RxJS Subjects

An RxJS Subject is like an Observable. It is a special type of Observable that allows values to be multicasted to many Observers. In simple words, we can say that an RxJS subject is an Observable can multicast or talk to many observers.

According to its official definition, "A Subject is like an Observable, but can multicast to many Observers. Subjects are like EventEmitters: they maintain a registry of many listeners."

An [RxJS](https://www.javatpoint.com/rxjs) subject can be subscribed to, just like we usually do with Observables. It also has methods such as next(), error() and complete(), which we have already seen and used in our Observable creation function.

Difference between RxJS Observable and RxJS Subject

Every Subject is an Observable. We can subscribe to a given Subject just like an observable, and it will start receiving values usually. From the perspective of the Observer, it cannot be decided whether the Observable execution is coming from a plain unicast Observable or a Subject.

The main difference between an Observable and a Subject is that a plain Observable by default is unicast. It means that each subscribed Observer owns an independent execution of the Observable. On the other hand, Subjects are multicast. A Subject is like an Observable, but it can multicast to many Observers. The main reason behind using Subjects is to multicast.

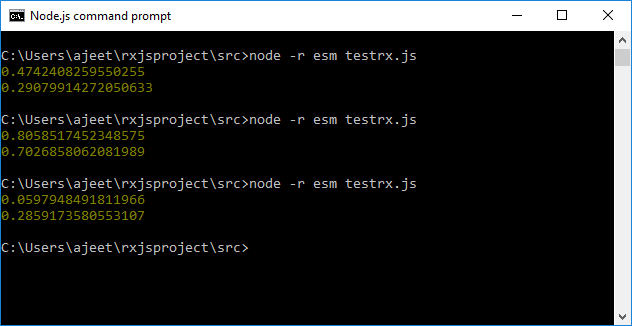
See an example of Observable:

**Example 1**

1. **import** \* as Rx from "rxjs";
2. **const** observable = Rx.Observable.create((observer) => {
3. observer.next(Math.random());
4. });
5. // subscription 1
6. observable.subscribe((data) => {
7. console.log(data);
8. });
9. // subscription 2
10. observable.subscribe((data) => {
11. console.log(data);
12. });

After executing the above example, we will see the following result. Here, you will see that Observables are unicast by design so, they will produce different random results every time you execute the above example. See the following output:

**Output:**



Here, you can see that result is different for both subscriptions every time we execute the program. This isn't very pleasant if you expect that each subscriber receives the same values. Subjects are used to overcome this issue because subjects can multicast. It means that one Observable execution is shared among multiple subscribers.

Subjects are like EventEmitters. They are used to maintain a registry of many listeners, so when we call subscribe on a Subject, it does not invoke a new execution. It simply registers the given Observer in a list of Observers.

Using RxJS Subjects

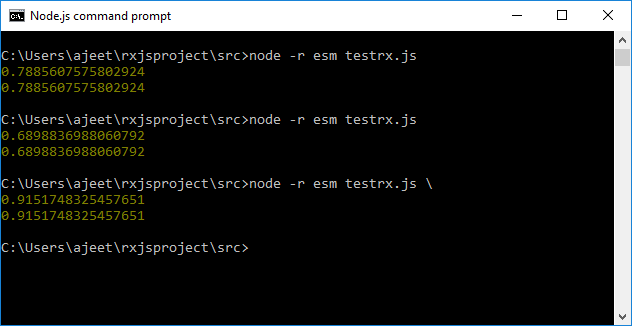
Let's see how to use Subjects to multicast and to overcome the above issue.

**Example 2**

1. **import** \* as Rx from "rxjs";
2. **const** subject = **new** Rx.Subject();
3. // subscriber 1
4. subject.subscribe((data) => {
5. console.log(data);
6. });
7. // subscriber 2
8. subject.subscribe((data) => {
9. console.log(data);
10. });
11. subject.next(Math.random());

**Output:**

After executing the above example, we will see the following result.



Here, you can see that every time we execute the program, it shows the different random numbers, but the values for both subscriptions are the same. It means the two subscriptions are getting the same data.

How to Convert Observables from Unicast to Multicast

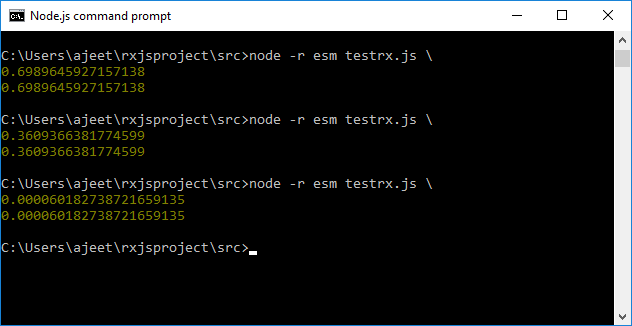
The RxJS Observables are solely data producers, but the RxJS Subjects can be used as a data producer as well as a data consumer. By using Subjects as a data consumer, we can use them to convert Observables from unicast to multicast. See the following example:

**Example 3**

1. **import** \* as Rx from "rxjs";
2. **const** observable = Rx.Observable.create((observer) => {
3. observer.next(Math.random());
4. });
5. **const** subject = **new** Rx.Subject();
6. // subscriber 1
7. subject.subscribe((data) => {
8. console.log(data);
9. });
10. // subscriber 2
11. subject.subscribe((data) => {
12. console.log(data);
13. });
14. observable.subscribe(subject);

**Output:**

After executing the above example, we will see the following result.



Here, we have passed our Subject to the subscribe function. It took the Observable's values, and then all the subscribers to that Subject immediately receive that value.

How to create an RxJS Subject?

Let's see how to work with RxJS subjects. To work with an RxJS subject, we need to import Subject in the following manner:

1. **import** { Subject } from 'rxjs';

Now, use the following method to create a subject object:

1. **const** subject\_test = **new** Subject();

Same as the RxJS Observables, an RxJS Subject also has the following three methods:

* next(v)
* error(e)
* complete()

How to subscribe to the RxJS Subject?

After creating the RxJS subject, we have to subscribe to it. A Subscription is an object that is used to represent a disposable resource, usually the execution of the Subject. We can easily create multiple subscriptions on the Subject by using the following method:

1. subject\_test.subscribe({
2. next: (v) => console.log(`From Subject : ${v}`)
3. });
4. subject\_test.subscribe({
5. next: (v) => console.log(`From Subject: ${v}`)
6. });

How to pass data to Subject?

After subscription, we need to pass data to the subject we have created. We can do this by using the next() method.

1. subject\_test.next("A");

This data will be passed to all the subscription added on the subject.

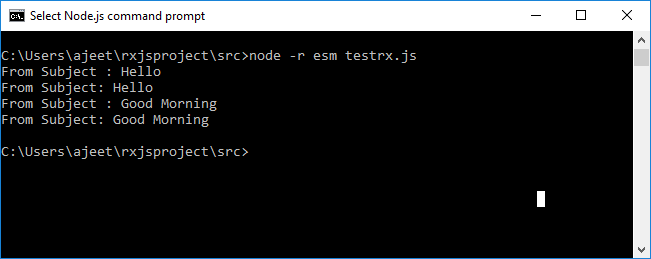
Let's see a complete example of RxJS Subject. Here, we will use above three methods: next(v), error(e), and complete()

**Example using next(v) method:**

1. **import** { Subject } from 'rxjs';
2. **const** subject\_test = **new** Subject();
3. subject\_test.subscribe({
4. next: (v) => console.log(`From Subject : ${v}`)
5. });
6. subject\_test.subscribe({
7. next: (v) => console.log(`From Subject: ${v}`)
8. });
9. subject\_test.next("Hello");
10. subject\_test.next("Good Morning");

Here, we have created an object named "subject\_test" by calling a new Subject(). After that, the subject\_test object has reference to next() method.

**Output:**

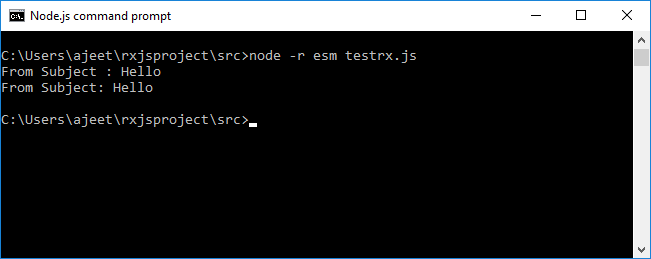


**Example using complete() method:**

We can use the complete() method to stop the subject execution. See the following example:

1. **import** { Subject } from 'rxjs';
2. **const** subject\_test = **new** Subject();
3. subject\_test.subscribe({
4. next: (v) => console.log(`From Subject : ${v}`)
5. });
6. subject\_test.subscribe({
7. next: (v) => console.log(`From Subject: ${v}`)
8. });
9. subject\_test.next("Hello");
10. subject\_test.complete();
11. subject\_test.next("Good Morning");

**Output:**

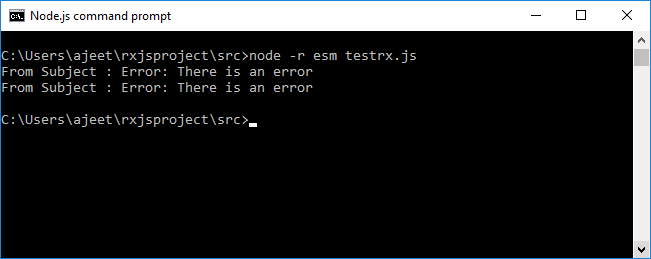


**Example using error() method:**

Let's see how to call error () method.

1. **import** { Subject } from 'rxjs';
2. **const** subject\_test = **new** Subject();
3. subject\_test.subscribe({
4. error: (e) => console.log(`From Subject : ${e}`)
5. });
6. subject\_test.subscribe({
7. error: (e) => console.log(`From Subject : ${e}`)
8. });
9. subject\_test.error(**new** Error("There is an error message"));

**Output:**



Types of RxJS Subjects

There are mainly four variants of RxJS subjects:

1. **Subject -** This is the standard RxJS Subject. It doesn't have any initial value or replay behaviour.
2. **BehaviorSubject -** This variant of RxJS subject requires an initial value and emits its current value (last emitted item) to new subscribers.
3. **ReplaySubject -** This variant of RxJS subject is used to emit a specified number of last emitted values (a replay) to new subscribers.
4. **AsyncSubject -** The AsyncSubject emits the latest value to observers upon completion.

BehaviorSubject

The BehaviorSubject is used to denote "the current and latest value when called". It stores the latest value emitted to its consumers, and whenever a new Observer subscribes, it will immediately receive the "current value" from the BehaviorSubject.

BehaviorSubjects are mainly used to represent "values over time". For example, an event stream of birthdays is a Subject, but the stream of a person's age would be a BehaviorSubject.

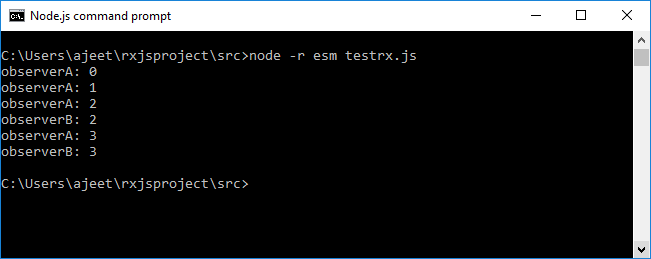
**Syntax:**

1. **import** { BehaviorSubject } from 'rxjs';
2. **const** subject = **new** BehaviorSubject("given\_value");
3. // initialized the behaviour subject with value: given\_value

**Example**

1. **import** { BehaviorSubject } from 'rxjs';
2. **const** subject = **new** BehaviorSubject(0); // 0 is the initial value
3. subject.subscribe({
4. next: (v) => console.log(`observerA: ${v}`)
5. });
6. subject.next(1);
7. subject.next(2);
8. subject.subscribe({
9. next: (v) => console.log(`observerB: ${v}`)
10. });
11. subject.next(3);

**Output:**



ReplaySubject

A ReplaySubject is pretty similar to a BehaviorSubject. It is also used to send old values to new subscribers, but it can also record a part of the Observable execution. A ReplaySubject records multiple values from the Observable execution and replays them to new subscribers.

**Syntax:**

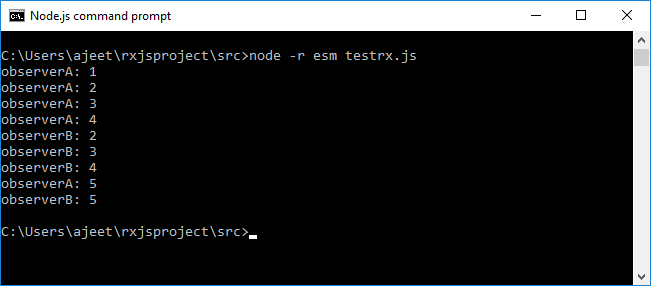
1. **import** { ReplaySubject } from 'rxjs';
2. **const** replay\_subject = **new** ReplaySubject(given\_value);

While creating a ReplaySubject, you can specify how many values you have to replay. See the following example:

**Example**

1. **import** { ReplaySubject } from 'rxjs';
2. **const** subject = **new** ReplaySubject(3); // buffer 3 values for new subscribers
3. subject.subscribe({
4. next: (v) => console.log(`observerA: ${v}`)
5. });
6. subject.next(1);
7. subject.next(2);
8. subject.next(3);
9. subject.next(4);
10. subject.subscribe({
11. next: (v) => console.log(`observerB: ${v}`)
12. });
13. subject.next(5);

**Output:**



How to set a Window time?

You can also specify a window time in milliseconds, besides of the buffer size, to determine how old the recorded values can be.

See the following example where we have used a large buffer size of 100, but a window time parameter of just 500 milliseconds.

**Example**

1. **import** { ReplaySubject } from 'rxjs';
2. **const** subject = **new** ReplaySubject(100, 500 /\* windowTime \*/);
3. subject.subscribe({
4. next: (v) => console.log(`observerA: ${v}`)
5. });
6. let i = 1;
7. setInterval(() => subject.next(i++), 200);
8. setTimeout(() => {
9. subject.subscribe({
10. next: (v) => console.log(`observerB: ${v}`)
11. });
12. }, 1000);

**Output:**

observerA: 1

observerA: 2

observerA: 3

observerA: 4

observerB: 3

observerB: 4

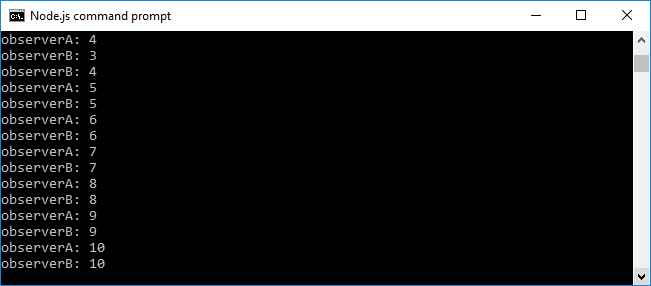
observerA: 5

observerB: 5

observerA: 6

observerB: 6

//..... and so on.



AsyncSubject

In this variant of RxJS Subject, only the last value of the Observable execution is sent to its observers and is also done after the complete() method is called.

**Syntax:**

1. **import** { AsyncSubject } from 'rxjs';
2. **const** async\_subject = **new** AsyncSubject();

**Example**

1. **import** { AsyncSubject } from 'rxjs';
2. **const** subject = **new** AsyncSubject();
3. subject.subscribe({
4. next: (v) => console.log(`observerA: ${v}`)
5. });
6. subject.next(1);
7. subject.next(2);
8. subject.next(3);
9. subject.next(4);
10. subject.subscribe({
11. next: (v) => console.log(`observerB: ${v}`)
12. });
13. subject.next(5);
14. subject.complete();

**Output:**

