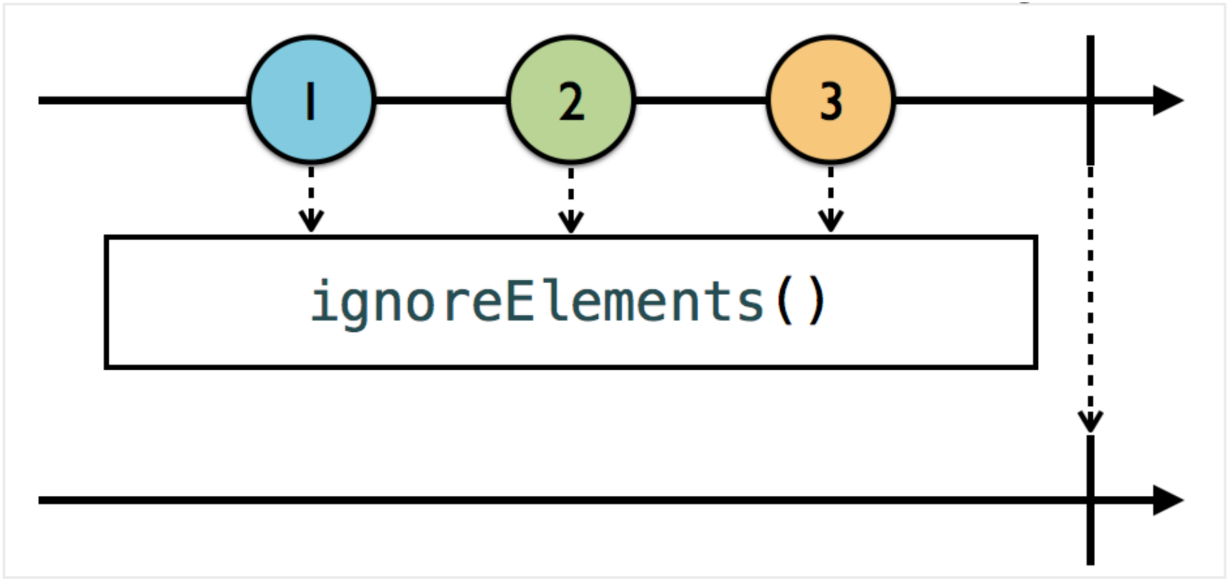
RXSwift - Filtering Operators

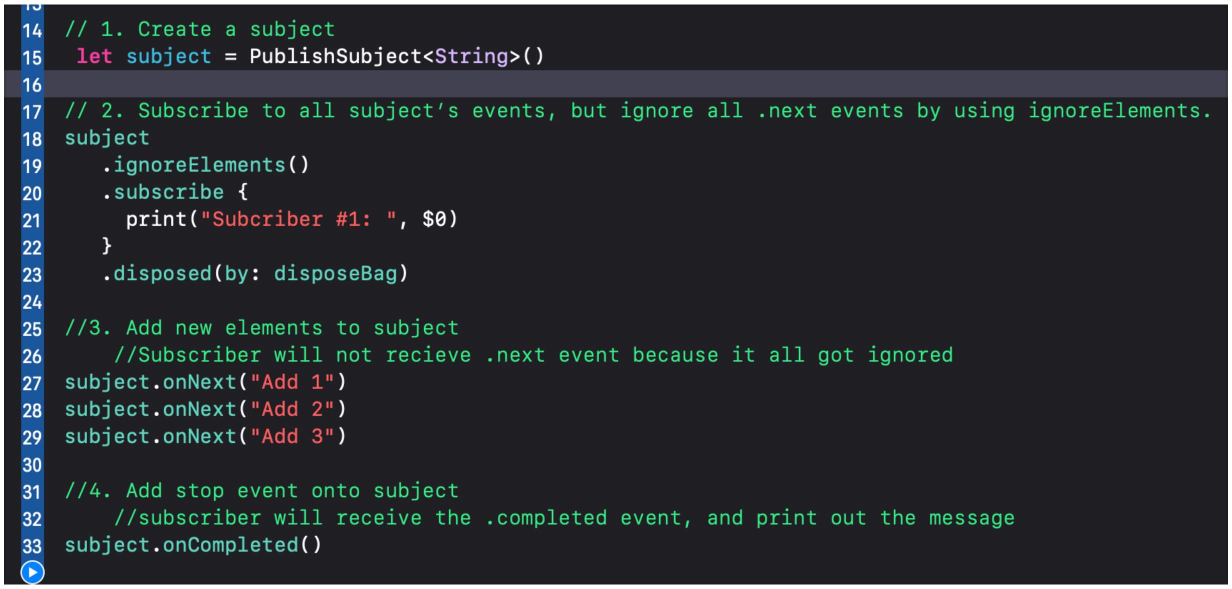
RxSwift's filtering operators that you can use to apply conditional constraints to .next events, so that the subscriber only receives the elements it wants to deal with.

✓ Ignoring operators

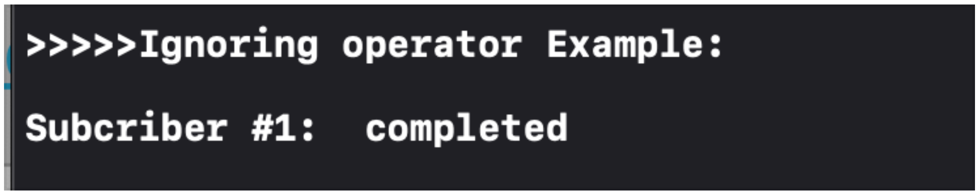
- **ignoreElement:**

– **ignore .next event** elements.
– However, it will **allow stop events** through, such as .completed or .error events.





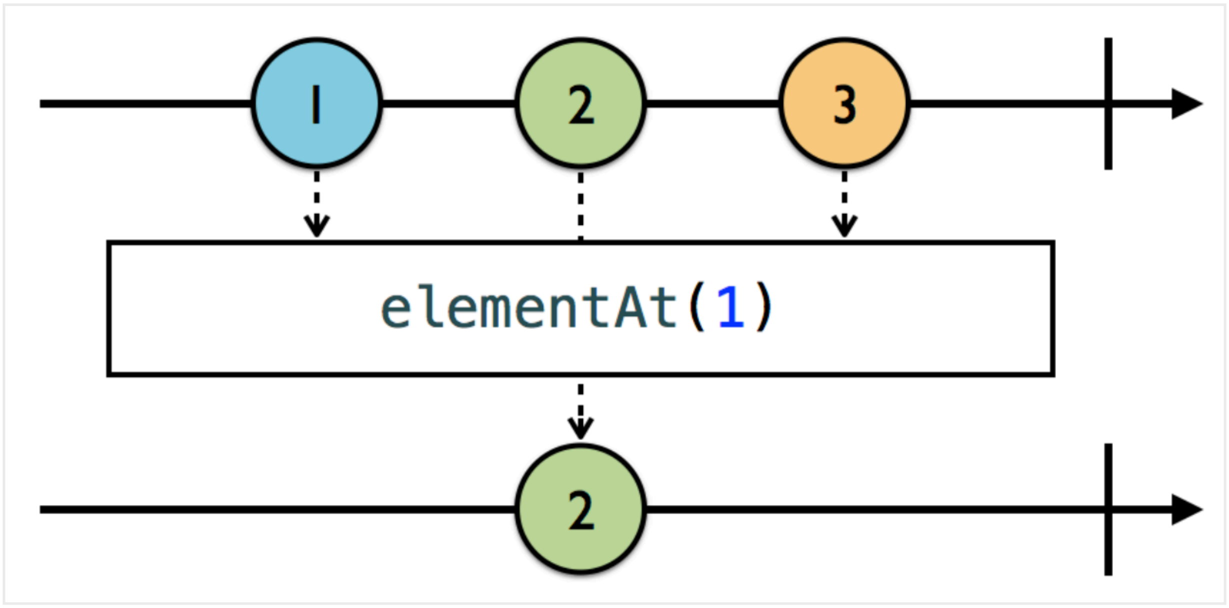
```
// 1. Create a subject
 let subject = PublishSubject<String>()

// 2. Subscribe to all subject's events, but ignore all .next events by using ignoreElements.
subject
   .ignoreElements()
   .subscribe {
     print("Subcriber #1: ", $0)
   }
   .disposed(by: disposeBag)

//3. Add new elements to subject
    //Subscriber will not recieve .next event because it all got ignored
subject.onNext("Add 1")
subject.onNext("Add 2")
subject.onNext("Add 3")

//4. Add stop event onto subject
    //subscriber will receive the .completed event, and print out the message
subject.onCompleted()
```

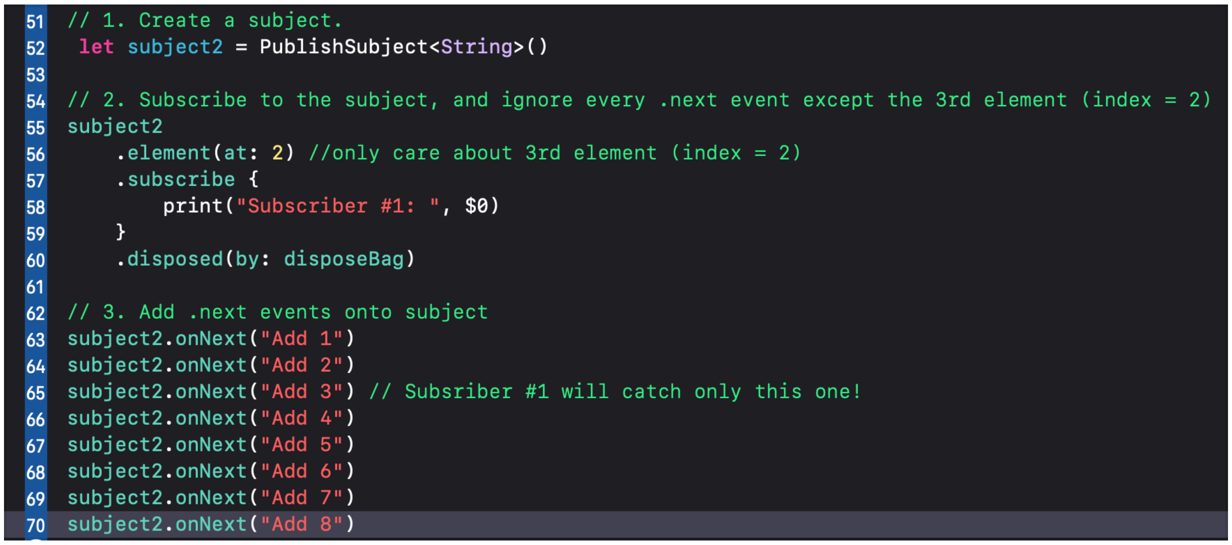


```
>>>>>Ignoring operator Example:

Subcriber #1:  completed
```

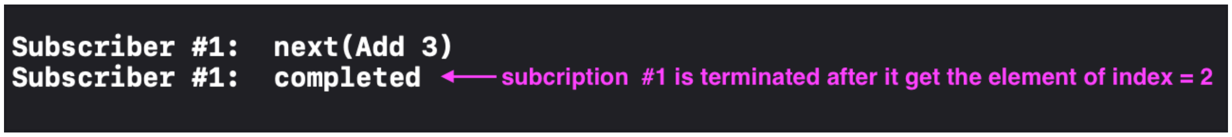
- **elementAt:**

– Takes the index of the element you want to receive, and it ignores everything else





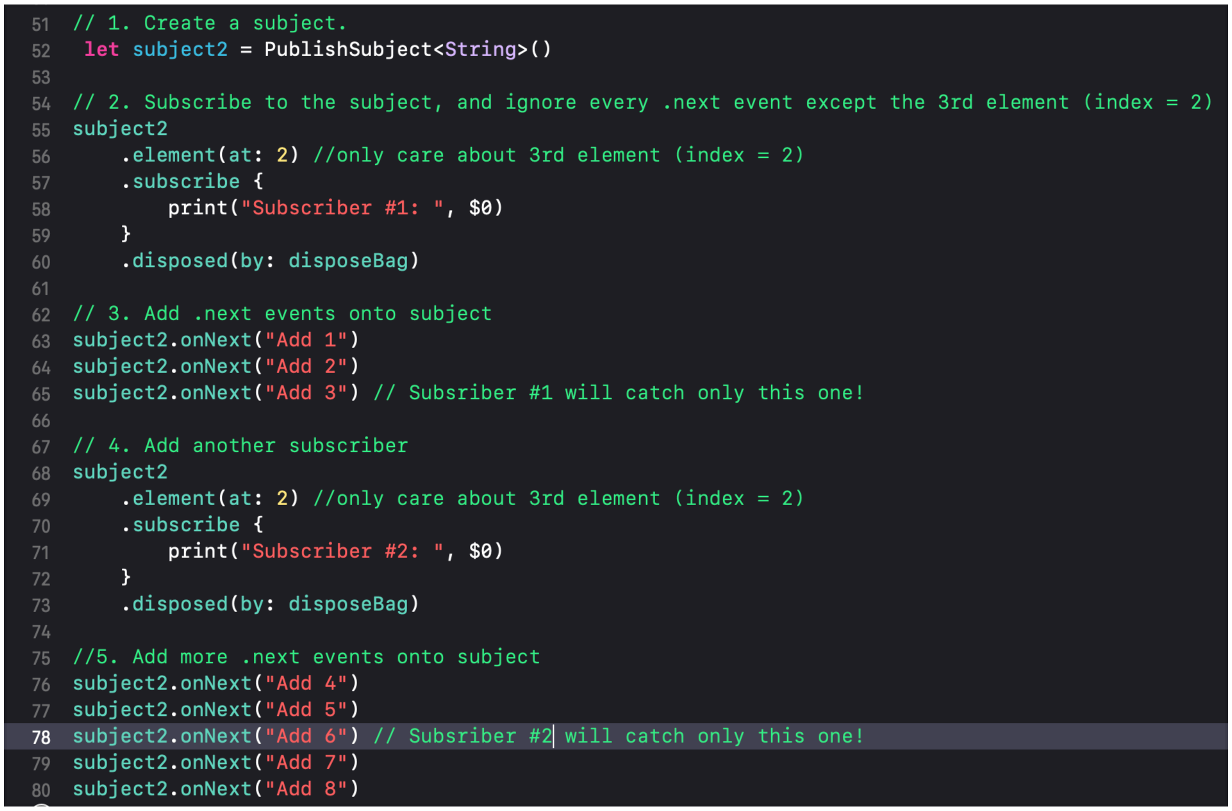
```swift
// 1. Create a subject.
 let subject2 = PublishSubject<String>()

// 2. Subscribe to the subject, and ignore every .next event except the 3rd element (index = 2)
subject2
    .element(at: 2) //only care about 3rd element (index = 2)
    .subscribe {
        print("Subscriber #1: ", $0)
    }
    .disposed(by: disposeBag)

// 3. Add .next events onto subject
subject2.onNext("Add 1")
subject2.onNext("Add 2")
subject2.onNext("Add 3") // Subsriber #1 will catch only this one!
subject2.onNext("Add 4")
subject2.onNext("Add 5")
subject2.onNext("Add 6")
subject2.onNext("Add 7")
subject2.onNext("Add 8")
```



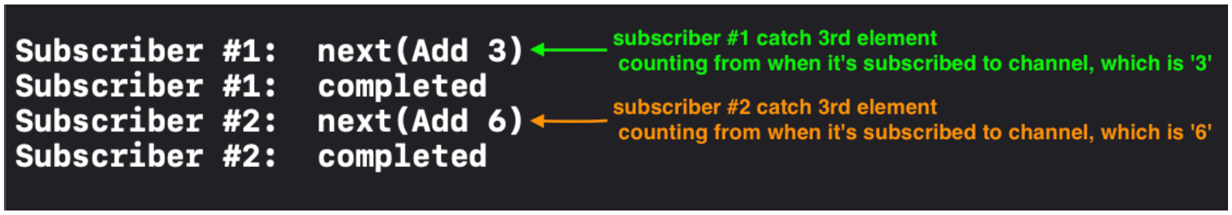
```
Subscriber #1:  next(Add 3)
Subscriber #1:  completed  ←— subcription #1 is terminated after it get the element of index = 2
```

NOTE: As soon as an element is emitted at the provided index, the subscription will be terminated.

Assume we add another subscriptions after subject emit 3rd element:



```swift
// 1. Create a subject.
 let subject2 = PublishSubject<String>()

// 2. Subscribe to the subject, and ignore every .next event except the 3rd element (index = 2)
subject2
    .element(at: 2) //only care about 3rd element (index = 2)
    .subscribe {
        print("Subscriber #1: ", $0)
    }
    .disposed(by: disposeBag)

// 3. Add .next events onto subject
subject2.onNext("Add 1")
subject2.onNext("Add 2")
subject2.onNext("Add 3") // Subsriber #1 will catch only this one!

// 4. Add another subscriber
subject2
    .element(at: 2) //only care about 3rd element (index = 2)
    .subscribe {
        print("Subscriber #2: ", $0)
    }
    .disposed(by: disposeBag)

//5. Add more .next events onto subject
subject2.onNext("Add 4")
subject2.onNext("Add 5")
subject2.onNext("Add 6") // Subsriber #2 will catch only this one!
subject2.onNext("Add 7")
subject2.onNext("Add 8")
```



```
Subscriber #1:  next(Add 3)
Subscriber #1:  completed
Subscriber #2:  next(Add 6)
Subscriber #2:  completed
```

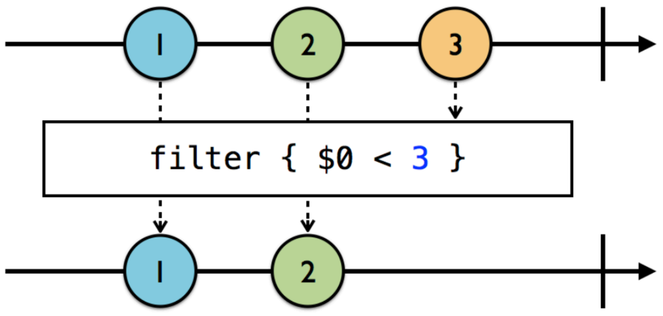
subscriber #1 catch 3rd element
counting from when it's subscribed to channel, which is '3'
subscriber #2 catch 3rd element
counting from when it's subscribed to channel, which is '6'

So, subscription will ignore every event and only receive 2nd element counting from when it started to subscribe the subject. Then, subscription will terminate after it.

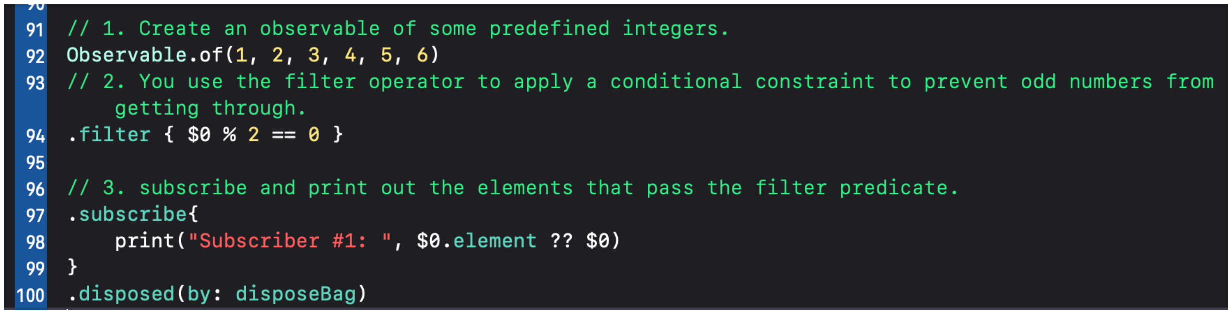
- **filter:**

– It takes a predicate closure, which it applies to every element emitted, allowing through only those elements for which the predicate resolves to **true**.
  - filter takes a predicate that returns a Bool. Return true to let the element through or false to prevent it.
  - filter will filter elements for the life of the subscription.

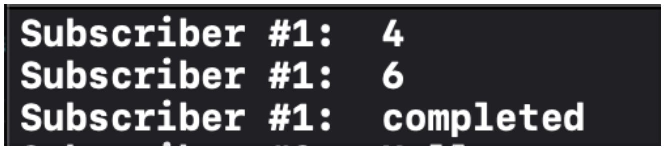
Check out this marble diagram, where only 1 and 2 are let through, because the filter's predicate only allows elements that are less than 3.



EX1: filter only odd element

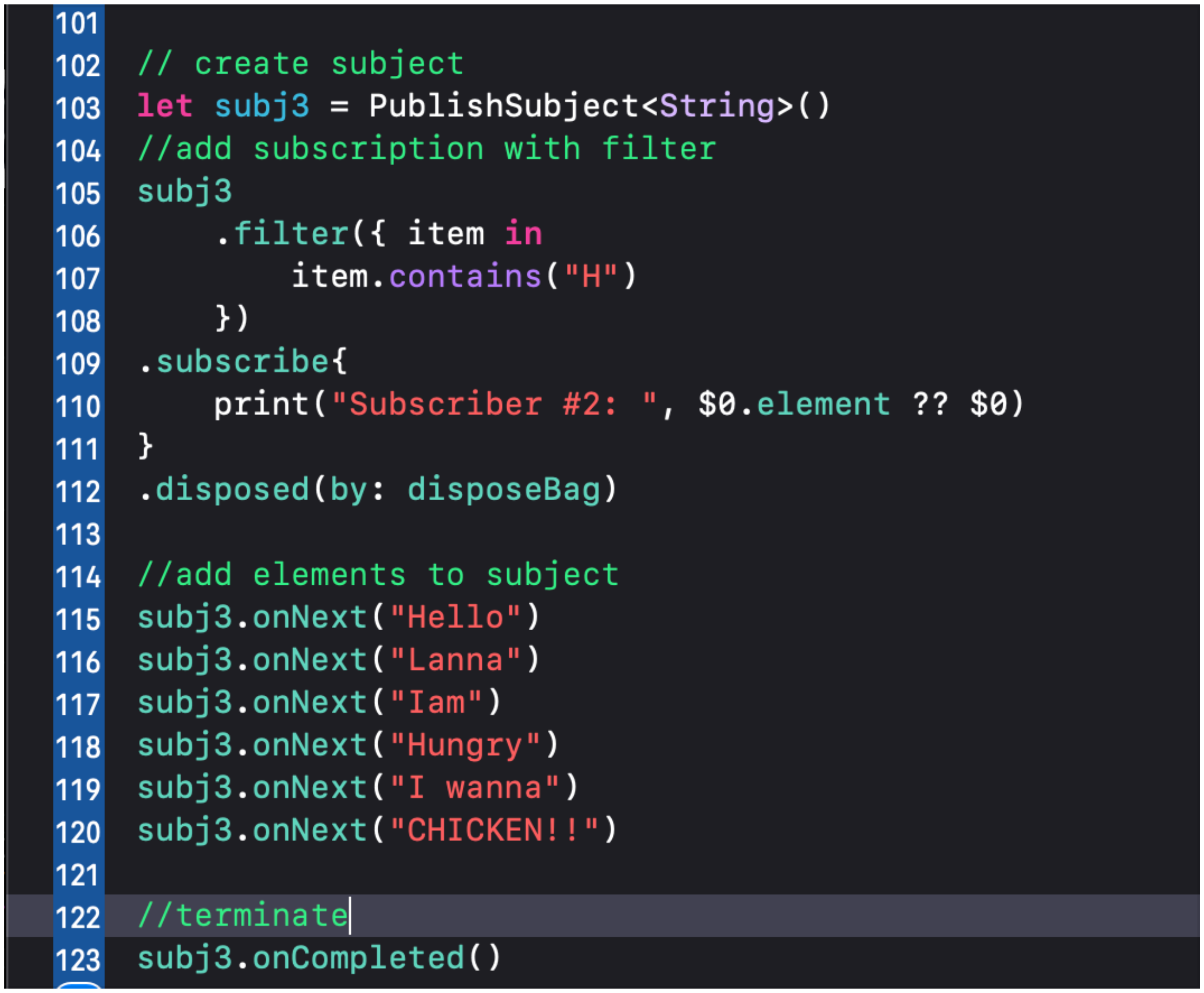


```
// 1. Create an observable of some predefined integers.
Observable.of(1, 2, 3, 4, 5, 6)
// 2. You use the filter operator to apply a conditional constraint to prevent odd numbers from
    getting through.
.filter { $0 % 2 == 0 }

// 3. subscribe and print out the elements that pass the filter predicate.
.subscribe{
    print("Subscriber #1: ", $0.element ?? $0)
}
.disposed(by: disposeBag)
```

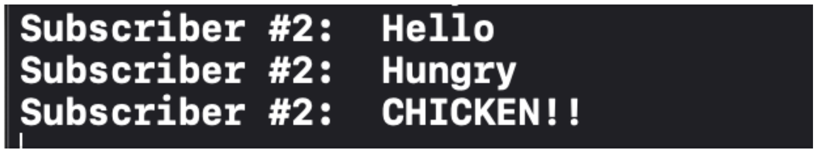


```
Subscriber #1:  4
Subscriber #1:  6
Subscriber #1:  completed
```

EX2: filter only element that contains "H"



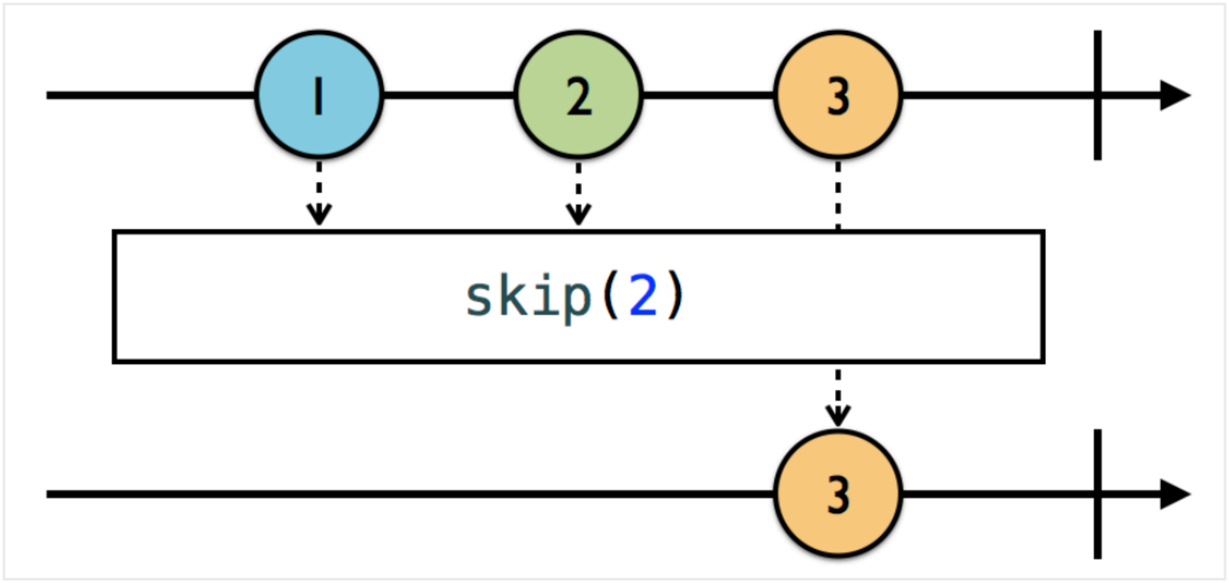
```swift

// create subject
let subj3 = PublishSubject<String>()
//add subscription with filter
subj3
    .filter({ item in
        item.contains("H")
    })
.subscribe{
    print("Subscriber #2: ", $0.element ?? $0)
}
.disposed(by: disposeBag)

//add elements to subject
subj3.onNext("Hello")
subj3.onNext("Lanna")
subj3.onNext("Iam")
subj3.onNext("Hungry")
subj3.onNext("I wanna")
subj3.onNext("CHICKEN!!")

//terminate
subj3.onCompleted()
```



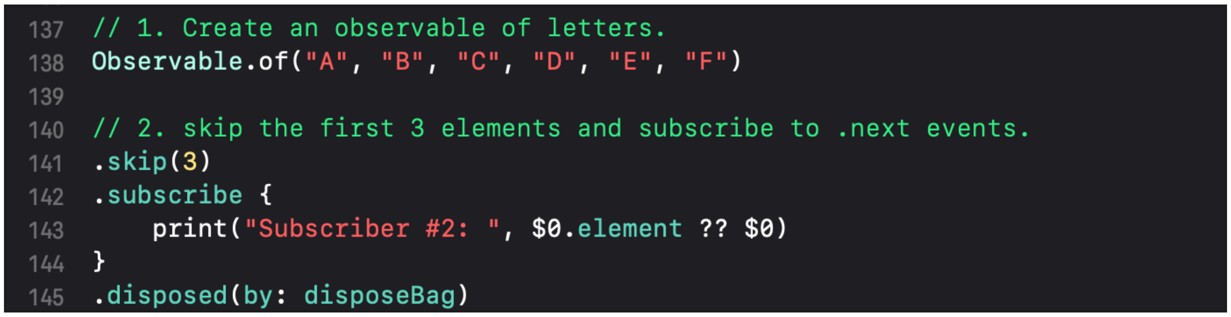
```
Subscriber #2:  Hello
Subscriber #2:  Hungry
Subscriber #2:  CHICKEN!!
```

✓ Skipping operators

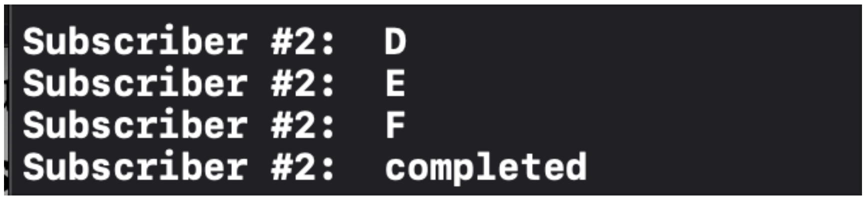
- **skip:**
- skip operator allows you to ignore from the 1st to the number you pass as its parameter.



EX: using skip with observable

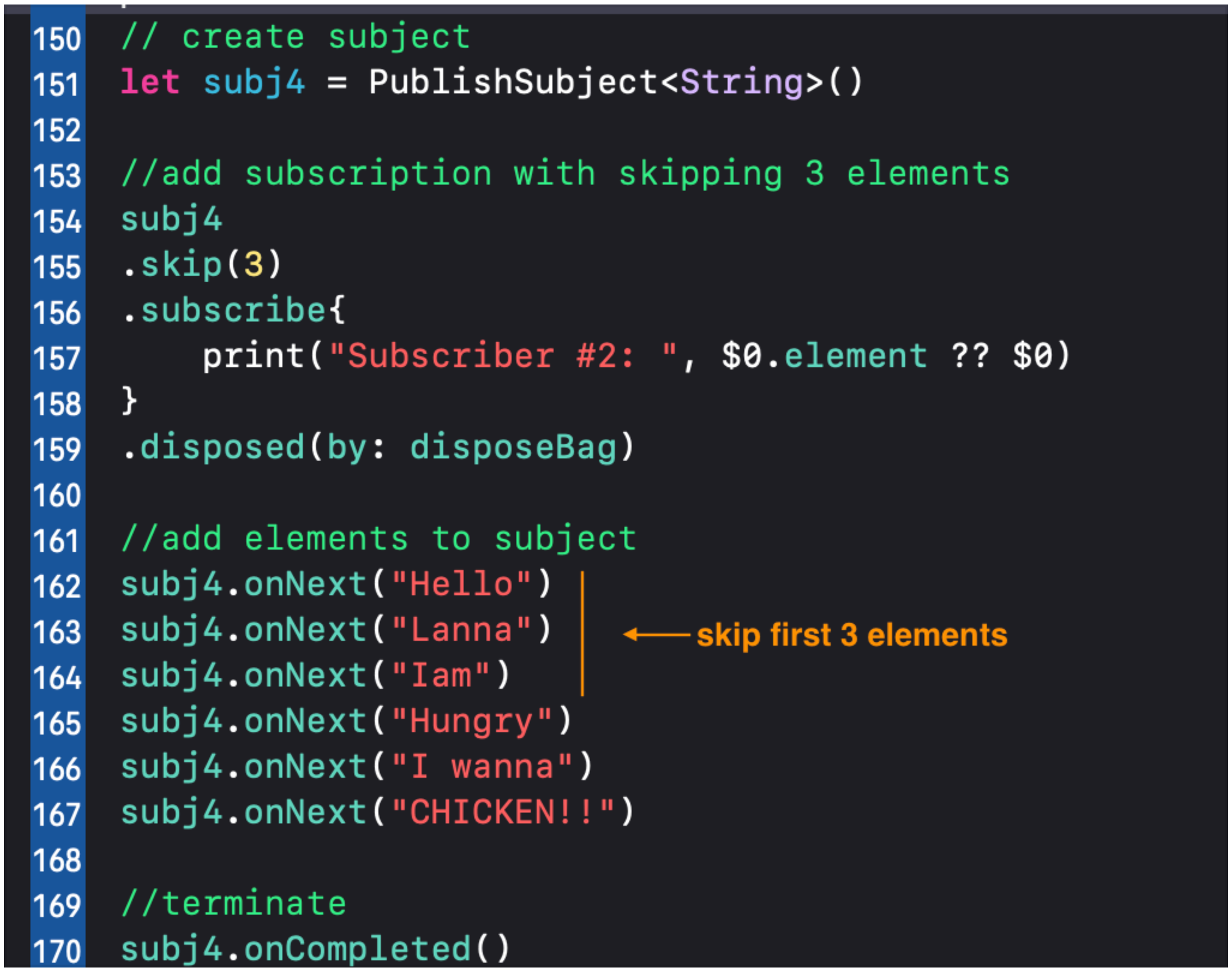


```
// 1. Create an observable of letters.
Observable.of("A", "B", "C", "D", "E", "F")

// 2. skip the first 3 elements and subscribe to .next events.
.skip(3)
.subscribe {
    print("Subscriber #2: ", $0.element ?? $0)
}
.disposed(by: disposeBag)
```

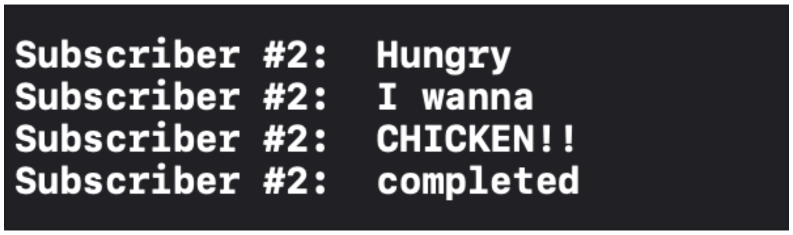


```
Subscriber #2:  D
Subscriber #2:  E
Subscriber #2:  F
Subscriber #2:  completed
```

EX: using skip with subject



```swift
// create subject
let subj4 = PublishSubject<String>()

//add subscription with skipping 3 elements
subj4
.skip(3)
.subscribe{
    print("Subscriber #2: ", $0.element ?? $0)
}
.disposed(by: disposeBag)

//add elements to subject
subj4.onNext("Hello")
subj4.onNext("Lanna")
subj4.onNext("Iam")
subj4.onNext("Hungry")
subj4.onNext("I wanna")
subj4.onNext("CHICKEN!!")

//terminate
subj4.onCompleted()
```



```
Subscriber #2:  Hungry
Subscriber #2:  I wanna
Subscriber #2:  CHICKEN!!
Subscriber #2:  completed
```

- **skip(while:...):**

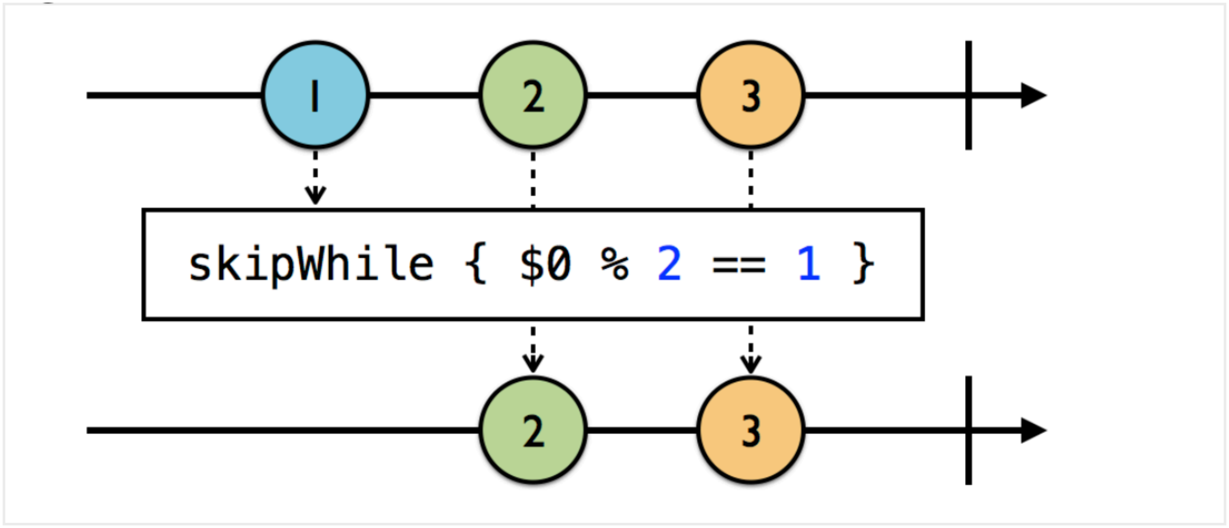
– Like filter, skipWhile lets you include a predicate to determine what should be skipped.

– **Unlike filter, , skipWhile will only skip up until something is *not* skipped, and then it will let everything else through from that point on.

  - filter will filter elements for the life of the subscription

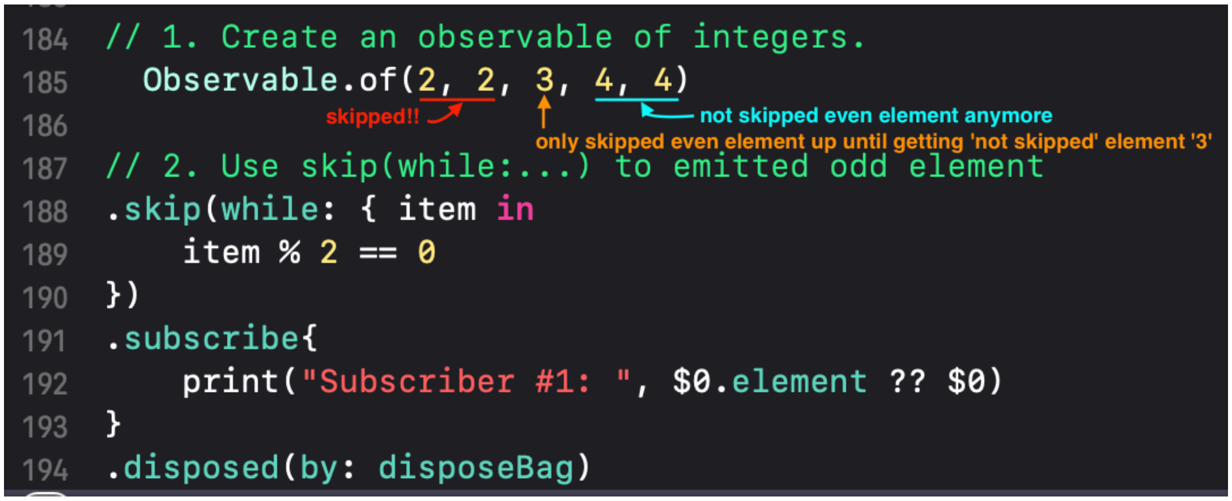
– Returning true will cause the element to be skipped, and returning *false* will let it through.

Compare	filter	skipWhile
Similarity	have a predicate to determine what should be filtered out	have a predicate to determine what should be skipped
Differences	filter will filter elements for the life of the subscription	**only skip up until something is *not* skipped, and then it will let everything else through from that point on.
	Return true to let the element through or false to prevent it.	Returning true will cause the element to be skipped, and returning *false* will let it through.



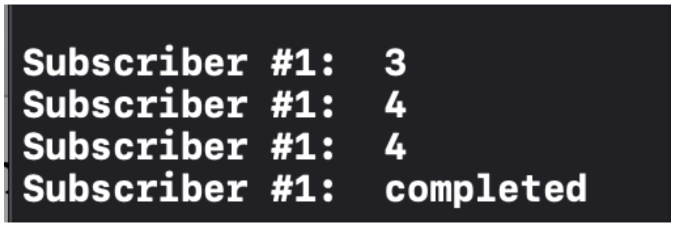
- 1 is prevented because 1 % 2 equals 1,
- 2 is allowed through because it fails the predicate,
- 3 (and everything else going forward) gets through **because skipWhile is no longer skipping.**

EX: Using skipWhile() with observable



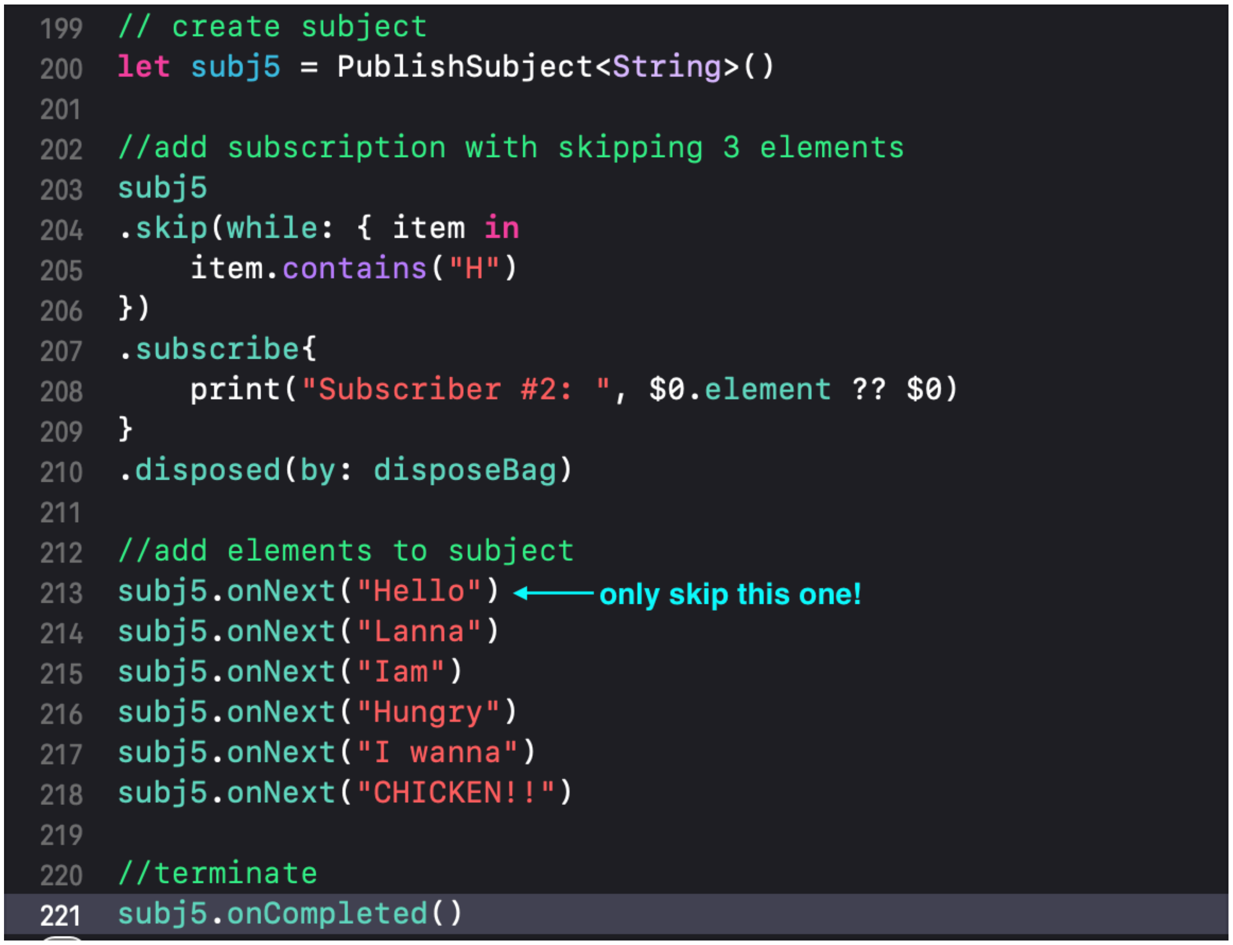
```
// 1. Create an observable of integers.
  Observable.of(2, 2, 3, 4, 4)

// 2. Use skip(while:...) to emitted odd element
.skip(while: { item in
    item % 2 == 0
})
.subscribe{
    print("Subscriber #1: ", $0.element ?? $0)
}
.disposed(by: disposeBag)
```

skipped!!
not skipped even element anymore
only skipped even element up until getting 'not skipped' element '3'

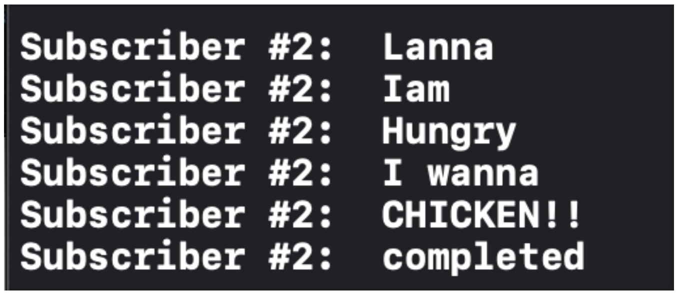


```
Subscriber #1:  3
Subscriber #1:  4
Subscriber #1:  4
Subscriber #1:  completed
```

EX: Using subject



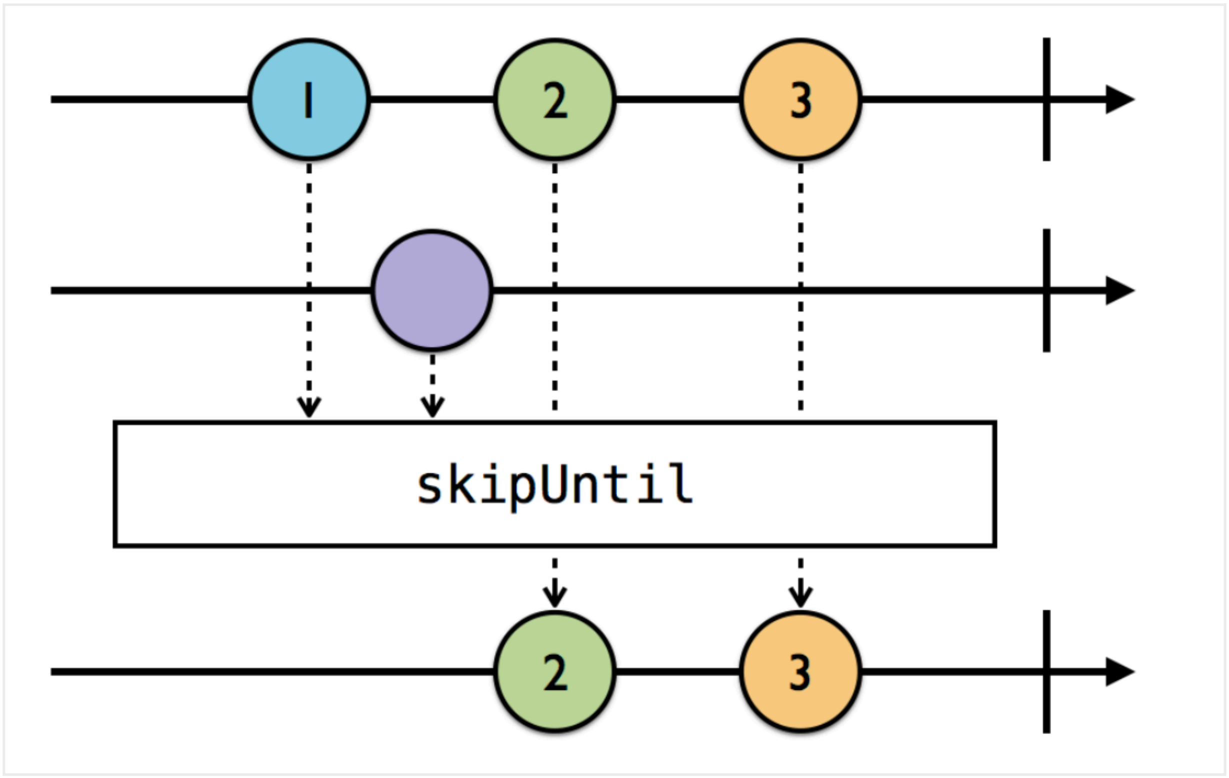
```swift
// create subject
let subj5 = PublishSubject<String>()

//add subscription with skipping 3 elements
subj5
.skip(while: { item in
    item.contains("H")
})
.subscribe{
    print("Subscriber #2: ", $0.element ?? $0)
}
.disposed(by: disposeBag)

//add elements to subject
subj5.onNext("Hello")
subj5.onNext("Lanna")
subj5.onNext("Iam")
subj5.onNext("Hungry")
subj5.onNext("I wanna")
subj5.onNext("CHICKEN!!")

//terminate
subj5.onCompleted()
```



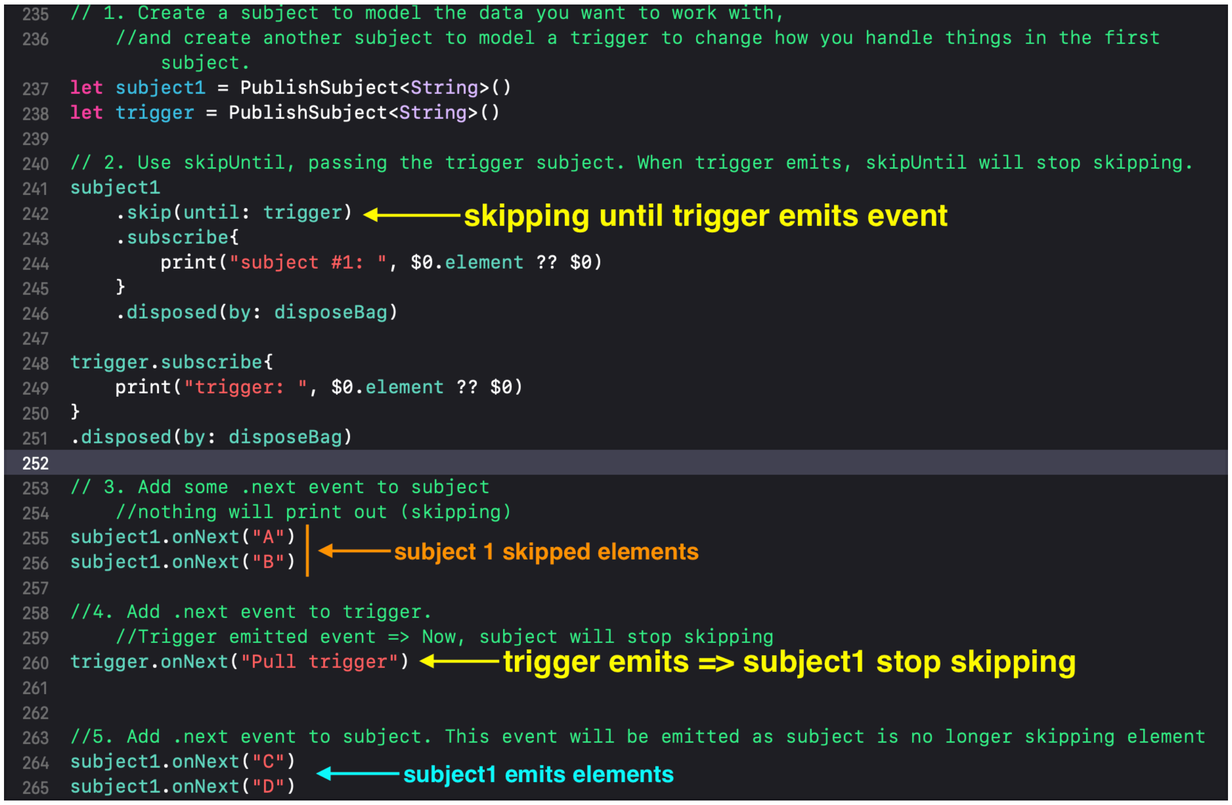
```
Subscriber #2:  Lanna
Subscriber #2:  Iam
Subscriber #2:  Hungry
Subscriber #2:  I wanna
Subscriber #2:  CHICKEN!!
Subscriber #2:  completed
```

- **skip(until:…):**

– keep skipping elements from the source observable (the one you're subscribing to) until some other *trigger* observable emits

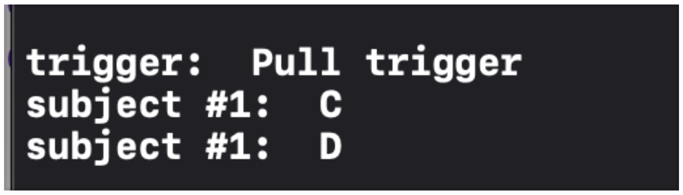


- skipUntil ignores elements emitted by the source observable (the top line) until the trigger observable (second line) emits a .next event.
- Then it stops skipping and lets everything through from that point on.



```
// 1. Create a subject to model the data you want to work with,
    //and create another subject to model a trigger to change how you handle things in the first
        subject.
let subject1 = PublishSubject<String>()
let trigger = PublishSubject<String>()

// 2. Use skipUntil, passing the trigger subject. When trigger emits, skipUntil will stop skipping.
subject1
    .skip(until: trigger)
    .subscribe{
        print("subject #1: ", $0.element ?? $0)
    }
    .disposed(by: disposeBag)

trigger.subscribe{
    print("trigger: ", $0.element ?? $0)
}
.disposed(by: disposeBag)

// 3. Add some .next event to subject
    //nothing will print out (skipping)
subject1.onNext("A")
subject1.onNext("B")

//4. Add .next event to trigger.
    //Trigger emitted event => Now, subject will stop skipping
trigger.onNext("Pull trigger")


//5. Add .next event to subject. This event will be emitted as subject is no longer skipping element
subject1.onNext("C")
subject1.onNext("D")
```

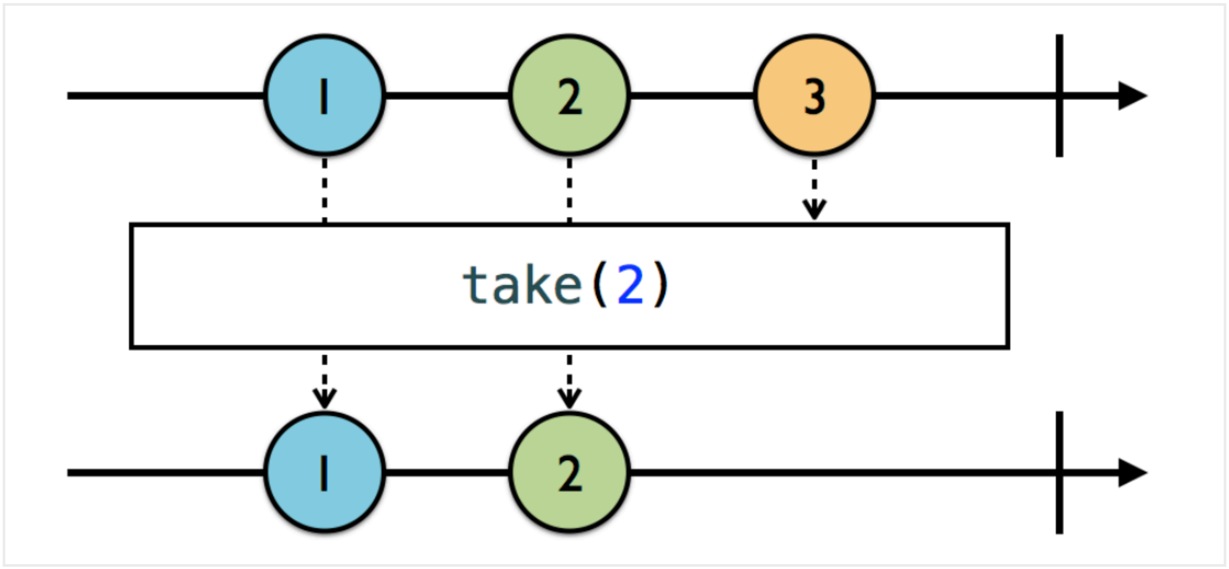
skipping until trigger emits event
subject 1 skipped elements
trigger emits => subject1 stop skipping
subject1 emits elements

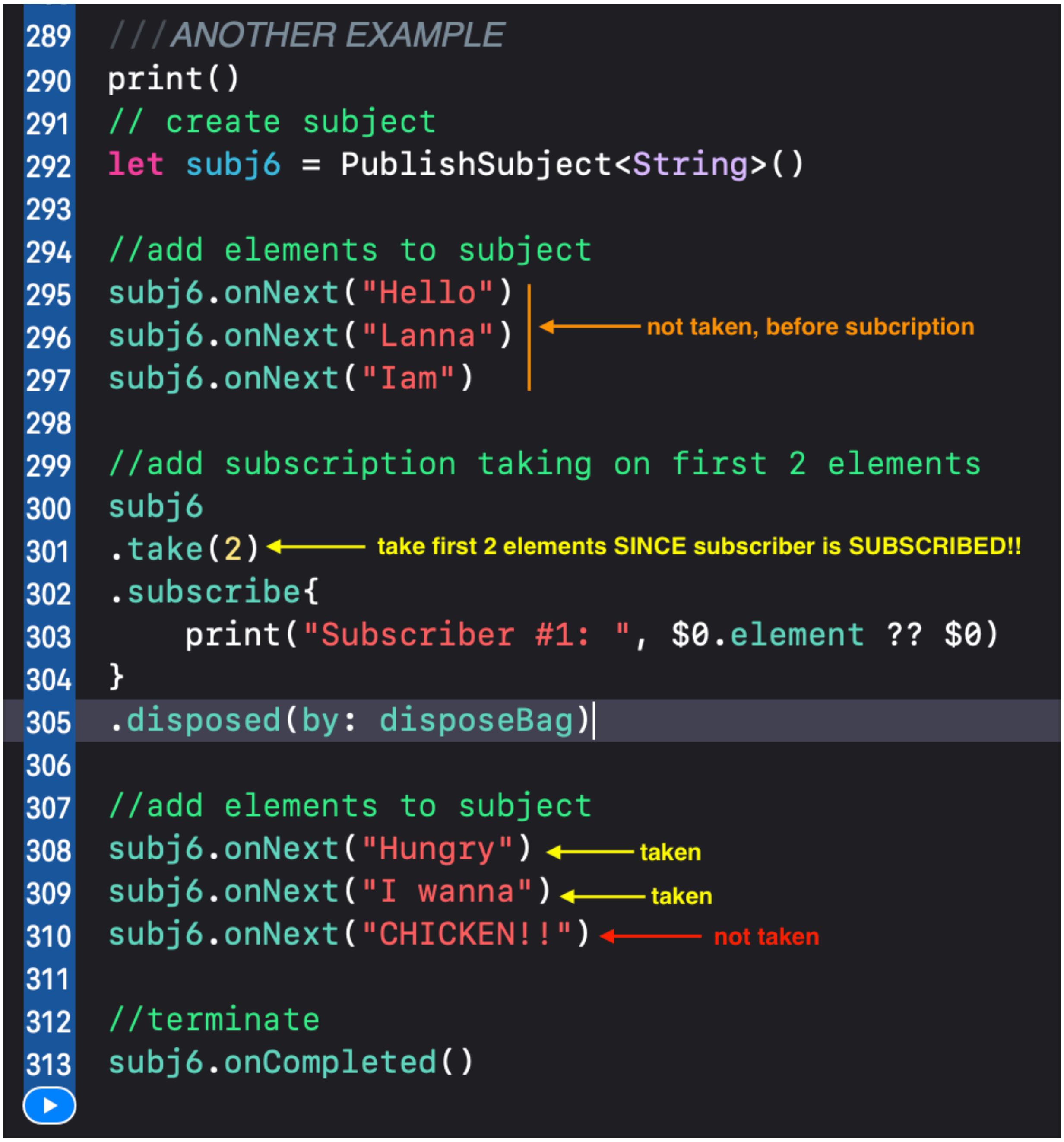


```
trigger:  Pull trigger
subject #1:  C
subject #1:  D
```

✓ Taking operators:

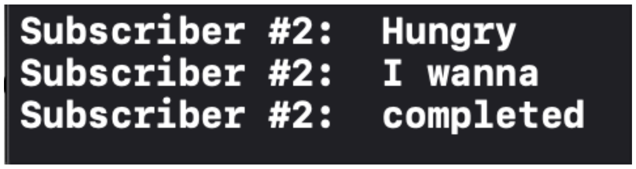
- **take:**


1
2
3
take(2)
1
2



```swift
///ANOTHER EXAMPLE
print()
// create subject
let subj6 = PublishSubject<String>()

//add elements to subject
subj6.onNext("Hello")
subj6.onNext("Lanna")
subj6.onNext("Iam")

//add subscription taking on first 2 elements
subj6
.take(2)
.subscribe{
    print("Subscriber #1: ", $0.element ?? $0)
}
.disposed(by: disposeBag)

//add elements to subject
subj6.onNext("Hungry")
subj6.onNext("I wanna")
subj6.onNext("CHICKEN!!")

//terminate
subj6.onCompleted()
```

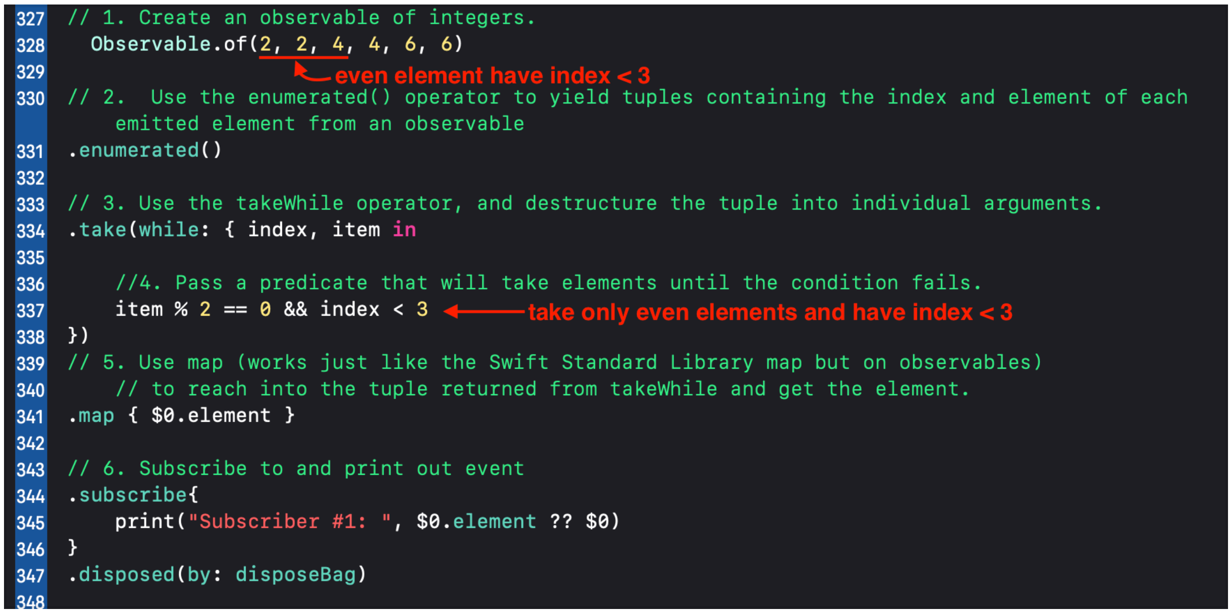
not taken, before subcription
take first 2 elements SINCE subscriber is SUBSCRIBED!!
taken
taken
not taken



```
Subscriber #2:  Hungry
Subscriber #2:  I wanna
Subscriber #2:  completed
```

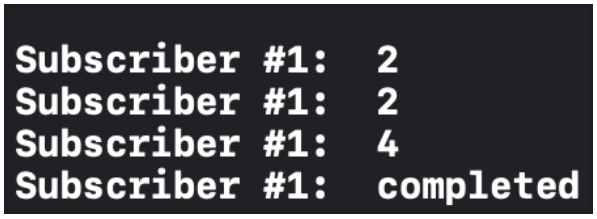
- takeWhile():

– takeWhile operator that works similarly to skipWhile, except you're taking instead of skipping.



```
// 1. Create an observable of integers.
  Observable.of(2, 2, 4, 4, 6, 6)

// 2.  Use the enumerated() operator to yield tuples containing the index and element of each
    emitted element from an observable
.enumerated()

// 3. Use the takeWhile operator, and destructure the tuple into individual arguments.
.take(while: { index, item in

    //4. Pass a predicate that will take elements until the condition fails.
    item % 2 == 0 && index < 3
})
// 5. Use map (works just like the Swift Standard Library map but on observables)
    // to reach into the tuple returned from takeWhile and get the element.
.map { $0.element }

// 6. Subscribe to and print out event
.subscribe{
    print("Subscriber #1: ", $0.element ?? $0)
}
.disposed(by: disposeBag)
```

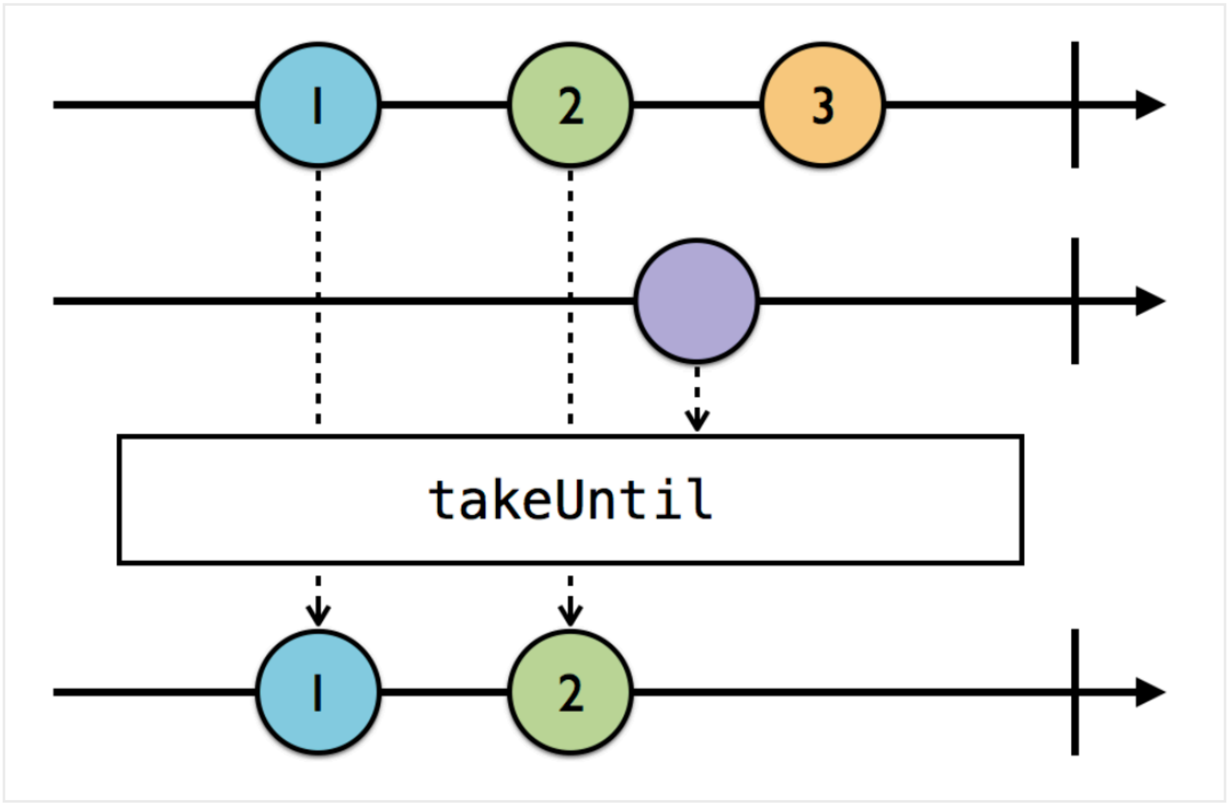
even element have index < 3
take only even elements and have index < 3

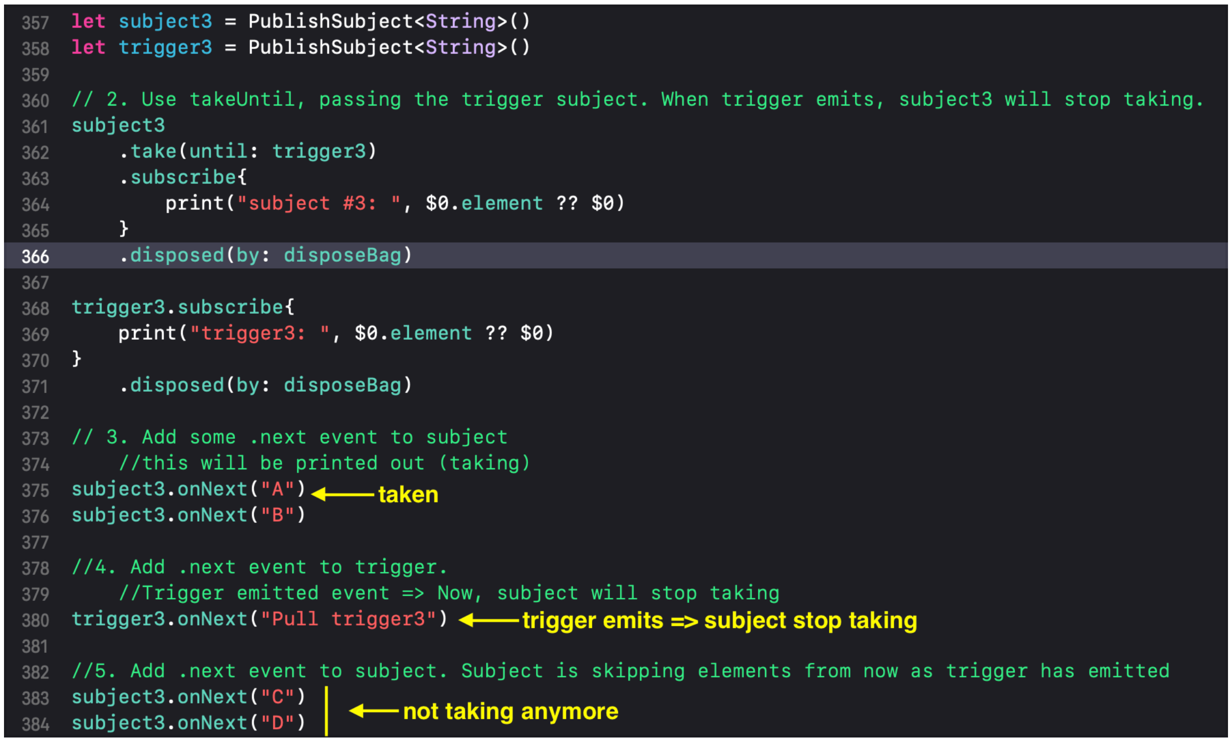


```
Subscriber #1:  2
Subscriber #1:  2
Subscriber #1:  4
Subscriber #1:  completed
```

- takeUntil():

– takeUntil operator that works similarly to skipUntil, except you’re taking instead of skipping.





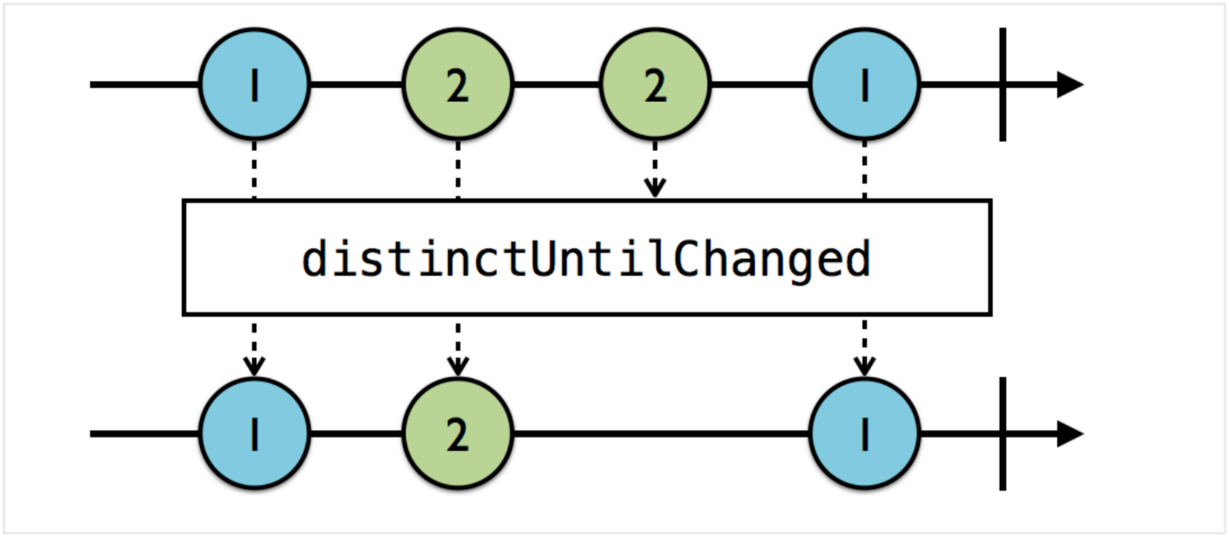
```
let subject3 = PublishSubject<String>()
let trigger3 = PublishSubject<String>()

// 2. Use takeUntil, passing the trigger subject. When trigger emits, subject3 will stop taking.
subject3
    .take(until: trigger3)
    .subscribe{
        print("subject #3: ", $0.element ?? $0)
    }
    .disposed(by: disposeBag)

trigger3.subscribe{
    print("trigger3: ", $0.element ?? $0)
}
    .disposed(by: disposeBag)

// 3. Add some .next event to subject
    //this will be printed out (taking)
subject3.onNext("A")
subject3.onNext("B")

//4. Add .next event to trigger.
    //Trigger emitted event => Now, subject will stop taking
trigger3.onNext("Pull trigger3")

//5. Add .next event to subject. Subject is skipping elements from now as trigger has emitted
subject3.onNext("C")
subject3.onNext("D")
```

```
subject #3:  A
subject #3:  B
subject #3:  completed
trigger3:  Pull trigger3
```

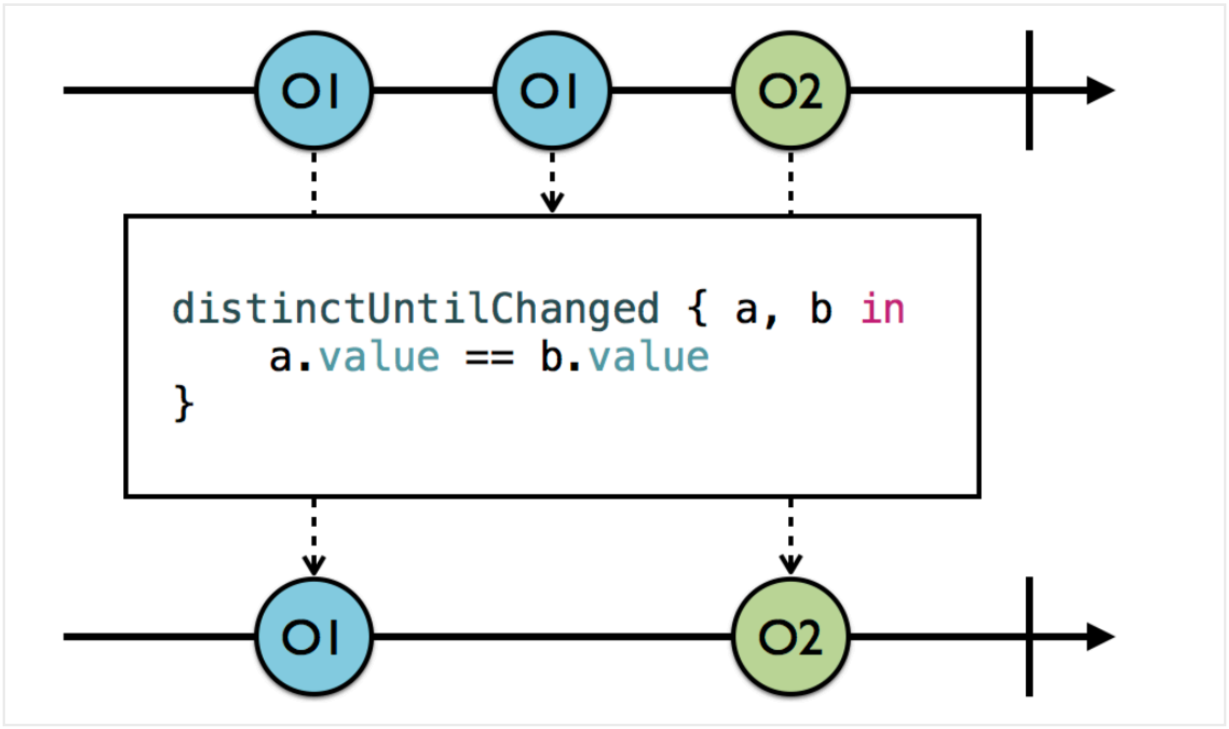
✓ Distinct operator

- Let you prevent duplicate contiguous items from getting through

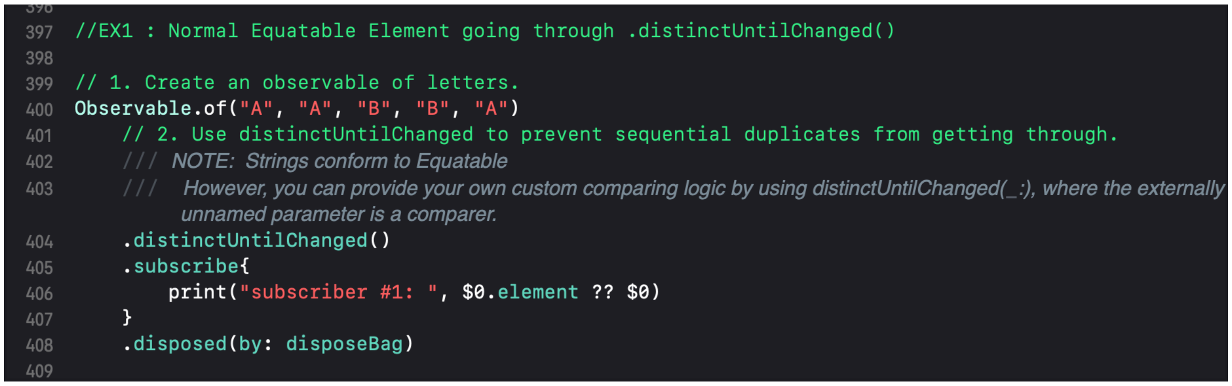
- **distinceUntilChanged**():

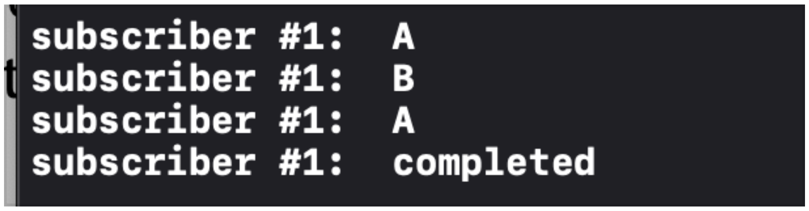


- distinctUntilChanged only prevents **duplicates that are right next to each other,** so the second 1 gets through.
- Elements are compared for equality based on their implementation conforming to **Equatable**.
  - **distinctUntilChanged**(_:) is also useful when you want to distinctly prevent duplicates for types that do not conform to Equatable as well
    - you can provide your own custom comparing logic by using distinctUntilChanged(_:), where the externally unnamed parameter is a comparer.



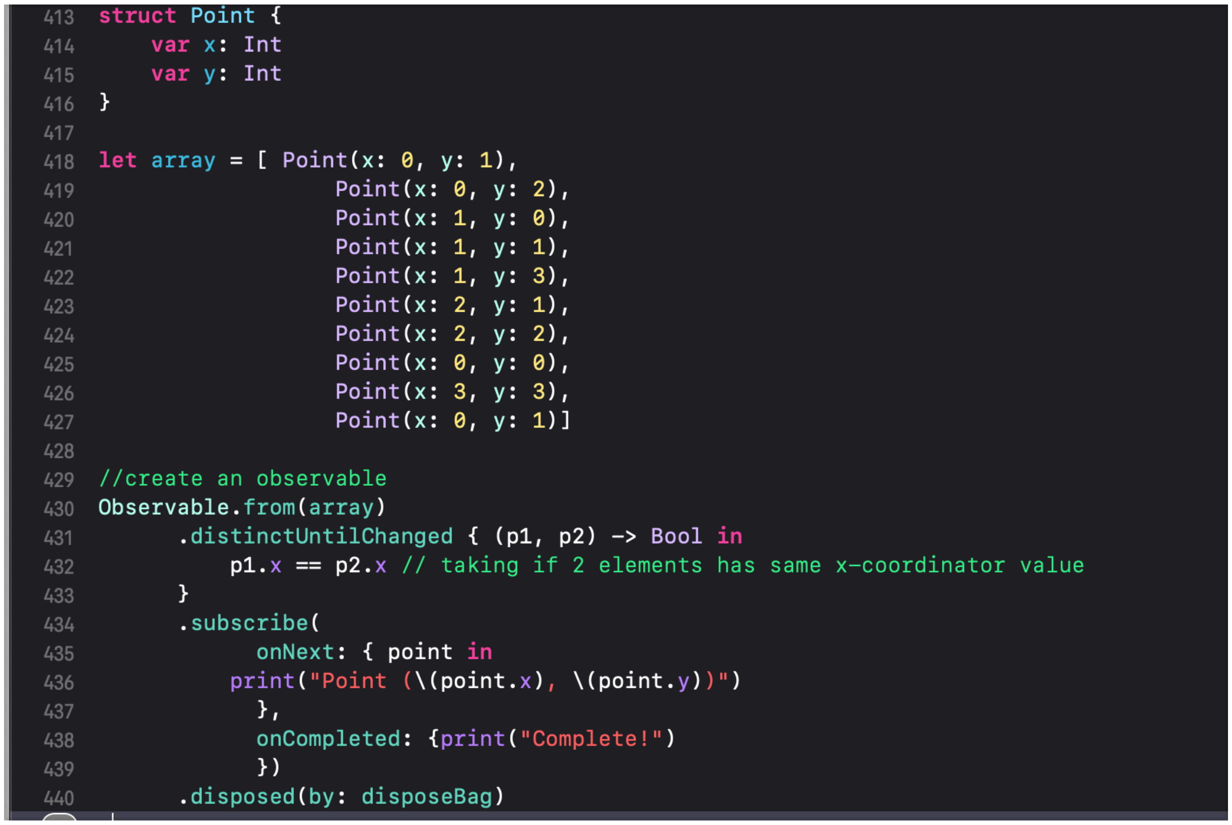
EX1: Normal using


```
//EX1 : Normal Equatable Element going through .distinctUntilChanged()

// 1. Create an observable of letters.
Observable.of("A", "A", "B", "B", "A")
    // 2. Use distinctUntilChanged to prevent sequential duplicates from getting through.
    /// NOTE: Strings conform to Equatable
    ///  However, you can provide your own custom comparing logic by using distinctUntilChanged(_:), where the externally
         unnamed parameter is a comparer.
    .distinctUntilChanged()
    .subscribe{
        print("subscriber #1: ", $0.element ?? $0)
    }
    .disposed(by: disposeBag)
```

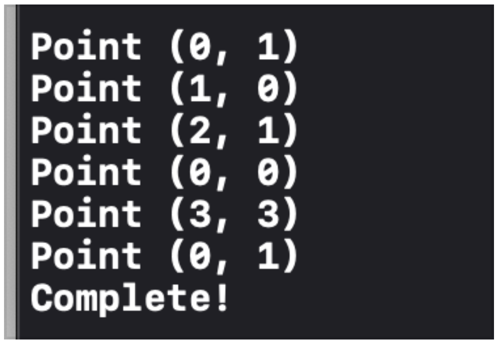

```
subscriber #1:  A
subscriber #1:  B
subscriber #1:  A
subscriber #1:  completed
```

EX2: distinceUntilChanged() with custom comparer

We want only the distinct point where x coordinate value of 2 elements right next to each other are different



```swift
struct Point {
    var x: Int
    var y: Int
}

let array = [ Point(x: 0, y: 1),
                  Point(x: 0, y: 2),
                  Point(x: 1, y: 0),
                  Point(x: 1, y: 1),
                  Point(x: 1, y: 3),
                  Point(x: 2, y: 1),
                  Point(x: 2, y: 2),
                  Point(x: 0, y: 0),
                  Point(x: 3, y: 3),
                  Point(x: 0, y: 1)]

//create an observable
Observable.from(array)
      .distinctUntilChanged { (p1, p2) -> Bool in
          p1.x == p2.x // taking if 2 elements has same x-coordinator value
      }
      .subscribe(
            onNext: { point in
          print("Point (\(point.x), \(point.y))")
            },
            onCompleted: {print("Complete!")
            })
      .disposed(by: disposeBag)
```



```
Point (0, 1)
Point (1, 0)
Point (2, 1)
Point (0, 0)
Point (3, 3)
Point (0, 1)
Complete!
```