# Transforming Data - Map, Filter, and Reduce / part 4

In the previous part, we started discussing the use of RxJS in our programs. To start using it, we compared Observables in Reactive Extensions with EventStreams and Properties in bacon.js. Then we looked at some of the most common sources of data we can use to create Observables, as follows:

- Arrays
- Range
- Interval
- Promises and callbacks DOM events
- Any arbitrary source
-

After this initial overview of the RxJS API on how to create Observables, we learned how to subscribe to them so we are able to take action whenever new data is made available in the Observable. Subscribing to an Observable means being able to not only react to new incoming data, but also to take some action when an error occurs or an Observable finishes. To do this, we learned how to use the `subscribe()` method, using functions or by creating an Observer.

We also learned how to use Subjects. Subjects allow us to create new Observables and keep pushing data to them.

RxJS gives us a lot more control of the life cycle of our Observables and even our application. Also, in the previous part, we used Disposables and Schedulers. They let us perform a teardown of our objects and control how we publish items in our Observable sequences.

With all of the knowledge we have gathered, so far we are now ready to add operators to our observables and make them more reusable and testable.

In this part, we will have a look at the four operators that are the backbone of any functional reactive program. They are as follows:

- `map()`
- `flatMap()`
- `filter()`
- `reduce()`

These are the most widely used/important operators. This part will be filled with lots of examples that will help you understand the usage of these operators. If you learn them well, they will make your road through functional reactive programming a lot easier, enabling you to learn about more specific operators and also how you can combine operators.

In this part, you will learn the following topics:

- Adding operators to Observables
- Using the `map()` and `flatMap()` operators
- Using the `filter()` operator
- Using the `reduce()` operator

## Adding operators to observables

In Part 2, we used operators using bacon.js. They let us transform our data before it reaches the subscribers of the given Observable. The operators in RxJS work similar to the ones in bacon.js; some even have the same name. They are just methods called from Observable objects, as you can see in the following example:

```
Rx.Observable
  .just('Hello ')
  .map((msg)=>msg+'World')
  .subscribe((msg)=> console.log(msg));
```

In this example, we created an Observable, which emits only one string, and called the `map()` operator over this Observable and subscribed to it to show the result in the console:

```
Hello World
```

Every time you call an operator over an observable, it returns a new observable with the transformation applied. This way, we can chain multiples operators easily, as follows:

```
Rx.Observable
  .of(1,2,3)
  .map((num)=>num*2)
  .filter((num)=> num>2)
  .subscribe((num)=>console.log(num));
```

In the preceding example, we created an Observable that emits three numbers. Then we called the `map()` operator over this Observable to first transform data and then apply the `filter()` operator to generate a new Observable. This code can also be written in the following way:

```
var initialObservable = Rx.Observable.of(1,2,3);
var initialMappedObservable = initialObservable
          .map((num)=>num*2);
var initialMappedAndFilteredObservable = initialMappedObservable
          .filter((num)=> num>2);
initialMappedAndFilteredObservable.subscribe((num)=>console.log(num));
```

Both provide the same output when executed:

```
4
6
```

Each of the Observables that is created after you apply an operator can be subscribed to, increasing the reuse of your code. Using operators, we can write cleaner and more testable code as it lets us detach the origin of our data from what to do with the data.

## The map() operator

The `map()` operator is common when working with arrays or any other kinds of sequences in functional languages and frameworks. In JavaScript, array objects have a method called `map()`. This method is available in all modern browsers now.

The `map()` operator calls the provided function once for each element in the Observable. This function takes the object in the Observable as input and returns another object. So the Observable returned by the `map()` operator will propagate the result of calling the map function for each element of the original Observable.

The provided function is called as soon as the object is propagated by the original Observable; it is called in the same order.

So the `map()` operator has the following signature:

```
observable.map(mapFunction,[thisContext]);
```

The first parameter is mandatory and the second is optional:

- `mapFunction`: This is a function that takes an element of the observable as input and returns another object to be propagated instead
- `thisContext`: This is any object to be used as the `this` (JavaScript context) of `mapFunction`

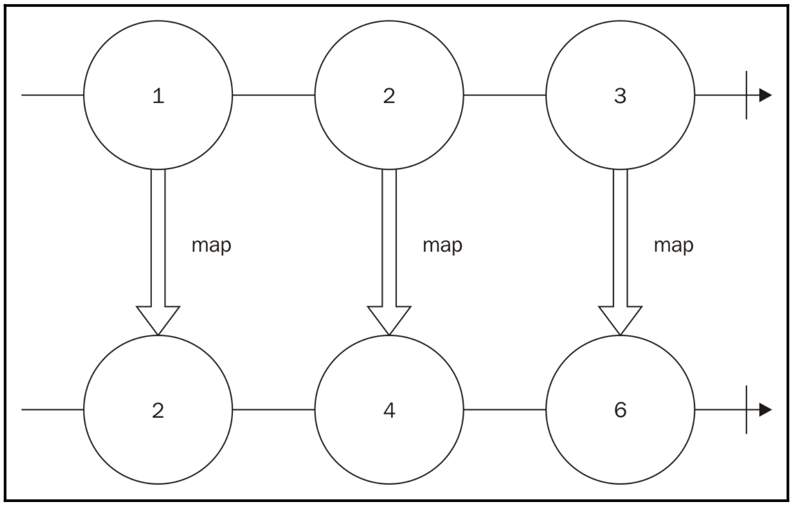
If we have an Observable of numbers and want to create a new Observable by doubling these numbers, we can use the `map()` operator, as follows:

```
Rx.Observable
    .of(1,2,3)
    .map((i)=>i*2)
    .subscribe((i)=>console.log(i));
```

In this code, first we created an Observable that propagated three numbers (`1`, `2`, and `3`). Then, we applied the `map()` operator over this Observable; this created a new Observable with the mapped objects. Then we subscribed to this Observable to log all the elements. If you run the preceding code, you will see this output in your console:

```
2
4
6
```

This operation can be represented by the following diagram:



This operator can be used over any kind of object, and as described in this part, it doesn't change the initial Observable and always creates a new Observable:

```
var namesObservable = Rx.Observable.of("John","Mary")

var helloObservable = namesObservable.map((name)=>"Hello "+name)
helloObservable.subscribe((i)=>console.log(i));

var thanksObservable = namesObservable.map((name)=>name+" thanks for your
visit.")
thanksObservable.subscribe((i)=>console.log(i));
```

In the preceding code, we created an Observable from two usernames (`John` and `Mary`). We then mapped it to an Observable of a string saying `hello` to each user, and we subscribed to the observable to print each message in the console.We also created a new Observable for a different message using the original Observable (with only the names), then we subscribed to it to log the message. If you run the preceding code, you will be presented with following output:

```
Hello John
Hello Mary
John thanks for your visit.
Mary thanks for your visit.
```

There is an alias for this operator called `select`.

## The flatMap() operator

In the previous section, we saw the `map()` operator and how we can use it to map an object into another object in an observable. The `flatMap()` operator works similarly. It receives a function as a parameter. The function provided as argument must returns an Observable, and the elements of this Observable is propagated instead.

This operator may seen a little bit confusing, but there is another way to understand it. The `flatMap()` operator runs the given function for each element in the original array. It creates an Observable that will emit each piece of data coming out from other Observables keeping the order.

The `flatMap()` operator is illustrated in the following signature:

```
observable.flatMap(flatMapFunction,[thisContext]);
```

The first parameter is mandatory and the second is optional:

- `flatMapFunction`: This is a function that takes an element of the observable as input and returns another observable whose elements are going to be propagated instead
- `thisContext`: This is any object to be used as the `this` (JavaScript context) of `flatMapFunction`

You can see an example usage of this operator in the following code:

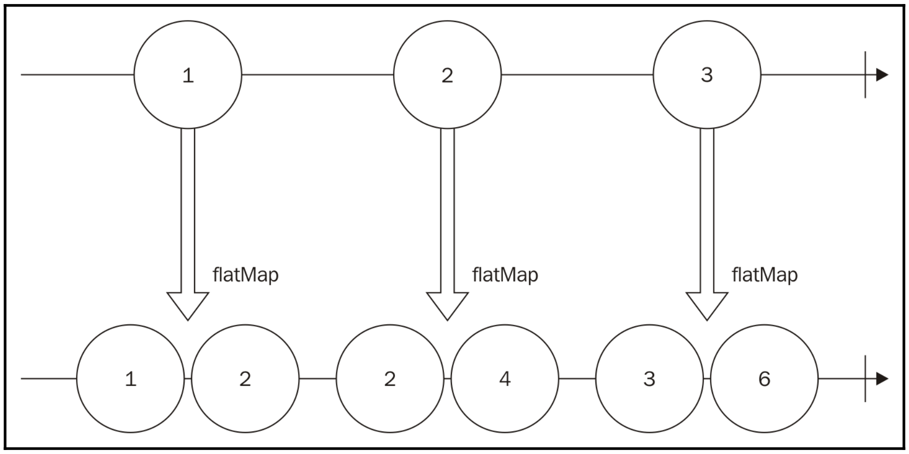
```
Rx.Observable
    .of(1,2,3)
    .flatMap((i)=>Rx.Observable.of(i,i*2))
    .subscribe((i)=>console.log(i));
```

In the preceding example, we first create an Observable that will propagate three numbers (`1`, `2`, and `3`). Then, we apply the `flatMap()` operator over this Observable; this will create a new Observable that will emit each value emitted by each Observable, and it will be returned by the `flatMapFunction`. The `flatMapFunction` returns an Observable containing the value and its double. So in the value `1` example, we create an Observable emitting the values `1` and `2`; for value `2`, we emit `2` and `4`; and for value `3`, we emit `3` and `6`.

If we run the preceding code, it will display this output:

```
1
2
2
4
3
6
```

This operation can be represented by the following diagram:



We can chain different operators easily, as follows:

```
Rx.Observable
    .of(1,2,3)
    .map((i)=>i+1)
    .flatMap((i)=>Rx.Observable.of(i,i*2))
    .subscribe((i)=>console.log(i));
```

In the preceding example, we chained the `map()` and `flatMap()` operators. First we create an Observable for `1`, `2`, and `3`, then we use the `map()` operator to add `1` to each value (now we have an Observable with values `2`, `3`, and `4`). Then, we use the `flatMap()` operator to emit a new sequence containing the value and its double.

If you run the preceding code, you will see following output:

```
2
4
3
6
4
8
```

There is an alias for this operator called `selectMany`.

## The filter() operator

The `filter()` operator lets you create a new Observable after omitting some of the data from the original Observable. It receives a function as argument that is called for each element in the original Observable, and it must return an either true or false (actually it can be any truthy or falsy value). If the result of the execution of `filterFunction` for an object is true, then this object will be propagated; otherwise, it will be omitted.

The `filter()` operator has the following signature:

```
observable.filter(filterFunction,[thisContext]);
```

The first parameter is mandatory and the second is optional:

- `filterFunction`: This is a function that takes an element of the observable as input and returns any truthy or falsy value. If the result of the execution of this function for a given object is true, then this object is propagated; otherwise, it is omitted.
- `thisContext`: This is any object to be used as the `thisContext` of `flatMapFunction`

We can see an example of the usage of `filter()` as follows:

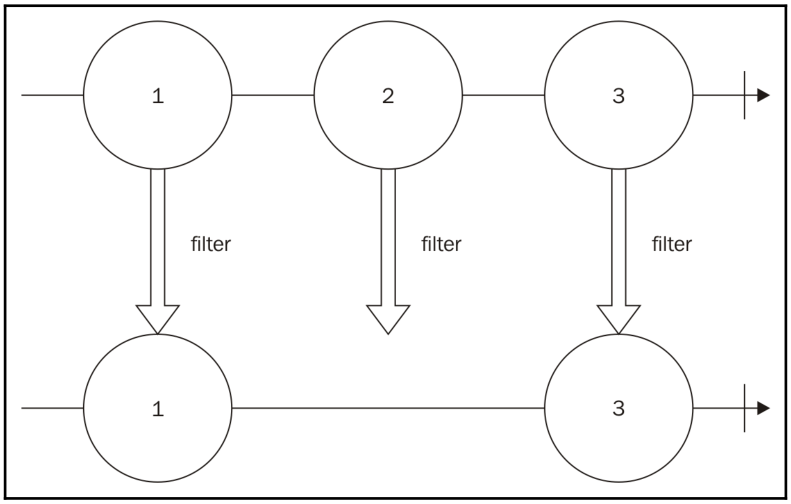
```
Rx.Observable
    .of(1,2,3)
    .filter((i)=> i % 2 === 1 )
    .subscribe((i)=>console.log(i));
```

In the preceding example, we first create an Observable that will propagate three numbers (`1`, `2`, and `3`). Then, we apply the `filter()` operator over this Observable; this will create a new Observable that will emit only the value `i`, where `i %2 === 1`. So the new Observable propagates only odd numbers.

If you run the preceding code, you will see this output:

```
1
3
```

This operation can be represented by the following diagram as well:



In this example, we are always returning true or false, but we can also return any truthy or falsy value.

A truthy value is any object that acts true when evaluated as a Boolean.
Falsy values are those that act false when evaluated as a Boolean.

Falsy values are as follows:

- False
- 0
- ""(empty string) Null Undefined NaN
-
-
-

All other values available in JavaScript are truthy values.

The following is an example that filters an Observable for only truthy values:

```
Rx.Observable
    .of(0,1,"hello",null,"")
    .filter((i)=> i )
    .subscribe((i)=>console.log(i));
```

If you run the preceding code, you will see this output:

```
1
hello
```

There is an alias for this operator called `where`.

## The reduce() operator

The `reduce()` operator lets you run a function to accumulate all the values of an Observable to generate a new Observable containing only one value (the accumulated value).

This operator has the following signature:

```
observable.reduce(accumulatorFunction,[initialValue]);
```

The first parameter is optional and the second one is mandatory:

- `accumulatorFunction`: This is a function that is used to accumulate values from an observable. This function can receive up to four parameters:
    - `acc`: This is an accumulated value
    - `currentValue`: This is the value used in this iteration
    - `currentIndex`: This is a zero-based index of this iteration
    - `source`: This is the observable used
- `initialValue`: This is the initial accumulator

We can use this function to sum up all the values in an Observable containing numbers, as follows:

```
Rx.Observable
     .of(1,2,3)
     .reduce((acc,current)=>acc+current)
     .subscribe((i)=>console.log(i));
```

In this example, we first create an Observable that will propagate three numbers (`1`, `2`, and `3`). Then, we apply the `reduce()` operator over this Observable; this will create a new Observable with only one value, and the value will be the sum of each element in this sequence.

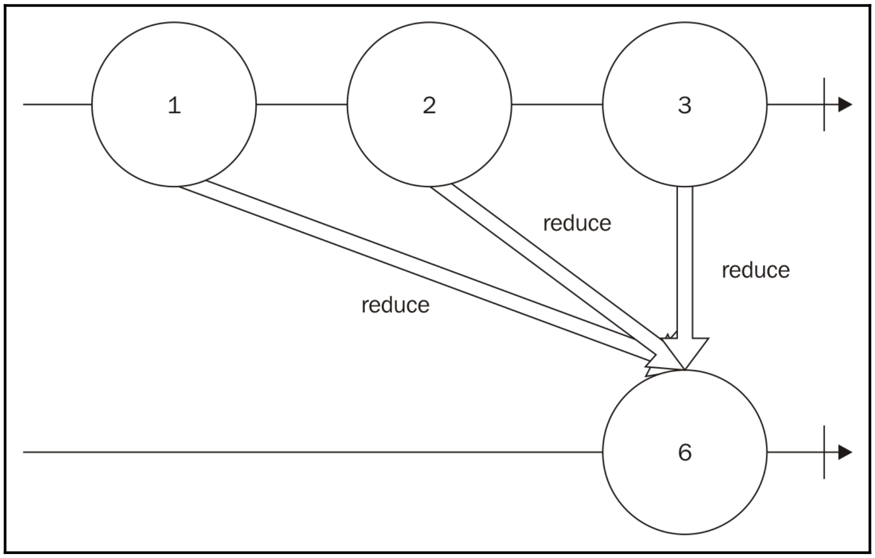
This method runs in the following iteration:

1. If an initial value is supplied, we use it as the accumulator and the first element in the sequence as the current value; if not, we use the first element in the sequence as the accumulator and the second as the current value.
2. Run `accumulatorFunction` for the accumulator and the current value.
3. Set the current value as the next element in the sequence.
4. Go back to *step 2* until you process the whole list.

So if you run the preceding code, you will see following output:

```
6
```

This operation can be represented by the following diagram:



We can also supply an initial value for the `reduce()` operator, as can be seen in the following example:

```
Rx.Observable
    .of("w","o","r","l","d")
    .reduce((acc,current)=>acc+current, "Hello ")
    .subscribe((i)=>console.log(i));
```

As you can imagine, running this code will concatenate the whole string so you could see the following message on your console:

```
Hello world
```

In this part, we learned how to apply operators to observables. This helps decouple our code and improve its testability and readability. This is the first time we used the following operators in RxJS such as `map()`, `flatMap()`, `filter()`, and `reduce()`.

These are the most widely used operators in RxJS, and for this reason, they are also the most important ones. We used some of them in bacon.js, but in this part, we were able to have a more in-depth look at each one of them; diagrams and examples only helped us understand their usage better.