RxJs

It provides a way to process different kinds of streams with the same API

For example

1. Events firing by click of a button: infinite stream that is usually handled by click listener
2. An array: a finite stream that is handled by a for loop

# Observables

They can be synchronous or asynchronous

The building block of RxJs

# Observers

They are objects that implement the Observer interface which has 3 methods:  
next()  
error()  
complete()

Subscribers are also Observers because they implement the same interface

# How to subscribe to an observable:

1. Create a subscribe method and pass it as a parameter to Observable constructor
2. Use Observable.create (deprecated)
3. Using ‘of’ function

This takes a set of args and converts them to a stream so that each argument is fired separately

of("hello", 1, true, { book: allBooks[0] })

1. Using ‘from’ function

Takes an observable or a promise or an array(stream)

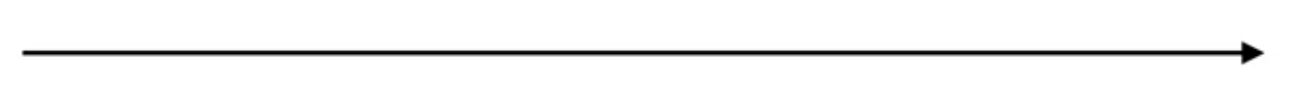
1. From ‘ajax’ request using ajax function

# Combining observables

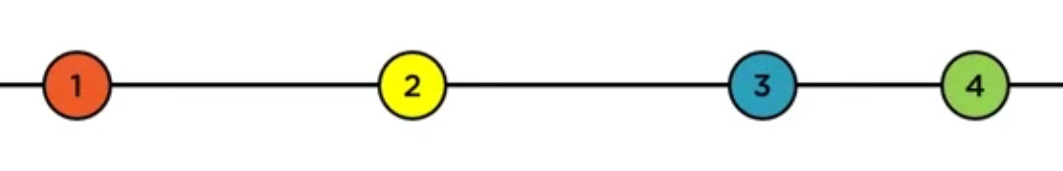
1. concat(): will produce a new observable consist of all values from first observable followed by all values from second observable

# Marble Diagram

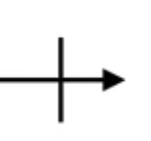
* The black horizontal arrow represents the source observable over time



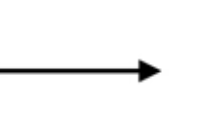
* The bubbles on the arrow represent the values which are produced over time by the observable



* The vertical line at the end of arrow represents the successful completion of observable



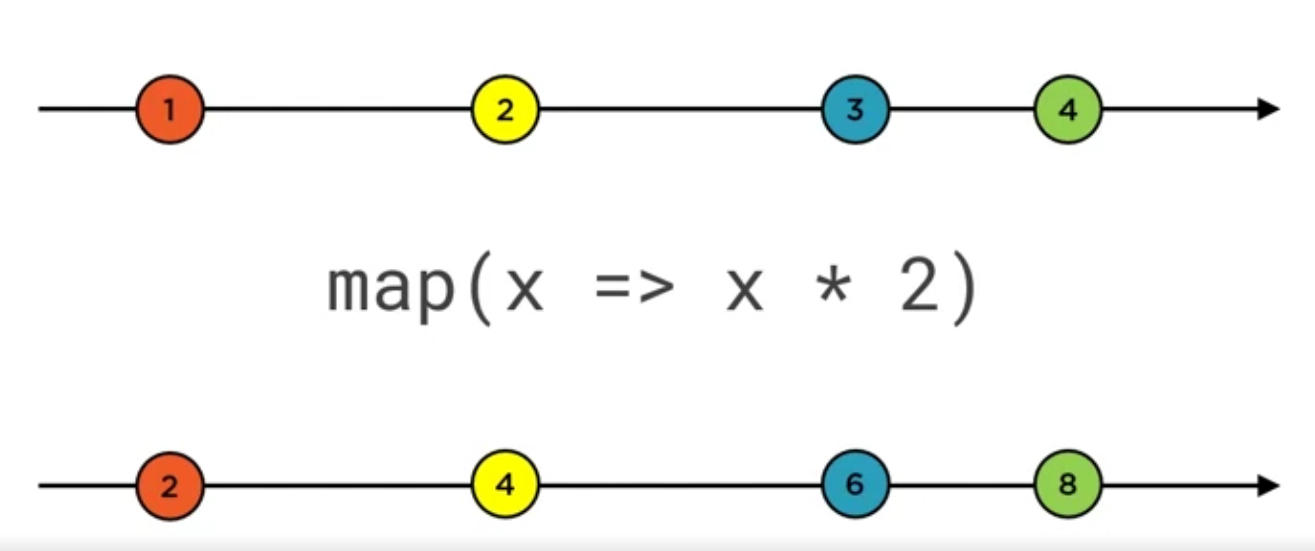
* Observable that will never complete



* Observable that failed to complete(has an error)



* The result of operator is represented in the bottom horizontal line



The colors associate the source value and result value of map operator

# Operators

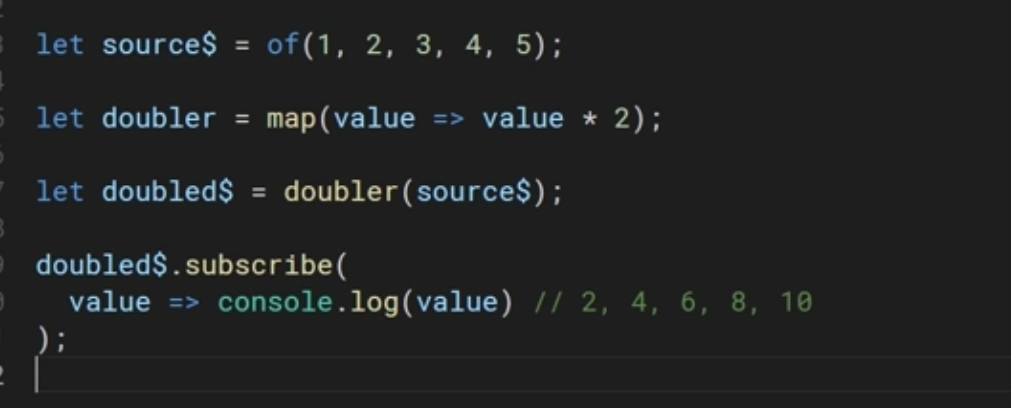
They add more functionalities to observables

It’s a function that returns a new function which returns an observable when called

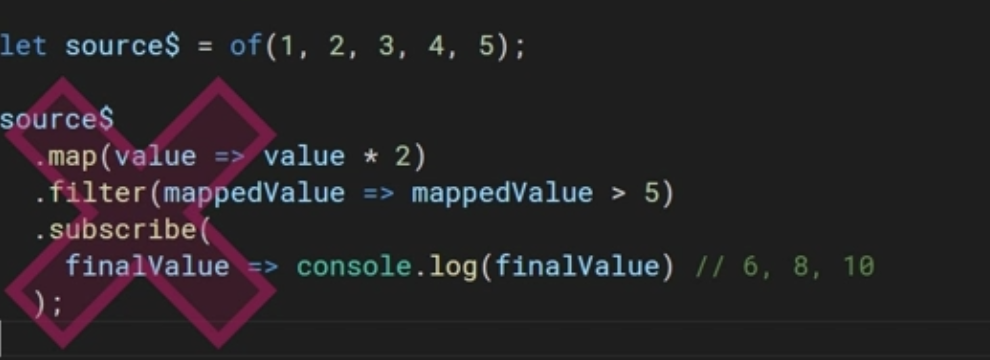
## Categories

* Transformation: the result will be different from the source
* Filtering: produces a subset of the source stream
* Combination: combine two or more observables in various ways
* Utility: how or when values are produced, without changing what values are produced
* Conditional: similar o filtering
* Aggregate: produce a single value as a result of all the stream for example: min, max, sum

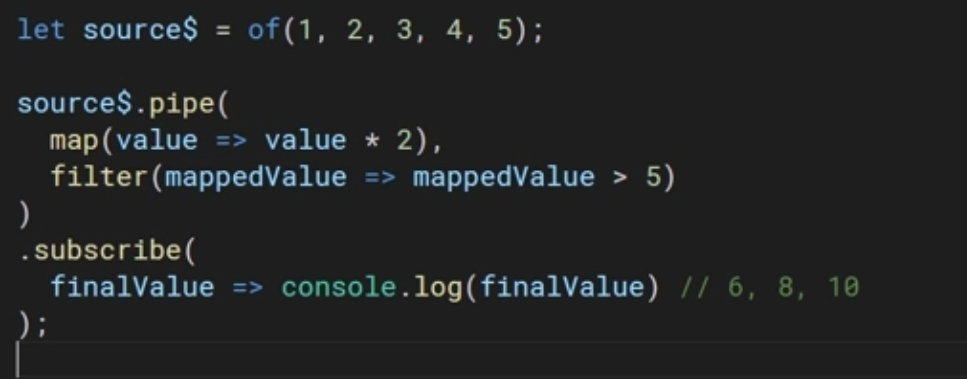
## Manual chanining:



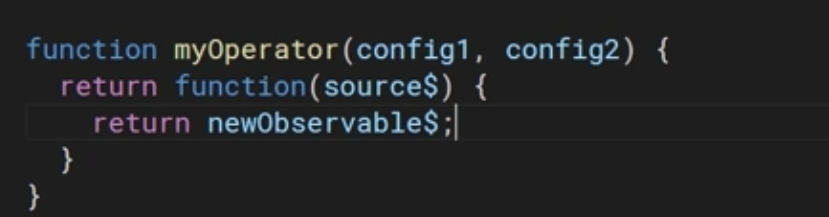
## Old method:



## New method after v5:



## Structure of operator:



It’s a function that could take configuration as parameters, and returns another function

The returned function takes the source observable and should return a new observable

## Custom operators

Why do we need it?

1. When we have a set of operators that we use together a lot, we can create a new one of the existing operators and use it
2. When we need new functionality that isn’t available in existing operators

## Notes:

Observable is only executed when there is a subscription to it, otherwise it wont fire.

# Subjects

They are both observables and observers

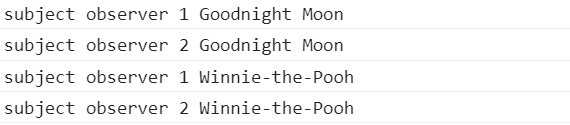
They are multicasted which means a single value is published to all subscribers at the same time before the next value is published

It contains an array of observers which it loops and pushes the single value to each of them

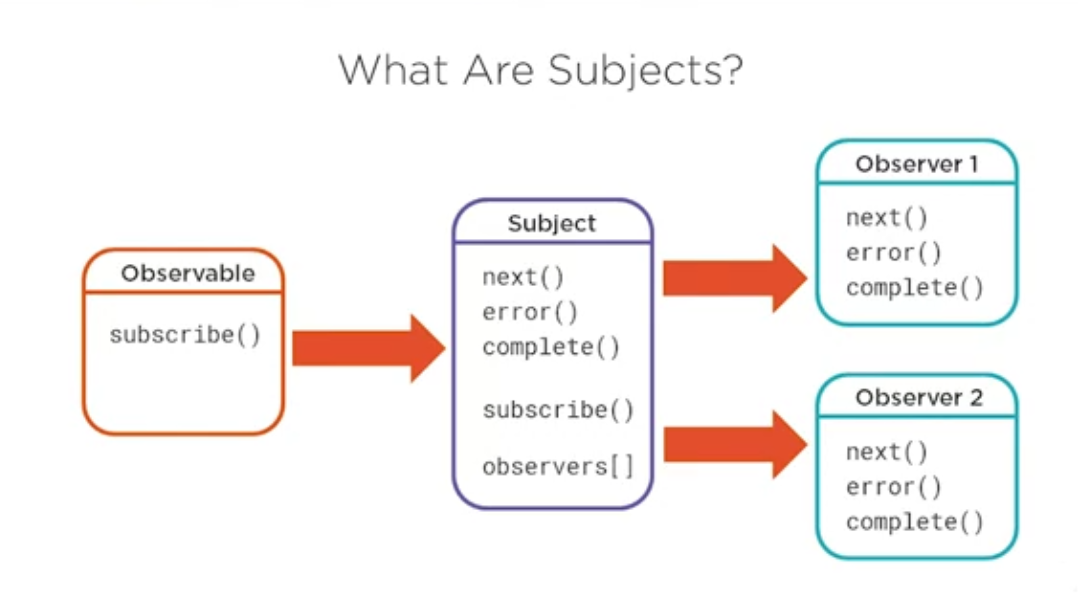
For example in observables all values are sent to the first subscriber first.



But in case of subjects each value is pushed to all observers



## Multicasting



A subject can sit between an observable and observers to multicast each single value to all observers simultaneously

Note: in this case make sure you subscribe to the observable before the observables subscribe to the subject

Because the execution of subject begins whether or not it has subscribers

If an observer subscribed after the subject has completed, it will get the completion message or get the error which stopped the execution

## Cold vs Hot observables

### Cold (Unicast)

The value producer is created inside the observable

For example: if it produced a series of number, the array of numbers will be created inside the observable.

One observer per execution, which starts when subscribe is called

### Hot (multicast)

They wrap a value producer which exist outside the observable

Multiple observers receive the same value over the same time

Example: fromEvents

## Multicase operators

### muticast()

It takes a subject as a parameter and returns a connectable observable which has a method called connect() to begin the execution

### refCount()

triggers the execution when number of observers is > 0

### publish()

similar to multicast but doesn’t require to pass a subject and creates one behind the scenes

### share()

similar to refCount but will re-subscribe to the source observable if the refCount goes to zero then increases again

so it re-triggers the execution of the source

## Types of subjects

### Async subject:

Publishes only the last value of an observable  
publishLast() operator

### Behavior subject

Emit an initial seed value if a subscriber has subscribed before the source emits a value. If the source has already emitted a value then it will publish it.  
publishBehavior() operator

### Replay subject

Emit multiple number of values to all observers. If an observer subscribed after all values were published, it will replay all these values and re-publish them to the last observer.  
publishReplay() operator

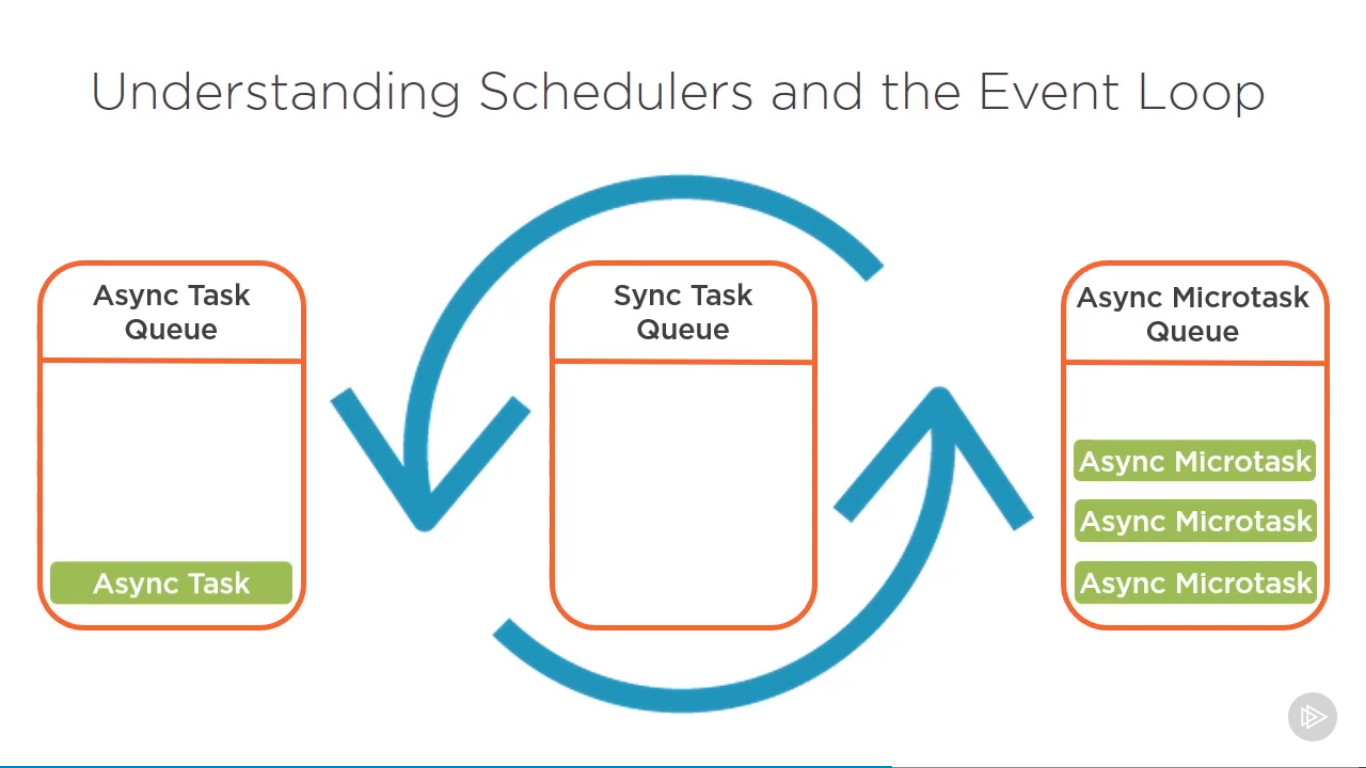
# Schedulers

Controls when an observable begins execution and when notifications are delivered to its observers

they are optional to pass

## Built in

* Queue scheduler: to execute observables synchronously
* Async scheduler: wont block js event loop while observable execute
* Asap scheduler: they are added to the micro task queue in event loop
* test scheduler: for unit testing observables

the async micro task queue has more priority than the async task queue and all of the micro tasks are added to the sync task queue at once, rather than waiting for next event loop to add the next microtask

# Testing

## Test schedulers

To test async code synchronously, in virtual time

only works with code that uses the asyncScheduler, doesnt work with asapScheduler or code that gets added to microTask queue

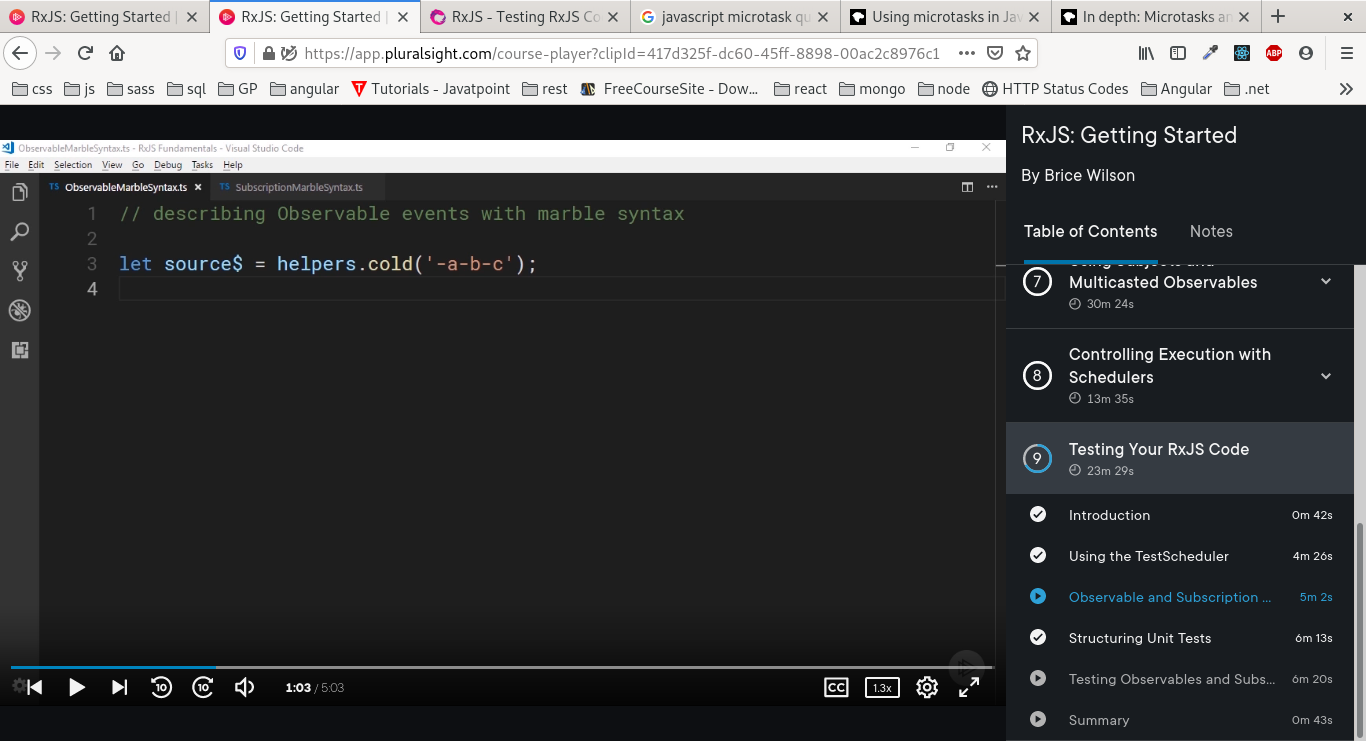
its constructor takes a function that gets used in equality tests; the function could be passed from the testing framework being used.

### run()

Accepts a callback function as a parameter that will perform the test

the recommended way to use it.

### Marble diagram for testing observable

This creates a new cold observable based on the pattern passed

1. each character in string represents a single frame in the diagram
2. a frame is the unit used to represent virtual time in a testScheduler
3. each frame represents 1 virtual millisecond (vms)
4. each dash (-) represents the passing of 1 vms
5. the values a,b,c will be produced with 1 vms between them

* ‘--a-4---c-8|’
  1. the observable will complete directly after 8 is produced
* ‘ --a-4 12ms c-8#’
  1. white spaces in beginning are ignored
  2. 12ms: instead of writing 12 dashes we can write the time between two spaces
  3. the hash-tag(#) an error will be produced
* ‘-a-^-b-(cde)---f|’
  1. ^: represents a subscription, only applies to hot observables

it means the subscriber wont receive the value ‘a’ because it hadn't subscribed yet

* 1. (cde): will occur synchronously in the same frame

### Marble diagram for testing subscriptions

* ‘^---!’: will begin then end after 3 ms
* ‘--^-’: will subscribe after 2ms and will never unsubscribe
* ‘^ 10ms !’