The Beginner's Guide to RxJS

Satyapriya Mishra

About the Book

The Beginner's Guide to RxJS is an open source book designed to help the JavaScript developers get a starting point to understand all the quirky parts of the RxJS library by presenting the concepts in a concise manner. The different sections of the Ebook are written in the form of articles so that a reader can start from any page and grab the concept.

About the Author

Satyapriya Mishra is Software Consultant, Technology Trainer, Public Speaker, Author and Blogger with years of experience in building enterprise software applications and mentoring individuals as well as teams. He is very passionate about sharing his knowledge to others and he is often found teaching, mentoring and guiding young professionals to achieve their career objectives. He is the founder of http://code2stepup.com/ where he discusses technical stuff.

He can be reached at http://imsatya.com if you have any interesting project to discuss.

What is RxJS

RxJS stands for Reactive Extensions for JavaScript. It is a library for reactive programming using observables that makes it easier to compose asynchronous or callback-based code. RxJS gives us the ability to handle asynchronous calls with multiple events.

Earlier Promises were used to handle this thing for us, but they had certain limitations. A Promise can only handle one value but an Observable can handle a stream of values and it can funnel the values from one action item to the other. Another drawback of Promise is that it is not cancellable. But Observables are cancellable. These are the two primary reasons that developers choose RxJS over Promises. RxJS library gives us a lot of flexibility in terms of operators to make our code easier to write and to make it more readable and maintainable.

In the following sections we will have a walk through of the most important variants and operators from the RxJS ecosystem.

Reference Count

In this section we will explore how RxJS keeps a count of the number of subscribers which are subscribed to a particular Observable. This reference count mechanism dictates whether an observable should execute or not. The magic lies with the refCount() method.

refCount() method returns an Observable that keeps track of how many subscribers the observable has got.If the number of subscribers is greater than 0, it calls connect() method and starts the execution.When the subscribers is 0, it stops further execution.

A simple code snippet demonstrating this concept is given below.

```
const refCounter = interval(1000); // (1)
let subscription1;
let subscription2;
console.log('observerA subscribed');
 subscription1 = refCounter.subscribe({ // (2)
  next: (v) => console.log(`observerA: ${v}`)
});

setTimeout(() => {
  console.log('observerB subscribed');
   subscription2 = refCounter.subscribe({ // (3)
    next: (v) => console.log(`observerB: ${v}`)
  });
}, 600);

setTimeout(() => {
  console.log('observerA unsubscribed');
  subscription1.unsubscribe(); // (4)
}, 1200);

// This is when the shared Observable execution will stop, because
// `refCounted` would have no more subscribers after this
setTimeout(() => {
  console.log('observerB unsubscribed');
  subscription2.unsubscribe(); // (5)
}, 2000);
```

On step 1, we create a const refCounter with the help of the interval operator, which emits numerical values every 1000ms as an observable. The detailed explanation of the interval operator is given in the operators sections. By this step we create an observable.

In step2 and step3 we subscribe to this refCounter observable twice and pass the next() method to capture the values. A discussion about the next() method is given in the Subjects section.

In step4, we unsubscribe the first subscription by using the unsubscribe() method. So now we have only one active subscription.

In the last step we unsubscribe the second subscription using the same unsubscribe() method.Now the observable refCounter does not have any active subscriptions.
At this point the execution of the observable stops.

The output of the above example will be as follows.

```
// observerA subscribed
// observerB subscribed
// observerA: 0
// observerA unsubscribed
// observerB: 0
// observerB unsubscribed
```

We can clearly see that the observers get subscriptions and then start emitting values. Then when both the observers unsubscribe they stop emitting any value.

The reference count is the internal mechanism of RxJS due to which to execute an observable we need at least one subscription.

Subject

A Subject is a special type of Observable which shares a single execution path among observers.
Think of this as a single speaker talking at a microphone in a room full of people.
Their message (the subject) is being delivered to many (multicast) people (the observers) at once. This is the basis of multicasting. Typical **unicast** observables would be comparable to a 1 on 1 conversation.

By default Observables are unicast which essentially means that the source observable emitting the values can be subscribed by only one subscriber, which is not very useful as for a real time UI application we want to show the same value at different places by subscribing the value simultaneously. Subject solves this problem by enabling multicasting.

Whenever a Subject is subscribed by a new observer, the observer gets registered in the registry of the Subject such that it will receive new values from the Subject any time it emits a next value. If a subscriber is subscribed to a Subject, it does not how many other subscribers are attached to the same subject. It is like the native addEventListener() method which does not know how many event listeners are listening to the same event triggered on a single DOM element.

Subject is an object with **next**(v), **error**(e) and **complete**() methods. The next() method is called to set the next value of the Subject, error() is called in the eventuality of an error and complete() method is reserved for signalling that the source does not have any more values to emit to the observers.

There are 4 types of subjects which are listed below.

1.subject

2.behaviorsubject

3.replaysubject

4.async subject

The above variants of Subject are all of type Observable and very much similar except for the data they emit.

A very simple example of the subject is given in the next section.

```
const subject = new Subject<number>();

subject.subscribe({
  next: (val) => console.log(`observerA: ${v}`)
});
subject.subscribe({
  next: (val) => console.log(`observerB: ${v}`)
});

const observable = from([1, 2, 3]);
observable.subscribe(subject);
```

The output of the above code snippet will be:

```
// observerA: 1
// observerB: 1
// observerA: 2
// observerB: 2
// observerA: 3
// observerB: 3
```

In the above code snippet we have instantiated a new Subject on the first line. Then we subscribe to the subject twice through two subscribers. Inside the subscription function we have used the next() method to set the next value of the subject and simply printing it to the developer's console.

In the subsequent lines we create an observable with the **from** operator which emits values one after another. Then we subscribe to the above observable.

In the output we can see that for each value emitted from the observable, the two subscribers receive the value separately and print the emitted values in sequence.

*** If we don't call the **next()** method, Subject does not return any value.

This was a very simplistic overview of how Subject works.

Behavior Subject

Behavior Subject is a type of Subject which in turn is a special type of Observable which we can subscribe to get the last value. It needs to be initialized. Even if it has not been executed through a next() method, it can still return the last value which is the initialized value.

```
import { BehaviorSubject } from 'rxjs';

  const bs$ = new BehaviorSubject(1);
    bs$.subscribe((value) => {
      console.log(value);
    });

    bs$.next(2);
```

Here we have instantiated a new BehaviorSubject through the constant bs$. We have initialized it with a value 1. Next we subscribe to the BehaviorSubject to print the current value. On the next statement we set the value of the BehaviorSubject to 2. If we look at the developers tool in the browser we get the following output.

```
  // 1
  // 2
```

***BehaviorSubject is an observable as well as an observer. We can send it a value apart from subscribing it.

***We can get an observable from BehaviorSubject using the asObservable() method on BehaviorSubject.

***An observable can emit values when the next() method is executed at least once. But for BehaviorSubject we don't need to call the next() method. It will return at least one value.

***Observables return different values to different observers whereas BehaviorSubject returns the same value i.e the last value to all the observers.

This fact can be demonstrated by the following code snippet.

```
  const randomNumber1 = new Observable((observer) => { // (1)
      observer.next(Math.random());
    });

    randomNumber1
           .subscribe(num => console.log('observer 1: '+ num)); // (2)

    randomNumber1
           .subscribe(num => console.log('observer 2: '+ num)); // (3)


    const  randomNumber2 = new BehaviorSubject(0); // (4)
    randomNumber2.next(Math.random());

    randomNumber2
           .subscribe(num => console.log('observer behaviorsubject 1: ' +
num));

    randomNumber2
           .subscribe(num => console.log('observer behaviorsubject 2: ' +
num));
    }
```

In the above code snippets, in step 1, we created an Observable and set the next() value from a random number generated by the Math object. In the next steps 2 and 3 , we subscribe to the observable. Next in step 4, we create a BehaviorSubject with the initial value as 0. In the next subsequent steps, we subscribe to the BehaviorSubject.

When we run the above code we will get the following output printed onto the console.

```
    // observer 1: 0.08208077981971873
    // observer 2: 0.2697879588072649
    // observer behavior subject 1: 0.46334149248702117
    // observer behavior subject 2: 0.46334149248702117
```

We can see that the observable returns different values, but the subscribers to the BehaviorSubject return the same value i.e. the last value emitted by the BehaviorSubject. Some of the key takeaways related to BehaviorSubject are listed below.

- Observable does not have a state and it is just a function, BS has state.
- Different instances of code run for different observers, for BehaviorSubject the same instance runs for all.
- Observables are cold i.e code gets executed when they have at least a single observer.
- BehaviorSubject is hot i.e: code gets executed and emits value even if there is no observer.
- Observable works on a PUSH pattern, whereas BehaviorSubject works on both push and pull as BS can be an observer as well as observable.

Replay Subject

ReplaySubject is a special variant of the Subject that emits old values to observers even though at the time of execution of those values the observer is not created. It provides for the setting of the number of old values that it should remember which will be emitted once a new subscriber registers with it.

A simple demonstration for replay subject is given with the following code snippet.

```
import { ReplaySubject } from 'rxjs';

const rs$ = new ReplaySubject(2); // (1)

// subscriber A
rs$.subscribe((data) => { // (2)
  console.log('subscriber A: ' + data);
});

rs$.next(Math.random()); // (3)
rs$.next(Math.random());
rs$.next(Math.random());

// Subscriber B
rs$.subscribe((data) => { // (4)
  console.log('subscriber B:  ' + data);
});

rs$.next(Math.random());
```

In step 1 we instantiate a new ReplaySubject and set the number of in memory values to be 2. That means it will store at least 2 values in the memory so that when it is subscribed, it does have at least 2 values to be emitted. In step2 we have subscribed to the replay subject created in step1 and here we are just printing the value to the console. In step3, we are using the next() method to pass new values to the replay subject. We do this step 3 times to check if we can really store at least 2 values in the memory. In the next step4, we create another subscriber

called subscriberB and just like the previous subscriber it prints the value to the console. In the last step we do a call to the next() method again and set the value to some random number. The key observation here is that by the time subscriberB comes into existence, the replay subject has already emitted three values and through our configuration we have saved two past values in memory.Let's see the output now.

```
// subscriber A: 0.8107208546492104  // (1)
// subscriber A: 0.04243867985866512 // (2)
// subscriber A: 0.5350443486512133  // (3)
// subscriber B: 0.04243867985866512 // (4)
// subscriber B: 0.5350443486512133  // (5)
// subscriber A: 0.07003469196276346 // (6)
// subscriber B: 0.07003469196276346 // (7)
```

Here in the output we have got 7 values and looking at the output values we can draw the following inferences.

- Values 1, 2 and 3 are coming from the next() method which we implemented in step3 of the code snippet. By this time the subscriberB is not in existence. So there is no value being printed from subscriberB.

- Now subscriberB is created and as per the configuration, there are 2 past values in the memory which the subscriber prints.

- One thing to observe here is that values 4 & 2, and values 5 & 3 are the same since they are the in memory values of the replay subject.

- Next values 6 & 7 are coming from the last statement of the code snippet.

The next variant of the Observable is the asyncSubject which we will see in the next section.

Async Subject

AsyncSubject is a variant of Subject where only the last value of the Observable execution is sent to its observers, and only when the execution is complete. Until the Observable is complete, there will not be any value subscribed to the subscribers.

The asyncSubject is a very good candidate for network requests through GET or fetch API where only one value is expected.

A simple demonstration of the asyncSubject is given below.

```
import {  AsyncSubject } from 'rxjs';

    const as$ = new AsyncSubject(); // (1)

    as$.subscribe((data) => { // (2)
      console.log('Subscriber A:  ' + data);
    });
    as$.next(Math.random()); // (3)
    as$.complete(); // (4)


    as$.subscribe((data) => {
      console.log('Subscriber B:  ' + data);
    });
    as$.next(Math.random());
    as$.next(Math.random());
    as$.complete();
```

Let’s check what is happening in each of the steps in the above code snippet.
In step1, we create a new asyncSubject through the constant as$. Then in step2, we subscribe to the asyncSubject created in the previous step. Then we assign a new value to the

asyncSubject with the next() operator and then we complete it. We repeat the same steps for creating a new subscriber called B.

Now first let's see the output for the above code snippet.

```
// Subscriber A:  0.38896475571548295
// Subscriber B:  0.38896475571548295
```

In the output we can see two random numbers which are coming from the next() methods as we had implemented in the above snippet.
Here one thing to observe is that, even if we have called the next() method twice for the subscriber B, we are getting only one value printed. This essentially means that the subscriber got only one value from the source and it got the value when the source asyncSubject was completed. That is the reason that we got only one value for both the subscribers. That is the reason why asyncSubject is an ideal candidate for fetch api calls.

This finishes the major variants of Observables.In the next sections we will explore the operators who make RxJS so powerful.

ajax operator

The Ajax operator creates an observable for an ajax request. It is used to make API calls. We can perform all http request methods with this ajax operator. It converts a normal ajax call to an observable and emits the data as an observable.

An example of the ajax operator is given below.

```
    const postsUrl = `https://jsonplaceholder.typicode.com/posts`;
    const posts = ajax(postsUrl);
    const subscribe = posts.subscribe(
      res => console.log(res),
      err => console.error(err)
    );
```

And the response of the above code without formatting in commented sequence will be:

```
  /** AjaxResponse {originalEvent: ProgressEvent, xhr: XMLHttpRequest,
request: {…}, status: 200, responseType: "json", …}
originalEvent: ProgressEvent {isTrusted: true, lengthComputable: false,
loaded: 27520, total: 0, type: "load", …}
xhr: XMLHttpRequest {__zone_symbol__timeoutfalse: Array(1),
__zone_symbol__errorfalse: Array(1), __zone_symbol__ON_PROPERTYtimeout: ƒ,
__zone_symbol__ON_PROPERTYerror: ƒ,
__zone_symbol__ON_PROPERTYreadystatechange: ƒ, …}
request: {async: true, crossDomain: true, withCredentials: false, headers:
{…}, createXHR: ƒ, …}
status: 200
responseType: "json"
response: (100) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}]
__proto__: Object */
```

Since the ajax gives output in the form of an observable, we can subscribe to the output value emitted by the operator to get the data in JSON format which can be used in our application. So the code for getting the formatted JSON data through the ajax operator is given below.

```
const postsUrl = `https://jsonplaceholder.typicode.com/posts`; // (1)
const posts = ajax.getJSON(postsUrl); // (2)
const subscribe = posts.subscribe( // (3)
  res => console.log(res),
  err => console.error(err)
);
```

In the first step, we set a url which we will hit to fetch the data. In the next step, we pass the url to the getJSON method which is a method part of the ajax operator . Then in the next step, we subscribe to the value so that when the data is fetched from the remote API, we print it. Please note that here we also have an error handler for the eventuality if any error occurs .

If for the ajax operator we want to send object to the remote API we will have to perform the below code.

```
const postsUrl = `https://jsonplaceholder.typicode.com/posts`;
 const posts  = ajax({
   url: postsUrl,
   method: 'GET',
   headers: {
     'Content-Type': 'application/json'
   },
   body: {

   }
 })
 posts.subscribe(
   res => console.log(res),
   err => console.error(err)
 );
```

Here we can see that with the ajax method, we are sending the url, the method, headers and a body object with the request.Other things remain the same for subscription and error handling.

audit operator

The audit operator ignores the values emitted from the source observable for a duration as specified by another observable and when this duration is over, it emits the latest value from the source observable. All the previous values emitted by the source observable previous to the latest value, are ignored. One thing to observe here is that the parameter which dictates the duration of ignoring the source values must be an observable itself.

Signature:

audit(durationSelector: Observable): MonoTypeOperatorFunction

durationSelector: It is a function that receives the time duration from the source observable.
MonoTypeOperatorFunction: It is an observable that performs rate-limiting of emissions from the source.

A simple demonstration of the audit operator can be shown as follows.

```
const timer = interval(1000).pipe(take(20)); // (1)
const result = timer.pipe(audit(ev => interval(5000))); // (2)
result.subscribe(val => console.log(val)); // (3)
```

Let's follow the above code snippet step by step.

In step1, we created a new observable called times through the interval operator that emits continuous values after a given time interval, 1000ms here. Then we pass this observable value to the pipe operator and apply that to the **take** operator. The take operator executes only the first 20 values emitted from the source observable i.e. the interval operator here. So in a nutshell we can say that the constant value timer essentially emits 20 values starting from 0.

In step2, we pass the timer to a pipe again, this time through the audit operator and assign this to a new constant called result. The audit operator takes another observable as input to get the duration value and in our case it is again the interval operator which emits after every 5000ms.

In the last step, as with all our observables, we subscribe to the result and print the values. Quite straight forward.

In the output we get the following result.

```
// 5
// 11
// 17
```

auditTime operator

The auditTime operator waits for a given duration of time and then emits the most recent value emitted by the source observable. This process is repeated again and again till the source observable emitting the values is complete.

Signature:

auditTime(duration: number, scheduler: SchedulerLike = async): MonoTypeOperatorFunction

duration: Time in milliseconds for which the values from the source observable are ignored.
scheduler: It manages the timer that rate limits the emissions to the output observable
Default value is async.
MonoTypeOperatorFunction: It is an observable that does the rate limiting emission from the source observable

A simple code block for the auditTime operator is given below.

```
const source = interval(1000).pipe(take(20)); // (1)
const result = source.pipe(auditTime(3000)); // (2)
result.subscribe(val => console.log(val)); // (3)
```

Let's analyze the above code block in steps.

In step1, we create an observable through the interval operator which emits values in certain durations and we decided to take only the first 20 values , through the take operator. In the next step, we pipe that source observable and pass in the auditTime operator with 3000ms as the duration time. Next step we subscribe to the observable and print the emitted values.One thing to observe here is that the audit operator takes the duration specifier as an observable, but the auditTime operator takes it as a number.

The output of the above code comes as numbers emitted after 3ms duration which is very evident.

```
// 3
// 7
// 11
// 15
```

buffer operator

Buffers the source Observable values until **closingNotifier** emits.Collects values from the past, i.e which were not emitted and waits for the next emit from the notifier.
In simple words we can say the buffer operator waits till another provided observable emits, and as soon as the inner observable emits, it collects the previously emitted values from the source, combines them in an array and emits them.

Signature:
buffer(closingNotifier: Observable<any>): OperatorFunction

closingNotifier: An observable that emits to notify
OperatorFunction: Function that emits the output as an observable

A sample code snippet for the buffer operator is given below.

```
const intervalObs = interval(1000).pipe(take(10)); // (1)
const timeInterval = interval(4000); // (2)
const ob$ = intervalObs.pipe(buffer(timeInterval)); // (3)
ob$.subscribe(val => console.log(val)); // (4)
```

In step1, we create an observable called intervalObs through the interval operator and take only the first 10 emissions. In the second step we create another observable through the interval operator again, but it emits values after every 4000ms. Then in the next step, we pipe the intervalObs through the buffer operator with the timeInterval observable as parameter and assign it to a constant value called ob$. As with other observables we subscribe to it and print the value onto the console.

The result of the above code snippet comes out as follows.

```
// [0, 1, 2]
// [3, 4, 5, 6]
```

After printing [3, 4, 5, 6] which is the collection of the past values after printing 2, the intervalObs completes and so there is no further emission by the buffer operator.

bufferCount

Buffers the source Observable values until the size hits the maximum bufferSize provided. It also takes an optional parameter called startBufferEvery which is the value interval of creating a new buffer.

Signature:
bufferCount(bufferSize: number, startBufferEvery: number = null): OperatorFunction

Buffers a number of values from the source Observable by bufferSize then emits the buffer and clears it, and starts a new buffer each startBufferEvery values.
If startBufferEvery is not provided or is null, then new buffers are started immediately at the start of the source and when each buffer closes and is emitted.

A sample code snippet for the bufferCount operator is given below.

```
const source = of(1, 2, 3, 4, 5); // (1)
const buffered = source.pipe(bufferCount(2, 2)); // (2)
buffered.subscribe(val => console.log(val));
```

In step1, we create an observable with the **of** operator which eits values from the provided values sequentially. In the next step we pass that observable to the bufferCount operator with the maximum size as 2 and the startBufferEvery as 2. Next we subscribe to it.

The output of the above code snippet will be

```
// [1, 2]
// [3, 4]
// [5]
```

As we see due to the parameters we passed to the bufferCount operator the output comes in this format. The parameters can be tweaked in different ways to get different outputs.

combineLatest operator

Combines multiple Observables to create an Observable whose values are calculated from the latest values of each of its input Observables.Whenever any input observable emits a value, it computes a formula using the latest values from all the inputs, then emits the output of that formula.

combineLatest combines the values from all the Observables passed as arguments. This happens by subscribing to each Observable in order and, whenever any Observable emits, collecting an array of the most recent values from each Observable. So if we pass an observable to the combineLatest operator, the returned Observable will always emit an array of n values, in order corresponding to order of passed Observables (value from the first Observable on the first place and so on).

*** Passing an empty array will result in an Observable that completes immediately.
*** If an error emitted from one observable, the combineLatest throws an error and completes.
*** Emits values, if at least one input observable emits.

A simple code snippet implementing the combineLatest operator is given below.

```
const ob1$ = interval(1000).pipe(take(4));
const ob2$ = of(5, 6, 7, 8);
const ob3$ = timer(1000, 1000).pipe(take(4));
combineLatest(ob1$, ob2$, ob3$).subscribe(val => console.log(val));
```

Here we take 3 different source observables and use the combineLatest operator to combine them all and finally we are printing the result.
The result for the above code is given below.

```
// [0, 8, 0]
// [1, 8, 0]
// [1, 8, 1]
// [2, 8, 1]
// [2, 8, 2]
// [3, 8, 2]
// [3, 8, 3]
```

concat operator

The concat operator creates an output Observable which sequentially emits all values from given Observable and then moves on to the next.This operator first emits the values from the first observable, then the second observable and so on.

Signature:
concat(...observables): Observable

The simplest example of the concat operator can be as follows.

```
const ob1$ = of(1, 2, 3, 4);
const ob2$ = interval(1000).pipe(take(4));
const result = concat(ob1$, ob2$);
result.subscribe(val => console.log(val));
```

Here we pass two observables and then pass the references of these observables to the concat operator and store the expression in result.Then we subscribe to the result and print the output.

The output for the above code will be:

```
// 1
// 2
// 3
// 4
// 0
// 1
// 2
// 3
```

We can see that the output is sequentially combined as per the sequence of the provided observables.If we reverse the sequence of the operands, we get reversed emitted values.

Following are some of the important points regarding the concat operator.

- concat subscribes to each observables one at a time in a sequence and merges their result.

- it will subscribe to the first observable and emit its values until it completes, then move to the next.This will be repeated till the concat runs out of observables.

- when the last observable is completed, the concat also completes.

- at one time only one observable can emit values.

- if one observable does not complete, then concat will never complete and the observables next to the incomplete observable will never be subscribed and not emit any value.

- if any observable emits an error, then the next observables never gets subscribed.

concatMapTo operator

The concatMapTo operator maps each source value to the given Observable innerObservable regardless of the source value, and then flattens those resulting Observables into one single Observable, which is the output Observable. Each new value emitted is concatenated to the previous value.It projects each source value to the source observable and flattens it.

Signature:
concatMapTo(innerObservable, resultSelector, outerIndex): OperatorFunction

resultSelector: optional parameter
OperatorFunction: A function that emits output as observable

An example of the concatMapTo operator is given below.

```
const source  = of(1,2,3);
source.pipe(concatMapTo(interval(1000).pipe(take(3))))
.subscribe((val) => {
  console.log(val);
});
```

And the corresponding output will be:

```
// 0
// 1
// 2
// 0
// 1
// 2
// 0
// 1
// 2
```

count operator

The count operator counts the number of values emitted from the source observable.This operator returns an output observable as a number specifying the number of elements emitted by the source observable. The count operator returns the value after the source observable completes.

Signature:
count(predicate?: Function): OperatorFunction

predicate: it is a function that returns a boolean and it specifies what values are to be counted.
OperatorFunction: A function that returns an observable as a number.

Without the predicate passed the count operator counts all the values emitted by the source observable as given below.

```
const source = of(1, 2, 3, 4, 5);
const result = source.pipe(count());
result.subscribe(val => console.log(val))
```

Since we are passing an observable through the **of** operator with 5 values, the output of the above program is 5.

With the predicate passed the count operator counts the values emitted by the source observable as per the precondition provided to it.

```
const numbers = of(1,2,3,4,5,6);
const result = numbers.pipe(count(i => i > 3));
result.subscribe(val => console.log(val));
```

Here since only 3 values pass the predicate condition, we get 3 as output on the console.

debounce operator

The debounce operator emits a value from the source Observable only after a particular time span determined by another Observable has passed.After the time span it emits the last value emitted by the source observable.

Signature:
debounce(durationSelector: Function): MonoTypeOperatorFunction

durationSelector: It is a function that receives the time span value from the source observable and stops emitting input values till the time span is elapsed.
MonoTypeOperatorFunction: It is a function that returns an observable as output.

An example of the debounce operator is given below.

```
const source = of(1,2,3,4,5,6);
const result = source.pipe(debounce(() => interval(2000)));
result.subscribe(val => console.log(val));
```

Here we take an observable that emits values through the **of** operator. Then on it we apply the debounce operator with the interval operator with time set to 2000ms. By the time the interval operator emits its value, the source observable completes its emissions. So the output of the above operation is 6 which is the last value emitted by the source observable.

delay operator

Delays the emission of items from the source Observable by a given timeout or until a given Date.

signature:
delay(delay: number | Date, scheduler: SchedulerLike = async):

delay: the delay duration in milliseconds or date after which the observable will start emitting values
scheduler: optional, defaults to async.

The simplest example of the delay operator can be as follow.

```
const timer1 = interval(1000);

timer1.pipe(delay(5000)).subscribe(val => {console.log(val); });
```

Here simply we take an observable with the interval operator and then apply the delay operator to it with a delay of 5000. The above code starts emitting values after 5s.

We can also set a future value to emit values after that particular date is elapsed. The code for using a date alongside the delay operator is given below.

```
const timer1 = interval(1000);
timer1.pipe(delay(new Date('January 01, 2030 12:00:00')))
.subscribe(val => { console.log(val); });
```

The above code will start emitting values only after the specified date.

distinct operator

The distinct operator returns all the values emitted by the source observable which are not the same as the previous emitted values. In other words, it filters out the duplicate values from the source observable.

Signature:
distinct(keySelector): MonoTypeOperatorFunction

keySelector: A function to select which value to check as distinct. This is an optional parameter.
MonoTypeOperatorFunction: It is an output observable that returns the distinct values.

A basic code example of the distinct operator is given below.

```
const source = of(1,2,3,1,3,2,4,2,1,5,6,1).pipe(
  distinct(),
)
source.subscribe(val => console.log(val));
```

The output of all the unique values from the above code will be:

```
// 1
// 2
// 3
// 4
// 5
// 6
```

empty operator

The empty operator creates an observable that emits no items to the Observer and immediately emits a complete notification.This operator never returns any value and just completes the observable. This operator never returns any value and just completes the observable.

Signature:
empty(scheduler?: SchedulerLike)

schedulerLike: It schedules the completion of the emission.

```
const result = empty().subscribe(val => {console.log(val)});
```

This operator is used with composing other observables such as mergeMap.

every operator

The every operator returns an observable of type boolean specifying if all the items emitted from the source observable meets a certain precondition.
If any of the items don't meet the condition, it returns false.
If all the items meet the condition, it returns true.

Signature:

every<T>(predicate: Function): OperatorFunction

predicate: It is a function that takes the precondition to test all the emitted values from the source
OperatorFunction: It is an output observable function that returns a boolean.

A simple code example of the **every** operator is given below.

```
const source = of(1, 2, 3, 4, 5);
const result = source.pipe(
  every(x => x > 0), // (1)
);
result.subscribe(val => console.log(val));
```

Here we pass a condition to the operator which checks if the emitted values from the source pass the precondition.

The output of the above code will be true as all the values emitted from the source observable pass the condition.

expand operator

The expand operator recursively calls a provided function and projects each source value to an observable.

Signature:
expand(project: Function, concurrent: number , scheduler): OperatorFunction

The expand operator is similar to the mergeMap but it calls the function recursively to each source and output observable.

It applies a function to each of the emitted values from the source observable and then merges these resulting observables to the output observable and repeats this process.

Expand will re-emit on the output Observable every source value. Then, each output value is given to the project function which returns an inner Observable to be merged on the output Observable.Those output values resulting from the projection are also given to the project function to produce new output values.

An example explaining the functionality of the expand operator is given below.

```
    from([1,2,3,4,5]) // source obs emitting the value
    .pipe(
      expand(val => { // project function
        return of(val * 2); // operation to be performed and returned as
an observable
      }),
      take(5) // taking the first 5 values
    ).subscribe((val) => { // subscribing to the output obs stream and
consoling it
      console.log(val);
    });
```

In step1, we create an observable source through the **from** operator. In the next step, we write the project function and here we are just doubling up the emitted values and we only take the first five values. In the next step we subscribe to the output values and print them.

The output of the above code will come in this format.

```
// 1
// 2
// 4
// 8
// 16
```

This was a basic demo of the expand operator.

filter operator

The filter operator filters emitted items based on a condition. It is very similar to the Array.prototype.filter() method in JavaScript.If none of the elements satisfy the condition, the filter operator returns nothing.

Signature:
filter(predicate: Function): MonoTypeOperatorFunction

predicate: A mandatory parameter passed as a function which checks the condition with each item emitted by the source observable
MonoTypeOperatorFunction: A function which outputs an observable stream of values that passes the predicate function.

An example of the filter operator is given below.

```
const source = of(1,2,3,1,3,2,4,2,1,5,6,1).pipe(
  filter(i => i > 3),
)
source.subscribe(val => console.log(val));
```

The above is self explanatory and the output for the above code snippet will be:

```
// 4
// 5
// 6
```

find operator

The find operator emits the first value from the source observable that meets a certain condition. If none of the values match the condition, it returns undefined.

Signature:
find(predicate: Function): OperatorFunction

predicate: It is the function that matches the precondition to the emitted values from the source observable.
OperatorFunction: It is a function that emits the first value that matches the condition as an observable

***The difference between **find** and **first** is that the predicate function in find operator is mandatory. The first operator is given in the next section.

A code example of the find operator is given below.

```
const source = of(1,2,3,4,5,6);
const result = source.pipe(find(i => i % 2 === 0));
result.subscribe(val => console.log(val));
```

The output of the above code will be 2 as it is the first element that passes the predicate function.

first operator

The first operator returns the first value emitted by the source observable if no condition is provided.
If a condition is provided, then it returns the first value from the source observable which matches this condition.

Signature:
first(predicate): OperatorFunction

Predicate: It is an optional parameter that tests each emitted value with a function to check a precondition.
OperatorFunction: A function that emits the output as an observable.

A sample code example of the first operator without any predicate function is given below. If there is no predicate function provided, the first operator returns the first value emitted by the source.

```
const source = of(1,2,3,1,3,2,4,2,1,5,6,1).pipe(
  first(),
)
source.subscribe(val => console.log(val));
```

It is evident from the above code that the output will be 1, which is the first emitted value from the source observable.

With a predicate value provided, it returns the first value satisfying the condition as shown below.

```
const source = of(1,2,3,1,3,2,4,2,1,5,6,1).pipe(
  first(i => i > 3),
)
source.subscribe(val => console.log(val));  }
```

Here we have provided a predicate function that checks the first emitted value that is greater than 3. So here the output will be 4.

forkJoin operator

ForkJoin accepts an array of observable inputs or a dictionary of observable inputs and returns an observable that emits either an array of last values from the respective inputs in the exact same order as the passed array or a dictionary of values in the same shape as the passed dictionary.

Signature:
forkJoin(...inputObservables: any[]): Observable<any>

Let's first see the forkJoin operator in action with a simple example and then we will explore some other facts about this operator.

```
const ob1$ = interval(1000).pipe(take(4));
const ob2$ = of(5, 6, 7, 8);
const ob3$ = timer(1000, 1000).pipe(take(4));
forkJoin(ob1$, ob2$, ob3$).subscribe(val => console.log(val));
```

Here we have taken three observable sources and forkJoined them.
The output of the above will be:

```
// [3, 8, 3]
```

Some of the interesting points about forkJoin are as follows.

- it takes inputs as an array/dictionary of observables.
- if the number of observables are provided, then it completes immediately.
- It waits for all the input observables to complete and then finally emits the last values emitted by the input observables in sequence.
- it emits once and then completes.
- if any obs gives error, forkJoin will also give error and all obs will get unsubscribed.

fromEvent operator

Creates an Observable that emits events of a specific type coming from the given event target.

Signature:
fromEvent(target, eventName): Observable<T>

This operator creates an Observable from DOM events, or Node.js EventEmitter events or others.

```
const source = fromEvent(document, 'click');
source.subscribe((event: any) => {
  console.log(`Clicked on coordinates(X: ${event.pageX},
   Y: ${event.pageY})`);
});
```

When we click on any body part of the browser the above code prints the coordinates of the position of the click.

The output of the above code snippet will come in the below format.

```
// Clicked on coordinates(X: 609, Y: 340)
// Clicked on coordinates(X: 582, Y: 402)
```

fromEventPattern operator

Creates an observable from an arbitrary API for registering event handlers.

Signature:

fromEventPattern(addHandler, removeHandler = undefined) => void): Observable

addHandler => takes a function and attaches it to the source of the event.
removeHandler => takes a function and removes it from the source of event, default is undefined.

A simple example of the fromEventPattern operator is given below.

```
function addHandler(handler) {
  document.addEventListener('click', handler);
}

function removeHandler(handler) {
  document.removeEventListener('click', handler);
}

const ob$ = fromEventPattern( // here we are attaching the handlers
  addHandler,
  removeHandler
);
ob$.subscribe((value: any) => {
  console.log(value);
  console.log(`You clicked on (X: ${value.pageX}, Y: ${value.pageY}`);
});
```

The above code gives us the coordinates of the clicks inside the browser body.

groupBy operator

The groupBy operator groups the emitted input observables based on certain conditions and emits them as grouped observables. The output groupedObservable is a collection of observables by a key which is generated by the pre condition.

Signature:
groupBy(keySelector, elementSelector, durationSelector):GroupedObservable

keySelector: A function that extracts the key for each input observable item.
elementSelector: It is Optional. Default is undefined.A function that extracts the return element for each item.
durationSelector: Optional. Default is undefined.A function that returns an Observable to determine how long each group should exist.
GroupedObservable: An Observable that emits GroupedObservables, each of which corresponds to a unique key value.

A code example of the groupBy operator is given below.

```
of(1,2,3,4,5,6)
.pipe(
  groupBy(item => item % 2), // condition
)
.subscribe(val => console.log(val));
```

Here we take a source observable and apply the groupBy operator on it to group the emitted values in multiples of 2. And the corresponding output of the above code snippet will be:

```
GroupedObservable {_isScalar: false, key: 1, groupSubject: Subject,
refCountSubscription: GroupBySubscriber}
groupSubject: Subject {_isScalar: false, observers: Array(0), closed:
false, isStopped: true, hasError: false, …}
key: 1
refCountSubscription: GroupBySubscriber {closed: true, _parent: null,
_parents: null, _subscriptions: null, syncErrorValue: null, …}
_isScalar: false
__proto__: Observable
```

```
GroupedObservable {_isScalar: false, key: 0, groupSubject: Subject,
refCountSubscription: GroupBySubscriber}
groupSubject: Subject {_isScalar: false, observers: Array(0), closed:
false, isStopped: true, hasError: false, …}
key: 0
refCountSubscription: GroupBySubscriber {closed: true, _parent: null,
_parents: null, _subscriptions: null, syncErrorValue: null, …}
_isScalar: false
__proto__: Observable
```

In the output we can clearly see that the output observables are grouped by keys 0 and 1.

interval operator

The interval operator creates an observable that emits positive numbers after provided intervals. It is like the JavaScript setInterval() method in the form of an Observable.

Signature:
interval(count: number, scheduler: SchedulerLike = async)

Interval operator returns an Observable that emits an infinite sequence of ascending integers, with a constant interval of time of your choosing between those emissions. The first emission is not sent immediately, but only after the first period has passed. By default, this operator uses the async SchedulerLike to provide a notion of time, but you may pass any SchedulerLike to it.

A very simple demonstration of the interval operator is given below.

```
const source = interval(1000);
source.subscribe(val => console.log(val));
```

The above code keeps on printing numbers starting from 0 with an interval of 1000ms.

The interval operator can also be piped and can be subjected to the take operator to take a finite count of emitted values like below example.

```
const source = interval(1000).pipe(take(4));
source.subscribe(val => console.log(val));
```

The above code only emits the first 4 values from the source.

last operator

The last operator emits the last emitted value from the source observable if no condition is passed and it returns the last value emitted that passes the condition if some condition/predicate is provided.

Signature:
last(predicate?: Function): OperatorFunction

predicate: A function to check the last item of the emitted values which passes the pre-condition.
OperatorFunction: A function that returns the output as observable

With no predicate provided, 1 is the last value emitted by the source observable. The sample code is given below.

```
const source = of(1,2,3,1,3,2,4,2,1,5,6,1,2,3,2,1).pipe(
  last(),
)
source.subscribe(val => console.log(val));
```

With a predicate value provided, 6 is the last emitted value from the source that passes the condition. After that all emitted values from the source are less than 3.

```
const source = of(1,2,3,1,3,2,4,2,1,5,6,1,2,3,2,1).pipe(
  last(i => i > 3),
)
source.subscribe(val => console.log(val));
```

map operator

The map operator is very similar to the JavaScript Array.prototype.map method. As in the case of array map method which iterates over an array and transforms each value according to the function provided, the RXJS map operator takes individual values from source observables and transforms it based on the function. The map operator outputs an observable.

Signature:
map(project: Function): OperatorFunction

OperatorFunction: The OperatorFunction is the output emitted as an observable.

A simple code example for the map operator is given below.

```
 of(1,2,3,4,5,6) // source observable
.pipe(
   map(item => item * 2),
 )
 .subscribe(val => console.log(val));
```

Here the map operator is applied on an observable , and it transforms all the emitted values by simply multiplying 2 to each operand.
Here the output is very evident and it will be:

```
// 2
// 4
// 6
// 8
// 10
// 12
```

mapTo operator

The mapTo operator emits a constant value each time a new value is emitted from the source observable.It takes the constant value as an argument and always emits that as output observable corresponding to each emission of input value. So in nutshell we can say that it always maps to a constant value.

A straightforward example of the mapTo operator is given below.

```
of(1,2,3,4,5,6)
.pipe(
   mapTo(10),
 )
 .subscribe(val => console.log(val));
```

Here we have passed 10 as the constant value. So the output will be 10 corresponding to each emitted value from the input.

```
// 10
// 10
// 10
// 10
// 10
// 10
```

max operator

The max operator operates on a source observable that emits comparable values and when the source completes emission, the operator returns the maximum of the values emitted by the source observable

Signature:
max(comparer ?: Function): MonoTypeOperatorFunction

comparer: it is the function which takes the criterion for comparison
MonoTypeOperatorFunction: it is a function that returns the output maximum value as an observable

***Without comparer function the max operator by default returns the highest value from all the emitted values by the source observable.

An example demonstrating the max operator without the comparer function is given below.

```
const source = of(1,2,-2, 3,4,5,6);
const result = source.pipe(max());
result.subscribe((val) => {console.log(val);});
```

Here since we did not pass any comparer function, it will return the max values among the emitted values i.e. 6.

With the comparer function provided the max operator compares the values recursively as per the given criterion as provided in the function and emits the highest value as output.

```
of(10, 22, 32, 11, 4)
.pipe(
  max( (a, b) => a < b ? -1 : 1),
)
.subscribe((val) => console.log(val));
```

Here the output will be 32.

merge operator

Creates an output observable which concurrently emits all values from every given input observable.

Signature:
merge(...observables: any[]): Observable

The merge operator **Flattens** multiple Observables together by blending their values into one Observable.The merge operator subscribes to each given input Observable (as arguments), and simply forwards (without doing any transformation) all the values from all the input Observables to the output Observable.
The output Observable only completes once all input Observables have completed. Any error delivered by an input Observable will be immediately emitted on the output Observable.

```
const ob1$ = interval(1000).pipe(take(2));
const ob2$ = of(5, 6, 7, 8);
const ob3$ = timer(1000, 1000).pipe(take(2));
merge(ob1$, ob2$, ob3$).subscribe(val => console.log(val));
```

The output of the above code will be:

```
// 5
// 6
// 7
// 8
// 0
// 0
// 1
// 1
```

mergeMap operator

The mergeMap operator projects source emitted values to a function and merges the output of the transformation to the stream of output observable.

Signature:
mergeMap(project: Function, resultSelector?: number, outerIndex: number, innerIndex: number, concurrent: number = Number.POSITIVE_INFINITY): OperatorFunction

project: A transformation function that takes the emitted value as input and transforms it
concurrent: Maximum number of input Observables being subscribed to concurrently.
OperatorFunction: A function that returns the output observable

An example of mergeMap is given below.

```
const source = of(1,2);
const result = source.pipe(
  mergeMap(x => interval(1000).pipe(take(x)) ),
);
result.subscribe(val => console.log(val));
```

And the corresponding output will be:

```
// 0
// 0
// 1
```

mergeMapTo operator

The mergeMapTo operator projects each source value to the same observable which is merged multiple times to the output observable.The difference of mergeMapTo from mergeMap is that it maps each emitted value to the same observable.

Signature:
mergeMapTo(innerObservable: O, resultSelector, concurrent: number = Number.POSITIVE_INFINITY): OperatorFunction

innerObservable: It replaces each value from the source observable.
resultSelector: Optional, default value is undefined.
concurrent: Maximum number of input Observables being subscribed to concurrently.

The mergeMapTo operator returns an observable which is termed as the OperatorFunction.

```
of(1,2,3,4,5) // source observable
.pipe(mergeMapTo(interval(1000).pipe(take(5))))
.subscribe(val => console.log(val));
```

```
// 0 (5 times)
// 1 (5 times)
// 2 (5 times)
// 3 (5 times)
// 4 (5 times)
```

min operator

The min operator operates on a source observable that emits comparable values and when the source completes emission, the operator returns the minimum of the values emitted by the source observable.

Signature:
min(comparer ?: Function): MonoTypeOperatorFunction

comparer: it is the function which takes the criterion for comparison
MonoTypeOperatorFunction: it is a function that returns the output min value as an observable

Without comparer function the min operator by default returns the least value from all the emitted values by the source observable.

```
const source = of(1,2,-2, 3,4,5,6);
const result = source.pipe(min());
result.subscribe((val) => {console.log(val);});
```

As evident by the code, the output here will be -2.

With the comparer function provided the min operator compares the values recursively as per the given criterion as provided in the function and emits the least value as output.

```
of(10, 22, 32, 11, 4)
.pipe(
  min( (a, b) => a < b ? -1 : 1),
)
.subscribe((val) => console.log(val));
```

Here we have provided a comparer function to the min operator and the output here will be 10 as it is the minimum emitted value from the source observable.

race operator

Race operator returns an observable, from a list of input observable, which wins the race of emitting the first value.Whichever observable emits the first value, the race operator mirrors that observable, and it completes once the winner observable completes.

Signature:
race(...observables: any[]): Observable

A typical example of the race operator is given below.

```
const obs1 = interval(1000);
const obs2 = of(1, 2, 3, 4, 5);
const obs3 = interval(5000);

race(obs1, obs2, obs3)
  .subscribe(
    next => console.log(next)
  );
```

The output of the above will be:

```
// 1
// 2
// 3
// 4
// 5
```

Here the **of** operator emits first and so it wins the race and all the values emitted by the of operator is printed onto the console.

range operator

It emits a sequence of numbers within a given range as an observable.

Signature:
range(start: number = 0, count?: number, scheduler?: SchedulerLike): Observable<number>

An example of the range operator is given below.

```
const ob$ = range(1, 5);
ob$.subscribe(val => console.log(val));
```

The above code prints the following.

```
// 1
// 2
// 3
// 4
// 5
```

sample operator

This operator emits the last value emitted by the source of the observable, whenever another observable called as the notifier emits.It ignores the in between values.
If there is no emitted value in between two subsequent emissions from the notifier, then no value is emitted.

Signature:
sample(notifier: Observable<any>): MonoTypeOperatorFunction

notifier: Observable used for sampling the source observable.

A simple example for the sample operator is given below.

```
const timer$ = interval(1000);
const clickEvents = fromEvent(document, 'click');
const ob$ = timer$.pipe(sample(clickEvents));
ob$.subscribe(x => console.log(x));
```

Here the clickEvents is the notifier which has been passed to the sample operator as an argument.

We have an interval operator working emitting values every second.
Whenever a click event occurs or in other words , the notifier emits, we log the last value emitted from the interval operator. So here the output comes as numbers and depends on the time when the click event occured.

skip operator

The skip operator is used to skip some values emitted by the observable. The number of values to be skipped is provided as an argument.

signature:
skip(count: number): MonoTypeOperatorFunction

count: number of values to be skipped provided as an argument
MonoTypeOperatorFunction: it is a function that returns the output values as an observable

A simple example demonstrating the use of the skip operator is given below.

```
const source = of(1, 2, 3, 4, 5, 6);
const result = source.pipe(skip(3));
result.subscribe(val => console.log(val));
```

Here we have taken an observable through the skip operator and skipped three values. So in the output we get the last three values as 4,5,6.

*** If we pass 0 as argument to the skip operator all the values get emitted.
***If we pass -1 as argument, still all the emitted values from the source observable get printed onto the console.

switchMap operator

The switchMap operator projects each source value to an Observable which is merged in the output Observable and it emits values only from the most recently projected Observable.

Signature:
switchMap(project: Function, resultSelector, outerIndex: number, innerIndex: number) => R): OperatorFunction

project: It is a function that takes the emitted values from the source observable and transforms them.
resultSelector: optional value, default is undefined.
OperatorFunction: It is a function that returns an observable output stream.

A sample code snippet for the switchMap operator is given below.

```
    const source = of(1, 2, 3, 4, 5);
    const result = source.pipe(switchMap(() =>
interval(1000).pipe(take(4))));
    result.subscribe(val => console.log(val));
```

The output for the above code snippet will be values from the interval operator which gets merged at the last and it will be as follows.

```
    // 0
    // 1
    // 2
    // 3
```

switchMapTo operator

The switchMapTo operator projects each source value to the same Observable which is flattened multiple times.
This operator maps each source value to the given innerObservable and flattens each value and makes it as a single observable which is the output observable.

Signature:
switchMapTo(innerObservable: any, resultSelector): OperatorFunction

A sample code snippet for demonstrating the use of the switchMap operator is given below.

```
const source= of(1, 2, 3, 4, 5);
const result = source.pipe(switchMapTo(interval(1000).pipe(take(4))));
result.subscribe(val => console.log(val));
```

The output of the above operation will be as follows.

```
// 0
// 1
// 2
// 3
```

take operator

Emits only the first count values emitted by the source observable. In other words it executes only the first emitted values from the source observable as specified by the provided parameter. The take operator is often used alongside the pipe operator.

Signature:

take(count: number): MonoTypeOperatorFunction
MonoTypeOperatorFunction<T>: An Observable that emits only the first count values emitted by the source Observable, or all of the values from the source if the source emits fewer than count values.

Whatever be the count of emitted values, it takes only the first count of values after subscription.

```
const source = of(1, 2, 3, 4, 5, 6, 7);
const takeFive = source.pipe(take(5));
takeFive.subscribe((val) => {
  console.log(val);
});
```

```
// 1
// 2
// 3
// 4
// 5
```

*** If count = 0, then it does not emit any values.
***If count < 0, then it throws an out of range error.

takeLast operator

The takeLast operator emits the last count values emitted by the source observable.When the source observable completes, it emits the last count values from the end of the stream.

signature:
takeLast(count: number): MonoTypeOperatorFunction

MonoTypeOperatorFunction: An Observable that emits at most the last count values emitted by the source Observable.

If the observable emits fewer values than the count value, then all of the values are emitted. This operator waits for the complete notification from the source in order to emit the next values to make sure only the counted last values are emitted.The values are emitted synchronously.

```
const source = of(1, 2, 3, 4, 5, 6, 7);
const takeFive = source.pipe(takeLast(3));
takeFive.subscribe((val) => {
  console.log(val);
});
```

Here since we have provided the count value as 3, the output will be:

```
// 5
// 6
// 7
```

If we pass the count value negative, we get an out of range error.

takeUntil operator

The takeUntil operator emits values until a notifier which itself is an observable emits a value thus stopping the takeUntil observable.

A simple code example of the takeUntil operator is given below.

```
const source = interval(1000);
const clickEvents = fromEvent(document, 'click');
const ob$ = source.pipe(takeUntil(clickEvents));
ob$.subscribe(val => console.log(val));
```

The above code continues to emit values from the source observable, till the user clicks on the browser body. Once the user clicks, the values stop being emitted.

The takeUntil operator is used to unsubscribe a subscription.It has the following advantages while unsubscribing to an existing subscription.

- no need of manual unsubscription
- no need to deal with messy unsubscription scattered around the file
- Complexity is less
- clean approach

takeWhile operator

The takeWhile operator emits values emitted by the source Observable so long as each value satisfies the given predicate, and then completes as soon as this predicate is not satisfied. Just like while loop, it emits value till a condition is being met.When the condition becomes false, it does not emit value and completes.

Signature:
takeWhile(predicate: Function, inclusive: boolean = false): MonoTypeOperatorFunction

predicate:it is a function that evaluates an emitted value and returns boolean.
inclusive: returns the value that made the observable complete or predicate false.
Its default value is false.

```
const values$ = of(1, 2, 3, 4, 5, 6);
const result = values$.pipe(takeWhile(val => {
  return val < 4;
}));
result.subscribe(val => console.log(val));
```

Here we have taken the predicate function to be testing if the emitted value is less than 4. So it is quite evident that the above will return values which are less than 4.

So the output will be:

```
// 1
// 2
// 3
```

Next we reverse the condition and do it like the following.

```
const values$ = of(1, 2, 3, 4, 5, 6);
const result = values$.pipe(takeWhile(val => {
  return val > 4;
}));
result.subscribe(val => console.log(val));
```

Here there will be no output as the first value itself fails to pass the condition.

throttle operator

Emits a value from the source Observable, then ignores subsequent source values for a duration determined by another Observable, then repeats this process.

Signature:
throttle(durationSelector: (value: T) => SubscribableOrPromise<any>, config: ThrottleConfig = defaultThrottleConfig): MonoTypeOperatorFunction<T>

durationSelector: function that computes the silence duration for each subsequent emission of the observable value.
config: Optional. Default is defaultThrottleConfig.

A simple code for the throttle operator is as follows.

```
const clicks = fromEvent(document, 'click');
 const result = clicks.pipe(throttle(ev => interval(10000)));
 result.subscribe(x => console.log(x));
```

It logs the mouse event on first click on the document, then waits for 10 seconds(10000ms) before emitting the mouse event.

throttleTime operator

It emits a value from the source Observable, then ignores subsequent source values for duration milliseconds, then repeats this process.

Signature:
throttleTime<T>(duration: number, scheduler: SchedulerLike = async, config: ThrottleConfig = defaultThrottleConfig): MonoTypeOperatorFunction<T>

duration: time to wait before emitting the next value after emitting the last value
scheduler: optional parameter, default is async
config: optional, default is defaultThrottleConfig

A simple example of the operator is given below.

```
const source = interval(1000).pipe(take(10));
const throttle$ = source.pipe(throttleTime(5000));
throttle$.subscribe(val => console.log(val));
```

Here the source operator waits for 5000ms to emit the next value. So the output will be:

```
// 0
// 6
```

We can also configure it to accept leading and trailing values as follows.

```
const interval$ = interval(1000);
const ob$ = interval$.pipe(throttleTime(
  5000,
  asyncScheduler,
  { trailing: true }
));
ob$.subscribe(val => console.log(val));
```

Here the output will be 5 11 and 17… etc.

For capturing leading values we can do the following.

```
const interval$ = interval(1000);
const ob$ = interval$.pipe(throttleTime(
  5000,
  asyncScheduler,
  { trailing: false, leading: true }
));
ob$.subscribe(val => console.log(val));
```

Here we have made the leading flag to be true. So the output will be 6, 12, 18 and so on.

throwError operator

Creates an Observable that emits no items to the Observer and immediately emits an error notification.

Signature:
throwError(error: any, scheduler?: SchedulerLike): Observable<never>

error: The particular Error to pass to the error notification.

It returns an error Observable: emits only the error notification using the given error argument.

A sample code example for the throwError operator is given below.

```
const ob$ = throwError('Some error occured');
ob$.subscribe(x => console.log(x));
```

timer operator

Creates an Observable that starts emitting after an dueTime and emits ever increasing numbers after each period of time thereafter.

Signature:

timer(dueTime: number | Date = 0, periodOrScheduler, scheduler?: SchedulerLike): Observable

dueTime: initial delay time to emit the first observable data..till delayTime is elapsed the observable does not emit any value.the default value is 0.
periodOrScheduler: period or interval after which the next sequence of values are to be emitted.

timer returns an Observable that emits an infinite sequence of ascending integers, with a constant interval of time, period of your choosing between those emissions. The first emission happens after the specified dueTime. The initial delay may be a Date. By default, this operator uses the async SchedulerLike to provide a notion of time, but you may pass any SchedulerLike to it. If period is not specified, the output Observable emits only one value, 0. Otherwise, it emits an infinite sequence.

A code example for the timer operator is given below.

```
const timer$ = timer(10000, 1000);
timer$.subscribe(val => {
  console.log(val);
});
```

If we don't provide the second parameter i.e the interval to emit the values, the observable will emit the first value i.e 0 and it will complete.

window operator

The window operator branches out source observable values in nested form whenever another observable emits values. It clubs the already emitted values from the source observable when the window observable emits and repeats this process.

Signature:
window(windowBoundaries: Observable<any>): OperatorFunction

windowBoundaries: It is an observable that ends the old window boundary and starts a new boundary upon the emission of which the values emitted from the source observable are grouped and nested.

OperatorFunction: It is a function that returns an output observable.

```
    const sec = interval(1000); // Returns a number every second
    const result = sec.pipe(
      window(interval(5000)), //  the windowsBoundary observable that
emits every 5s
      map(win => win.pipe(take(2)))
    );
    result.subscribe(val => console.log(val));
```

The above code emits an observable every 5s. The output of the above code block will be as follows.

```
AnonymousSubject {_isScalar: false, observers: Array(0), closed: false,
isStopped: false, hasError: false, …}
    closed: false
    destination: Subject {_isScalar: false, observers: Array(0), closed:
false, isStopped: false, hasError: false, …}
    hasError: false
    isStopped: false
    observers: []
    operator: TakeOperator {total: 2}
    source: Subject {_isScalar: false, observers: Array(0), closed: false,
isStopped: false, hasError: false, …}
    thrownError: null
    _isScalar: false
    __proto__: Subject
```

windowCount operator

The windowCount operator branches out the values from the source observable based on the windowSize parameter. When the windowSize number of values are emitted from the source observable, it creates a new branch and starts the process all over again.

Signature:
windowCount(windowSize: number, startWindowEvery: number = 0): OperatorFunction

windowSize: It is the maximum number of values emitted by each window, or the maximum number of values accomodated in the same window.
startWindowEvery: Interval at which to start a new window. Default value is 0.

A simple code example of the windowCount operator is given below.

```
const source = of(1, 2, 3, 4, 5, 6);
const result = source.pipe(
  windowCount(2),
  map(win => win.pipe(skip(1))), // skip first of every 2 values
  mergeAll()                     // flatten the observables
);
result.subscribe(val => console.log(val));
```

The output of the above will come in the below format.

```
// 2
// 4
// 6
```

windowTime operator

The windowTime operator is very similar to the windowCount operator but the difference lies in the fact that after the elapse of the given time the observable nests out.The time interval or period after which the source values are nested out are passed as a parameter to the windowTime operator.

Signature:
windowTime(windowTimeSpan: number): OperatorFunction

windowTimeSpan: It is the time after which a new window is created
OperatorFunction: It is a function that returns an observable.

A simple code for the windowCount operator is given below.

```
    const source = of(1, 2, 3, 4, 5, 6);
    const result = source.pipe(
      windowTime(1000),
      map(win => win.pipe(take(5))),
      mergeAll()                    // flatten the
Observable-of-Observables
    );
    result.subscribe(val => console.log(val));
```

The output of the above operator will be as follows.

```
    // 1
    // 2
    // 3
    // 4
    // 5
```

For more technical content please visit http://code2stepup.com/

THANK YOU