# Angular RxJS

## RxJS Terms and Syntax

**Start the stream (subscribe)**

* Emits items into the stream

**Items pass through a set of operation (pipe the data through set of operators)**

**An observer**

* Next item, process it – next()
* Error occurred, handle it – error()
* Complete, you are done – complete()

**Stop the stream (unsubscribe)**

### Observer/Subscriber

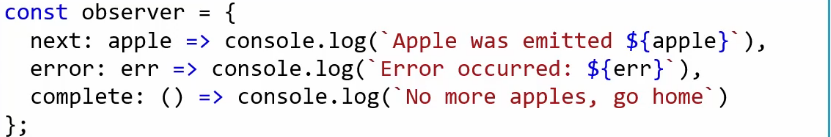
Observer is an object monitors the stream and responds to notification

* Next item, process it
* Error occurred, handle it
* Complete, you’re done

Observer is a collection of callbacks that knows how to listen to values delivered by the Observable.

In RxJS observer is also defined as an interface with next, error and complete methods. One such class implements the Observer is **Subscriber**. Subscriber is an observer with additional feature to unsubscribe from an observable.

In simple way an observer object can be created as object literal as below



### Observable Streams

Stream is the data moving along the conveyor. In RxJS stream of data is called observable stream.

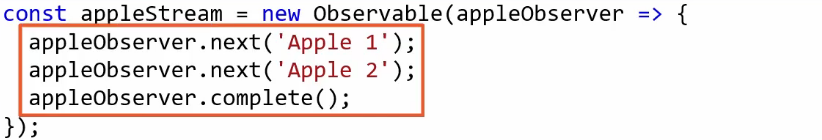
We emit any type of data into an observable stream.

* Number
* String
* Event
* Object literal
* Response returned from an HTTP request
* Other observable streams

Observable streams are called Observable sequence, Observable or Stream.

Observable can be synchronous (Items emitted immediately) or asynchronous (items emitted in delay)

We create a observable by passing a observer as below

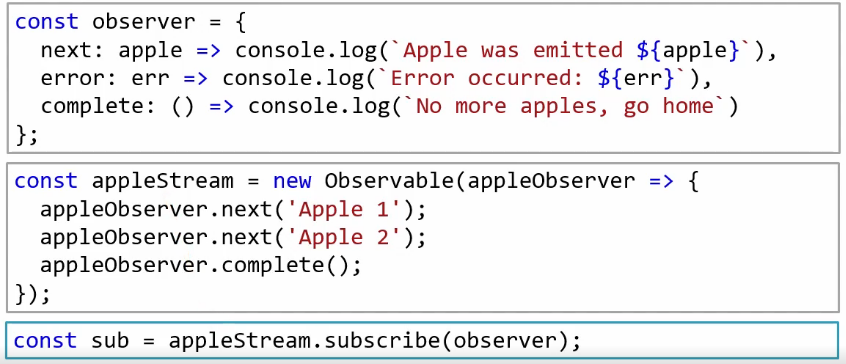


The code in the function executes when stream is started. Observable are lazy they don’t execute when defined.

But we normally don’t write the code like this, there is easier way to create observable stream.

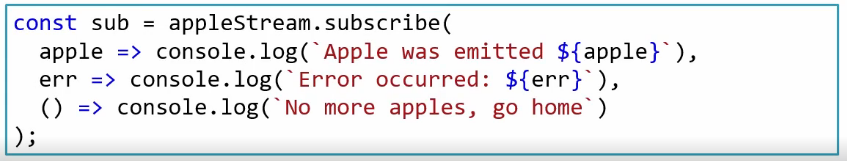
### Starting the Observable stream

With RxJS we start the streams by calling **subscribe()** method of Observable. We must subscribe the observable to start the stream. Otherwise no values are emitted to stream and we have nothing to observe.



The subscribe() method returns a subscription to represent the execution of observable.

When working with observable we don’t create a separate observer and pass it to subscribe method rather we pass next, error and complete method directly into the subscribe method like below.

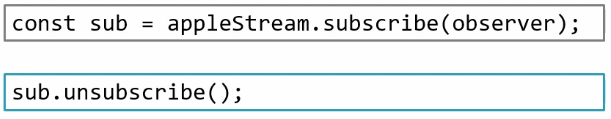


### Stopping the Observable Stream

In RxJS properly stopping each of Observable streams helps avoiding potential memory lick in our application. There are many ways to stop the stream.

* Call complete
* Use completing operator
* Throw an error
* Unsubscribe

Properly **unsubscribing** from each observable which doesn’t complete their own prevents memory leaks.

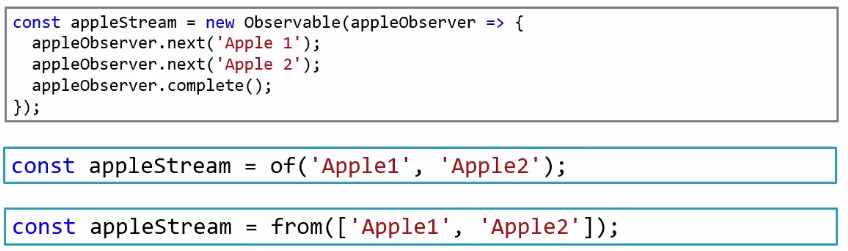


### Creation Functions

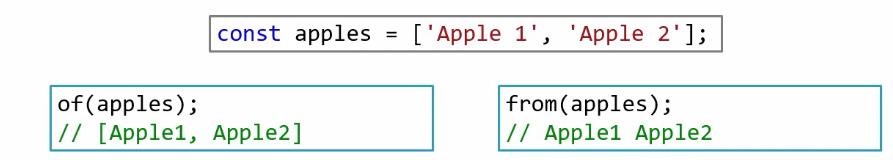
Using RxJS in angular application we often don’t create our own observable rather we work with the observables angular creates for us for there are times we create for us.

We can create observable using Observable constructor (in above case) but the recommended way is to use creation functions (static functions).

* **of –** Creates an observable using set of defined values. Emitting each value and then completes the stream. Automatically unsubscribes when last item is emitted.
* **from –** from creates an observable from an array or other data structures. Emitting each of the values from other structure and then completing the stream.



**of** v/s **from**

****

Here **of** emits the array of item as a single item in the stream.

To make it with **of** we need to use javascript **spread** operator this spreads the elements of the array to **of** function argument.



There are several other creation functions available in RxJS.

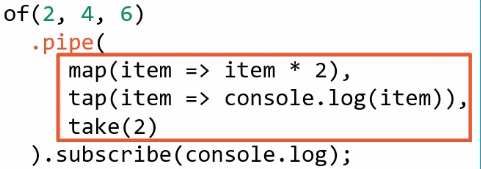
* **fromEvent –** creates an observable from dom event
* **interval –** creates an observable with emits a value at defined interval time

### Creation function demo

**“Stackblitz”** is a online tool powered by VSCode where we can code for any UI framework including Angular JS.

## RxJS Operators

An operator is a function used to transform and manipulate items in an observable stream. We apply operators in sequence using observable **pipe** method.



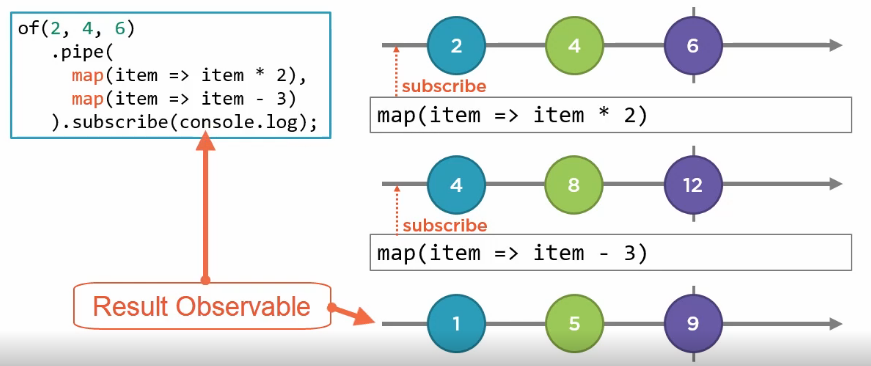
Inside pipe we pass the operators separated by commas. Items processed through pipe method hence through all the operators one by one. In the above case first 2 is emitted will be processed completely then 4 and then 6.

More technically when we subscribe to observable with operators. Each operator takes the observable as an input the operator subscribe to that input observable and creates and returns an output observable after process in the way defined in operator argument. The output observable is an input observable to the operator that follows in the sequence.

There are 100s of RxJS operators are defined and available can be referred from

<https://rxjs.dev/api>

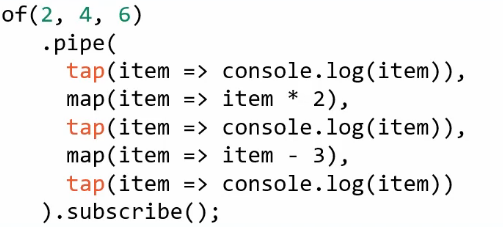
**Marble diagram**

****

To note each item is processed through all of the operator before next item is processed.

### Tap operator

tap() operator is a best friend when we are debugging.



tap() is a utility operator. The operation it performs inside it doesn’t change the output. Which means tap() output an observable identical to input observable. Hence tap() is very helpful if we want to perform some operation like incrementing counter etc.

### Take operator

Emits a specific number of items. Ex. take(2)

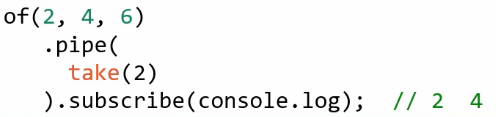
take() operator used for

* Taking a specific number of items
* Limiting unlimited streams

Once take(2) limit is completed or taken, it unsubscribes its input observable which unsubscribes up to the chain to source observable and no further item is emitted.

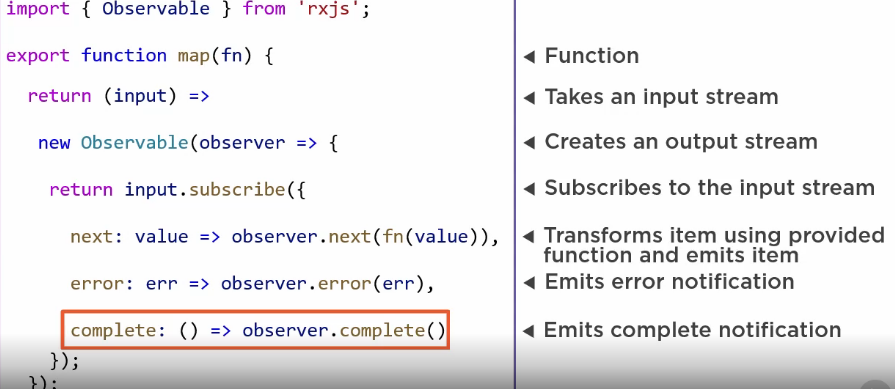
Take is a filtering operator

* Takes in an input stream, subscribes
* Creates an output stream



### Operators internal

map() takes an input operator stream returns output operator stream/observable



## Going Reactive

Here instead of retrieving data into an array and then binding to that array, we work with observables and bind to those observables using RxJS.

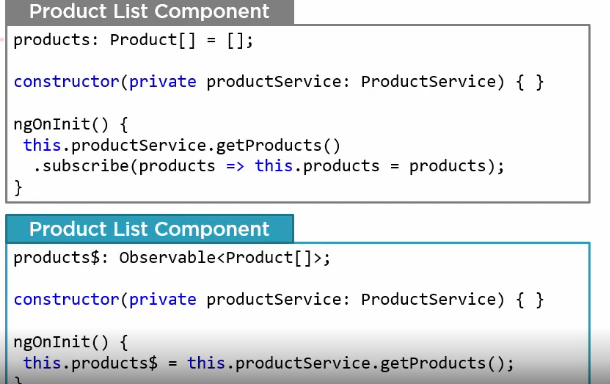
### Working with Async pipe

We access the observable directly inside the template using async pipe.

* The async pipe automatically subscribes to an observable when component is initialized and returns each emitted value from that observable.
* When a new item is emitted the component is marked to be checked for changes.
* It automatically unsubscribes to observable when component is destroyed to avoid potential memory leaks.

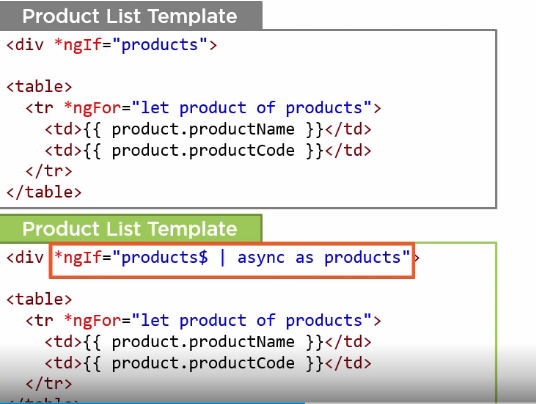
Async pipe doesn’t required service code to be changed only component and template will be enough

Below is 2 differences designed using common data retrieval pattern using observable and using async pipe.



1. We start by replacing product[] with Observable<Product[]>. Here $ is the notation to variable to say this is an Observable.
2. Since async pipe handles the subscription we don’t need to subscribe in our component code.
3. We simply assign the Observable return to our observable property.

Below is the template change:



1. We pipe the observable property through async pipe to access its emitted values. This automatically subscribes for us and handles unsubscribing.
2. We use **as** clause to catch emitted item to a variable.

### Error Handling

Any error stops the observable stream, so it doesn’t emit any value.

We can handle error as

* Catch and replace
* Catch and re-throw

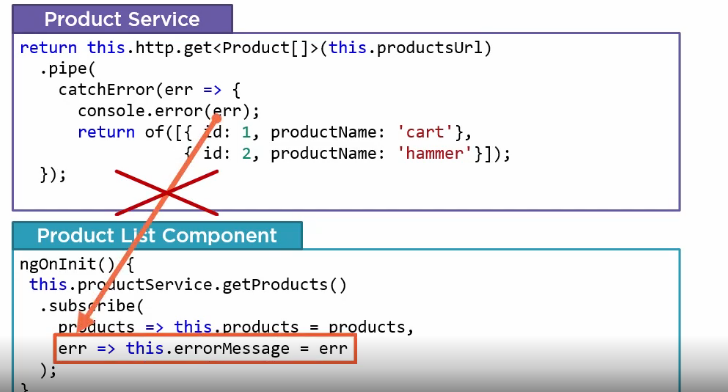
RxJS **catchError(this.handleError)** operator catches any error occurred in observable.

Use catchError for catching error and

* Rethrowing an error.
* Or replace the error observable to continue after error occurred. (may be an observable created from hardcoded or local data or can also empty RxJS constant)

### Catch and Replace

If we catch error the observer will not be notified for the error as the error is catched. Instead observer’s next method is called with the new emitted observable we created and returned.



In summary **catchError** is an error handling method

* Takes an input stream, subscribes to it
* Creates an output stream.

When an error occurs in source observable, catchError

* Catches the error
* Unsubscribes from the input stream
* Returns a replacement observable
* Optionally re throws the error propagating it further up the chain

### Re-throw the error

In this strategy we will catch the error with **catchError** will log it, we use **throwError(err)** to propagate the error further up the chain to other parts of application.

throwError is a creation function which

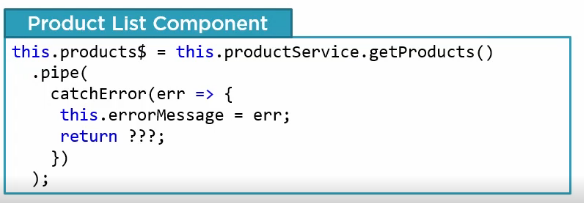
* Creates an observable that emits no items
* And immediately emits an error notification



handleError(err) could be used to log the error and converts to an user friendly error message.

**throwError(err)** replaces the original Observable with Observable<never> then emits an error essentially re throwing.

Since the service re-throws the error, it’s up to the component to catch the error.



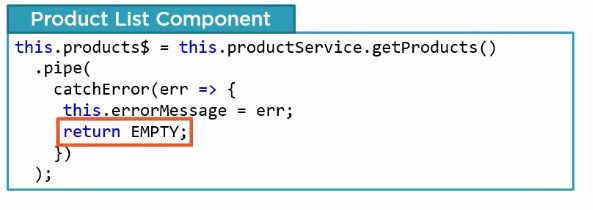
??? here we may not want to re-throw the error as it will propagate the template, rather we could catch and replace with empty observable.

### EMPTY

EMPTY is a RxJS constant that creates an observable that emits no items. And immediately emits a complete notification.

Used for

* Returning an empty observable
* Especially helpful in replacing an error observable when we don’t have any default value.



In template for adding error message:



### Improving change detection

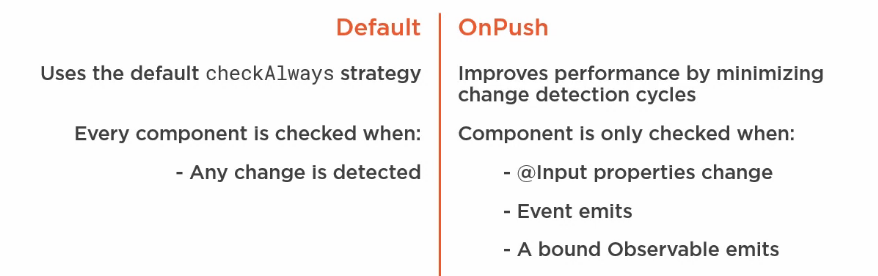
Benefits of using angular async pipe over traditional Observable subscription

* No need to subscribe
* No need to unsubscribe
* Improve change detection

Angular uses **change detection** to track changes to angular data structure. So it knows when to update UI with changed data.

Change detection ensures our template binding to render the current data from component.

There are basic 2 change detection strategies



**onPush** change detection strategy improves performance as it minimize change detection cycles.

We set the change detection strategy as part of component decorator.



With onPush change detection strategy we just can’t set to a bound property and expect it to update in the user interface. Change detection won’t know this change.

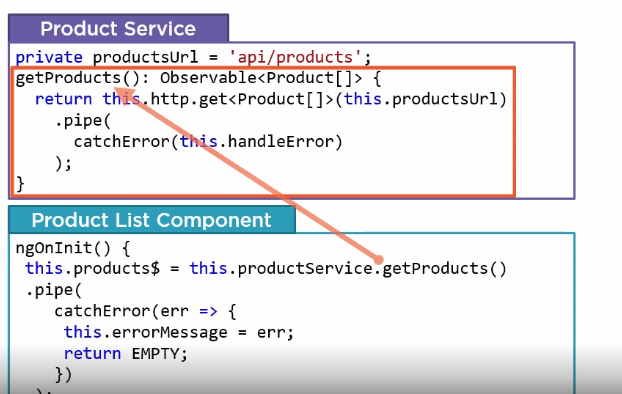
Now our UI won’t show/update the error message unless an observable emits another item.



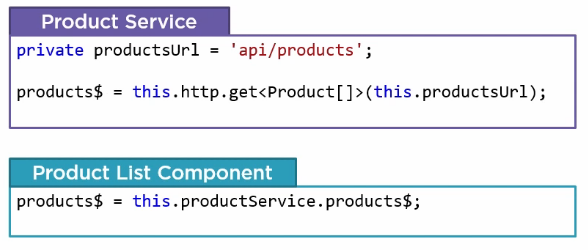
We need to change our error message to observable stream which we will cover later.

### Declarative pattern for data retrieval

The data retrieval pattern we are currently using is procedural. Because in the service we have a procedure that we call from our component



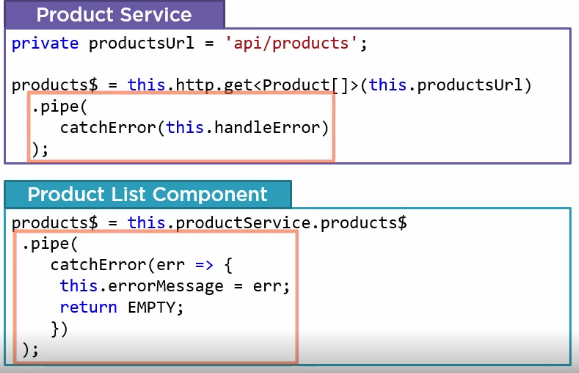
Can we make it more **declarative ?** Below is the code



Here in service we declared an observable property and assign it directly to an http GET. And in component we assign a local property to an Observable property in service.

No more getProducts() method no more ngOnInit() method.

We add error handling or other operation as below. We pipe through sequence of operators



Benefit of declarative approach

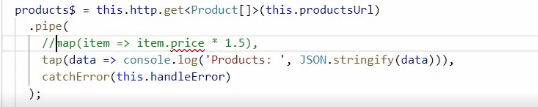
* Better leverage power RxJS observable and operators because we works with streams directly instead calling method that returns streams
* Effectively combine observable streams for more complex data managements
* Easily share observables
* Readily react to user actions

## Mapping Returned data

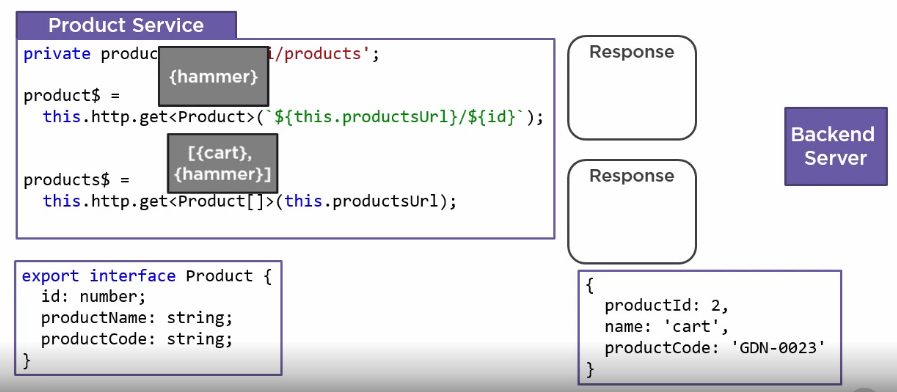
### Mapping an HTTP response

We use map() operator for mapping the data from an Observable.

The way till we are using to get products$ observable, with this transforming individual elements using map() operator will error out because here http.get is not returning a single product Observable but Observable<Product[]>.



When we issue a HTTP GET request asking for an single item such as a product, we receive a single response containing that item. HttpClientService automatically map the response to defined shape (i.e. product interface shape). And when we subscribes to this observable it emit one item product. But often we issue a HTTP GET request asking for a set of items such as all products, here we still receives a single response but that response contains a set of requested item, which is in this case an array of items. And when subscribed to the Observable emits one item that is the array of products.

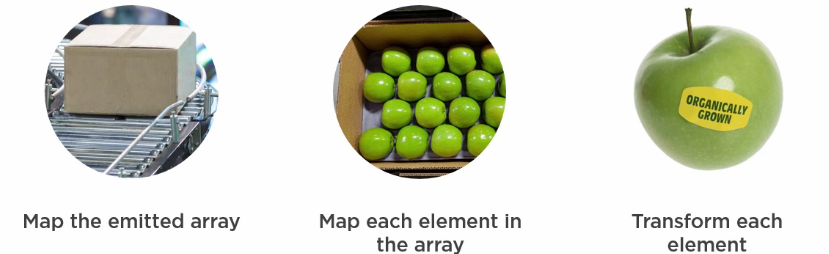


In picture what we think of getting observable v/s what we received as observable:



This box represents the array of items emitted by HTTP response. **So we can’t use simple mapping techniques.**

If we need to modify every element emitted in HTTP response. We first need to map the emitted array, then map each element in the array and lastly transform each element. Here in below picture we modify each apple by fixing a label.



### Mapping the emitted array and Elements in the array



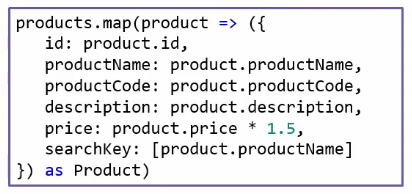
In the above code we have mapped the emitted array (first map) and then each element to uplift the price to 1.5 (second map).

This will not work in our case ? Because the second map highlighted will return **Observable of numbers** as we converted the product only price property to uplift 1.5.

**Solution** we need to rebuild a new product and emit instead only price to uplift, which is called transforming.

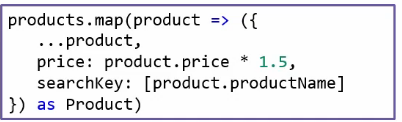
### Transforming array elements

Either we can do it in traditional way by building a new product object by replacing the part we want to change. Or we can use spread operator.

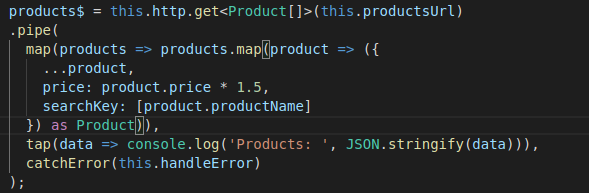


To define object literal and not to make it ambiguous for compiler for considering lambda body we need to enclose with **( -** parenthesis and **as Product** is for strongly typing.

But we can use javascript **spread …** operator to copy values and then can only modify/add values we want.



Result code:

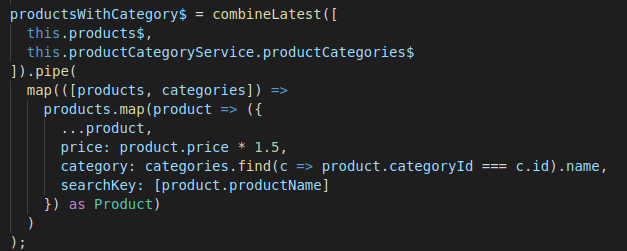


## Combining Multiple Streams

We need to combine multiple streams to get some desired output. Below are the methods with which we can combine streams

* combineLatest
* forkJoin
* withLatestFrom

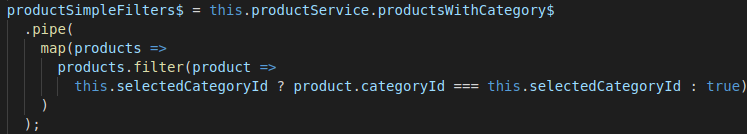
Below is an example with “combineLatest” where we combined product$ and productCategories$ observable to map from categoryId to category



## Reacting to Actions

### Filtering a stream

“filter” is a transformation operator, takes an input stream subscribes to it and creates an output stream based on evaluation.

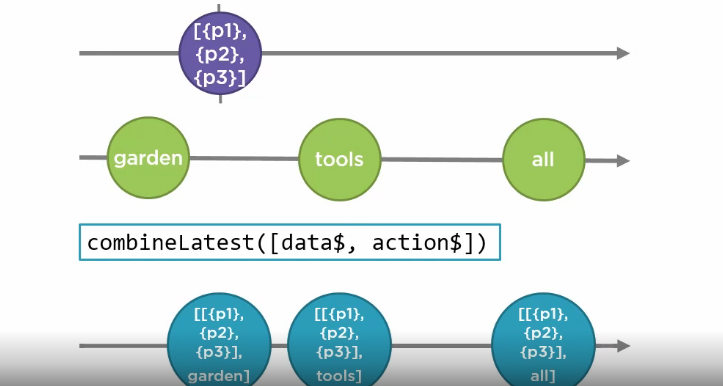


### Data Stream v/s Action Stream

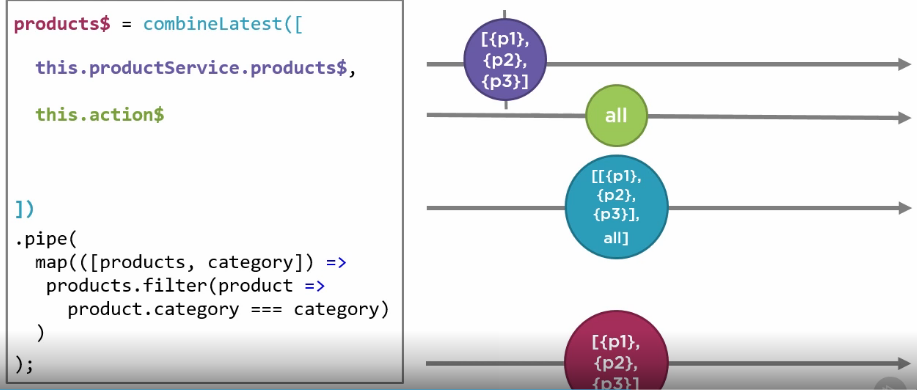
Action stream, emits value every time when any action is performed until it’s stopped.

Now, if we use **combineLatest,** it emits first when each stream emits and then when action stream emits in our below example.

The array emitted from combineLatest contents our data and latest action data. And when combineLatest emits it re-fires the downstream pipe line, so if pipeline filters the data, it will perform the filter again. This is key to reacting to action



Below is an example:



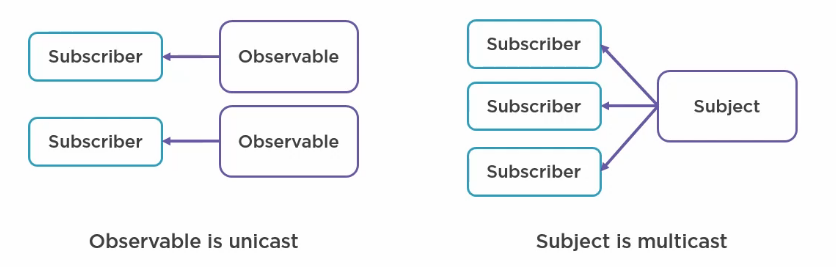
We already have product$ stream, but how to create action$ stream. To create action$ stream we have 3 different ways

* Use built-in stream
* fromEvent
* Subject/BehaviorSubject (it’s a common way)

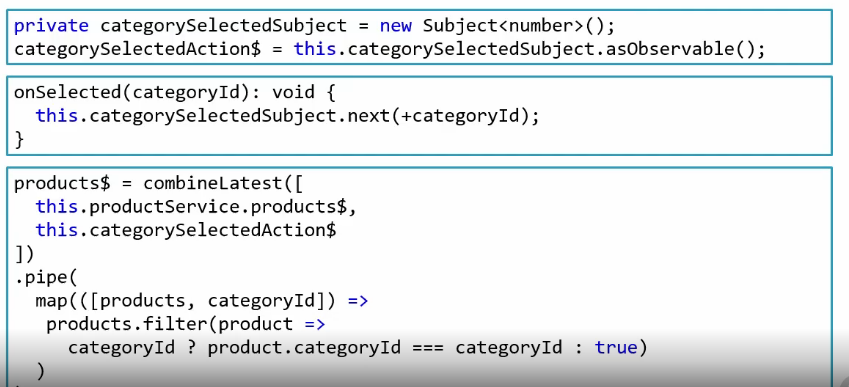
### Subject and BehaviorSubject

**Subject** is a special type of observable that is both observable and observer.

Observable us unicast (each subscriber will get their own copy) where Subject is multicast where every subscriber will share the same copy.



Below is the code:

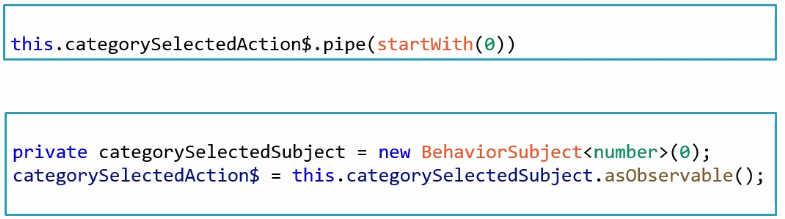


### Setting up with an initial value

In the previous case as long we don’t select the filter elements won’t be displayed because categorySelectedAction$ doesn’t emit any value and hence combineLatest won’t emit.

So we need to set an initial value to start with.

There are several ways to do so, below are two ways



### Reacting to an Add Operation

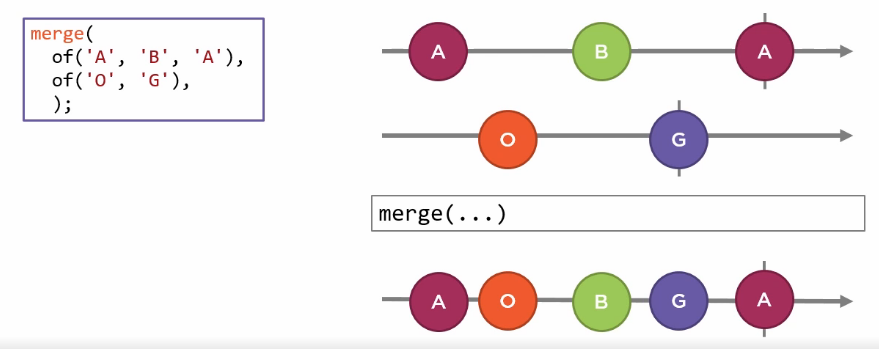
**merge(a$, b$, c$):**

merge combines multiple streams by merging their emissions.

Merge is a static creation function, not a pipeable operator.

Used for combining sequences of similar types to blend their emitted value.

Marble diagram:



“merge” is a combination function

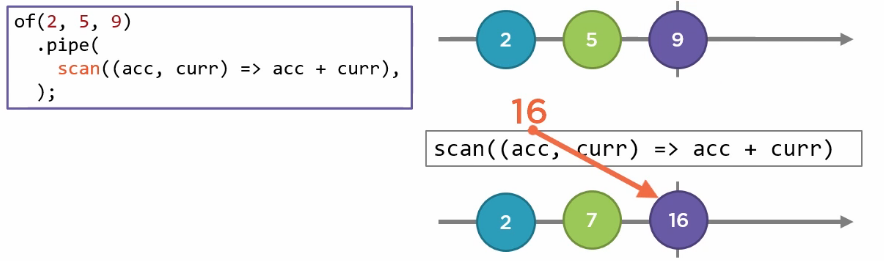
* Takes in a set of streams, subscribes
* Creates an output stream
* It completes when all input stream completes.

**Scan**

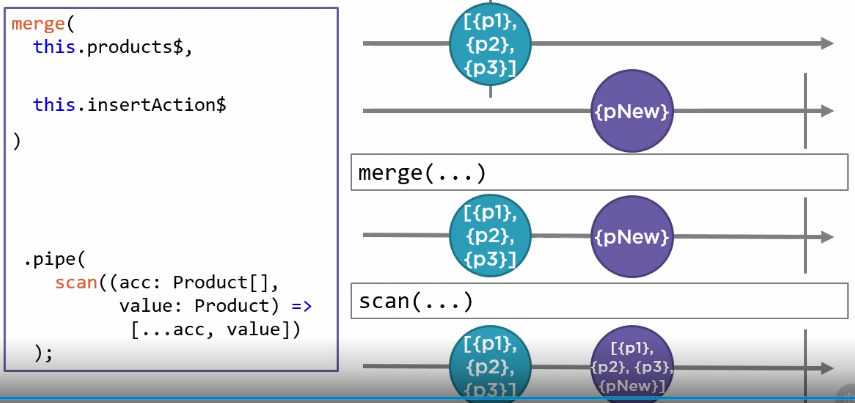
“scan” accumulates items in a stream.

“scan” used for

* Totaling amounts
* Accumulating items into array



Example:



## Caching Observables

When the data doesn’t change so frequently, caching improves the performance, reduces network load.

We can cache the data inside and outside of our application.

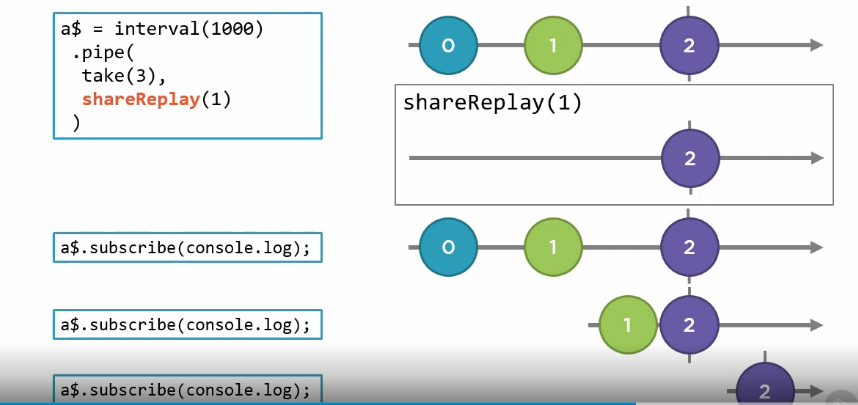
### Patterns for data caching

**“shareReplay”** shares the stream with other subscribers, used for caching data in application.

Replays the defined number of emissions on subscription. Takes an optional buffer size, which is the number of items cached and replayed.

On subscribe it emits the specified number of emissions.

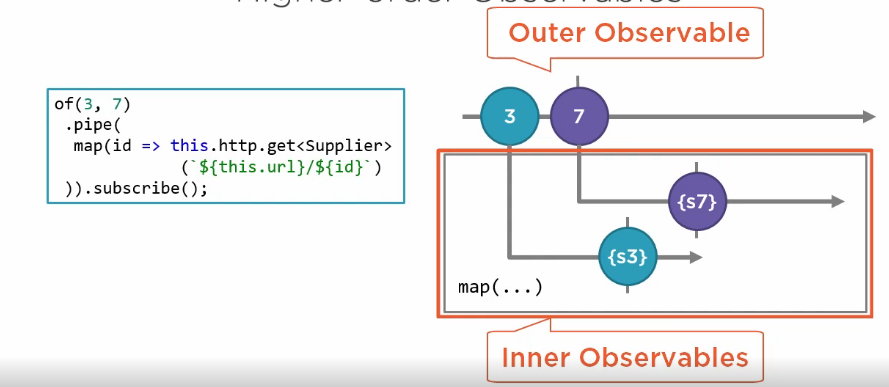
Items stays cached forever, even after there are no more subscribers.



## Higher Order Mapping

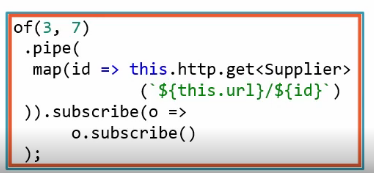
Some observable emits primitive, some emits arrays. Some observables emit other observable.

Observable emits other observables called higher order observables.



There is problem with above code, we are subscribing to outer observable, but instead we need to subscribe to inner observable.

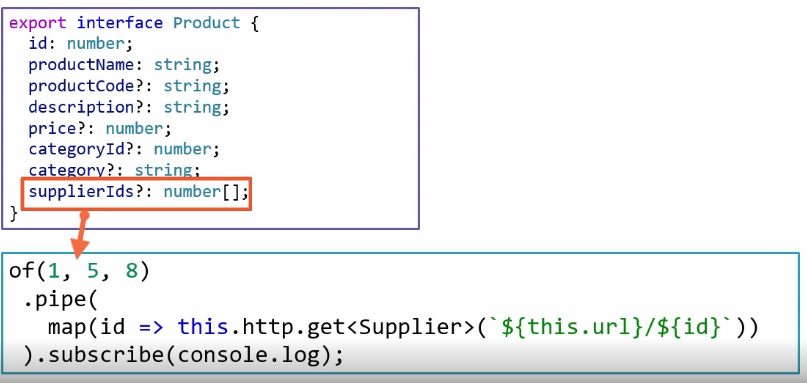
So the code should look like below:



This technique is not recommended as how we will subscribe to inner observable using async pipe and how we unsubscribe it. We **should not have nested subscription in our code.**

Si instead we need to use a Higher Order Mapping operator, which transforms higher order observables.

### Higher Order Mapping Operators

Here in our code, we will use each of supplier id to get each supplier for a product using HTTP GET.

Below are some higher order mapping operator:

* concatMap
* mergeMap
* switchMap

### concatMap

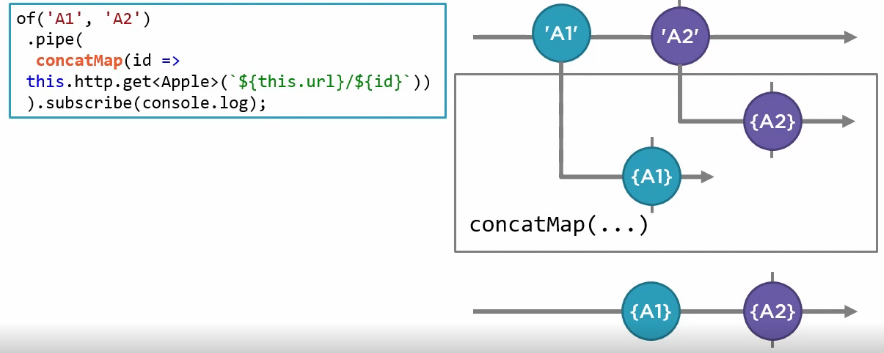
concatMap is Higher Order Mapping + Concatenation

This transforms each emitted items to a new (inner) Observables as defined by function.



It waits each inner Observable to complete before processing next one.

It concatenates results in sequence.



“concatMap” is a transformation operator

* takes in an input stream, subscribes and creates an output stream

When an item is emitted from input stream, its queued

* Item is mapped to an inner observable as specified by a provided function
* concatMap subscribes to inner observable and **waits!** For it to complete.
* Inner observable emissions are concatenated to output stream
* When inner observables then only next item from outer stream is processed.
* concatMap completes when outer observable is completes.

Use “concatMap” to

* To wait for the prior observable to complete before starting next one.
* To process items in sequence
* Example: If we have set of ids we need to process/display data in sequence/order
* If we need update or delete data in sequence

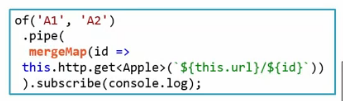
### mergeMap

“mergeMap” is a higher order mapping + merging

Transform each emitted item into a new inner observable as defined by a function



“mergeMap” executes inner observable **in parallel** and merges their results.



Use “mergeMap” to

* Process in parallel
* When order doesn’t matter

### switchMap

“switchMap” is a higher order mapping + switching

Transforms each emitted items to a new (inner) observable as defined by function.



“switchMap” **stops** the prior inner observable then **switches** to inner observable

Use “switchMap” to

* Stop any prior observable before switching to next one
* Example: Type ahead auto completion, User selection from a list