RXSwift - Time-based Operator

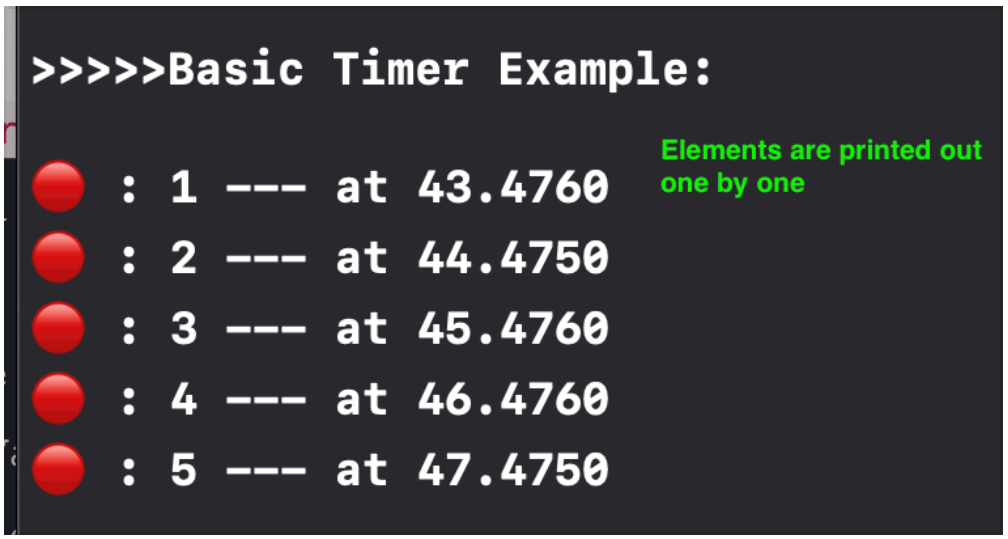
- Buffering operators
  - deal with buffering
  - either replay past elements to new subscribers, or buffer them and deliver them in bursts.
  - they allow you to control how and when past and new elements get delivered.
  - Basic steps:

EX: create an observable can emit element by time: 1 event per 1 second

```
let elementsPerSecond = 1
let maxElements = 5
let replayedElements = 1
let replayDelay: TimeInterval = 3
```

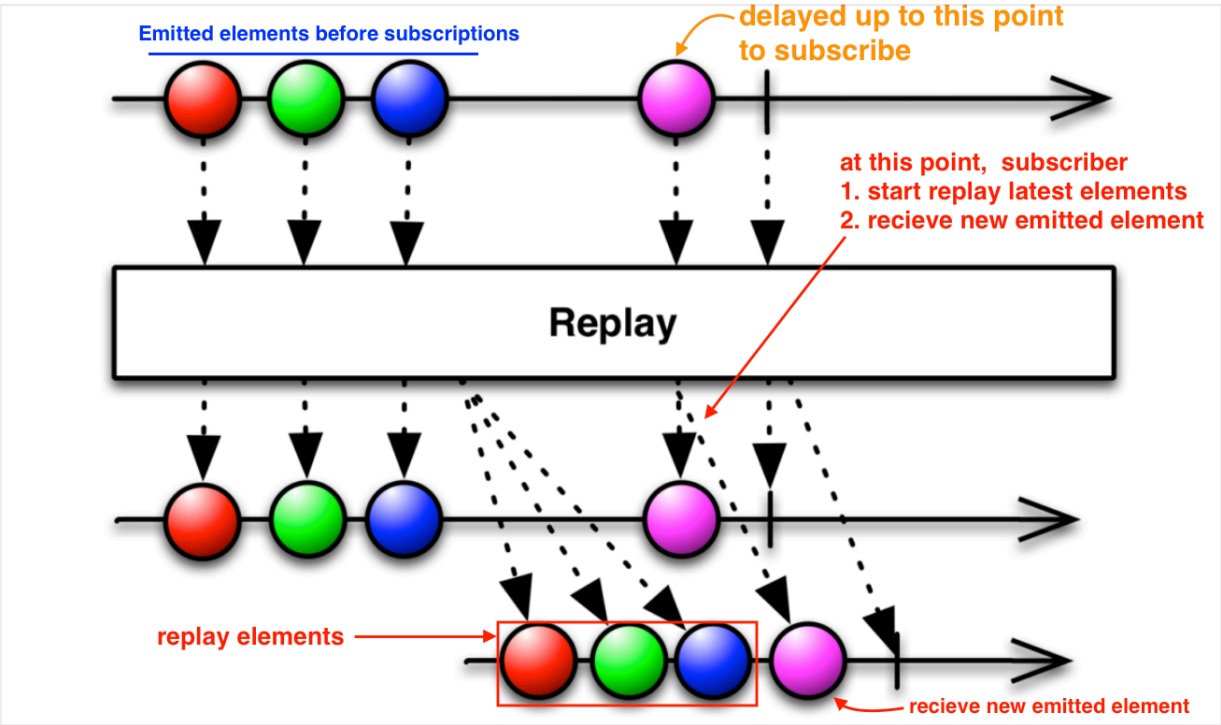
```
let source = Observable<Int>.create { observer in
    var value = 1
    /// create a timer on main queue
    let source = DispatchSource.makeTimerSource(queue: .main)   <-- 1. create a timer

    /// handle event after repeated time elapse
    source.setEventHandler {                                    <-- 2. write a closure to handle event
       if value <= maxElements {
           observer.onNext(value)
           value += 1
       }
    }
    /// schedule event
    source.schedule(deadline: .now(), repeating: 1.0 / Double(elementsPerSecond), leeway: .nanoseconds(0))
                    // start emitting imediately   // 3. schedule time to emit event   // 1 event per second

    /// start source
    source.resume()                                             <-- 4. start source

    ///stop source in Disposables.create()
    return Disposables.create {
              source.suspend()                                  <-- 5. stop source in Disposables.create()
          }
}
```

```
DispatchQueue.main.async {//Subscribe in main thread
    observable
        .subscribe(onNext: { value in
            print("🔵 : ", value)
        }, onCompleted: {
            print("🔵 Completed")
        }, onDisposed: {
            print("🔵 Disposed")
        })
        .disposed(by: bag)

}
```



** NOTE: We don't code like above as it may contains unexpected error

- Replaying past elements



- replay:
  - we need to provide it a specific number of buffer element to replay it to subscriber
  - Any saved elements are the latest elements that Observable has emitted

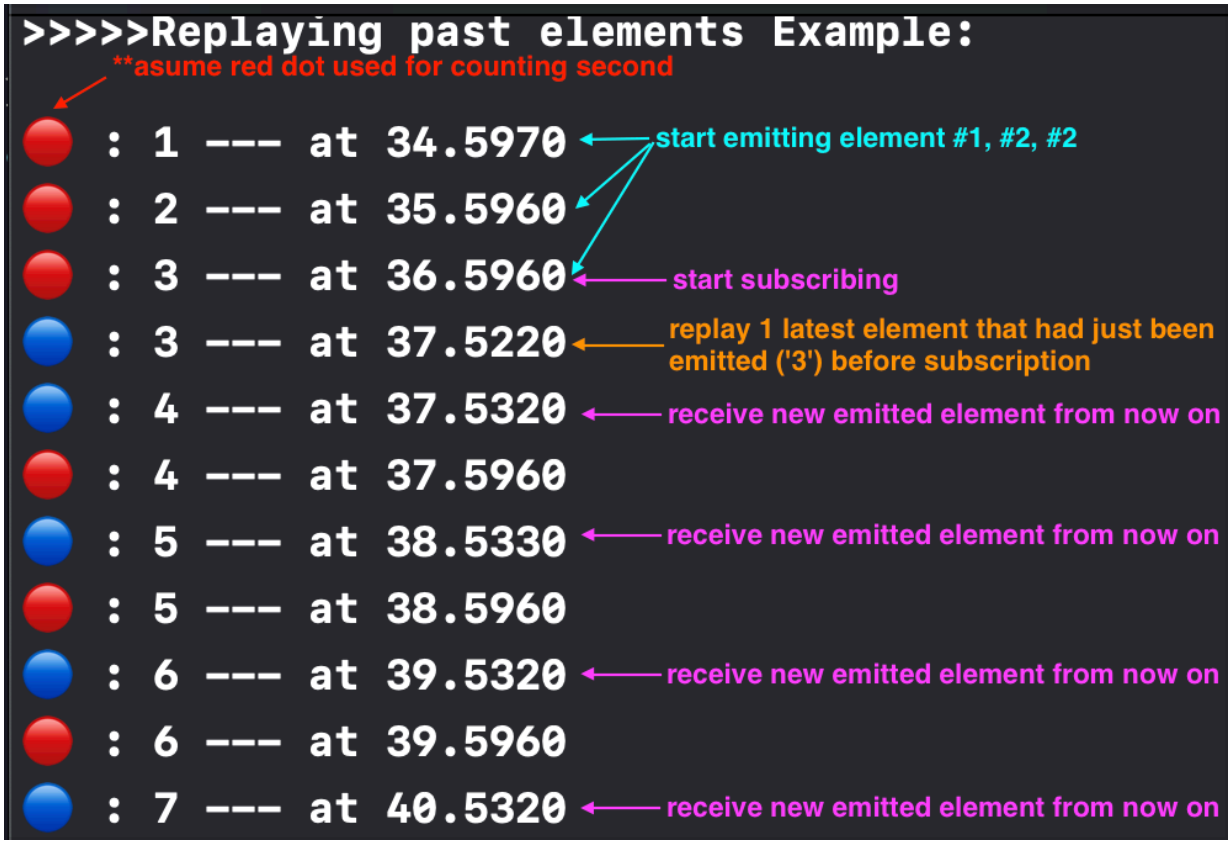
EX:
replay the latest element before subscriber start subscribing

```swift
let source1 = Observable<Int>.create { observer in
    var value = 1
    /// create a timer on main queue
    let source = DispatchSource.makeTimerSource(queue: .main)

    /// handle event after repeated time elapse
    source.setEventHandler {
       if value <= maxElements {
           observer.onNext(value)
           value += 1
       }
    }
    /// schedule event
    source.schedule(deadline: .now(), repeating: 1.0 / Double(elementsPerSecond), leeway:
        .nanoseconds(0))

    /// start source
    source.resume()

    ///stop source in Disposables.create()
    return Disposables.create {
              source.suspend()
          }
}
```

start emitting immediately

```swift
// Create a Observable Replay
let replaySource = source1.replay(1)

//Start subscribe after 3 seconds delay
DispatchQueue.main.asyncAfter(deadline: .now() + 3) {
    replaySource
        .subscribe(onNext: { value in
            printValue("🔵 : \(value)")
        }, onCompleted: {
            print("🔵 Completed")
        }, onDisposed: {
            print("🔵 Disposed")
        })
        .disposed(by: disposeBag)
}
///** NOTE: because it belongs to ConnectableObservable, therefore, to execute, we must connect
    it to observable by calling .connect()
replaySource.connect()
```

wait for 3 sec before start subscribing



- replayAll:
  - replay all emitted elements before subscriber start subscribing

```swift
let source1 = Observable<Int>.create { observer in
    var value = 1
    /// create a timer on main queue
    let source = DispatchSource.makeTimerSource(queue: .main)

    /// handle event after repeated time elapse
    source.setEventHandler {
       if value <= maxElements {
           observer.onNext(value)
           value += 1
       }
    }
    /// schedule event
    source.schedule(deadline: .now()+1, repeating: 1.0 /
        Double(elementsPerSecond), leeway: .nanoseconds(0))

    /// start source
    source.resume()

    ///stop source in Disposables.create()
    return Disposables.create {
              source.suspend()
          }
}
```

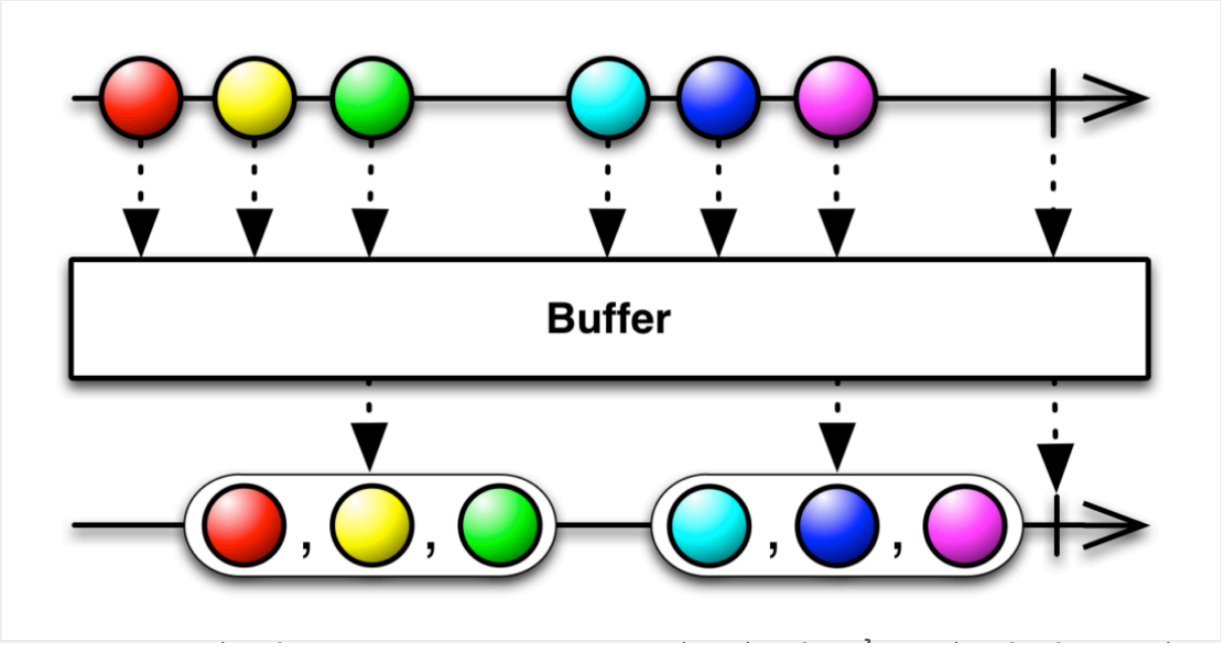
wait for 1 sec before emitting events

```swift
let replaySource2 = source1.replayAll()
DispatchQueue.main.asyncAfter(deadline: .now() + 3) {
    replaySource2
        .subscribe(onNext: { value in
            printValue("🟠 : \(value)")
        }, onCompleted: {
            print("🟠 Completed")
        }, onDisposed: {
            print("🟠 Disposed")
        })
        .disposed(by: disposeBag)
}
replaySource2.connect()
```

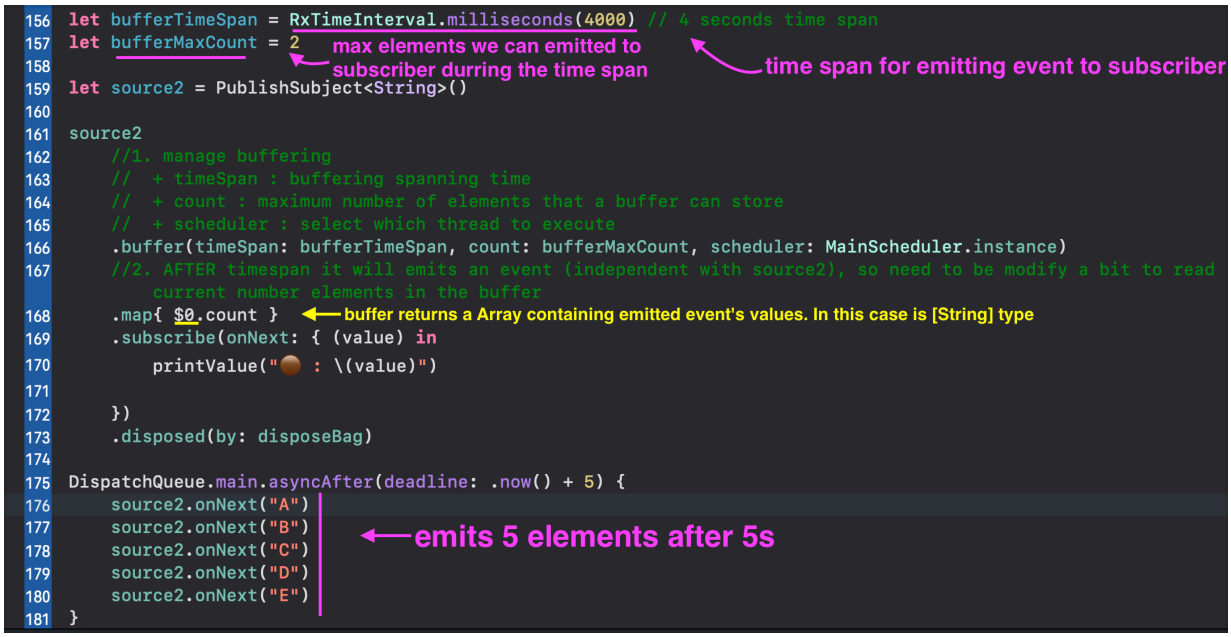
start subscribing after 3 sec

```
>>>>>Replaying past elements Example:
  Red dot is used to count second
: 1 --- at 24.6850
: 2 --- at 25.6820   <- after 1s, event #1 & #2 have been emitted
: 3 --- at 26.6820   <- After 3s, start subscribing
: 1 --- at 27.6170   <- Replay ALL elements (2 elements) that had been
: 2 --- at 27.6180      emitted before subscription
: 3 --- at 27.6190   <- recieve new emitted element
: 4 --- at 27.6820
: 4 --- at 28.6190   <- recieve new emitted element
: 5 --- at 28.6840
: 5 --- at 29.6170   <- recieve new emitted element
: 6 --- at 29.6820
: 6 --- at 30.6170   <- recieve new emitted element
```

- Controlled buffering

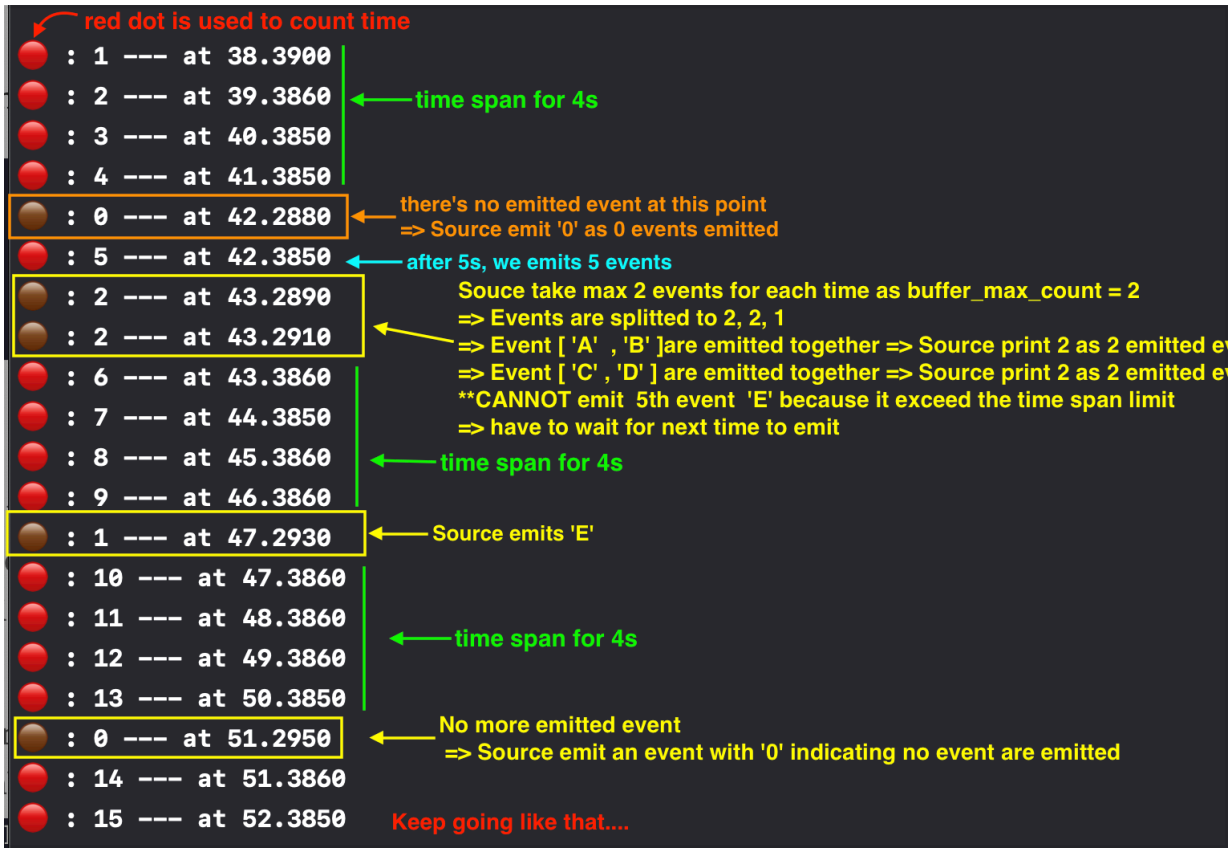


EX:

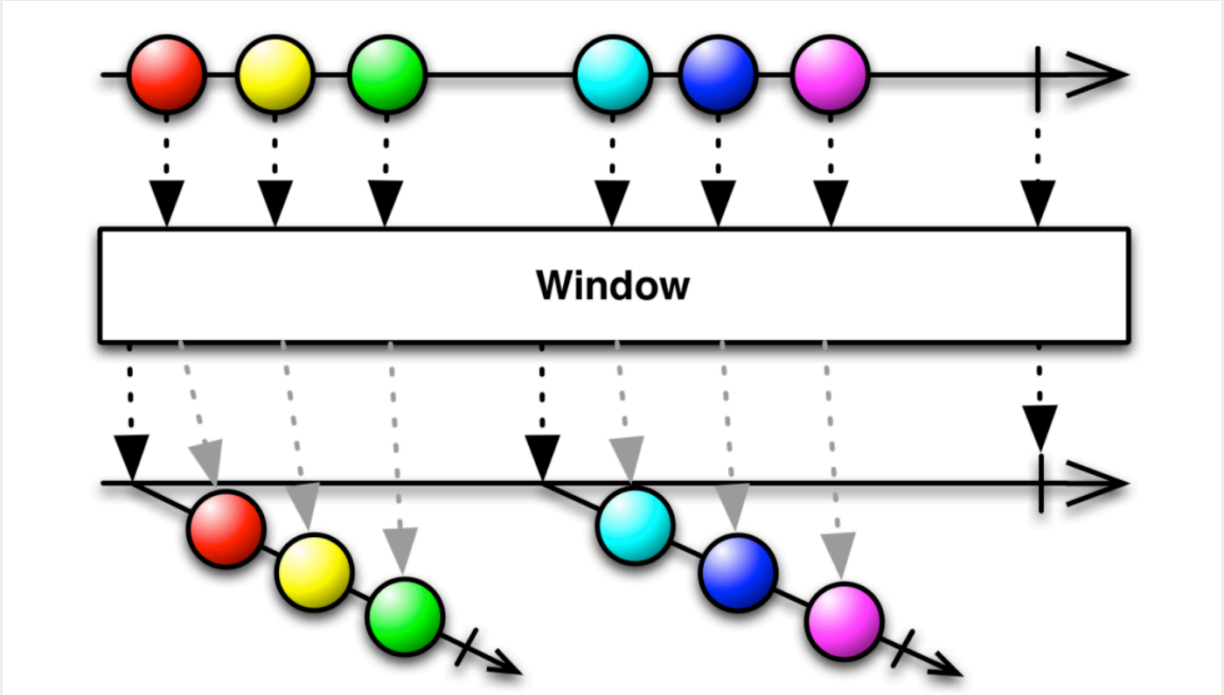


Giải thích code:

- Sẽ có khung thời gian **4s để nhận event hoặc đóng không làm gì cả.**
- Khi đến **thời gian 4s** cho phép Source **nhận event**, Source sẽ emit events cho subscriber.
  - Lưu ý là số event được emit cũng sẽ bị giới hạn bởi ***bufferMaxCount***.
  - Source không thể emit nhiều hơn số lượng cho phép
  - Nếu có event được phát ra:
    - nếu số event <= buffer_max_count :
      - emit tất cả event rồi delay tiếp
    - nếu số event > buffer_max_count :
      - chỉ emit đúng số event cho phép (max_count). Còn như event dư lại sẽ được emit đợt sau
      - Sau khi emit xong thì delay tiếp
  - Nếu không có event được phát ra:
    - emit 1 event rỗng, không chứa dữ liệu
- Khi đến **thời gian 4s đóng:**
  - Sẽ không nhận phát bất cứ event nào trong thời gian này



- window:
  - similar to *buffer*
  - BUT in *window*, the buffer data type is a observable, not an Array type like *buffer*



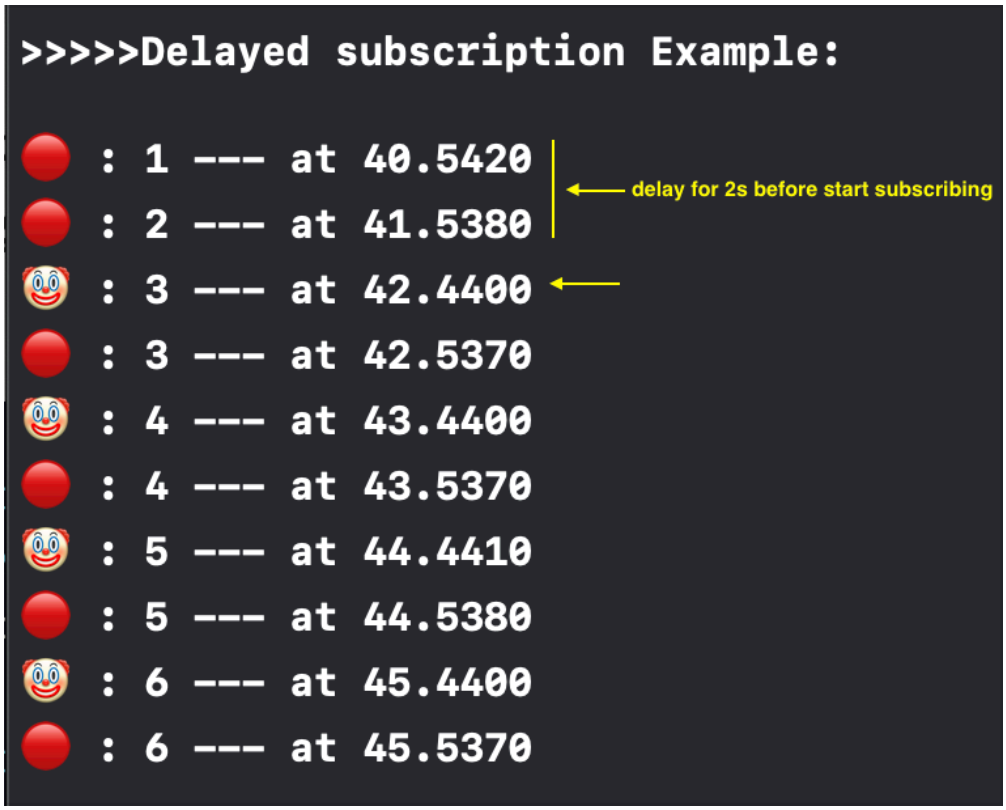
```
source2
    //source.window : Observable<String>
    .window(timeSpan: bufferTimeSpan, count: bufferMaxCount, scheduler: MainScheduler.instance)
    //convert Observable<String> to Array String with flat map
    .flatMap({ obs -> Observable<[String]> in
        // because it's an Observable, we scan all its emitted element into 1 single value
        obs.scan(into: []) { $0.append($1) }
    })
    .subscribe(onNext: { (value) in
        printValue("👻 \(value)")
    })
    .disposed(by: disposeBag)
```

```
>>>>>Controlled buffering Example:

🔴 : 1 --- at 57.2250
🔴 : 2 --- at 58.2230
🔴 : 3 --- at 59.2230
👻 ["A"] --- at 00.2070
👻 ["A", "B"] --- at 00.2090
👻 ["C"] --- at 00.2110
👻 ["C", "D"] --- at 00.2120
👻 ["E"] --- at 00.2140
👻 ["E", "F"] --- at 00.2160
👻 ["G"] --- at 00.2180
🔴 : 4 --- at 00.2220
🔴 : 5 --- at 01.2230
🔴 : 6 --- at 02.2230
🔴 : 7 --- at 03.2220
🔴 : 8 --- at 04.2220
```

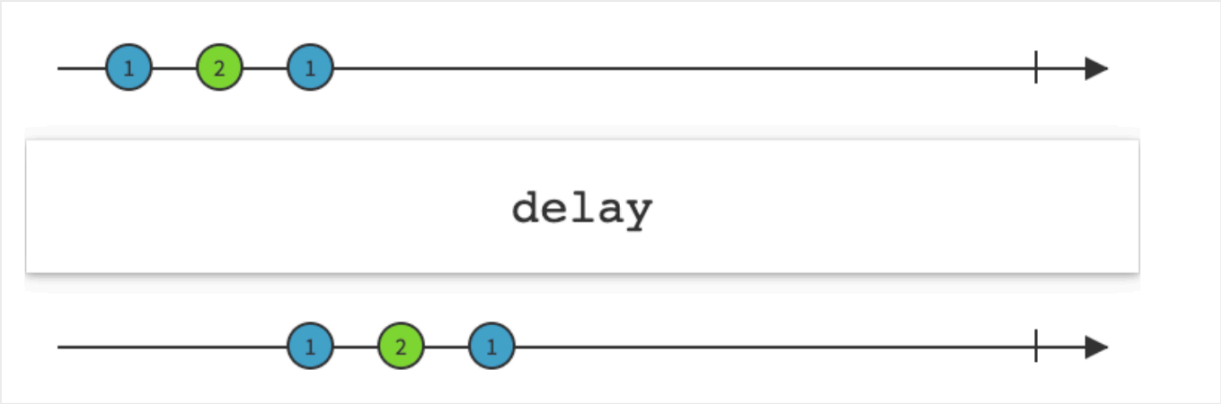
✅ Time-shifting operators

- Delayed subscription:
  - delaying register a subscription to certain amount of time.
  - Of course, any emitted element before the subscription will not be received

```swift
let source3 = PublishSubject<String>()

source3
    // delay scubcription for 2 second at start
    .delaySubscription(RxTimeInterval.seconds(2), scheduler: MainScheduler.instance)
    .subscribe(onNext: { value in
        printValue("🤡 : \(value)")
    })
    .disposed(by: disposeBag)

var count = 1
// for every 1s, emit event once.
/// **NOTE:
///  + the 1st element we recieved is 3
///  + if we don't do : timer.suspend(), it will never end
var timer = DispatchSource.timer(interval: 1.0, queue: .main) {
    source3.onNext("\(count)")
    count += 1
}
```



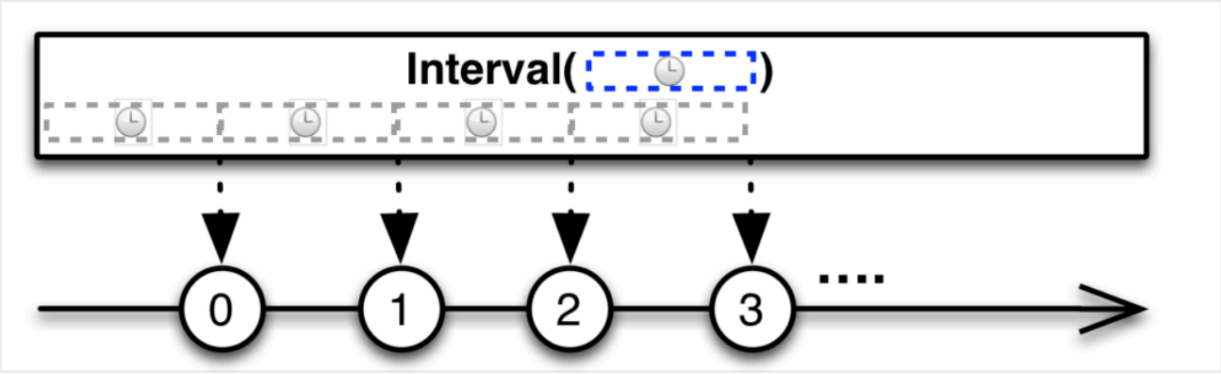
- Delayed elements
  - delay receiving elements




```swift
let source4 = PublishSubject<String>()

source4
    // delay elements for 2 second
    .delay(RxTimeInterval.seconds(2), scheduler: MainScheduler.instance)
    .subscribe(onNext: { value in
        printValue("🤖 recieve: \(value)")
    })
    .disposed(by: disposeBag)

var count4 = 1
var timer4 = DispatchSource.timer(interval: 1.0, queue: .main) {
    printValue("emit: \(count4)")
    source4.onNext("\(count4)")
    count4 += 1
}
```

```
>>>>>Delayed elements Example:

emit: 1 --- at 48.3160
emit: 2 --- at 49.2290
emit: 3 --- at 50.2280
🤖 recieve: 1 --- at 50.3210
emit: 4 --- at 51.2290
🤖 recieve: 2 --- at 51.3230
emit: 5 --- at 52.2290
🤖 recieve: 3 --- at 52.3260
emit: 6 --- at 53.2290
🤖 recieve: 4 --- at 53.3280
emit: 7 --- at 54.2280
🤖 recieve: 5 --- at 54.3290
emit: 8 --- at 55.2280
🤖 recieve: 6 --- at 55.3310
emit: 9 --- at 56.2280
🤖 recieve: 7 --- at 56.3320
```

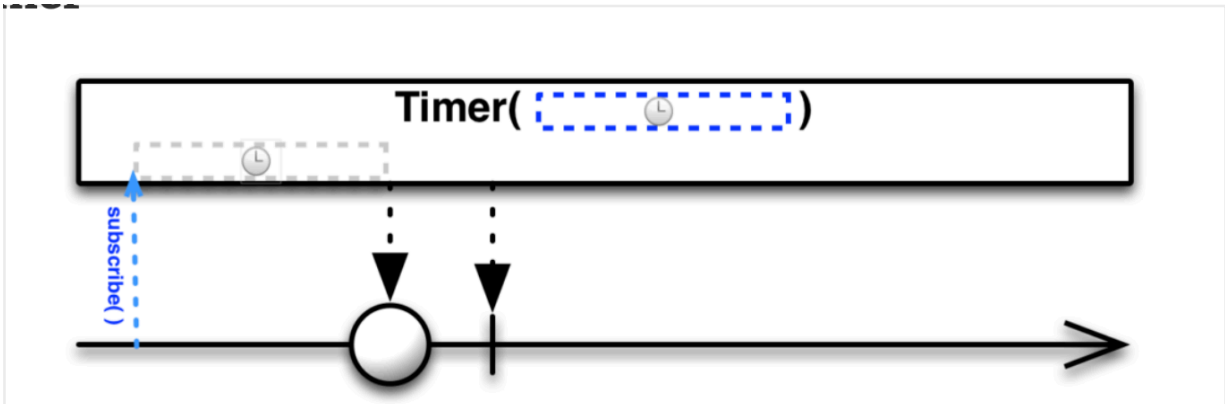
- Timer operators
  - Interval
    - Another way to do DispatchSource.timer()



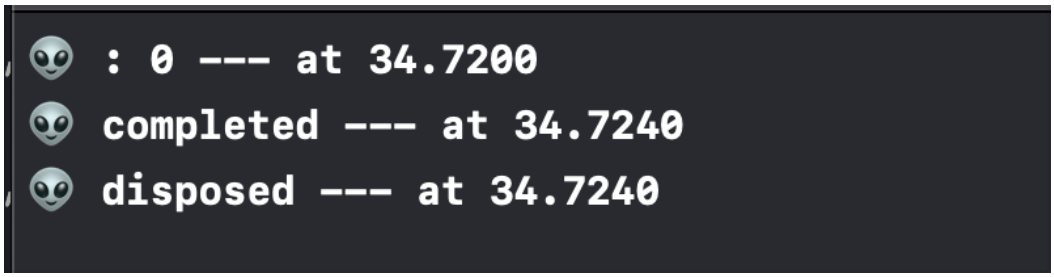
```swift
// for every 1s, emits 1 element
let source5 = Observable<Int>.interval(RxTimeInterval.seconds(1), scheduler: MainScheduler.instance)
let replay5 = source5.replay(2)

DispatchQueue.main.asyncAfter(deadline: .now()) {
    source5
        .subscribe(
            onNext: { value in
            printValue("😈 : \(value)")
        }, onCompleted: {
            print("😈 Completed")
        }, onDisposed: {
            print("😈 Disposed")
        })
        .disposed(by: disposeBag)

}
replay5.connect()
```

```
😈 : 0 --- at 43.4900
😈 : 1 --- at 44.4900
😈 : 2 --- at 45.4890
😈 : 3 --- at 46.4890
😈 : 4 --- at 47.4890
```

- Timer:
  - similar to interval
  - set a timer, and when time comes, emit the value



```swift
let source6 = Observable<Int>.timer(RxTimeInterval.seconds(3), scheduler: MainScheduler.instance)

/// **NOTE: it only emits 1 element then terminate**
source6
    /// **To make it emits more than 1, we use flatMap()**
    /// Here in flatMap(), we continue delaying for 2s more before subscriber can recieve events
    .flatMap({ _ in
        source6.delay(RxTimeInterval.seconds(2), scheduler: MainScheduler.instance)
    })
    .subscribe { value in
        printValue("👽 : \(value)")
    } onError: { error in
        printValue("👽 error")
    } onCompleted: {
        printValue("👽 completed")

    } onDisposed: {
        printValue("👽 disposed")
    }
    .disposed(by: disposeBag)
```



- Timeout:
    - Similar to request API, if there are no emits left, it will automatically terminate

```swift
let source7 = PublishSubject<Int>()

source7
    /// ** time set for 5s. After 5s, terminate if doesn't recieve anymore emits**
    .timeout(RxTimeInterval.seconds(5), scheduler: MainScheduler.instance)
    .subscribe { value in
        printValue("🐶 : \(value)")
    } onError: { error in
        printValue("🐶 error")
    } onCompleted: {
        printValue("🐶 completed")

    } onDisposed: {
        printValue("🐶 disposed")
    }
    .disposed(by: disposeBag)

DispatchQueue.main.asyncAfter(deadline: .now() + 1) {
    source7.onNext(1)
}

DispatchQueue.main.asyncAfter(deadline: .now() + 2) {
    source7.onNext(2)
}

DispatchQueue.main.asyncAfter(deadline: .now() + 6) {
    source7.onNext(3)
}

///**NOTE: this exceed 5s rule, "4" emits after 6s, therefore, subscriber won't recieve this as source has already terminated**
DispatchQueue.main.asyncAfter(deadline: .now() + 12) {
    source7.onNext(4)
}
```

```
🐶 : 1 --- at 07.8260
🐶 : 2 --- at 08.8260
🐶 : 3 --- at 12.8260
🐶 error --- at 17.8280
🐶 disposed --- at 17.8290
```