RxJS Operators

Operators are the important part of RxJS. RxJS provides a huge collection of operators. There are over a 100+ operators in RxJS that you can use with observables. An operator is a pure function that takes a observable as an input and provide the output in also in the form of an observable.

In simple words, we can say that operators are simply methods that can be used with Observables or Subjects of [RxJS](https://www.javatpoint.com/rxjs). RxJS operators facilitate us to change the original observable in some manner and return a new observable. The operators do not change the existing observable. They simply modify it and return a new one. Operators are known as the type of functions that do not modify the variables outside of its scope. There are mainly two types of RxJS operators:

**Static Operators:** The static operators are generally used to create observables. These types of operators can be found mainly under the creation operators.

**Instance Operators:** The instance operators are methods on observable instances. These are used to account the majority of RxJS operators that we have used.

How to work with Operator?

As we know that operators are pure functions which take in observable as the input and the provided output is also an observable.

We need a pipe() method to work with operators. Let's see an example of pipe() function.

**Syntax of using pipe() Method**

1. let obs = of(1,2,3); // an observable
2. obs.pipe(
3. operator1(),
4. operator2(),
5. operator3(),
6. )

In the above syntax, we have created a observable using **of()** method. This method takes in values 1, 2 and 3. On this observable, you can perform many operations using different types of operators using pipe() method as the above syntax.

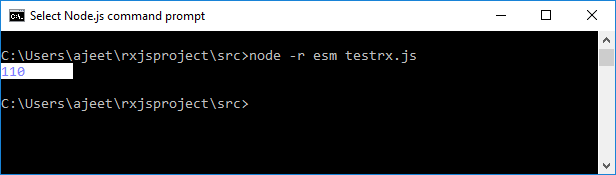
Let's see a working example of operators to understand the concept clearly. In this example, we are using a filter operator to filter the even numbers and then we will use a reduce() operator that will add all the even values and give the result.

**Example 1:**

1. **import** { of } from 'rxjs';
2. **import** { map, reduce, filter } from 'rxjs/operators';
3. let test1 = of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20);
4. let case1 = test1.pipe(
5. filter(x => x % 2 === 0),
6. reduce((acc, one) => acc + one, 0)
7. )
8. case1.subscribe(x => console.log(x));

After the execution of the program by using the **node -r esm testrx.js** command, we will see the following result.

**Output:**



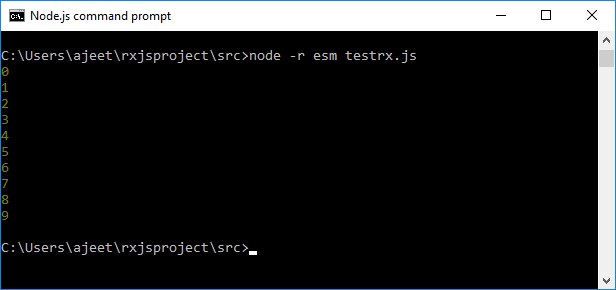
You can see that 110 is the total of all the even values of the above numbers.

**Example2:**

Let's see another example of interval operator. This operator will create an Observable every time for the given time interval. In the following example, the program will return an observable after every 1000 millisecond.

1. **import** { interval } from 'rxjs';
2. **import** { take } from 'rxjs/operators';
3. let test = interval(1000);
4. let case1 = test.pipe(take(10));
5. case1.subscribe(x => console.log(x));

**Output:**



Categorization of RxJS Operators

According to the official documentation for categories of RxJS operators, you can find the following categories of operators:

* [Creation](https://www.javatpoint.com/rxjs-operators#Creation)
* [Mathematical](https://www.javatpoint.com/rxjs-operators#Mathematical)
* [Join or Combination](https://www.javatpoint.com/rxjs-operators#Join-or-Combination)
* [Transformation](https://www.javatpoint.com/rxjs-operators#Transformation)
* [Filtering](https://www.javatpoint.com/rxjs-operators#Filtering)
* [Utility](https://www.javatpoint.com/rxjs-operators#Utility)
* [Conditional](https://www.javatpoint.com/rxjs-operators#Conditional)
* [Multicasting](https://www.javatpoint.com/rxjs-operators#Multicasting)
* [Error handling](https://www.javatpoint.com/rxjs-operators#Error-handling)
* Boolean
* Aggregate

Creation Operators

Following is the list of operators that can be used as creation operators:

|  |  |  |
| --- | --- | --- |
| **Index** | **Operators** | **Description** |
| 1. | [ajax](https://www.javatpoint.com/rxjs-ajax-creation-operator) | The **ajax** operator is used to make an ajax request for the given URL. |
| 2. | [from](https://www.javatpoint.com/rxjs-from-creation-operator) | The **from** operator is used to create an observable from an array, an array-like object, a promise, an iterable object, or an observable-like object. |
| 3. | [fromEvent](https://www.javatpoint.com/rxjs-fromevent-creation-operator) | The **fromEvent** operator is used to give output as an observable that is to be used on elements that emit an event for example buttons, clicks, etc. |
| 4. | [fromEventPattern](https://www.javatpoint.com/rxjs-fromeventpattern-creation-operator) | The **fromEventPattern** operator is used to create an observable from the input function that is used to register event handlers. |
| 5. | [interval](https://www.javatpoint.com/rxjs-interval-creation-operator) | The **interval** operator is used to create an Observable for every time for the given time interval. |
| 6. | [of](https://www.javatpoint.com/rxjs-of-creation-operator) | The **of** operator is used to take in the arguments passed and convert them to observable. |
| 7. | [range](https://www.javatpoint.com/rxjs-range-creation-operator) | The **range** operator is used to create an Observable that gives you a sequence of numbers based on the range provided. |
| 8. | [throwError](https://www.javatpoint.com/rxjs-throwerror-creation-operator) | The **throwError** operator is used to create an observable that notifies an error. |
| 9. | [timer](https://www.javatpoint.com/rxjs-timer-creation-operator) | The **timer** operator is used to create an observable that emits the value after the timeout and the value will keep increasing after each call. |
| 10. | [iif](https://www.javatpoint.com/rxjs-iif-creation-operator) | The **iif** operator is used to decide which Observable will be subscribed. |

Mathematical Operators

Following is the list of operators that can be used as mathematical operators:

|  |  |  |
| --- | --- | --- |
| **Index** | **Operators** | **Description** |
| 1. | [count](https://www.javatpoint.com/rxjs-count-mathematical-operator) | The count operator is used to take an observable with values and convert it into an observable that gives a single value. |
| 2. | [max](https://www.javatpoint.com/rxjs-max-mathematical-operator) | The max method is used to take an observable with all values and return an observable with the maximum value from the list. |
| 3. | [min](https://www.javatpoint.com/rxjs-min-mathematical-operator) | The min method is used to take an observable with all values and return an observable with the minimum value from the list. |
| 4. | [reduce](https://www.javatpoint.com/rxjs-reduce-mathematical-operator) | In reduce operator, an accumulator function is used on the input observable. It returns the accumulated value in an observable form, with an optional seed value passed to the accumulator function. So, the reduce function takes two arguments, one accumulator function, and second the seed value. |

Join Operators

Following is the list of operators that can be used as join operators:

|  |  |  |
| --- | --- | --- |
| **Index** | **Operators** | **Description** |
| 1. | [concat](https://www.javatpoint.com/rxjs-concat-join-operator) | The concat operator is used to sequentially emit the Observable given as input and proceed to the next one. |
| 2. | [forkJoin](https://www.javatpoint.com/rxjs-forkjoin-join-operator) | The forkJoin operator is used to take in an array or dict object as an input and wait for the observable to complete and return the last values emitted from the given observable. |
| 3. | [merge](https://www.javatpoint.com/rxjs-merge-join-operator) | The merge operator is used to take in the input observable and emit all the values from the observable and emit one single output observable. |
| 4. | [race](https://www.javatpoint.com/rxjs-race-join-operator) | The race operator is used to return an observable that will be a mirror copy of the first source observable. |

Transformation Operators

Following is the list of operators that can be used as transformation operators:

|  |  |  |
| --- | --- | --- |
| **Index** | **Operators** | **Description** |
| 1. | [buffer](https://www.javatpoint.com/rxjs-buffer-transformation-operator) | The buffer operator is used to be operated on an observable and take in argument as an observable. It starts buffering the values emitted on its original observable array and emits when the observable taken as an argument, emits. Once the observable taken as arguments emits, the buffer is reset and starts buffering again on original till the input observable emits, and the same process repeats itself. |
| 2. | [bufferCount](https://www.javatpoint.com/rxjs-buffercount-transformation-operator) | The buffercount operator is used to collect the values from the observable on which it is called and emit the same when the buffer size is given to buffercount matches. |
| 3. | [bufferTime](https://www.javatpoint.com/rxjs-buffertime-transformation-operator) | The bufferTime operator is similar to bufferCount. It is used to collect the values from the observable on which it is called and emit the bufferTimeSpan. It takes in 1 argument, i.e., bufferTimeSpan. |
| 4. | [bufferToggle](https://www.javatpoint.com/rxjs-buffertoggle-transformation-operator) | The bufferToggle operator is used to take two arguments, openings and closingSelector. The opening arguments are subscribable or a promise to start the buffer. The second argument closingSelector is again subscribable or promises an indicator to close the buffer and emit the values collected. |
| 5. | [bufferWhen](https://www.javatpoint.com/rxjs-bufferwhen-transformation-operator) | The bufferWhen operator is used to give the values in the array form, it takes in one argument as a function that will decide when to close, emit and reset the buffer. |
| 6. | [expand](https://www.javatpoint.com/rxjs-expand-transformation-operator) | The expand operator is used to take in a function as an argument applied on the source observable recursively and on the output observable. The final value is observable. |
| 7. | [groupBy](https://www.javatpoint.com/rxjs-groupby-transformation-operator) | Te groupBy operator is used to group the output according to specific conditions, and these group items are emitted as GroupedObservable. |
| 8. | [map](https://www.javatpoint.com/rxjs-map-transformation-operator) | In the map operator's case, a project function is applied on each value on the source Observable, and the same output is emitted as an Observable. |
| 9. | [mapTo](https://www.javatpoint.com/rxjs-mapto-transformation-operator) | In the mapTo operator, a constant value is given as output along with the Observable every time the source Observable emits a value. |
| 10. | [mergeMap](https://www.javatpoint.com/rxjs-mergemap-transformation-operator) | In the mergeMap operator, a project function is applied on each source value, and the output of it is merged with the output Observable. |
| 11. | [switchMap](https://www.javatpoint.com/rxjs-switchmap-transformation-operator) | The switchMap operator is used to apply a project function on each source value. The output of it is merged with the output Observable, and the value given is the most recent projected Observable. |
| 12. | [window](https://www.javatpoint.com/rxjs-window-transformation-operator) | The window operator is used to take an argument windowboundaries which is an observable and give back a nested observable whenever the given windowboundaries emits. |

Filtering Operators

Following is the list of operators that can be used as filtering operators:

|  |  |  |
| --- | --- | --- |
| **Index** | **Operators** | **Description** |
| 1. | [debounce](https://www.javatpoint.com/rxjs-debounce-filtering-operator) | In the debounce operator, a value is emitted from the source Observable after a while, and the emission is determined by another input given as Observable or promise. |
| 2. | [debounceTime](https://www.javatpoint.com/rxjs-debouncetime-filtering-operator) | The debounceTime operator is used to emit value from the source observable only after the time is complete. |
| 3. | [distinct](https://www.javatpoint.com/rxjs-distinct-filtering-operator) | This operator is used to give all the values from the source observable that are distinct compared with the previous values. |
| 4. | [elementAt](https://www.javatpoint.com/rxjs-elementat-filtering-operator) | This operator is used to give a single value from the source observable based upon the given index. |
| 5. | [filter](https://www.javatpoint.com/rxjs-filter-filtering-operator) | This operator is used to filter the values from source Observable based on the predicate function given. |
| 6. | [first](https://www.javatpoint.com/rxjs-first-filtering-operator) | The first operator is used to give the first value emitted by the source Observable. |
| 7. | [last](https://www.javatpoint.com/rxjs-last-filtering-operator) | The last operator is used to give the last value emitted by the source Observable. |
| 8. | [ignoreElements](https://www.javatpoint.com/rxjs-ignoreelements-filtering-operator) | The ignoreElements operator is used to ignore all the values from the source Observable and only execute calls to complete or error callback functions. |
| 9. | [sample](https://www.javatpoint.com/rxjs-sample-filtering-operator) | The sample operator is used to give the most recent value from the source Observable, and the output will depend upon the argument passed to it emits. |
| 10. | [skip](https://www.javatpoint.com/rxjs-skip-filtering-operator) | The skip operator is used to give back an observable that will skip the first occurrence of count items taken as input. |
| 11. | [throttle](https://www.javatpoint.com/rxjs-throttle-filtering-operator) | The throttle operator is used to give output and ignore values from the source observable for the time determined by the input function taken as an argument, and the same process will be repeated. |

Utility Operators

Following is the list of operators that can be used as utility operators:

|  |  |  |
| --- | --- | --- |
| **Index** | **Operators** | **Description** |
| 1. | [tap](https://www.javatpoint.com/rxjs-tap-utility-operator) | The tap operator contains the output, same as the source observable, and it can be used to log the values to the user from the observable. |
| 2. | [delay](https://www.javatpoint.com/rxjs-delay-utility-operator) | The delay operator is used to delay the values emitted from the source observable based on the timeout given. |
| 3. | [delayWhen](https://www.javatpoint.com/rxjs-delaywhen-utility-operator) | The delayWhen operator is used to delay the values emitted from the source observable based on the timeout from another observable taken as input. |
| 4. | [observeOn](https://www.javatpoint.com/rxjs-observeon-utility-operator) | The observeOn operator is based on the input scheduler. It is used to reemit the notifications from the source observable. |
| 5. | [subscribeOn](https://www.javatpoint.com/rxjs-subscribeon-utility-operator) | The subscribeOn operator helps to asynchronous subscribes to the source observable based on the scheduler taken as input. |
| 6. | [timeInterval](https://www.javatpoint.com/rxjs-timeinterval-utility-operator) | The timeInterval operator is used to return an object containing current value, and the time elapsed between the current and previous value calculated using the scheduler input. |
| 7. | [timestamp](https://www.javatpoint.com/rxjs-timestamp-utility-operator) | The timestamp operator is used to return the timestamp along with the value emitted from the source observable. It tells about the time when the value was emitted. |
| 8. | [timeout](https://www.javatpoint.com/rxjs-timeout-utility-operator) | The timeout operator is used to throw an error if the source observable does not emit a value after the given timeout. |
| 9. | [toArray](https://www.javatpoint.com/rxjs-toarray-utility-operator) | The toArray operator is used to accumulate all the source values from the observable and provide their outputs as an array when the source completes. |

Conditional Operators

Following is the list of operators that can be used as conditional operators:

|  |  |  |
| --- | --- | --- |
| **Index** | **Operators** | **Description** |
| 1. | [defaultIfEmpty](https://www.javatpoint.com/rxjs-defaultifempty-conditional-operator) | The defaultIfEmpty operator is used to return a default value if the source observable is empty. |
| 2. | [every](https://www.javatpoint.com/rxjs-every-conditional-operator) | The every operator is used to return an observable based on the input function, which satisfies the condition on each of the values on source observable. |
| 3. | [find](https://www.javatpoint.com/rxjs-find-conditional-operator) | The find operator is used to return the observable when the source observable's first value satisfies the condition for the predicate function taken as input. |
| 4. | [findIndex](https://www.javatpoint.com/rxjs-findindex-conditional-operator) | The findIndex operator is based on the input schedule. It is used to reemit the notifications from the source observable. |
| 5. | [isEmpty](https://www.javatpoint.com/rxjs-isempty-conditional-operator) | The isEmpty operator gives the output "True" if the input observable goes for complete callback without emitting any values and "False" if the input observable emits any values. |

Multicasting Operators

Following is the list of operators that can be used as multicasting operators:

|  |  |  |
| --- | --- | --- |
| **Index** | **Operators** | **Description** |
| 1. | [multicast](https://www.javatpoint.com/rxjs-multicast-multicasting-operator) | A multicast operator shares the single subscription created with other subscribers. The params that multicast takes in is a subject or a factory method that returns a ConnectableObservable that has a connect method. The connect method is called to subscribe. |
| 2. | [publish](https://www.javatpoint.com/rxjs-publish-multicasting-operator) | The publish operator is used to give back ConnectableObservable and use connect method to subscribe to the observables. |
| 3. | [publishBehavior](https://www.javatpoint.com/rxjs-publishbehaviour-multicasting-operator) | The publishBehaviour makes use of BehaviourSubject, and returns ConnectableObservable. The connect method is used to subscribe to the observable created. |
| 4. | [publishLast](https://www.javatpoint.com/rxjs-publishlast-multicasting-operator) | The publishBehaviour operator makes use of AsyncSubject, and returns ConnectableObservable. The connect method is used to subscribe to the observable created. |
| 5. | [publishReplay](https://www.javatpoint.com/rxjs-publishreplay-multicasting-operator) | The publishReplay operator uses behavior subject where it can buffer the values, replay the same to the new subscribers, and return ConnectableObservable. The connect method is used to subscribe to the observable created. |
| 6. | [share](https://www.javatpoint.com/rxjs-share-multicasting-operator) | The share operator is an alias for a multicast operator. The only difference is that you don't have to called the connect method manually to start the subscription. |

Error Handling Operators

Following is the list of operators that can be used as error handling operators:

|  |  |  |
| --- | --- | --- |
| **Index** | **Operators** | **Description** |
| 1. | [catchError](https://www.javatpoint.com/rxjs-catcherror-error-handling-operator) | The catchError operator is used to take care of catching errors on the source observable by returning a new observable or an error. |
| 2. | [retry](https://www.javatpoint.com/rxjs-retry-error-handling-operator) | The retry operator is used to take care of retrying back on the source observable if there is an error. The retry is done according to the given input count. |
| 3. | [retryWhen](https://www.javatpoint.com/rxjs-retrywhen-error-handling-operator) |  |