Extracted from:

Reactive Programming with RxJS

Untangle Your Asynchronous JavaScript Code

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Reactive Programming with RxJS

Untangle Your Asynchronous JavaScript Code

Sergi Mansilla

edited by Rebecca Gulick

Reactive Programming with RxJS

Untangle Your Asynchronous JavaScript Code

Sergi Mansilla

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

Rebecca Gulick (editor)
Potomac Indexing, LLC (index)
Candace Cunningham (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)
Ellie Callahan (support)

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-129-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P2.0—August 2016

RxJS's Subject Class

A Subject is a type that implements both Observer and Observable types. As an Observer, it can subscribe to Observables, and as an Observable it can produce values and have Observers subscribe to it.

In some scenarios a single Subject can do the work of a combination of Observers and Observables. For example, for making a proxy object between a data source and the Subject's listeners, we could use this:

spaceship_reactive/subjects.js

```
var subject = new Rx.Subject();
var source = Rx.Observable.interval(300)
  .map(function(v) { return 'Interval message #' + v; })
  .take(5);

source.subscribe(subject);

var subscription = subject.subscribe(
  function onNext(x) { console.log('onNext: ' + x); },
  function onError(e) { console.log('onError: ' + e.message); },
  function onCompleted() { console.log('onCompleted'); }
);

subject.onNext('Our message #1');
subject.onNext('Our message #2');

setTimeout(function() {
  subject.onCompleted();
}, 1000);
```

Output:

```
❮ onNext: Our message #1
onNext: Our message #2
onNext: Interval message #0
onNext: Interval message #1
onNext: Interval message #2
onCompleted
```

In the preceding example we create a new Subject and a source Observable that emits an integer every 300 milliseconds. Then we subscribe the Subject to the Observable. After that, we subscribe an Observer to the Subject itself. The Subject now behaves as an Observable.

Next we make the Subject emit values of its own (message1 and message2). In the final result, we get the Subject's own messages and then the proxied values from the source Observable. The values from the Observable come later because they are asynchronous, whereas we made the Subject's own values immediate. Notice that even if we tell the source Observable to take the first five values, the output shows only the first three. That's because after one second we call onCompleted on the Subject. This finishes the notifications to all subscriptions and overrides the take operator in this case.

The Subject class provides the base for creating more specialized Subjects. In fact, RxJS comes with some interesting ones: AsyncSubject, ReplaySubject, and BehaviorSubject.

AsyncSubject

AsyncSubject emits the last value of a sequence only if the sequence completes. This value is then cached forever, and any Observer that subscribes after the value has been emitted will receive it right away. AsyncSubject is convenient for asynchronous operations that return a single value, such as Ajax requests.

Let's see a simple example of an AsyncSubject subscribing to a range:

spaceship_reactive/subjects.js

```
var delayedRange = Rx.Observable.range(0, 5).delay(1000);
var subject = new Rx.AsyncSubject();

delayedRange.subscribe(subject);

subject.subscribe(
  function onNext(item) { console.log('Value:', item); },
  function onError(err) { console.log('Error:', err); },
  function onCompleted() { console.log('Completed.'); }
);
```

In that example, delayedRange emits the values 0 to 4 after a delay of a second. Then we create a new AsyncSubject subject and subscribe it to delayedRange. The output is the following:

```
Value: 4
Completed.
```

As expected, we get only the last value that the Observer emits. Let's now use AsyncSubject for a more realistic scenario. We'll retrieve some remote content:

spaceship_reactive/subjects.js

```
function getProducts(url) {
  var subject;

① return Rx.Observable.create(function(observer) {
    if (!subject) {
      subject = new Rx.AsyncSubject();
②     Rx.DOM.get(url).subscribe(subject);
    }
③   return subject.subscribe(observer);
  });
}
4 var products = getProducts('/products');
// Will trigger request and receive the response when read
5 products.subscribe(
  function onNext(result) { console.log('Result 1:', result.response); },
  function onError(error) { console.log('ERROR', error); }
);

// Will receive the result immediately because it's cached
6 setTimeout(function() {
  products.subscribe(
    function onNext(result) { console.log('Result 2:', result.response); },
    function onError(error) { console.log('ERROR', error); }
  );
}, 5000);
```

In this code, when getProducts is called with a URL, it returns an Observer that emits the result of the HTTP GET request. Here's how it breaks down:

1. getProducts returns an Observable sequence. We create it here.
2. If we haven't created an AsyncSubject yet, we create it and subscribe it to the Observable that Rx.DOM.Request.get(url) returns.
3. We subscribe the Observer to the AsyncSubject. Every time an Observer subscribes to the Observable, it will actually be subscribed to the AsyncSubject, which is acting as a proxy between the Observable retrieving the URL and the Observers.
4. We create the Observable that retrieves the URL "products" and store it in the products variable.
5. This is the first subscription and will kick off the URL retrieval and log the results when the URL is retrieved.
6. This is the second subscription, which runs five seconds after the first one. Since at that time the URL has already been retrieved, there's no need for another network request. It will receive the result of the request immediately because it is already stored in the AsyncSubject subject.

The interesting bit is that we're using an AsyncSubject that subscribes to the Rx.DOM.Request.get Observable. Because AsyncSubject caches the last result, any subsequent subscription to products will receive the result right away, without causing another network request. We can use AsyncSubject whenever we expect a single result and want to hold onto it.

Joe asks:
Does That Mean AsyncSubject Acts Like a Promise?

Indeed.

AsyncSubject represents the result of an asynchronous action, and you can use it as a substitute for a promise. The difference internally is that a promise will only ever process a single value, whereas AsyncSubject processes all values in a sequence, only ever emitting (and caching) the last one.

Being able to so easily simulate promises shows the flexibility of the RxJS model. (Even without AsyncSubject, it would be pretty easy to simulate a promise using Observables.)

BehaviorSubject

When an Observer subscribes to a BehaviorSubject, it receives the last emitted value and then all the subsequent values. BehaviorSubject requires that we provide a starting value, so that all Observers will always receive a value when they subscribe to a BehaviorSubject.

Imagine we want to retrieve a remote file and print its contents on an HTML page, but we want placeholder text while we wait for the contents. We can use a BehaviorSubject for this:

spaceship_reactive/behavior_subject.js

```
var subject = new Rx.BehaviorSubject('Waiting for content');

subject.subscribe(
  function(result) {
    document.body.textContent = result.response || result;
  },
  function(err) {
    document.body.textContent = 'There was an error retrieving content';
  }
);

Rx.DOM.get('/remote/content').subscribe(subject);
```

In the code, we initialize a new BehaviorSubject with our placeholder content. Then we subscribe to it and change the HTML body content in both onNext and onError, depending on the result.

Now the HTML body contains our placeholder text, and it will stay that way until the Subject emits a new value. Finally, we request the resource we want and we subscribe our Subject to the resulting Observer.

BehaviorSubject guarantees that there will always be at least one value emitted, because we provide a default value in its constructor. Once the BehaviorSubject completes it won't emit any more values, freeing the memory used by the cached value.

ReplaySubject

A ReplaySubject caches its values and re-emits them to any Observer that subscribes late to it. Unlike with AsyncSubject, the sequence doesn't need to be completed for this to happen.

Subject	ReplaySubject

`var subject = new Rx.Subject();`
 	
`subject.onNext(1);`
 	
`subject.subscribe(function(n) {`
`  console.log('Received value:', n);`
`});`
 	
`subject.onNext(2);`
`subject.onNext(3);`
	`var subject = new Rx.ReplaySubject();`
	
	`subject.onNext(1);`
	
	`subject.subscribe(function(n) {`
	`  console.log('Received value:', n);`
	`});`
	
	`subject.onNext(2);`
	`subject.onNext(3);`

❮ `Received value: 2`
`Received value: 3`
	❮ `Received value: 1`
	`Received value: 2`
	`Received value: 3`

ReplaySubject is useful to make sure that Observers get all the values emitted by an Observable from the start. It spares us from writing messy code that caches previous values, saving us from nasty concurrency-related bugs.

Of course, to accomplish that behavior ReplaySubject caches all values in memory. To prevent it from using too much memory, we can limit the amount of data it stores by buffer size or window of time, or by passing particular parameters to the constructor.

The first parameter to the constructor of ReplaySubject takes a number that represents how many values we want to buffer:

```
var subject = new Rx.ReplaySubject(2); // Buffer size of 2

subject.onNext(1);
subject.onNext(2);
subject.onNext(3);

subject.subscribe(function(n) {
  console.log('Received value:', n);
});
```

```
❮ Received value: 2
  Received value: 3
```

The second parameter takes a number that represents the time in milliseconds during which we want to buffer values:

```
var subject = new Rx.ReplaySubject(null, 200); // Buffer size of 200ms

setTimeout(function() { subject.onNext(1); }, 100);
setTimeout(function() { subject.onNext(2); }, 200);
setTimeout(function() { subject.onNext(3); }, 300);
setTimeout(function() {
  subject.subscribe(function(n) {
    console.log('Received value:', n);
  });

  subject.onNext(4);
}, 350);
```

In this example we set a buffer based on time, instead of the number of values. Our ReplaySubject will cache values that were emitted up to 200 milliseconds ago. We emit three values, each separated by 100 milliseconds, and after 350 milliseconds we subscribe an Observer and we emit yet another value. At the moment of the subscription the items cached are *2* and *3*, because *1* happened too long ago (around 250 milliseconds ago), so it is no longer cached.

Subjects are a powerful tool that can save you a lot of time. They provide great solutions to common scenarios like caching and repeating. And since at their core they are just Observables and Observers, you don’t need to learn anything new.