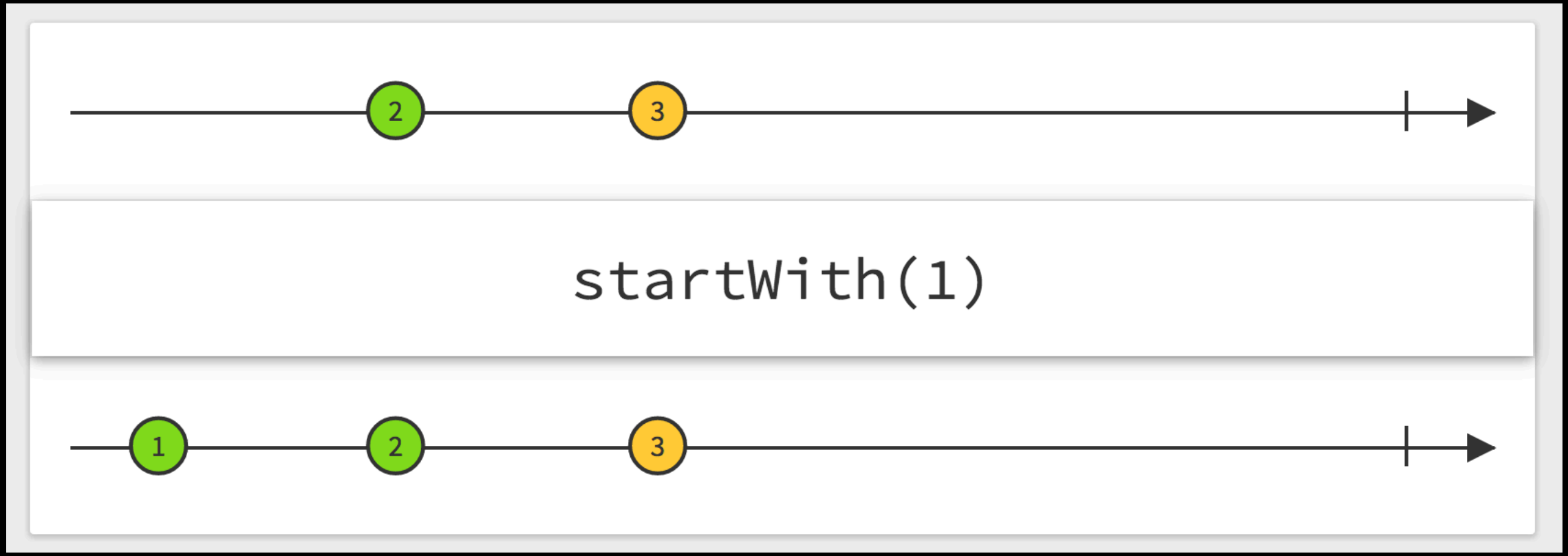
```
let stream: Observable<Int> =
    Observable<Int>.range(start: 0, count: 4)
    // 0, 1, 2, 3

let strings: Observable<String> = stram.map {
    (number: Int) -> String in
    return "\(number) to liczba"
}

strings.subscribe {
    (event: Event) -> () in
    print(event)
}.addDisposableTo(disposeBag)
```

```
// stream 0
Next("0 to liczba")
// stream 1
Next("1 to liczba")
// stream 2
Next("2 to liczba")
// stream 3
Next("3 to liczba")
// stream Completed
Completed
```

OPERATORY-START WITH



OPERATORY-START WITH

- `Observable<A>.startWith(A) -> Observable<A>`
- Dokleja dany element na początek strumienia
- Wynikowy strumień zawsze ma początkowy element (ma to znaczenie dla niektórych operatorów)

START WITH- PRZYKŁAD

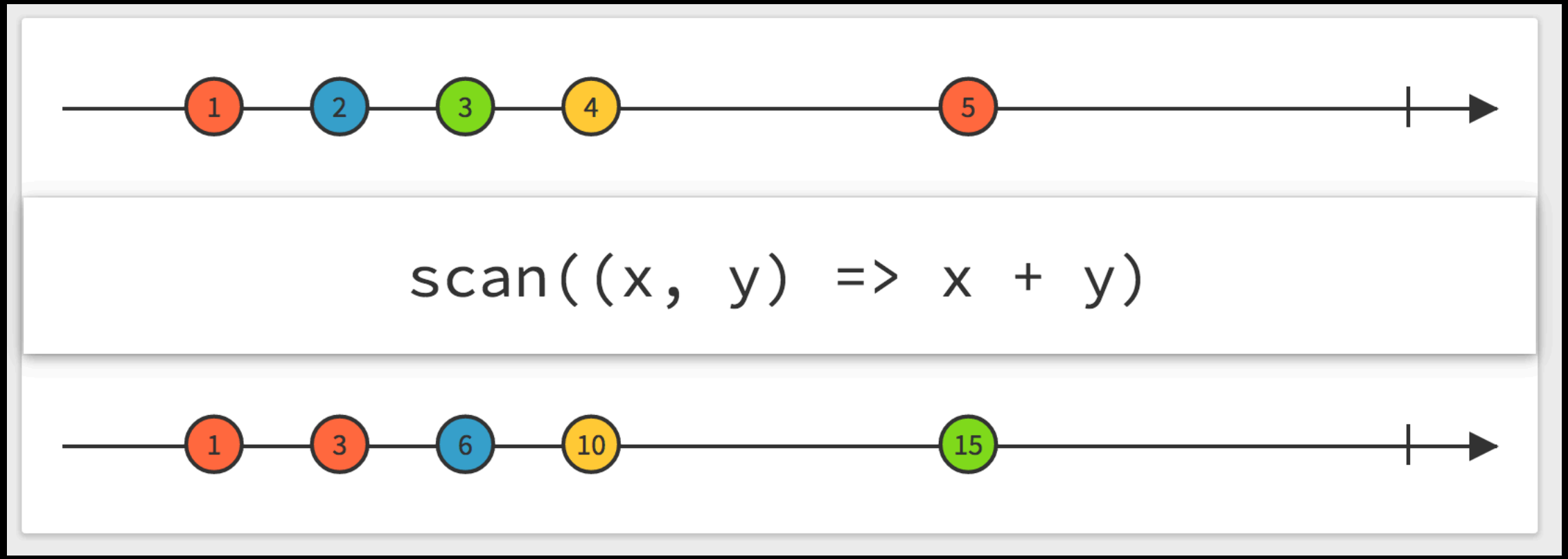
```
let stream: Observable<Int> =
    Observable<Int>.range(start: 0, count: 3)
    // 0, 1, 2, 3

let ints: Observable<Int> = stram.startWith(-1)

ints.subscribe {
    (event: Event) -> () in
    print(event)
}.addDisposableTo(disposeBag)
```

```
Next(-1)
// stream 0
Next(0)
// stream 1
Next(1)
// stream 2
Next(2)
// stream 3
Next(3)
// stream Completed
Completed
```

OPERATORY - SCAN



OPERATORY-SCAN

- `Observable<A>.scan(seed: B, (B,A) -> B) -> Observable<B>`
- W normalnym świecie nazywałby się foldem
- Nowy strumień generowany jest w poprzez aplikację przekazanej funkcji do ostatniego elementu nowego strumienia (albo `seed` jeśli generujemy pierwszy element) oraz elementu bazowego strumienia
- Przydatny gdy chcemy zdobyć strumień stanu obiektu mając strumień jego zmian
- Kończy się gdy bazowy strumień się zakończy
- Propaguje błąd bazowego strumienia

SCAN - PRZYKŁAD

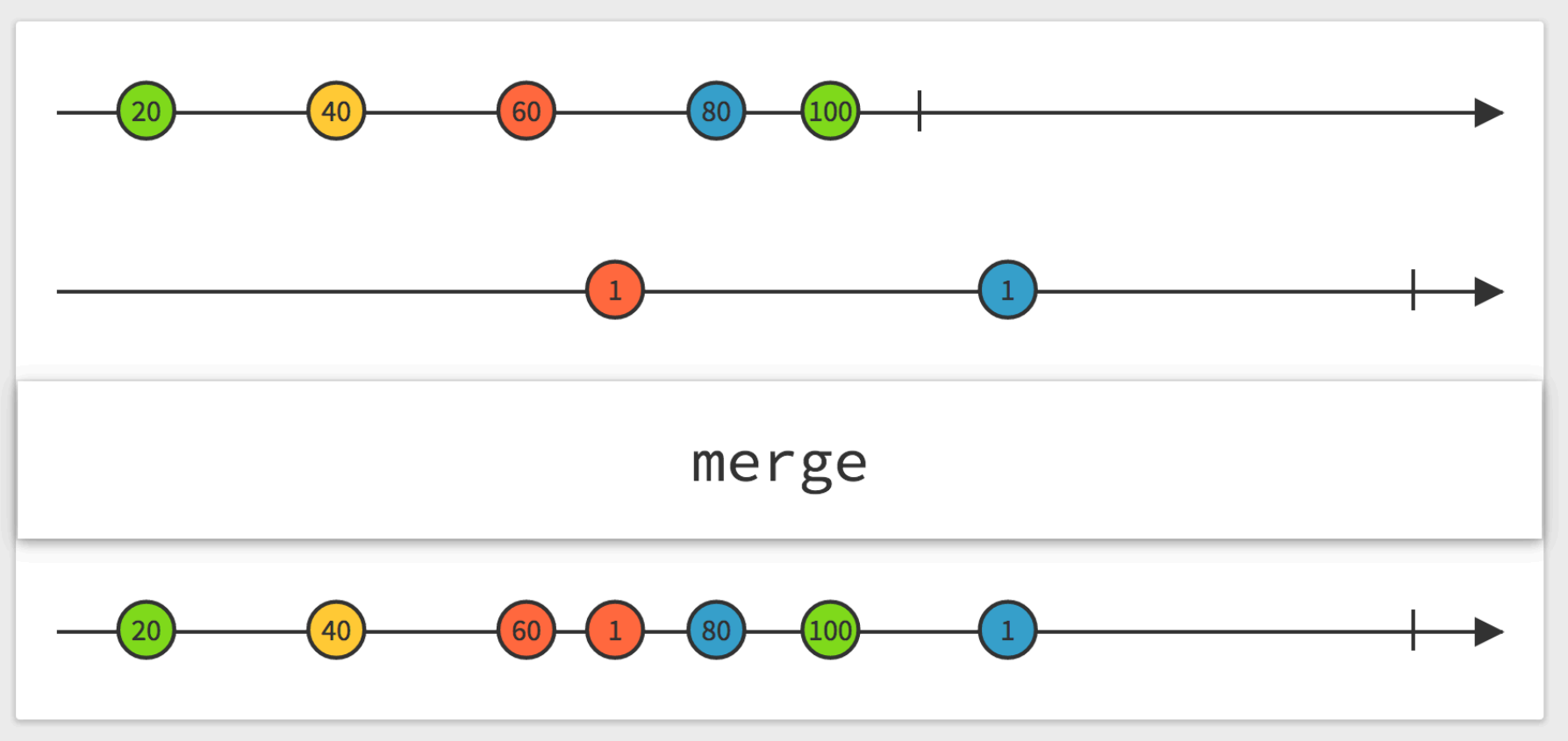
```
let stream: Observable<Int> =
    Observable<Int>.range(start: 0, count: 4)
    // 0, 1, 2, 3

let strings: Observable<String> = stram.scan("") {
    (value: String, change: Int) -> String in
    return value + "\(change)"
}

strings.subscribe {
    (event: Event) -> () in
    print(event)
}.addDisposableTo(disposeBag)
```

```
// stream 0
Next("0")
// stream 1
Next("01")
// stream 2
Next("012")
// stream 3
Next("0123")
// stream Completed
Completed
```

OPERATORY - MERGE



OPERATORY - MERGE

- `merge(Observable<A>, Observable<A>) -> Observable<A>`
- Kolejne wartości to kolejne wartości strumieni na przestrzeni czasu
- Kończy się gdy oba strumienie się zakończą
- Emituje błąd gdy któryś ze strumieni wyemituje błąd

MERGE- PRZYKŁAD

```
let ints1: Observable<Int>
let ints2: Observable<Int>
    //  1 2       3
    //      40 70

let hmm: Observable<Int> =
    Observable<Int>.of(
        ints1, ints2
    ).merge()

hmm.subscribe {
    (event: Event) -> () in
    print(event)
}.addDisposableTo(disposeBag)
```

```
// ints1 1
Next(1)
// ints1 2
Next(2)
// ints2 40
Next(40)
// ints2 70
Next(70)
// ints2 Completed
// ints1 3
Next(3)
// ints1 Completed
Completed
```

PRZERYWAJKA 1

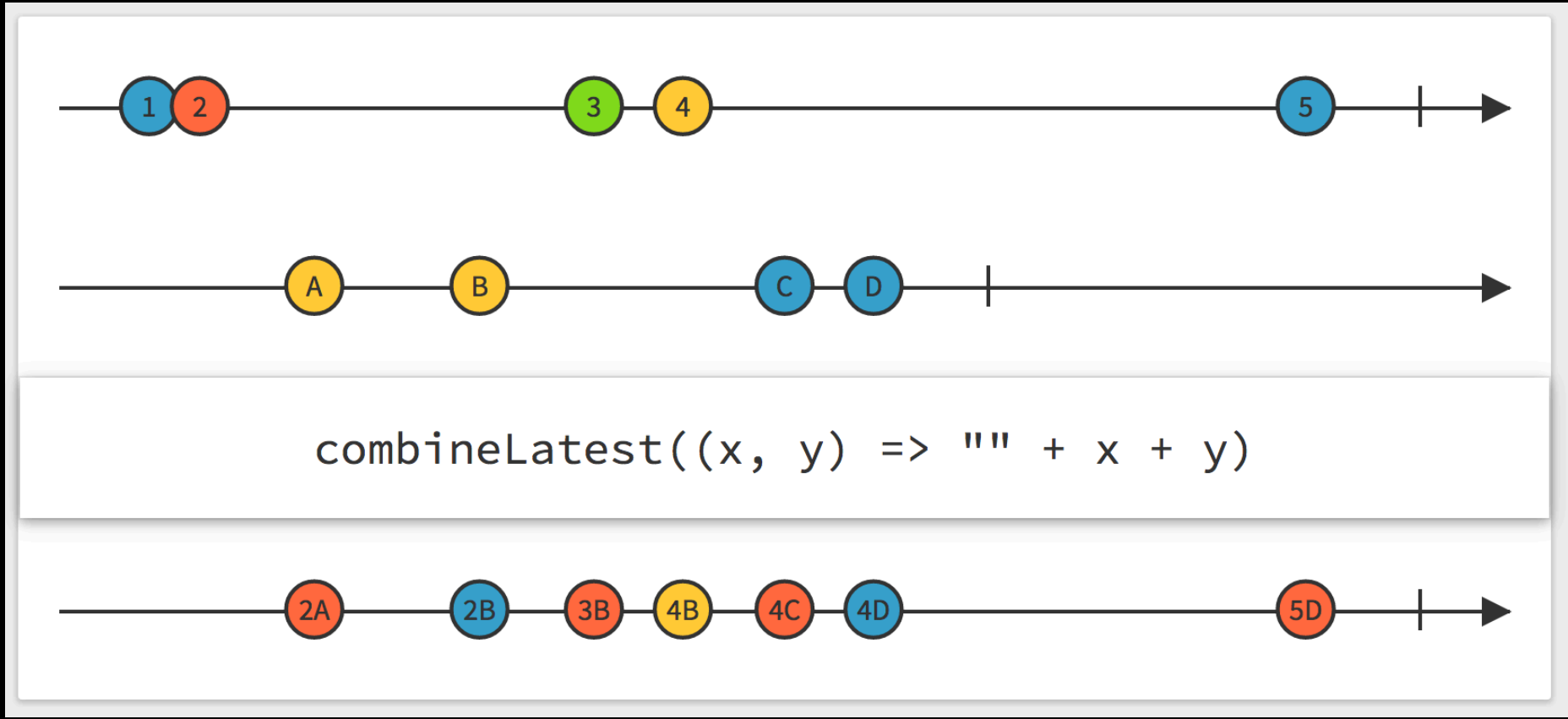
- Mamy 2 konta bankowe
- Dla każdego konta mamy strumień zmian ilości środków (wpłacono x$, wypłacono Y$ …)
- Początkowa wartość obydwu kont to 0$
- Jak otrzymać strumień sumy środków na obydwu kontach?

PRZERYWAJKA 1

```
let konto1: Observable<Int>
let konto2: Observable<Int>

let hajs: Observable<Int> = Observable<Int>.of(
    konto1, konto2
).merge().scan(0) {
    (value: Int, change: Int) -> Int in
    return  value + change
}.startWith(0)
```

OPERATORY – COMBINE LATEST



OPERATORY – COMBINE LATEST

- `combineLatest(Observable<A>, Observable<B>, (A, B) -> C) -> Observable<C>`
- Łączy 2 strumienie w 1
- Czeka aż oba strumienie wygenerują co najmniej 1 wartość (zapomina wartości szybszego strumienia)
- Elementy to wartości przekazanej funkcji dal kolejnych wartości
- Kolejne elementy są generowane za każdym razem gdy któryś ze strumieni wygeneruje jakąś wartość.
- Kończy się gdy obydwa strumienie się zakończą
- Generuje błąd gdy któryś ze strumieni wygeneruje błąd

COMBINE LATEST- PRZYKŁAD

```
let ints: Observable<Int>
let strings: Observable<String>
    //  1 2           3
    //      "0" "01”

let hmm: Observable<Double> =
    Observable<Double>.combineLatest(
        ints, strings
    ) {
        (int: Int, str: String) -> Double in
        return Double(int) / Double(str.characters.count)
    }

hmm.subscribe {
    (event: Event) -> () in
    print(event)
}.addDisposableTo(disposeBag)
```

```
// ints 1
// ints 2
// strings "0"
Next(2.0)
// strings "01"
Next(1.0)
// strings Completed
// ints 3
Next(1.5)
// ints Completed
Completed
```

PRZERWYAJKA 2

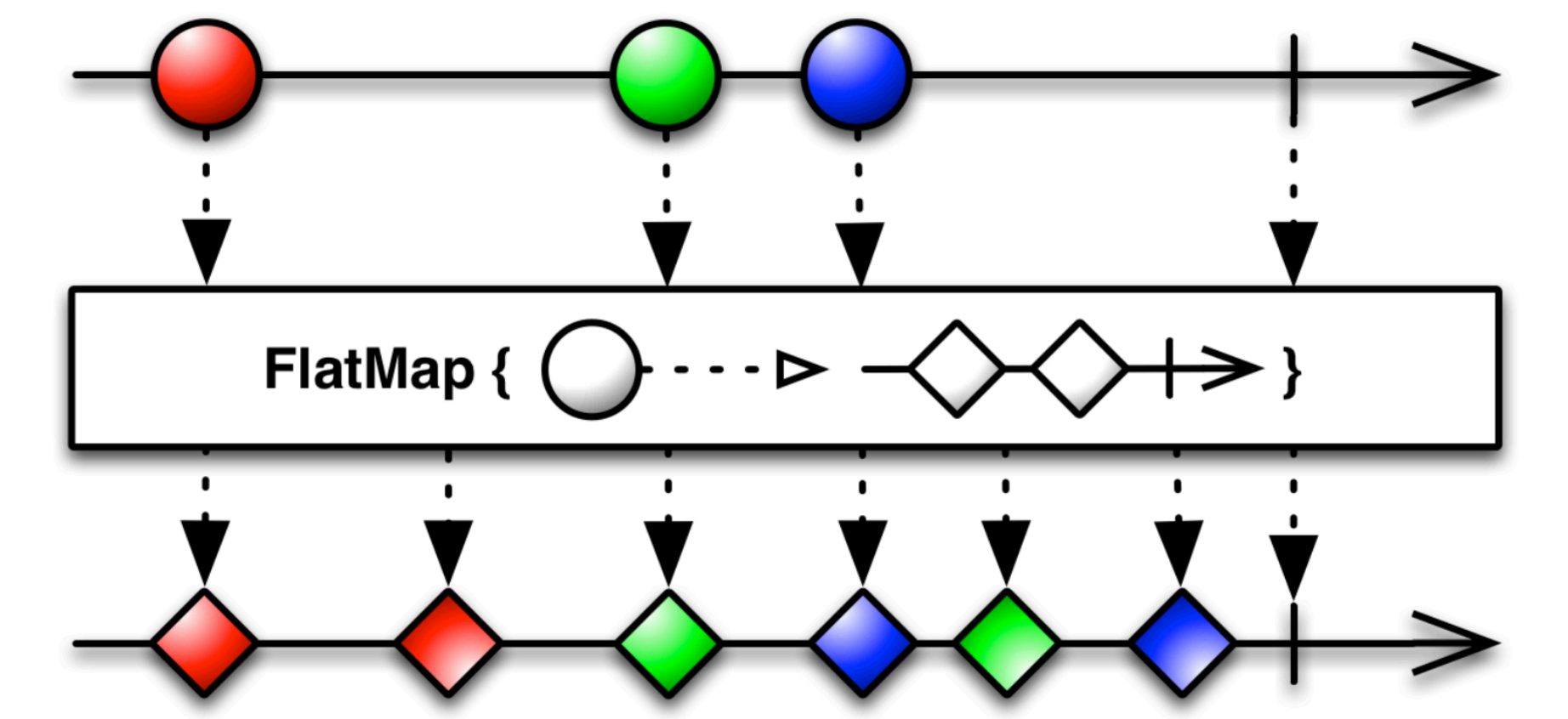
- Wracamy do liczenia $$$
- Tym razem mamy strumienie aktualnego stanu każdego konta
- Znów chcemy wyliczyć ile mamy pieniędzy

PRZERYWAJKA 2

```
let konto1: Observable<Int>
let konto2: Observable<Int>

let hajs = Observable<Int>.combineLatest(
    konto1.startWith(0),
    konto2.startWith(0)
) {
    (konto1: Int, konto2: Int) -> Int in
    return konto1 + konto2
}
```

OPERATORY – FLAT MAP



OPERATORY – FLAT MAP

- `Observable<A>.flatMap(A -> Observable<B>) -> Observable<B>`
- Dla każdego elementu generujemy nowy strumień
- W efekcie „mamy listę wielu strumieni”
- Mergujemy je
- W efekcie otrzymujemy wynikowy strumień
- Błąd na którymkolwiek ze strumieni powoduje błąd na wynikowym strumieniu
- Kończy się gdy wszystkie utworzone strumienie się zakończą

FLAT MAP - PRZYKŁAD

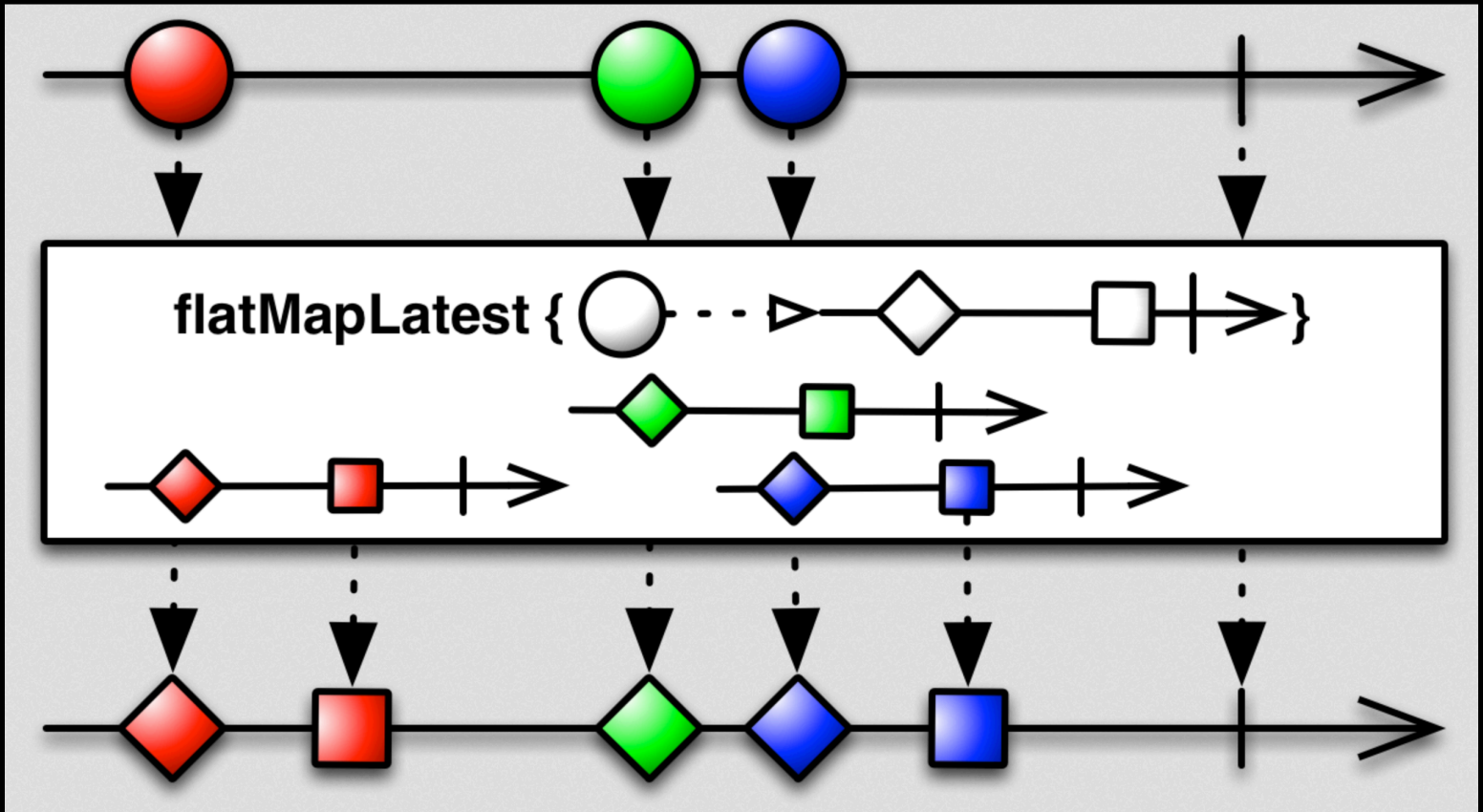
```
/// zwraca strumień który emituje dany element,
/// czeka 10s i ponownie go emituje, po czym się
/// kończy
func getStream(Int) -> Observable<Int>
let ints: Observable<Int>
    // 1 2

let hmm = ints.flatMap {
    (elem: Int) -> Observable<Int> in
    return getStream(elem)
}

hmm.subscribe {
    (event: Event) -> () in
    print(event)
}.addDisposableTo(disposeBag)
```

```
// ints 1
// s1 1
Next(1)
// ints 2
// s2 2
Next(2)
// ints Completed
// s1 1
Next(1)
// s1 Completed
// s2 2
Next(2)
// s2 Completed
Completed
```

OPERATORY – FLAT MAP LATEST



OPERATORY – FLAT MAP LATEST

- `Observable<A>.flatMapLatest(A -> Observable<B>) -> Observable<B>`
- To samo co `flatMap` tylko patrzymy tylko na ostatni wygenerowany obserwator
- Kończy się gdy ostatni strumień się zakończy

FLAT MAP LATEST - PRZYKŁAD

```
/// zwraca strumień który emituje dany element,
/// czeka 10s i ponownie go emituje, po czym się
/// kończy
func getStream(Int) -> Observable<Int>
let ints: Observable<Int>
    // 1 2

let hmm = ints.flatMapLatest {
    (elem: Int) -> Observable<Int> in
    return getStream(elem)
}

hmm.subscribe {
    (event: Event) -> () in
    print(event)
}.addDisposableTo(disposeBag)
```

```
// ints 1
// s1 1
Next(1)
// ints 2
// s2 2
Next(2)
// ints Completed
// s2 2
Next(2)
// s2 Completed
Completed
```

PRZERYWAJKA 3

- $$$
- Mamy funkcję `getHajs() -> Observable<Int>`
- Zwracany strumień pobiera z serwera stan konta
- Mamy strumień kliknięć w guzik (`let guzik: Observable<Void>`)
- Chcemy strumień który dla każdego kliknięcia pobierze nam stan konta z serwera

PRZERYWAJKA 3

```
let wynik: Observable<Int> = guzik.flatMapLatest {
    (_: Void) -> Observable<Int> in
    return getHajs()
}
```

PRZERYWAJKA 3

```
let wynik: Observable<Int> = guzik.flatMapLatest {
    (_: Void) -> Observable<Int> in
    return getHajs()
}
```

Źle!

Pobieranie może się nie powieść

Wtedy strumień `getHajs()` rzuci błąd

Zostanie on przesłany do wynikowego strumienia

Od tego momentu wynikowy strumień nie będzie zwracać kolejnych wyników

Nawet jeśli guzki będzie klikany

PRZERYWAJKA 3

```
let wynik: Observable<Int> = guzik.flatMapLatest {
    (_: Void) -> Observable<Int> in
    return getHajs().retry(5).catchError {
        (error: Error) -> Observable<Int> in
        return Observable<Int>.empty()
    }
}
```

Można to poprawić w ten sposób.
`retry` spróbuje powtórzyć nasz strumień 5 razy (co to znaczy zależy od implementacji `getHajs`)
`catchError` złapie błąd i na koniec strumienia doklei zwracany strumień

PRZERYWAJKA 3

```
let wynik: Observable<Int> = guzik.flatMapLatest {
    (_: Void) -> Observable<Int> in
    return getHajs().retry(5).catchError {
        (error: Error) -> Observable<Int> in
        return Observable<Int>.empty()
    }
}
```

Raczej nie jest to dobre rozwiązanie.
Bezpowrotnie tracimy informację o błędzie pobierania.
Może zamiast `Observable<Int>` powinniśmy zwracać `Observable<(Int?, Error?)>`?

TWORZENIE - SUBJECT

- `Subject<A>` jest strumieniem, który implementuje protokół `Observer<A>` czyli posiada metody:
  - `onNext(A)` – powoduje emisję podanego elementu
  - `onError(Error)` – powoduje zakończenie z podanym błędem
  - `onCompleted()` – powoduje zakończenie
- Jest kilka typów Subjectów, najczęstsze to:
  - `PublishSubject` – jedynie emituje wartość (jak ktoś się zasubskrybuje później może nie otrzymać żadnej wartości, mimo że wrzuciliśmy jakieś)
  - `BehaviurSubject` – każdy subskrybent dostanie ostatnią emitowaną wartość (albo ustawioną wartość początkową)
  - `ReplaySubject` – każdemu subskrybentowi zwraca wszystkie elementy (trzyma historię)

SUBJECT-PRZYKŁAD

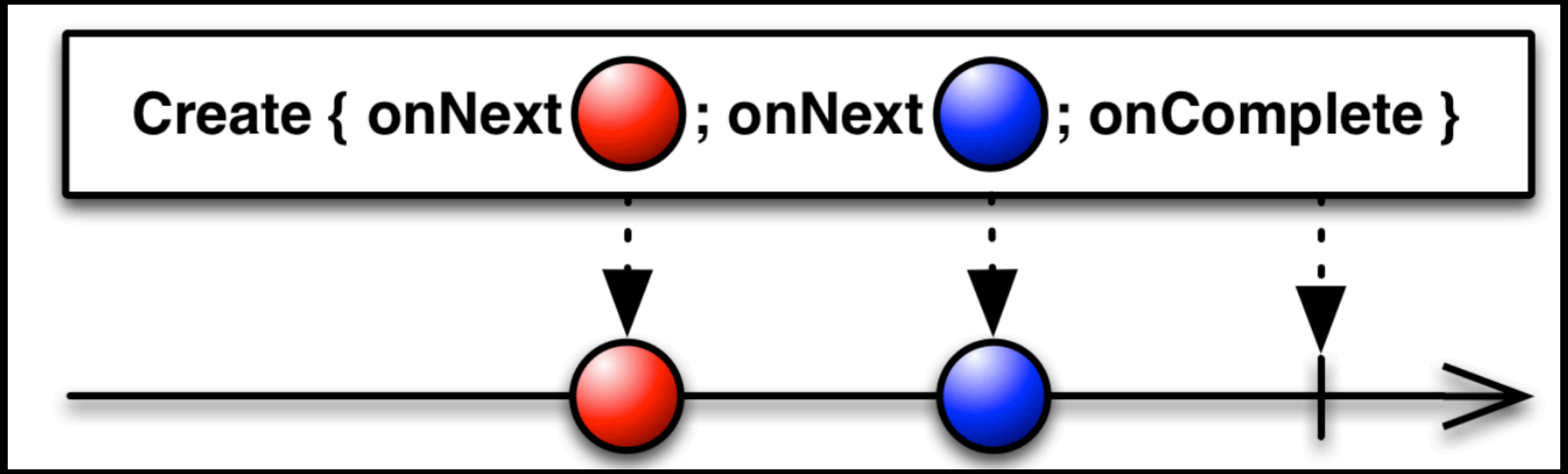
```
public class RxButton: Button {
    private let clickSubject: PublishSubject<Void> = PublishSubject<Void>()
    public  let clickStream: Observable<Void>

    init() {
        clickStream = clickSubject
    }

    deinit {
        clickSubject.onCompleted()
    }

    override func didClick() {
        super.didClick()
        clickSubject.onNext(Void())
    }
}
```

TWORZENIE - CREATE



TWORZENIE - CREATE

- `create((Observer<A>) -> Disposable) -> Observable<A>`
- Tworzy obserwator na podstawie przekazanej funkcji
- W momencie subskrypcji uruchamiana jest nasza funkcja
- Komunikuje się ona z naszym strumieniem za pomocą observera którego dostaje w parametrach (tj. za każdym razem gdy wywoła `onNext` nasz strumień wyemituje kolejny element)

CREATE - PRZYKŁAD

```
/// pobiera stan z konta z serwera
/// request pozwala nam anulować zapytanie
getHajs(completion: ((Int?, Error?) -> ())) -> Request

let hajsStream = Observable<Int>.create {
    (observer: AnyObserver<Int>) -> Disposable in
    let request: Request = getHajs {
        (value: Int?, error: Error?) -> () in
        if let value: Int = value && error == nil {
            observer.onNext(value)
            observer.onCompleted()
        } else {
            observer.onError(error ?? UnknownError())
        }
    }
    request.start()
    return AnonymousDisposable {
        request.cancel()  // to zostanie wykonane podczas usuwana subskrypcji
    }
}
```

PODSUMOWANIE

- Efekt końcowy zazwyczaj jest zadawalający
- Pozwala w czytelny sposób oddzielić UI od logiki aplikacji
- Można łatwo skakać między wątkami
- Jest podobne do programowania funkcyjnego (a co za tym idzie trzeba poświęcić dużo czasu aby wypracować jakąś intuicję)
- Łatwo popełnić błąd niszczący aplikację (flatMap)
- Sprawia wrażenie prostego, ale bez wcześniejszego planu, bardzo łatwo stracić panowanie nad projektem
- Czasem utrudnia proste rzeczy (jeśli mamy prostą logikę i możemy operować jedynie na 1 wątku to Rx będzie jedynie problemem