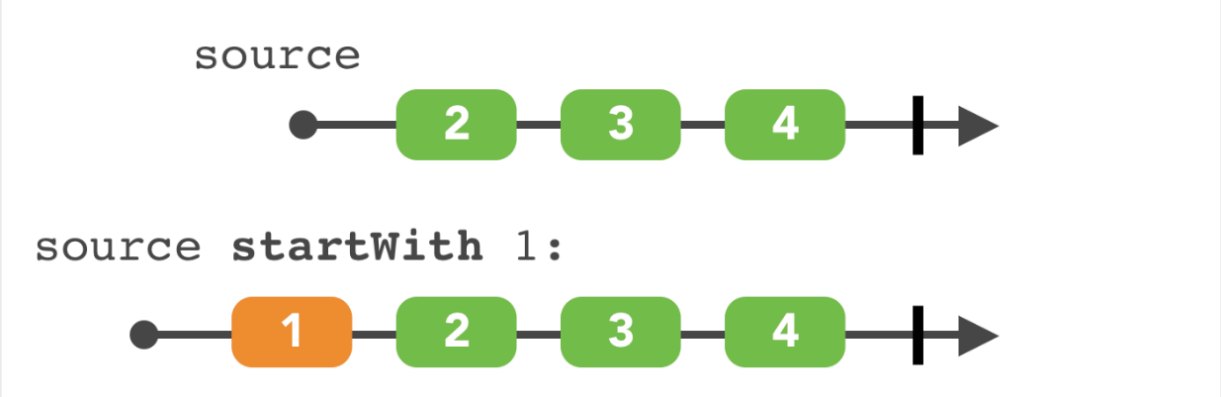
RXSwift - Combining Operator

Prefixing and concatenating

- **startWith**:
  - prefixes an observable sequence with the given initial value.
  - This value must be of the same type as the observable elements.
  - Your initial value is a sequence of one element, to which RxSwift appends the sequence that startWith(_:) chains to.

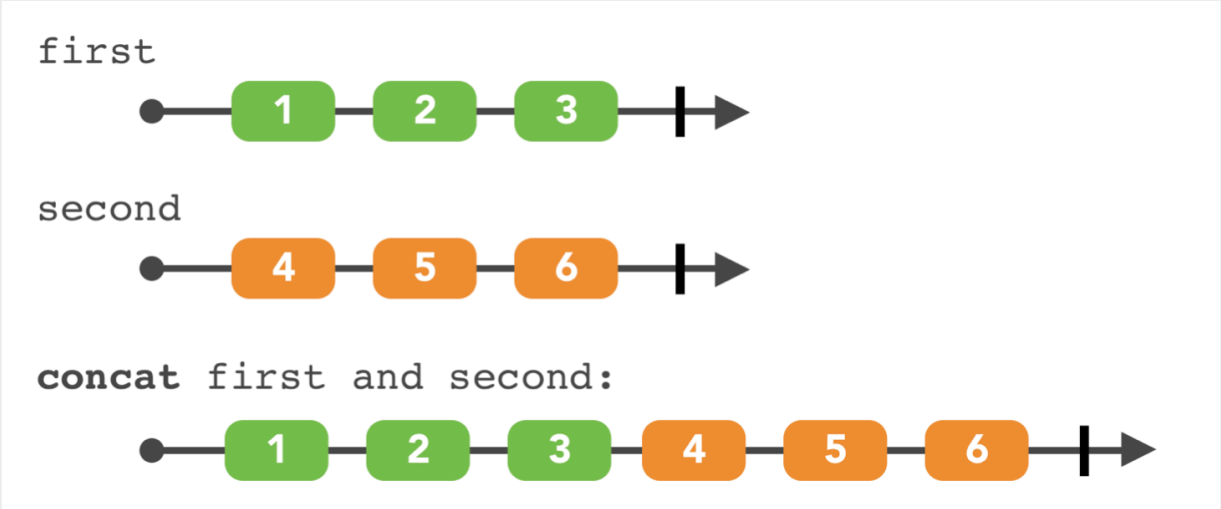



```
example(of: "startWith") {
  // 1
  let numbers = Observable.of(2, 3, 4)

  // 2
  let observable = numbers.startWith(1)
  _ = observable.subscribe(onNext: { value in
    print(value)
  })
}
```

```
1
2
3
4
```

- **concat:**

- Takes an **ordered collection of observables** (i.e. an array).
- It subscribes to the 1st sequence of the collection, relays its elements until it completes, then moves to the next one.
- The process repeats until all the observables in the collection have been used.
- If at any point an inner observable emits an error, the concatenated observable in turns emits the error and terminates.



EX1: Using **Observable.concat()**

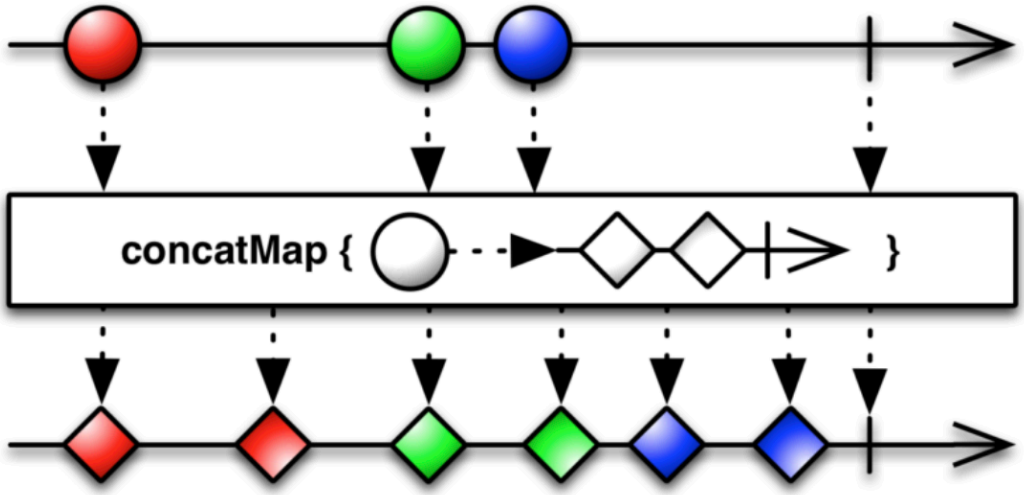
```
example(of: "Observable.concat") {
  // 1
  let first = Observable.of(1, 2, 3)
  let second = Observable.of(4, 5, 6)

  // 2
  let observable = Observable.concat([first, second])

  observable.subscribe(onNext: { value in
    print(value)
  })
}
```

EX2: Another way to append sequence using **concat()**

```
let germanCities = Observable.of("Berlin", "Münich", "Frankfurt")
let spanishCities = Observable.of("Madrid", "Barcelona", "Valencia")
```

```
    let observable = germanCities.concat(spanishCities)
    _ = observable.subscribe(onNext: { value in
      print(value)
    })
  }
```

- **concatMap**()
  - closely related to flatMap(_:)
  - guarantees that each sequence produced by the closure will **run to completion** before the next is subscribed to
  - a handy way to guarantee sequential order.



  - 1 observable (circle) emits 2 elements to subscriber (diamond shape)
  - toán tử đảm bảo việc các chuỗi được đóng lại trước khi chuỗi tiếp theo được đăng kí vào.

```
    // 1
    let sequences = [
      "German cities": Observable.of("Berlin", "Münich", "Frankfurt"),
      "Spanish cities": Observable.of("Madrid", "Barcelona", "Valencia")
    ]

    // 2
    let observable = Observable.of("German cities", "Spanish cities")
      .concatMap { country in sequences[country] ?? .empty() }

    // 3
    _ = observable.subscribe(onNext: { string in
        print(string)
      })
}
```

✅ Merging

- subscribes to each of the sequences it receives and emits the elements as soon as they arrive — there's no predefined order.
- The order in which the inner sequences complete is irrelevant.
  - If any of the sequences emit an error, the merge() observable immediately relays the error, then terminates.

-

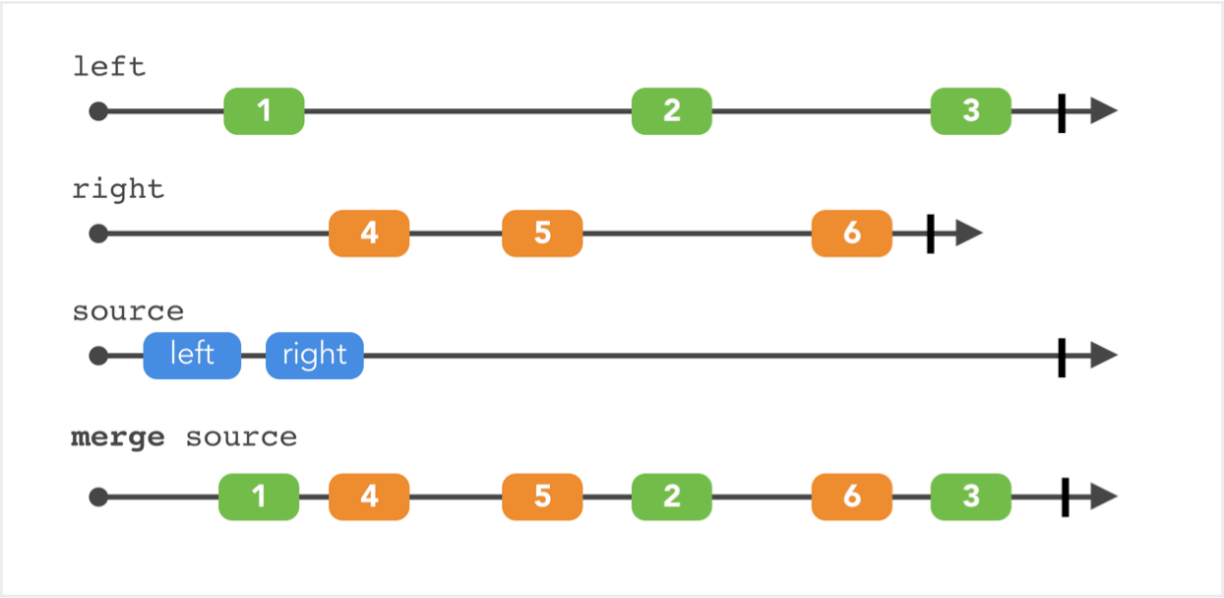
-

Q: when and how merge() completes?

A: merge() completes after its source sequence completes **and** all inner sequences have completed.

NOTE:

- merge() takes a *source* observable, which itself emits observables sequences of the element type. This means that you could send a *lot* of sequences for merge() to subscribe to!
  - To limit the number of sequences subscribed to at once, you can use **merge(maxConcurrent:)**.
  - This variant keeps subscribing to incoming sequences until it reaches the maxConcurrent limit.
  - After that, it puts incoming observables in a queue. It will subscribe to them in order, as soon as one of current sequences completes.




```
let chu = PublishSubject<String>()
let so = PublishSubject<String>()

let source = Observable.of(chu.asObserver(), so.asObserver())

let observable = source.merge()

observable
    .subscribe(onNext: { (value) in
        print(value)
    })
    .disposed(by: bag)

chu.onNext("Một")
so.onNext("1")
chu.onNext("Hai")
so.onNext("2")
chu.onNext("Ba")
so.onCompleted()
so.onNext("3")
chu.onNext("Bốn")
chu.onCompleted()
```

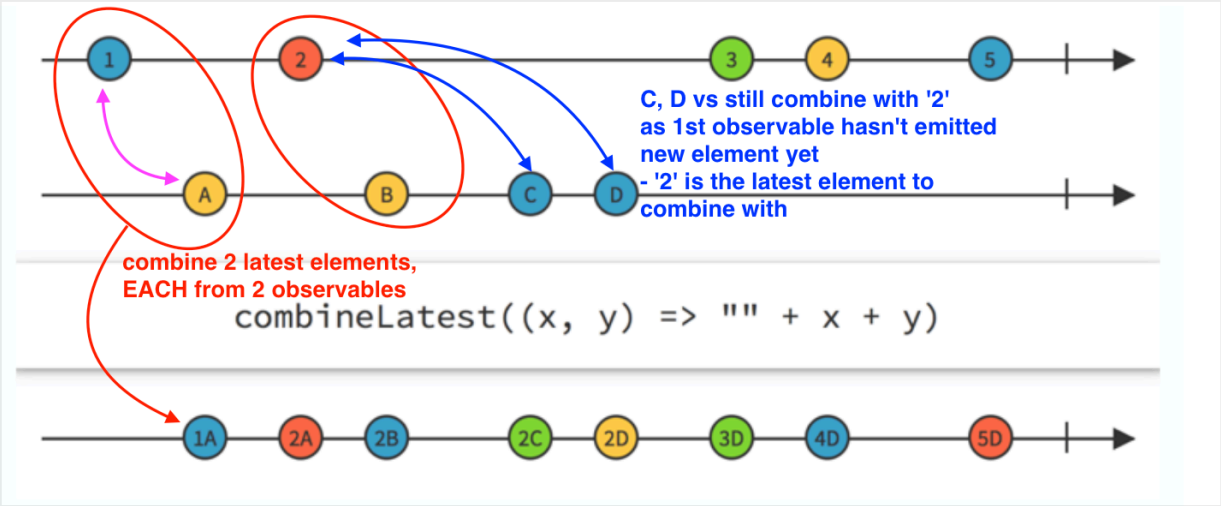
```
Một
1
Hai
2
Ba
Bốn
```

**NOTE:

- merge() không quan tâm tới thứ tự các Observable được thêm vào. Nên nếu có bất cứ phần tử nào từ các Observable đó phát đi thì Subscriber cũng đều nhận được
- Observable của merge sẽ kết thúc khi tất cả đều kết thúc

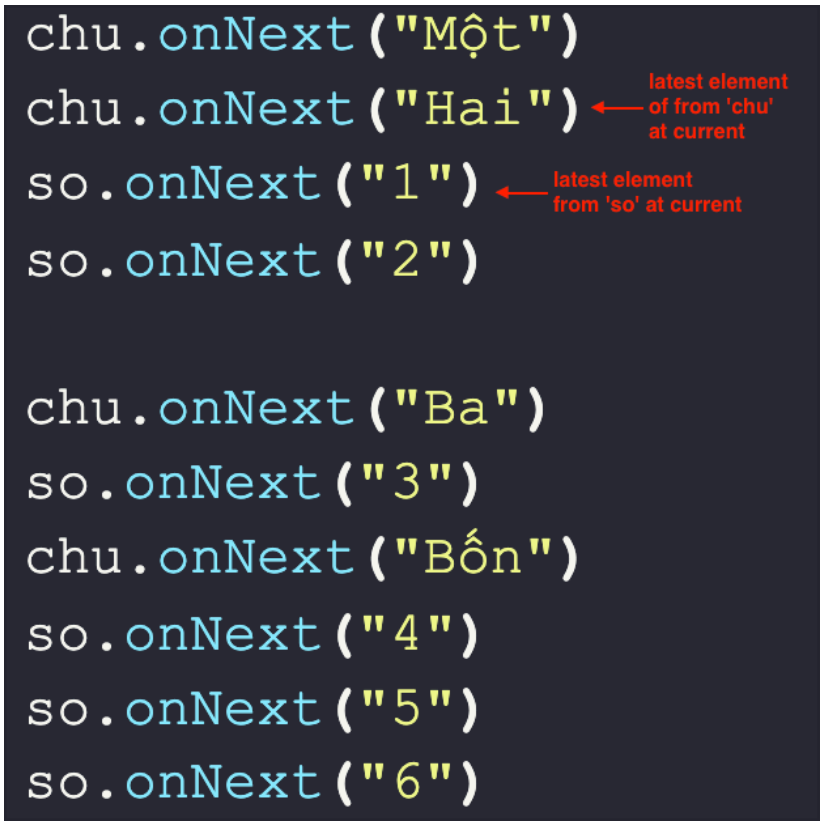
✅ Combining elements

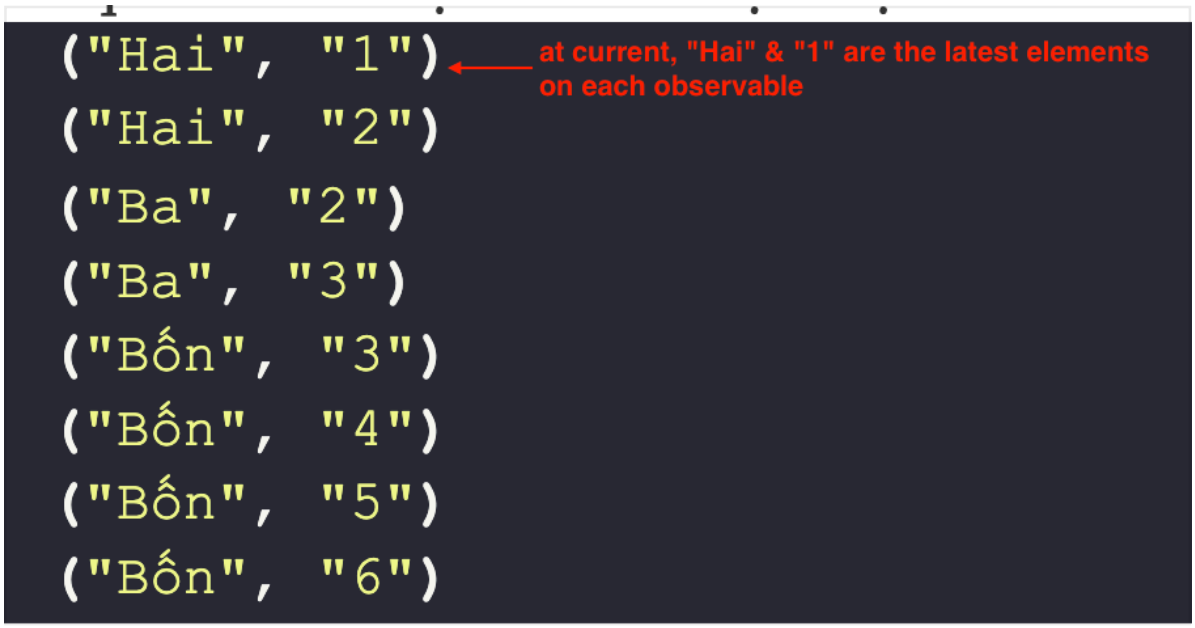
- **combineLatest**
- You receive the last value emitted by each of the inner sequences.
- Every time one of the inner (combined) sequences emits a value, it calls the closure you provide.
- Nothing happens until each of the combined observables emits one value.
    - After that, each time one emits a new value, the closure receives the **latest** value of each of the observable and produces its element.
- **combineLatest** take between two and eight observable sequences as parameters.
- Sequences don’t need to have the same element type.
    - Usage example : This has many concrete applications, such as observing several text fields at once and combining their values, watching the status of multiple sources, and so on.




```
let chu = PublishSubject<String>()
let so = PublishSubject<String>()
```

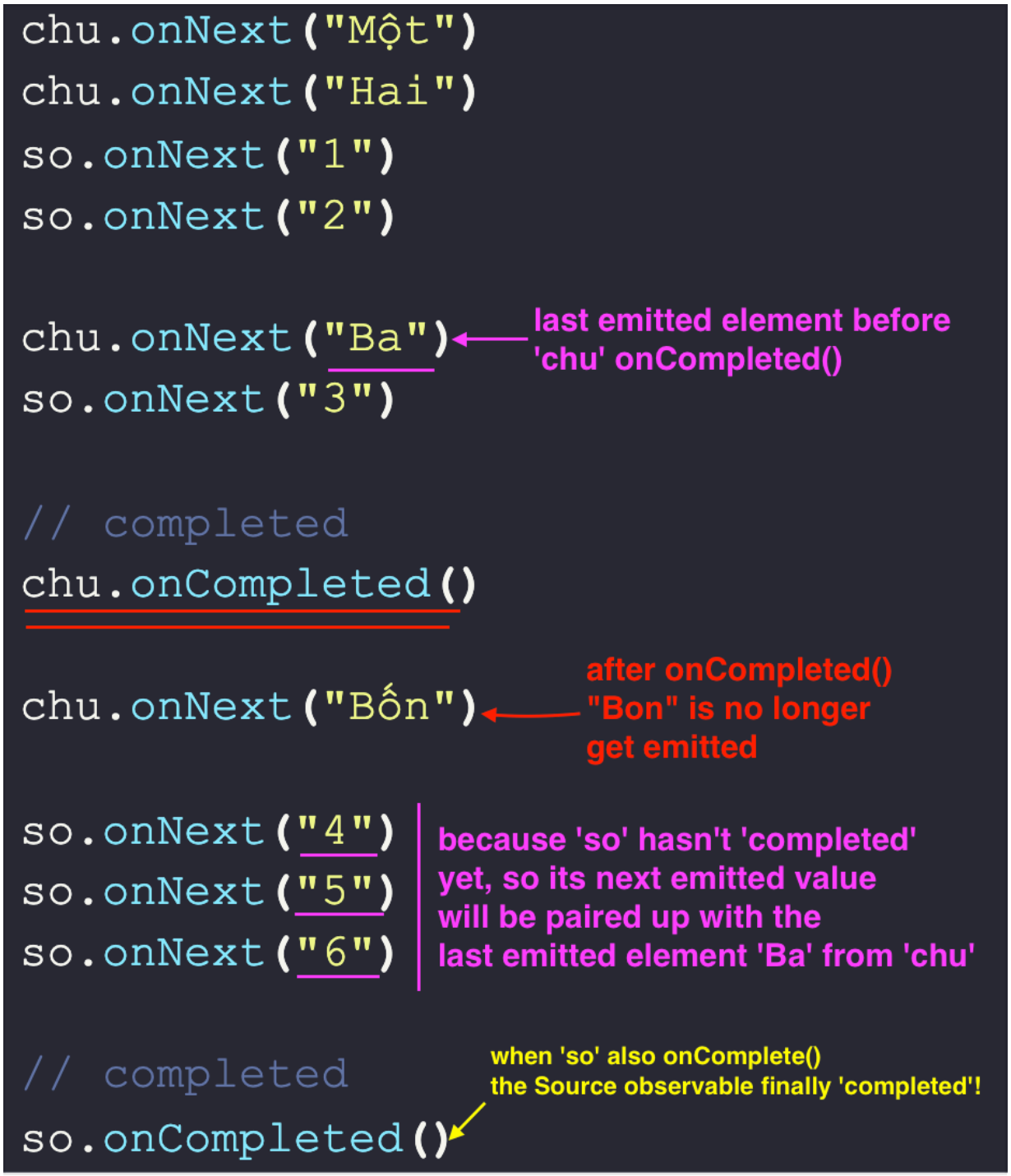
```
let observable = Observable.combineLatest(chu, so)

observable
    .subscribe(onNext: { (value) in
        print(value)
    })
.disposed(by: bag)
```

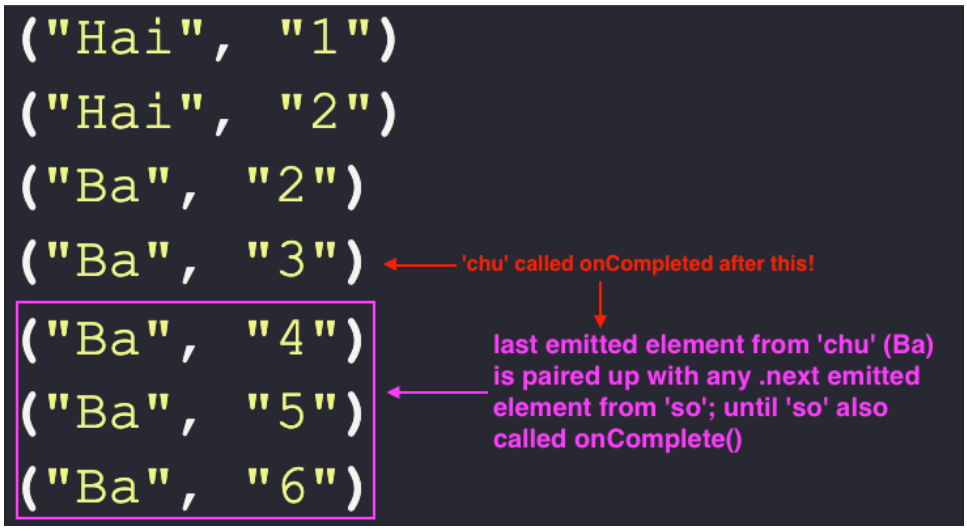




- NOTE: không có ("Một", "1"). Vì lúc đó Observable so chưa phát ra gì cả. Khi so phát ra phần tử đầu tiên thì sẽ kết hợp với phần tử mới nhất của chu, đó là Hai.

(Continue)


chu.onNext("Một")
chu.onNext("Hai")
so.onNext("1")
so.onNext("2")
chu.onNext("Ba")
last emitted element before 'chu' onCompleted()
so.onNext("3")
// completed
chu.onCompleted()
chu.onNext("Bốn")
after onCompleted() "Bon" is no longer get emitted
so.onNext("4")
so.onNext("5")
so.onNext("6")
because 'so' hasn't 'completed' yet, so its next emitted value will be paired up with the last emitted element 'Ba' from 'chu'
// completed
so.onCompleted()
when 'so' also onComplete() the Source observable finally 'completed'!



- **combineLatest (:_:_resultSelector):**
- Just like normal combineLatest
- But, we can use resultSelector to 'format' the output result as **resultSelector** provide us a closure to do so
- Change this:

```
let observable = Observable.combineLatest(chu, so)
```

```
.filter { !$0.0.isEmpty }
```

=> filter guarantee no empty value passed to this:

```
let observable = Observable.combineLatest(chu, so) { chu, so in
    "\(chu) : \(so)"
}
```

AND we will have nicer formatted output like below:

```
Hai : 1
Hai : 2
Ba : 2
Ba : 3
Ba : 4
Ba : 5
Ba : 6
```

- Usage example: automatic updates of on-screen values when the user settings change

```
-------------------------------------------------------------------------
-----------------------------------
    let choice: Observable<DateFormatter.Style> =
Observable.of(.short, .long)

    let dates = Observable.of(Date())

    let observable =
Observable.combineLatest(choice, dates) { format,
when -> String in
        let formatter = DateFormatter()
        formatter.dateStyle = format
        return formatter.string(from: when)
    }

    _ = observable.subscribe(onNext: { value in
        print(value)
    })
-------------------------------------------------------------------------
-------------------------------
```

Với 1 giá trị Date ta có thể lựa chọn **kiểu format dữ liệu** để in ra một cách tiện lợi nhất. Không cần gọi hàm đi gọi hàm lại hay là for các kiểu nữa.

**Good to know:

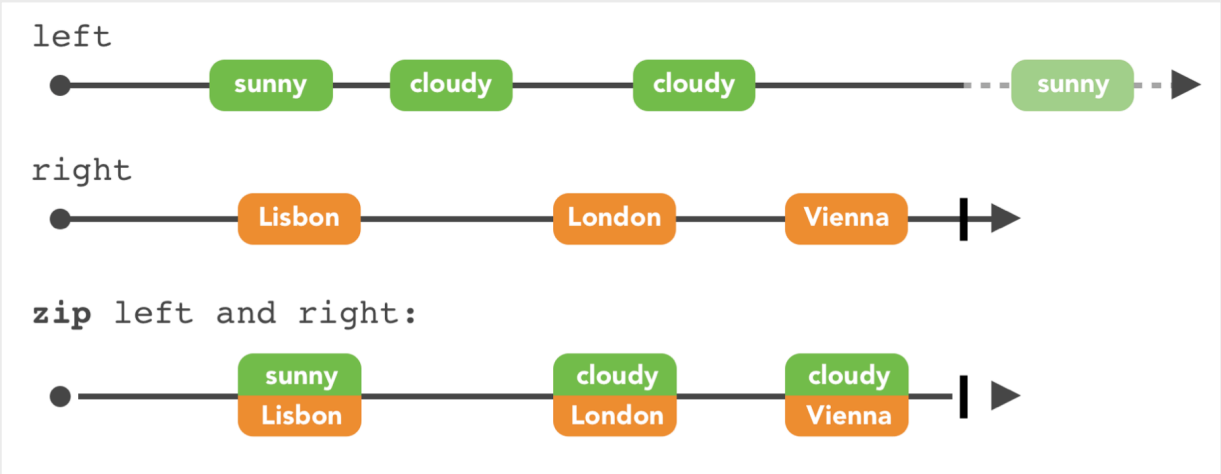
- **combineLatest** family **can take a collection of observables** and a combining closure, which **receives latest values in an array.**
  - Since it's a collection, all observables carry elements of the same type.
  - It's less flexible than the multiple parameter variants, it is seldom-used but still handy to know about
  - Example above could be rewritten as:

```
// 1
let observable = Observable.combineLatest([left, right]) {
  strings in strings.joined(separator: " ")
}
```

- **zip**

- Similar to combineLatest, zip will:
  - Subscribed to the observables you provided.
  - Waited for each to emit a new value.
  - Called your closure with both new values.

- They pair each *next* value of each observable at the same logical position (1st with 1st, 2nd with 2nd, etc.).
  - if **no *next* value from one of the inner observables** is available at the next logical position (i.e. because it completed, like in the example above), **zip won't emit anything anymore**.
  - This is called **indexed sequencing**, which is a way to walk sequences in lockstep.
- In the end, zip won't itself complete until all its inner observables complete, making sure each can complete its work.




```
  enum Weather {
    case cloudy
    case sunny
  }
  let left: Observable<Weather> = Observable.of(.sunny, .cloudy, .cloudy,
.sunny)
  let right = Observable.of("Lisbon", "Copenhagen", "London", "Madrid",
"Vienna")
```

```
  let observable = Observable.zip(left, right) { weather, city in
    return "It's \(weather) in \(city)"
  }
  _ = observable.subscribe(onNext: { value in
    print(value)
  })
}
```

```
—— Example of: zip ——
It's sunny in Lisbon
It's cloudy in Copenhagen
It's cloudy in London
It's sunny in Madrid
```

** Good to know:

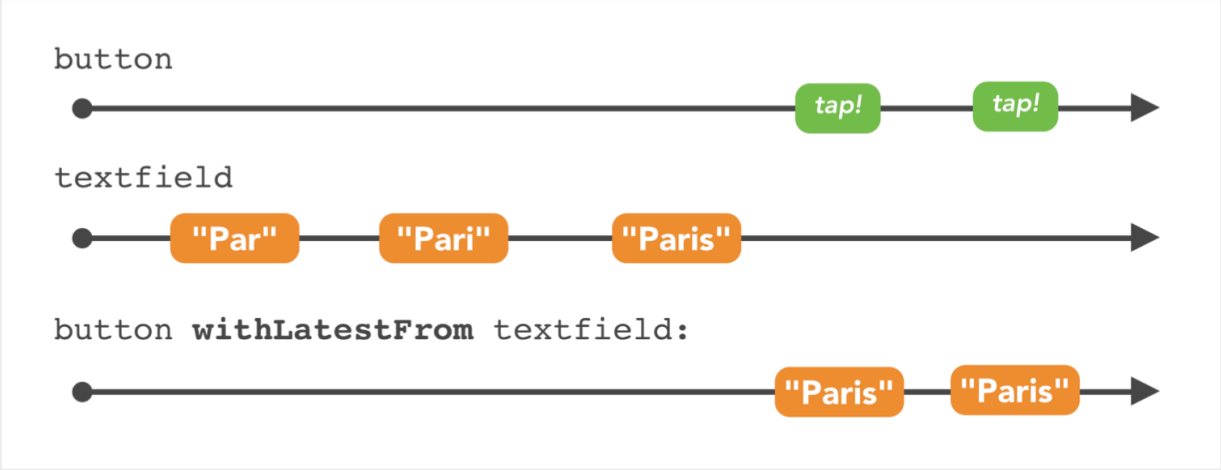
- Swift also has a zip(_:_:) collection operator.
  - It creates a new collection of tuples with items from both collections.
- RxSwift offers variants for two to eight observables, plus a variant for collections, like **combineLatest** does.

✅ Triggers

You'll often need to accept input from several observables at once. Some will simply trigger actions in your code, while others will provide data.

- **withLatestFrom**

Trigger sẽ được sử dụng, khi bạn muốn tạo ra **1 điều kiện** nào đó từ Observable để một Observable khác được phép hoạt động.




```
// 1
let button = PublishSubject<Void>()
let textField = PublishSubject<String>()

// 2
let observable = button.withLatestFrom(textField)
_ = observable.subscribe(onNext: { value in
  print(value)
})

// 3
textField.onNext("Par")
textField.onNext("Pari")
textField.onNext("Paris")
button.onNext(())
button.onNext(())
```

since button carry no real data
we declare it as Void type

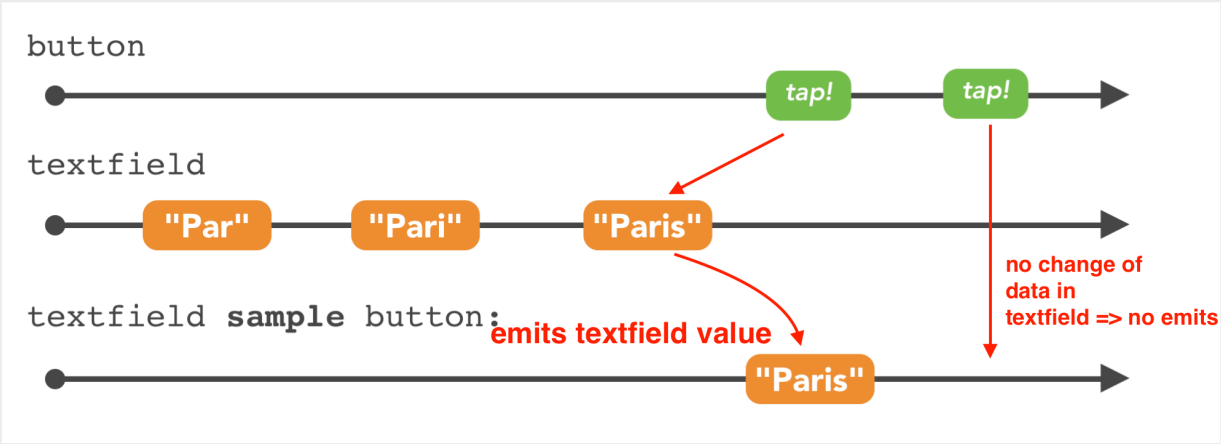
when button emitted 'value',
it emited the latest value from textField instead

```
Paris
Paris
```

- **Sample**

– Nearly the same thing withLatestFrom, excepts:

    - **sample(_:)** emits the latest value from the “other” observable, but only if the data is different from previous one.
        - If no new data arrived, sample(_:) won’t emit anything.
- You could have achieved the same behavior by adding a distinctUntilChanged() to the **withLatestFrom(_:)** observable, but **smallest possible operator chains are the Zen of Rx**.



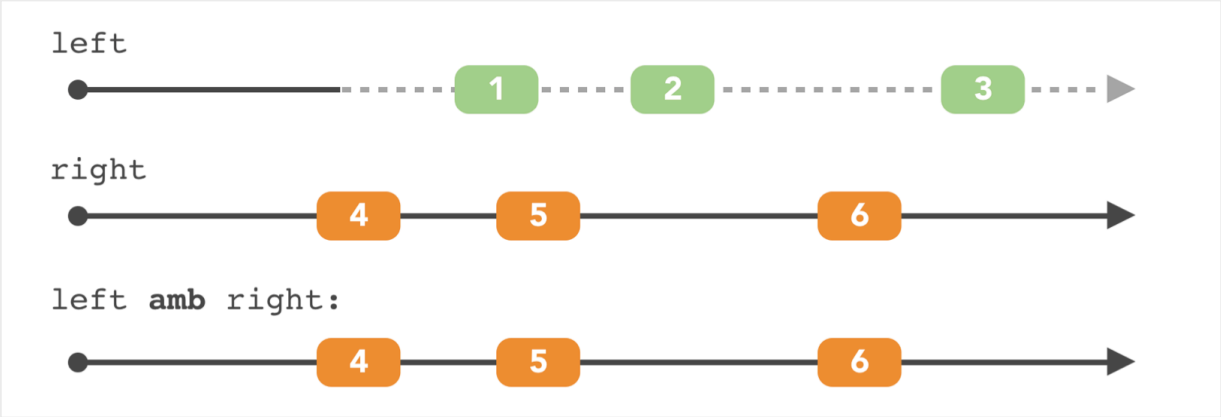

```
let observable9 = button.sample(textField)

observable9
    .subscribe{
    print("observable 8: ",$0)
    }
    .disposed(by: disposeBag)

textField.onNext("H")
textField.onNext("Hel")
textField.onNext("Hello")

button.onNext(()) //when button emit .next event, it will emit the last emitted value of textField
button.onNext(()) //this time, button won't emits everything because no new data from textField, still "Hello"
```

Switches

- There're two main so-called “switching” operators: amb(_:) and switchLatest()
    - both allow you to produce an observable sequence by switching between the events of the combined or source sequences.
        - This allows you to decide which sequence’s events will the subscriber receive at runtime.

- **amb**:
- Think of “**amb**” as in “**ambiguous**”.

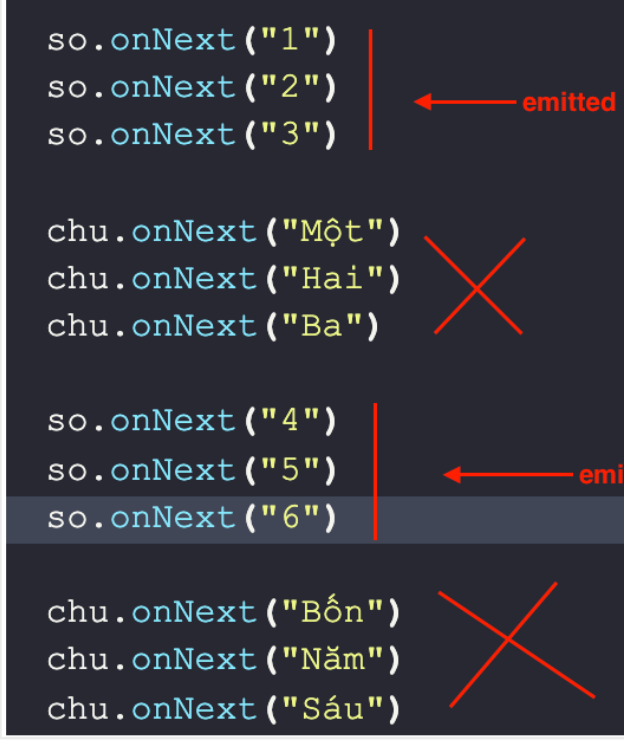
- Nó sẽ tạo ra một Observable để giải quyết vấn đề quyết định nhận dữ liệu từ nguồn nào
  - Trong khi cả 2 nguồn đều có thể phát dữ liệu. Thì nguồn nào phát trước, thì nó sẽ nhận dữ liệu từ nguồn đó.
  - **Nguồn phát sau sẽ bị âm thầm ngắt kết nối**



- The amb(_:) operator **subscribes to the left and right observables**.
- It **waits** for any of them to emit an element, then **unsubscribes** from the *other* one. After that, it only relays elements from the first active observable.

EX:

```swift
let bag = DisposeBag()

let chu = PublishSubject<String>()
let so = PublishSubject<String>()

let observable = chu.amb(so)

observable
    .subscribe(onNext: { (value) in
        print(value)
    })
.disposed(by: bag)

so.onNext("1")
so.onNext("2")
so.onNext("3")

chu.onNext("Một")
chu.onNext("Hai")
chu.onNext("Ba")

so.onNext("4")
so.onNext("5")
so.onNext("6")

chu.onNext("Bốn")
chu.onNext("Năm")
chu.onNext("Sáu")
```

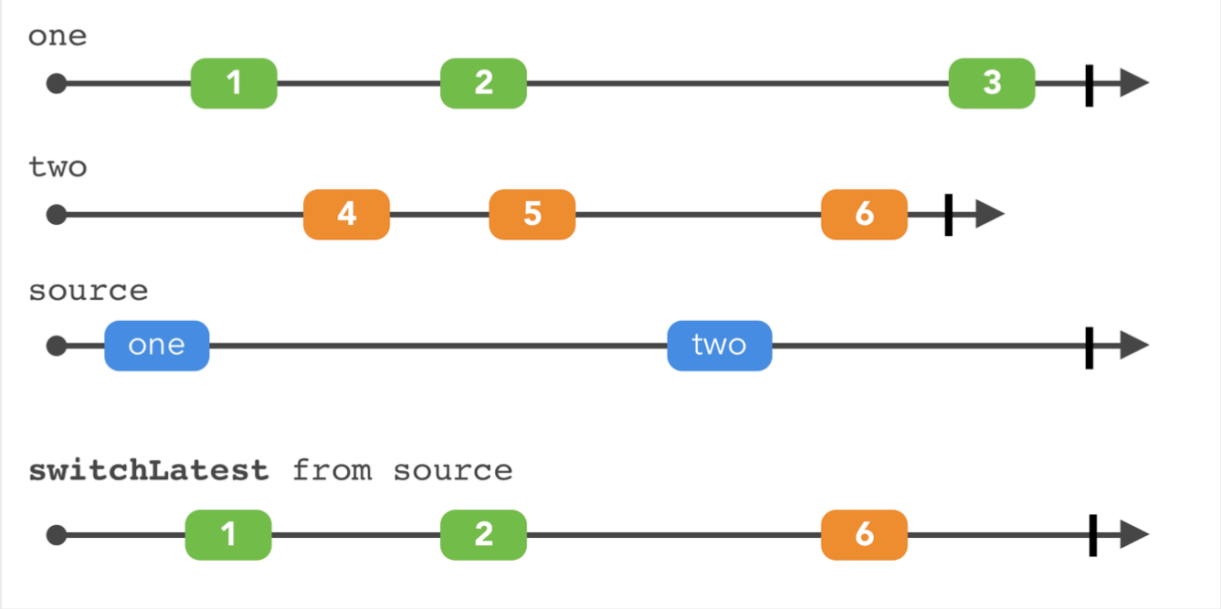



```
1
2
3
4
5
6
```

=> Vì "**so**" đã phát trước, nên các dữ liệu từ "**chu**" sẽ không nhận được.

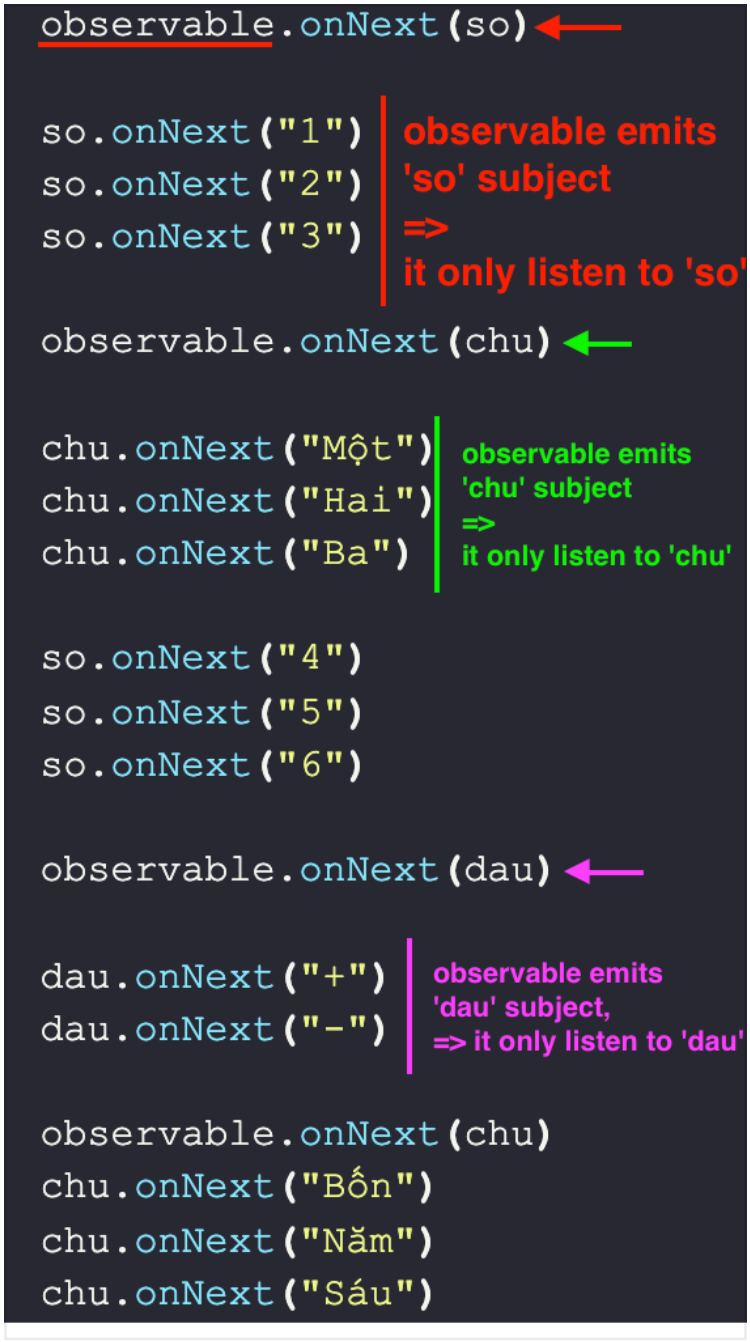
- Nếu ta cho thêm "**chu**" phát onNext trước số thì sẽ thấy dữ liệu nhận được sẽ toàn là từ "**chu**".

- **switchLatest**
- More popular than amb()
- Similar to flatMapLatest




```
let chu = PublishSubject<String>()
let so = PublishSubject<String>()
let dau = PublishSubject<String>()

let observable = PublishSubject<Observable<String>>()
```

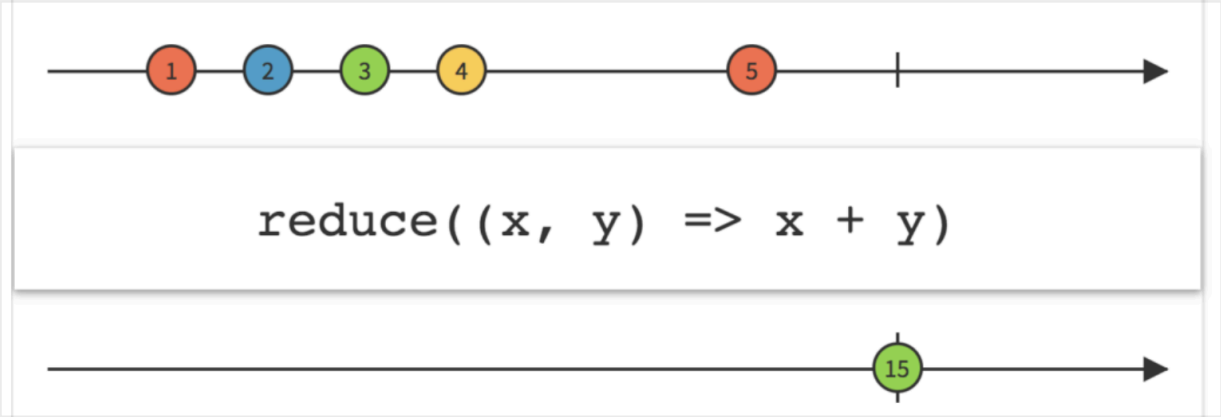
```
observable
    .switchLatest()
    .subscribe(onNext: { (value) in
        print(value)
    }, onCompleted: {
        print("completed")
    })
.disposed(by: bag)
```



=> **observable** sẽ phát đi **subject** nào thì **subscriber** trên sẽ nhận được giá trị phát của **subject** đó.

✅ Combining elements within a sequence

- **reduce**



- It starts with the initial value you provide (in the example below, you start with 0).
- Each time the source observable emits an item, reduce(_:_:) calls your closure to produce a new summary.
- **When the source observable completes**, reduce(_:_:) **emits the summary value, then completes.**

EX:

```
let source = Observable.of(1, 2, 3, 4, 5, 6, 7, 8, 9)

let observable = source.reduce(0, accumulator: +)

_ = observable                    initial value    operator in use
    .subscribe(onNext: { value in
        print(value)
    })
```

There are also different way to write reduce() such as:

```
let observable = source.reduce(0) { $0 + $1 }
```

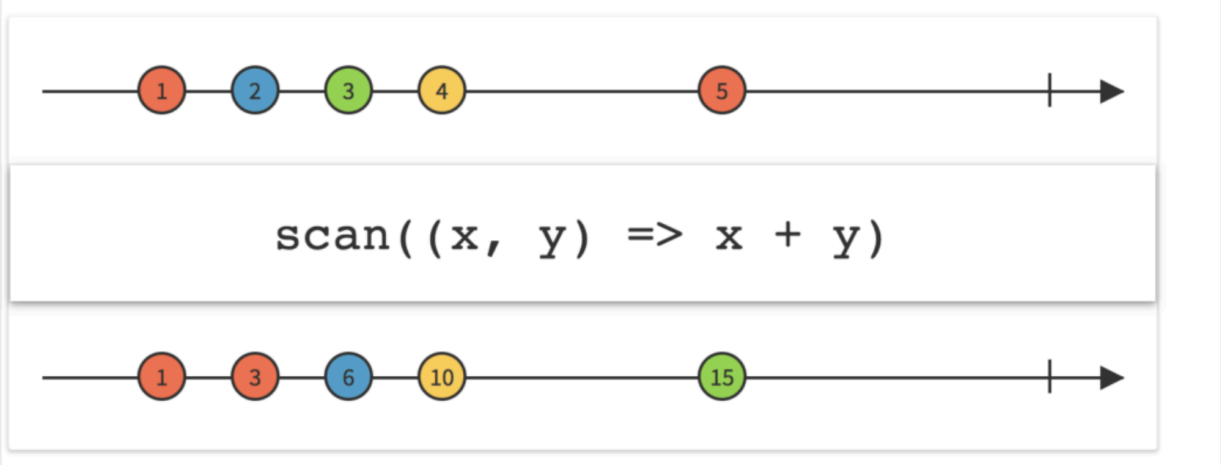
**NOTE:

- $0 : summary , giá trị tích luỹ đến điểm hiện tại
- khi nhận được 1 giá trị, thì giá trị đó là $1
- After closure, result again returns as new updated value for $0. Kết quả trả về sẽ gán lại cho $0

OR:

```
let observable = source.reduce(0) { summary, newValue in
  return summary + newValue
}
```

- **scan(_:accumulator:)**
  - Each time the source observable emits an element, scan(_:accumulator:) invokes your closure.
    - It passes the running value along with the new element
    - the closure returns the new accumulated value.
  - Like reduce(_:_:), the resulting observable type is the closure return type.




```
let source = Observable.of(1, 3, 5, 7, 9)

let observable = source.scan(0, accumulator: +)
_ = observable.subscribe(onNext: { value in
  print(value)
})
```

```
1
4
9
16
25
```

- Thay vì chờ Observable kết thúc và đưa ra kết quả cuối cùng.

Thì scan nó sẽ tính toán và phát đi **từng kết quả** tinh toán được, sau khi có dữ liệu từ Observable phát ra. Không quan tâm Observable kết thúc mới thực hiện.