# Difference between Rxjs Subject and Observable

In order to understand the difference between a Subject and an Observable, you need to be aware of two distinct concepts

– A data producer  
– A data consumer

An observable, by definition is a data producer. that can produce data over time.

A Subject on the other hand can act as both – a data producer and a data consumer.

This implies two things.  
1. A subject can be subscribed to, just like an observable.  
2. A subject can subscribe to other observables.

That being said, there is one critical difference between a subject and an observable.

All subscribers to a subject share the same execution of the subject. i.e. when a subject produces data, all of its subscribers will receive the same data. This behavior is different from observables, where each subscription causes an independent execution of the observable.

An **RxJS Subject** is a special type of Observable that allows values to be multicasted to many Observers. While plain Observables are unicast (each subscribed Observer owns an independent execution of the Observable), **Subjects** are multicast. A **Subject** is like an Observable, but can multicast to many Observers.

A subject in Rx is a special hybrid that can act as both an observable and an observer at the same time. This way, data can be pushed into a subject and the subject’s subscribers will in turn receive that pushed data.

**RxJS Subjects Tutorial - Subjects, BehaviorSubject, ReplaySubject & AsyncSubject**

## Creating a Subject

A Subject, in contrast to an observable, is simply an observer that's also able to emit values. It's both an observable and an observer simultaneously. This is unlike an observable, as an observer that's subscribed to an observable can only read values emitted from an observable.

Let's create a Subject using the project from the previous tutorial, and let's get started with the following code in /src/code.ts:

import { Subject } from "rxjs/Subject";

var subject = new Subject()

// Our handy function for showing the values:

// We will use this shortly...

function addItem(val:any) {

var node = document.createElement("li");

var textnode = document.createTextNode(val);

node.appendChild(textnode);

document.getElementById("output").appendChild(node);

}

As you can see, creating an actual Subject is ridiculously simple.

Next, we have to .subscribe to the Subject to create an observer:

var subject = new Subject()

subject.subscribe(

data => addItem('Observer 1: '+ data),

err => addItem(err),

() => addItem('Observer 1 Completed')

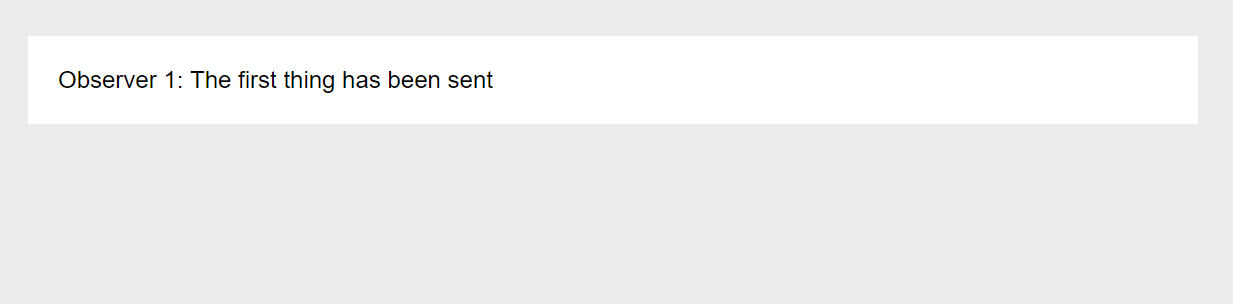
)

If you save the project, at this point nothing happens. We have to use .next to start emitting values from the observer:

// Previous code from above removed for brevity

subject.next('The first thing has been sent')

Now, if you save the project, we'll see the following output:

  
Simple enough, but let's make it more interesting by defining another observer and pushing some more values:

// Previous code from above removed for brevity

var observer2 = subject.subscribe(

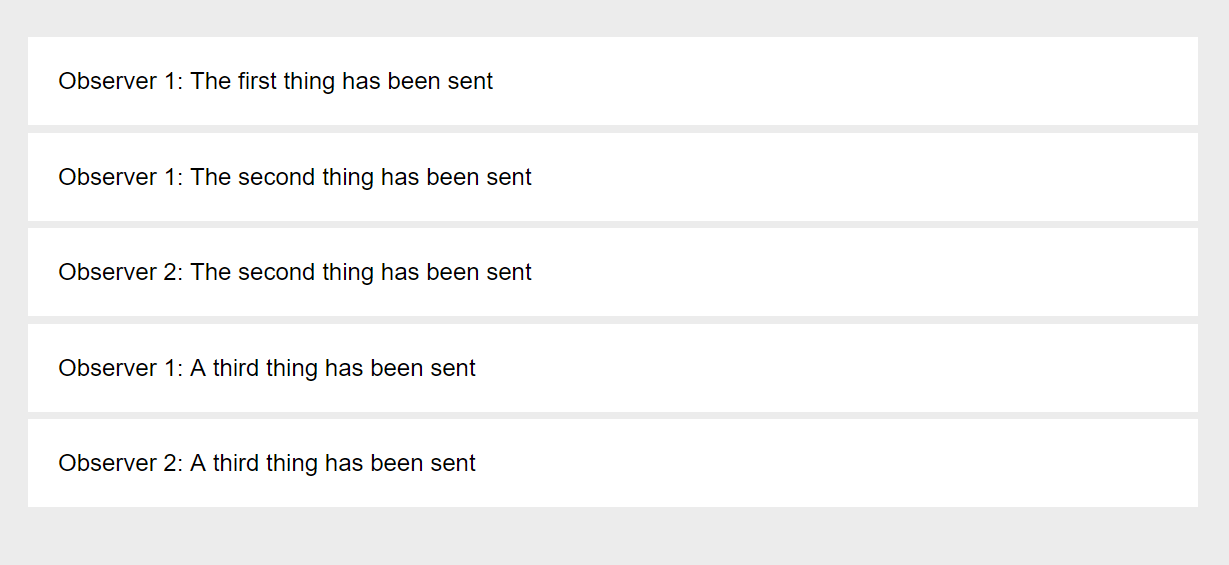
data => addItem('Observer 2: '+ data)

)

subject.next('The second thing has been sent')

subject.next('A third thing has been sent')

The result:



We can see that the 2nd observer doesn't receive 'The first thing has been sent' but only the values emitted after it was created.

Subject observers also have the following methods:

* **add()** - You can add child observers.
* **remove()** - Removing child observers that were added.
* **unsubscribe()** - You can close an observer subscription.

Let's try unsubscribing the second observer and sending a value after it. This will include the full code up to this point:

import { Subject } from "rxjs/Subject";

var subject = new Subject()

subject.subscribe(

data => addItem('Observer 1: '+ data),

err => addItem(err),

() => addItem('Observer 1 Completed')

)

subject.next('The first thing has been sent')

var observer2 = subject.subscribe(

data => addItem('Observer 2: '+ data)

)

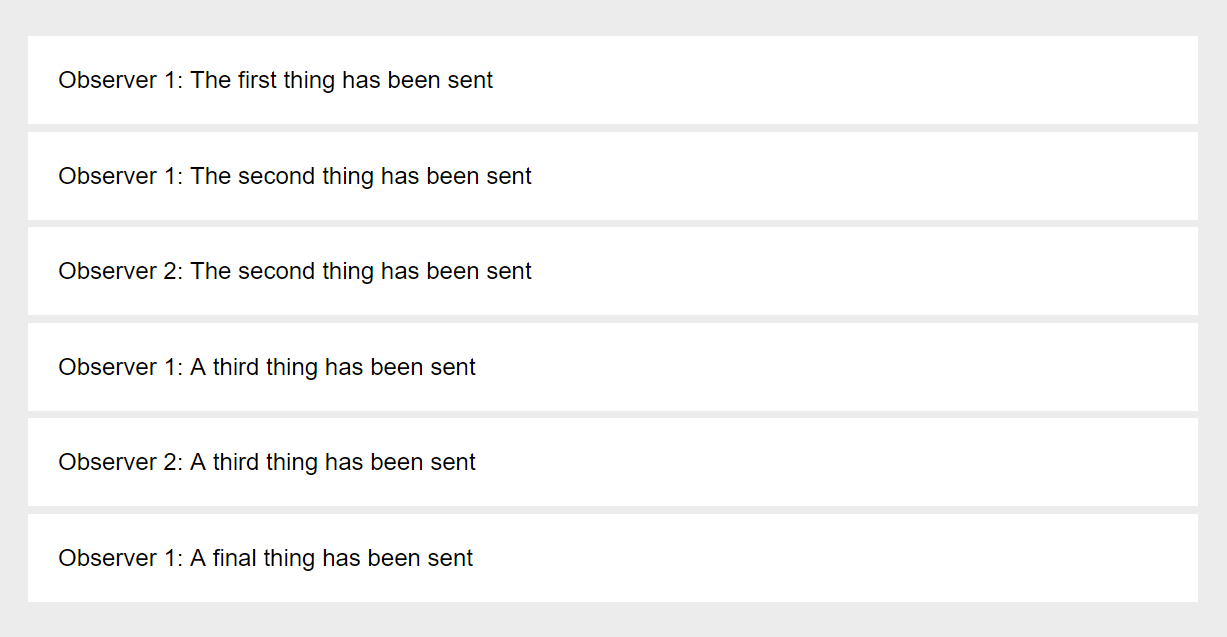
subject.next('The second thing has been sent')

subject.next('A third thing has been sent')

observer2.unsubscribe();

subject.next('A final thing has been sent')

This results in the following output:



Great! We can see that the first observer is still receiving data but the second no longer does.

## BehaviorSubject

We've just created a regular Subject, but what about BehaviorSubject?

BehaviorSubject is a special type of Subject whose only different is that it will emit the **last value** upon a new observer's subscription.

For instance, in the above example of a regular Subject, when Observer 2 subscribed, it did not receive the previously emitted value 'The first thing has been sent' -- In the case of a BehaviorSubject, it would.

Let's create a BehaviorSubject using our existing code. Simply change lines 1 and 3 to the following:

import { BehaviorSubject } from "rxjs/BehaviorSubject";

var subject = new BehaviorSubject('First')

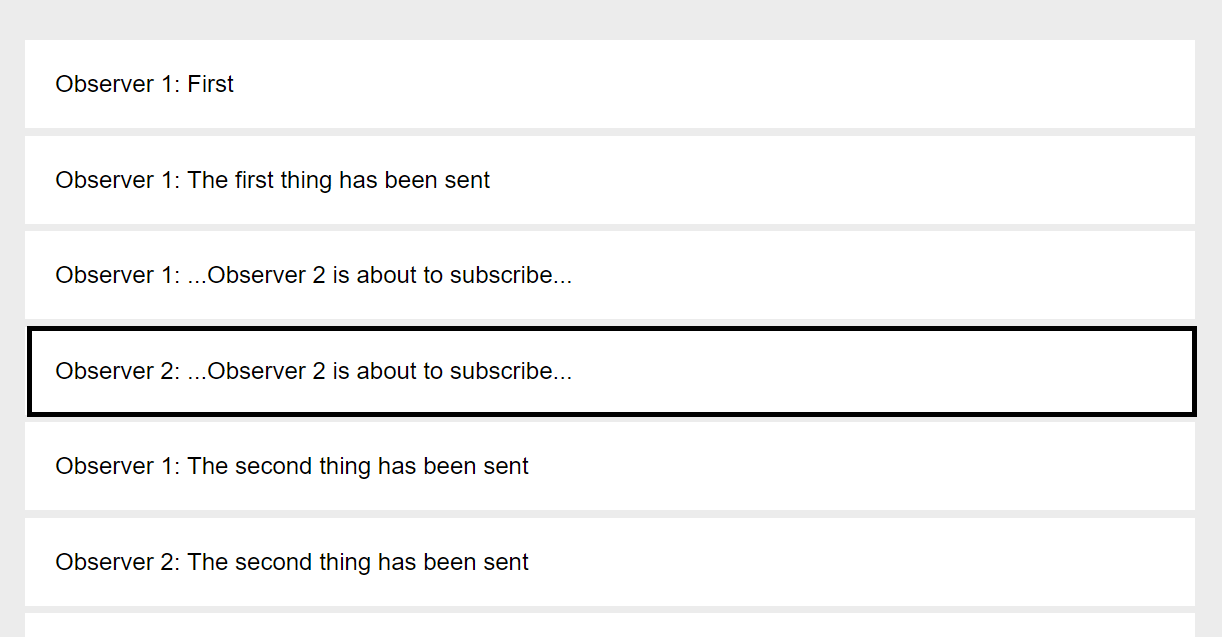
Also, just after our first call to .next(), make the adjustment:

subject.next('The first thing has been sent')

// Add this

subject.next('...Observer 2 is about to subscribe...')

If you view the result in the browser, you will see that Observer 2 received the last emitted value, despite being created after it:



That's all there is to BehaviorSubject.

## ReplaySubject

Another variation of the Subject is a ReplaySubject.

It's like BehaviorSubject, except it allows you to specify a buffer, or number of emitted values to dispatch to observers. BehaviorSubject only dispatches the last emitted value, and ReplaySubject allows you to dispatch any designated number of values.

Let's give it a try in our project:

import { ReplaySubject } from "rxjs/ReplaySubject";

// We will only return the last 2 emitted values to new observers:

var subject = new ReplaySubject(2)

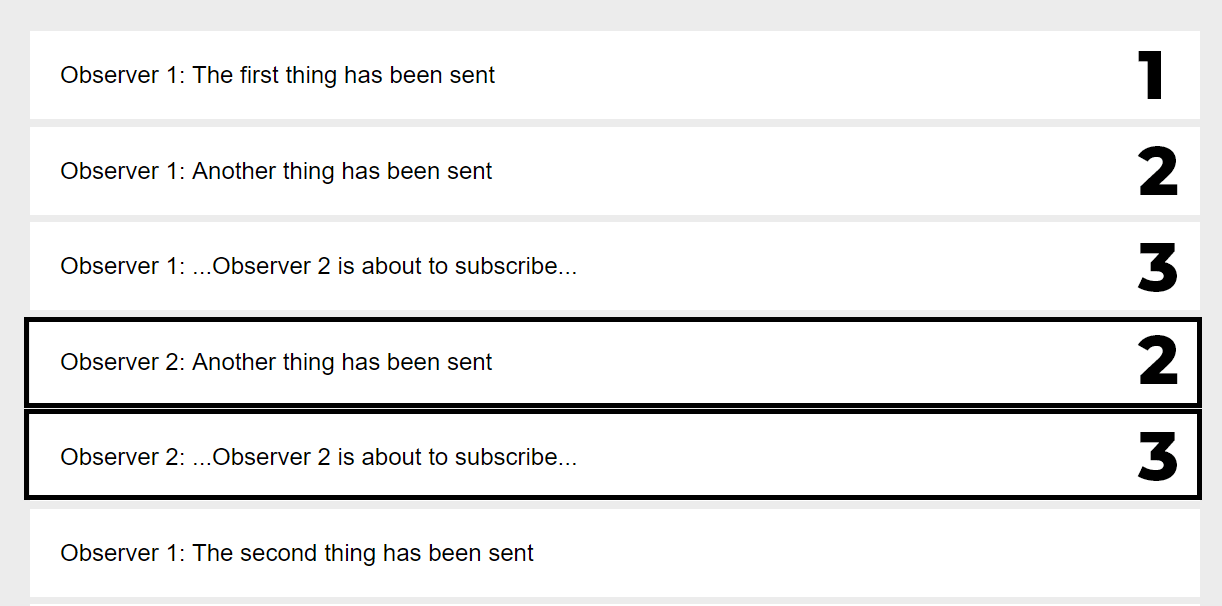
Also, let's once again make adjustments to our .next() calls:

subject.next('The first thing has been sent')

subject.next('Another thing has been sent')

subject.next('...Observer 2 is about to subscribe...')

The result in the browser reveals that Observer 2 received the last 2 emitted values:



See? All of this stuff is easy to understand.

ReplaySubject accepts an optional second argument upon creation, which is referred to as the window time, and it's defined in milliseconds. It allows you to define a maximum number of events to return in the first argument, and the second argument is the time in milliseconds.

To demonstrate this, we're going to make a number of changes:

var subject = new ReplaySubject(30, 200)

subject.subscribe(

data => addItem('Observer 1: '+ data),

err => addItem(err),

() => addItem('Observer 1 Completed')

)

var i = 1;

var int = setInterval(() => subject.next(i++), 100);

setTimeout(() => {

var observer2 = subject.subscribe(

data => addItem('Observer 2: '+ data)

)

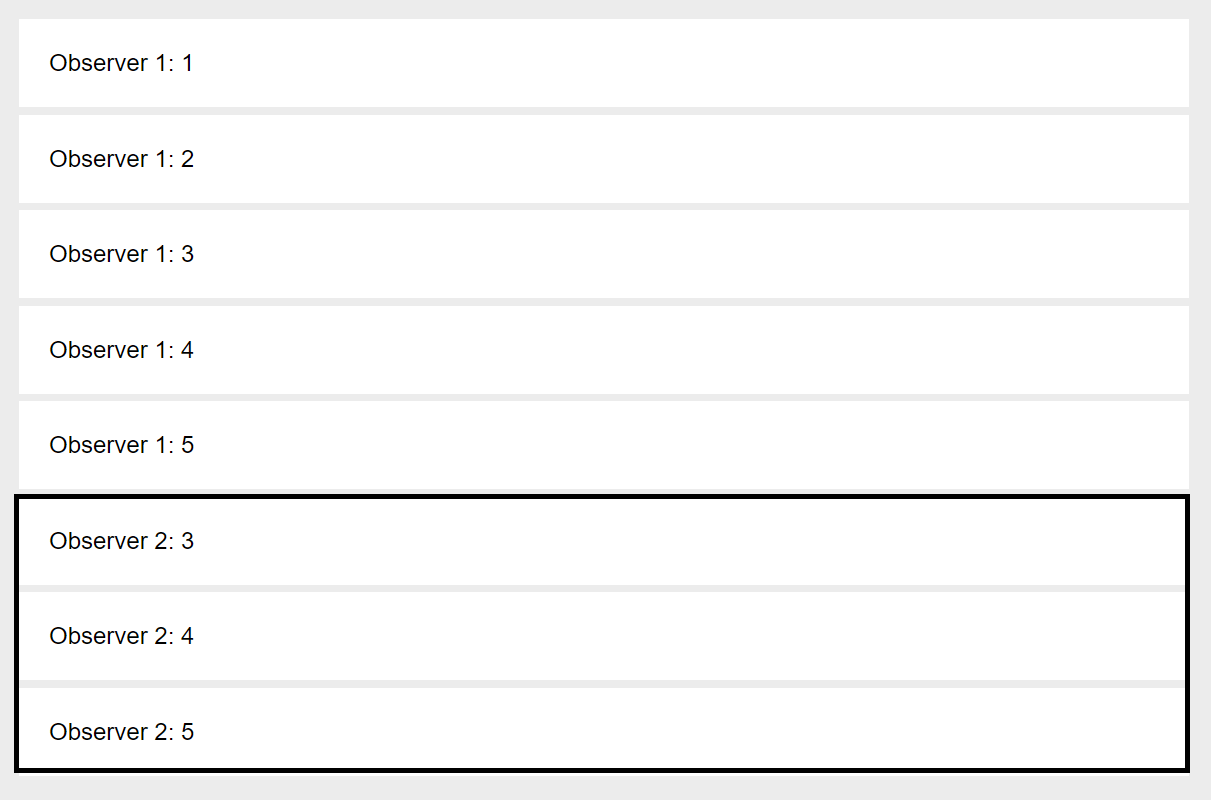
clearInterval(int);

}, 500);

On line 3, we're saying **return the last 30 emitted values (events)** ...**within 200 milliseconds** of a new subscription.

Then, we're using setInterval() to call .next() to dispatch a new event every 100 milliseconds.

Then, we create our second observer after 500 milliseconds.

The result in the browser:

If you try changing the window time on line 3 from 200 to 500, Observer 2 will receive all 5 emitted values, because they occurred within 500 milliseconds.

## AsyncSubject

This is the last subject variation, and it's perhaps the most simple to understand.

AsyncSubject only emits the very last value, and will only do so once .complete() has been called on the subject.

Using our example from above, let's make some slight adjustments as shown below:

import { AsyncSubject } from "rxjs/AsyncSubject";

var subject = new AsyncSubject()

subject.subscribe(

data => addItem('Observer 1: '+ data),

() => addItem('Observer 1 Completed')

)

var i = 1;

setInterval(() => subject.next(i++), 100);

setTimeout(() => {

var observer2 = subject.subscribe(

data => addItem('Observer 2: '+ data)

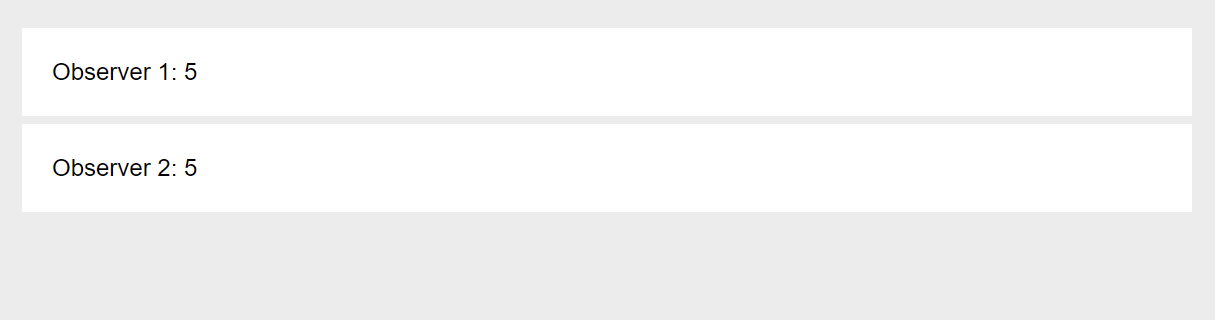
)

subject.complete();

}, 500);

Notice that we're calling **subject.complete()** in the setTimeout() call near the bottom.

This is the result:



Simple enough!

## Conclusion

Now, with your understanding of Observables and Subjects (variations of the Observable), you should have a pretty solid understanding of RxJS in respect to Observables.

In the next section, we're going to take a look at operators.

* **Note**: This tutorial is a part our free comprehensive **[RxJS Tutorial](https://coursetro.com/courses/25/A-Comprehensive-RxJS-Tutorial---Learn-ReactiveX-for-JavaScript-)**