Observable is the foundation of RxJS library but it is mostly known for its different operators that allows us to easily compose complex asynchronous code in a declarative manner.

RxJS operators are pure functions that enable a functional programming style of dealing with collections. There are mainly two types of operators.

**1. Pipeable Operators:-**A**pipeable operator**is a pure function that takes an observable as its input and returns another observable. It does not change anything in the original observable.

They are further sub categorized into different types such as**Join Operators, Transformation operators, Filtering Operators, etc...**

The syntax for using a pipeable operator is:

1. observableInstance.pipe(operator())

**example:-** filter(), map(), mergeMap(), etc..

**2. Creation Operators:-**These operators are **standalone**functions used to create a new Observable. Some of the creation operators are**of(), from(), etc..**

Next, we will learn about these operators in detail.

**of() method:-**

**of()** method is used to create an observable using a given set of values. It takes some values like numbers, strings, arrays, and converts them into an Observable sequence.

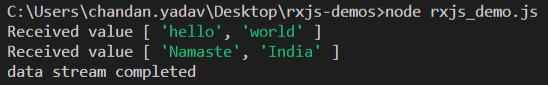
**Syntax:-**

1. of<T>(...args: (SchedulerLike | T)[]): Observable<T>

**Example:-**

1. import { of } from 'rxjs';
2. let observable = of(["hello", "world"], ['Namaste', 'India']);
3. observable.subscribe(
4. data => console.log("Received value", data),
5. error => console.log("error", error),
6. () => console.log("data stream completed")
7. )

**Output:-**



**from() method:-**

**from()** method is used to create an observable from an array, array-like object, a Promise, an iterable object or an Observable like object. We cannot use a number or a boolean value directly to convert into an observable. A string value is treated as an array of characters when passed to **from()**method.

**Syntax:-**

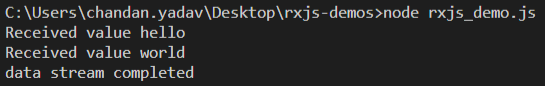
1. from<T>(input: any, scheduler?: SchedulerLike): Observable<T>

**input** is the data to be converted to observables, **schedulerLike**is optional and is **undefined**by default.

**Example:-**

1. import { from } from 'rxjs';
2. let observable = from(["hello", "world"]);
3. observable.subscribe(
4. data => console.log("Received value", data),
5. error => console.log("error", error),
6. () => console.log("data stream completed")
7. )

**Output:-**



**difference between of() and from():-**

**1. of()**can be used to converts any value such as numbers, strings, boolean, as it is to an observables, whereas **from()**accepts only array like values, promises or an iterable objects to convert them into an observable.

1. import { of, from } from 'rxjs';
2. let observable1 = of(100); */\* returns an observable with value 100 \*/*
3. let observable2 = from(100); */\* throws type error \*/*

**2. of()**treats a string as a value and converts to observable as it is where as**from()** treats string value as an array of characters and converts them to observable of characters.

1. import { of, from } from 'rxjs';
2. let observable1 = of("hello"); */\* returns an observable with value "hello" \*/*
3. let observable2 = from("hello"); */\* returns an observable with values "h", "e", "l", "l", "o" \*/*

Next, we will see how to create an observable with error values.

# throwError() method:-

It creates an Observable which does not emit any values to the observer, instead unsubscribes from the observable by emitting an error notification. It takes an **error** that needs to be passed when the observable is subscribed by the subscriber.

**Syntax:-**

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

**scheduler**is **optional**and its value is **undefined** by default

**Example:-**

1. import { throwError } from 'rxjs';
2. let observable = throwError({ "message": "Something Broke!!" });
3. observable.subscribe(
4. data => console.log("Received value", data),
5. error => console.log("error", error),
6. () => console.log("data stream completed")
7. );

**Output:-**

https://academy.onwingspan.com/common-content-store/Shared/Shared/Public/lex_auth_013038515371704320336_shared/web-hosted/assets/rxjsopr3.PNG

Next we will see some of the pipeable operators.

We already know ***Pipeable operators*** are the ones which take an observable as an input and returns another observable as a result without affecting the original observable(s).

As Pipeable operators are also functions, they can be used by invoking like any other function as:

1. operator()(observable)

But in actual applications, we might have to use many such operators together, and hence using normal functions can soon become unreadable:

1. operator4()(operator3()(operator2()(operator1()(observable))))

Observables provide a method called **pipe()**which helps to use multiple operators together in a much readable and maintainable manner:

1. observable.pipe(
2. operator1(),
3. operator2(),
4. operator3(),
5. operator4()
6. )

Based on their usage they are further classified into different categories:

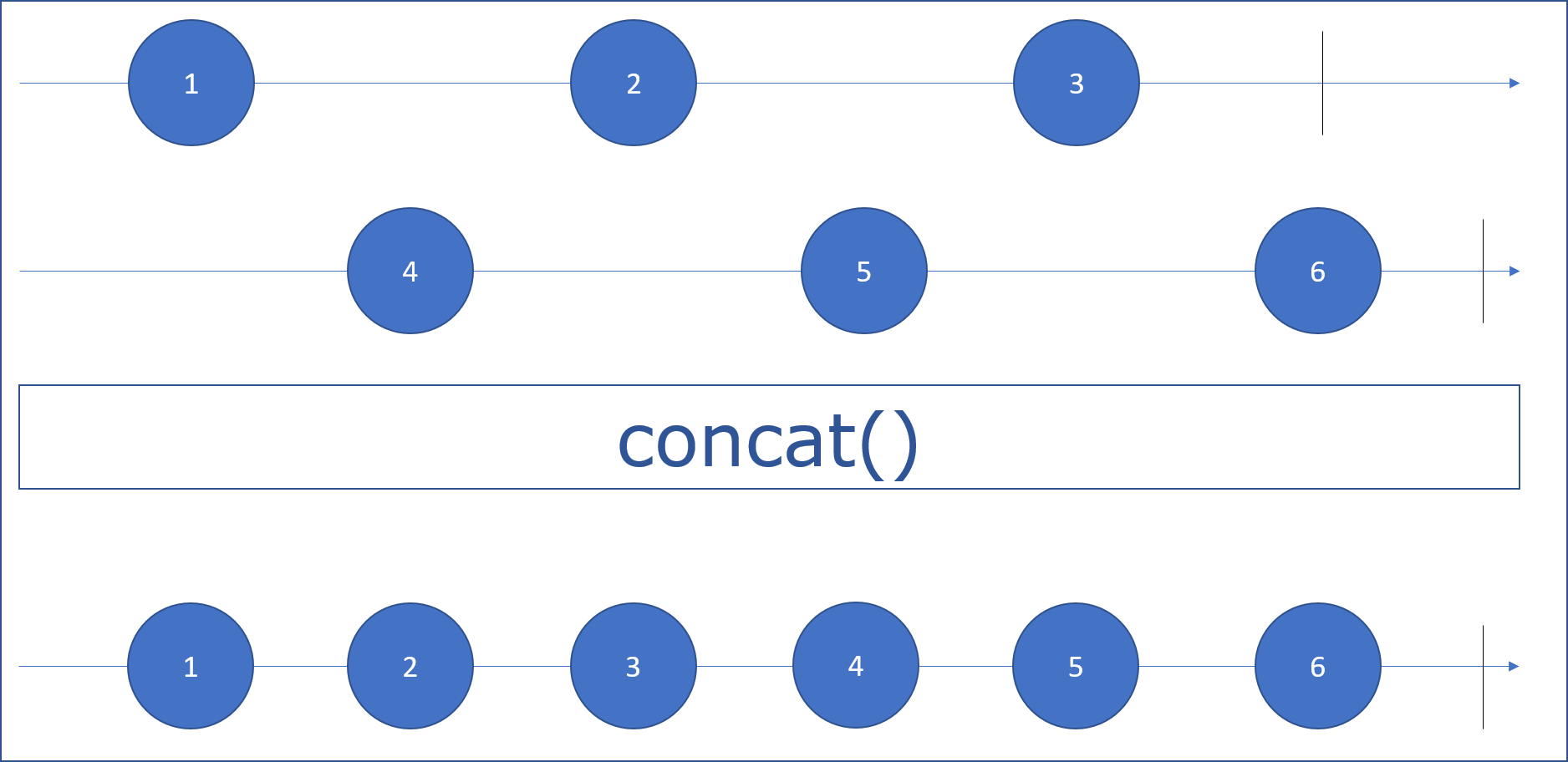
* Join Creation Operators
* Mathematical and Aggregate Operators
* Transformation Operators
* Filterting Operators
* Error Handling Operators
* Conditional and Boolean Operators

Next, we will see some of the commonly used pipeable operators.

Join Creation Operators help us to join two or more already existing observables. Some of the commonly used Join Creation Operators are concat(), merge(), forkJoin() etc.

# concat() method:-

It Concatenates multiple observables together and emits the values from each observable one after the other. It takes an array of observables or the observables directly as a parameter and then returns a single observable. Observe the given marble diagram to understand use of **concat()**method:



**Syntax:-**

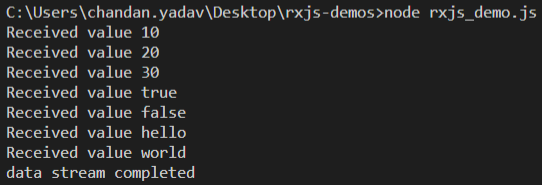
1. concat(...observables): Observable

While execution concat will initially subscribe to the first observable and emit all its values, once it completes, it will move on to the next observable passed and again emit its values. This will continue till all the observables are not completed. This means, at a given instant of time, only one of the observables used in concat will be emitting the values and hence concurrent emission of data will not happen. For concurrent execution of observables we can use the operator merge() method which we will see later.

**Example:-**

1. import { of } from 'rxjs';
2. import { concat } from 'rxjs/operators';
3. let observable1 = of(10, 20, 30);
4. let observable2 = of(true, false);
5. let observable3 = of("hello", "world");
6. let observable = observable1.pipe(concat(observable2, observable3))
7. observable.subscribe(
8. data => console.log("Received value", data),
9. error => console.log("error", error),
10. () => console.log("data stream completed")
11. );

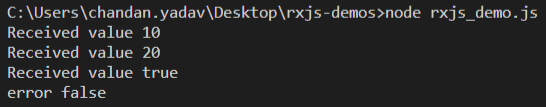
**Output:-**



**Note:-**If an observable in concat completes immediately after it is subscribed, it will be ignored by concat and it will move on to the next observable if there are any but if any observable in the chain throws an error, in the case, concat wont go to the next observable and terminate emitting values immediately.

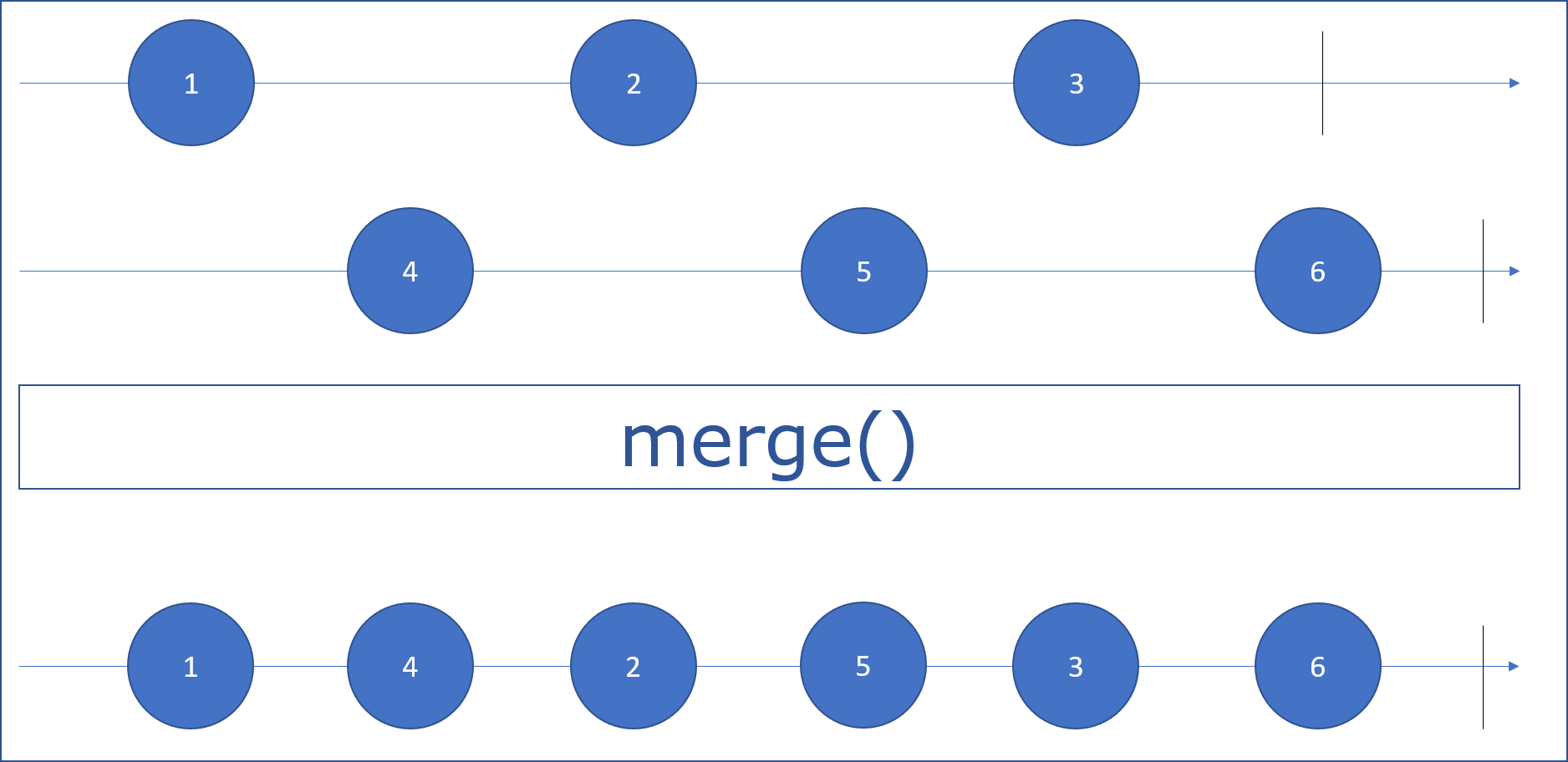
1. import { Observable } from 'rxjs';
2. import { concat } from 'rxjs/operators';
3. let observable1 = new Observable(observer => {
4. observer.next(10);
5. observer.next(20);
6. observer.complete();
7. });
8. let observable2 = new Observable(observer => {
9. observer.next(true);
10. observer.error(false);
11. });
12. let observable3 = new Observable(observer => {
13. observer.next("hello");
14. });
15. let observable = observable1.pipe(concat(observable2, observable3));
16. observable.subscribe(
17. data => console.log("Received value", data),
18. error => console.log("error", error),
19. () => console.log("data stream completed")
20. );

**Output:-**



# merge() method:-

It merges multiple observables together and emits values from each observable concurrently. It takes a set of input observables and creates an output observables that emits values concurrently from the set of input observables. Observe the given marble diagram to understand use of **merge()**method:



**Syntax:-**

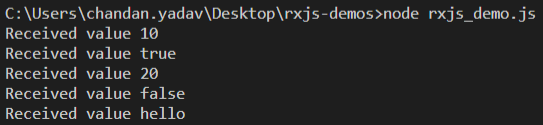
1. merge(...observables): Observable

merge() will subscribe to all the input observables and forward all the values from input observable to the output observable without doing any transformation. The output observable will complete only after all the observables have completed. Any error in input observable, will immediately be emitted from the output observable and it will emitting values.

**Example:-**

1. import { Observable } from 'rxjs';
2. import { merge } from 'rxjs/operators';
3. let observable1 = new Observable(observer => {
4. setTimeout(() => observer.next(10), 1000);
5. setTimeout(() => observer.next(20), 1500);
6. setTimeout(() => observer.complete(), 1500);
7. });
8. let observable2 = new Observable(observer => {
9. setTimeout(() => observer.next(true), 1000);
10. setTimeout(() => observer.next(false), 1500);
11. setTimeout(() => observer.complete(), 1500);
12. });
13. let observable3 = new Observable(observer => {
14. setTimeout(() => {
15. observer.next("hello");
16. observer.complete();
17. }, 1500);
18. });
19. let observable = observable1.pipe(merge(observable2, observable3));
20. observable.subscribe(
21. data => console.log("Received value", data),
22. error => console.log("error", error),
23. () => console.log("data stream completed")
24. );

**Output:-**



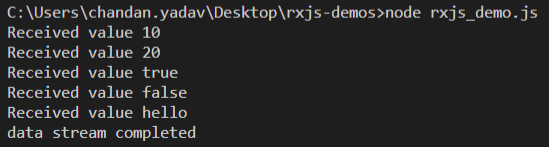
# merge() vs concat():-

merge() and concat() would return exactly the same output if the observables have data stream coming synchronously. But as soon as the data stream becomes asynchronous. merge() will start emitting the data concurrently where as concat() will emit the data from the observables one after the other.

for the same example, if we use concat operator as below:

1. import { Observable } from 'rxjs';
2. import { concat } from 'rxjs/operators';
3. let observable1 = new Observable(observer => {
4. setTimeout(() => observer.next(10), 1000);
5. setTimeout(() => observer.next(20), 1500);
6. setTimeout(() => observer.complete(), 1500);
7. });
8. let observable2 = new Observable(observer => {
9. setTimeout(() => observer.next(true), 1000);
10. setTimeout(() => observer.next(false), 1500);
11. setTimeout(() => observer.complete(), 1500);
12. });
13. let observable3 = new Observable(observer => {
14. setTimeout(() => {
15. observer.next("hello");
16. observer.complete();
17. }, 1500);
18. });
19. let observable = observable1.pipe(concat(observable2, observable3));
20. observable.subscribe(
21. data => console.log("Received value", data),
22. error => console.log("error", error),
23. () => console.log("data stream completed")
24. );

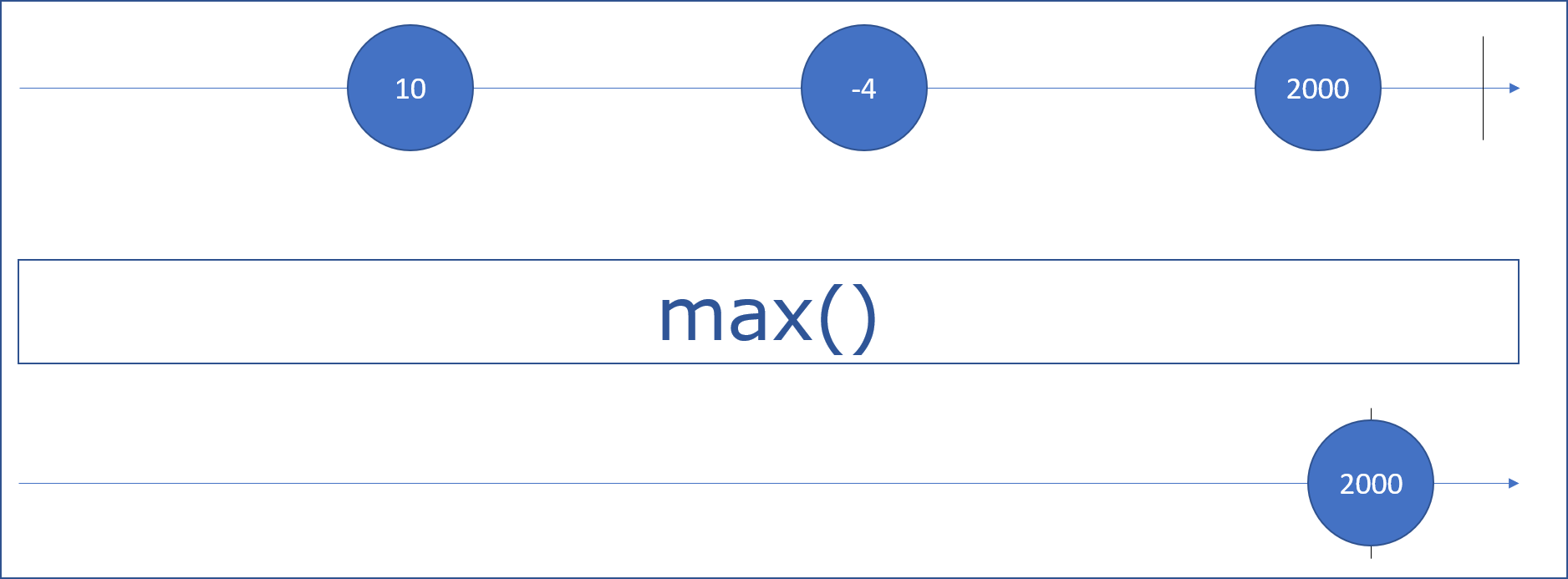
The**output** will be:-



rxjs also provides operators which helps to perform mathematical and arithmetic operations with the observable data stream. These operators are **max(), min(), count()**and **reduce()**

# max():-

This method takes an observable with all the values and returns an observable with the maximum value. It operates on an observable that emits numbers or items that can be compared with a given function and when the source observable completes, max() returns a single value, i.e. highest value. Observe the below marble diagram for **max()** operator:



**Syntax:-**

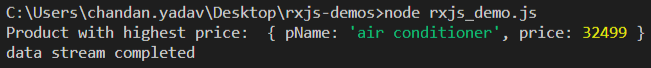
1. max(comparer\_func?: number): Observable

**max()** method takes an optional **comparer\_func**as parameter i.e. a function that filters the values to be considered for max values from the source observable. If we don't provide **comparer\_func,**it will use a default function.

**Example 1:- (with comparer\_func) -**Finding the product with highest price

1. import { of } from 'rxjs';
2. import { max } from 'rxjs/operators';
3. interface Product {
4. price: number,
5. pName: string
6. }
7. let products = of<Product>(
8. { pName: "televison", price: 17999 },
9. { pName: "air conditioner", price: 32499 },
10. { pName: "smartphone", price: 24299 },
11. )
12. let costly\_product = products.pipe(max<Product>((a: Product, b: Product) => a.price < b.price ? -1 : 1));
13. costly\_product.subscribe(
14. (x: Product) => console.log('Product with highest price: ', x),
15. error => console.log("error", error),
16. () => console.log("data stream completed")
17. );

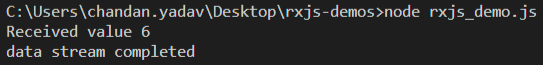
**Output:-**



**Example 2:- (without comparer\_func) -**Finding maximum value from a series of numbers

1. import { of } from 'rxjs';
2. import { max } from 'rxjs/operators';
3. let numbers = of(1, 2, 3, 4, 5, 6);
4. let observable = numbers.pipe(max());
5. observable.subscribe(
6. data => console.log("Received value", data),
7. error => console.log("error", error),
8. () => console.log("data stream completed")
9. );

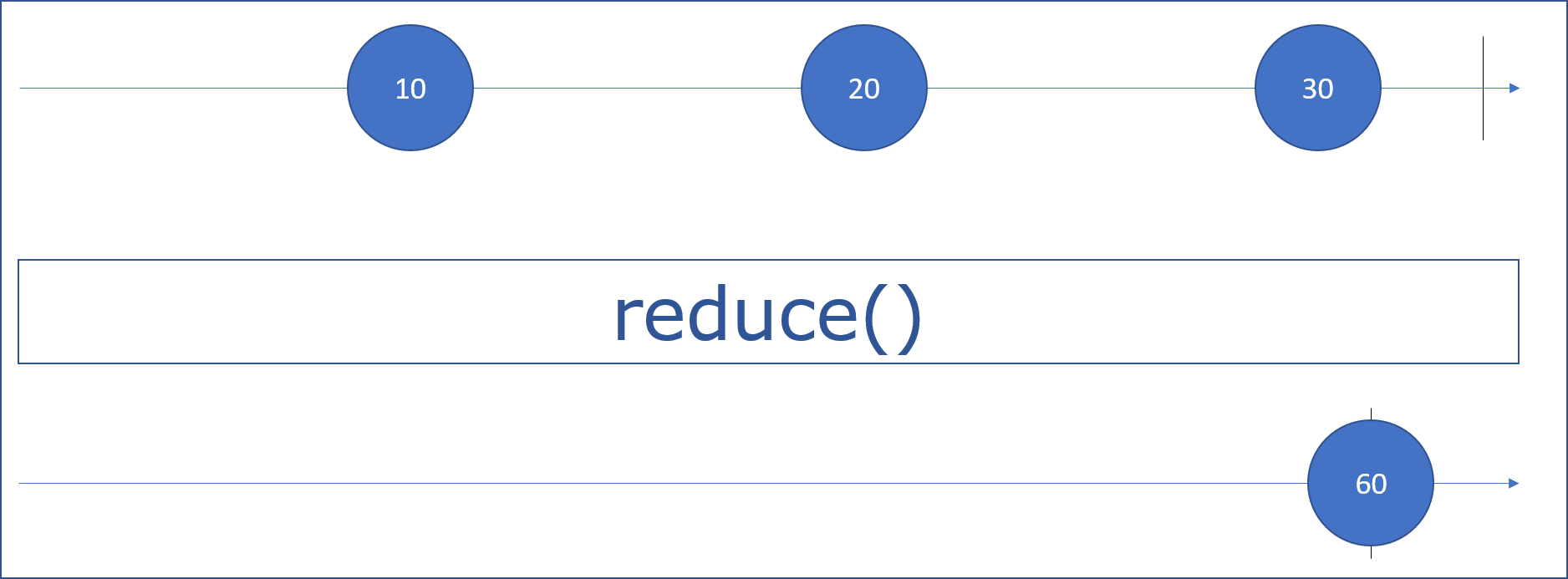
**Output:-**



Similarly, you can try using **min()** operator to find the minimum value from the given observable.

# reduce():-

**reduce()** method combines together all the values emitted by an observable using an accumulator function which knows how to join the new value to the already accumulated values of past. Observe the below marble diagram for reduce() operator:



**Syntax:-**

1. reduce(accumulator\_func, seeder?) : Observable

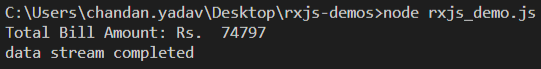
**reduce()**method takes 2 parameters i.e. **accumulator\_func**and a **seeder**value **(optional)**. **reduce()** operator executes the **accumulator\_func**on the observable and returns an accumulated value in the form of an observable. **seeder**is**undefined**by default and acts as the intial value for accumulation.

**Note:-**If **seeder**value is provided, **reduce()** takes that as the initial value, if not provided, it uses the first value of the observable as the **seeder.**

**Example:-**

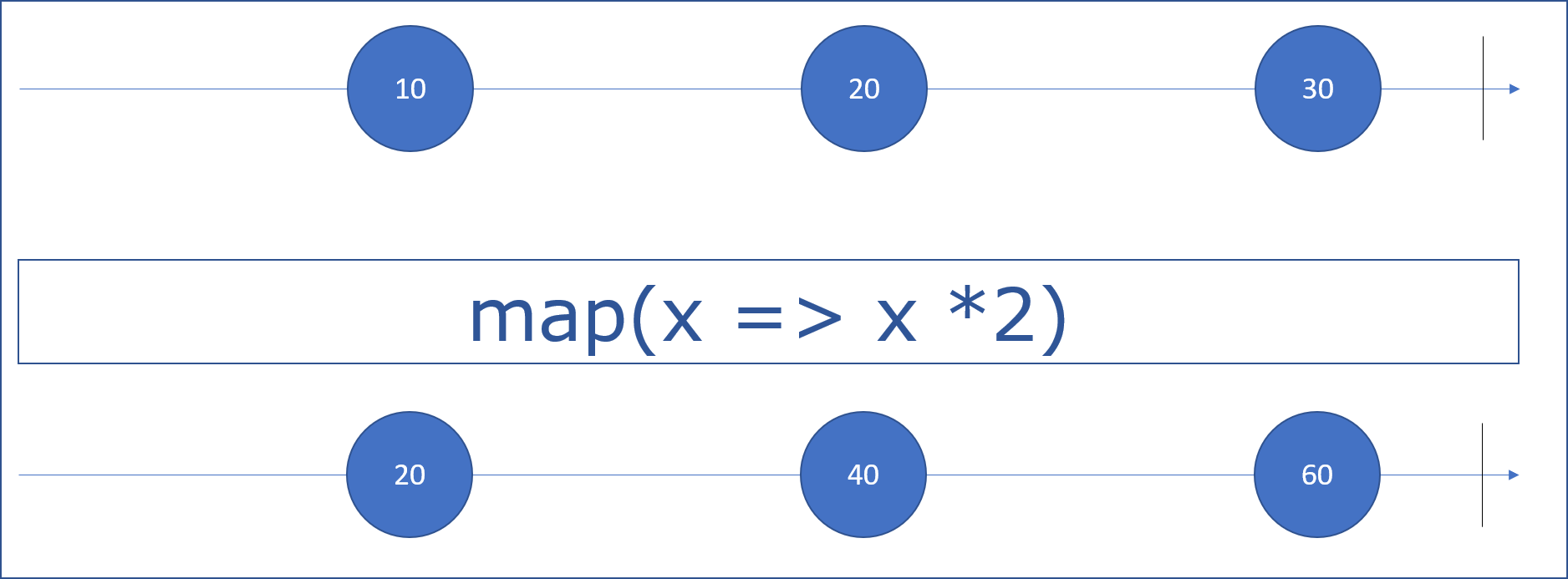
1. import { of } from 'rxjs';
2. import { reduce } from 'rxjs/operators';
3. interface Product {
4. price: number,
5. pName: string
6. }
7. let products = of<Product>(
8. { pName: "televison", price: 17999 },
9. { pName: "air conditioner", price: 32499 },
10. { pName: "smartphone", price: 24299 }
11. )
12. let total\_price = products.pipe(reduce((acc, itemsdet) => acc + itemsdet.price, 0));
13. total\_price.subscribe(
14. x => console.log('Total Bill Amount: Rs. ', x),
15. error => console.log("error", error),
16. () => console.log("data stream completed")
17. );

**Output:-**



# map():-

**map()** operator works the same way like an array map function does. It passes each value through a tranformation function to get an observable with the new transformed values. Observe the given marble diagram for**map()** operator:



**Syntax:-**

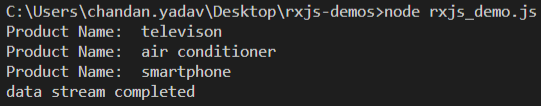
1. map(projectFunction: function): Observable

It takes a **projectFunction**as a parameter and executes it over each value of the input observable and returns a new observable with new values.

**Example:-**

1. import { of } from 'rxjs';
2. import { map } from 'rxjs/operators';
3. interface Product {
4. price: number,
5. pName: string
6. }
7. let products = of<Product>(
8. { pName: "televison", price: 17999 },
9. { pName: "air conditioner", price: 32499 },
10. { pName: "smartphone", price: 24299 }
11. )
12. let productList = products.pipe(map((product) => product.pName));
13. productList.subscribe(
14. x => console.log('Product Name: ', x),
15. error => console.log("error", error),
16. () => console.log("data stream completed")
17. );

**Output:-**



# mergeMap():-

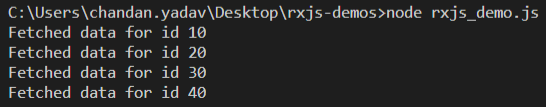
**mergeMap()**operator maps/projects each source value to an observable which is merged into the output observable.

Consider a scenario where an Observable emits an array and for each array element we need to fetch some data from the server. We could do this using a map function that calls another function responsible for API calls, and then subscribe to the result.

i.e. consider an **outer**Observable with array value **from([10, 20, 30, 40])**and **getData()**is the function that does API calls and returns an observable which is the **inner**observable. To get the data, we will have to subscribe to both **inner**as well as **outer**observables as shown:

1. import { from, Observable } from 'rxjs';
2. import { map } from 'rxjs/operators';
3. const getData = (id) => {
4. return new Observable(observer => {
5. setTimeout(() => {
6. observer.next(`Fetched data for id ${id}`)
7. }, 1000);
8. })
9. }
10. let arrayObservable = from([10, 20, 30, 40])
11. let dataObservable = arrayObservable.pipe(map(param => getData(param)))
12. dataObservable.subscribe(val => val.subscribe(data => console.log(data)));

**Output:-**



As you can see here, we had to subscribe twice to fetch the data from the observables which makes the things very much complicated. The same can be done in a much easier way using the **mergeMap()**operator. **mergeMap()** is actually a combination of **mergeAll()** and **map()** operators. **mergeMap** takes care of subscribing to the inner observable so that we need not subscribe twice as mergeAll will itself merge the values of inner observable to the outer observable.

**Syntax:-**

1. mergeMap(project: function, resultSelector ?: number, concurrent ?: number):OperatorFunction

**Example 1:-**The below code uses mergeMap and returns the same output as above

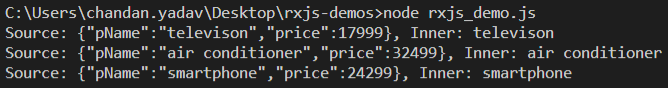
1. import { from, Observable } from 'rxjs';
2. import { mergeMap } from 'rxjs/operators';
3. const getData = (id) => {
4. return new Observable(observer => {
5. setTimeout(() => {
6. observer.next(`Fetched data for id ${id}`)
7. }, 1000);
8. })
9. }
10. let arrayObservable = from([10, 20, 30, 40])
11. let dataObservable = arrayObservable.pipe(mergeMap(param => getData(param)))
12. dataObservable.subscribe(val => console.log(val));

Next, lets see some another example for the use of**mergeMap()**operator.

**Example 2:-** **mergeMap with 2nd parameter i.e. resultReslector function**

1. import { of } from 'rxjs';
2. import { mergeMap } from 'rxjs/operators';
3. interface Product {
4. price: number,
5. pName: string
6. }
7. let products = of<Product>(
8. { pName: "televison", price: 17999 },
9. { pName: "air conditioner", price: 32499 },
10. { pName: "smartphone", price: 24299 }
11. )
12. let finalObservable = products.pipe(
13. mergeMap(
14. val => of(val.pName), */\* takes each element from the outer observable and returns pName as an observable \*/*
15. */\**
16. *we can also supply a second argument i.e. resultSelector which receives the source value and emitted*
17. *value of inner observable as arguments and returns the final observable*
18. *\*/*
19. (valueFromSourceObservable, valueFromInnerObservable) => {
20. return `Source: ${JSON.stringify(valueFromSourceObservable)}, Inner: ${valueFromInnerObservable}`;
21. }
22. )
23. )
24. finalObservable.subscribe(val => console.log(val));

**Output:-**



RxJS also provides a few operators to handle the error thrown by the source observable. The error handling operators are catchError(), retry() and retryWhen(). We will focus here only on **catchError()**operator.

# catchError() :-

**catchError()** operator will help in catching the errors from the source observable by returning a new observable or throwing an error.

**Syntax:-**

1. catchError(selectorFunc: (errFunc: any, caught: Observable) => O):Observable

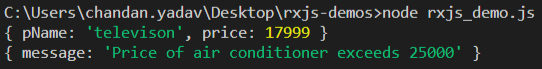
where **selectorFunc**is a function that takes 2 arguments, first being an error function and second is **caught**which is an observable.

**Note:-**The return value of **selectorFunc** must be an **Observable**.

**Example:-**

1. import { of } from 'rxjs';
2. import { map, catchError } from 'rxjs/operators';
3. interface Product {
4. price: number,
5. pName: string
6. }
7. let products = of<Product>(
8. { pName: "televison", price: 17999 },
9. { pName: "air conditioner", price: 32499 },
10. { pName: "smartphone", price: 24299 }
11. )
12. let finalObservable = products.pipe(
13. map(product => {
14. if (product.price > 25000) throw new Error(`Price of ${product.pName} exceeds 25000`);
15. else return product;
16. }),
17. catchError(err => of({ message: err.message })),
18. )
19. finalObservable.subscribe(x => console.log(x));

**Output:-**



Yoiu can read more about the other error handling operators [retry](https://rxjs-dev.firebaseapp.com/api/operators/retry) and [retryWhen](https://rxjs-dev.firebaseapp.com/api/operators/retryWhen)