A BehaviorSubject holds one value. When it is subscribed it emits the value immediately. A Subject doesn't hold a value.

Subject example (with RxJS 5 API):

```
const subject = new Subject();
subject.next(1);
subject.subscribe(x => console.log(x));
```

Console output will be empty

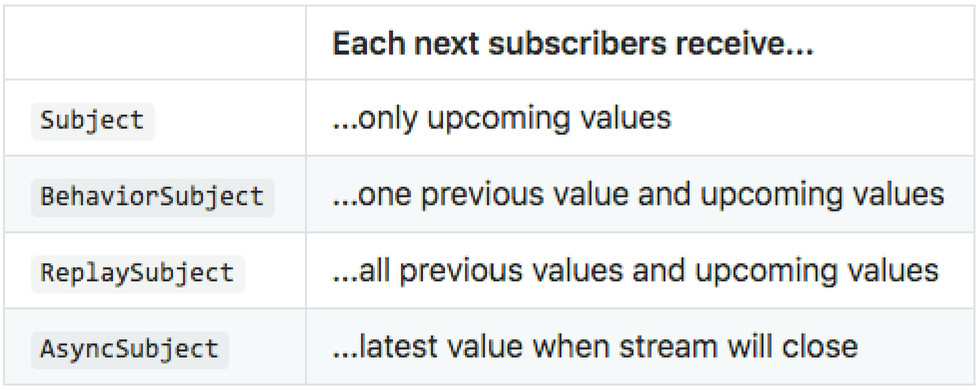
BehaviorSubject example:

```
const subject = new BehaviorSubject(0);
subject.next(1);
subject.subscribe(x => console.log(x));
```

Console output: 1

In addition:

- BehaviorSubject should be created with an initial value: new BehaviorSubject(1)
- Consider ReplaySubject if you want the subject to get previously publised values.



	Each next subscribers receive...
`Subject`	...only upcoming values
`BehaviorSubject`	...one previous value and upcoming values
`ReplaySubject`	...all previous values and upcoming values
`AsyncSubject`	...latest value when stream will close

Understanding rxjs BehaviorSubject, ReplaySubject and AsyncSubject

Subjects are used for multicasting Observables. This means that Subjects will make sure each subscription gets the exact same value as the Observable execution is shared among the subscribers. You can do this using the Subject class. But rxjs offers different types of Subjects, namely: BehaviorSubject, ReplaySubject and AsyncSubject.

If you think you understand Subjects, read on! Else i would suggest to read my other article about Subjects: Understanding rxjs Subjects.

The BehaviorSubject

One of the variants of the Subject is the BehaviorSubject. The BehaviorSubject has the characteristic that it stores the “current” value. This means that you can always directly get the last emitted value from the BehaviorSubject.

There are two ways to get this last emited value. You can either get the value by accessing the `.value` property on the BehaviorSubject or you can subscribe to it. If you subscribe to it, the BehaviorSubject will directly emit the current value to the subscriber. Even if the subscriber subscribes much later than the value was stored. See the example below:

```
import * as Rx from "rxjs";
          const subject = new Rx.BehaviorSubject();
          // subscriber 1
          subject.subscribe((data) => {
          console.log('Subscriber A:', data);
          });
          subject.next(Math.random());
          subject.next(Math.random());
          // subscriber 2
```

```
subject.subscribe((data) => {
console.log('Subscriber B:', data);
});
subject.next(Math.random());
console.log(subject.value)
// output
// Subscriber A: 0.24957144215097515
// Subscriber A: 0.8751123892486292
// Subscriber B: 0.8751123892486292
// Subscriber A: 0.1901322109907977
// Subscriber B: 0.1901322109907977
// 0.1901322109907977
```

There are a few things happening here:

1. We first create a subject and subscribe to that with Subscriber A. The Subject then emits it’s value and Subscriber A will log the random number.
2. The subject emits it’s next value. Subscriber A will log this again
3. Subscriber B starts with subscribing to the subject. Since the subject is a BehaviorSubject the new subscriber will automatically receive the last stored value and log this.
4. The subject emits a new value again. Now both subscribers will receive the values and log them.
5. Last we log the current Subjects value by simply accessing the `.value` property. This is quite nice as it’s synchronous. You don’t have to call subscribe to get the value.

Last but not least, you can create BehaviorSubjects with a start value. When creating Observables this can be quite hard. With BehaviorSubjects this is as easy as passing along an initial value. See the example below:

```
import * as Rx from "rxjs";
    const subject = new Rx.BehaviorSubject(Math.random());
    // subscriber 1
    subject.subscribe((data) => {
    console.log('Subscriber A:', data);
    });
    // output
    // Subscriber A: 0.24957144215097515
```

The ReplaySubject

The ReplaySubject is comparable to the BehaviorSubject in the way that it can send “old” values to new subscribers. It however has the extra characteristic that it can record a part of the observable execution and therefore store multiple old values and “replay” them to new subscribers.

When creating the ReplaySubject you can specify how much values you want to store and for how long you want to store them. In other words you can specify: “I want to store the last 5 values, that have been executed in the last second prior to a new subscription”. See example code below:

```
import * as Rx from "rxjs";
    const subject = new Rx.ReplaySubject(2);
    // subscriber 1
    subject.subscribe((data) => {
    console.log('Subscriber A:', data);
    });
    subject.next(Math.random())
    subject.next(Math.random())
    subject.next(Math.random())
    // subscriber 2
    subject.subscribe((data) => {
    console.log('Subscriber B:', data);
    });
    subject.next(Math.random());
    // Subscriber A: 0.3541746356538569
    // Subscriber A: 0.12137498878080955
    // Subscriber A: 0.531935186034298
    // Subscriber B: 0.12137498878080955
    // Subscriber B: 0.531935186034298
    // Subscriber A: 0.6664809293975393
    // Subscriber B: 0.6664809293975393
```

There are a few things happening here:

1. We create a ReplaySubject and specify that we only want to store the last 2 values
2. We start subscribing to the Subject with Subscriber A
3. We execute three new values trough the subject. Subscriber A will log all three.
4. Now comes the magic of the ReplaySubject. We start subscribing with Subscriber B. Since we told the ReplaySubject to store 2 values, it will directly emit those last values to Subscriber B and Subscriber B will log those.
5. Subject emits another value. This time both Subscriber A and Subscriber B just log that value.

As mentioned before you can also specify for how long you wan to store values in the replay subject. Let's see an example of that:

```
import * as Rx from "rxjs";
    const subject = new Rx.ReplaySubject(2, 100);
    // subscriber 1
    subject.subscribe((data) => {
    console.log('Subscriber A:', data);
    });
    setInterval(() => subject.next(Math.random()), 200);
```

```
// subscriber 2
setTimeout(() => {
subject.subscribe((data) => {
console.log('Subscriber B:', data);
});
}, 1000)
// Subscriber A: 0.44524184251927656
// Subscriber A: 0.5802631630066313
// Subscriber A: 0.9792165506699135
// Subscriber A: 0.3239616040117268
// Subscriber A: 0.6845077617520203
// Subscriber B: 0.6845077617520203
// Subscriber A: 0.41269171141525707
// Subscriber B: 0.41269171141525707
// Subscriber A: 0.8211466186035139
// Subscriber B: 0.8211466186035139
```

Again, there are a few things happening here.

1. We create the ReplaySubject and specify that we only want to store the last 2 values, but no longer than a 100 ms
2. We start subscribing with Subscriber A
3. We start emiting Subject values every 200 ms. Subscriber A will pick this up and log every value that's being emited by the Subject.
4. We start subscribing with Subscriber B, but we do that after 1000 ms. This means that 5 values have already been emitted by the Subject before we start subscribing. When we created the Subject we specified that we wanted to store max 2 values, but no longer then 100ms. This means that after a 1000 ms, when Subscriber B starts subscribing, it will only receive 1 value as the subject emits values every 200ms.

The AsyncSubject

While the BehaviorSubject and ReplaySubject both store values, the AsyncSubject works a bit different. The AsyncSubject is aSubject variant where only the last value of the Observable execution is sent to its subscribers, and only when the execution completes. See the example code below:

```
import * as Rx from "rxjs";
    const subject = new Rx.AsyncSubject();
    // subscriber 1
    subject.subscribe((data) => {
    console.log('Subscriber A:', data);
    });
    subject.next(Math.random())
    subject.next(Math.random())
    subject.next(Math.random())
    // subscriber 2
    subject.subscribe((data) => {
    console.log('Subscriber B:', data);
    });
    subject.next(Math.random());
    subject.complete();
    // Subscriber A: 0.4447275989704571
    // Subscriber B: 0.4447275989704571
```

This time there's not a lot happening. But let's go over the steps:

1. We create the AsyncSubject
2. We subscribe to the Subject with Subscriber A
3. The Subject emits 3 values, still nothing hapening
4. We subscribe to the subject with Subscriber B
5. The Subject emits a new value, still nothing happening

6. The Subject completes. Now the values are emitted to the subscribers which both log the value.

Conclusion

The BehaviorSubject, ReplaySubject and AsyncSubject can still be used to multicast just like you would with a normal Subject. They do however have additional characteristics that are very handy in different scenario’s.

Subject vs ReplaySubject vs BehaviorSubject

I recently was helping another developer understand the difference between Subject, ReplaySubject, and BehaviourSubject. And thought that the following examples explain the differences perfectly.

Subject

A subject is like a turbocharged observable. It can almost be thought of an event message pump in that everytime a value is emitted, all subscribers receive the same value. The same analogy can be used when thinking about “late subscribers”. A Subject does not have a memory, therefore when a subscriber joins, it only receives the messages from that point on (It doesn’t get backdated values).

So as an example :

```
let mySubject = new Subject<number>();

mySubject.subscribe(x => console.log("First Subscription : " + x));

mySubject.next(1);

mySubject.next(2);

mySubject.next(3);

mySubject.subscribe(x => console.log("Second Subscription : " + x));

mySubject.next(4);
```

This will output :

```
First Subscription : 1

First Subscription : 2

First Subscription : 3

First Subscription : 4

Second Subscription : 4
```

Pretty straight forward. The first 3 values were output from the subject before the second subscription, so it doesn’t get those, it only gets new values going forward. Whereas the first subscription, as it subscribed before the first values were output, gets everything.

Now for the most part, you’ll end up using Subjects for the majority of your work. But there can be issues when you have async code that you can’t be sure that all subscriptions have been added before a value is emitted. For example :

```
let mySubject = new Subject<number>();

myAsyncMethod(mySubject);

mySubject.subscribe(x => console.log("First Subscription : " + x));
```

Imagine that “myAsyncMethod” is an asynchronous method that calls an API and emits a value on the given subject. This method may or may not complete before the subscription is added and therefore in rare cases, the subject did output a value, but you weren’t subscribed in time. These sort of race conditions on subscribing is a big cause of headaches when using plain Subjects.

ReplaySubject

That’s where ReplaySubject comes in. Imagine the same code, but using a ReplaySubject :

```
let mySubject = new ReplaySubject<number>();

mySubject.subscribe(x => console.log("First Subscription : " + x));

mySubject.next(1);

mySubject.next(2);

mySubject.next(3);

mySubject.subscribe(x => console.log("Second Subscription : " + x));

mySubject.next(4);
```

This outputs :

```
First Subscription : 1

First Subscription : 2

First Subscription : 3

Second Subscription : 1

Second Subscription : 2

Second Subscription : 3
```

```
First Subscription : 4

Second Subscription : 4
```

Notice how we get the first 3 values output on the first subscription. Then immediately as the Second Subscription joins, it also outputs the first 3 values, even though when they were emitted, the second subscriber had not yet joined the party. Then going forward, both subscribers emit the 4th value.

So what’s going on here? It’s actually quite simple. A ReplaySubject remembers the previous X values output, and on any new subscription, immediately “replays” those values to the new subscription so they can catch up. I say previous “X” values because by default, a ReplaySubject will remember *all* previous values, but you can configure this to only remember so far back.

For example :

```
let mySubject = new ReplaySubject(2);
```

This will remember only the last 2 values, and replay these to any new subscribers. This can be an important performance impact as replaying a large amount of values could cause any new subscriptions to really lag the system (Not to mention constantly holding those values in memory).

Back to our problem async code with Subject. If we change it to a ReplaySubject :

```
let mySubject = new ReplaySubject<number>();

myAsyncMethod(mySubject);

mySubject.subscribe(x => console.log("First Subscription : " + x));
```

Then it actually doesn’t matter if myAsyncMethod finishes before the subscription is added as the value will always be replayed to the subscription. Pretty nifty!

BehaviorSubject

A BehaviorSubject can sometimes be thought of a type of ReplaySubject, but with additional functionality (Or limitations depending on how you look at it).

If you think of a BehaviorSubject as simply being a ReplaySubject with a buffersize of 1 (That is, they will only replay the last value), then you're half way there to understanding BehaviorSubjects. The one large caveat is that BehaviourSubjects *require* an initial value to be emitted.

For example :

```
let mySubject = new BehaviorSubject<number>(1);

mySubject.subscribe(x => console.log("First Subscription : " + x));

mySubject.next(2);

mySubject.next(3);

mySubject.subscribe(x => console.log("Second Subscription : " + x));

mySubject.next(4);
```

This outputs :

```
First Subscription : 1

First Subscription : 2

First Subscription : 3

Second Subscription : 3

First Subscription : 4

Second Subscription : 4
```

So again, we have the ReplaySubject type functionality that when the second subscriber joins, it immediately outputs the last value of 3. But we also have to specify an initial value of 1 when creating the BehaviorSubject.

But why is an initial value important? Because you can also do things like so :

```
let mySubject = new BehaviorSubject<number>(1);

console.log(mySubject.value);
```

Notice we can just call mySubject.value and get the current value as a synchronize action. For this to work, we always need a value available, hence why an initial value is required. Again, if you don’t think that you can provide an initial output value, then you should use a ReplaySubject with a buffer size of 1 instead.