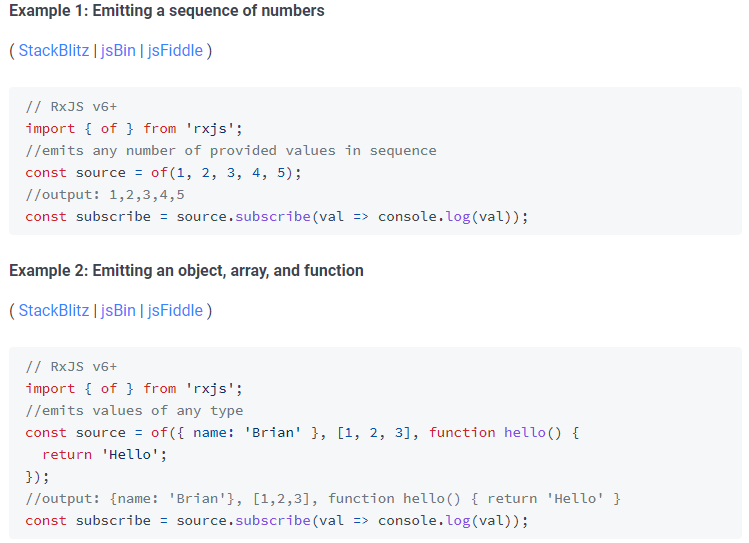
# Creation operators(Dùng để tạo observable)(dùng trong service)

## Of: là function nhận vào giá trị (bất cứ giá trị nào: string, number, boolean, array, object, Promise.resolve(something)) và lấy giá trị đó tạo observable và emit giá trị đó (nếu giá trị trống thì sẽ complete ngay lập tức, nếu giá trị đó là Promise thì trả về promise chứ ko resolve) và complete



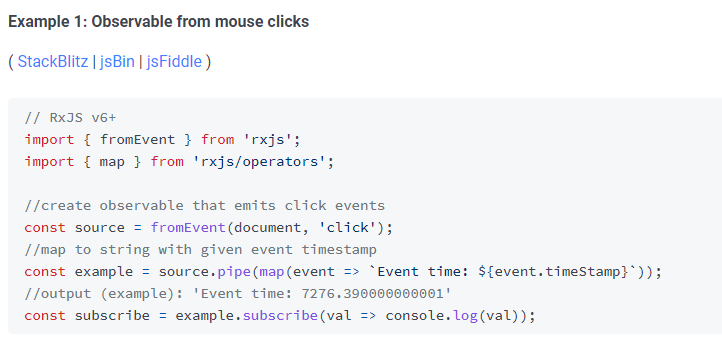
## From: tương tự of nhưng khác là nó chỉ nhận vào Promise, array, Iterable (iterable là những giá trị có thể iterate được map, string, array, set)

* Khi truyền vào Promise.resolve(something) thì nó sẽ resolve và trả về something (và cũng là cách chuyển promise sang observable)
* Khi truyền array (hoặc dãy phần tử) thì nó emit từng phần tử chứ ko emit nguyên cả array giống như of
* Khi truyền vào chuỗi string thì nó sẽ emit từng kí tự của chuỗi đó





## FromEvent: lắng nghe một sự kiện mà người dùng tương tác và chuyển nó thành 1 observable và nó không tự complete giống như of và from khi mình subscribe



## FromEventPattern: nhận vào 2 callback(1 cái là addhandler, 1 cái là removehandler (thường dùng để chuyển đổi websocket và signal alr)

## Interval: truyền vào số milisends tạo một observable khi subscribe thì mỗi giây sẽ emit giá trị và sẽ không bao giờ complete cho khi ta complete (dùng trong trường hợp ta có timer, count down)//tương tự setinterval

## Timer: có 2 cách sử dụng timer(milisecond), timer (miliseconds, milisecond)//tương tự settimeout

* Delay một khoảng thời gian sau đó emit giá trị

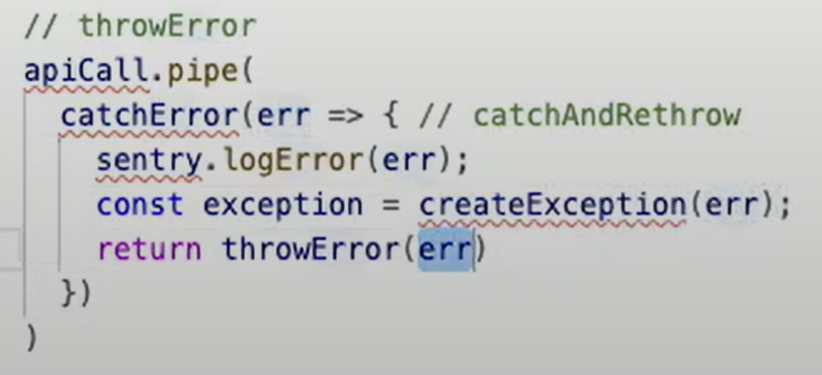
Nhưng khác settimeout ở chỗ là nó sẽ tự động complete và không phải clear timeout

* Sau milisecond đầu tiên thì nó sẽ interval vs milisecond 2



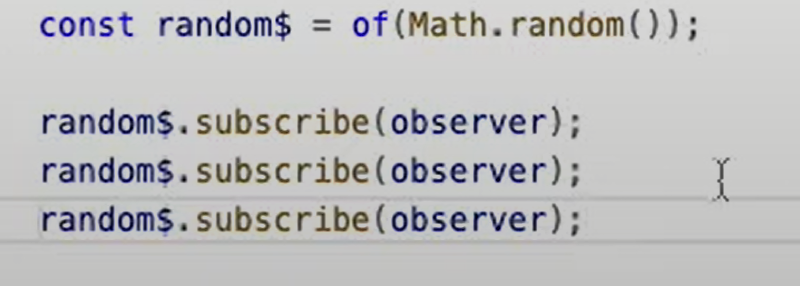
## ThrowError: hàm error sẽ chạy và không complete, được sử dụng khi mình muốn catch một lỗi nào đó



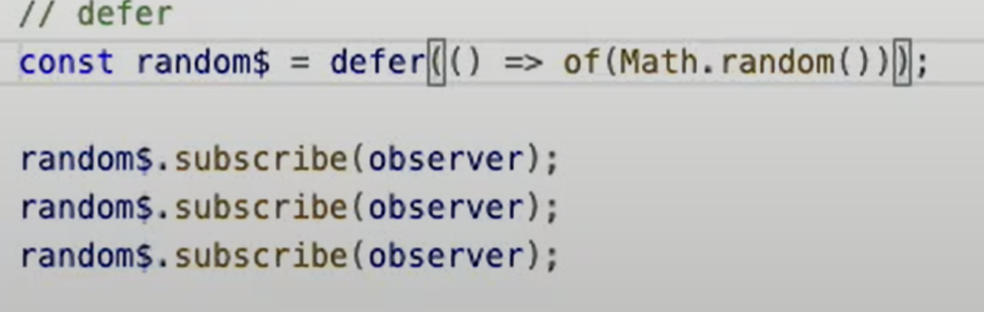


## Defer

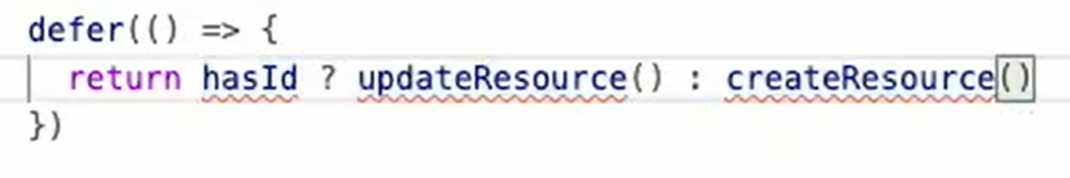
Đặt vấn đề ta có 1 API và ta muốn tạo random id và muốn so sánh id có tồn tại hay không nếu đã tồn tại rồi thì retry chạy lại nhưng nếu sử dụng of thì không thực hiện được việc đó vì nó chỉ cho cùng 1 kết quả



* Defer sẽ giải quyết vấn đề trên trong mỗi subscriber ta sẽ tạo 1 observable mới



* Defer nhận vào một observable factory vì nó là một function nên nó sẽ được khi nào có subscribe thì nó sẽ chạy function trong defer và khi funciton được chạy thì hàm of sẽ được chạy tạo cho mình observable mới khi of chạy thì math random cũng chạy ngay thời điểm mà subscribe nên là nó sẽ nhận được giá trị mới
* vì defer nhận vào một funciton nên ta có thể dùng defer để tạo 1 observable dựa vào cái điều kiện



# Transformation operator

Sử dụng pipable operator (thay cho . function): thay vì các operator được call độc lập thì nó sẽ được call trong pipe() method: nhận vào một observable và trả ra 1 observable khác

Cú pháp: 

* với cú pháp trên thì có pipe bao nhiêu operator đi nữa thì observableInstance vẫn không đổi nên để sử dụng được ta cần .subscribe ngay sau khi pipe hoặc gán lại như sau

  
\* TS chúng ta lại sử dụng pipable operator?

Trước v5.5 thì dùng cú pháp prototype method chain. (gọi nối nhau) và ta phải thêm các phương thực được gọi và observable thông qua prototype object mà như vậy mỗi lần chạy 1 tính năng nào đó nó phải load hết những operator đó lên thì dẫn đến hiệu năng ko tốt nên từ v5.5 trở đi ngta đã chuyển sang dùng pipe operator để call các operator một cách độc lập.

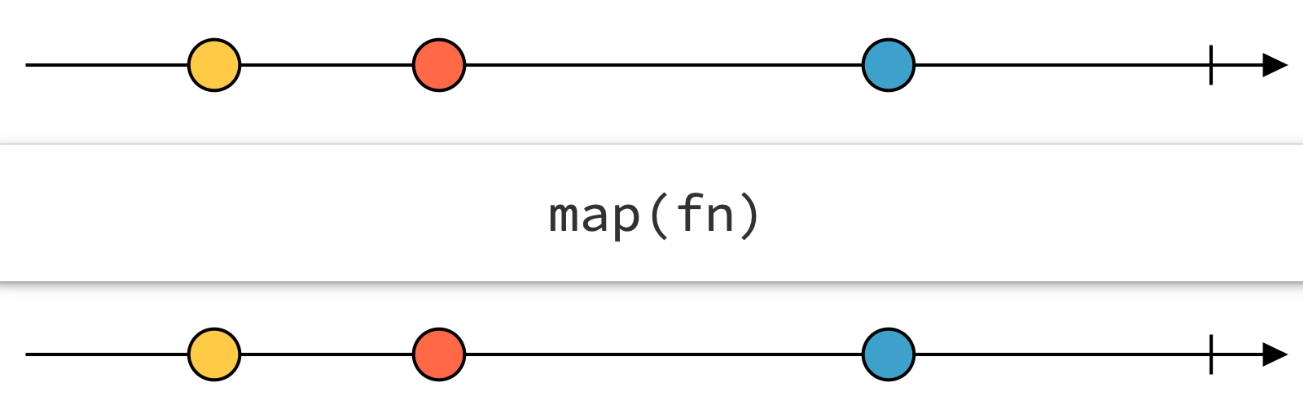
* pipable operator có thể chia thành nhiều loại khác nhau như transformations operator

## map

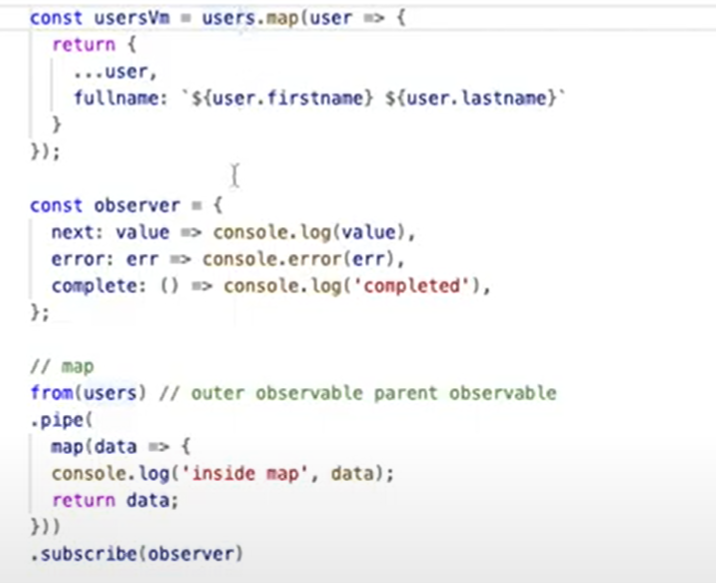
* khi observable trả về data thì thay vì ta vào hàm next để thực hiện logic thì ta có thể transform nó trước khi nó đi tới cuối nên ta sẽ sử dụng operator
* nhận vào một function

ví dụ: 

observable source sau khi pipe sẽ trả về id của user và sau đó in ra iduser từ hàm next của observer.

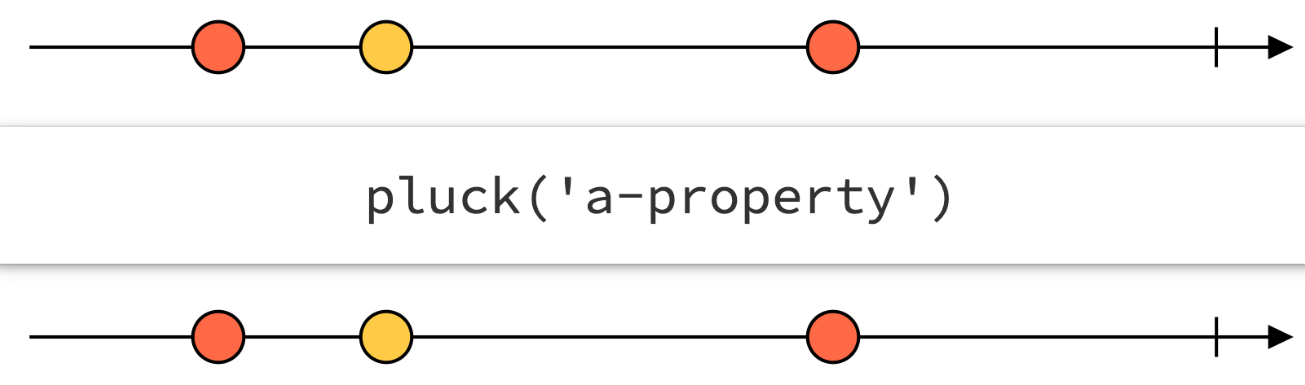


Nó sẽ map với từng phần tử mà observable emit cho ta

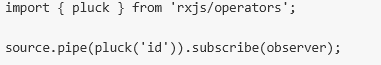


Ví dụ đối với operator là from thì nó sẽ map vs từng phần tử nhưng đối với toán tử of thì nó sẽ map với toàn bộ giá trị mà of nhận vào.

## pluck

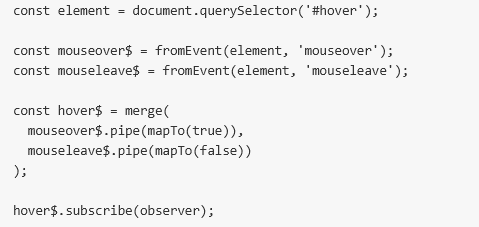


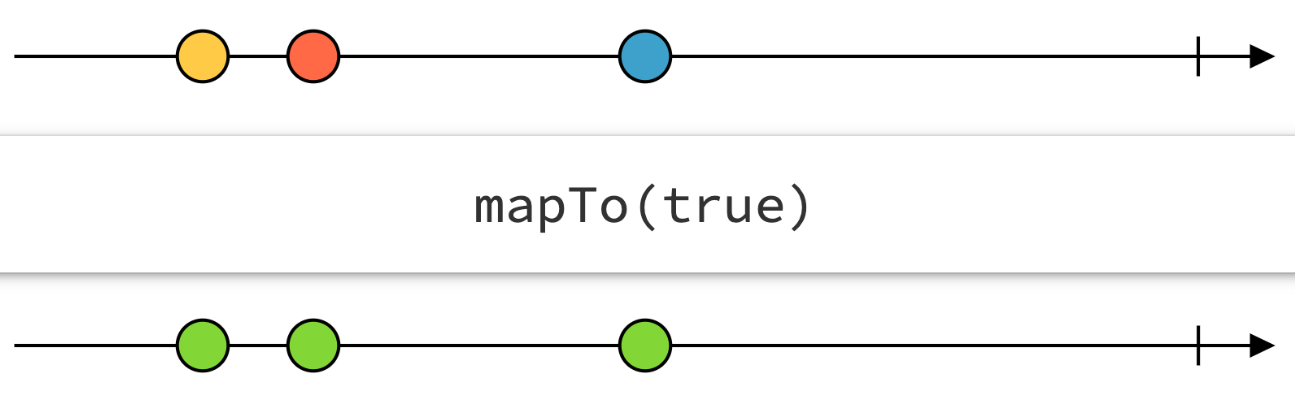
* lấy ra property của một object



## mapTo: map 1 observable sang 1 gt mặc định (dùng trong trường hợp ta có 1 event nào đó mà trên DOM mà mình ko muốn quan tâm đến giá trị của event đó mình chỉ muốn biết là khi mà event đó xảy ra thì ta có gt này)

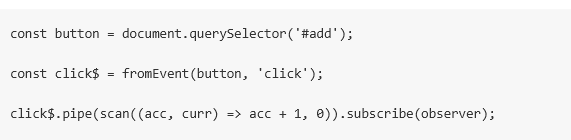
ví dụ:

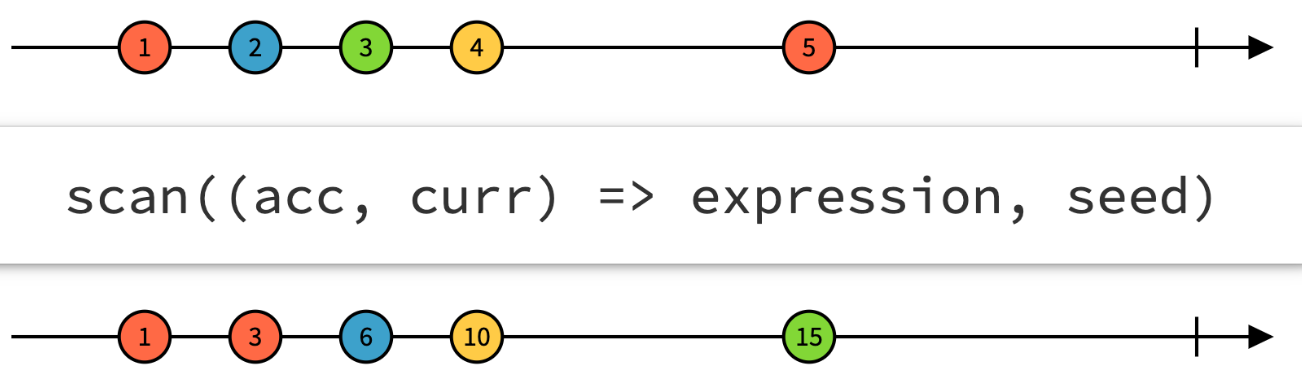


Khi mouse over trả về true và khi mouse leave trả về false (và giờ đây ta đã có 1 stream hover$ để bik được khi nào ta in/out element) 

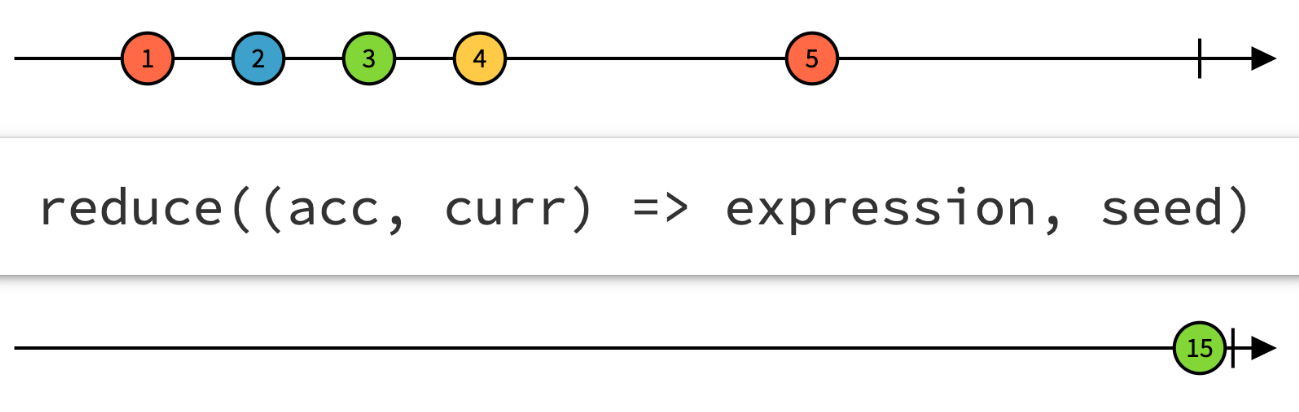
## scan (giống reduce trong array js)

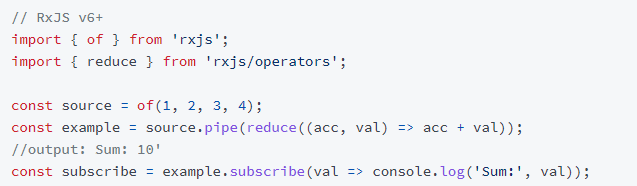
* mỗi lần stream emit value, ta sẽ apply một function lên value đó , sau đó có một biến lưu trữ kết quả trước đó.
* Ví dụ: đếm số lần người dùng click vào button





## reduce: hoạt động tương tự array tuy nhiên nó chỉ chạy khi observable complete





## toArray

* Giả sử ta cần gôm hết toàn bộ value emit bởi 1 stream rồi lưu trữ thành một array, sau đó đợi stream complete thì emit một array và complete, ta làm như sau:

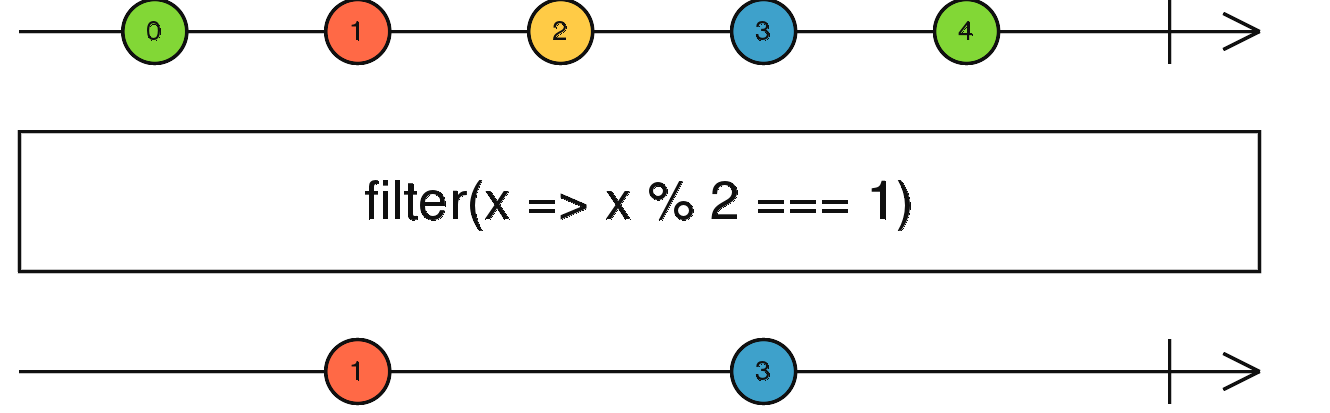


* Nhưng ngắn gọn hơn ta dùng:



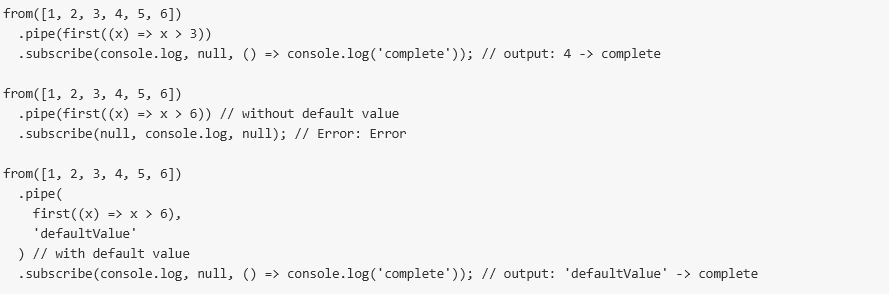
# Filtering operators

## Filter: cách hoạt động như filter() trong array js. Nhận vào một function mà function này trả về true hoặc false. Nếu true thì filter sẽ emit giá trị của obsevable tại thời điểm đó. Ngược lại false thì filter sẽ không emit giá trị đó.



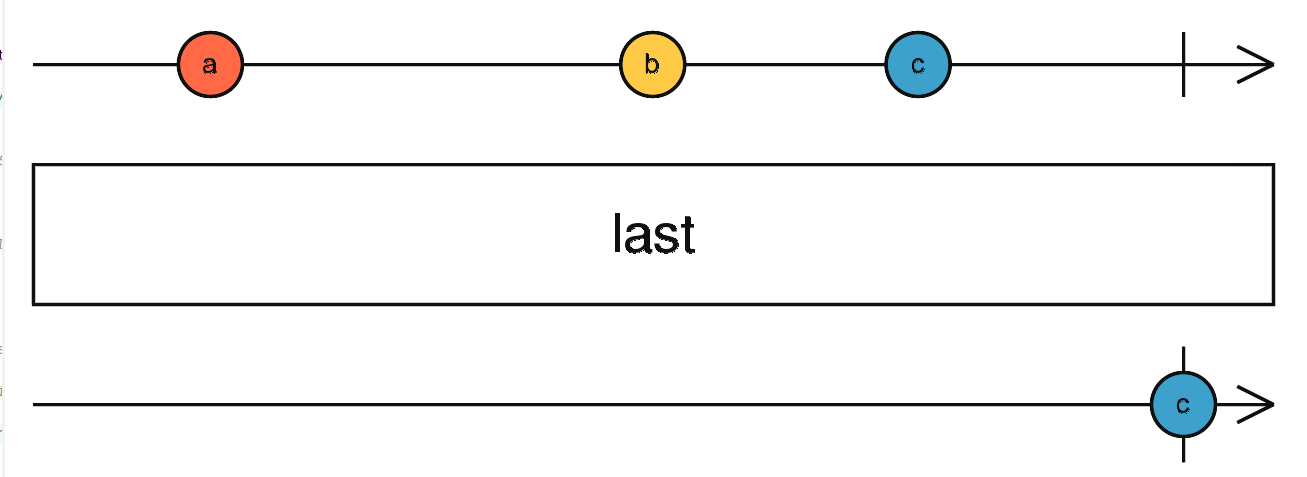
## First: emit giá trị đầu tiên của 1 observable rồi complete. First() sẽ throw EmptyError nếu như observable tự complete trước khi 1 giá trị nào (ví dụ như empty chẳng hạn, là 1 observable rỗng, hoặc of() không nhận vào giá trị nào) \_ kiểu observable ko có giá trị nào để emit cả.

Ngoài ra first() còn nhận vào 2 tham số optional: funciton(predicate) và defaultValue. Nếu như ta truyền vào predicate thì first() sẽ throw Error nếu như observable đã complete mà chưa có giá trị nào thảo được điều kiện của predicate. Nhưng nếu bạn vẫn muốn truyền vào predicate mà không muốn có error thì truyền thêm vào defaultValue



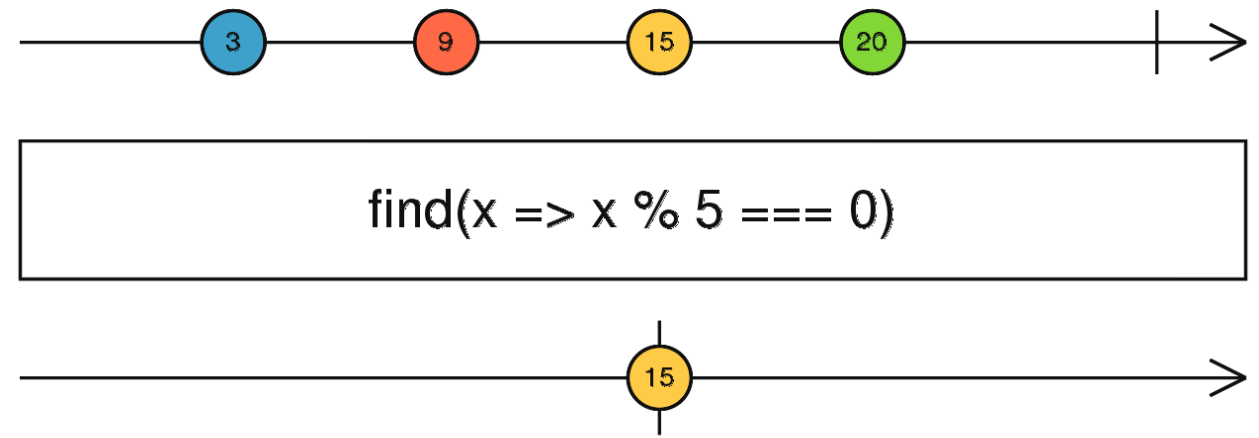
## Last(): ngược lại với first() thì last() sẽ emit giá trị cuối cùng của observable trước khi observable này complete. Tất cả behaviors mà first() có thì last() cũng có. Nghĩa là:

* 1. Throw emptyError nếu như observable tự complete trước khi emit bất kì giá trị nào
  2. Cũng nhận vào 2 tham số optional: predicate và defaultValue
  3. Throw error nếu như chỉ có predicate mà không có giá trị nào thỏa điều kiện
  4. Emit defaultValue nếu như có predicate và defaultValue và không có giá trị nào thảo điều kiện



## Find()

* Giống first nhưng bắt buộc phải truyền vào function, trả về giá trị đầu tiên thỏa điều kiện rồi complete nhưng khác với first ở chỗ nếu điều kiện không thỏa cái nào hết sẽ trả về underfined mà không có lỗi

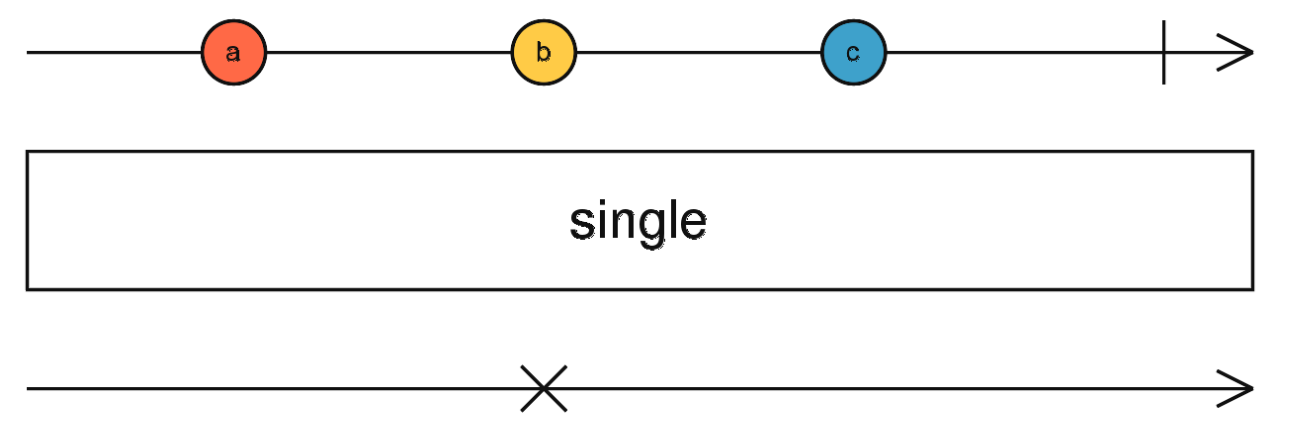


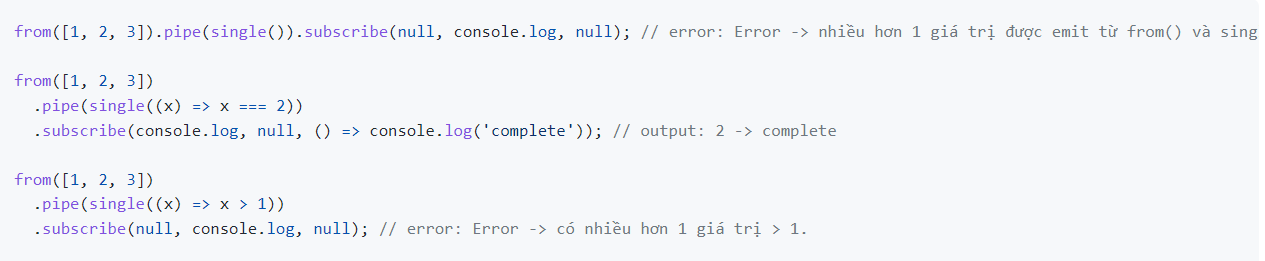
Ví dụ:



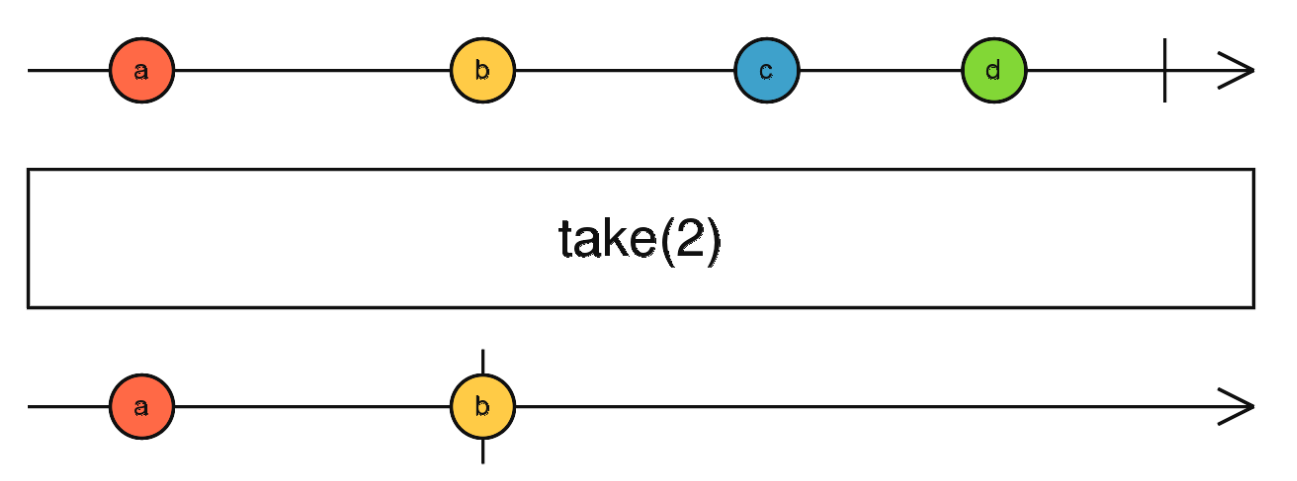
## Single()

* Khá là nghiêm ngặc (không cần truyền vô function), nhưng nếu không truyền vô mà stream emit có nhiều hơn 1 giá trị thì nó sẽ bị lỗi
* Chỉ được sử dụng khi có duy nhất 1 element thỏa điều kiện
* Nó nghiệm ngặc hơn find (find nếu tìm nhiều hơn sẽ trả về cái đầu tiên, single sẽ trả về lỗi)

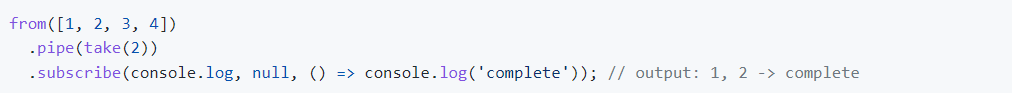




## Take()

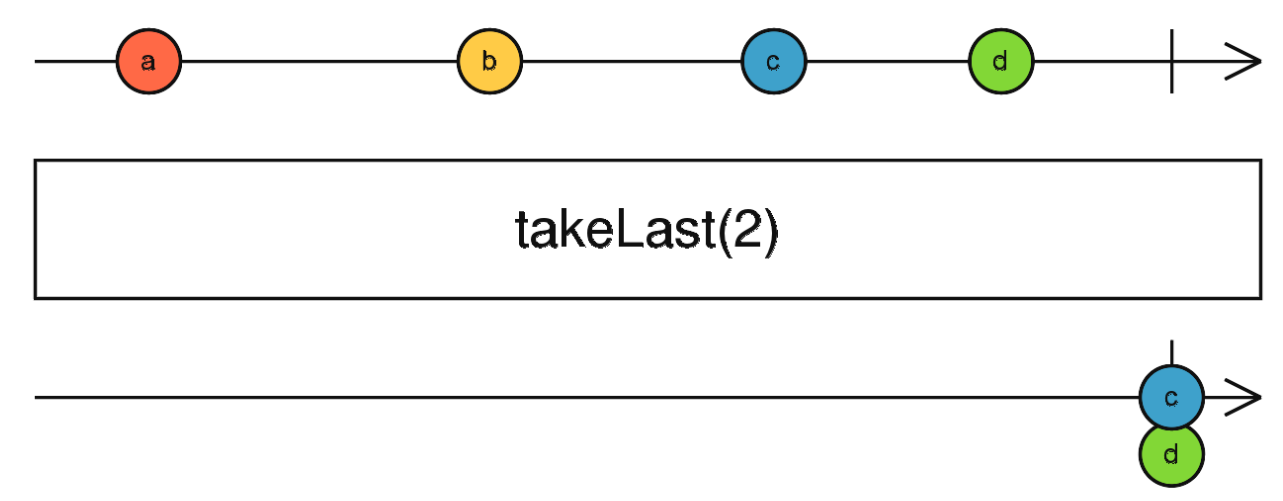


* Nhận vào tham số là số giá trị muốn được emit



Chú ý: take(1) = first() tuy nhiên nếu observable ko có emit giá trị nào thì nó vẫn ko bị throw error giống như first

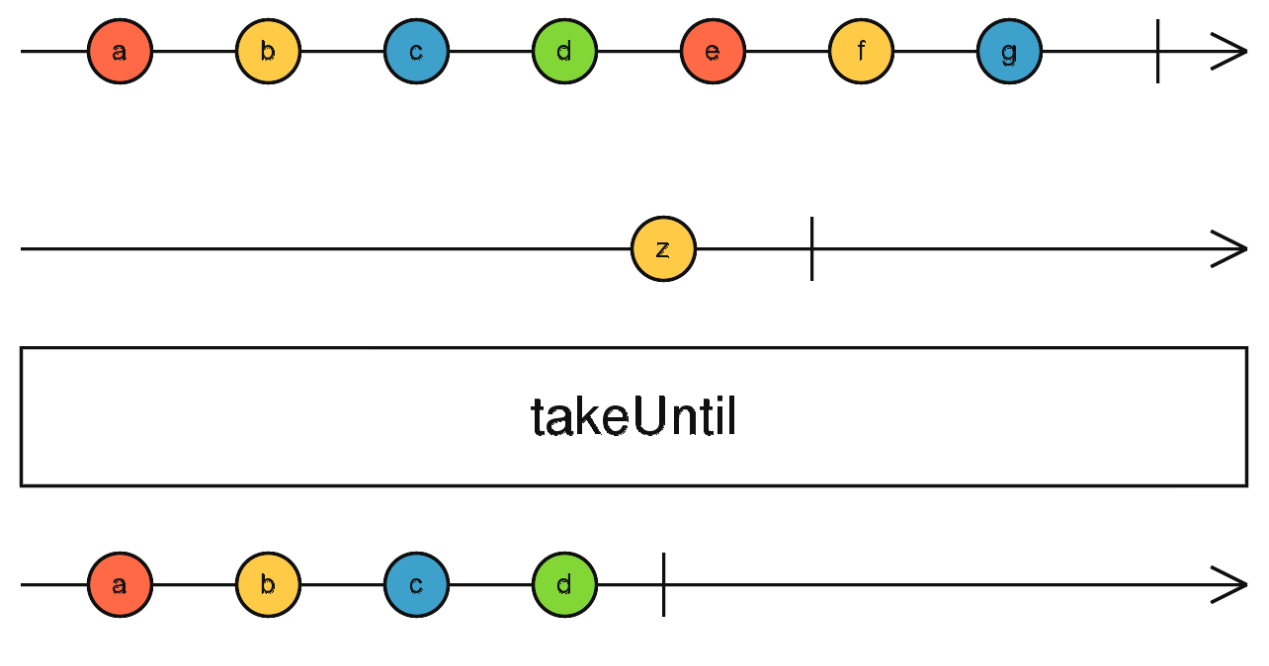
## takeLast()



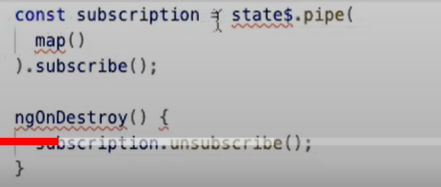
* nó sẽ đợi cho đến khi observable nó complete thì nó mới emit ra số giá trị cuối cho chúng ta

## takeUntil()

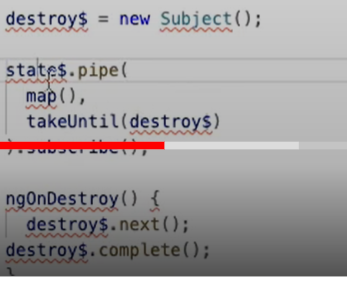
* truyền vào observable (cái nào mà emit được giá trị á) – notifier
* takeUtil() sẽ emit giá trị của observable gốc cho tới khi notifier emit

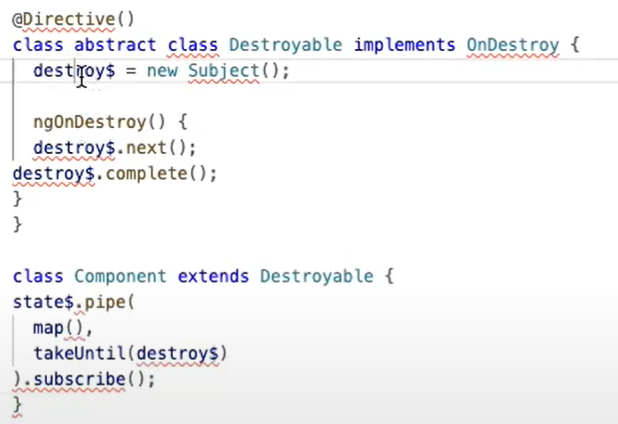
Trong angular:



Thay vì như trên ta dùng takeUntil()



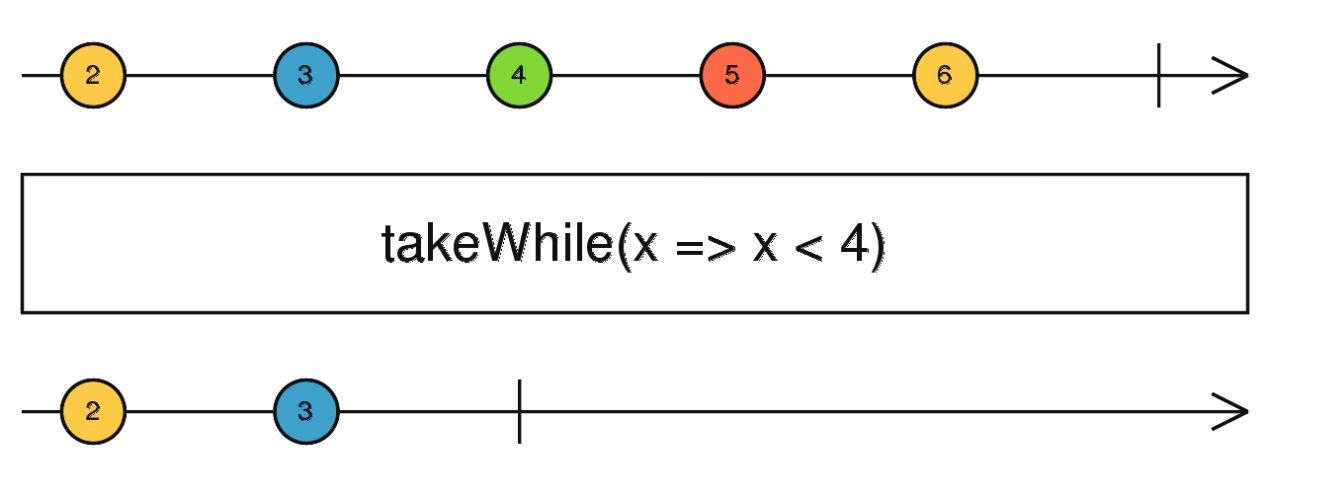
Ý muốn nói ta muốn lấy giá trị này cho đến khi destroy$ nó emit và destroy$ này emit khi ngOnDestroy chạy khi mà component này được bị destroy(unmount)



Đó là cách sử dụng trong angular

## takeWhile() (take trong khi thỏa điều kiện)

* nhận vào một điều kiện nếu điều kiện trả về true thì nó sẽ emit, nếu đều kiện trả về false thì sẽ complete

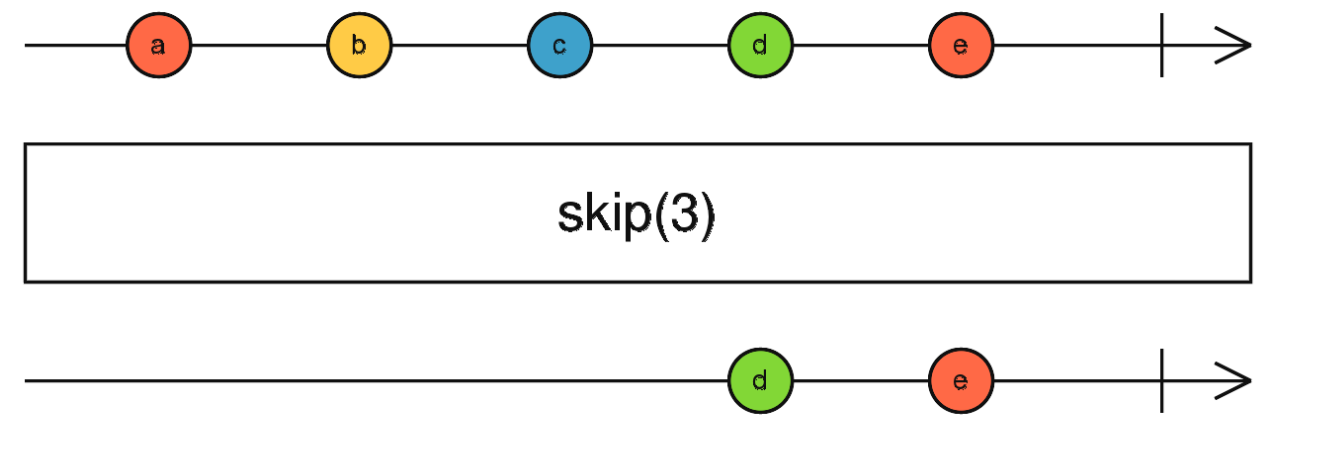


* cũng được dùng để unsubcribe một observable khi mà component destroy
* Khi nào ta sử dụng takewhile? Khi mà điều kiện dừng nó phụ thuộc vào chính giá trị của observable đó emit ra và nó có thể stop 1 cái giá trị countdown xuống 0 chẳng hạn. (khi mà dùng chung với switch, map.. nối thêm operator khác đằng sau thì sẽ có unexpected behavior)
* Dùng takeUntil() khi mà có tín hiệu từ bên ngoài

Ví dụ cái timer() là một observable khác cái observable interval này và không liên quan lúc đó nên sử dụng takeUntil

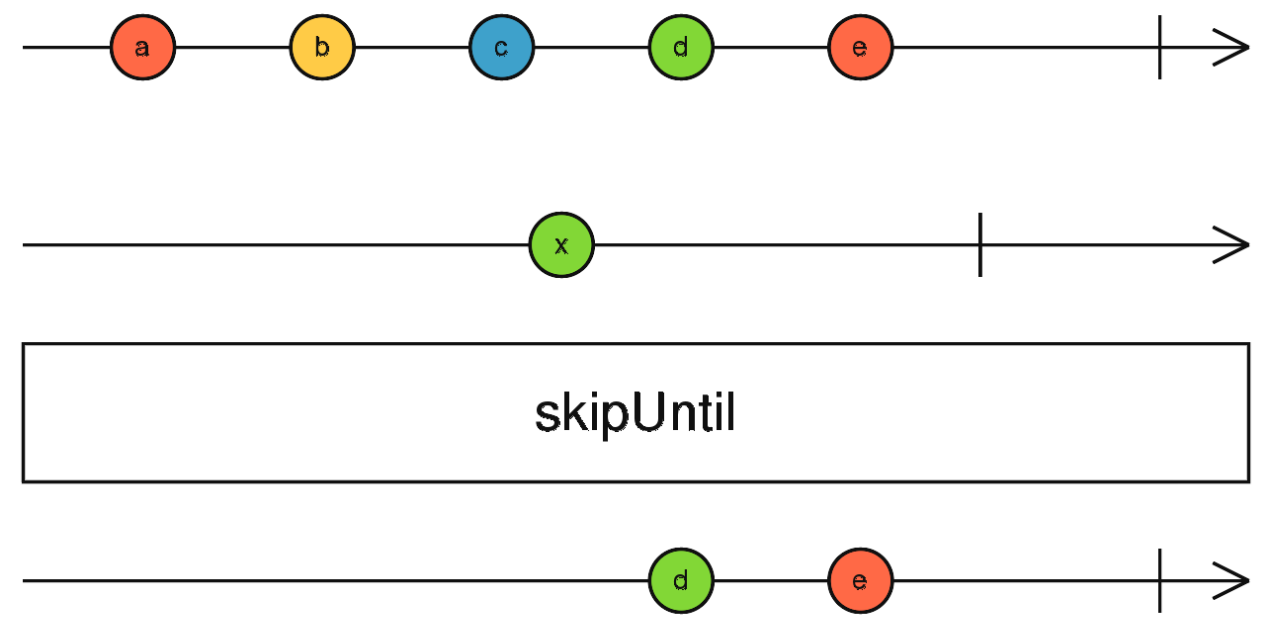


## skip()



* nhận vào số giá trị sẽ được bỏ qua

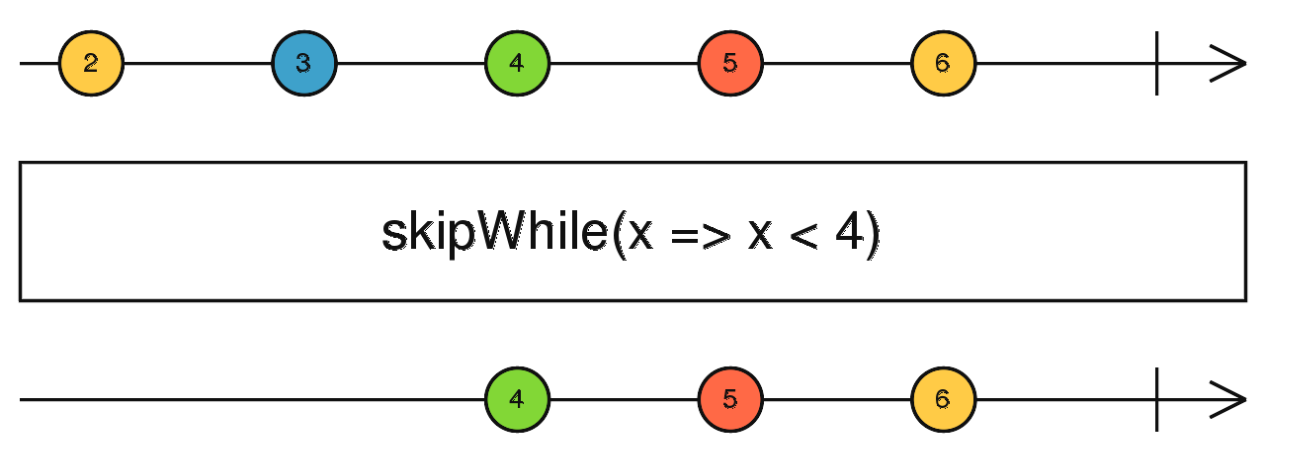
## skipUntil(): emit cái observable gốc bỏ qua giá trị cho tới khi thằng observable trong nó emit



Ví dụ: 

Timer chạy 6s thì nó bắt đầu emit

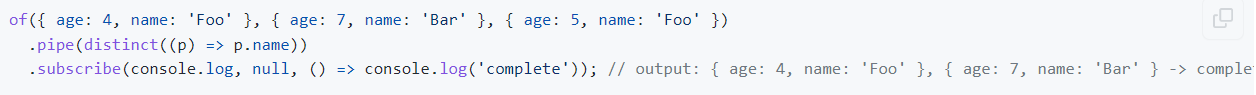
## skipWhile(): thỏa mãn while thì bỏ qua thôi



## distinct(): bỏ qua các giá trị đã emit ròi

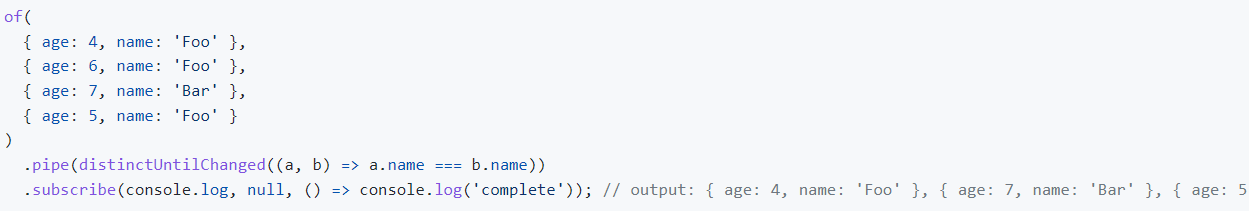


* distinct(): hoặc nhận vào function với mục tiêu so sánh lấy ra đối tượng ko trùng lặp về một properties nào đó

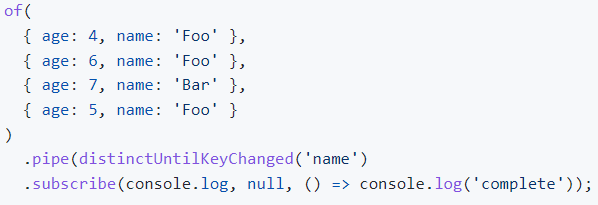


## distinctUntilChanged() – dùng nhiều trong state management vì không muốn có emit khi mà state không thay đổi

* khác với distinct ở chỗ distinct là so sánh toàn bộ giá trị trong 1 observable phát ra còn distinctUntilChanged chỉ so sánh 2 giá trị gần nhau nhất thôi (là thằng được emit sau nó sẽ so sánh với thằng được emit trước đó mà distinct
* nó vẫn có thể nhận vào 1 cái hàm để so sánh (nó sẽ dùng === để so sánh 2 giá trị)



## distinctuntilKeyChanged(): nó là distinctUntilChanged với keySelector



\*\*\*phía dưới là 8 operator giảm số lần emit

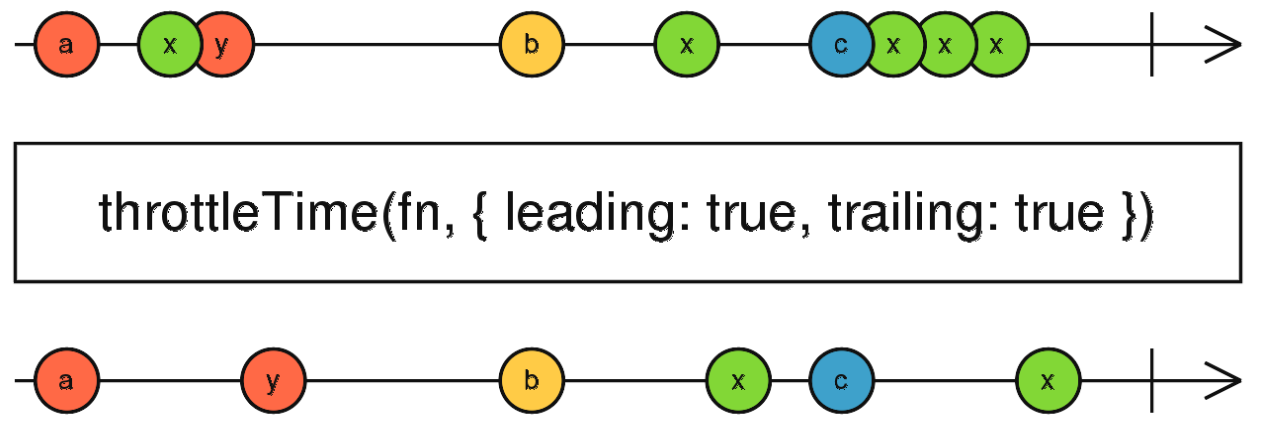
Có 2 loại 1 cái có time và một cái không có time và nó hoạt động như nhau chỉ khác

Cái có time thì truyền vào time

Cái không time thì truyền vào một observable

## throttle()/throttleTime()

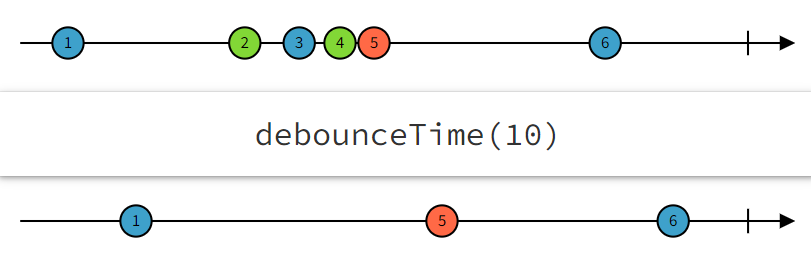
* lấy giá trị emit đầu tiên sau đó time chạy



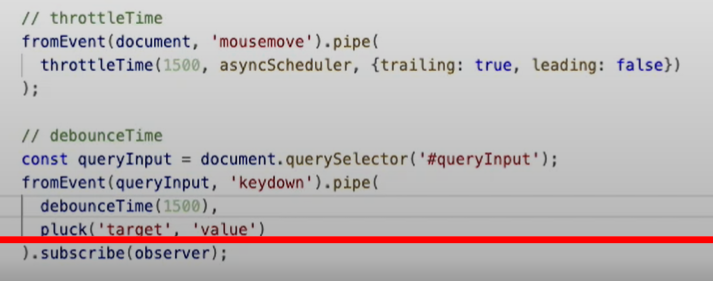


## debounce()/debounceTime()

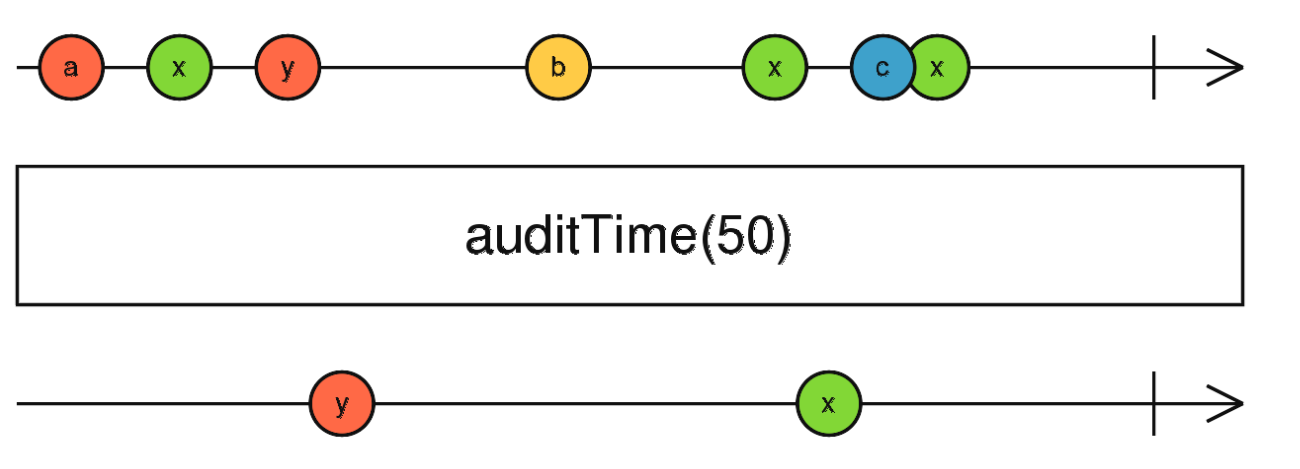
* sẽ không emit cho tới khi mà không có giá trị nào được emit trong khoảng thời gian timer chạy (được dùng trong sự kiện gõ vào input lấy giá trị để đi làm gì đó)







## audit()/auditTime()

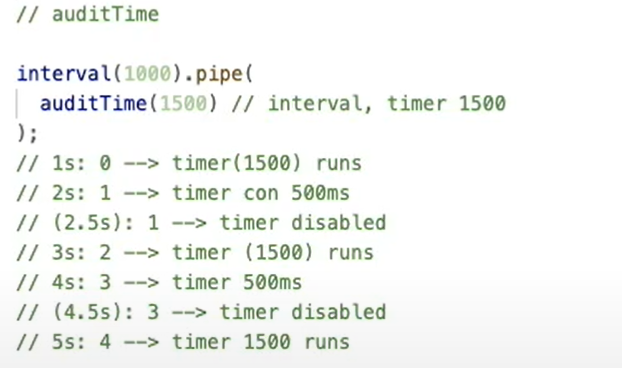


* nhận vào số miliseconds



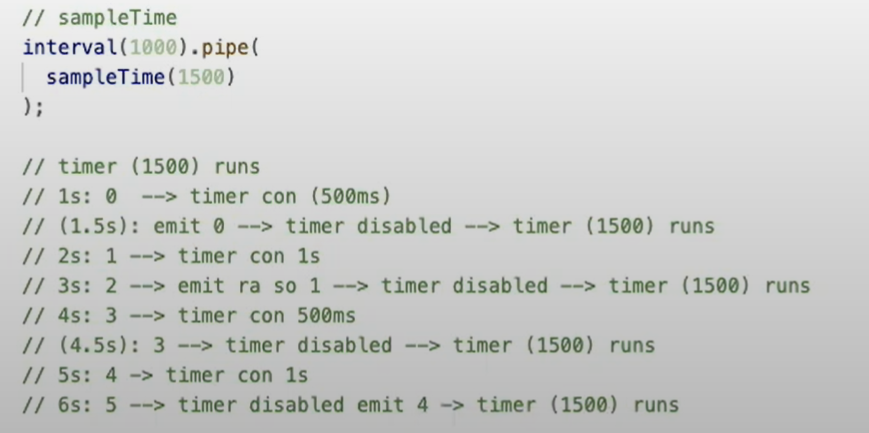
Nghĩa là trong interval của của auditTime có 1 timer 1s: khi timer này chạy thì tất cả giá trị emit từ observable này sẽ bị bỏ qua cho tới khi timer này chạy xong thì auditTime sẽ emit giá trị cuối cùng mà thằng obsevable này nó emit

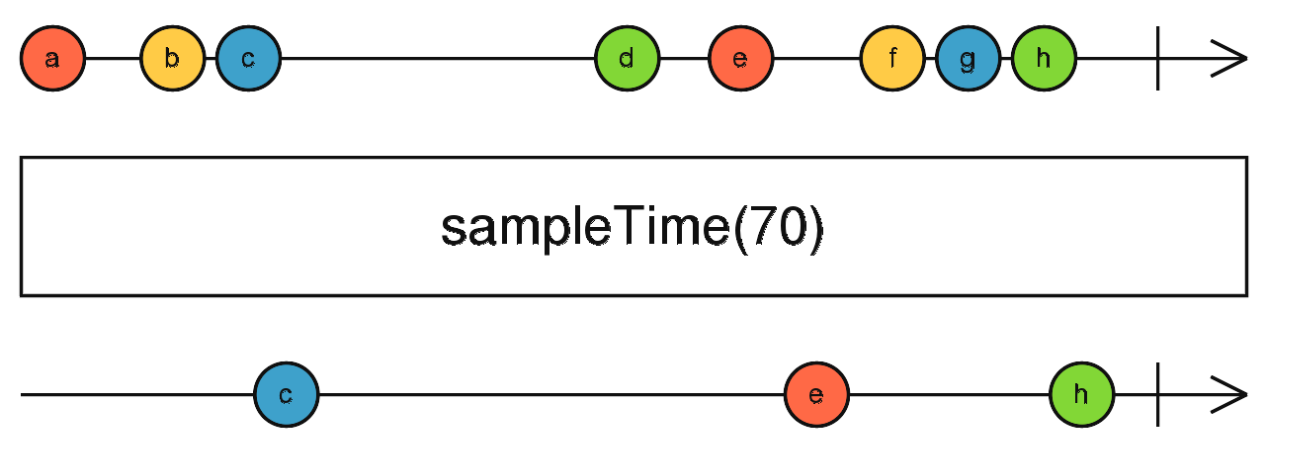
Chú ý auditTime chạy khi mà observable emit và nó sẽ bỏ giá trị được emit và lưu lại giá trị cuối cùng



## sample()/samepleTime()

* samepleTime khác auditTime ở chỗ timer này sẽ được chạy lặp đi lặp lại liên tục. Khi mà ta subscribe thì nó bắt đầu chạy rồi





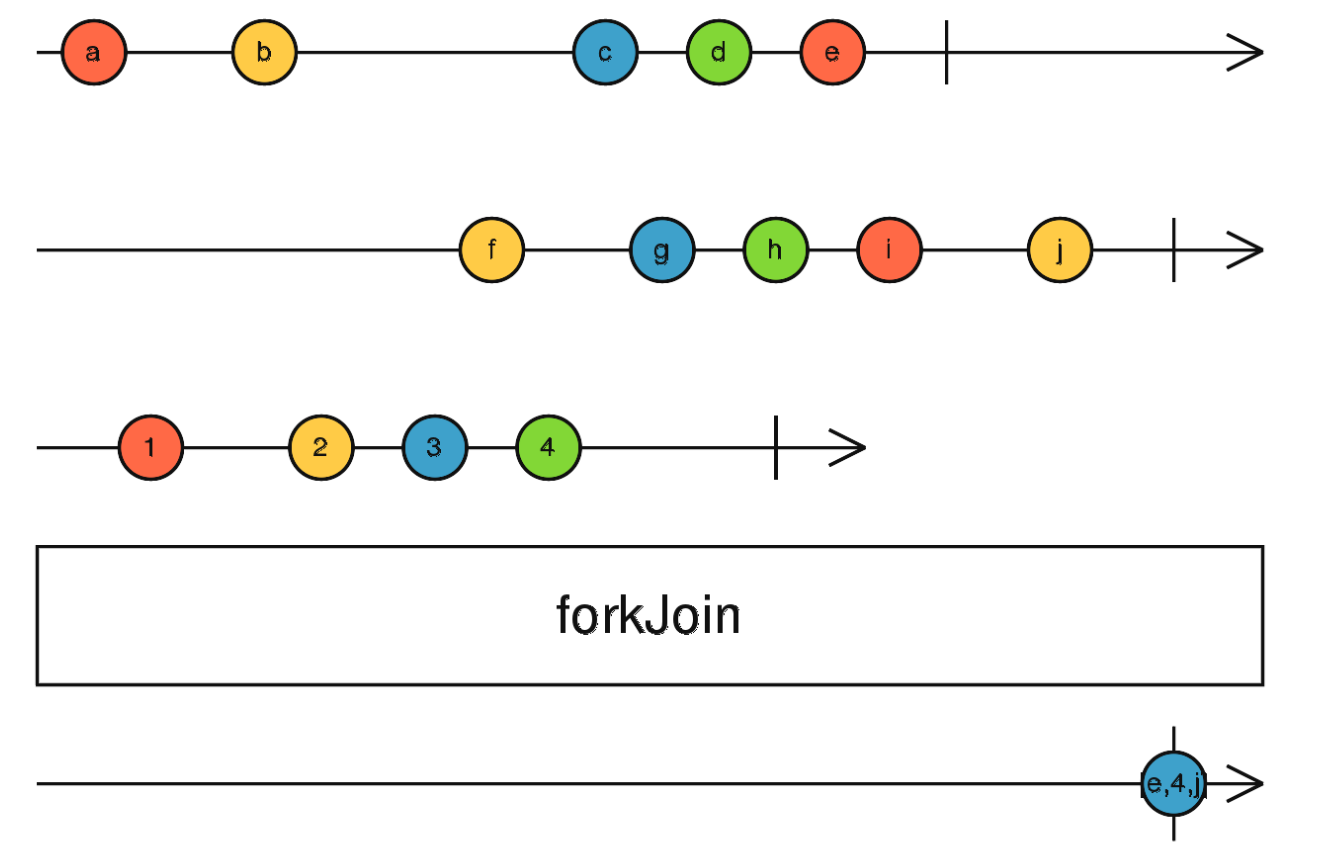
# Combination operators

Kết hợp nhiều stream khác nhau thành 1 stream

Cho phép kết hợp nhiều observable lại với nhau.

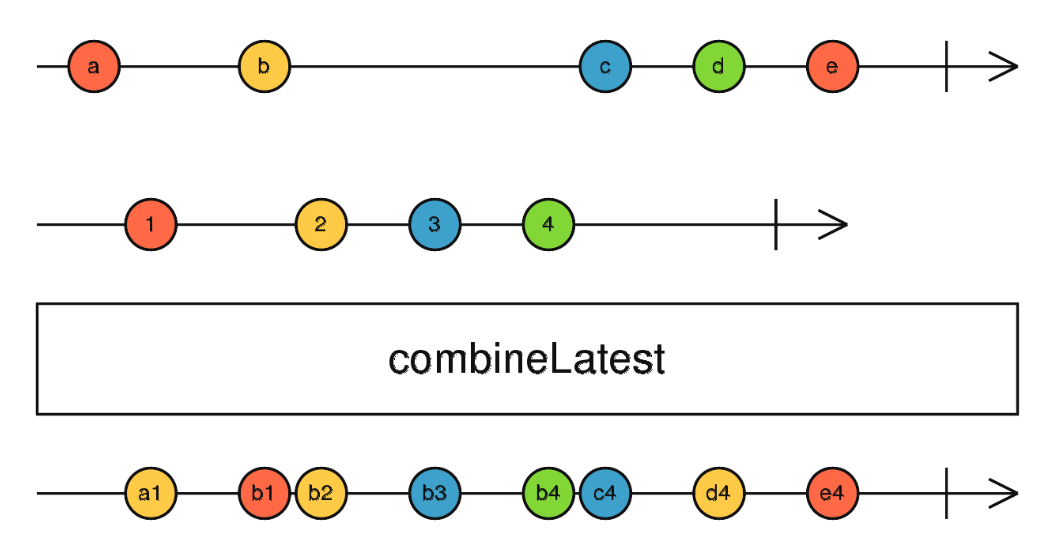
## forkJoin(): gộp những stream này lại chạy lần lượt khi mà những stream này complete thì forkjoin mới emit giá trị cuối cùng của tất cả các stream ra

* và các stream bên trong phải luôn được emit, nhận vào array gồm 3 giá trị theo tứ tự stream truyền vào và nên đặt các stream trong mảng, hoặc nhận vào object từ các stream
* thường được dùng khi mình muốn lấy dữ liệu từ 3 stream khác nhau và nó không phụ thuộc vào nhau ví dụ dropdown/select

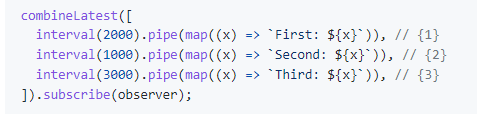


## Combinelatest()

* Hoạt động tương tự forkJoin nhưng có một điểm khác biệt hoàn toàn đó là forkjoin thì phải complete hết mới chạy còn combineLatest thì không cần. Conbinelatest để combine các stream mà chưa chắc đã complete



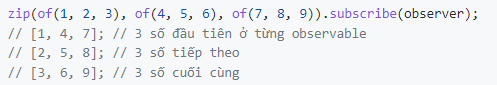
* Nhận vào tham số là 1 array<observable>
* Và nó emit theo giá trị stream phát ra muộn nhất kết hợp với lần lượt các stream còn lại

ví dụ

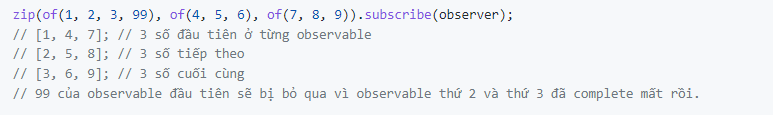
* combineLatest sẽ complete khi tất cả các stream complete, sẽ không bao giờ complete nếu như 1 trong các stream không complete
* combineLatest sẽ throw ra lỗi nếu như 1 trong số các stream throw ra error và các giá trị của các observable khác đã emit sẽ bị nuốt (hehavior này giống forkJoin)
* thường dùng trong trường hợp:
  + Dùng trong việc combineState khi dùng service trong angular , vì tính chất long-lived không complete sau mỗi lần emit, combineLatest là sự lựa chọn tốt cho việc combine các state trong Service và kết hợp với AsyncPipe để dùng trong template

## Zip()

* Nhận vào tham số là observable hoặc là function để tượng trưng cho các obsevable con được truyền vào lần lượt. zip() sẽ gom tất cả các giá trị được emit bởi các observable con theo cặp

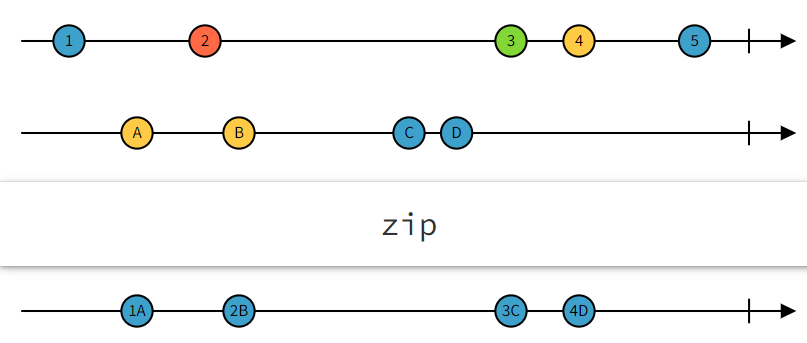
Ví dụ: 

* Zip() sẽ complete khi 1 trong các observable con complete
* Zip() sẽ throw error nếu 1 trong các observable con throw error
* Nếu tham số cuối cùng của zip là 1 function thì zip sẽ coi nó như 1 project function



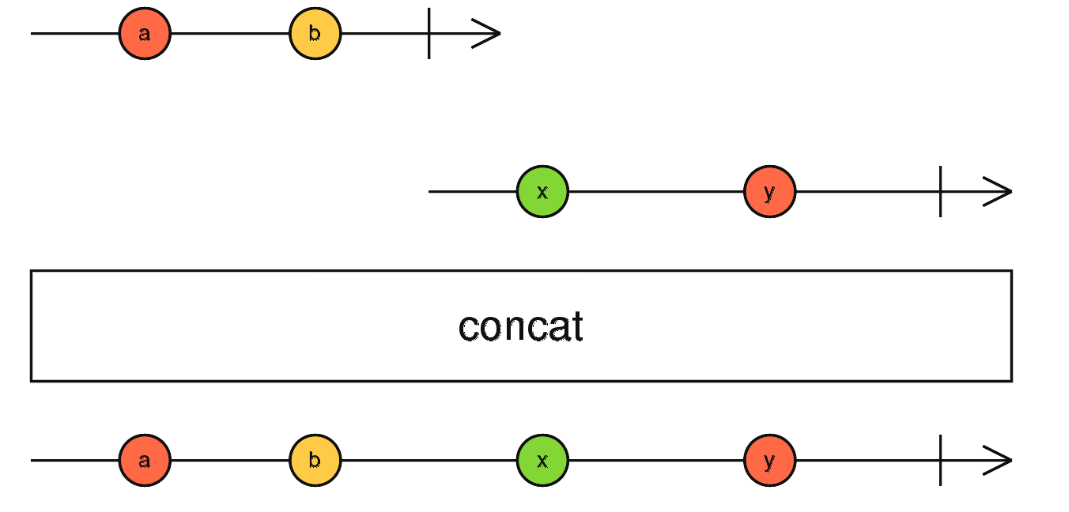
Trường hợp sử dụng:

* Zip cặp event
* Giả dụ ta có 1 ds toàn id, 1 ds toàn name, 1ds toàn age. Lúc đó mình sẽ dùng zip để gôm lại cả 2 stream đó thành 1 stream chứa object user hoàn chỉnh
* Zip sẽ complete khi 1 trong 3 complete



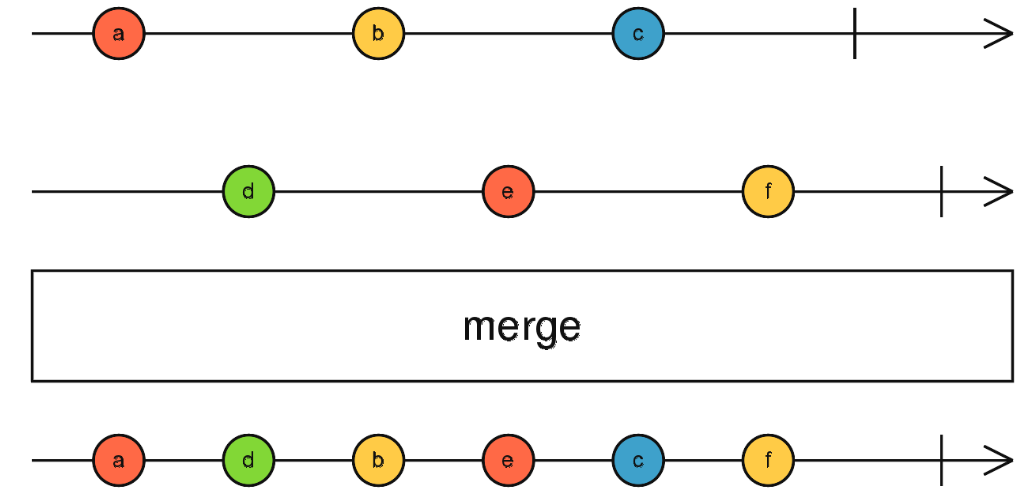
## Concat()

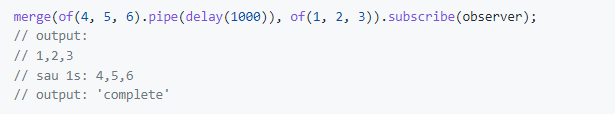
* Operators dùng để kết hợp stream lại với nhau nhưng những stream này sẽ chạy theo điều kiện là stream đầu tiên kết thúc sau đó stream thứ hai mới chạy
* Được dùng trong các trường hợp thứ tự chạy quan trọng: như đèn giao thông, số báo danh



## Merge()

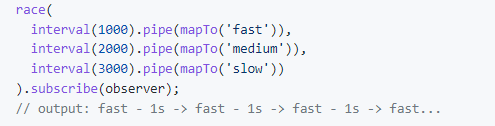
* Nhận vào tham số là observable hoặc number, khác với concat merge không quan tâm đến thứ tự các children observable emit nên là không bị giới hạn bởi các observable con phải complete thì observable kế tiếp mới được subscribe, mặc định là các observable con trong merge sẽ được chạy song song nếu ta muốn bao nhiêu observable được chạy song song thì ta thêm tham số number ở cuối
* Merge() sẽ subscribe vào tất cả các observable con (tùy thuộc vào số concurrent chạy song song mà ta truyền) và sẽ:
  + Emit giá trị mà bất cứ observable nào emit
  + Throw error nếu một trong các observable con throw error
  + Complete chỉ khi tất cả observable con complete
* Trong angular, được sử dụng trong FormGroup và ta muốn lắng nghe từng FormControl.valueChanges để thực hiện nghiệp vụ nào đó





## Race():

* Nhận tham số giống như merge và concat
* Race() sẽ emit giá trị của observable nào emit đầu tiên (nhanh nhất), sau đó lặp lại cho đến khi 1 trong các observables complete
* Race() sẽ error ngay lập tức nếu observable nhanh nhất throwError

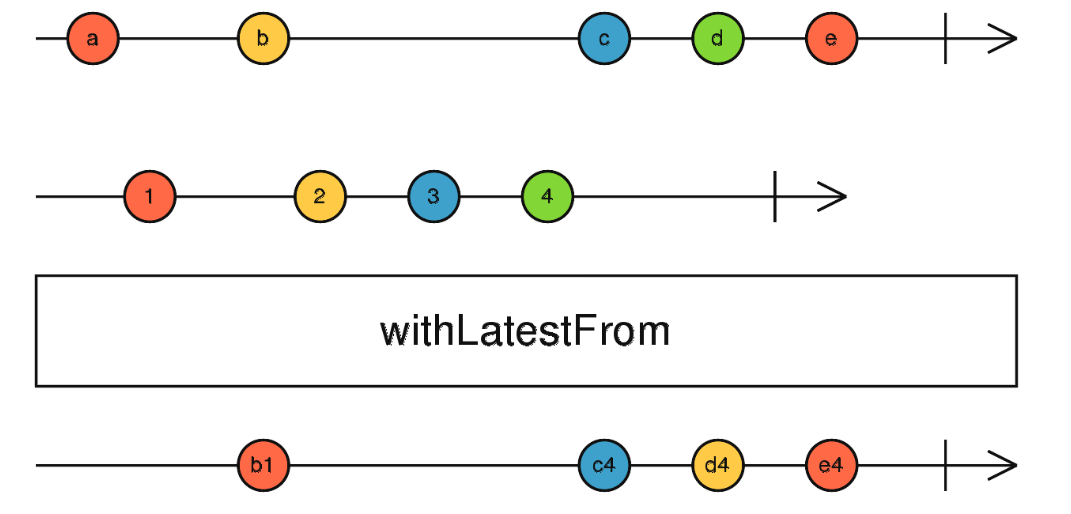


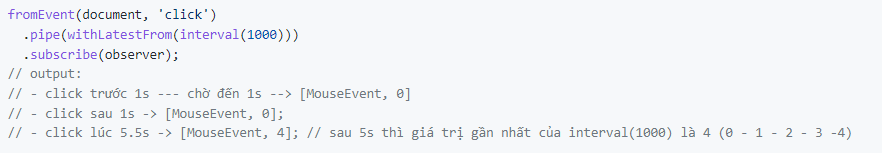
\*\*Chú ý: forkJoin, combinelatest, zip, concat, merge, race nó đều là những funcition được import từ rxjs chứ ko phải là một pipeable operators

Tiếp theo sẽ tìm hiểu về pipable oprators

## withLastestFrom()

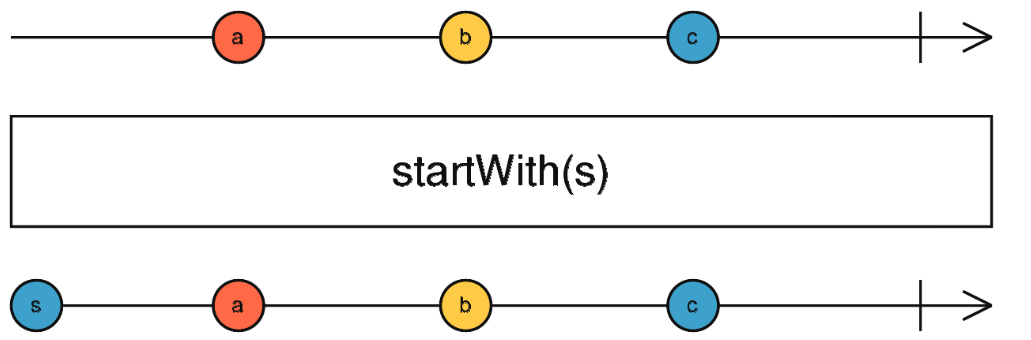
* Nhận vào tham số là 1 observable. withlatestFrom sẽ gộp giá trị emit của outer observable với giá trị gần nhất của tham số observable thành 1 array rồi emit array này
* lấy giá trị cuối cùng của một stream khác khi mà stream ở bên ngoài nó emit
* sử dụng khi nào? Khi muốn xử lý nghiệp vụ gì đó mà tại thời điểm nghiệp vụ đó xảy ra mình muốn lấy giá trị cuối cùng từ một chỗ khác (muốn combine 2 stream nhưng ko bị phụ thuộc vào 1 trong 2 stream)



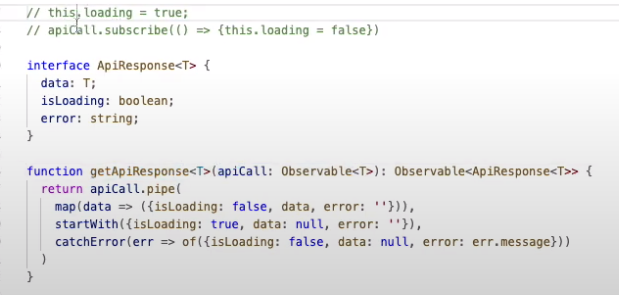


## StartWith()

* Nhận vào một list các tham số. sẽ làm cho cả observable emit giá trị của startwith trước rồi mới emit đến giá trị của outer observable. Startwith sẽ emit ngay lập tức mà không cần phụ thuộc vào outer observable có hay chưa emit
* Muốn start stream với một giá trị nào đó

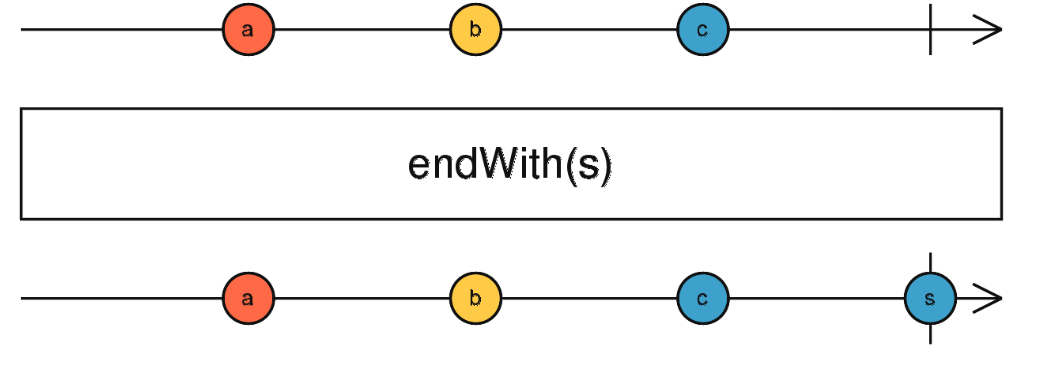


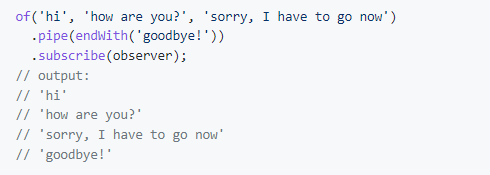
* Sử dụng:



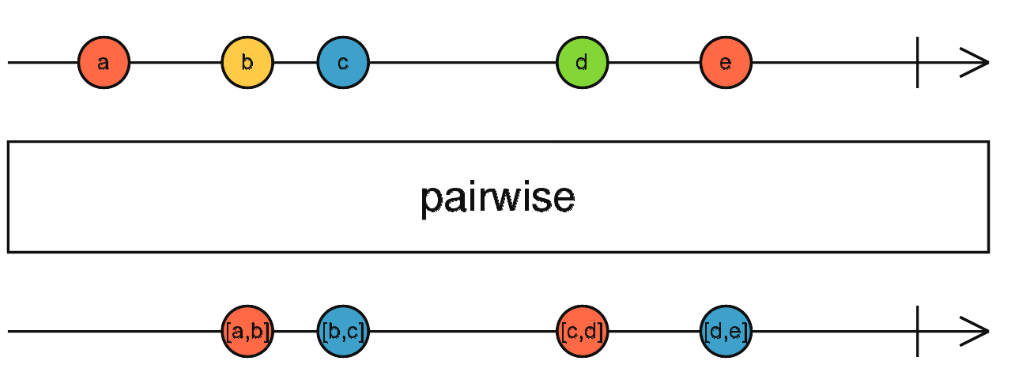
## EndWith()

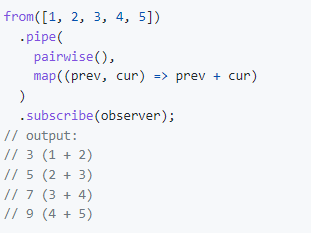
* Ngược lại với startwith thì outer observable complete thì endWith mới chạy được





## Pairwise()



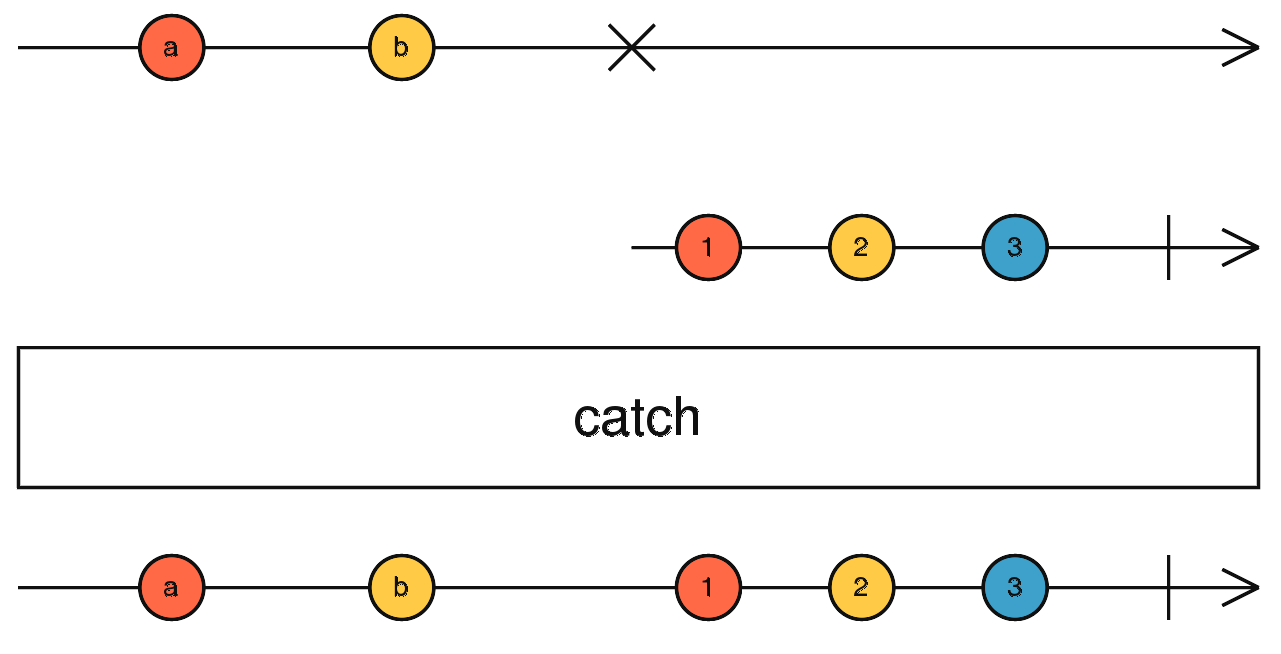


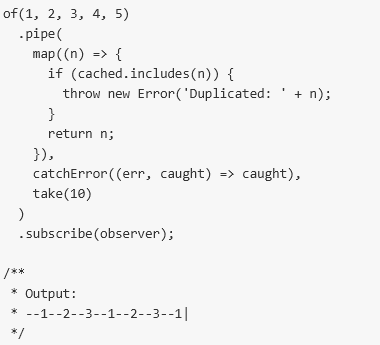
# Error handling and conditional operators

Mỗi observable có thể gửi về các message cho next, error, complete. Và nếu như error được phát sinh thì observable sẽ dừng lại vậy nên ta cần dùng operators để xử lý lỗi và một số khác để làm việc với các loại điều kiện (Error handling and conditinal operators)

## catchError:

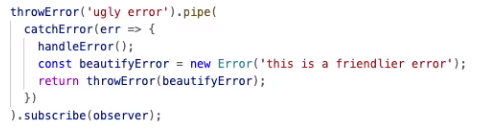
* truyền vào callback nhận vào 2 tham số: error (sẽ throw), observable throw error này (caught) trong hàm đó 1 là ta throw lỗi mặc định hai là ta rethrow cho handler kế tiếp làm việc và callback return về 1 observable



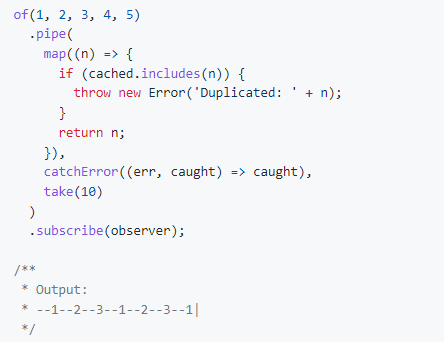
(ví dụ)



Nếu ta không catchError thì observer.error sẽ là nơi đón error, nhưng vì ta trả về một observable với giá trị là err nên observer.next sẽ là nơi handle lỗi này



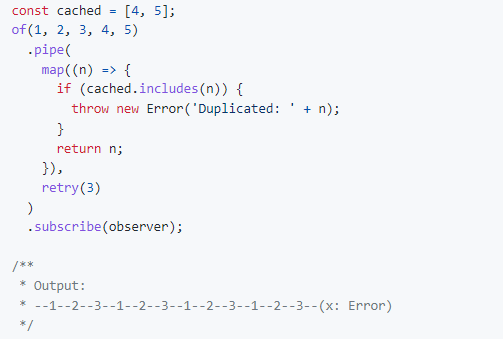
Trường hợp rethrow



Trường hợp 3: sử dụng caught để retry caught, ta thấy observable ở ngoài là 1 observable throw error ngay lập tức, liên tục lúc nào cũng throw error nếu ta retry thì nó vẫn chạy vào catcherror và bị infinited loop

## retry

* có thể truyền vào số lần mình muốn retry



* Cũng giống khi dùng caught và take và có thể điều khiển số lần ta muốn retry
* Nhưng khác nhau là take nó sẽ tính trên số lần emit của of còn retry là tính luôn toàn bộ giá trị có trong observable
* Retry when (nhận vào modifer: observable hoặc subject)

1. defaultfEmpty/throwEmpty
2. every
3. iif

Higher Order Observables(HOOs) and Utility Operators

* HOOs cũng là những transformation operators nhưng thay vì transform thành value mới thì chúng sẽ trả về observable mới để chúng ta có thể subscibe vào observable mới này để lấy giá trị mới
* HOOs nhận vào Outer Observable (Source) và sẽ trả về Inner Observable (Distination) khác.

1. switchMap()
2. mergeMap()
3. concatMap()
4. exhaustMap()
5. switch/concat/mergeMapTo()
6. partition()

Utility Operators

1. tab()
2. delay()/delayWhen()
3. finalize()
4. repeat()
5. timeInterval()
6. timeout()
7. timeoutWith()
8. toPromise()

Subject and Multicasting

1. Subject
2. BehaviorSubject
3. ReplaySubject
4. AsyncSubject
5. Subject completion
6. Multicasting
7. Multicast
8. refCount
9. SubjectFactory
10. Publish
11. Share
12. ShareReplay