Table of Contents

RxJS v4.0

Reactive Extensions (Rx) is a library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators.

Data sequences can take many forms, such as a stream of data from a file or web service, web services requests, system notifications, or a series of events such as user input.

Reactive Extensions represents all these data sequences as observable sequences. An application can subscribe to these observable sequences to receive asynchronous notifications as new data arrive.

RxJS has no dependencies which complements and interoperates smoothly with both synchronous data streams such as iterable objects in JavaScript and single-value asynchronous computations such as Promises as the following diagram shows:

	Single return value	Mutiple return values
Pull/Synchronous/Interactive	Object	Iterables (Array | Set | Map | Object)
Push/Asynchronous/Reactive	Promise	Observable

To put it more concretely, if you know how to program against Arrays using the Array#extras, then you already know how to use RxJS!

Example code showing how similar high-order functions can be applied to an Array and an Observable

Iterable	Observable
| ```
getDataFromLocalMemory()
    .filter (s => s != null)
    .map(s => `${s} transformed`)
    .forEach(s => console.log(`next => ${s}`))
``` | ```
getDataFromNetwork()
    .filter (s => s != null)
    .map(s => `${s} transformed`)
    .subscribe(s => console.log(`
``` |

Why RxJS?

One question you may ask yourself, is why RxJS? What about Promises? Promises are good for solving asynchronous operations such as querying a service with an XMLHttpRequest, where the expected behavior is one value and then completion. The Reactive Extensions for JavaScript unifies both the world of Promises, callbacks as well as evented data such as DOM Input, Web Workers, Web Sockets. Once we have unified these concepts, this enables rich composition.

To give you an idea about rich composition, we can create an autocompletion service which takes the user input from a text input and then query a service, making sure not to flood the service with calls for every key stroke, but instead allow to go at a more natural pace.

First, we'll reference the JavaScript files, including jQuery, although RxJS has no dependencies on jQuery...

```html
<script src="http://code.jquery.com/jquery.js"></script>
<script src="rx.lite.js"></script>
```

Next, we'll get the user input from an input, listening to the keyup event by using the `Rx.Observable.fromEvent` method. This will either use the event binding from jQuery, Zepto, AngularJS and Ember.js if available, and if not, falls back to the native event binding. This gives you consistent ways of thinking of events depending on your framework, so there are no surprises.

part1.js

```js
var $input = $('#input'),
    $results = $('#results');

/* Only get the value from each key up */
var keyups = Rx.Observable.fromEvent($input, 'keyup')
    .map(e => e.target.value)
    .filter(text => text.length > 2);

/* Now throttle/debounce the input for 500ms */
var throttled = keyups.throttle(500 /* ms */);

/* Now get only distinct values, so we eliminate the arrows and other control characters
var distinct = throttled.distinctUntilChanged();
```

Now, let's query Wikipedia! In RxJS, we can instantly bind to any Promises A+ implementation through the `Rx.Observable.fromPromise` method or by just directly returning it, and we wrap it for you.

```
function searchWikipedia (term) {
    return $.ajax({
        url: 'http://en.wikipedia.org/w/api.php',
        dataType: 'jsonp',
        data: {
            action: 'opensearch',
            format: 'json',
            search: term
        }
    }).promise();
}
```

Once that is created, now we can tie together the distinct throttled input and then query the service. In this case, we'll call `flatMapLatest` to get the value and ensure that we're not introducing any out of order sequence calls.

```
var suggestions = distinct.flatMapLatest(searchWikipedia);
```

Finally, we call the subscribe method on our observable sequence to start pulling data.

```
suggestions.subscribe(data => {
    var res = data[1];

    /* Do something with the data like binding */
    $results.empty();

    $.each(res, (_, value) => $('<li>' + value + '</li>').appendTo($results));
}, error => {
    /* handle any errors */
    $results.empty();

    $('<li>Error: ' + error + '</li>').appendTo($results);
});
```

RxJS Design Guidelines

Introduction

This document describes guidelines that aid in developing applications and libraries that use the Reactive Extensions for RxJS library.

The guidelines listed in this document have evolved over time by the Rx team during the development of the RxJS library.

As RxJS continues to evolve, these guidelines will continue to evolve with it. Make sure you have the latest version of this document.

All information described in this document is merely a set of guidelines to aid development. These guidelines do not constitute an absolute truth. They are patterns that the team found helpful; not rules that should be followed blindly. There are situations where certain guidelines do not apply. The team has tried to list known situations where this is the case. It is up to each individual developer to decide if a certain guideline makes sense in each specific situation.

The guidelines in this document are listed in no particular order. There is neither total nor partial ordering in these guidelines.

Please contact us through the RxJS Issues for feedback on the guidelines, as well as questions on whether certain guidelines are applicable in specific situations.

The introduction to Reactive Programming you've been missing

(by @andrestaltz)

So you're curious in learning this new thing called Reactive Programming, particularly its variant comprising of Rx, Bacon.js, RAC, and others.

Learning it is hard, even harder by the lack of good material. When I started, I tried looking for tutorials. I found only a handful of practical guides, but they just scratched the surface and never tackled the challenge of building the whole architecture around it. Library documentations often don't help when you're trying to understand some function. I mean, honestly, look at this:

> **Rx.Observable.prototype.flatMapLatest(selector, [thisArg])**
>
> Projects each element of an observable sequence into a new sequence of observable sequences by incorporating the element's index and then transforms an observable sequence of observable sequences into an observable sequence producing values only from the most recent observable sequence.

Holy cow.

I've read two books, one just painted the big picture, while the other dived into how to use the Reactive library. I ended up learning Reactive Programming the hard way: figuring it out while building with it. At my work in Futurice I got to use it in a real project, and had the support of some colleagues when I ran into troubles.

The hardest part of the learning journey is **thinking in Reactive**. It's a lot about letting go of old imperative and stateful habits of typical programming, and forcing your brain to work in a different paradigm. I haven't found any guide on the internet in this aspect, and I think the world deserves a practical tutorial on how to think in Reactive, so that you can get started. Library documentation can light your way after that. I hope this helps you.

"What is Reactive Programming?"

There are plenty of bad explanations and definitions out there on the internet. Wikipedia is too generic and theoretical as usual. Stackoverflow's canonical answer is obviously not suitable for newcomers. Reactive Manifesto sounds like the kind of thing you show to your project manager or the businessmen at your company. Microsoft's Rx terminology "Rx = Observables + LINQ + Schedulers" is so heavy and Microsoftish that most of us are left confused. Terms like "reactive" and "propagation of change" don't convey anything specifically different to what your typical MV* and favorite language already does. Of course my framework views react to the models. Of course change is propagated. If it wouldn't, nothing would be rendered.

So let's cut the bullshit.
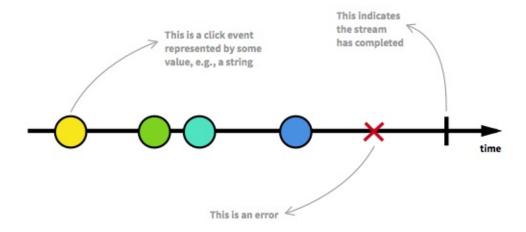
Reactive programming is programming with asynchronous data streams.

In a way, this isn't anything new. Event buses or your typical click events are really an asynchronous event stream, on which you can observe and do some side effects. Reactive is that idea on steroids. You are able to create data streams of anything, not just from click and hover events. Streams are cheap and ubiquitous, anything can be a stream: variables,

user inputs, properties, caches, data structures, etc. For example, imagine your Twitter feed would be a data stream in the same fashion that click events are. You can listen to that stream and react accordingly.

On top of that, you are given an amazing toolbox of functions to combine, create and filter any of those streams. That's where the "functional" magic kicks in. A stream can be used as an input to another one. Even multiple streams can be used as inputs to another stream. You can *merge* two streams. You can *filter* a stream to get another one that has only those events you are interested in. You can *map* data values from one stream to another new one.

If streams are so central to Reactive, let's take a careful look at them, starting with our familiar "clicks on a button" event stream.

A stream is a sequence of **ongoing events ordered in time**. It can emit three different things: a value (of some type), an error, or a "completed" signal. Consider that the "completed" takes place, for instance, when the current window or view containing that button is closed.

We capture these emitted events only **asynchronously**, by defining a function that will execute when a value is emitted, another function when an error is emitted, and another function when 'completed' is emitted. Sometimes these last two can be omitted and you can just focus on defining the function for values. The "listening" to the stream is called **subscribing**. The functions we are defining are observers. The stream is the subject (or "observable") being observed. This is precisely the Observer Design Pattern.

An alternative way of drawing that diagram is with ASCII, which we will use in some parts of this tutorial:

```
--a---b-c---d---X---|->

a, b, c, d are emitted values
X is an error
| is the 'completed' signal
---> is the timeline
```
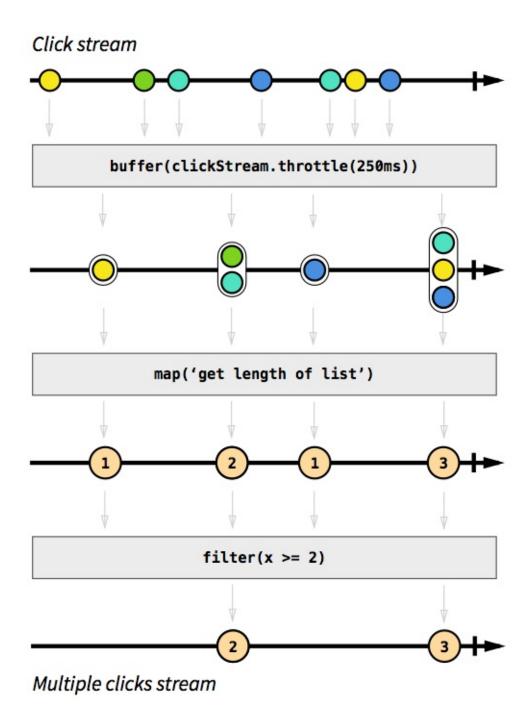
Since this feels so familiar already, and I don't want you to get bored, let's do something new: we are going to create new click event streams transformed out of the original click event stream.

First, let's make a counter stream that indicates how many times a button was clicked. In common Reactive libraries, each stream has many functions attached to it, such as `map`, `filter`, `scan`, etc. When you call one of these functions, such as `clickStream.map(f)`, it returns a **new stream** based on the click stream. It does not modify the original click stream in any way. This is a property called **immutability**, and it goes together with Reactive streams just like pancakes are good with syrup. That allows us to chain functions like `clickStream.map(f).scan(g)`:

```
   clickStream: ---c----c--c----c------c-->
                vvvvv map(c becomes 1) vvvv
                ---1----1--1----1------1-->
                vvvvvvvvv scan(+) vvvvvvvvv
 counterStream: ---1----2--3----4------5-->
```

The `map(f)` function replaces (into the new stream) each emitted value according to a function `f` you provide. In our case, we mapped to the number 1 on each click. The `scan(g)` function aggregates all previous values on the stream, producing value `x = g(accumulated, current)`, where `g` was simply the add function in this example. Then, `counterStream` emits the total number of clicks whenever a click happens.

To show the real power of Reactive, let's just say that you want to have a stream of "double click" events. To make it even more interesting, let's say we want the new stream to consider triple clicks as double clicks, or in general, multiple clicks (two or more). Take a deep breath and imagine how you would do that in a traditional imperative and stateful fashion. I bet it sounds fairly nasty and involves some variables to keep state and some fiddling with time intervals.

Well, in Reactive it's pretty simple. In fact, the logic is just 4 lines of code. But let's ignore code for now. Thinking in diagrams is the best way to understand and build streams, whether you're a beginner or an expert.

Click stream

Multiple clicks stream

Grey boxes are functions transforming one stream into another. First we accumulate clicks in lists, whenever 250 milliseconds of "event silence" has happened (that's what `buffer(stream.throttle(250ms))` does, in a nutshell. Don't worry about understanding the details at this point, we are just demoing Reactive for now). The result is a stream of lists, from which we apply `map()` to map each list to an integer matching the length of that list. Finally, we ignore `1` integers using the `filter(x >= 2)` function. That's it: 3 operations to produce our intended stream. We can then subscribe ("listen") to it to react accordingly how we wish.

I hope you enjoy the beauty of this approach. This example is just the tip of the iceberg: you can apply the same operations on different kinds of streams, for instance, on a stream of API responses; on the other hand, there are many other functions available.

"Why should I consider adopting RP?"

Reactive Programming raises the level of abstraction of your code so you can focus on the interdependence of events that define the business logic, rather than having to constantly fiddle with a large amount of implementation details. Code in RP will likely be more concise.

The benefit is more evident in modern webapps and mobile apps that are highly interactive with a multitude of UI events related to data events. 10 years ago, interaction with web pages was basically about submitting a long form to the backend and performing simple rendering to the frontend. Apps have evolved to be more real-time: modifying a single form field can automatically trigger a save to the backend, "likes" to some content can be reflected in real time to other connected users, and so forth.

Apps nowadays have an abundancy of real-time events of every kind that enable a highly interactive experience to the user. We need tools for properly dealing with that, and Reactive Programming is an answer.

Thinking in RP, with examples

Let's dive into the real stuff. A real-world example with a step-by-step guide on how to think in RP. No synthetic examples, no half-explained concepts. By the end of this tutorial we will have produced real functioning code, while knowing why we did each thing.

I picked **JavaScript** and **RxJS** as the tools for this, for a reason: JavaScript is the most familiar language out there at the moment, and the Rx* library family is widely available for many languages and platforms (.NET, Java, Scala, Clojure, JavaScript, Ruby, Python, C++, Objective-C/Cocoa, Groovy, etc). So whatever your tools are, you can concretely benefit by following this tutorial.

Implementing a "Who to follow" suggestions box

In Twitter there is this UI element that suggests other accounts you could follow:

We are going to focus on imitating its core features, which are:

- On startup, load accounts data from the API and display 3 suggestions
- On clicking "Refresh", load 3 other account suggestions into the 3 rows
- On click 'x' button on an account row, clear only that current account and display another
- Each row displays the account's avatar and links to their page

We can leave out the other features and buttons because they are minor. And, instead of Twitter, which recently closed its API to the unauthorized public, let's build that UI for following people on Github. There's a Github API for getting users.

The complete code for this is ready at http://jsfiddle.net/staltz/8jFJH/48/ in case you want to take a peak already.

Request and response

How do you approach this problem with Rx? Well, to start with, (almost) *everything can be a stream*. That's the Rx mantra. Let's start with the easiest feature: "on startup, load 3 accounts data from the API". There is nothing special here, this is simply about (1) doing a request, (2) getting a response, (3) rendering the response. So let's go ahead and represent our requests as a stream. At first this will feel like overkill, but we need to start from the basics, right?

On startup we need to do only one request, so if we model it as a data stream, it will be a stream with only one emitted value. Later, we know we will have many requests happening, but for now, it is just one.

```
--a------|->

Where a is the string 'https://api.github.com/users'
```

This is a stream of URLs that we want to request. Whenever a request event happens, it tells us two things: when and what. "When" the request should be executed is when the event is emitted. And "what" should be requested is the value emitted: a string containing the URL.

To create such stream with a single value is very simple in Rx*. The official terminology for a stream is "Observable", for the fact that it can be observed, but I find it to be a silly name, so I call it *stream*.

```
var requestStream = Rx.Observable.just('https://api.github.com/users');
```

But now, that is just a stream of strings, doing no other operation, so we need to somehow make something happen when that value is emitted. That's done by subscribing to the stream.

```
requestStream.subscribe(function(requestUrl) {
  // execute the request
  jQuery.getJSON(requestUrl, function(responseData) {
    // ...
  });
}
```

Notice we are using a jQuery Ajax callback (which we assume you should know already) to handle the asynchronicity of the request operation. But wait a moment, Rx is for dealing with **asynchronous** data streams. Couldn't the response for that request be a stream containing the data arriving at some time in the future? Well, at a conceptual level, it sure looks like it, so let's try that.

```
requestStream.subscribe(function(requestUrl) {
  // execute the request
  var responseStream = Rx.Observable.create(function (observer) {
    jQuery.getJSON(requestUrl)
    .done(function(response) { observer.onNext(response); })
    .fail(function(jqXHR, status, error) { observer.onError(error); })
    .always(function() { observer.onCompleted(); });
  });

  responseStream.subscribe(function(response) {
    // do something with the response
  });
}
```

What `Rx.Observable.create()` does is create your own custom stream by explicitly informing each observer (or in other words, a "subscriber") about data events (`onNext()`) or errors (`onError()`). What we did was just wrap that jQuery Ajax Promise. **Excuse me, does this mean that a Promise is an Observable?**

Yes.

Observable is Promise++. In Rx you can easily convert a Promise to an Observable by doing `var stream = Rx.Observable.fromPromise(promise)` , so let's use that. The only difference is that Observables are not Promises/A+ compliant, but conceptually there is no clash. A Promise is simply an Observable with one single emitted value. Rx streams go beyond promises by allowing many returned values.

This is pretty nice, and shows how Observables are at least as powerful as Promises. So if you believe the Promises hype, keep an eye on what Rx Observables are capable of.
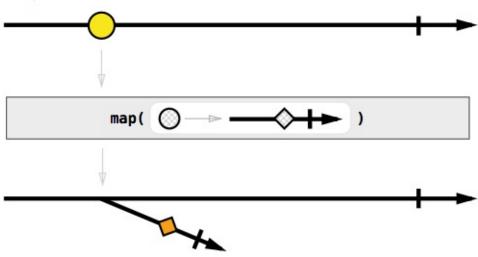
Now back to our example, if you were quick to notice, we have one `subscribe()` call inside another, which is somewhat akin to callback hell. Also, the creation of `responseStream` is dependent on `requestStream` . As you heard before, in Rx there are simple mechanisms for transforming and creating new streams out of others, so we should be doing that.

The one basic function that you should know by now is `map(f)` , which takes each value of stream A, applies `f()` on it, and produces a value on stream B. If we do that to our request and response streams, we can map request URLs to response Promises (disguised as streams).

```
var responseMetastream = requestStream
  .map(function(requestUrl) {
    return Rx.Observable.fromPromise(jQuery.getJSON(requestUrl));
  });
```

Then we will have created a beast called "*metastream*": a stream of streams. Don't panic yet. A metastream is a stream where each emitted value is yet another stream. You can think of it as pointers): each emitted value is a *pointer* to another stream. In our example, each request URL is mapped to a pointer to the promise stream containing the corresponding response.
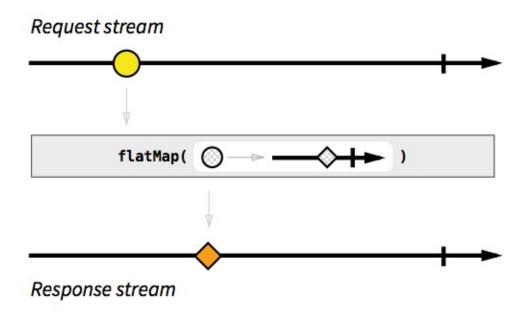
A metastream for responses looks confusing, and doesn't seem to help us at all. We just want a simple stream of responses, where each emitted value is a JSON object, not a 'Promise' of a JSON object. Say hi to Mr. Flatmap: a version of `map()` that "flattens" a metastream, by emitting on the "trunk" stream everything that will be emitted on "branch" streams. Flatmap is not a "fix" and metastreams are not a bug, these are really the tools for dealing with asynchronous responses in Rx.

```
var responseStream = requestStream
  .flatMap(function(requestUrl) {
    return Rx.Observable.fromPromise(jQuery.getJSON(requestUrl));
  });
```

Request stream

flatMap( )

Response stream

Nice. And because the response stream is defined according to request stream, **if** we have later on more events happening on request stream, we will have the corresponding response events happening on response stream, as expected:

```
requestStream:   --a-----b--c------------|->
responseStream: -----A--------B-----C---|->

(lowercase is a request, uppercase is its response)
```

Now that we finally have a response stream, we can render the data we receive:

```javascript
responseStream.subscribe(function(response) {
  // render `response` to the DOM however you wish
});
```

Joining all the code until now, we have:

```javascript
var requestStream = Rx.Observable.just('https://api.github.com/users');

var responseStream = requestStream
  .flatMap(function(requestUrl) {
    return Rx.Observable.fromPromise(jQuery.getJSON(requestUrl));
  });

responseStream.subscribe(function(response) {
  // render `response` to the DOM however you wish
});
```

The refresh button

I did not yet mention that the JSON in the response is a list with 100 users. The API only allows us to specify the page offset, and not the page size, so we're using just 3 data objects and wasting 97 others. We can ignore that problem for now, since later on we will see how to cache the responses.

Everytime the refresh button is clicked, the request stream should emit a new URL, so that we can get a new response. We need two things: a stream of click events on the refresh button (mantra: anything can be a stream), and we need to change the request stream to depend on the refresh click stream. Gladly, RxJS comes with tools to make Observables from event listeners.

```
var refreshButton = document.querySelector('.refresh');
var refreshClickStream = Rx.Observable.fromEvent(refreshButton, 'click');
```

Since the refresh click event doesn't itself carry any API URL, we need to map each click to an actual URL. Now we change the request stream to be the refresh click stream mapped to the API endpoint with a random offset parameter each time.

```
var requestStream = refreshClickStream
  .map(function() {
    var randomOffset = Math.floor(Math.random()*500);
    return 'https://api.github.com/users?since=' + randomOffset;
  });
```

Because I'm dumb and I don't have automated tests, I just broke one of our previously built features. A request doesn't happen anymore on startup, it happens only when the refresh is clicked. Urgh. I need both behaviors: a request when *either* a refresh is clicked *or* the webpage was just opened.

We know how to make a separate stream for each one of those cases:

```
var requestOnRefreshStream = refreshClickStream
  .map(function() {
    var randomOffset = Math.floor(Math.random()*500);
    return 'https://api.github.com/users?since=' + randomOffset;
  });

var startupRequestStream = Rx.Observable.just('https://api.github.com/users');
```

But how can we "merge" these two into one? Well, there's `merge()` . Explained in the diagram dialect, this is what it does:

```
stream A: ---a--------e-----o----->
stream B: -----B---C-----D-------->
          vvvvvvvvv merge vvvvvvvvv
          ---a-B---C--e--D--o----->
```

It should be easy now:

```
var requestOnRefreshStream = refreshClickStream
  .map(function() {
    var randomOffset = Math.floor(Math.random()*500);
    return 'https://api.github.com/users?since=' + randomOffset;
  });

var startupRequestStream = Rx.Observable.just('https://api.github.com/users');

var requestStream = Rx.Observable.merge(
  requestOnRefreshStream, startupRequestStream
);
```

There is an alternative and cleaner way of writing that, without the intermediate streams.

```
var requestStream = refreshClickStream
  .map(function() {
    var randomOffset = Math.floor(Math.random()*500);
    return 'https://api.github.com/users?since=' + randomOffset;
  })
  .merge(Rx.Observable.just('https://api.github.com/users'));
```

Even shorter, even more readable:

```
var requestStream = refreshClickStream
  .map(function() {
    var randomOffset = Math.floor(Math.random()*500);
    return 'https://api.github.com/users?since=' + randomOffset;
  })
  .startWith('https://api.github.com/users');
```

The `startWith()` function does exactly what you think it does. No matter how your input stream looks like, the output stream resulting of `startWith(x)` will have `x` at the beginning. But I'm not DRY enough, I'm repeating the API endpoint string. One way to fix this is by moving the `startWith()` close to the `refreshClickStream`, to essentially "emulate" a refresh click on startup.

```
var requestStream = refreshClickStream.startWith('startup click')
  .map(function() {
    var randomOffset = Math.floor(Math.random()*500);
    return 'https://api.github.com/users?since=' + randomOffset;
  });
```

Nice. If you go back to the point where I "broke the automated tests", you should see that the only difference with this last approach is that I added the `startWith()`.

Modelling the 3 suggestions with streams

Until now, we have only touched a *suggestion* UI element on the rendering step that happens in the responseStream's `subscribe()`. Now with the refresh button, we have a problem: as soon as you click 'refresh', the current 3 suggestions are not cleared. New suggestions come in only after a response has arrived, but to make the UI look nice, we need to clean out the current suggestions when clicks happen on the refresh.

```
refreshClickStream.subscribe(function() {
  // clear the 3 suggestion DOM elements
});
```

No, not so fast, pal. This is bad, because we now have **two** subscribers that affect the suggestion DOM elements (the other one being `responseStream.subscribe()`), and that doesn't really sound like Separation of concerns. Remember the Reactive mantra?

So let's model a suggestion as a stream, where each emitted value is the JSON object containing the suggestion data. We will do this separately for each of the 3 suggestions. This is how the stream for suggestion #1 could look like:

```
var suggestion1Stream = responseStream
  .map(function(listUsers) {
    // get one random user from the list
    return listUsers[Math.floor(Math.random()*listUsers.length)];
  });
```

The others, `suggestion2Stream` and `suggestion3Stream` can be simply copy pasted from `suggestion1Stream` . This is not DRY, but it will keep our example simple for this tutorial, plus I think it's a good exercise to think how to avoid repetition in this case.

Instead of having the rendering happen in responseStream's subscribe(), we do that here:

```
suggestion1Stream.subscribe(function(suggestion) {
  // render the 1st suggestion to the DOM
});
```

Back to the "on refresh, clear the suggestions", we can simply map refresh clicks to `null` suggestion data, and include that in the `suggestion1Stream` , as such:

```
var suggestion1Stream = responseStream
  .map(function(listUsers) {
    // get one random user from the list
    return listUsers[Math.floor(Math.random()*listUsers.length)];
  })
  .merge(
    refreshClickStream.map(function(){ return null; })
  );
```

And when rendering, we interpret `null` as "no data", hence hiding its UI element.

```
suggestion1Stream.subscribe(function(suggestion) {
  if (suggestion === null) {
    // hide the first suggestion DOM element
  }
  else {
    // show the first suggestion DOM element
    // and render the data
  }
});
```

The big picture is now:

```
refreshClickStream: ----------o--------o---->
     requestStream: -r--------r--------r---->
    responseStream: ----R---------R------R-->
 suggestion1Stream: ----s-----N---s----N-s-->
 suggestion2Stream: ----q-----N---q----N-q-->
 suggestion3Stream: ----t-----N---t----N-t-->
```

Where `N` stands for `null`.

As a bonus, we can also render "empty" suggestions on startup. That is done by adding `startWith(null)` to the suggestion streams:

```
var suggestion1Stream = responseStream
  .map(function(listUsers) {
    // get one random user from the list
    return listUsers[Math.floor(Math.random()*listUsers.length)];
  })
  .merge(
    refreshClickStream.map(function(){ return null; })
  )
  .startWith(null);
```

Which results in:

```
refreshClickStream: ----------o---------o---->
     requestStream: -r--------r---------r---->
    responseStream: ----R----------R------R-->
 suggestion1Stream: -N--s-----N----s----N-s-->
 suggestion2Stream: -N--q-----N----q----N-q-->
 suggestion3Stream: -N--t-----N----t----N-t-->
```

Closing a suggestion and using cached responses

There is one feature remaining to implement. Each suggestion should have its own 'x' button for closing it, and loading another in its place. At first thought, you could say it's enough to make a new request when any close button is clicked:

```javascript
var close1Button = document.querySelector('.close1');
var close1ClickStream = Rx.Observable.fromEvent(close1Button, 'click');
// and the same for close2Button and close3Button

var requestStream = refreshClickStream.startWith('startup click')
  .merge(close1ClickStream) // we added this
  .map(function() {
    var randomOffset = Math.floor(Math.random()*500);
    return 'https://api.github.com/users?since=' + randomOffset;
  });
```

That does not work. It will close and reload *all* suggestions, rather than just only the one we clicked on. There are a couple of different ways of solving this, and to keep it interesting, we will solve it by reusing previous responses. The API's response page size is 100 users while we were using just 3 of those, so there is plenty of fresh data available. No need to request more.

Again, let's think in streams. When a 'close1' click event happens, we want to use the *most recently emitted* response on `responseStream` to get one random user from the list in the response. As such:

```
   requestStream: --r--------------->
  responseStream: ------R----------->
close1ClickStream: -----------c----->
suggestion1Stream: ------s-----s----->
```

In Rx* there is a combinator function called `combineLatest` that seems to do what we need. It takes two streams A and B as inputs, and whenever either stream emits a value, `combineLatest` joins the two most recently emitted values `a` and `b` from both streams

and outputs a value `c = f(x,y)`, where `f` is a function you define. It is better explained with a diagram:

```
stream A: --a-----------e--------i-------->
stream B: -----b----c--------d-------q---->
          vvvvvvvv combineLatest(f) vvvvvvv
          ----AB---AC--EC---ED--ID--IQ---->

where f is the uppercase function
```

We can apply combineLatest() on `close1ClickStream` and `responseStream`, so that whenever the close 1 button is clicked, we get the latest response emitted and produce a new value on `suggestion1Stream`. On the other hand, combineLatest() is symmetric: whenever a new response is emitted on `responseStream`, it will combine with the latest 'close 1' click to produce a new suggestion. That is interesting, because it allows us to simplify our previous code for `suggestion1Stream`, like this:

```javascript
var suggestion1Stream = close1ClickStream
  .combineLatest(responseStream,
    function(click, listUsers) {
      return listUsers[Math.floor(Math.random()*listUsers.length)];
    }
  )
  .merge(
    refreshClickStream.map(function(){ return null; })
  )
  .startWith(null);
```

One piece is still missing in the puzzle. The combineLatest() uses the most recent of the two sources, but if one of those sources hasn't emitted anything yet, combineLatest() cannot produce a data event on the output stream. If you look at the ASCII diagram above, you will see that the output has nothing when the first stream emitted value `a`. Only when the second stream emitted value `b` could it produce an output value.

There are different ways of solving this, and we will stay with the simplest one, which is simulating a click to the 'close 1' button on startup:

```
var suggestion1Stream = close1ClickStream.startWith('startup click') // we added this
  .combineLatest(responseStream,
    function(click, listUsers) {
      return listUsers[Math.floor(Math.random()*listUsers.length)];
    }
  )
  .merge(
    refreshClickStream.map(function(){ return null; })
  )
  .startWith(null);
```

Wrapping up

And we're done. The complete code for all this was:

```javascript
var refreshButton = document.querySelector('.refresh');
var refreshClickStream = Rx.Observable.fromEvent(refreshButton, 'click');

var closeButton1 = document.querySelector('.close1');
var close1ClickStream = Rx.Observable.fromEvent(closeButton1, 'click');
// and the same logic for close2 and close3

var requestStream = refreshClickStream.startWith('startup click')
  .map(function() {
    var randomOffset = Math.floor(Math.random()*500);
    return 'https://api.github.com/users?since=' + randomOffset;
  });

var responseStream = requestStream
  .flatMap(function (requestUrl) {
    return Rx.Observable.fromPromise($.ajax({url: requestUrl}));
  });

var suggestion1Stream = close1ClickStream.startWith('startup click')
  .combineLatest(responseStream,
    function(click, listUsers) {
      return listUsers[Math.floor(Math.random()*listUsers.length)];
    }
  )
  .merge(
    refreshClickStream.map(function(){ return null; })
  )
  .startWith(null);
// and the same logic for suggestion2Stream and suggestion3Stream

suggestion1Stream.subscribe(function(suggestion) {
  if (suggestion === null) {
    // hide the first suggestion DOM element
  }
  else {
    // show the first suggestion DOM element
    // and render the data
  }
});
```

You can see this working example at http://jsfiddle.net/staltz/8jFJH/48/

That piece of code is small but dense: it features management of multiple events with proper separation of concerns, and even caching of responses. The functional style made the code look more declarative than imperative: we are not giving a sequence of instructions to execute, we are just **telling what something is** by defining relationships between streams. For instance, with Rx we told the computer that `suggestion1Stream` *is the 'close 1' stream combined with one user from the latest response, besides being* `null` *when a refresh happens or program startup happened.*

Notice also the impressive absence of control flow elements such as `if` , `for` , `while` , and the typical callback-based control flow that you expect from a JavaScript application. You can even get rid of the `if` and `else` in the `subscribe()` above by using `filter()` if you want (I'll leave the implementation details to you as an exercise). In Rx, we have stream functions such as `map` , `filter` , `scan` , `merge` , `combineLatest` , `startWith` , and many more to control the flow of an event-driven program. This toolset of functions gives you more power in less code.

What comes next

If you think Rx* will be your preferred library for Reactive Programming, take a while to get acquainted with the big list of functions for transforming, combining, and creating Observables. If you want to understand those functions in diagrams of streams, take a look at RxJava's very useful documentation with marble diagrams. Whenever you get stuck trying to do something, draw those diagrams, think on them, look at the long list of functions, and think more. This workflow has been effective in my experience.

Once you start getting the hang of programming with Rx*, it is absolutely required to understand the concept of Cold vs Hot Observables. If you ignore this, it will come back and bite you brutally. You have been warned. Sharpen your skills further by learning real functional programming, and getting acquainted with issues such as side effects that affect Rx.*

But Reactive Programming is not just Rx*. There is Bacon.js which is intuitive to work with, without the quirks you sometimes encounter in Rx*. The Elm Language lives in its own category: it's a Functional Reactive Programming **language** that compiles to JavaScript + HTML + CSS, and features a time travelling debugger. Pretty awesome.

Rx works great for event-heavy frontends and apps. But it is not just a client-side thing, it works great also in the backend and close to databases. In fact, RxJava is a key component for enabling server-side concurrency in Netflix's API. Rx is not a framework restricted to one specific type of application or language. It really is a paradigm that you can use when programming any event-driven software.

If this tutorial helped you, tweet it forward.

When to use RxJS

Use RxJS for orchestrating asynchronous and event-based computations

Code that deals with more than one event or asynchronous computation gets complicated quickly as it needs to build a state-machine to deal with ordering issues. Next to this, the code needs to deal with successful and failure termination of each separate computation. This leads to code that doesn't follow normal control-flow, is hard to understand and hard to maintain.

RxJS makes these computations first-class citizens. This provides a model that allows for readable and composable APIs to deal with these asynchronous computations.

Sample

```javascript
var input = document.getElementById('input');
var dictionarySuggest = Rx.Observable.fromEvent(input, 'keyup')
  .map(() => input.value)
  .filter(text => !!text)
  .distinctUntilChanged()
  .throttle(250)
  .flatMapLatest(searchWikipedia)
  .subscribe(
    results => {
      list = [];
      list.concat(results.map(createItem));
    },
    err => logError(err)
  );
```

This sample models a common UI paradigm of receiving completion suggestions while the user is typing input.

RxJS creates an observable sequence that models an existing `keyup` event on the input.

It then places several filters and projections on top of the event to make the event only fire if a unique value has come through. (The `keyup` event fires for every key stroke, so also if the user presses left or right arrow, moving the cursor but not changing the input text).

Next it makes sure the event only gets fired after 250 milliseconds of activity by using the `throttle` operator. (If the user is still typing characters, this saves a potentially expensive lookup that will be ignored immediately).

In traditionally written programs, this throttling would introduce separate callbacks through a timer. This timer could potentially throw exceptions (certain timers have a maximum amount of operations in flight).

Once the user input has been filtered down it is time to perform the dictionary lookup. As this is usually an expensive operation (e.g. a request to a server on the other side of the world), this operation is itself asynchronous as well.

The `flatMap` / `selectMany` operator allows for easy combining of multiple asynchronous operations. It doesn't only combine success values; it also tracks any exceptions that happen in each individual operation.

In traditionally written programs, this would introduce separate callbacks and a place for exceptions occurring.

If the user types a new character while the dictionary operation is still in progress, we do not want to see the results of that operation anymore. The user has typed more characters leading to a more specific word, seeing old results would be confusing.

The `flatMapLatest` operation makes sure that the dictionary operation is ignored once a new `keyup` has been detected.

Finally we subscribe to the resulting observable sequence. Only at this time our execution plan will be used. We pass two functions to the `subscribe` call:

1. Receives the result from our computation.
2. Receives exceptions in case of a failure occurring anywhere along the computation.

When to ignore this guideline

If the application/library in question has very few asynchronous/event-based operations or has very few places where these operations need to be composed, the cost of depending on RxJS (redistributing the library as well as the learning curve) might outweigh the cost of manually coding these operations.

Use RxJS to deal with asynchronous sequences of data

Several other libraries exist to aid asynchronous operations in the JavaScript ecosystem. Even though these libraries are powerful, they usually work best on operations that return a single message. They usually do not support operations that produce multiple messages over the lifetime of the operation.

RxJS follows the following grammar: `onNext` `*` (`onCompleted` | `onError`)?. This allows for multiple messages to come in over time. This makes RxJS suitable for both operations that produce a single message, as well as operations that produce multiple messages.

```javascript
var fs = require('fs');
var Rx = require('rx');

// Read/write from stream implementation
function readAsync(fd, chunkSize) { /* impl */ }
function appendAsync(fd, buffer) { /* impl */ }
function encrypt(buffer) { /* impl */}

//open a 4GB file for asynchronous reading in blocks of 64K
var inFile = fs.openSync('4GBfile.txt', 'r+');
var outFile = fs.openSync('Encrypted.txt', 'w+');

readAsync(inFile, 2 << 15)
  .map(encrypt)
  .flatMap(data => appendAsync(outFile, data))
  .subscribe(
    () => {},
    err => {
      console.log('An error occurred while encrypting the file: %s', err.message);
      fs.closeSync(inFile);
      fs.closeSync(outFile);
    },
    () => {
      console.log('Successfully encrypted the file.');
      fs.closeSync(inFile);
      fs.closeSync(outFile);
    }
  );
```

In this sample, a 4 GB file is read in its entirety, encrypted and saved out to another file.

Reading the whole file into memory, encrypting it and writing out the whole file would be an expensive operation.

Instead, we rely on the fact that RxJS can produce many messages.

We read the file asynchronously in blocks of 64K. This produces an observable sequence of byte arrays. We then encrypt each block separately (for this sample we assume the encryption operation can work on separate parts of the file). Once the block is encrypted, it is immediately sent further down the pipeline to be saved to the other file. The `writeAsync` operation is an asynchronous operation that can process multiple messages.

When to ignore this guideline

If the application/library in question has very few operations with multiple messages, the cost of depending on RxJS (redistributing the library as well as the learning curve) might outweigh the cost of manually coding these operations.

The RxJS Contract

Assume the RxJS Grammar

Messages sent to instances of the `Observer` object follow the following grammar: onNext* (onCompleted | onError)?

This grammar allows observable sequences to send any amount (0 or more) of `onNext` messages to the subscribed observer instance, optionally followed by a single success (`onCompleted`) or failure (`onError`) message.

The single message indicating that an observable sequence has finished ensures that consumers of the observable sequence can deterministically establish that it is safe to perform cleanup operations.

A single failure further ensures that abort semantics can be maintained for operators that work on multiple observable sequences.

Sample

```
var count = 0;
xs.subscribe(
  () => count++,
  err => console.log('Error: %s', err.message),
  () => console.log('OnNext has been called %d times', count)
);
```

In this sample we safely assume that the total amount of calls to the OnNext method won't change once the OnCompleted method is called as the observable sequence follows the Rx grammar.

When to ignore this guideline

Ignore this guideline only when working with a non-conforming implementation of the Observable object.

Assume resources are cleaned up after an `onError` or `onCompleted` message

Paragraph 3.1 states that no more messages should arrive after an `onError` or `onCompleted` message. This makes it possible to cleanup any resource used by the subscription the moment an `onError` or `onCompleted` arrives. Cleaning up resources immediately will make sure that any side-effect occurs in a predictable fashion. It also makes sure that the runtime can reclaim these resources.

Sample

```
var fs = require('fs');
var Rx = require('rx');

function appendAsync(fd, buffer) { /* impl */ }

function openFile(path, flags) {
  var fd = fs.openSync(path, flags);
  return Rx.Disposable.create(() => fs.closeSync(fd));
}

Rx.Observable.
  using(
    () => openFile('temp.txt', 'w+'),
    fd => Rx.Observable.range(0, 10000).map(v => Buffer(v)).flatMap(buffer => appendAsync
  ).subscribe();
```

In this sample the Using operator creates a resource that will be disposed upon unsubscription. The Rx contract for cleanup ensures that unsubscription will be called automatically once an `onError` or `onCompleted` message is sent.

When to ignore this guideline

There are currently no known cases where to ignore this guideline.

Assume a best effort to stop all outstanding work on Unsubscribe

When unsubscribe is called on an observable subscription, the observable sequence will make a best effort attempt to stop all outstanding work. This means that any queued work that has not been started will not start.

Any work that is already in progress might still complete as it is not always safe to abort work that is in progress. Results from this work will not be signaled to any previously subscribed observer instances.

Sample 1

```
Observable.timer(2000).subscribe(...).dispose()
```

In this sample subscribing to the observable sequence generated by Timer will queue an action on the `Scheduler.timeout` scheduler to send out an `onNext` message in 2 seconds. The subscription then gets canceled immediately. As the scheduled action has not started yet, it will be removed from the scheduler.

Sample 2

```
Rx.Observable.startAsync(() => Q.delay(2000)).subscribe(...).dispose();
```

In this sample the `startAsync` operator will immediately schedule the execution of the lambda provided as its argument. The subscription registers the observer instance as a listener to this execution. As the lambda is already running once the subscription is disposed, it will keep running and its return value is ignored.
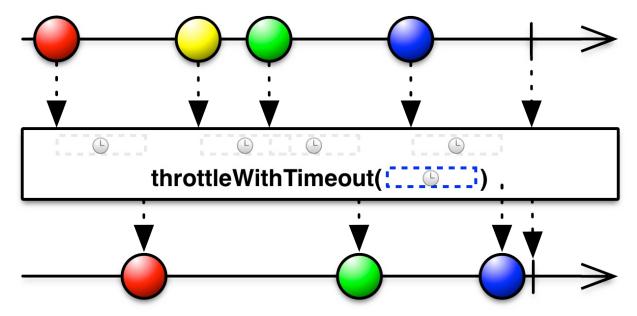
Using Rx

Consider drawing a Marble-diagram

Draw a marble-diagram of the observable sequence you want to create. By drawing the diagram, you will get a clearer picture on what operator(s) to use.

A marble-diagram is a diagram that shows event occurring over time. A marble diagram contains both input and output sequences(s).

By drawing the diagram we can see that we will need some kind of delay after the user input, before firing of another asynchronous call. The delay in this sample maps to the `throttle` operator. To create another observable sequence from an observable sequence we will use the `flatMap` or `selectMany` operator. This will lead to the following code:

```
var dictionarySuggest = userInput.throttle(250).flatMap(input => serverCall(input));
```

When to ignore this guideline

This guideline can be ignored if you feel comfortable enough with the observable sequence you want to write. However, even the Rx team members will still grab the whiteboard to draw a marble-diagram once in a while.

Consider passing multiple arguments to `subscribe`

For convenience, Rx provides an overload to the `subscribe` method that takes functions instead of an Observer argument.

The Observer object would require implementing all three methods (`onNext` , `onError` & `onCompleted`). The extensions to the `subscribe` method allow developers to use defaults chosen for each of these methods.

E.g. when calling the `subscribe` method that only has an `onNext` argument, the `onError` behavior will be to rethrow the exception on the thread that the message comes out from the observable sequence. The `onCompleted` behavior in this case is to do nothing.

In many situations, it is important to deal with the exception (either recover or abort the application gracefully).

Often it is also important to know that the observable sequence has completed successfully. For example, the application notifies the user that the operation has completed.

Because of this, it is best to provide all 3 arguments to the subscribe function.

RxJS also provides three convenience methods which only subscribe to the part of the sequence that is desired. The other handlers will default to their original behaviors. There are three of such functions:

- `subscribeOnNext` : for `onNext` messages only
- `subscribeOnError` : for `onError` messages only
- `subscribeOnCompleted` : for `onCompleted` messages only.

When to ignore this guideline

- When the observable sequence is guaranteed not to complete, e.g. an event such as keyup.
- When the observable sequence is guaranteed not to have `onError` messages (e.g. an event, a materialized observable sequence etc…).
- When the default behavior is the desirable behavior.

Consider passing a specific scheduler to concurrency introducing operators

Rather than using the `observeOn` operator to change the execution context on which the observable sequence produces messages, it is better to create concurrency in the right place to begin with. As operators parameterize introduction of concurrency by providing a scheduler argument overload, passing the right scheduler will lead to fewer places where the ObserveOn operator has to be used.

Sample

```
Rx.Observable.range(0, 90000, Rx.Scheduler.requestAnimationFrame).subscribe(draw);
```

In this sample, callbacks from the `range` operator will arrive by calling `window.requestAnimationFrame`. The default overload of `range` would place the `onNext` call on the `Rx.Scheduler.currentThread` which is used when recursive scheduling is required immediately. By providing the `Rx.Scheduler.requestAnimationFrame` scheduler, all messages from this observable sequence will originiate on the `window.requestAnimationFrame` callback.

When to ignore this guideline

When combining several events that originate on different execution contexts, use guideline 4.4 to put all messages on a specific execution context as late as possible.

Call the `observeOn` operator as late and in as few places as possible

By using the `observeOn` operator, an action is scheduled for each message that comes through the original observable sequence. This potentially changes timing information as well as puts additional stress on the system. Placing this operator later in the query will reduce both concerns.

Sample

```
var result = xs.throttle(1000)
  .flatMap(x => ys.takeUntil(zs).sample(250).map(y => x + y))
  .merge(ws)
  .filter(x => x < 10)
  .observeOn(Rx.Scheduler.requestAnimationFrame);
```

This sample combines many observable sequences running on many different execution contexts. The query filters out a lot of messages. Placing the `observeOn` operator earlier in the query would do extra work on messages that would be filtered out anyway. Calling the `observeOn` operator at the end of the query will create the least performance impact.

When to ignore this guideline

Ignore this guideline if your use of the observable sequence is not bound to a specific execution context. In that case do not use the `observeOn` operator.

Consider limiting buffers

RxJS comes with several operators and classes that create buffers over observable sequences, e.g. the `replay` operator. As these buffers work on any observable sequence, the size of these buffers will depend on the observable sequence it is operating on. If the buffer is unbounded, this can lead to memory pressure. Many buffering operators provide policies to limit the buffer, either in time or size. Providing this limit will address memory pressure issues.

Sample

```
var result = xs.replay(null, 10000, 1000 * 60 /* 1 hr */).refCount();
```

In this sample, the `replay` operator creates a buffer. We have limited that buffer to contain at most 10,000 messages and keep these messages around for a maximum of 1 hour.

When to ignore this guideline

When the amount of messages created by the observable sequence that populates the buffer is small or when the buffer size is limited.

Make side-effects explicit using the `do` / `tap` operator

As many Rx operators take functions as arguments, it is possible to pass any valid user code in these arguments. This code can change global state (e.g. change global variables, write to disk etc...).

The composition in Rx runs through each operator for each subscription (with the exception of the sharing operators, such as `publish`). This will make every side-effect occur for each subscription.

If this behavior is the desired behavior, it is best to make this explicit by putting the side-effecting code in a `do` / `tap` operator. There are overloads to this method which call the specified method only, for example `doOnNext` / `tapOnNext` , `doOnError` / `tapOnError` , `doOnCompleted` / `tapOnCompleted`

Sample

```
var result = xs.filter(x => x.failed).tap(x => log(x));
```

In this sample, messages are filtered for failure. The messages are logged before handing them out to the code subscribed to this observable sequence. The logging is a side-effect (e.g. placing the messages in the computer's event log) and is explicitly done via a call to the `do` / `tap` operator.

Assume messages can come through until unsubscribe has completed

As RxJS uses a push model, messages can be sent from different execution contexts. Messages can be in flight while calling unsubscribe. These messages can still come through while the call to unsubscribe is in progress. After control has returned, no more messages will arrive. The unsubscription process can still be in progress on a different context.

When to ignore this guideline

Once the `onCompleted` or `onError` method has been received, the RxJS grammar guarantees that the subscription can be considered to be finished.

Use the `publish` operator to share side-effects

As many observable sequences are cold (see cold vs. hot on Channel 9), each subscription will have a separate set of side-effects. Certain situations require that these side-effects occur only once. The `publish` operator provides a mechanism to share subscriptions by broadcasting a single subscription to multiple subscribers.

There are several overloads of the `publish` operator. The most convenient overloads are the ones that provide a function with a wrapped observable sequence argument that shares the side-effects.

Sample

```
var xs = Rx.Observable.create(observer => {
  console.log('Side effect');
  observer.onNext('hi!');
  observer.onCompleted();
});

xs.publish(sharedXs => {
  sharedXs.subscribe(console.log);
  sharedXs.subscribe(console.log);
  return sharedXs;
}).subscribe();
```

In this sample, xs is an observable sequence that has side-effects (writing to the console). Normally each separate subscription will trigger these side-effects. The `publish` operator uses a single subscription to xs for all subscribers to sharedXs.

When to ignore this guideline

Only use the `publish` operator to share side-effects when sharing is required. In most situations you can create separate subscriptions without any problems: either the subscriptions do not have side-effects or the side effects can execute multiple times without any issues.

Operator implementations

Implement new operators by composing existing operators

Many operations can be composed from existing operators. This will lead to smaller, easier to maintain code. The Rx team has put a lot of effort in dealing with all corner cases in the base operators. By reusing these operators you'll get all that work for free in your operator.

Sample

```
Rx.Observable.prototype.flatMap = selector => this.map(selector).mergeAll();
```

In this sample, the `flatMap` operator uses two existing operators: `map` and `mergeAll`. The `map` operator already deals with any issues around the selector function throwing an exception. The `mergeAll` operator already deals with concurrency issues of multiple observable sequences firing at the same time.

When to ignore this guideline

- No appropriate set of base operators is available to implement this operator.
- Performance analysis proves that the implementation using existing operators has performance issues. Such can be caused by `materialize`.

Implement custom operators using `Observable.create`

When it is not possible to follow guideline 5.1, use the Observable.Create(WithDisposable) method to create an observable sequence as it provides several protections make the observable sequence follow the RxJS contract.

- When the observable sequence has finished (either by firing `onError` or `onCompleted`), any subscription will automatically be unsubscribed.
- Any subscribed observer instance will only see a single OnError or OnCompleted message. No more messages are sent through. This ensures the Rx grammar of onNext* (onError|onCompleted)?

Sample

```
Rx.Observable.prototype.map = (selector, thisArg) => {
  var source = this;
  return Rx.Observable.create(observer => {
    var idx = 0;
    return source.subscribe(
      x => {
        var result;
        try {
          result = selector.call(thisArg, x, idx++, source);
        } catch (e) {
          observer.onError(e);
          return;
        }

        observer.onNext(result);
      },
      observer.onError.bind(observer),
      observer.onCompleted.bind(observer)
    );
  })
};
```

In this sample, `map` uses the `Observable.create` operator to return a new instance of the
Observable class. This ensures that no matter the implementation of the source observable
sequence, the output observable sequence follows the Rx contract . It also ensures that the
lifetime of subscriptions is a short as possible.

When to ignore this guideline

- The operator needs to return an observable sequence that doesn't follow the Rx
  contract. This should usually be avoided (except when writing tests to see how code
  behaves when the contract is broken)
- The object returned needs to implement more than the Observable class (e.g. Subject,
  or a custom class).

Protect calls to user code from within an operator

When user code is called from within an operator, this is potentially happening outside of the
execution context of the call to the operator (asynchronously). Any exception that happens
here will cause the program to terminate unexpectedly. Instead it should be fed through to
the subscribed observer instance so that the exception can be dealt with by the subscribers.

Common kinds of user code that should be protected:

- Selector functions passed in to the operator.

- Comparer functions passed into the operator.

Note: calls to `Scheduler` implementations are not considered for this guideline. The reason for this is that only a small set of issues would be caught as most schedulers deal with asynchronous calls. Instead, protect the arguments passed to schedulers inside each scheduler implementation.

Sample

```
Rx.Observable.prototype.map = (selector, thisArg) => {
  var source = this;
  return Rx.Observable.create(observer => {
    var idx = 0;
    return source.subscribe(
      x => {
        var result;
        try {
          result = selector.call(thisArg, x, idx++, source);
        } catch (e) {
          observer.onError(e);
          return;
        }

        observer.onNext(result);
      },
      observer.onError.bind(observer),
      observer.onCompleted.bind(observer)
    );
  })
};
```

This sample invokes a selector function which is user code. It catches any exception resulting from this call and transfers the exception to the subscribed observer instance through the `onError` call.

When to ignore this guideline

Ignore this guideline for calls to user code that are made before creating the observable sequence (outside of the `Observable.create` call). These calls are on the current execution context and are allowed to follow normal control flow.

Note: do not protect calls to `subscribe` , `dispose` , `onNext` , `onError` and `onCompleted` methods. These calls are on the edge of the monad. Calling the `onError` method from these places will lead to unexpected behavior.

`subscribe` implementations should not throw

As multiple observable sequences are composed, subscribe to a specific observable sequence might not happen at the time the user calls `subscribe` (e.g. Within the `concat` operator, the second observable sequence argument to `concat` will only be subscribed to once the first observable sequence has completed). Throwing an exception would bring down the program. Instead exceptions in subscribe should be tunneled to the `onError` method.

Sample

```
var CLOSED = 3;

function readWebSocket(socket) {
  return Rx.Observable.create(observer => {
    if (socket.readyState === CLOSED) {
      observer.onError(new Error('The websocket is no longer open.'));
      return;
    }
    // Rest of the implementation goes here
  });
}
```

In this sample, an error condition is detected in the subscribe method implementation. An error is raised by calling the `onError` method instead of throwing the exception. This allows for proper handling of the exception if `subscribe` is called outside of the execution context of the original call to Subscribe by the user.

When to ignore this guideline

When a catastrophic error occurs that should bring down the whole program anyway.

`onError` messages should have abort semantics

As normal control flow in JavaScript uses abort semantics for exceptions (the stack is unwound, current code path is interrupted), RxJS mimics this behavior. To ensure this behavior, no messages should be sent out by an operator once one of it sources has an error message or an exception is thrown within the operator.

Sample

```
Rx.Observable.prototype.minimumBuffer = bufferSize => {
  var source = this;
  return Rx.Observable.create(observer => {
    var data = [];

    return source.subscribe(
      value => {
        data = data.concat(value);
        if (data.length > bufferSize) {
          observer.onNext(data.slice(0));
          data = [];
        }
      },
      observer.onError.bind(observer),
      () => {
        if (data.length > 0) {
          observer.onNext(data.slice(0));
        }
        observer.onCompleted();
      }
    );
  });
};
```

In this sample, a buffering operator will abandon the observable sequence as soon as the subscription to source encounters an error. The current buffer is not sent to any subscribers, maintain abort semantics.

When to ignore this guideline

There are currently no known cases where to ignore this guideline.

Parameterize concurrency by providing a scheduler argument

As there are many different notions of concurrency, and no scenario fits all, it is best to parameterize the concurrency an operator introduces. The notion of parameterizing concurrency in RxJS is abstracted through the `Scheduler` class.

Sample

```
Rx.Observable.just = (value, scheduler) => {
  return Rx.Observable.create(observer => {
    return scheduler.schedule(() => {
      observer.onNext(value);
      observer.onCompleted();
    });
  });
};
```

In this sample, the `just` operator parameterizes the level of concurrency the operator has by providing a scheduler argument. It then uses this scheduler to schedule the firing of the `onNext` and `onCompleted` messages.

When to ignore this guideline

- The operator is not in control of creating the concurrency (e.g. in an operator that converts an event into an observable sequence, the source event is in control of firing the messages, not the operator).
- The operator is in control, but needs to use a specific scheduler for introducing concurrency.

Provide a default scheduler

In most cases there is a good default that can be chosen for an operator that has parameterized concurrency through guideline 5.6. This will make the code that uses this operator more succinct.

Note: Follow guideline 5.9 when choosing the default scheduler, using the immediate scheduler where possible, only choosing a scheduler with more concurrency when needed.

Sample

```
Rx.Observable.just = (value, scheduler) => {
  // Pick a default scheduler, in this case, immediately
  Rx.helpers.isScheduler(scheduler) || (scheduler = Rx.Scheduler.immediate);

  return Rx.Observable.create(observer => {
    return scheduler.schedule(() => {
      observer.onNext(value);
      observer.onCompleted();
    });
  });
};
```

In this sample, we provided a default scheduler if not provided by the caller.

When to ignore this guideline

Ignore this guideline when no good default can be chosen.

The scheduler should be the last argument to the operator

Adding the scheduler as the last argument is a must for all operators introducing concurrency. This is to ensure that the schedulers are optional, and a default one can be chosen. This also makes the programming experience much more predictable.

Sample

```
Rx.Observable.just = (value, scheduler) => {
  // Pick a default scheduler, in this case, immediately
  Rx.helpers.isScheduler(scheduler) || (scheduler = Rx.Scheduler.immediate);

  return Rx.Observable.create(observer => {
    return scheduler.schedule(() => {
      observer.onNext(value);
      observer.onCompleted();
    });
  });
};
```

In this sample the `return` operator has two parameters, and the scheduler parameter defaults to the immediate scheduler if not provided. As the scheduler argument is the last argument, adding or omitting the argument is clearly visible.

When to ignore this guideline

JavaScript supports rest arguments syntax. With this syntax, the rest arguments has to be the last argument. Make the scheduler the final to last argument in this case.

Avoid introducing concurrency

By adding concurrency, we change the timeliness of an observable sequence. Messages will be scheduled to arrive later. The time it takes to deliver a message is data itself, by adding concurrency we skew that data. This includes not using such mechanisms as `setTimeout`, `setImmediate`, `requestAnimationFrame`, `process.nextTick`, etc which should be avoided directly in your code, and instead be wrapped by a `Scheduler` class.

Sample 1

```javascript
Rx.Observable.prototype.map = (selector, thisArg) => {
  var source = this;
  return Rx.Observable.create(observer => {
    var idx = 0;
    return source.subscribe(
      x => {
        var result;
        try {
          result = selector.call(thisArg, x, idx++, source);
        } catch (e) {
          observer.onError(e);
          return;
        }

        observer.onNext(result);
      },
      observer.onError.bind(observer),
      observer.onCompleted.bind(observer)
    );
  })
};
```

In this sample, the select operator does not use a scheduler to send out the `onNext` message. Instead it uses the source observable sequence call to `onNext` to process the message, hence staying in the same time-window.

Sample 2

```javascript
Rx.Observable.just = (value, scheduler) => {
  // Pick a default scheduler, in this case, immediately
  Rx.helpers.isScheduler(scheduler) || (scheduler = Rx.Scheduler.immediate);

  return Rx.Observable.create(observer => {
    return scheduler.schedule(() => {
      observer.onNext(value);
      observer.onCompleted();
    });
  });
};
```

In this case, the default scheduler for the `just` operator is the immediate scheduler. This scheduler does not introduce concurrency.

When to ignore this guideline

Ignore this guideline in situations where introduction of concurrency is an essential part of what the operator does.

NOTE: When we use the Immediate scheduler or call the observer directly from within the call to `subscribe`, we make the `subscribe` call blocking. Any expensive computation in this situation would indicate a candidate for introducing concurrency.

Hand out all disposables instances created inside the operator to consumers

`Disposable` instances control lifetime of subscriptions as well as cancelation of scheduled actions. RxJS gives users an opportunity to unsubscribe from a subscription to the observable sequence using disposable instances.

After a subscription has ended, no more messages are allowed through. At this point, leaving any state alive inside the observable sequence is inefficient and can lead to unexpected semantics.

To aid composition of multiple disposable instances, RxJS provides a set of classes implementing `Disposable` such as:

Name	Description
CompositeDisposable	Composes and disposes a group of disposable instances together.
SerialDisposable	A place holder for changing instances of disposable instances. Once new disposable instance is placed, the old disposable instance is disposed.
SingleAssignmentDisposable	A place holder for a single instance of a disposable.
ScheduledDisposable	Uses a scheduler to dispose an underlying disposable instance.

Sample

```js
Observable.prototype.zip = () => {
  var parent = this,
      sources = slice.call(arguments),
      resultSelector = sources.pop();

  sources.unshift(parent);
  return new AnonymousObservable(observer => {
    var n = sources.length,
      queues = arrayInitialize(n, () => []),
      isDone = arrayInitialize(n, () => false);
```

```javascript
    function next(i) {
      var res, queuedValues;
      if (queues.every(x => x.length > 0)) {
        try {
          queuedValues = queues.map(x => x.shift());
          res = resultSelector.apply(parent, queuedValues);
        } catch (ex) {
          observer.onError(ex);
          return;
        }
        observer.onNext(res);
      } else if (isDone.filter((x, j) => j !== i).every(identity)) {
        observer.onCompleted();
      }
    };

    function done(i) {
      isDone[i] = true;
      if (isDone.every(x => x)) {
        observer.onCompleted();
      }
    }

    var subscriptions = new Array(n);
    for (var idx = 0; idx < n; idx++) {
      (i => {
        var source = sources[i], sad = new SingleAssignmentDisposable();
        Rx.helpers.isPromise(source) && (source = Rx.Observable.fromPromise(source));
        sad.setDisposable(source.subscribe(x => {
          queues[i].push(x);
          next(i);
        }, observer.onError.bind(observer), () => {
          done(i);
        }));
        subscriptions[i] = sad;
      })(idx);
    }

    return new CompositeDisposable(subscriptions);
  });
};
```

In this sample, the operator groups all disposable instances controlling the various subscriptions together and returns the group as the result of subscribing to the outer observable sequence. When a user of this operator subscribes to the resulting observable sequence, he/she will get back a disposable instance that controls subscription to all underlying observable sequences.

When to ignore this guideline

There are currently no known instances where this guideline should be ignored.

Operators should not block

RxJS is a library for composing asynchronous and event-based programs using observable collections.

By making an operator blocking we lose these asynchronous characteristics. We also potentially lose composability (e.g. by returning a value typed as `T` instead of `Observable<T>`). This is in contrast to the Array#extras such as `sum` , `reduce` , `some` and `every` which return a single value.

Sample

```
Rx.Observable.prototype.sum = () => this.reduce((acc, x) => acc + x, 0);
```

In this sample, the `sum` operator has a return type of `Observable<Number>` instead of `Number` . By doing this, the operator does not block. It also allows the result value to be used in further composition.

When to ignore this guideline

There are currently no known instances where this guideline should be ignored.

Avoid deep stacks caused by recursion in operators

As code inside Rx operators can be called from different execution context in many different scenarios, it is nearly impossible to establish how deep the stack is before the code is called. If the operator itself has a deep stack (e.g. because of recursion), the operator could trigger a stack overflow quicker than one might expect.

There are two recommended ways to avoid this issue:

- Use the recursive `scheduleRecursive` methods on the `Scheduler`
- Implement an infinite looping generator using the yield iterator pattern, convert it to an observable sequence using the `from` operator.

Sample 1

```
Rx.Observable.repeat = (value, scheduler) => {
  return Rx.Observable.create(observer => {
    return scheduler.scheduleRecursive(self => {
      observer.onNext(value);
      self();
    });
  });
};
```

In this sample, the recursive `scheduleRecursive` method is used to allow the scheduler to schedule the next iteration of the recursive function. Schedulers such as the current thread scheduler do not rely on stack semantics. Using such a scheduler with this pattern will avoid stack overflow issues.

Sample 2

```
Rx.Observable.repeat = value => {
  return Rx.Observable.from(
    function* () {
      while(true) { yield value; }
    }());
};
```

The yield iterator pattern ensures that the stack depth does not increase drastically. By returning an infinite generator with the `from` operator can build an infinite observable sequence.

When to ignore this guideline

There are currently no known instances where this guideline should be ignored.

Argument validation should occur outside `Observable.create`

As guideline 5.3 specifies that the `Observable.create` operator should not throw, any argument validation that potentially throws exceptions should be done outside the `Observable.create` operator.

Sample

```
Rx.Observable.prototype.map = (selector, thisArg) => {
  if (this == null) {
    throw new TypeError('Must be an instance of an Observable');
  }
  if (selector == null) {
    throw new TypeError('selector cannot be null/undefined');
  }
  var selectorFn = typeof selector !== 'function' ? () => selector : selector;
  var source = this;
  return Rx.Observable.create(observer => {
    var idx = 0;
    return source.subscribe(
      x => {
        var result;
        try {
          result = selectorFn.call(thisArg, x, idx++, source);
        } catch (e) {
          observer.onError(e);
          return;
        }

        observer.onNext(result);
      },
      observer.onError.bind(observer),
      observer.onCompleted.bind(observer)
    );
  };
};
```

In this sample, the arguments are checked for null values before the `Observable.create` operator is called.

When to ignore this guideline

Ignore this guideline if some aspect of the argument cannot be checked until the subscription is active.

Unsubscription should be idempotent

The observable `subscribe` method returns a `Disposable` instance that can be used to clean up the subscription. The `Disposable` instance doesn't give any information about what the state of the subscription is. As consumers do not know the state of the subscription, calling the `dispose` method multiple times should be allowed. Only the first call the side-effect of cleaning up the subscription should occur.

Sample

```
var subscription = xs.subscribe(console.log.bind(console));
subscription.dispose();
subscription.dispose();
```

In this sample, the subscription is disposed twice, the first time the subscription will be cleaned up and the second call will be a no-op.

Unsubscription should not throw

As the RxJS's composition makes that subscriptions are chained, so are unsubscriptions. Because of this, any operator can call an unsubscription at any time. Because of this, just throwing an exception will lead to the application crashing unexpectedly. As the observer instance is already unsubscribed, it cannot be used for receiving the exception either. Because of this, exceptions in unsubscriptions should be avoided.

When to ignore this guideline

There are currently no known cases where to ignore this guideline.

Custom `Observable` implementations should follow the RxJS contract

When it is not possible to follow guideline 5.1, the custom implementation of the `Observable` class should still follow the RxJS contract in order to get the right behavior from the RxJS operators.

When to ignore this guideline

Only ignore this guideline when writing observable sequences that need to break the contract on purpose (e.g. for testing).

Operator implementations should follow guidelines for RxJS usage

As Rx is a composable API, operator implementations often use other operators for their implementation (see paragraph 5.1). RxJS usage guidelines should be strongly considered when implementing these operators.

When to ignore this guideline

As described in the introduction, only follow a guideline if it makes sense in that specific situation.

Getting Started With RxJS

- What are the Reactive Extensions?
- Exploring Major Concepts in RxJS
- Creating and Querying Observable Sequences
- Subjects
- Scheduling and Concurrency
- Testing and Debugging
- Implementing Your Own Operators

What are the Reactive Extensions for JavaScript (RxJS)?

The Reactive Extensions for JavaScript (RxJS) is a library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators. Using RxJS, developers **represent** asynchronous data streams with Observables, **query** asynchronous data streams using LINQ operators, and **parameterize** the concurrency in the asynchronous data streams using Schedulers. Simply put, Rx = Observables + LINQ + Schedulers.

Whether you are authoring a web-based application or server-side applications with Node.js, you have to deal with asynchronous and event-based programming constantly. Web applications and Node.js applications have I/O operations and computationally expensive tasks that might take a long time to complete and potentially block the main thread. Furthermore, handling exceptions, cancellation, and synchronization is difficult and error-prone.

Using RxJS, you can represent multiple asynchronous data streams (that come from diverse sources, e.g., stock quote, tweets, computer events, web service requests, etc.), and subscribe to the event stream using the `Observer` object. The `Observable` object notifies the subscribed `Observer` object whenever an event occurs.

Because observable sequences are data streams, you can query them using standard query operators implemented by the Observable extension methods. Thus you can filter, project, aggregate, compose and perform time-based operations on multiple events easily by using these standard query operators. In addition, there are a number of other reactive stream specific operators that allow powerful queries to be written. Cancellation, exceptions, and synchronization are also handled gracefully by using the extension methods provided by Rx.

RxJS complements and interoperates smoothly with both synchronous data streams such as Arrays, Sets and Maps and single-value asynchronous computations such as Promises as the following diagram shows:

	Single return value	Mutiple return values
Pull/Synchronous/Interactive	Object	Iterables (Array | Set | Map | Object)
Push/Asynchronous/Reactive	Promise	Observable

Pushing vs. Pulling Data

In interactive programming, the application actively polls a data source for more information by retrieving data from a sequence that represents the source. Such behavior is represented by the iterator pattern of JavaScript Arrays, Objects, Sets, Maps, etc. In interactive programming, one must get the next item by either getting an item by an index in an Array, or through ES6 iterators.

The application is active in the data retrieval process: it decides about the pace of the retrieval by calling `next` at its own convenience. This enumeration pattern is synchronous, which means your application might be blocked while polling the data source. Such pulling pattern is similar to visiting your library and checking out a book. After you are done with the book, you pay another visit to check out another one.

On the other hand, in reactive programming, the application is offered more information by subscribing to a data stream (called observable sequence in RxJS), and any update is handed to it from the source. The application is passive in the data retrieval process: apart from subscribing to the observable source, it does not actively poll the source, but merely react to the data being pushed to it. When the event has completed, the source will send a notice to the subscriber. In this way, your application will not be blocked by waiting for the source to update. This is the push pattern employed by Reactive Extensions for JavaScript. This is similar to joining a book club in which you register your interest in a particular genre, and books that match your interest are automatically sent to you as they are published. You do not need to stand in a line to acquire something that you want. Employing a push pattern is especially helpful in heavy UI environment in which the UI thread cannot be blocked while the application is waiting for some events, which is essential in JavaScript environments which has its own set of asynchronous requirements. In summary, by using RxJS, you can make your application more responsive.

The push model implemented by Rx is represented by the observable pattern of `Observable` / `Observer`. The `Observable` will notify all the observers automatically of any state changes. To register an interest through a subscription, you use the `subscribe` method of `Observable`, which takes on an `Observer` and returns a `Disposable` object. This gives you the ability to track your subscription and be able to dispose the subscription. You can essentially treat the observable sequence (such as a sequence of mouseover events) as if it were a normal collection. RxJS�s built-in query implementation over observable sequences allows developers to compose complex event processing queries over push-based sequences such as events, callbacks, Promises, HTML5 Geolocation APIs, and much much more.. For more information on these two interfaces, see Exploring The Major Concepts in RxJS.

Exploring The Major Concepts in RxJS

This topic describes the major Reactive Extensions for JavaScript (Rx) objects used to represent observable sequences and subscribe to them.

The `Observable` / `Observer` objects are available in the core distribution of RxJS.

`Observable` / `Observer`

Rx exposes asynchronous and event-based data sources as push-based, observable sequences abstracted by the `Observable` object in the core distribution of RxJS. It represents a data source that can be observed, meaning that it can send data to anyone who is interested.

As described in What is RxJS, the other half of the push model is represented by the `Observer` object, which represents an observer who registers an interest through a subscription. Items are subsequently handed to the observer from the observable sequence to which it subscribes.

In order to receive notifications from an observable collection, you use the `subscribe` method of `Observable` to hand it an `Observer` object. In return for this observer, the `subscribe` method returns a `Disposable` object that acts as a handle for the subscription. This allows you to clean up the subscription after you are done. Calling `dispose` on this object detaches the observer from the source so that notifications are no longer delivered. As you can infer, in RxJS you do not need to explicitly unsubscribe from an event as in the common JavaScript event model.

Observers support three publication events, reflected by the object's methods. The `onNext` can be called zero or more times, when the observable data source has data available. For example, an observable data source used for mouse move events can send out an event object every time the mouse has moved. The other two methods are used to indicate completion or errors.

The following lists the `Observable` / `Observer` objects in addition to the `Disposable` object.

```javascript
/**
 * Defines a method to release allocated resources.
 */
function Disposable() { }

/**
 * Performs application-defined tasks associated with freeing, releasing, or resetting re
 */
Disposable.prototype.dispose =  () => { ... }

/**
 * Defines a provider for push-based notification.
 */
function Observable() { }

/**
 * Notifies the provider that an observer is to receive notifications.
 *
 * @param {Observer} observer The object that is to receive notifications.
 * @returns {Disposable} A reference to disposable that allows observers to stop receivin
 */
Observable.prototype.subscribe =  observer => { ... }

/**
 * Provides a mechanism for receiving push-based notifications.
 */
function Observer() { }

/**
 * Provides the observer with new data.
 *
 * @param {Any} value The current notification information.
 */
Observer.prototype.onNext = value => { ... };

/**
 * Notifies the observer that the provider has experienced an error condition.
 *
 * @param {Error} error An object that provides additional information about the error.
 */
Observer.prototype.onError = error => { ... };

/**
 * Notifies the observer that the provider has finished sending push-based notifications.
 */
Observer.prototype.onCompleted = () => { ... };
```

RxJS also provides `subscribe` capabilities so that you can avoid implementing the `Observer` object yourself. For each publication event (`onNext` , `onError` , `onCompleted`) of an observable sequence, you can specify a function that will be invoked, as shown in the

following example. If you do not specify an action for an event, the default behavior will occur.

```javascript
// Creates an observable sequence of 5 integers, starting from 1
var source = Rx.Observable.range(1, 5);

// Prints out each item
var subscription = source.subscribe(
    x => console.log('onNext: ' + x),
    e => console.log('onError: ' + e.message),
    () => console.log('onCompleted'));

// => onNext: 1
// => onNext: 2
// => onNext: 3
// => onNext: 4
// => onNext: 5
// => onCompleted
```

You can treat the observable sequence (such as a sequence of mouse-over events) as if it were a normal collection. Thus you can write queries over the collection to do things like filtering, grouping, composing, etc. To make observable sequences more useful, the RxJS libraries provide many factory operators so that you do not need to implement any of these on your own. This will be covered in the Querying Observable Sequences topic.

Caution:

You do not need to implement the Observable/Observer objects yourself. Rx provides internal implementations of these interfaces for you and exposes them through various extension methods provided by the Observable and Observer types. See the Creating and Querying Observable Sequences topic for more information.

See Also

Concepts

- Querying Observable Sequences

Other Resources

- Creating and Querying Observable Sequences

Creating and Querying Observable Sequences

- Creating and Subscribing to Simple Observable Sequences
- Bridging to Events
- Bridging to Callbacks
- Bridging to Promises
- Generators and Observable Sequences
- Querying Observable Sequences
- Error Handling With Observable Sequences
- Transducers with Observable Sequences
- Backpressure with Observable Sequences
- Operators by Category

Creating and Subscribing to Simple Observable Sequences

You do not need to implement the `Observable` class manually to create an observable sequence. Similarly, you do not need to implement `Observer` either to subscribe to a sequence. By installing the Reactive Extension libraries, you can take advantage of the `Observable` type which provides many operators for you to create a simple sequence with zero, one or more elements. In addition, RxJS provides an overloaded `subscribe` method which allows you to pass in `onNext`, `onError` and `onCompleted` function handlers.

Creating a sequence from scratch

Before getting into many operators, let's look at how to create an `Observable` from scratch using the `Rx.Observable.create` method.

First, we need to ensure we reference the core `rx.js` file.

```
<script src="rx.js"></script>
```

Or if we're using Node.js, we can reference it as such:

```
var Rx = require('rx');
```

In this example, we will simply yield a single value of 42 and then mark it as completed. The return value is completely optional if no cleanup is required.

```
var source = Rx.Observable.create(observer => {
  // Yield a single value and complete
  observer.onNext(42);
  observer.onCompleted();

  // Any cleanup logic might go here
  return () => console.log('disposed')
});

var subscription = source.subscribe(
  x => console.log('onNext: %s', x),
  e => console.log('onError: %s', e),
  () => console.log('onCompleted'));

// => onNext: 42
// => onCompleted

subscription.dispose();
// => disposed
```

For most operations, this is completely overkill, but shows the very basics of how most RxJS operators work.

Creating and subscribing to a simple sequence

The following sample uses the `range` operator of the `Observable` type to create a simple observable collection of numbers. The observer subscribes to this collection using the Subscribe method of the Observable class, and provides actions that are delegates which handle `onNext`, `onError` and `onCompleted`. In our example, it creates a sequence of integers that starts with x and produces y sequential numbers afterwards.

As soon as the subscription happens, the values are sent to the observer. The `onNext` function then prints out the values.

```
// Creates an observable sequence of 5 integers, starting from 1
var source = Rx.Observable.range(1, 5);

// Prints out each item
var subscription = source.subscribe(
  x => console.log('onNext: %s', x),
  e => console.log('onError: %s', e),
  () => console.log('onCompleted'));

// => onNext: 1
// => onNext: 2
// => onNext: 3
// => onNext: 4
// => onNext: 5
// => onCompleted
```

When an observer subscribes to an observable sequence, the `subscribe` method may be using asynchronous behavior behind the scenes depending on the operator. Therefore, the `subscribe` call is asynchronous in that the caller is not blocked until the observation of the sequence completes. This will be covered in more details in the Using Schedulers topic.

Notice that the `subscribe` method returns a `Disposable`, so that you can unsubscribe to a sequence and dispose of it easily. When you invoke the `dispose` method on the observable sequence, the observer will stop listening to the observable for data. Normally, you do not need to explicitly call `dispose` unless you need to unsubscribe early, or when the source observable sequence has a longer life span than the observer. Subscriptions in Rx are designed for fire-and-forget scenarios without the usage of a finalizer. Note that the default behavior of the Observable operators is to dispose of the subscription as soon as possible (i.e, when an `onCompleted` or `onError` messages is published). For example, the code will subscribe x to both sequences a and b. If a throws an error, x will immediately be unsubscribed from b.

```
var x = Rx.Observable.zip(a, b, (a1, b1) => a1 + b1).subscribe();
```

You can also tweak the code sample to use the Create operator of the Observer type, which creates and returns an observer from specified OnNext, OnError, and OnCompleted action delegates. You can then pass this observer to the Subscribe method of the Observable type. The following sample shows how to do this.

```
// Creates an observable sequence of 5 integers, starting from 1
var source = Rx.Observable.range(1, 5);

// Create observer
var observer = Rx.Observer.create(
  x => console.log('onNext: %s', x),
  e => console.log('onError: %s', e),
  () => console.log('onCompleted'));

// Prints out each item
var subscription = source.subscribe(observer);

// => onNext: 1
// => onNext: 2
// => onNext: 3
// => onNext: 4
// => onNext: 5
// => onCompleted
```

In addition to creating an observable sequence from scratch, you can convert existing Arrays, events, callbacks and promises into observable sequences. The other topics in this section will show you how to do this.

Notice that this topic only shows you a few operators that can create an observable sequence from scratch. To learn more about other LINQ operators, see Querying Observable Sequences using LINQ Operators.

Using a timer

The following sample uses the `timer` operator to create a sequence. The sequence will push out the first value after 5 second has elapsed, then it will push out subsequent values every 1 second. For illustration purpose, we chain the `timestamp` operator to the query so that each value pushed out will be appended by the time when it is published. By doing so, when we subscribe to this source sequence, we can receive both its value and timestamp.

First, we need to ensure we reference the proper files if in the browser. Note that the RxJS NPM Package already includes all operators by default.

```html
<script src="rx.js"></script>
<script src="rx.time.js"></script>
```

Now on to our example

```
console.log('Current time: ' + Date.now());

var source = Rx.Observable.timer(
  5000, /* 5 seconds */
  1000 /* 1 second */)
    .timestamp();

var subscription = source.subscribe(
  x => console.log(x.value + ': ' + x.timestamp));

/* Output may be similar to this */
// Current time: 1382560697820
// 0: 1382560702820
// 1: 1382560703820
// 2: 1382560704820
```

By using the `timestamp` operator, we have verified that the first item is indeed pushed out 5 seconds after the sequence has started, and each item is published 1 second later.

Converting Arrays and Iterables to an Observable Sequence

Using the `Rx.Observable.from` operator, you can convert an array to observable sequence.

```
var array = [1,2,3,4,5];

// Converts an array to an observable sequence
var source = Rx.Observable.from(array);

// Prints out each item
var subscription = source.subscribe(
  x => console.log('onNext: %s', x),
  e => console.log('onError: %s', e),
  () => console.log('onCompleted'));

// => onNext: 1
// => onNext: 2
// => onNext: 3
// => onNext: 4
// => onNext: 5
// => onCompleted
```

You can also convert array-like objects such as objects with a length property and indexed with numbers. In this case, we'll simply have an object with a length of 5.

```
var arrayLike = { length: 5 };

// Converts an array to an observable sequence
var source = Rx.Observable.from(arrayLike, (v, k) => k);

// Prints out each item
var subscription = source.subscribe(
  x => console.log('onNext: %s', x),
  e => console.log('onError: %s', e),
  () => console.log('onCompleted'));

// => onNext: 1
// => onNext: 2
// => onNext: 3
// => onNext: 4
// => onNext: 5
// => onCompleted
```

In addition, we can also use ES6 Iterable objects such as `Map` and `Set` using `from` to an observable sequence. In this example, we can take a `Set` and convert it to an observable sequence.

```
var set = new Set([1,2,3,4,5]);

// Converts a Set to an observable sequence
var source = Rx.Observable.from(set);

// Prints out each item
var subscription = source.subscribe(
  x => console.log('onNext: %s', x),
  e => console.log('onError: %s', e),
  () => console.log('onCompleted'));

// => onNext: 1
// => onNext: 2
// => onNext: 3
// => onNext: 4
// => onNext: 5
// => onCompleted
```

We can also do a `Map` as well by applying the same technique.

```
var set = new Map([['key1', 1], ['key2', 2]]);

// Converts a Set to an observable sequence
var source = Rx.Observable.from(set);

// Prints out each item
var subscription = source.subscribe(
  x => console.log('onNext: %s', x),
  e => console.log('onError: %s', e),
  () => console.log('onCompleted'));

// => onNext: key1, 1
// => onNext: key2, 2
// => onCompleted
```

The `from` method can also support ES6 generators which may already be in your browser, or coming to a browser near you. This allows us to do such things as Fibonacci sequences and so forth and convert them to an observable sequence.

```
function* fibonacci () {
  var fn1 = 1;
  var fn2 = 1;
  while (1){
    var current = fn2;
    fn2 = fn1;
    fn1 = fn1 + current;
    yield current;
  }
}

// Converts a generator to an observable sequence
var source = Rx.Observable.from(fibonacci()).take(5);

// Prints out each item
var subscription = source.subscribe(
  x => console.log('onNext: %s', x),
  e => console.log('onError: %s', e),
  () => console.log('onCompleted'));

// => onNext: 1
// => onNext: 1
// => onNext: 2
// => onNext: 3
// => onNext: 5
// => onCompleted
```

Cold vs. Hot Observables

Cold observables start running upon subscription, i.e., the observable sequence only starts pushing values to the observers when Subscribe is called. Values are also not shared among subscribers. This is different from hot observables such as mouse move events or stock tickers which are already producing values even before a subscription is active. When an observer subscribes to a hot observable sequence, it will get the current value in the stream. The hot observable sequence is shared among all subscribers, and each subscriber is pushed the next value in the sequence. For example, even if no one has subscribed to a particular stock ticker, the ticker will continue to update its value based on market movement. When a subscriber registers interest in this ticker, it will automatically get the latest tick.

The following example demonstrates a cold observable sequence. In this example, we use the Interval operator to create a simple observable sequence of numbers pumped out at specific intervals, in this case, every 1 second.

Two observers then subscribe to this sequence and print out its values. You will notice that the sequence is reset for each subscriber, in which the second subscription will restart the sequence from the first value.

First, we need to ensure we reference the proper files if in the browser. Note that the RxJS NPM Package already includes all operators by default.

```html
<script src="rx.lite.js"></script>
```

And now to the example.

```
var source = Rx.Observable.interval(1000);

var subscription1 = source.subscribe(
  x => console.log('Observer 1: onNext: ' + x),
  e => console.log('Observer 1: onError: ' + e.message),
  () => console.log('Observer 1: onCompleted'));

var subscription2 = source.subscribe(
  x => console.log('Observer 2: onNext: ' + x),
  e => console.log('Observer 2: onError: ' + e.message),
  () => console.log('Observer 2: onCompleted'));

setTimeout(() => {
  subscription1.dispose();
  subscription2.dispose();
}, 5000);

// => Observer 1: onNext: 0
// => Observer 2: onNext: 0
// => Observer 1: onNext: 1
// => Observer 2: onNext: 1
// => Observer 1: onNext: 2
// => Observer 2: onNext: 2
// => Observer 1: onNext: 3
// => Observer 2: onNext: 3
```

In the following example, we convert the previous cold observable sequence source to a hot one using the `publish` operator, which returns a `ConnectableObservable` instance we name `hot`. The `publish` operator provides a mechanism to share subscriptions by broadcasting a single subscription to multiple subscribers. The `hot` variable acts as a proxy by subscribing to `source` and, as it receives values from `source`, pushing them to its own subscribers. To establish a subscription to the backing source and start receiving values, we use the `ConnectableObservable.prototype.connect` method. Since `ConnectableObservable` inherits `Observable`, we can use `subscribe` to subscribe to this hot sequence even before it starts running. Notice that in the example, the hot sequence has not been started when `subscription1` subscribes to it. Therefore, no value is pushed to the subscriber. After calling Connect, values are then pushed to `subscription1`. After a delay of 3 seconds, `subscription2` subscribes to `hot` and starts receiving the values immediately from the current position (3 in this case) until the end. The output looks like this:

```
// => Current time: 1382562433256
// => Current Time after 1st subscription: 1382562433260
// => Current Time after connect: 1382562436261
// => Observer 1: onNext: 0
// => Observer 1: onNext: 1
// => Current Time after 2nd subscription: 1382562439262
// => Observer 1: onNext: 2
// => Observer 2: onNext: 2
// => Observer 1: onNext: 3
// => Observer 2: onNext: 3
// => Observer 1: onNext: 4
// => Observer 2: onNext: 4
```

First, we need to ensure we reference the proper files if in the browser. Note that the RxJS NPM Package already includes all operators by default.

```html
<script src="rx.lite.js"></script>
```

Now onto the example!

```
console.log('Current time: ' + Date.now());

// Creates a sequence
var source = Rx.Observable.interval(1000);

// Convert the sequence into a hot sequence
var hot = source.publish();

// No value is pushed to 1st subscription at this point
var subscription1 = hot.subscribe(
  x => console.log('Observer 1: onNext: %s', x),
  e => console.log('Observer 1: onError: %s', e),
  () => console.log('Observer 1: onCompleted'));

console.log('Current Time after 1st subscription: ' + Date.now());

// Idle for 3 seconds
setTimeout(() => {

  // Hot is connected to source and starts pushing value to subscribers
  hot.connect();

  console.log('Current Time after connect: ' + Date.now());

  // Idle for another 3 seconds
  setTimeout(() => {

    console.log('Current Time after 2nd subscription: ' + Date.now());

    var subscription2 = hot.subscribe(
      x => console.log('Observer 2: onNext: %s', x),
      e => console.log('Observer 2: onError: %s', e),
      () => console.log('Observer 2: onCompleted'));

  }, 3000);
}, 3000);

// => Current Time after connect: 1431197578426
// => Observer 1: onNext: 0
// => Observer 1: onNext: 1
// => Observer 1: onNext: 2
// => Current Time after 2nd subscription: 1431197581434
// => Observer 1: onNext: 3
// => Observer 2: onNext: 3
// => Observer 1: onNext: 4
// => Observer 2: onNext: 4
// => Observer 1: onNext: 5
// => Observer 2: onNext: 5
// => ...
```

Analogies

It helps to think of cold and hot Observables as movies or performances that one can watch ("subscribe").

- Cold Observables: movies.
- Hot Observables: live performances.
- Hot Observables replayed: live performances recorded on video.

Whenever you watch a movie, your run of the movie is independent of anyone else's run, even though all movie watchers see the same effects. On the other hand, a live performance is shared to multiple viewers. If you arrive late to a live performance, you will simply miss some of it. However, if it was recorded on video (in RxJS this would happen with a BehaviorSubject or a ReplaySubject), you can watch a "movie" of the live performance. A `.publish().refCount()` live performance is one where the artists quit playing when no one is watching, and start playing again when there is at least one person in the audience.

Bridging to Events

RxJS provides factory methods for you to bridge with existing asynchronous sources in the DOM or Node.js so that you can employ the rich composing, filtering and resource management features provided by RxJS on any kind of data streams. This topic examines the `fromEvent` and `fromEventPattern` operator that allows "importing" a DOM or custom event into RxJS as an observable sequence. Every time an event is raised, an `onNext` message will be delivered to the observable sequence. You can then manipulate event data just like any other observable sequences.

RxJS does not aim at replacing existing asynchronous programming models such as promises or callbacks. However, when you attempt to compose events, RxJS's factory methods will provide you the convenience that cannot be found in the current programming model. This is especially true for resource maintenance (e.g., when to unsubscribe) and filtering (e.g., choosing what kind of data to receive). In this topic and the ones that follow, you can examine how these RxJS features can assist you in asynchronous programming.

Natively, RxJS supports a number of libraries and hooks into them such as jQuery, Zepto.js, AngularJS, Ember.js and Backbone.js for using their event system. This behavior, however, can be overridden to only use native bindings only. By default, RxJS also has hooks for Node.js `EventEmitter` events natively supported.

Converting a DOM event to a RxJS Observable Sequence

The following sample creates a simple DOM event handler for the mouse move event, and prints out the mouse's location on the page.

```
var result = document.getElementById('result');

document.addEventListener('mousemove', e => result.innerHTML = e.clientX + ', ' + e.clien
```

To import an event into RxJS, you can use the `fromEvent` operator, and provide the event arguments that will be raised by the event being bridged. It then converts the given event into an observable sequence.

In the following example, we convert the mousemove event stream of the DOM into an observable sequence. Every time a mouse-move event is fired, the subscriber will receive an `onNext` notification. We can then examine the event arguments value of such notification and get the location of the mouse-move.

```javascript
var result = document.getElementById('result');

var source = Rx.Observable.fromEvent(document, 'mousemove');

var subscription = source.subscribe(e => result.innerHTML = e.clientX + ', ' + e.clientY)
```

Notice that in this sample, move becomes an observable sequence in which we can manipulate further. The Querying Observable Sequences topic will show you how you can project this sequence into a collection of Points type and filter its content, so that your application will only receive values that satisfy a certain criteria.

Cleaning up of the event handler is taken care of by the `Disposable` object returned by the `subscribe` method. Calling `dispose` will release all resources being used by the sequence including the underlying event handler. This essentially takes care of unsubscribing to an event on your behalf.

The `fromEvent` method also supports adding event handlers to multiple items, for example a DOM NodeList. This example will add the 'click' to each element in the list.

```javascript
var result = document.getElementById('result');
var sources = document.querySelectorAll('div');

var source = Rx.Observable.fromEvent(sources, 'click');

var subscription = source.subscribe(e => result.innerHTML = e.clientX + ', ' + e.clientY)
```

In addition, `fromEvent` also supports libraries such as jQuery, Zepto.js, AngularJS, Ember.js and Backbone.js:

```javascript
var $result = $('#result');
var $sources = $('div');

var source = Rx.Observable.fromEvent($sources, 'click');

var subscription = source.subscribe(e => $result.html(e.clientX + ', ' + e.clientY));
```

If this behavior is not desired, you can override it by setting the `Rx.config.useNativeEvents` to `true` which will disregard any library for which we support events.

```
// Use only native events even if jQuery
Rx.config.useNativeEvents = true;

// Native events only
var result = document.getElementById('result');

var source = Rx.Observable.fromEvent(document, 'mousemove');

var subscription = source.subscribe(e => result.innerHTML = e.clientX + ', ' + e.clientY)
```

In addition, you could easily add many shortcuts into the event system for events such as `mousemove` , and even extending to Pointer and Touch Events.

```
Rx.dom = {};

var events = "blur focus focusin focusout load resize scroll unload click dblclick " +
  "mousedown mouseup mousemove mouseover mouseout mouseenter mouseleave " +
  "change select submit keydown keypress keyup error contextmenu";

if (root.PointerEvent) {
  events += " pointerdown pointerup pointermove pointerover pointerout pointerenter point
}

if (root.TouchEvent) {
  events += " touchstart touchend touchmove touchcancel";
}

events.split(' ').forEach(e => {
  Rx.dom[e] = (element, selector) => Rx.Observable.fromEvent(element, e, selector)
});
```

Now we can rewrite a simple mouse drag as the following:

```
var draggable = document.getElementById('draggable');

var mousedrag = Rx.dom.mousedown(draggable).flatMap(md => {
  md.preventDefault();

  var start = getLocation(md);

  return Rx.dom.mousemove(document)
    .map(mm => getDelta(start, mm))
    .takeUntil(Rx.dom.mouseup(draggable));
});
```

Note this is already available in the [RxJS-DOM](#) project, but is small enough for you to implement yourself.

Converting a Node.js event to a RxJS Observable Sequence

Node.js is also supported such as an `EventEmitter` :

```javascript
var Rx = require('rx'),
  EventEmitter = require('events').EventEmitter;

var eventEmitter = new EventEmitter();

var source = Rx.Observable.fromEvent(eventEmitter, 'data')

var subscription = source.subscribe(data => console.log('data: ' + data));

eventEmitter.emit('data', 'foo');
// => data: foo
```

Bridging to Custom Events with FromEventPattern

There may be instances dealing with libraries which have different ways of subscribing and unsubscribing from events. The `fromEventPattern` method was created exactly for this purpose to allow you to bridge to each of these custom event emitters.

For example, you might want to bridge to using jQuery `on` method. We can convert the following code which alerts based upon the click of a table row.

```javascript
$( "#dataTable tbody" ).on('click', 'tr', () => alert($( this ).text()));
```

The converted code looks like this while using the `fromEventPattern` method. Each function passes in the handler function which allows you to call the `on` and `off` methods to properly handle disposal of events.

```
var $tbody = $('#dataTable tbody');

var source = Rx.Observable.fromEventPattern(
  function addHandler (h) { $tbody.on('click', 'tr', h); },
  function delHandler (h) { $tbody.off('click', 'tr', h); });

var subscription = source.subscribe(e => alert($( this ).text()));
```

In addition to this normal support, we also support if the `addHandler` returns an object, it can be passed to the `removeHandler` to properly unsubscribe. In this example, we'll use the Dojo Toolkit and the `on` module.

```
require(['dojo/on', 'dojo/dom', 'rx', 'rx.async', 'rx.binding'], (on, dom, rx) => {

    var input = dom.byId('input');

    var source = Rx.Observable.fromEventPattern(
        function addHandler (h) {
            return on(input, 'click', h);
        },
        function delHandler (_, signal) {
            signal.remove();
        }
    );

    var subscription = source.subscribe(
        x => console.log('Next: Clicked!'),
        err => console.log('Error: ' + err),
        () => console.log('Completed'));

    on.emit(input, 'click');
    // => Next: Clicked!
});
```

See Also

Concepts

- Querying Observable Sequences

Bridging to Callbacks

Besides events, other asynchronous data sources exist in the the web and server-side world. One of them is the the simple callback pattern which is frequently used in Node.js. In this design pattern, the arguments are passed to the function, and then a callback is usually the last parameter, which when executed, passes control to the inner scope with the data. Node.js has a standard way of doing callbacks in which the the callback is called with the `Error` object first if there is an error, else null, and then the additional parameters from the callback.

Converting Callbacks to Observable Sequences

Many asynchronous methods in Node.js and the many JavaScript APIs are written in such a way that it has a callback as the last parameter. These standard callbacks are executed with the data passed to it once it is available. We can use the `Rx.Observable.fromCallback` to wrap these kinds of callbacks. Note that this does not cover the Node.js style of callbacks where the `Error` parameter is first. For that operation, we provide the `Rx.Observable.fromNodeCallback` which we will cover below.

In the following example, we will convert the Node.js `fs.exists` function. This function takes a path and returns a `true` or `false` value whether the file exists, in this case we will check if 'file.txt' exists. The arguments returned when wrapped in `Rx.Observable.fromCallback` will return an array containing the arguments passed to the callback.

```
var Rx = require('rx'),
    fs = require('fs');

// Wrap the exists method
var exists = Rx.Observable.fromCallback(fs.exists);

var source = exists('file.txt');

// Get the first argument only which is true/false
var subscription = source.subscribe(
    x => console.log('onNext: %s', x),
    e => console.log('onError: %s', e),
    () => console.log('onCompleted'));

// => onNext: true
// => onCompleted
```

Converting Node.js Style Callbacks to Observable Sequences

Node.js has adopted a convention in many of the callbacks where an error may occur, such as File I/O, Network requests, etc. RxJS supports this through the `Rx.Observable.fromNodeCallback` method in which the error, if present, is captured and the `onError` notification is sent. Otherwise, the `onNext` is sent with the rest of the callback arguments, followed by an `onCompleted` notification.

In the following example, we will convert the Node.js `fs.rename` function to an Observable sequence.

```
var fs = require('fs'),
    Rx = require('rx');

// Wrap fs.rename
var rename = Rx.Observable.fromNodeCallback(fs.rename);

// Rename file which returns no parameters except an error
var source = rename('file1.txt', 'file2.txt');

var subscription = source.subscribe(
    x => console.log('onNext: success!'),
    e => console.log('onError: %s', e),
    () => console.log('onCompleted'));

// => onNext: success!
// => onCompleted
```

Converting Observable sequences to Callbacks

We can easily go in another direction and convert an observable sequence to a callback. This of course requires the observable sequence to yield only one value for this to make sense. Let's convert using the `timer` method to wait for a certain amount of time. The implementation of `toCallback` could look like the following. Note that it is not included in RxJS but you can easily add it if needed.

```
Rx.Observable.prototype.toCallback = cb => {
  var source = this;
  return () => {
    var val, hasVal = false;
    source.subscribe(
      x=> { hasVal = true; val = x; },
      e => throw e, // Default error handling
      () => hasVal && cb(val)}
    );
  };
};
```

Then we could execute our command simply like the following:

```
function cb (x) { console.log('hi!'); }

setTimeout(
  Rx.Observable.timer(5000)
    .toCallback(cb)
  , 500);
```

Converting Observable sequences to Node.js Style Callbacks

The same could also apply to Node.js style callbacks should you desire that behavior. Once again the same restrictions apply with regards to having a single value and an end much like above. The implementation of `toNodeCallback` could look like the following. Note that it is not included in RxJS but you can easily add it if needed.

```
Rx.Observable.prototype.toNodeCallback = cb => {
  var source = this;
  return () => {
    var val, hasVal = false;
    source.subscribe(
      x => { hasVal = true; val = x; },
      e => cb(e),
      () => hasVal && cb(null, val)}
    );
  };
};
```

We could then take this and for example if we had an observable sequence which gets a value from a REST call and then convert it to Node.js style.

```
getData().toNodeCallback((err, data) => {
    if (err) { throw err; }
    // Do something with the data
});
```

Bridging to Promises

Promises are a defacto standard within JavaScript community and is part of the ECMAScript Standard. A promise represents the eventual result of an asynchronous operation. The primary way of interacting with a promise is through its then method, which registers callbacks to receive either a promise's eventual value or the reason why the promise cannot be fulfilled. You can create them very easily where the constructor has two functions, `resolve` and `reject` which resolves the value or rejects it for a given reason. RxJS is fully committed to standards and has native support for Promises for any number of methods where they can be used interchangeably with Observable sequences.

The advantage that you get when you intermix Promises with Observable sequences is that unlike the ES6 Promise standard, you get cancellation semantics which means you can disregard values if you no longer are interested. One of the biggest problems around Promises right now are around cancellation, as to cancel the operation, such as an XHR is not easily done with the existing standard, nor is it to only get the last value to ensure no out of order requests. With Observable sequences, you get that behavior for free in a multicast behavior, instead of the unicast Promise behavior.

The following list of operators natively support Promises:

- `Rx.Observable.amb` | `Rx.Observable.prototype.amb`
- `Rx.Observable.case`
- `Rx.Observable.catch` | `Rx.Observable.prototype.catch`
- `Rx.Observable.combineLatest` | `Rx.Observable.prototype.combineLatest`
- `Rx.Observable.concat` | `Rx.Observable.prototype.concat`
- `Rx.Observable.prototype.concatMap`
- `Rx.Observable.prototype.concatMapObserver`
- `Rx.Observable.defer`
- `Rx.Observable.prototype.flatMap`
- `Rx.Observable.prototype.flatMapLatest`
- `Rx.Observable.forkJoin` | `Rx.Observable.prototype.forkJoin`
- `Rx.Observable.if`
- `Rx.Observable.merge`
- `Rx.Observable.prototype.mergeAll`
- `Rx.Observable.onErrorResumeNext` | `Rx.Observable.prototype.onErrorResumeNext`
- `Rx.Observable.prototype.selectMany`
- `Rx.Observable.prototype.selectSwitch`
- `Rx.Observable.prototype.sequenceEqual`

- `Rx.Observable.prototype.skipUntil`
- `Rx.Observable.startAsync`
- `Rx.Observable.prototype.switch`
- `Rx.Observable.prototype.takeUntil`
- `Rx.Observable.prototype.debounceWithSelector`
- `Rx.Observable.prototype.timeoutWithSelector`
- `Rx.Observable.while`
- `Rx.Observable.prototype.window`
- `Rx.Observable.withLatestFrom`
- `Rx.Observable.zip` | `Rx.Observable.prototype.zip`

Because of this, we can now do a number of very interesting things such as combining Promises and Observable sequences.

```
var source = Rx.Observable.range(0, 3)
  .flatMap(x => Promise.resolve(x * x));

var subscription = source.subscribe(
  x => console.log('onNext: %s', x),
  e => console.log('onError: %s', e),
  () => console.log('onCompleted'));

// => onNext: 0
// => onNext: 1
// => onNext: 4
// => onCompleted
```

This is just scatching the surface of what Promises and RxJS can do together so that we have first class single values and first class multiple values working together.

Converting Promises to Observable Sequences

It's quite simple to convert a Promise object which conforms to the ES6 Standard Promise where the behavior is uniform across implementations. To support this, we provide the `Rx.Observable.fromPromise` method which calls the `then` method of the promise to handle both success and error cases.

In the following example, we create promise objects using RSVP library.

```
// Create a promise which resolves 42
var promise1 = new RSVP.Promise((resolve, reject) => resolve(42));

var source1 = Rx.Observable.fromPromise(promise1);

var subscription1 = source1.subscribe(
  x => console.log('onNext: %s', x),
  e => console.log('onError: %s', e),
  () => console.log('onCompleted'));

// => onNext: 42
// => onCompleted

// Create a promise which rejects with an error
var promise2 = new RSVP.Promise((resolve, reject) => reject(new Error('reason')));

var source2 = Rx.Observable.fromPromise(promise2);

var subscription2 = source2.subscribe(
  x => console.log('onNext: %s', x),
  e => console.log('onError: %s', e),
  () => console.log('onCompleted'));

// => onError: reject
```

Notice that in this sample, these promises becomes an observable sequences in which we can manipulate further. The Querying Observable Sequences topic will show you how you can project this sequence into another, filter its content, so that your application will only receive values that satisfy a certain criteria.

Converting Observable Sequences to Promises

Just as you can convert a Promise to an Observable sequence, you can also convert an Observable sequence to a Promise. This either requires native support for Promises, or a Promise library you can add yourself, such as Q, RSVP, when.js among others. These libraries must conform to the ES6 standard for construction where it provides two functions to resolve or reject the promise.

```
var p = new Promise((resolve, reject) => resolve(42));
```

We can use the `toPromise` method which allows you to convert an Observable sequence to a Promise. This method accepts a Promise constructor, and if not provided, will default to a default implementation. In this first example, we will use RSVP to construct our Promise objects.

```
// Return a single value
var source1 = Rx.Observable.just(1).toPromise(RSVP.Promise);

source1.then(
  value => console.log('Resolved value: %s', value),
  reason => console.log('Rejected reason: %s', reason));

// => Resolved value: 1

// Reject the Promise
var source2 = Rx.Observable.throwError(new Error('reason')).toPromise(RSVP.Promise);

source2.then(
  value => console.log('Resolved value: %s', value),
  reason => console.log('Rejected reason: %s', reason));

// => Rejected reason: Error: reason
```

If an implementation is not given with the `toPromise` method, it will fall back to the Promise implementation specified in the `Rx.config.Promise` field. By default this will be set to the runtime's ES6 Promise implementation, but can easily be overridden by specifying the configuration information.

```
Rx.config.Promise = RSVP.Promise;

var source1 = Rx.Observable.just(1).toPromise();

source1.then(
  value => console.log('Resolved value: %s', value),
  reason => console.log('Rejected reason: %s', reason));

// => Resolved value: 1
```

If you are in a pure ES6 environment, this should just work without any settings on your part as it will use the runtime's ES6 Promise implementation.

```
var source1 = Rx.Observable.just(1).toPromise();

source1.then(
  value => console.log('Resolved value: %s', value),
  reason => console.log('Rejected reason: %s', reason));

// => Resolved value: 1
```

Concepts

- Querying Observable Sequences

Generators and Observable Sequences

One of the more exciting features of ES6 is a new function type called generators. They have been in Firefox for years, although they have now been finally standardized in ES6, and will be shipping in a browser or runtime near you. How generators differ from normal functions is that a normal function such as the following will run to completion, regardless of whether it is asynchronous or not.

```
function printNumberOfTimes(msg, n) {
  for (var i = 0; i < n; i++) {
    console.log(msg);
  }
}

printNumberOfTime('Hello world', 1);
// => Hello world

// Asynchronous
setTimeout(() => console.log('Hello from setTimeout after one second'), 1000);
// => Hello from setTimeout after one second
```

Instead of running to completion, generators allow us to interrupt the flow of the function by introducing the `yield` keyword which pauses the function. The function cannot resume on its own without the external consumer saying that they need the next value.

To create a generator function, you must use the `function*` syntax which then becomes a generator. In this particular example, we will yield a single value, the meaning of life.

```
function* theMeaningOfLife() {
  yield 42;
}
```

To get the value out, we need to invoke the function, and then call `next` to get the next value. The return value from the `next` call will have a flag as to whether it is done, as well as any value that is yielded. Note that the function doesn't do anything until we start to call `next`.

```
var it = theMeaningOfLife();

it.next();
// => { done: false, value: 42 }

it.next();
// => { done: true, value: undefined }
```

We can also use some ES6 shorthand for getting values from a generator such as the
`for..of` .

```
for (var v of theMeaningOfLife()) {
  console.log(v);
}
// => 42
```

This of course is only scratching the surface of what generators are capable of doing as
we're more focused on the simple nature of yielding values.

Since RxJS believes heavily in standards, we also look for ways to incorporate new
language features as they become standardized so that you can take advantage of them,
combined with the power of RxJS.

Async/Await Style and RxJS

One common complaint of JavaScript is the callback nature to asynchronous behavior.
Luckily, this can be solved quite easily with a library approach. To that end, we introduce
`Rx.Observable.spawn` which allows you to write straight forward code manner and can yield
not only Observable sequences, but also Promises, Callbacks, Arrays, etc. This allows you
to write your code in a very imperative manner without all the callbacks, but also brings the
power of RxJS whether you want to call `timeout` , `retry` , `catch` or any other method for
that matter. Note that this only yields a single value, but in RxJS terms, this is still quite
useful.

For example, we could get the HTML from Bing.com and write it to the console, with a
timeout of 5 seconds which will throw an error should it not respond in time. We could also
add in things like `retry` and `catch` so that we could for example try three times and then if
it fails, give a default response or cached version.

```
var Rx = require('rx');
var request = require('request');
var get = Rx.Observable.fromNodeCallback(request);

Rx.Observable.spawn(function* () {
  var data;
  try {
    data = yield get('http://bing.com').timeout(5000 /*ms*/);
  } catch (e) {
    console.log('Error %s', e);
  }

  console.log(data);
}).subscribe();
```

Mixing Operators with Generators

Many of the operators inside RxJS also support generators. For example, we could use the `Rx.Observable.from` method to take a generator function, in this case, a Fibonacci sequence, take 10 of them and display it.

```
function* fibonacci(){
  var fn1 = 1;
  var fn2 = 1;
  while (1) {
    var current = fn2;
    fn2 = fn1;
    fn1 = fn1 + current;
    yield current;
  }
}

Rx.Observable.from(fibonacci())
  .take(10)
  .subscribe(x => console.log('Value: %s', x));

//=> Value: 1
//=> Value: 1
//=> Value: 2
//=> Value: 3
//=> Value: 5
//=> Value: 8
//=> Value: 13
//=> Value: 21
//=> Value: 34
//=> Value: 55
```

That's just the beginning, as there are several operators such as `concatMap` / `selectConcat` and `flatMap` / `selectMany` which take iterables as an argument so that we can further enable composition. For example, we could project using generators from a `flatMap` operation.

```
var source = Rx.Observable.of(1,2,3)
  .flatMap(
    (x, i) => function* () { yield x; yield i; }(),
    (x, y, i1, i2) => x + y + i1 + i2
  );

var subscription = source.subscribe(
  x => console.log('onNext: %s', x),
  e => console.log('onError: %s', e),
  () => console.log('onCompleted'));

// => Next: 2
// => Next: 2
// => Next: 5
// => Next: 5
// => Next: 8
// => Next: 8
// => Completed
```

The future of JavaScript is exciting and generators add new possibilities to our applications to allow them to mix and match our programming styles.

Notice that in this sample, move becomes an observable sequence in which we can manipulate further. The Querying Observable Sequences topic will show you how you can project this sequence into a collection of Points type and filter its content, so that your application will only receive values that satisfy a certain criteria.

See Also

Concepts

- Querying Observable Sequences

Querying Observable Sequences

In Bridging to Events, we have converted existing DOM and Node.js events into observable sequences to subscribe to them. In this topic, we will look at the first-class nature of observable sequences as IObservable objects, in which generic LINQ operators are supplied by the Rx assemblies to manipulate these objects. Most operators take an observable sequence and perform some logic on it and output another observable sequence. In addition, as you can see from our code samples, you can even chain multiple operators on a source sequence to tweak the resulting sequence to your exact requirement.

Using Different Operators

We have already used the `create` and `range` operators in previous topics to create and return simple sequences. We have also used the `fromEvent` and `fromEventPattern` operators to convert existing events into observable sequences. In this topic, we will use other operators of the `Observable` type so that you can filter, group and transform data. Such operators take observable sequence(s) as input, and produce observable sequence(s) as output.

Combining different sequences

In this section, we will examine some of the operators that combine various observable sequences into a single observable sequence. Notice that data are not transformed when we combine sequences. In the following sample, we use the Concat operator to combine two sequences into a single sequence and subscribe to it. For illustration purpose, we will use the very simple `range(x, y)` operator to create a sequence of integers that starts with x and produces y sequential numbers afterwards.

```
var source1 = Rx.Observable.range(1, 3);
var source2 = Rx.Observable.range(1, 3);

source1.concat(source2)
    .subscribe(console.log.bind(console));

// => 1
// => 2
// => 3
// => 1
// => 2
// => 3
```

Notice that the resultant sequence is 1,2,3,1,2,3. This is because when you use the `concat` operator, the 2nd sequence (source2) will not be active until after the 1st sequence (source1) has finished pushing all its values. It is only after source1 has completed, then source2 will start to push values to the resultant sequence. The subscriber will then get all the values from the resultant sequence.

Compare this with the `merge` operator. If you run the following sample code, you will get 1,1,2,2,3,3. This is because the two sequences are active at the same time and values are pushed out as they occur in the sources. The resultant sequence only completes when the last source sequence has finished pushing values.

```
var source1 = Rx.Observable.range(1, 3);
var source2 = Rx.Observable.range(1, 3);

source1.merge(source2)
    .subscribe(console.log.bind(console));

// => 1
// => 1
// => 2
// => 2
// => 3
// => 3
```

Another comparison can be done with the `catch` operator. In this case, if source1 completes without any error, then source2 will not start. Therefore, if you run the following sample code, you will get 1,2,3 only since source2 (which produces 4,5,6) is ignored.

```
var source1 = Rx.Observable.range(1, 3);
var source2 = Rx.Observable.range(4, 6);

source1.catch(source2)
    .subscribe(console.log.bind(console));

// => 1
// => 2
// => 3
```

Finally, let's look at `onErrorResumeNext` . This operator will move on to source2 even if source1 cannot be completed due to an error. In the following example, even though source1 represents a sequence that terminates with an exception by using the `throw` operator, the subscriber will receive values (1,2,3) published by source2. Therefore, if you expect either source sequence to produce any error, it is a safer bet to use `onErrorResumeNext` to guarantee that the subscriber will still receive some values.

```
var source1 = Rx.Observable.throw(new Error('An error has occurred.'));
var source2 = Rx.Observable.range(1, 3);

source1.onErrorResumeNext(source2)
    .subscribe(console.log.bind(console));

// => 1
// => 2
// => 3
```

Projection

The `select` or `map` operator can translate each element of an observable sequence into another form.

In the following example, we project a sequence of strings into an a series of integers representing the length.

```
var array = ['Reactive', 'Extensions', 'RxJS'];

var seqString = Rx.Observable.from(array);

var seqNum = seqString.map(x =>x.length);

seqNum
    .subscribe(console.log.bind(console));

// => 8
// => 10
// => 4
```

In the following sample, which is an extension of the event conversion example we saw in the Bridging with Existing Events topic, we use the `select` or `map` operator to project the event arguments to a point of x and y. In this way, we are transforming a mouse move event sequence into a data type that can be parsed and manipulated further, as can be seen in the next "Filtering" section.

```
var move = Rx.Observable.fromEvent(document, 'mousemove');

var points = move.map(e => ({x: e.clientX, y: e.clientY }));

points.subscribe(
    pos => console.log('Mouse at point ' + pos.x + ', ' + pos.y));
```

Finally, let's look at the `selectMany` or `flatMap` operator. The `selectMany` or `flatMap` operator has many overloads, one of which takes a selector function argument. This selector function is invoked on every value pushed out by the source observable. For each of these values, the selector projects it into a mini observable sequence. At the end, the `selectMany` or `flatMap` operator flattens all of these mini sequences into a single resultant sequence, which is then pushed to the subscriber.

The observable returned from `selectMany` or `flatMap` publishes `onCompleted` after the source sequence and all mini observable sequences produced by the selector have completed. It fires `onError` when an error has occurred in the source stream, when an exception was thrown by the selector function, or when an error occurred in any of the mini observable sequences.

In the following example, we first create a source sequence which produces an integer every 5 seconds, and decide to just take the first 2 values produced (by using the `take` operator). We then use `selectMany` or `flatMap` to project each of these integers using another

sequence of {100, 101, 102}. By doing so, two mini observable sequences are produced, {100, 101, 102} and {100, 101, 102}. These are finally flattened into a single stream of integers of {100, 101, 102, 100, 101, 102} and pushed to the observer.

```
var source1 = Rx.Observable.interval(5000).take(2);
var proj = Rx.Observable.range(100, 3);
var resultSeq = source1.flatMap(proj);

var subscription = resultSeq.subscribe(
  x => console.log('onNext: %s', x),
  e => console.log('onError: %s', e.message),
  () => console.log('onCompleted'));

// => onNext: 100
// => onNext: 101
// => onNext: 102
// => onNext: 100
// => onNext: 101
// => onNext: 102
// => onCompleted
```

Filtering

In the following example, we use the `generate` operator to create a simple observable sequence of numbers. The `generate` operator has several versions including with relative and absolute time scheduling. In our example, it takes an initial state (0 in our example), a conditional function to terminate (fewer than 10 times), an iterator (+1), a result selector (a square function of the current value), and prints out only those smaller than 5 using the `filter` or `where` operators.

```
var seq = Rx.Observable.generate(
    0,
    i => i < 10,
    i => i + 1,
    i => i * i);

var source = seq.filter(n => n < 5);

var subscription = source.subscribe(
  x => console.log('onNext: %s', x),
  e => console.log('onError: %s', e.message),
  () => console.log('onCompleted'));

// => onNext: 0
// => onNext: 1
// => onNext: 4
// => onCompleted
```

The following example is an extension of the projection example you have seen earlier in this topic. In that sample, we have used the `select` or `map` operator to project the event arguments into a point with x and y. In the following example, we use the `filter` or `where` and `select` or `map` operator to pick only those mouse movement that we are interested. In this case, we filter the mouse moves to those over the first bisector (where the x and y coordinates are equal).

```
var move = Rx.Observable.fromEvent(document, 'mousemove');

var points = move.map(e => ({ x: e.clientX, y: e.clientY }));

var overfirstbisector = points.filter(pos => pos.x === pos.y);

var movesub = overfirstbisector.subscribe(pos => console.log('mouse at ' + pos.x + ', ' p
```

Time-based Operation

You can use the Buffer operators to perform time-based operations.

Buffering an observable sequence means that an observable sequence's values are put into a buffer based on either a specified timespan or by a count threshold. This is especially helpful in situations when you expect a tremendous amount of data to be pushed out by the sequence, and the subscriber does not have the resource to process these values. By buffering the results based on time or count, and only returning a sequence of values when the criteria is exceeded (or when the source sequence has completed), the subscriber can process OnNext calls at its own pace.

In the following example, we first create a simple sequence of integers for every second. We then use the `bufferWithCount` operator and specify that each buffer will hold 5 items from the sequence. The `onNext` is called when the buffer is full. We then tally the sum of the buffer using calling `Array.reduce`. The buffer is automatically flushed and another cycle begins. The printout will be 10, 35, 60… in which 10=0+1+2+3+4, 35=5+6+7+8+9, and so on.

```javascript
var seq = Rx.Observable.interval(1000);

var bufSeq = seq.bufferWithCount(5);

bufSeq
    .map(arr => arr.reduce((acc, x) => acc + x, 0))
    .subscribe(console.log.bind(console));

// => 10
// => 35
// => 60
...
```

We can also create a buffer with a specified time span in milliseconds. In the following example, the buffer will hold items that have accumulated for 3 seconds. The printout will be 3, 12, 21… in which 3=0+1+2, 12=3+4+5, and so on.

```javascript
var seq = Rx.Observable.interval(1000);

var bufSeq = seq.bufferWithTime(3000);

bufSeq
    .map(arr => arr.reduce((acc, x) => acc + x, 0))
    .subscribe(console.log.bind(console));
```

Note that if you are using any of the `buffer*` or `window*` operators, you have to make sure that the sequence is not empty before filtering on it.

Operators by Categories

The Operators by Categories topic lists of all major operators implemented by the `Observable` type by their categories; specifically: creation, conversion, combine, functional, mathematical, time, exceptions, miscellaneous, selection and primitives.

See Also

Reference

- Observable

Concepts

- Operators by Categories

Error Handling in the Reactive Extensions

One of the most difficult tasks in asynchronous programming is dealing with errors. Unlike interactive style programming, we cannot simply use the try/catch/finally approach that we use when dealing with blocking code.

```
try {
  for (var obj in objs) {
    doSomething(obj);
  }
} catch (e) {
  handleError(e);
} finally {
  doCleanup();
}
```

These actions mirror exactly our `Observer` class which has the following contract for handing zero to infinite items with `onNext` and optionally handling either an `Error` with `onError` or successful completion with `onCompleted`.

```
interface Observable<T> {
  onNext(value: T) : void
  onError(error: Error) : void
  onCompleted() : void
}
```

But the try/catch/finally approach won't work with asynchronous code. Instead, we have a myriad of ways of handling errors as they occur, and ensure proper disposal of resources.

For example, we might want to do the following:

- swallow the error and switch over to a backup Observable to continue the sequence
- swallow the error and emit a default item
- swallow the error and immediately try to restart the failed Observable
- swallow the error and try to restart the failed Observable after some back-off interval

We'll cover each of those scenarios and more in this section.

Catching Errors

The first topic is catching errors as they happen with our streams. In the Reactive Extensions, any error is propogated through the `onError` channel which halts the sequence. We can compensate for this by using the `catch` operator, at both the class and instance level.

Using the class level `catch` method, we can catch errors as they happen with the current sequence and then move to the next sequence should there be an error. For example, we could try getting data from several URLs, it doesn't matter which since they all have the same data, and then if that fails, default to a cached version, so an error should never propagate. One thing to note is that if `get('url')` calls succeed, then it will not move onto the next sequence in the list.

```
var source = Rx.Observable.catch(
  get('url1'),
  get('url2'),
  get('url3'),
  getCachedVersion()
);

var subscription = source.subscribe(
  data => {
    // Display the data as it comes in
  }
);
```

We also have an instance version of `catch` which can be used two ways. The first way is much like the example above, where we can take an existing stream, catch the error and move onto the next stream or `Promise`.

```
var source = get('url1').catch(getCachedVersion());

var subscription = source.subscribe(
  data => {
    // Display the data as it comes in
  }
);
```

The other overload of `catch` allows us to inspect the error as it comes in so we can decide which route to take. For example, if an error status code of 500 comes back from our web server, we can assume it is down and then use a cached version.

```
var source = get('url1').catch(e => {
  if (e.status === 500) {
    return cachedVersion();
  } else {
    return get('url2');
  }
});

var subscription = source.subscribe(
  data => {
    // Display the data as it comes in
  }
);
```

This isn't the only way to handle errors as there are plenty of others as you'll see below.

Ignoring Errors with `onErrorResumeNext`

The Reactive Extensions borrowed from a number of languages in our design. One of those features is bringing `On Error Resume Next` from Microsoft Visual Basic. This operation specifies that when a run-time error occurs, control goes to the statement immediately following the statement where the error occurred, and execution continues from that point. There are some instances with stream processing that you simply want to skip a stream which produces an error and move to the next stream. We can achieve this with a class based and instance based `onErrorResumeNext` method.

The class based `onErrorResumeNext` continues a stream that is terminated normally or by an `Error` with the next stream or `Promise`. Unlike `catch`, `onErrorResumeNext` will continue to the next sequence regardless of whether the previous was in error or not. To make this more concrete, let's use a simple example of mixing error sequences with normal sequences.

```
var source = Rx.Observable.onErrorResumeNext(
  Rx.Observable.just(42),
  Rx.Observable.throw(new Error()),
  Rx.Observable.just(56),
  Rx.Observable.throw(new Error()),
  Rx.Observable.just(78)
);

var subscription = source.subscribe(
  data => console.log(data)
);
// => 42
// => 56
// => 78
```

The instance based `onErrorResumeNext` is similar to the class based version, the only difference being that it is attached to the prototype, but can take another sequence or `Promise` and continue.

Retrying Sequences

When catching errors isn't enough and we want to retry our logic, we can do so with `retry` or `retryWhen` operators. With the `retry` operator, we can try a certain operation a number of times before an error is thrown. This is useful when you need to get data from a resource which may have intermittent failures due to load or any other issue.

Let's take a look at a simple example of trying to get some data from a URL and giving up after three tries.

```javascript
// Try three times to get the data and then give up
var source = get('url').retry(3);

var subscription = source.subscribe(
  data => console.log(data),
  err => console.log(err)
);
```

In the above example, it will give up after three tries and thus call `onError` if it continues to fail after the third try. We can remedy that by adding `catch` to use an alternate source.

```javascript
// Try three times to get the data and then return cached data if still fails
var source = get('url').retry(3).catch(cachedVersion());

var subscription = source.subscribe(
  data => {
    // Displays the data from the URL or cached data
    console.log(data);
  }
);
```

The above case retries immediately upon failure. But what if you want to control when a retry happens? We have the `retryWhen` operator which allows us to deeply control when the next try happens. We incrementally back off trying again by using the following method:

```
var source = get('url').retryWhen(
   attempts =>
    attempts
      .zip(Observable.range(1, 3), (_, i) => i)
      .flatMap(i => {
        console.log('delay retry by ' + i + ' second(s)');
        return Rx.Observable.timer(i * 1000);
      });
);

var subscription = source.subscribe(
  data => {
    // Displays the data from the URL or cached data
    console.log(data);
  }
);
// => delay retry by 1 second(s)
// => delay retry by 2 second(s)
// => Data
```

Ensuring Cleanup with Finally

We've already covered the try/catch part of try/catch/finally, so what about finally? We have the `finally` operator which calls a function after the source sequence terminates gracefully or exceptionally. This is useful if you are using external resources or need to free up a particular variable upon completion.

In this example, we can ensure that our `WebSocket` will indeed be closed once the last message is processed.

```
var socket = new WebSocket('ws://someurl', 'xmpp');

var source = Rx.Observable.from(data)
  .finally(() => socket.close());

var subscription = source.subscribe(
  data => {
    socket.send(data);
  }
);
```

But,we can do a better job in terms of managing resources if need be by using the `using` method.

Ensuring Resource Disposal

As stated above, `finally` can be used to ensure proper cleanup of any resources or perform any side effects as necessary. There is a cleaner approach we can take by creating a disposable wrapper around our object with a `dispose` method so that when our scope is complete, then the resource is automatically disposed through the `using` operator.

```javascript
function DisposableWebSocket(url, protocol) {
  var socket = new WebSocket(url, protocol);

  // Create a way to close the WebSocket upon completion
  var d = Rx.Disposable.create(() => socket.close());

  d.socket = socket;

  return d;
}

var source = Rx.Observable.using(
  () => new DisposableWebSocket('ws://someurl', 'xmpp'),
  d =>
    Rx.Observable.from(data)
      .tap(data => d.socket.send(data));
  }
);

var subscription = source.subscribe();
```

Delaying Errors with `mergeDelayError`

Another issue may arise when you are dealing with flattening sequences into a single sequence and there may be errors along the way. We want a way to flatten without being interrupted by one of our sources being in error. This is much like the other operator `mergeAll` but the main difference is, instead of immediately bubbling up the error, it holds off until the very end.

To illustrate, we can create this little sample that has an errored sequence in the middle when it is trying to flatten the sequences.

```
var source1 = Rx.Observable.of(1,2,3);
var source2 = Rx.Observable.throwError(new Error('woops'));
var source3 = Rx.Observable.of(4,5,6);

var source = Rx.Observable.mergeDelayError(source1, source2, source3);

var subscription = source.subscribe(
  x => console.log('onNext: %s', x),
  e => console.log('onError: %s', e),
  () => console.log('onCompleted'));

// => 1
// => 2
// => 3
// => 4
// => 5
// => 6
// => Error: Error: woops
```

Further Reading

- Using Generators For Try/Catch Operations
- Testing and Debugging Your RxJS Application

Transducers with Observable Sequences

Much like Language Integrated Query (LINQ), Transducers are composable algorithmic transformations. They, like LINQ, are independent from the context of their input and output sources and specify only the essence of the transformation in terms of an individual element. Because transducers are decoupled from input or output sources, they can be used in many different processes - collections, streams, observables, etc. Transducers compose directly, without awareness of input or creation of intermediate aggregates. There are two major libraries currently out there, Cognitect's `transducer-js` and James Long's `transducers.js` which are both great for getting high performance over large amounts of data. Because it is collection type neutral, it is a perfect fit for RxJS to do transformations over large collections.

The word `transduce` is just a combination of `transform` and `reduce`. The reduce function is the base transformation; any other transformation can be expressed in terms of it (`map`, `filter`, etc).

```
var arr = [1, 2, 3, 4];

arr.reduce((result, x) => result.concat(x + 1), []);

// => [ 2, 3, 4, 5 ]
```

Using transducers, we can model the following behavior while breaking apart the map aspect of adding 1 to the concat operation, adding the seed and then the "collection" to transduce.

```
var arr = [1, 2, 3, 4];

function increment(x) { return x + 1; }
function concatItem(acc, x) { return acc.concat(x); }

transduce(map(increment), concatItem, [], arr);

// => [ 2, 3, 4, 5 ]
```

Using Cognitect's `transducers-js` library, we can easily accomplish what we had above.

```
var t = transducers;

var arr = [1, 2, 3, 4];

function increment(x) { return x + 1; }

into([], t.comp(t.map(increment)), arr);

// => [ 2, 3, 4, 5 ]
```

We can go a step further and add filtering as well to get only even values.

```
var t = transducers;

var arr = [1, 2, 3, 4];

function increment(x) { return x + 1; }
function isEven(x) { return x % 2 === 0; }

into([], t.comp(t.map(increment), t.filter(isEven)), arr);

// => [ 2, 4 ]
```

Since it works so well using Arrays, there's no reason why it cannot work for Observable sequences as well. To that end, we have introduced the `transduce` method which acts exactly like it does for Arrays, but for Observable sequences. Once again, let's go over the above example, this time using an Observable sequence.

```
var t = transducers;

var source = Rx.Observable.range(1, 4);

function increment(x) { return x + 1; }
function isEven(x) { return x % 2 === 0; }

var transduced = source.transduce(t.comp(t.map(increment), t.filter(isEven)));

transduced.subscribe(
  x => console.log('onNext: %s', x),
  e => console.log('onError: %s', e),
  () => console.log('onCompleted'));

// => Next: 2
// => Next: 4
// => Completed
```

Note that this above example also works the same with `transducers.js` as well with little to no modification. This example will in fact work faster than the traditional LINQ style (as of now) which most use currently.

```javascript
var source = Rx.Observable.range(1, 4);

function increment(x) { return x + 1; }
function isEven(x) { return x % 2 === 0; }

var transduced = source.map(increment).filter(isEven);

transduced.subscribe(
  x => console.log('onNext: %s', x),
  e => console.log('onError: %s', e),
  () => console.log('onCompleted'));

// => Next: 2
// => Next: 4
// => Completed
```

This opens up a wide new set of possibilities making RxJS even faster over large collections with no intermediate Observable sequences.

Backpressure

When it comes to streaming data, streams can be overly chatty in which the consumer cannot keep up with the producer. To that end, we need mechanisms to control the source so that the consumer does not get overwhelmed. These mechanisms can come in either the form of lossy or loss-less operations, each of which depends on the requirements. For example, if you miss a few mouse movements, it may not be a problem, however, if you miss a few bank transactions, that could be a definite problem. This section covers which techniques you can use to handle backpressure in either lossy or loss-less ways.

For example, imagine using the `zip` operator to zip together two infinite Observables, one of which emits items twice as frequently as the other. A naive implementation of the zip operator would have to maintain an ever-expanding buffer of items emitted by the faster Observable to eventually combine with items emitted by the slower one. This could cause RxJS to seize an unwieldy amount of system resources.

Hot and Cold Observables and Multicast

A cold Observable emits a particular sequence of items, but can begin emitting this sequence when its Observer finds it to be convenient, and at whatever rate the Observer desires, without disrupting the integrity of the sequence. For example if you convert a iterable such as array, Map, Set, or generator into an Observable, that Observable will emit the same sequence of items no matter when it is later subscribed to or how frequently those items are observed. Examples of items emitted by a cold Observable might include the results of a database query, file retrieval, or web request.

A hot Observable begins generating items to emit immediately when it is created. Subscribers typically begin observing the sequence of items emitted by a hot Observable from somewhere in the middle of the sequence, beginning with the first item emitted by the Observable subsequent to the establishment of the subscription. Such an Observable emits items at its own pace, and it is up to its observers to keep up. Examples of items emitted by a hot Observable might include mouse & keyboard events, system events, or stock prices.

When a cold Observable is multi-cast (when it is converted into a `ConnectableObservable` and its `connect` method is called), it effectively becomes hot and for the purposes of backpressure and flow-control it should be treated as a hot Observable.

Cold Observables are ideal subjects for the reactive pull model of backpressure described below. Hot observables are typically not designed to cope well with a reactive pull model, and are better candidates for some of the other flow control strategies discussed on this page, such as the use of the `pausableBuffered` or `pausable` operators, throttling, buffers, or windows.

Lossy Backpressure

There are a number of ways that an observable sequence can be controlled so that the consumer does not get overwhelmed through lossy operations, meaning that packets will be dropped in between pause and resume actions.

Debounce

The first technique for lossy backpressure is called `debounce` which only emits an item from the source Observable after a particular timespan has passed without the Observable emitting any other items. This is useful in scenarios such as if the user is typing too fast and you do not want to yield every keystroke, and instead wait half a second after the person stopped typing before yielding the value.

```
var debounced = Rx.Observable.fromEvent(input, 'keyup')
  .map(e => e.target.value)
  .debounce(500 /* ms */);

debounced.subscribeOnNext(value => console.log('Input value: %s', value));
```

Throttling

Another technique to deal with an observable sequence which is producing too much for the consumer is through throttling with the use of the `throttleFirst` method which emits the first items emitted by an Observable within periodic time intervals. Throttling can be especially useful for rate limiting execution of handlers on events like resize and scroll.

```
var throttled = Rx.Observable.fromEvent(window, 'resize')
  .throttleFirst(250 /* ms */);

throttled.subscribeOnNext(e => {
  console.log('Window inner height: %d', window.innerHeight);
  console.log('Window inner width: %d', window.innerWidth);
});
```

Sampling Observables

You can also at certain intervals extract values from the observable sequence using the `sample` method. This is useful if you want values from say a stock ticker every five seconds or so without having to consume the entire observable sequence.

```
var sampled = getStockData()
  .sample(5000 /* ms */);

sampled.subscribeOnNext(data => console.log('Stock data: %o', data));
```

Pausable Observables

The ability to pause and resume is also a powerful concept which is offered in RxJS in both lossy and loss-less versions. In the case of lossy backpressure, the `pausable` operator can be used to stop listening and then resume listening at a later time by calling `pause` and `resume` respectively on the observable sequence. For example we can take some observable sequence and call `pausable`, then call `pause` to pause the sequence and `resume` within 5 seconds. Note that any data that comes in between the pause and resume are lost. Note that this only works for hot observables and is unsuitable for cold observables as they will restart upon resume.

```
var pausable = getSomeObservableSource()
  .pausable();

pausable.subscribeOnNext(data => console.log('Data: %o', data));

pausable.pause();

// Resume in five seconds
setTimeout(() => pausable.resume(), 5000);
```

Loss-less Backpressure

In addition to supporting lossy backpressure mechanisms, RxJS also supports ways of getting the data in such a way that it is able to be fully consumed by the consumer at its own pace. There are a number of strategies at work including using buffers that work with timespans, count or both, pausable buffers, reactive pull, etc.

Buffers and Windows

The first strategy of dealing with an overly chatty producer is through the use of buffers. This allows the consumer to set either the number of items they wish to wait for at a time, or a particular timespan, or both, whichever comes first. This is useful in a number of cases, for example if you want some data within a window for comparison purposes in addition to chunking up data as you need it.

The `bufferWithCount` method allows us to specify the number of items that you wish to capture in a buffer array before yielding it to the consumer. An impractical yet fun use of this is to calculate whether the user has input the Konami Code for example.

```javascript
var codes = [
  38, // up
  38, // up
  40, // down
  40, // down
  37, // left
  39, // right
  37, // left
  39, // right
  66, // b
  65  // a
];

function isKonamiCode(buffer) {
  return codes.toString() === buffer.toString();
}

var keys = Rx.Observable.fromEvent(document, 'keyup')
  .map(e => e.keyCode)
  .bufferWithCount(10, 1)
  .filter(isKonamiCode)
  .subscribeOnNext(() => console.log('KONAMI!'));
```

On the other hand, you can also get the data within a buffer for a given amount of time with the `bufferWithTime` . This is useful for example if you are tracking volume of data that is coming across the network, which can then be handled uniformly.

```javascript
var source = getStockData()
  .bufferWithTime(5000, 1000) // time in milliseconds
  .subscribeOnNext(data => data.forEach(d => console.log('Stock: %o', d)));
```

In order to keep buffers from filling too quickly, there is a method to cap the buffer by specifying ceilings for count and timespan, whichever occurs first. For example, the network could be particularly quick with the data for the specified time, and other times not, so to keep the data levels even, you can specify this threshold via the `bufferWithTimeOrCount` method

```
var source = getStockData()
  .bufferWithTimeOrCount(5000 /* ms */, 100 /* items */)
  .subscribeOnNext(data => data.forEach(d => console.log('Stock: %o', d)));
```

Pausable Buffers

The `pausable` method is great at dealing with hot observables where you would want to pause and resume while dropping data, however, you may want to preserve that data between the `pause` and `resume` calls. To that end, we have introduced the `pausableBuffered` method which keeps a running buffer between `pause` is called and is drained when `resume` is called. This then leaves the discretion up to the developer to decide when to pause and resume and in the mean time, no data is lost.

```
var source = getStockData()
  .pausableBuffered();

source.subscribeOnNext(stock => console.log('Stock data: %o', stock));

source.pause();

// Resume after five seconds
setTimeout(() => {
  // Drains the buffer and subscribeOnNext is called with the data
  source.resume();
}, 5000);
```

Controlled Observables

In more advanced scenarios, you may want to control the absolute number of items that you receive at a given time, and the rest is buffered via the `controlled` method. For example, you can pull 10 items, followed by 20 items, and is up to the discretion of the developer. This is more in-line with the efforts from the Reactive Streams effort to effectively turn the push stream into a push/pull stream.

```
var source = getStockData()
  .controlled();

source.subscribeOnNext(stock => console.log('Stock data: %o', stock));

source.request(2);

// Keep getting more after 5 seconds
setInterval(=> source.request(2), 5000);
```

Future Work

This is of course only the beginning of the work with backpressure as there are many other strategies that can be considered. In future versions of RxJS, the idea of the controlled observable will be baked into the subscription itself which then allows the backpressure to be an essential part of the contract or requesting n number of items.

Operators by Categories

This topic lists all major operators implemented by the `Observable` type by their categories, specifically: creation, conversion, combine, functional, mathematical, time, exceptions, miscellaneous, selection and primitives.

Operators by Categories

Usage	Operators
Creating an observable sequence	1. create 2. defer 3. generate 4. generateWithAbsoluteTime 5. generateWithRelativeTime 6. range 7. using
Converting events or asynchronous patterns to observable sequences, or between Arrays and observable sequences.	1. from 2. fromArray 3. fromCallback 4. fromNodeCallback 5. fromEvent 6. fromEventPattern 7. fromPromise 8. of 9. toArray 10. toMap 11. toPromise 12. toSet
Combining multiple observable sequences into a single sequence.	1. amb 2. prototype.amb 3. combineLatest 4. concat 5. prototype.concat 6. startWith 7. merge 8. prototype.merge 9. mergeAll 10. repeat 11. prototype.repeat 12. withLatestFrom 13. zip 14. prototype.zip

Functional - Sharing Side Effects	1. let 2. publish 3. publishLast 4. publishValue 5. replay 6. share 7. shareLast 8. shareReplay 9. shareValue
Mathemathical operators on sequences	1. aggregate 2. average 3. count 4. max 5. maxBy 6. min 7. minBy 8. reduce 9. sum
Time-based operations	1. debounce 2. debounceWithSelector 3. delay 4. interval 5. timeInterval 6. timer 7. timeout 8. timeoutWithSelector 9. timestamp
Handling Exceptions	1. catch 2. prototype.catch 3. finally 4. onErrorResumenext 5. prototype.onErrorResumeNext 6. retry
	1. concatMap 2. concatMapObserver 3. elementAt 4. elementAtOrDefault 5. filter 6. flatMap 7. flatMapLatest 8. flatMapObserver 9. find 10. findIndex 11. first 12. firstOrDefault 13. includes 14. last

Filtering and selecting values in a sequence	15. lastOrDefault 16. map 17. pluck 18. select 19. selectConcat 20. selectMany 21. selectManyObserver 22. selectSwitch 23. single 24. singleOrDefault 25. skip 26. skipLast 27. skipLastWithTime 28. skipUntil 29. skipWhile 30. take 31. takeLast 32. takeLastBuffer 33. takeLastBufferWithTime 34. takeLastWithTime 35. takeWhile 36. where
Grouping and Windowing	1. buffer 2. bufferWithCount 3. bufferWithTimeOrCount 4. groupBy 5. groupByUntil 6. groupJoin 7. join 8. window 9. windowWithCount 10. windowWithTime 11. windowWithTimeOrCount
Imperative Operators	1. case 2. do 3. doOnNext 4. doOnError 5. doOnCompleted 6. doWhile 7. for 8. if 9. tap 10. tapOnNext 11. tapOnError 12. tapOnCompleted 13. while
Primitives	1. empty 2. never 3. return

	4. throw

See Also

Reference

- `Observable`

Concepts

- Querying Observable Sequences

Using Subjects

The `Subject` class inherits both `Observable` and `Observer` , in the sense that it is both an observer and an observable. You can use a subject to subscribe all the observers, and then subscribe the subject to a backend data source. In this way, the subject can act as a proxy for a group of subscribers and a source. You can use subjects to implement a custom observable with caching, buffering and time shifting. In addition, you can use subjects to broadcast data to multiple subscribers.

By default, subjects do not perform any synchronization across threads. They do not take a scheduler but rather assume that all serialization and grammatical correctness are handled by the caller of the subject. A subject simply broadcasts to all subscribed observers in the thread-safe list of subscribers. Doing so has the advantage of reducing overhead and improving performance.

Using Subjects

In the following example, we create a subject, subscribe to that subject and then use the same subject to publish values to the observer. By doing so, we combine the publication and subscription into the same source.

In addition to taking an `Observer` , the `subscribe` method can also take a function for onNext, which means that the action will be executed every time an item is published. In our sample, whenever onNext is invoked, the item will be written to the console.

```javascript
var subject = new Rx.Subject();

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

subject.onNext(1);
// => onNext: 1

subject.onNext(2);
// => onNext: 2

subject.onCompleted();
// => onCompleted

subscription.dispose();
```

The following example illustrates the proxy and broadcast nature of a `Subject` . We first create a source sequence which produces an integer every 1 second. We then create a `Subject` , and pass it as an observer to the source so that it will receive all the values pushed out by this source sequence. After that, we create another two subscriptions, this time with the subject as the source. The `subSubject1` and `subSubject2` subscriptions will then receive any value passed down (from the source) by the `Subject` .

```
// Every second
var source = Rx.Observable.interval(1000);

var subject = new Rx.Subject();

var subSource = source.subscribe(subject);

var subSubject1 = subject.subscribe(
    x => console.log('Value published to observer #1: ' + x),
    e => console.log('onError: ' + e.message),
    () => console.log('onCompleted'));

var subSubject2 = subject.subscribe(
    x => console.log('Value published to observer #2: ' + x),
    e => console.log('onError: ' + e.message),
    () => console.log('onCompleted'));

setTimeout(() => {
    // Clean up
    subject.onCompleted();
    subSubject1.dispose();
    subSubject2.dispose();
}, 5000);

// => Value published to observer #1: 0
// => Value published to observer #2: 0
// => Value published to observer #1: 1
// => Value published to observer #2: 1
// => Value published to observer #1: 2
// => Value published to observer #2: 2
// => Value published to observer #1: 3
// => Value published to observer #2: 3
// => onCompleted
// => onCompleted
```

Different types of Subjects

The `Subject` object in the RxJS library is a basic implementation, but you can create your own using the `Subject.create` method. There are other implementations of Subjects that offer different functionalities. All of these types store some (or all of) values pushed to them

via onNext, and broadcast it back to its observers. In this way, they convert a Cold Observable into a Hot one. This means that if you Subscribe to any of these more than once (i.e. subscribe -> unsubscribe -> subscribe again), you will see at least one of the same value again. For more information on hot and cold observables, see the last section of the Creating and Subscribing to Simple Observable Sequences topic.

`ReplaySubject` stores all the values that it has published. Therefore, when you subscribe to it, you automatically receive an entire history of values that it has published, even though your subscription might have come in after certain values have been pushed out. `BehaviourSubject` is similar to `ReplaySubject`, except that it only stored the last value it published. `BehaviourSubject` also requires a default value upon initialization. This value is sent to observers when no other value has been received by the subject yet. This means that all subscribers will receive a value instantly on `subscribe`, unless the `Subject` has already completed. `AsyncSubject` is similar to the Replay and Behavior subjects, however it will only store the last value, and only publish it when the sequence is completed. You can use the `AsyncSubject` type for situations when the source observable is hot and might complete before any observer can subscribe to it. In this case, `AsyncSubject` can still provide the last value and publish it to any future subscribers.

Using Schedulers

A scheduler controls when a subscription starts and when notifications are published. It consists of three components. It is first a data structure. When you schedule for tasks to be completed, they are put into the scheduler for queueing based on priority or other criteria. It also offers an execution context which denotes where the task is executed (e.g., in the immediately, current thread, or in another callback mechanism such as `setTimeout` or `process.nextTick`). Lastly, it has a clock which provides a notion of time for itself (by accessing the `now` method of a scheduler). Tasks being scheduled on a particular scheduler will adhere to the time denoted by that clock only.

Schedulers also introduce the notion of virtual time (denoted by the `VirtualTimeScheduler` type), which does not correlate with real time that is used in our daily life. For example, a sequence that is specified to take 100 years to complete can be scheduled to complete in virtual time in a mere 5 minutes. This will be covered in the Testing and Debugging Observable Sequences topic.

Scheduler Types

The various Scheduler types provided by Rx all implement the `Scheduler` methods. Each of these can be created and returned by using static properties of the `Scheduler` object. The `ImmediateScheduler` (by accessing the static `immediate` property) will start the specified action immediately. The `CurrentThreadScheduler` (by accessing the static `currentThread` property) will schedule actions to be performed on the thread that makes the original call. The action is not executed immediately, but is placed in a queue and only executed after the current action is complete. The `DefaultScheduler` (by accessing the static `default` property) will schedule actions to be performed on a asynchronous callback, which is optimized for the particular runtime, such as `setImmediate` or `process.nextTick` on Node.js or in the browser with a fallback to `setTimeout` .

Using Schedulers

You may have already used schedulers in your Rx code without explicitly stating the type of schedulers to be used. This is because all Observable operators that deal with concurrency have optional schedulers. If you do not provide the scheduler, RxJS will pick a default scheduler by using the principle of least concurrency. This means that the scheduler which introduces the least amount of concurrency that satisfies the needs of the operator is

chosen. For example, for operators returning an observable with a finite and small number of messages, RxJS calls `immediate` . For operators returning a potentially large or infinite number of messages, `currentThread` is called. For operators which use timers, `default` is used.

Because RxJS uses the least concurrency scheduler, you can pick a different scheduler if you want to introduce concurrency for performance purpose. To specify a particular scheduler, you can use those operator methods that take a scheduler, e.g., `return(42, Rx.Scheduler.default)` .

In the following example, the source observable sequence is producing values at a frantic pace. The default scheduler of the generate operator would place onNext messages on the `currentThread` .

```
var obs = Rx.Observable.generate(
    0,
    () => true,
    x => x + 1,
    x => x);
```

This will queue up on the observer quickly. We can improve this code by using the observeOn operator, which allows you to specify the context that you want to use to send pushed notifications (onNext) to observers. By default, the `observeOn` operator ensures that `onNext` will be called as many times as possible on the current thread. You can use its overloads and redirect the `onNext` outputs to a different context. In addition, you can use the `subscribeOn` operator to return a proxy observable that delegates actions to a specific scheduler. For example, for a UI-intensive application, you can delegate all background operations to be performed on a scheduler running in the background by using `subscribeOn` and passing to it the `DefaultScheduler` .

The following example will schedule any onNext notifications on the current Dispatcher, so that any value pushed out is sent on the UI thread. This is especially beneficial to Silverlight developers who use RxJS.

```
Rx.Observable.generate(
    0,
    () => true,
    x => x + 1,
    x => x
    )
    .observeOn(Rx.Scheduler.default)
    .subscribe(...);
```

Instead of using the `observeOn` operator to change the execution context on which the observable sequence produces messages, we can create concurrency in the right place to begin with. As operators parameterize introduction of concurrency by providing a scheduler argument overload, passing the right scheduler will lead to fewer places where the ObserveOn operator has to be used. For example, we can unblock the observer and subscribe to the UI thread directly by changing the scheduler used by the source, as in the following example. In this code, by using the `generate` method passing a scheduler, and providing the `Rx.Scheduler.default` instance, all values pushed out from this observable sequence will originate via an asynchronous callback.

```
Rx.Observable.generate(
    0,
    () => true,
    x => x + 1,
    x => x,
    Rx.Scheduler.default)
    .subscribe(...);
```

You should also note that by using the `observeOn` operator, an action is scheduled for each message that comes through the original observable sequence. This potentially changes timing information as well as puts additional stress on the system. If you have a query that composes various observable sequences running on many different execution contexts, and you are doing filtering in the query, it is best to place `observeOn` later in the query. This is because a query will potentially filter out a lot of messages, and placing the `observeOn` operator earlier in the query would do extra work on messages that would be filtered out anyway. Calling the `observeOn` operator at the end of the query will create the least performance impact.

Another advantage of specifying a scheduler type explicitly is that you can introduce concurrency for performance purpose, as illustrated by the following code.

```
seq.groupBy(...)
  .map(x => x.observeOn(Rx.Scheduler.default))
  .map(x => expensive(x))  // perform operations that are expensive on resources
```

When to Use Which Scheduler

To make things a little easier when you are creating your own operators, or using the standard built-in ones, which scheduler you should use. The following table lays out each scenario with the suggested scheduler.

Scenario	Scheduler
Constant Time Operations	Rx.Scheduler.immediate
Tail Recursive Operations	Rx.Scheduler.immediate
Iteration Operations	Rx.Scheduler.currentThread
Time-based Operations	Rx.Scheduler.default
Asynchronous Conversions	Rx.Scheduler.default
Historical Data Operations	Rx.HistoricalScheduler
Unit Testing	Rx.TestScheduler

See Also

Reference

- Testing and Debugging Observable Sequences

Testing your Rx application

Let's face it, testing asynchronous code is a pain. In JavaScript, with so many asynchronous things to coordinate, testing is too hard for anyone to wrap their minds around. Luckily the Reactive Extensions for JavaScript makes this easy.

If you have an observable sequence that publishes values over an extended period of time, testing it in real time can be a stretch. The RxJS library provides the `TestScheduler` type to assist testing this kind of time-dependent code without actually waiting for time to pass. The `TestScheduler` inherits `VirtualScheduler` and allows you to create, publish and subscribe to sequences in emulated time. For example, you can compact a publication which takes 5 days to complete into a 2 minute run, while maintaining the correct scale. You can also take a sequence which actually has happened in the past (e.g., a sequence of stock ticks for a previous year) and compute or subscribe to it as if it is pushing out new values in real time.

The factory methods `startWithTiming` , `startWithCreate` and `startWithDispose` executes all scheduled tasks until the queue is empty, or you can specify a time to so that queued-up tasks are only executed to the specified time.

The following example creates a hot observable sequence with specified `onNext` notifications. It then starts the test scheduler and specifies when to subscribe to and dispose of the hot observable sequence. The Start method returns an instance of an `Observer` , which contains a `messages` property that records all notifications in a list.

After the sequence has completed, we use can define method such as `collectionAssert.assertEqual` to compare the `messages` property, together with a list of expected values to see if both are identical (with the same number of items, and items are equal and in the same order). By doing so, we can confirm that we have indeed received the notifications that we expect. In our example, since we only start subscribing at 150, we will miss out the value 'abc'. However, when we compare the values we have received so far at 400, we notice that we have in fact received all the published values after we subscribed to the sequence. And we also verify that the OnCompleted notification was fired at the right time at 500. In addition, subscription information is also captured by the `Observable` type returned by the `createHotObservable` method.

In the same way, you can use that same defined method such as our `collectionAssert.assertEqual` below to confirm that subscriptions indeed happened at expected times. It is easy to wrap this for your favorite unit testing framework whether it is QUnit, Mocha, Jasmine, etc. In this example, we'll write a quick wrapper for QUnit.

```
function createMessage(actual, expected) {
```

```
    return 'Expected: [' + expected.toString() + ']\r\nActual: [' + actual.toString() + '
}

// Using QUnit testing for assertions
var collectionAssert = {
    assertEqual: (expected, actual) => {
        var comparer = Rx.internals.isEqual,
            isOk = true;

        if (expected.length !== actual.length) {
            ok(false, 'Not equal length. Expected: ' + expected.length + ' Actual: ' + ac
            return;
        }

        for(var i = 0, len = expected.length; i < len; i++) {
            isOk = comparer(expected[i], actual[i]);
            if (!isOk) {
                break;
            }
        }

        ok(isOk, createMessage(expected, actual));
    }
};


var onNext = Rx.ReactiveTest.onNext,
    onCompleted = Rx.ReactiveTest.onCompleted,
    subscribe = Rx.ReactiveTest.subscribe;

test('buffer should join strings', () => {

    var scheduler = new Rx.TestScheduler();

    var input = scheduler.createHotObservable(
        onNext(100, 'abc'),
        onNext(200, 'def'),
        onNext(250, 'ghi'),
        onNext(300, 'pqr'),
        onNext(450, 'xyz'),
        onCompleted(500)
    );

    var results = scheduler.startWithTiming(
        function () {
            return input.buffer(() => input.debounce(100, scheduler))
            .map(b => b.join(','));
        },
        50,  // created
        150, // subscribed
        600  // disposed
    );

    collectionAssert.assertEqual(results.messages, [
```

```
        onNext(400, 'def,ghi,pqr'),
        onNext(500, 'xyz'),
        onCompleted(500)
    ]);

    collectionAssert.assertEqual(input.subscriptions, [
        subscribe(150, 500),
        subscribe(150, 400),
        subscribe(400, 500)
    ]);
});
```

Debugging your Rx application

You can use the `do` operator to debug your Rx application. The `do` operator allows you to specify various actions to be taken for each item of observable sequence (e.g., print or log the item, etc.). This is especially helpful when you are chaining many operators and you want to know what values are produced at each level.

In the following example, we are going to reuse the Buffer example which generates integers every second, while putting them into buffers that can hold 5 items each. In our original example in the Querying Observable Sequences topic, we subscribe only to the final `Observable` sequence when the buffer is full (and before it is emptied). In this example, however, we will use the `do` operator to print out the values when they are being pushed out by the original sequence (an integer every second). When the buffer is full, we use the `do` operator to print the status, before handing over all this as the final sequence for the observer to subscribe.

```
var seq1 = Rx.Observable.interval(1000)
    .do(console.log.bind(console))
    .bufferWithCount(5)
    .do(x => console.log('buffer is full'))
    .subscribe(x => console.log('Sum of the buffer is ' + x.reduce((acc, x) => acc + x, 0)

// => 0
// => 1
// => 2
// => 3
// => 4
// => buffer is full
// => Sum of the buffer is 10
// ...
```

As you can see from this sample, a subscription is on the recipient end of a series of chained observable sequences. At first, we create an observable sequence of integers separate by a second using the Interval operator. Then, we put 5 items into a buffer using the Buffer operator, and send them out as another sequence only when the buffer is full. Lastly, this is handed over to the Subscribe operator. Data propagate down all these intermediate sequences until they are pushed to the observer. In the same way, subscriptions are propagated in the reverse direction to the source sequence. By inserting the `do` operator in the middle of such propagations, you can "spy" on such data flow just like you use console.log perform debugging.

You can also use the `timestamp` operator to verify the time when an item is pushed out by an observable sequence. This can help you troubleshoot time-based operations to ensure accuracy. Recall the following example from the Creating and Subscribing to Simple Observable Sequences topic, in which we chain the `timestamp` operator to the query so that each value pushed out by the source sequence will be appended by the time when it is published. By doing so, when we subscribe to this source sequence, we can receive both its value and timestamp.

```javascript
console.log('Current time: ' + Date.now());

var source = Rx.Observable.timer(5000, 1000)
    .timestamp();

var subscription = source.subscribe(x => console.log(x.value + ': ' + x.timestamp));

/* Output will look similar to this */
// => Current time: 1382646947400
// => 0: 1382646952400
// => 1: 1382646953400
// => 2: 1382646954400
// => 3: 1382646955400
// => 4: 1382646956400
// => 5: 1382646957400
// => 6: 1382646958400
```

By using the `timestamp` operator, we have verified that the first item is indeed pushed out 5 seconds after the sequence, and each item is published 1 second later.

You can remove any `do` and `map` or `select` calls after you finish debugging.

Long Stack Traces Support

When dealing with large RxJS applications, debugging and finding where the error occurred can be a difficult operation. As you chain more and more operators together, the longer the stack trace gets, and the harder it is to find out where things went wrong. Inspiration for this feature came from the Q library from @kriskowal which helped us get started.

RxJS comes with optional support for "long stack traces" where the stack property of Error from onError calls is rewritten to be traced along asynchronous jumps instead of stopping at the most recent one. As an example:

```
var Rx = require('rx');

var source = Rx.Observable.range(0, 100)
  .timestamp()
  .map((x) => {
    if (x.value > 98) throw new Error();
    return x;
  });

source.subscribeOnError(err => console.log(err.stack));
```

The error stack easily becomes unreadable and hard to find where the error actually occurred:

```
$ node example.js

  Error
  at C:\GitHub\example.js:6:29
  at AnonymousObserver._onNext (C:\GitHub\rxjs\dist\rx.all.js:4013:31)
  at AnonymousObserver.Rx.AnonymousObserver.AnonymousObserver.next (C:\GitHub\rxjs\dist\r
  at AnonymousObserver.Rx.internals.AbstractObserver.AbstractObserver.onNext (C:\GitHub\r
  at AutoDetachObserverPrototype.next (C:\GitHub\rxjs\dist\rx.all.js:9226:23)
  at AutoDetachObserver.Rx.internals.AbstractObserver.AbstractObserver.onNext (C:\GitHub\
  at AnonymousObserver._onNext (C:\GitHub\rxjs\dist\rx.all.js:4018:18)
  at AnonymousObserver.Rx.AnonymousObserver.AnonymousObserver.next (C:\GitHub\rxjs\dist\r
  at AnonymousObserver.Rx.internals.AbstractObserver.AbstractObserver.onNext (C:\GitHub\r
  at AutoDetachObserverPrototype.next (C:\GitHub\rxjs\dist\rx.all.js:9226:23)
```

Instead, we can turn on this feature by setting the following flag:

```
Rx.config.longStackSupport = true;
```

When running the same example again with the flag set at the top, our stack trace looks much nicer and indicates exactly where the error occurred:

```
$ node example.js

Error
at C:\GitHub\example.js:6:29
From previous event:
at Object.<anonymous> (C:\GitHub\example.js:3:28)
From previous event:
at Object.<anonymous> (C:\GitHub\example.js:4:4)
From previous event:
at Object.<anonymous> (C:\GitHub\example.js:5:4)
```

Now, it is more clear that the error did occur exactly at line 6 with throwing an error and only shows the user code in this point. This is very helpful for debugging, as otherwise you end up getting only the first line, plus a bunch of RxJS internals, with no sign of where the operation started.

This feature does come with a serious performance and memory overhead, however, If you're working with lots of RxJS code, or trying to scale a server to many users, you should probably keep it off. In development, this is perfectly fine for finding those pesky errors!

In a future release, we may also release support for a node.js environment variable so that you can set it and unset it fairly easily.

See Also

Concepts

- [Creating and Subscribing to Simple Observable Sequences](#)
- [Querying Observable Sequences](#)
- [Using Schedulers](#)

Implementing Your Own Observable Operators

You can extend RxJS by adding new operators for operations that are not provided by the base library, or by creating your own implementation of standard query operators to improve readability and performance. Writing a customized version of a standard operator is useful when you want to operate with in-memory objects and when the intended customization does not require a comprehensive view of the query.

Creating New Operators

RxJS offers a full set of operators that cover most of the possible operations on a set of entities. However, you might need an operator to add a particular semantic meaning to your query�especially if you can reuse that same operator several times in your code. Adding new operators to RxJS is a way to extend its capabilities. However, you can also improve code readability by wrapping existing operators into more specialized and meaningful ones.

For example, let's see how we might implement the _.where method from Lo-Dash or Underscore, which takes a set of attributes and does a deep comparison for equality. We might try implementing this from scratch using the `Rx.Observable.createWithDisposable` method such as the following code.

```
Rx.Observable.prototype.whereProperties = properties => {
    var source = this,
        comparer = Rx.internals.isEqual;

    return Rx.Observable.filterByProperties(observer => {
        // Our disposable is the subscription from the parent
        return source.subscribe(
            data => {

                try {
                    var shouldRun = true;

                    // Iterate the properties for deep equality
                    for (var prop in properties) {
                        if (!comparer(properties[prop], data[prop])) {
                            shouldRun = false;
                            break;
                        }
                    }
                } catch (e) {
                    observer.onError(e);
                }

                if (shouldRun) {
                    observer.onNext(data);
                }
            },
            observer.onError.bind(observer),
            observer.onCompleted.bind(observer)
        );
    });
};
```

Many existing operators, such as this, instead could be built using other basic operators for example in this case, `filter` or `where` . In fact, many existing operators are built using other basic operators. For example, the `flatMap` or `selectMany` operator is built by composing the `map` or `select` and `mergeObservable` operators, as the following code shows.

```
Rx.Observable.prototype.flatMap = selector => this.map(selector).mergeObservable();
```

We could rewrite it as the following to take advantage of already built in operators.

```
Rx.Observable.prototype.filterByProperties = properties => {
    var comparer = Rx.internals.isEqual;

    return this.filter(data => {

        // Iterate the properties for deep equality
        for (var prop in properties) {
            if (!comparer(properties[prop], data[prop])) {
                return false;
            }
        }

        return true;
    });
};
```

By reusing existing operators when you build a new one, you can take advantage of the existing performance or exception handling capabilities implemented in the RxJS libraries. When writing a custom operator, it is good practice not to leave any disposables unused; otherwise, you may find that resources could actually be leaked and cancellation may not work correctly.

Testing Your New Operator

Just because you have implemented a new operator doesn't mean you are finished. Now, let's write a test to verify its behavior from what we learned in the Testing and Debugging topic. We'll reuse the `collectionAssert.assertEqual` from the previous topic so it is not repeated here.

```javascript
var onNext = Rx.ReactiveTest.onNext,
    onCompleted = Rx.ReactiveTest.onCompleted,
    subscribe = Rx.ReactiveTest.subscribe;

test('filterProperties should yield with match', () => {

    var scheduler = new Rx.TestScheduler();

    var input = scheduler.createHotObservable(
        onNext(210, { 'name': 'curly', 'age': 30, 'quotes': ['Oh, a wise guy, eh?', 'Poif
        onNext(220, { 'name': 'moe', 'age': 40, 'quotes': ['Spread out!', 'You knucklehea
        onCompleted(230)
    );

    var results = scheduler.startWithCreate(
        () => input.filterByProperties({ 'age': 40 })
    );

    collectionAssert.assertEqual(results.messages, [
        onNext(220, { 'name': 'moe', 'age': 40, 'quotes': ['Spread out!', 'You knucklehea
        onCompleted(230)
    ]);

    collectionAssert.assertEqual(input.subscriptions, [
        subscribe(200, 230)
    ]);
});
```

In order for this to be successfully tested, we should check for when there is no data, empty, single matches, multiple matches and so forth.

See Also

Resources

- Testing and Debugging

How Do I...?

- How do I wrap an existing API?
- How do I integrate jQuery with RxJS?
- How do I integrate Angular.js with RxJS?
- How do I create a simple event emitter?

Wrap an Existing API with RxJS

One question that often comes up is how can I wrap an existing API into an Observable sequence? The answer is fairly simple and not a lot of lines of code to make that happen.

To make this a bit more concrete, let's take a familiar HTML5 API like Geolocation API, in particular, the `navigator.geolocation.watchPosition` method.

The typical use of this method might be the following where we would hook up an event handler to listen for success and errors on watching the geolocation by using the `navigator.geolocation.watchPosition` method. When one wishes to terminate listening for geolocation updates, you simply call the `navigator.geolocation.clearWatch` method passing in the watch ID returned from the `watchPosition` method.

```javascript
function watchPositionChanged(e) {
    // Do something with the coordinates
}

function watchPositionError(e) {
    // Handle position error
}

var watchId = navigator.geolocation.watchPosition(
    watchPositionChanged,
    watchPositionError);

var stopWatching = document.querySelector('#stopWatching');
stopWatching.addEventListener('click', stopWatchingClicked, false);

// Clear watching upon click
function stopWatchingClicked(e) {
    navigator.geolocation.clearWatch(watchId)
}
```

In order to wrap this, we'll need to use the `Rx.Observable.create` method. From this, we can yield values to the observer or handle the errors. Let's see how the code might look, creating a watchPosition method which takes geolocation options.

```
function watchPosition(geolocationOptions) {
    return Rx.Observable.create(observer => {
        var watchId = window.navigator.geolocation.watchPosition(
            function successHandler (loc) {
                observer.onNext(loc);
            },
            function errorHandler (err) {
                observer.onError(err);
            },
            geolocationOptions);

        return () => {
            window.navigator.geolocation.clearWatch(watchId);
        };
    });
}
```

We need to also be aware of ensuring we're not adding too many watchPosition calls as we compose it together with other observable sequences. To do that, we'll need to utilize the `publish` and `refCount` methods from rx.binding.js.

Our final result should look like the following:

```
function watchPosition(geolocationOptions) {
    return Rx.Observable.create(observer => {
        var watchId = window.navigator.geolocation.watchPosition(
            function successHandler (loc) {
                observer.onNext(loc);
            },
            function errorHandler (err) {
                observer.onError(err);
            },
            geolocationOptions);

        return () => {
            window.navigator.geolocation.clearWatch(watchId);
        };
    }).publish().refCount();
}
```

And now we can consume the geolocation such as:

```
var source = watchPosition();

var subscription = source.subscribe(
    position => console.log(`Next: ${position.coords.latitude}, ${position.coords.longitu
    err => {
        var message = '';
        switch (err.code) {
            case err.PERMISSION_DENIED:
                message = 'Permission denied';
                break;
            case err.POSITION_UNAVAILABLE:
                message = 'Position unavailable';
                break;
            case err.PERMISSION_DENIED_TIMEOUT:
                message = 'Position timeout';
                break;
        }
        console.log('Error: ' + message);
    },
    () => console.log('Completed')
);
```

Together

How do I work with jQuery and RxJS

The jQuery project and RxJS play very well together as libraries. In fact, we supply bindings directly for RxJS to jQuery should you want to wrap animations, events, Ajax calls and more using Observables in RxJS-jQuery. The bindings library provides many handy features for bridging the world to Observables. If you're interested in that library, go ahead and use it.

Using RxJS with Rx-jQuery

Getting started with the bindings is easy. Each method is enumerated on the main page from the jQuery method to its RxJS counterpart.

```html
<div id="results"></div>
<script src="http://code.jquery.com/jquery-1.9.1.js"></script>
<script src="rx.js"></script>
<script src="rx.binding.js"></script>
<script src="rx.jquery.js"></script>
```

Now we can start using the bindings! For example, we can listen to a `click` event and then by using `flatMap` or `selectMap` we can animate by calling `animateAsObservable`. Finally, we can subscribe to cause the side effect and nothing more.

```js
$( "#go" ).clickAsObservable().flatMap(() => {

    return $( "#block" ).animateAsObservable({
        width: "70%",
        opacity: 0.4,
        marginLeft: "0.6in",
        fontSize: "3em",
        borderWidth: "10px"
    }, 1500 );
}).subscribe();
```

Using RxJS with jQuery

Let's start though by assuming you just have RxJS and wanted to get started with jQuery without the bridge library. There is already plenty you can do without even needing a bridge library with the support built in for events and promises.

Binding to an event

Using RxJS with jQuery to bind to an event using plain old RxJS is easy. For example, we could bind to the `mousemove` event from the DOM document easily.

First, we'll reference the files we need.

```
<div id="results"></div>
<script src="http://code.jquery.com/jquery-1.9.1.js"></script>
<script src="rx.js"></script>
<script src="rx.async.js"></script>
<script src="rx.binding.js"></script>
```

```
var observable = Rx.Observable.fromEvent(
    $(document),
    'mousemove');

var subscription = observable.subscribe(e => $('#results').text(e.clientX + ',' + e.clien
```

We could go a step further and create our own jQuery plugin which handles events with ease.

```
/**
 * Creates an observable sequence by adding an event listener to the matching jQuery elem
 *
 * @param {String} eventName The event name to attach the observable sequence.
 * @param {Function} [selector] A selector which takes the arguments from the event handl
 * @returns {Observable} An observable sequence of events from the specified element and
 */
jQuery.fn.toObservable = function (eventName, selector) {
    return Rx.Observable.fromEvent(this, eventName, selector);
};
```

Now we could rewrite our above example such as this.

```
var observable = $(document).toObservable('mousemove');

var subscription = observable.subscribe(e => $('#results').text(e.clientX + ',' + e.clien
```

Using RxJS with Ajax calls

Bridging to jQuery Ajax calls using `$.ajax` is easy as well with the built-in [Promises A+](#) support. Since jQuery 1.5, the `$.ajax` method has implemented a promise interface (even if not 100% pure) which allows us to bridge to an observable sequence via the `Rx.Observable.fromPromise` method.

For example, we could query Wikipedia by calling the `$.ajax` method and then calling the `promise` method which then exposes the minimum promise interface needed.

```javascript
function searchWikipedia (term) {
    var promise = $.ajax({
        url: 'http://en.wikipedia.org/w/api.php',
        dataType: 'jsonp',
        data: {
            action: 'opensearch',
            format: 'json',
            search: encodeURI(term)
        }
    }).promise();
    return Rx.Observable.fromPromise(promise);
}
```

Once we created the wrapper, we can query the service by getting the text and then using `flatMapLatest` to ensure we have no out of order results.

```javascript
$('#input').toObservable('keyup')
    .map(e => e.target.value;)
    .flatMapLatest(searchWikipedia)
    .subscribe(data => {

        var results = data[1];

        $.each(results, (_, result) => {
            // Do something with each result
        });

    });
```

Using RxJS with Callbacks to Handle Simple Animations

RxJS can also be used to bind to simple callbacks, such as the `.animate()` method. We can use `Rx.Observable.fromCallback` to supply the required arguments with the last argument is to be the callback. In this example, we'll take the animation example from above and use nothing but core RxJS to accomplish the same thing.

You'll note that we need a notion of `this` for the `block.animate` to properly work, so we have two choices, either use `Function.prototype.bind` available in most modern browsers...

```
var animate = Rx.Observable.fromCallback(block.animate.bind(block));
```

Or we can supply an optional argument which supplies the context to the callback such as the following...

```
var animate = Rx.Observable.fromCallback(
    block.animate,
    null, /* default scheduler used */
    block /* context */);
```

When viewed in its entirety, it will look like this where we call `flatMap` or `selectMany` to compose together two observable sequences. We then bind to the `animate` function through `Rx.Observable.fromCallback` and then return the observable which results from the function execution. Our `subscribe` does nothing in this case as there is nothing to print or do, and is simply a side effect.

```
var block = $('#block');

$('#go').toObservable('click').flatMap(() => {
    var animate = Rx.Observable.fromCallback(block.animate.bind(block));

    return animate({
        width: "70%",
        opacity: 0.4,
        marginLeft: "0.6in",
        fontSize: "3em",
        borderWidth: "10px"
    }, 1500);
}).subscribe();
```

How do I integrate Angular.js with RxJS?

AngularJS is a popular MV* framework for JavaScript which covers things such as data binding, controllers as well as things such as dependency injection. The Reactive Extensions for JavaScript plays well with this framework, and in fact has a dedicated library for interop called rx.angular.js. However, if you don't wish to use that, here are some simple ways you can integrate the two together.

Integration with Scopes

The `scope` is an object that refers to the application model. It is an execution context for expressions. Scopes are arranged in hierarchical structure which mimic the DOM structure of the application. Scopes can watch expressions and propagate events.

Scopes provide the ability to observe change mutations on the scope through the `$watch` method. This is a perfect opportunity to integrate the power of the Reactive Extensions for JavaScript with Angular. Let's look at a typical usage of `$watch`.

```javascript
// Get the scope from somewhere
var scope = $rootScope;
scope.name = 'Reactive Extensions';
scope.counter = 0;

scope.$watch('name', (newValue, oldValue) => {
  scope.counter = scope.counter + 1;
  scope.oldValue = oldValue;
  scope.newValue = newValue;
});

scope.name = 'RxJS';

// Process All the Watchers
scope.$digest();

// See the counter increment
console.log(counter);
// => 1
```

Using the Reactive Extensions for JavaScript, we're able to easily bind to this by wrapping the `$watch` as an observable. To do this, we'll create an observable sequence using `Rx.Observable.create` which gives us an observer to yield to. In this case, we'll capture both

the old and new values through our listener function. The `$watch` function returns a function, which when called, ceases the watch expression.

```
Rx.Observable.$watch = (scope, watchExpression, objectEquality) => {
    return Rx.Observable.create(observer => {
        // Create function to handle old and new Value
        function listener (newValue, oldValue) {
            observer.onNext({ oldValue: oldValue, newValue: newValue });
        }

        // Returns function which disconnects the $watch expression
        return scope.$watch(watchExpression, listener, objectEquality);
    });
};
```

Now that we have this, we're able to now take the above example and now add some RxJS goodness to it.

```
// Get the scope from somewhere
var scope = $rootScope;
scope.name = 'Reactive Extensions';
scope.isLoading = false;
scope.data = [];

// Watch for name change and throttle it for 1 second and then query a service
Rx.Observable.$watch(scope, 'name')
    .throttle(1000)
    .map(e => e.newValue)
    .do(() => {
        // Set loading and reset data
        scope.isLoading = true;
        scope.data = [];
    })
    .flatMapLatest(querySomeService)
    .subscribe(data => {

        // Set the data
        scope.isLoading = false;
        scope.data = data;
    });
```

Integration with Deferred/Promise Objects

AngularJS ships a promise/deferred implementation based upon Kris Kowal's Q called the `$q` service. Promises are quite useful in scenarios with one and done asynchronous operations such as querying a service through the `$http` service.

```
$http.get('/someUrl')
    .then(successCallback, errCallback);
```

Using the Reactive Extensions for JavaScript, we can also integrate using the
`Rx.Observable.fromPromise` bridge available in RxJS version 2.2+. We simply

```
// Query data
var observable = Rx.Observable.fromPromise(
    $http(
        method: 'GET',
        url: 'someurl',
        params: { searchString: $scope.searchString }
    )
);

// Subscribe to data and update UI
observable.subscribe(
    data => $scope.data = data,
    err => $scope.error = err.message
);
```

These are just only the beginnings of what you can do with the Reactive Extensions for
JavaScript and AngularJS.

How do I create a custom event emitter?

Publish/Subscribe is a common pattern within JavaScript applications. The idea is that you have a publisher that emits events and you have consumers which register their interest in a given event. Typically you may see something like the following where you listen for a 'data' event and then the event emitter publishes data to it.

```javascript
var emitter = new Emitter();

function logData(data) {
    console.log('data: ' + data);
}

emitter.on('data', logData);

emitter.emit('data', 'foo');
// => data: foo

// Destroy handler
emitter.off('data', logData);
```

How might one implement this using the Reactive Extensions for JavaScript? Using an `Rx.Subject` will solve this problem easily. As you may remember, an `Rx.Subject` is both an Observer and Observable, so it handles both publish and subscribe.

```javascript
var subject = new Rx.Subject();

var subscription = subject.subscribe(data => console.log(`data: ${data}`));

subject.onNext('foo');
// => data: foo
```

Now that we have a basic understanding of publish and subscribe through `onNext` and `subscribe`, let's put it to work to handle multiple types of events at once. First, we'll create an Emitter class which has three main methods, `emit`, `on` and `off` which allows you to emit an event, listen to an event and stop listening to an event.

```
var hasOwnProp = {}.hasOwnProperty;

function createName (name) {
    return `$ ${name}`;
}

function Emitter() {
    this.subjects = {};
}

Emitter.prototype.emit = (name, data) => {
    var fnName = createName(name);
    this.subjects[fnName] || (this.subjects[fnName] = new Rx.Subject());
    this.subjects[fnName].onNext(data);
};

Emitter.prototype.on = (name, handler) => {
    var fnName = createName(name);
    this.subjects[fnName] || (this.subjects[fnName] = new Rx.Subject());
    this.subjects[fnName].subscribe(handler);
};

Emitter.prototype.off = (name, handler) => {
    var fnName = createName(name);
    if (this.subjects[fnName]) {
        this.subjects[fnName].dispose();
        delete this.subjects[fnName];
    }
};

Emitter.prototype.dispose = () => {
    var subjects = this.subjects;
    for (var prop in subjects) {
        if (hasOwnProp.call(subjects, prop)) {
            subjects[prop].dispose();
        }
    }

    this.subjects = {};
};
```

Then we can use it much as we did above. As the call to `subscribe` returns a subscription, we might want to hand that back to the user instead of providing an off method. So, we could rewrite the above where we call the `on` method to `listen` and we return a subscription handle to the user to stop listening.

```
var hasOwnProp = {}.hasOwnProperty;

function createName (name) {
    return `$ ${name}`;
}

function Emitter() {
    this.subjects = {};
}

Emitter.prototype.emit = (name, data) => {
    var fnName = createName(name);
    this.subjects[fnName] || (this.subjects[fnName] = new Rx.Subject());
    this.subjects[fnName].onNext(data);
};

Emitter.prototype.listen = (name, handler) => {
    var fnName = createName(name);
    this.subjects[fnName] || (this.subjects[fnName] = new Rx.Subject());
    return this.subjects[fnName].subscribe(handler);
};

Emitter.prototype.dispose = () => {
    var subjects = this.subjects;
    for (var prop in subjects) {
        if (hasOwnProp.call(subjects, prop)) {
            subjects[prop].dispose();
        }
    }

    this.subjects = {};
};
```

Now we can use this to rewrite our example such as the following:

```
var emitter = new Emitter();

var subcription = emitter.listen('data', data => console.log(`data: ${data}`);

emitter.emit('data', 'foo');
// => data: foo

// Destroy the subscription
subscription.dispose();
```

Mapping RxJS from Different Libraries

- For Bacon.js Users
- For Async.js Users

RxJS for Bacon.js Users

[Bacon.js](#) is a popular Functional Reactive Programming (FRP) library which was inspired by RxJS, ReactiveBanana among other libraries. If you're not familiar with FRP, Conal Elliott summed it up nicely on [StackOverflow](#) so no need to repeat that here.

Bacon.js has two main concepts, Event Streams and Properties, which we will map to RxJS concepts.

Contents

- [Event Streams](#)

Common API Methods

Event Streams

In Bacon.js (and RxJS for that matter), an EventStream represents a stream of events. It is an Observable object, meaning that you can listen to events in the stream using, for instance, the onValue method with a callback.

Creating Event Streams

Bacon.js

Because Bacon.js is optimized for jQuery and Zepto, you can use the `$.fn.asEventStream` method to easily bind to create event streams.

For example we can get the clickable element, listen to the `click` event, and then we can subscribe via the `onValue` method to capture the clicks.

```
var clickable = $('#clickable').asEventStream('click');

clickable.onValue(e => console.log('clicked!'));
```

The support goes above just standard support, but also selectors and an optional argument selector which transforms the arguments of the event to a single object.

```
$("#my-div").asEventStream("click", ".more-specific-selector")

$("#my-div").asEventStream("click", ".more-specific-selector", (event, args) => args[0]);

$("#my-div").asEventStream("click", (event, args) => args[0]);
```

RxJS

It's very similar in RxJS core. Until recently, this feature was reserved for external libraries such as RxJS-jQuery, RxJS-DOM, RxJS-Dojo and RxJS-MooTools. RxJS 2.2 introduced two ways to bind to events with `fromEvent` and `fromEventPattern` so that bridge libraries are strictly not as necessary as they used to be.

For example, we can recreate the binding to the clickable element for the `click` event, and then call `subscribe` with a function which listens for each time the clickable is clicked.

```
var clickable = $('#clickable');

var clickableObservable = Rx.Observable.fromEvent(clickable, 'click')
    .subscribe(() => console.log('clicked!'));
```

In addition, RxJS also supports for event argument transformers for additional data. For example, if a Node.js `EventEmitter` emits more than one piece of data at a time, you can still capture it.

```
var Rx = require('rx'),
    EventEmitter = require('events').EventEmitter;

var e = new EventEmitter();

Rx.Observable.fromEvent(e, 'data', args => ({ first: args[0], second: args[1] }))
    .subscribe(data => console.log(`${data.first}, ${data.second}`));

e.emit('data', 'foo', 'bar');
// => foo,bar
```

Querying Streams

Event Streams support higher ordered functions much as RxJS does such as `map`, `filter` and more, although supports a more Underscore/Lo-Dash style than the callback selector style found in RxJS.

```
var plus = $("#plus").asEventStream("click").map(1);
var minus = $("#minus").asEventStream("click").map(-1);

// Combine both into one
var both = plus.merge(minus);

both.onValue (x => { /* returns 1 or -1 */ });
```

RxJS for Async.js Users

Async.js is a popular utility module which provides straight-forward, powerful functions for working with asynchronous JavaScript. Async provides around 20 functions that include the usual 'functional' suspects (map, reduce, filter, each...) as well as some common patterns for asynchronous control flow (parallel, series, waterfall...). All these functions assume you follow the node.js convention of providing a single callback as the last argument of your async function.

Many of these concepts in the library map directly to RxJS concepts. We'll go operator by operator on how each map to existing functionality in RxJS.

Collection Methods

- `async.each`
- `async.map`
- `async.filter`
- `async.reject`
- `async.reduce`
- `async.detect`
- `async.some`
- `async.every`
- `async.concat`

Control Flow

- `async.series`
- `async.parallel`
- `async.whilst`
- `async.doWhilst`
- `async.nextTick`
- `async.waterfall`
- `async.compose`

`async.each`

The `async.each` method applies an iterator function to each item in an array, in parallel. The iterator is called with an item from the list and a callback for when it has finished. If the iterator passes an error to this callback, the main callback for the each function is immediately called with the error.

async version

In this example, we will use `async.each` to iterate an array of files to write some contents and save.

```javascript
var async = require('async'),
    fs = require('fs');

var files = ['file1.txt', 'file2.txt', 'file3.txt'];

function saveFile (file, cb) {
    fs.writeFile(file, 'Hello Node', err => cb(err));
}

async.each(files, saveFile, err => {
    // if any of the saves produced an error, err would equal that error
});
```

RxJS version

Using RxJS, you can accomplish this task in a number of ways by using `Rx.Observable.fromNodeCallback` to wrap the `fs.writeFile` function, and then iterate the files by using the `Rx.Observable.for` method.

```javascript
var Rx = require('rx'),
    fs = require('fs');

var files = ['file1.txt', 'file2.txt', 'file3.txt'];

// wrap the method
var writeFile = Rx.Observable.fromNodeCallback(fs.writeFile);

Rx.Observable
    .for(files, file => writeFile(file, 'Hello Node'))
    .subscribe(
        () => console.log('file written!'),
        err => console.log(`err ${err}`),
        () => console.log('completed!')
    );
```

`async.map`

The `async.map` method produces a new array of values by mapping each value in the given array through the iterator function. The iterator is called with an item from the array and a callback for when it has finished processing. The callback takes 2 arguments, an error and the transformed item from the array. If the iterator passes an error to this callback, the main callback for the map function is immediately called with the error.

async version

In this example, we'll get the `fs.stat` for each file given and have the results returned as an array.

```javascript
var async = require('async'),
    fs = require('fs');

var files = ['file1.txt', 'file2.txt', 'file3.txt'];

async.map(files, fs.stat, (err, results) => results.forEach(result => console.log(`is fil
```

RxJS version

Using RxJS, we can achieve the same results of an array of all of our values by wrapping the `fs.stat` method again using our `Rx.Observable.fromNodeCallback`, then iterate using the `Rx.Observable.for` method, and finally, calling `.toArray()` to get our results as an entire array.

```javascript
var Rx = require('rx'),
    fs = require('fs');

var stat = Rx.Observable.fromNodeCallback(fs.stat);

var files = ['file1.txt', 'file2.txt', 'file3.txt'];

Rx.Observable
    .for(files, stat)
    .toArray()
    .subscribe(
        results =>
            results.forEach(result => console.log(`is file: ${result.isFile()}`)),
        err => console.log(`err: ${err}`)
    );
```

`async.filter`

The `async.filter` method returns a new array of all the values which pass an async truth test. The callback for each iterator call only accepts a single argument of true or false, it does not accept an error argument first! This is in-line with the way node libraries work with truth tests like fs.exists.

async version

In this example, we'll determine whether the file exists by calling `fs.exists` for each file given and have the results returned as an array.

```
var async = require('async'),
    fs = require('fs');

var files = ['file1.txt', 'file2.txt', 'file3.txt'];

async.filter(files, fs.exists, (err, results) => {
    results.forEach(result => console.log(`exists: ${result}`));
});
```

RxJS version

Using RxJS, we can achieve the same results of an array of all of our values by wrapping the `fs.exists` method using our `Rx.Observable.fromCallback` as it only has one result, a `true` or `false` value instead of the usual callback with error and result. Then we'll iterate using the `Rx.Observable.for` method, filter the existing files and finally, calling `.toArray()` to get our results as an entire array.

```
var Rx = require('rx'),
    fs = require('fs');

var exists = Rx.Observable.fromCallback(fs.exists);

Rx.Observable
    .for(files, exists)
    .where(x => x)
    .toArray()
    .subscribe(
        results =>
            results.forEach(result => console.log(`exists: ${result}`));
    );
```

`async.reject`

The `async.reject` method is the opposite of filter. Removes values that pass an async truth test.

async version

In this example, we'll determine whether the file exists by calling `fs.exists` for each file given and have the results returned as an array.

```
var async = require('async'),
    fs = require('fs');

var files = ['file1.txt', 'file2.txt', 'file3.txt'];

async.reject(files, fs.exists, (err, results) => {
    results.forEach(result => console.log(`exists: ${result}`));
});
```

RxJS version

Using RxJS, we can achieve the same results of an array of all of our values by wrapping the `fs.exists` method using our `Rx.Observable.fromCallback` as it only has one result, a `true` or `false` value instead of the usual callback with error and result. Then we'll iterate using the `Rx.Observable.for` method, filter the existing files using `filter` and finally, calling `.toArray()` to get our results as an entire array.

```
var Rx = require('rx'),
    fs = require('fs');

var exists = Rx.Observable.fromCallback(fs.exists);

Rx.Observable
    .for(files, exists)
    .where(x => !x)
    .toArray()
    .subscribe(
        results =>
            results.forEach(result => console.log(`exists: ${result}`));
        }
    );
```

`async.reduce`

The `async.reduce` method reduces a list of values into a single value using an async iterator to return each successive step. Memo is the initial state of the reduction. This function only operates in series. For performance reasons, it may make sense to split a call to this function into a parallel map, then use the normal `Array.prototype.reduce` on the results. This function is for situations where each step in the reduction needs to be async, if you can get the data before reducing it then it's probably a good idea to do so.

async version

In this example, we'll determine whether the file exists by calling `fs.exists` for each file given and have the results returned as an array.

```javascript
var async = require('async'),
    fs = require('fs');

function reduction (acc, x, cb) {
    process.nextTick(() => cb(null, acc + x));
}

async.reduce([1,2,3], 0, fs.reduction, (err, results) => console.log(results));

// => 6
```

RxJS version

In RxJS, we have a number of ways of doing this including using `Rx.Observable.fromArray` to turn an array into observable sequence, then we can call `reduce` to add the numbers. To ensure that it is indeed async, we can switch to the `Rx.Scheduler.timeout` to ensure that it is done via a callback.

```javascript
var Rx = require('rx'),
    fs = require('fs');

Rx.Observable
    .fromArray([1,2,3], Rx.Scheduler.timeout)
    .reduce((acc, x) => acc + x, 0)
    .subscribe(console.log.bind(console));
// => 6
```

`async.detect`

The `async.detect` method returns the first value in a list that passes an async truth test. The iterator is applied in parallel, meaning the first iterator to return true will fire the detect callback with that result.

async version

In this example, we'll get the first file that matches.

```
var async = require('async'),
    fs = require('fs');

var files = ['file1','file2','file3'];

async.detect(files, fs.exists, result => {
    // result now equals the first file in the list that exists
});
```

RxJS version

In RxJS, we can iterate over the files as above using `Rx.Observable.for` and then calling `first` to get the first matching file project forward the file name and whether the file exists.

```
var Rx = require('rx'),
    fs = require('fs');

var files = ['file1','file2','file3'];

var exists = Rx.Observable.fromCallback(fs.exists);

Rx.Observable
    .for(files, file => ({ file: file, exists: exists(file) }))
    .first(x => x.exists)
    .subscribe(
        result => {
            // result now equals the first file in the list that exists
        });
```

`async.some`

The `async.some` method returns `true` if at least one element in the array satisfies an async test. The callback for each iterator call only accepts a single argument of true or false, it does not accept an error argument first! This is in-line with the way node libraries work with truth tests like fs.exists. Once any iterator call returns true, the main callback is immediately called.

async version

In this example, we'll determine whether the file exists by calling `fs.exists` for each file given and have the results returned as an array.

```javascript
var async = require('async'),
    fs = require('fs');

var files = ['file1.txt', 'file2.txt', 'file3.txt'];

async.some(files, fs.exists, result => {
    // if result is true then at least one of the files exists
});
```

RxJS version

Using RxJS, we can achieve the same results of an array of all of our values by wrapping the `fs.exists` method using our `Rx.Observable.fromCallback` as it only has one result, a `true` or `false` value instead of the usual callback with error and result. Then we'll iterate using the `Rx.Observable.for` method, then call `some` to determine whether any match.

```javascript
var Rx = require('rx'),
    fs = require('fs');

var exists = Rx.Observable.fromCallback(fs.exists);

Rx.Observable
    .for(files, exists)
    .some()
    .subscribe(
        results => {
            // if result is true then at least one of the files exists
        });
```

`async.every`

The `async.every` method returns `true` if every element in the array satisfies an async test. The callback for each iterator call only accepts a single argument of true or false, it does not accept an error argument first! This is in-line with the way node libraries work with truth tests like fs.exists.

async version

In this example, we'll determine whether the file exists by calling `fs.exists` for each file given and have the results returned as an array.

```javascript
var async = require('async'),
    fs = require('fs');

var files = ['file1.txt', 'file2.txt', 'file3.txt'];

async.every(files, fs.exists, result => {
    // if result is true then every file exists
});
```

RxJS version

Using RxJS, we can achieve the same results of an array of all of our values by wrapping the `fs.exists` method using our `Rx.Observable.fromCallback` as it only has one result, a `true` or `false` value instead of the usual callback with error and result. Then we'll iterate using the `Rx.Observable.for` method, then call `every` to determine whether all match.

```javascript
var Rx = require('rx'),
    fs = require('fs');

var files = ['file1.txt', 'file2.txt', 'file3.txt'];

var exists = Rx.Observable.fromCallback(fs.exists);

Rx.Observable
    .for(files, exists)
    .every()
    .subscribe(
        results => {
            // if result is true then every file exists
        });
```

`async.concat`

The `async.concat` method applies an iterator to each item in a list, concatenating the results. Returns the concatenated list. The iterators are called in parallel, and the results are concatenated as they return.

async version

In this example, we'll determine whether the file exists by calling `fs.exists` for each file given and have the results returned as an array.

```js
var async = require('async'),
    fs = require('fs');

var directories = ['dir1', 'dir2', 'dir3'];

async.concat(files, fs.readdir, (err, files) => {
    // files is now a list of filenames that exist in the 3 directories
});
```

RxJS version

Using RxJS, we can achieve the same results of an array of all of our values by wrapping the `fs.readdir` method using our `Rx.Observable.fromNodeCallback`. Then we'll iterate using the `Rx.Observable.for` method, then call `reduce` to add each item to the item to the overall list.

```js
var Rx = require('rx'),
    fs = require('fs');

var readdir = Rx.Observable.fromNodeCallback(fs.readdir);

Rx.Observable
    .for(files, readdir)
    .reduce((acc, x) => { acc.push(x); return acc; }, [])
    .subscribe(
        files => {
            // files is now a list of filenames that exist in the 3 directories
        },
        err => {
            // handle error
        });
```

async.series

The `async.series` runs an array of functions in series, each one running once the previous function has completed. If any functions in the series pass an error to its callback, no more functions are run and the callback for the series is immediately called with the value of the error. Once the tasks have completed, the results are passed to the final callback as an array.

It is also possible to use an object instead of an array. Each property will be run as a function and the results will be passed to the final callback as an object instead of an array. This can be a more readable way of handling results from async.series.

async version

In this example we'll run some examples with both an array or an object.

```javascript
var async = require('async');

async.series([
    callback => setTimeout(() => callback(null, 'one'), 200),
    callback => setTimeout(() => callback(null, 'two'), 100)
],
// optional callback
(err, results) => {
    // results is now equal to ['one', 'two']
});


// an example using an object instead of an array
async.series({
    one: callback => setTimeout(() => callback(null, 1), 200),
    two: callback => setTimeout(() => callback(null, 2), 100)
},
(err, results) => {
    // results is now equal to: {one: 1, two: 2}
});
```

RxJS version

We can achieve the same functionality of `async.series` with an array by simply calling fromArray and calling `flatMap` to give us the observable of the current. Then we'll call `reduce` to add each item to a new array to return.

```
var Rx = require('rx');

function wrapArray (items) {
    return Rx.Observable
        .fromArray(items)
        .flatMap(x => x)
        .reduce((acc, x) => {
            var arr = acc.slice(0);
            arr.push(x);
            return arr;
        }, []);
}

wrapArray([
        Rx.Observable.return('one'),
        Rx.Observable.return('two')
    ])
    .subscribe(
        console.log.bind(console),
        err => console.log(`Error: ${err}`)
    );

// => ['one', 'two']
```

Using an object literal can also be achieved with a little bit more work, but totally reasonable. Instead of just returning the observable in `flatMap`, we'll add a property to a new object which will contain our key moving forward. Then, we'll call `reduce` much as before, copying the values to a new object, and then plucking the value from each time it comes through and adding it to our final object.

```
var Rx = require('rx');

function wrapObject (obj) {
    var keys = Object.keys(obj),
        hasOwnProperty = {}.hasOwnProperty;

    return Rx.Observable
        .fromArray(keys)
        .flatMap(key => {

            return obj[key].map(x => {
                var newObj = {};
                newObj[key] = x;
                return newObj;
            });
        })
        .reduce((acc, x) => {
            var newObj = {};
            for (var prop in acc) {
                if(!hasOwnProperty.call(acc)) {
                    newObj[prop] = acc[prop];
                }
            }

            var xKey = Object.keys(x)[0];
            newObj[xKey] = x[xKey];

            return newObj;
        }, {});
}

wrapObject({
        one: Rx.Observable.return(1),
        two: Rx.Observable.return(2)
    })
    .subscribe(
        console.log.bind(console),
        err => console.log(`Error: ${err}`)
    );

// => { one: 1, two: 2 }
```

`async.parallel`

The `async.parallel` runs an array of functions in parallel, without waiting until the previous
function has completed. If any of the functions pass an error to its callback, the main
callback is immediately called with the value of the error. Once the tasks have completed,

the results are passed to the final callback as an array.

It is also possible to use an object instead of an array. Each property will be run as a function and the results will be passed to the final callback as an object instead of an array. This can be a more readable way of handling results from async.parallel.

async version

In this example we'll run some examples with both an array or an object.

```javascript
var async = require('async');

async.parallel([
    callback => setTimeout(() => callback(null, 'one'), 200),
    callback => setTimeout(() => callback(null, 'two'), 100)
],

// optional callback
(err, results) => {
    // the results array will equal ['one','two'] even though
    // the second function had a shorter timeout.
});



// an example using an object instead of an array
async.parallel({
    one: callback => setTimeout(() => callback(null, 1), 200),
    two: callback => setTimeout(() => callback(null, 2), 100)
},
(err, results) => {
    // results is now equals to: {one: 1, two: 2}
});
```

RxJS version

We can achieve the same functionality of `async.series` with an array by calling `Rx.Observable.forkJoin` with our array of observable sequences. This returns the last value from each sequence in "parallel".

```
var Rx = require('rx');

function wrapArrayParallel (items) {
    return Rx.Observable.forkJoin.apply(null, items);
}

wrapArrayParallel([
        Rx.Observable.return('one'),
        Rx.Observable.return('two')
    ])
    .subscribe(
        console.log.bind(console),
        err => console.log(`Error: ${err}`)
    );

// => ['one', 'two']
```

Using an object literal can also be achieved with a little bit more work, but totally reasonable. Instead of simply calling `forkJoin`, we first need to extract the observable sequences by calling `map` on the keys we obtained by `Object.keys`. Because the order of observable sequences is deterministic, we can then call `map` to transform the array into an object, by calling `reduce` on the array, turning the array into an object with the appropriate keys.

```
var Rx = require('rx');

function wrapObjectParallel (obj) {
    var keys = Object.keys(obj);
    var mapped = keys.map(key => obj[key]);

    return Rx.Observable.forkJoin.apply(null, mapped)
        .map(arr => {
            var idx = 0;
            return arr.reduce((acc, x) => {
                var key = keys[idx++];

                var newObj = {};
                for (var prop in acc) {
                    if(!hasOwnProperty.call(acc)) {
                        newObj[prop] = acc[prop];
                    }
                }

                newObj[key] = x;

                return newObj;
            }, {})
        });
}

wrapObjectParallel({
        one: Rx.Observable.return(1),
        two: Rx.Observable.return(2)
    })
    .subscribe(
        console.log.bind(console),
        err => console.log(`Error: ${err}`)
    );

// => { one: 1, two: 2 }
```

async.whilst

The `async.whilst` method repeatedly call function, while test returns true. Calls the callback when stopped, or an error occurs.

async version

In this example we'll just run a keep calling the callback while the count is less than 5.

```
var async = require('async');

var count = 0;

async.whilst(
    () => count < 5,
    callback => {
        count++;
        setTimeout(callback, 1000);
    },
    err => {
        // 5 seconds have passed
    }
);
```

RxJS version

We can achieve the same kind of functionality by using the `Rx.Observable.while` method which takes a condition and an observable sequence that we created by calling `Rx.Observable.create`.

```
var Rx = require('rx');

var count = 0;

Rx.Observable.while(
        () => count < 5,
        Rx.Observable.create(function (obs) {
            setTimeout(() => observer.onNext(count++), 1000)
        }
    )
    .subscribe(
        x => { /* do something with each value */ },
        err => { /* handle errors */ },
        () => { /* 5 seconds have passed */ }
    );
```

`async.doWhilst`

The `async.doWhilst` method is a post check version of `whilst` . To reflect the difference in the order of operations test and fn arguments are switched. `doWhils` t is to `whilst` as `do while` is to `while` in plain JavaScript.

async version

In this example we'll just run a keep calling the callback while the count is less than 5.

```
var async = require('async');

var count = 0;

async.dowWilst(
    () => count < 5,
    callback => {
        count++;
        setTimeout(callback, 1000);
    },
    err => {
        // 5 seconds have passed
    }
);
```

RxJS version

We can achieve the same kind of functionality by using the `doWhile` on our observable sequence which takes a predicate to determine whether to continue running.

```
var Rx = require('rx');

var i = 0;

var source = Rx.Observable.return(42).doWhile(
    () => ++i < 2)
    .subscribe(
        consoel.log.bind(console),
        err => { /* handle errors */ },
        () => console.log('done')
    );
```

`async.nextTick`

The `async.nextTick` method calls the callback on a later loop around the event loop. In node.js this just calls process.nextTick, in the browser it falls back to setImmediate(callback) if available, otherwise setTimeout(callback, 0), which means other higher priority events may precede the execution of the callback.

async version

In this example we'll just run a keep calling the callback while the count is less than 5.

```
var async = require('async');

var call_order = [];

async.nextTick(() => {
    call_order.push('two');
    // call_order now equals ['one','two']
});

call_order.push('one');
```

RxJS version

We can achieve the same thing by using the `Rx.Scheduler.timeout` scheduler to schedule an item which will optimize for the runtime, for example, using `process.nextTick` if available, or `setImmediate` if available, or other fallbacks like `MessageChannel` , `postMessage` or even an async script load.

```
var Rx = require('rx');

var call_order = [];

Rx.Scheduler.timeout.schedule(() => {
    call_order.push('two');
    // call_order now equals ['one','two']
});

call_order.push('one');
```

`async.waterfall`

The `async.waterfall` method runs an array of functions in series, each passing their results to the next in the array. However, if any of the functions pass an error to the callback, the next function is not executed and the main callback is immediately called with the error.

async version

In this example, we'll check whether a file exists, then rename it and finally return its stats.

```javascript
var async = require('async'),
    fs = require('fs'),
    path = require('path');

// Get file and destination
var file = path.join(__dirname, 'file.txt'),
    dest = path.join(__dirname, 'file1.txt');

async.waterfall([
    callback => {
        fs.exists(file, flag => {
            if (flag) {
                callback(new Error('File does not exist.'))
            } else {
                callback(null);
            }
        });
    },
    callback => {
        fs.rename(file, dest, err => callback(err));
    },
    callback => {
        fs.stat(dest, (err, fsStat) => callback(err, fsStat));
    }
], (err, fsStat) => {
    if (err) {
        console.log(err);
    } else {
        console.log(JSON.stringify(fsStat));
    }
})
```

RxJS version

We can easily accomplish the same task as above using our wrappers for

`Rx.Observable.fromCallback` and `Rx.Observable.fromNodeCallback`, creating a waterfall-like method. First, let's implement a `waterfall` method using plain RxJS in which we enumerate the functions and call `flatMapLatest` on each resulting observable sequence to ensure we only get one value.

```
var Rx = require('rx');

var async = {
    waterfall: series => {
        return Rx.Observable.defer(() => {
            var acc = series[0]();
            for (var i = 1, len = series.length; i < len; i++) {

                // Pass in func to deal with closure capture
                (function (func) {

                    // Call flatMapLatest on each function
                    acc = acc.flatMapLatest(x => func(x));
                }(series[i]));
            }

            return acc;
        });
    }
}
```

Once we've defined this method, we can now use it such as the following, wrapping
`fs.exists` , `fs.rename` and `fs.stat` .

```
var Rx = require('rx'),
    fs = require('fs'),
    path = require('path');

var file = path.join(__dirname, 'file.txt'),
    dest = path.join(__dirname, 'file1.txt'),
    exists = Rx.Observable.fromCallback(fs.exists),
    rename = Rx.Observable.fromNodeCallback(fs.rename),
    stat = Rx.Observable.fromNodeCallback(fs.stat);

var obs = async.waterfall([
    () => exists(file),
    (flag) => {
        // Rename or throw computation
        return flag ?
            rename(file, dest) :
            Rx.Observable.throw(new Error('File does not exist.'));
    },
    () => stat(dest)
]);

// Now subscribe to get the results or error
obs.subscribe(
    fsStat => console.log(JSON.stringify(fsStat)),
    console.log.bind(console)
);
```

`async.compose`

The `async.compose` method creates a function which is a composition of the passed asynchronous functions. Each function consumes the return value of the function that follows. Composing functions f(), g() and h() would produce the result of f(g(h())), only this version uses callbacks to obtain the return values.

Each function is executed with the `this` binding of the composed function.

async version

In this example, we'll chain together two functions, one to add 1 to a supplied argument, and then chain it to another to multiply the result by 3.

```javascript
var async = require('async');

function add1(n, callback) {
    setTimeout(() => callback(null, n + 1), 10);
}

function mul3(n, callback) {
    setTimeout(() => callback(null, n * 3), 10);
}

var add1mul3 = async.compose(mul3, add1);

add1mul3(4, (err, result) => console.log(result));

// => 15
```

RxJS version

Using RxJS, we can accomplish this using the usual composition operator `selectMany` or `flatMap`. We'll wrap the `setTimeout` with a `wrapTimeout` method and ensure that we do deterministic cleanup via `clearTimeout`. Then we can compose together our `add1` and `mul3` functions which result in observable sequences.

```javascript
var Rx = require('rx');

function wrapTimeout (fn, arg) {
    return Rx.Observable.create(obs => {

        // Ensure the composition of the this argument
        var id = setTimeout(() => {
            obs.onNext(fn.call(fn, arg));
            obs.onCompleted();
        }, 10);

        // Handle cleanup/early disposal
        return () => clearTimeout(id)
    });
}

function add1 (n) {
    return wrapTimeout(x => x + 1, n);
}

function mul3 (n) {
    return wrapTimeout(x => x * 3, n);
}

add1(4)
    .flatMap(mul3)
    .subscribe(
        console.log.bind(console),
        err => console.log(`Error: ${e}`)
    );
// => 15
```

Reactive Extensions Configuration

Configuration information for the Reactive Extensions for JavaScript

Documentation

- `Rx.config.Promise`
- `Rx.config.useNativeEvents`

`Rx.config.Promise`

Sets the default Promise type to be used when the `toPromise` method is called. Note that the Promise implementation must conform to the ES6 specification. Some of those supported libraries are Q, RSVP, when.js among others. If not specified, this defaults to the native ES6 Promise, if available, else will throw an error.

Example

```
Rx.config.Promise = RSVP.Promise;

var p = Rx.Observable.just(1).toPromise()
  .then(value => console.log('Value: %s', s));
// => Value: 1
```

`Rx.config.useNativeEvents`

Determines whether the `fromEvent` method uses native DOM events only and disregards the referenced supported libraries such as jQuery, Zepto.js, AngularJS, Ember.js and Backbone.js

Example

For example, we could have jQuery referenced as part of our project, however, we only want native DOM events.

```html
<script src="jquery.js"></script>
<script src="rx.lite.js"></script>
```

We can do this by setting the `Rx.config.useNativeEvents` flag to `true`.

```js
Rx.config.useNativeEvents = true;

Rx.Observable.fromEvent(document, 'mousemove')
  .subscribe(e => console.log('ClientX: %d, ClientY: %d', e.clientX, e.clientY));
```

Helpers

- defaultComparer
- defaultSubComparer
- defaultError
- identity
- just
- isPromise
- noop
- pluck

`Rx.helpers.defaultComparer(x, y)`

The default equality comparer, used when a comparer is not supplied to a function. Uses an internal deep equality check.

Arguments

1. `x` *(Any)*: The first value to compare
2. `y` *(Any)*: The second value to compare

Returns

(Boolean): `true` if equal; else `false`.

Example

```javascript
var comparer = Rx.helpers.defaultComparer;

// Should return true
var x = 42, y = 42
console.log(comparer(x, y));
// => true

// Should return false
var x = new Date(0), y = new Date();
console.log(comparer(x, y));
// => false
```

`Rx.helpers.defaultSubComparer(x, y)`

The default comparer to determine whether one object is greater, less than or equal to another.

Arguments

1. `x` *(Any)*: The first value to compare
2. `y` *(Any)*: The second value to compare

Returns

(Number): Returns `1` if `x` is greater than `y` , `-1` if `y` is greater than `x` , and `0` if the objects are equal.

Example

```javascript
var comparer = Rx.helpers.defaultSubcomparer;

// Should return 0
var x = 42, y = 42
console.log(comparer(x, y));
// => 0

// Should return -1
var x = new Date(0), y = new Date();
console.log(comparer(x, y));
// => -1

// Should return 1
var x = 43, y = 42;
console.log(comparer(x, y));
// => 1
```

`Rx.helpers.defaultError(err)`

Throws the specified error

Arguments

1. `err` *(Any)*: The error to throw

Example

```
var defaultError = Rx.helpers.defaultError;

// Returns its value
defaultError(new Error('woops'))
// => Error: woops
```

Rx.helpers.identity(x)

A function which returns its value unmodified.

Arguments

1. `x` *(Any)*: The value to return.

Returns

(Any): The value given as the parameter.

Example

```
var identity = Rx.helpers.identity;

// Returns its value
var x = identity(42);
console.log(x);
// => 42
```

Rx.helpers.just(value)

A function which takes an argument and returns a function, when invoked, returns the argument.

Arguments

1. `value` *(Any)*: The value to return.

Returns

(Function): A function, when invoked, returns the value.

Example

```
var just = Rx.helpers.just;

Rx.Observable.timer(100)
  .map(just('foo'))
  .subscribe(console.log.bind(console));
// => foo
```

Rx.helpers.isPromise(p)

A function which determines whether the object is a `Promise` .

Arguments

1. `p` *(Any)*: The object to determine whether it is a promise.

Returns

(Boolean): `true` if the object is a `Promise` else `false`

Example

```
var isPromise = Rx.helpers.isPromise;

var p = RSVP.Promise(res => res(42));

console.log(isPromise(p));
// => true
```

Rx.helpers.noop()

A function which does nothing

Example

```
var noop = Rx.helpers.noop;

// This does nothing!
noop();
```

Rx.helpers.noop()

Rx.helpers.pluck(property)

Plucks a property from the object.

Arguments

1. `property` *(String)*: The property name to pluck from the object.

Returns

(Boolean): `true` if equal; else `false`.

Example

```
var pluck = Rx.helpers.pluck;

var source = Rx.Observable.interval(1000)
  .timeInterval()
  .map(pluck('value'))
  .take(3);

source.subscribe(console.log.bind(console));
// => 0
// => 1
// => 2
```

Observable object

The Observable object represents a push based collection.

The Observer and Objects interfaces provide a generalized mechanism for push-based notification, also known as the observer design pattern. The Observable object represents the object that sends notifications (the provider); the Observer object represents the class that receives them (the observer).

- Observable Methods
- Observable Instance Methods

Observable Methods

- amb
- case
- catch
- combineLatest
- concat
- create
- defer
- empty
- for | forIn
- forkJoin
- from
- fromCallback
- fromEvent
- fromEventPattern
- fromNodeCallback
- fromPromise
- generate
- generateWithAbsoluteTime
- generateWithRelativeTime
- if
- interval
- isObservable
- return | just
- merge
- mergeDelayError
- never
- of
- ofWithScheduler
- onErrorResumeNext
- pairs
- range
- repeat
- spawn
- start
- startAsync
- throw

- timer
- toAsync
- using
- when
- while | whileDo
- wrap
- zip

`Rx.Observable.amb(...args)`

</rx-marbles>

Propagates the observable sequence or Promise that reacts first. "amb" stands for [ambiguous](#).

Arguments

1. `args` *(Array|arguments)*: Observable sources or Promises competing to react first either as an array or arguments.

Returns

(`Observable`): An observable sequence that surfaces any of the given sequences, whichever reacted first.

Example

Using Observable sequences

Using Promises and Observables

`Rx.Observable.case(selector, sources, [elseSource|scheduler])`

Uses selector to determine which source in sources to use. There is an alias `switchCase` for browsers <IE9.

Arguments

1. `selector` (*Function*): The function which extracts the value for to test in a case statement.
2. `sources` (*Object*): A object which has keys which correspond to the case statement labels.
3. `[elseSource|scheduler]` (*Observable* | *Scheduler*): The observable sequence that will be run if the sources are not matched. If this is not provided, it defaults to `Rx.Observabe.empty` with the specified scheduler.

Returns

(*Observable*): An observable sequence which is determined by a case statement.

Example

`Rx.Observable.catch(...args)`

Continues an observable sequence that is terminated by an exception with the next observable sequence.

Arguments

1. `args` (*Array* | *arguments*): Observable sequences to catch exceptions for.

Returns

(*Observable*): An observable sequence containing elements from consecutive source sequences until a source sequence terminates successfully.

Example

`Rx.Observable.combineLatest(...args, [resultSelector])`

Merges the specified observable sequences into one observable sequence by using the selector function whenever any of the observable sequences produces an element. This can be in the form of an argument list of observables or an array. If the result selector is omitted, a list with the elements will be yielded.

Arguments

1. `args` *(arguments | Array)*: An array or arguments of Observable sequences.
2. `[resultSelector]` *(Function)*: Function to invoke whenever either of the sources produces an element. If omitted, a list with the elements will be yielded.

Returns

(Observable): An observable sequence containing the result of combining elements of the sources using the specified result selector function.

Example

Example

Rx.Observable.concat(...args)

Concatenates all of the specified observable sequences, as long as the previous observable sequence terminated successfully.

Arguments

1. `args` (*Array* | *arguments*): Observable sequences or Promises to concatenate.

Returns

(*Observable*): An observable sequence that contains the elements of each given sequence, in sequential order.

Example

Using Observable sequences

Using Promises and Observable sequences

`Rx.Observable.create(subscribe)`

Creates an observable sequence from a specified subscribe method implementation. This is an alias for the `createWithDisposable` method

Arguments

1. `subscribe` (*Function*): Implementation of the resulting observable sequence's subscribe method, optionally returning a function that will be wrapped in a disposable object. This could also be a disposable object.

Returns

(*Observable*): The observable sequence with the specified implementation for the subscribe method.

Example

Using a function

Using a disposable

Rx.Observable.defer(observableFactory)

Returns an observable sequence that invokes the specified factory function whenever a new observer subscribes.

Arguments

1. `observableFactory` (*Function*): Observable factory function to invoke for each observer that subscribes to the resulting sequence.

Returns

(*Observable*): An observable sequence whose observers trigger an invocation of the given observable factory function.

Example

Using an observable sequence

Using a promise

Rx.Observable.empty([scheduler])

Returns an empty observable sequence, using the specified scheduler to send out the single OnCompleted message.

Arguments

1. `[scheduler=Rx.Scheduler.immediate]` (*Scheduler*): Scheduler to send the termination call on.

Returns

(*Observable*): An observable sequence with no elements.

Example

`Rx.Observable.for(sources, resultSelector, [thisArg])`

Concatenates the observable sequences or Promises obtained by running the specified result selector for each element in source. There is an alias for this method called `forIn` for browsers <IE9

Arguments

1. `sources` *(Array)*: An array of values to turn into an observable sequence.
2. `resultSelector` *(Function)*: A function to apply to each item in the sources array to turn it into an observable sequence. The resultSelector is called with the following information:

    i. the value of the element
    ii. the index of the element
    iii. the Observable object being subscribed
3. `[thisArg]` *(Any)*: Object to use as `this` when executing `resultSelector` .

Returns

(Observable): An observable sequence from the concatenated observable sequences or Promises.

Example

Using Observables

Using Promises

Rx.Observable.forkJoin(...args [resultSelector])

Runs all observable sequences in parallel and collect their last elements.

Arguments

1. `args` (*Arguments* | *Array*): An array or arguments of Observable sequences or Promises to collect the last elements for.
2. `resultSelector` : (*Function*) - The result selector from all the values produced. If not specified, `forkJoin` will return the results as an array.

Returns

(*Observable*): An observable sequence with an array collecting the last elements of all the input sequences or the result of the result selector if specified.

Example

Example

`Rx.Observable.from(iterable, [mapFn], [thisArg], [scheduler])`

This method creates a new Observable sequence from an array-like or iterable object.

Arguments

1. `iterable` (*Array* | *Arguments* | *Iterable*): An array-like or iterable object to convert to an Observable sequence.
2. `[mapFn]` (*Function*): Map function to call on every element of the array.
3. `[thisArg]` (*Any*): The context to use calling the mapFn if provided.
4. `[scheduler=Rx.Scheduler.currentThread]` (*Scheduler*): Scheduler to run the enumeration of the input sequence on.

Returns

(*Observable*): The observable sequence whose elements are pulled from the given iterable sequence.

Example

Array-like object (arguments) to Observable

Set

Map

String

Array

Generate a sequence of numbers

`Rx.Observable.fromCallback(func, [scheduler], [context], [selector])`

Converts a callback function to an observable sequence.

Arguments

1. `func` (*Function*): Function with a callback as the last parameter to convert to an Observable sequence.
2. `[scheduler=Rx.Scheduler.timeout]` (*Scheduler*): Scheduler to run the function on. If not specified, defaults to `Rx.Scheduler.timeout` .
3. `[context]` (*Any*): The context for the func parameter to be executed. If not specified, defaults to undefined.
4. `[selector]` (*Function*): A selector which takes the arguments from the callback to produce a single item to yield on next.

Returns

(*Function*): A function, when executed with the required parameters minus the callback, produces an Observable sequence with a single value of the arguments to the callback as an array if no selector given, else the object created by the selector function.

Example

```javascript
var fs = require('fs'),
    Rx = require('rx');

// Wrap fs.exists
var exists = Rx.Observable.fromCallback(fs.exists);

// Check if file.txt exists
var source = exists('file.txt');

var subscription = source.subscribe(
    function (x) {
        console.log('Next: ' + result);
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

// => Next: true
// => Completed
```

`Rx.Observable.fromEvent(element, eventName, [selector])`

Creates an observable sequence by adding an event listener to the matching DOMElement, jQuery element, Zepto Element, Angular element, Ember.js element or EventEmitter.

Note that this uses the library approaches for jQuery, Zepto, Backbone.Marionette, AngularJS and Ember.js and falls back to native binding if not present. If you are using AMD you may need to include these libraries as dependencies of RxJs in your requirejs configuration file. RxJs will attempt to detect their presence when deciding which library to use.

Arguments

1. `element` (*Any*): The DOMElement, NodeList, jQuery element, Zepto Element, Angular element, Ember.js element or EventEmitter to attach a listener. For Backbone.Marionette this would be the application or an EventAggregator object.
2. `eventName` (*String*): The event name to attach the observable sequence.
3. `[selector]` (*Function*): A selector which takes the arguments from the event emitter so that you can return a single object.

Returns

(*Observable*): An observable sequence of events from the specified element and the specified event.

Example

Wrapping an event from [jQuery](#)

Using in Node.js with using an `EventEmitter` with a selector function (which is not required).

```javascript
var EventEmitter = require('events').EventEmitter,
    Rx = require('rx');

var eventEmitter = new EventEmitter();

var source = Rx.Observable.fromEvent(
    eventEmitter,
    'data',
    function (args) {
        return { foo: args[0], bar: args[1] };
    });

var subscription = source.subscribe(
    function (x) {
        console.log('Next: foo -' x.foo + ', bar -' + x.bar);
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

eventEmitter.emit('data', 'baz', 'quux');
// => Next: foo - baz, bar - quux
```

`Rx.Observable.fromEventPattern(addHandler, [removeHandler], [selector])`

Creates an observable sequence by using the addHandler and removeHandler functions to add and remove the handlers, with an optional selector function to project the event arguments.

Arguments

1. `addHandler` (*Function*): The DOMElement, NodeList or EventEmitter to attach a listener.
2. `removeHandler` (*Function*): The optional function to remove a handler from an emitter.
3. `[selector]` (*Function*): A selector which takes the arguments from the event handler to produce a single item to yield on next.

Returns

(*Observable*): An observable sequence of events from the specified element and the specified event.

Example

Wrapping an event from jQuery

Wrapping an event from the Dojo Toolkit

```
require(['dojo/on', 'dojo/dom', 'rx', 'rx.async', 'rx.binding'], function (on, dom, rx) {

    var input = dom.byId('input');

    var source = Rx.Observable.fromEventPattern(
        function add (h) {
            return on(input, 'click', h);
        },
        function remove (_, signal) {
            signal.remove();
        }
    );

    var subscription = source.subscribe(
        function (x) {
            console.log('Next: Clicked!');
        },
        function (err) {
            console.log('Error: ' + err);
        },
        function () {
            console.log('Completed');
        });

    on.emit(input, 'click');
    // => Next: Clicked!
});
```

Using in Node.js with using an `EventEmitter` .

```javascript
var EventEmitter = require('events').EventEmitter,
    Rx = require('rx');

var e = new EventEmitter();

// Wrap EventEmitter
var source = Rx.Observable.fromEventPattern(
    function add (h) {
        e.addListener('data', h);
    },
    function remove (h) {
        e.removeListener('data', h);
    },
    function (arr) {
        return arr[0] + ',' + arr[1];
    }
);

var subscription = source.subscribe(
    function (x) {
        console.log('Next: ' + result);
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

e.emit('data', 'foo', 'bar');
// => Next: foo,bar
```

`Rx.Observable.fromNodeCallback(func, [context], [selector])`

Converts a Node.js callback style function to an observable sequence. This must be in function (err, ...) format.

Arguments

1. `func` (*Function*): Function with a callback as the last parameter to convert to an Observable sequence.
2. `[context]` (*Any*): The context for the func parameter to be executed. If not specified, defaults to undefined.
3. `[selector]` (*Function*): A selector which takes the arguments from callback sans the error to produce a single item to yield on next.

Returns

(*Function*): A function which when applied, returns an observable sequence with the callback arguments as an array if no selector given, else the object created by the selector function on success, or an error if the first parameter is not falsy.

Example

```javascript
var fs = require('fs'),
    Rx = require('rx');

// Wrap fs.exists
var rename = Rx.Observable.fromNodeCallback(fs.rename);

// Rename file which returns no parameters except an error
var source = rename('file1.txt', 'file2.txt');

var subscription = source.subscribe(
    function () {
        console.log('Next: success!');
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

// => Next: success!
// => Completed
```

`Rx.Observable.fromPromise(promise)`

Converts a Promises/A+ spec compliant Promise and/or ES2015 compliant Promise to an Observable sequence.

Arguments

1. `promise` (*Promise*): Promises/A+ spec compliant Promise to an Observable sequence.

Returns

(*Observable*): An Observable sequence which wraps the existing promise success and failure.

Example

Create a promise which resolves 42

Create a promise which rejects with an error

Rx.Observable.generate(initialState, condition, iterate, resultSelector, [scheduler])

Converts an array to an observable sequence, using an optional scheduler to enumerate the array.

Arguments

1. `initialState` (*Any*): Initial state.
2. `condition` (*Function*): Condition to terminate generation (upon returning false).
3. `iterate` (*Function*): Iteration step function.
4. `resultSelector` (*Function*): Selector function for results produced in the sequence.
5. `[scheduler=Rx.Scheduler.currentThread]` (*Scheduler*): Scheduler on which to run the generator loop. If not provided, defaults to Scheduler.currentThread.

Returns

(*Observable*): The generated sequence.

Example

Rx.Observable.generateWithAbsoluteTime(initialState, condition, iterate, resultSelector, timeSelector, [scheduler])

Generates an observable sequence by iterating a state from an initial state until the condition fails.

Arguments

1. `initialState` (*Any*): Initial state.
2. `condition` (*Function*): Condition to terminate generation (upon returning false).
3. `iterate` (*Function*): Iteration step function.
4. `resultSelector` (*Function*): Selector function for results produced in the sequence.
5. `timeSelector` (*Function*): Time selector function to control the speed of values being produced each iteration, returning Date values.
6. `[scheduler=Rx.Scheduler.timeout]` (*Scheduler*): Scheduler on which to run the generator loop. If not provided, defaults to Scheduler.timeout.

Returns

(*Observable*): The generated sequence.

Example

Rx.Observable.generateWithRelativeTime(initial State, condition, iterate, resultSelector, timeSelector, [scheduler])

Generates an observable sequence by iterating a state from an initial state until the condition fails.

Arguments

1. `initialState` *(Any)*: Initial state.
2. `condition` *(Function)*: Condition to terminate generation (upon returning false).
3. `iterate` *(Function)*: Iteration step function.
4. `resultSelector` *(Function)*: Selector function for results produced in the sequence.
5. `timeSelector` *(Function)*: Time selector function to control the speed of values being produced each iteration, returning integer values denoting milliseconds.
6. `[scheduler=Rx.Scheduler.timeout]` *(Scheduler)*: Scheduler on which to run the generator loop. If not provided, defaults to Scheduler.timeout.

Returns

(Observable): The generated sequence.

Example

Rx.Observable.if(condition, thenSource, [elseSource])

Determines whether an observable collection contains values.

Arguments

1. `condition` *(Function)*: The condition which determines if the thenSource or elseSource will be run.
2. `thenSource` *(Observable)*: thenSource The observable sequence that will be run if the condition function returns true.
3. `[elseSource]` *(Observable|Scheduler)*: The observable sequence that will be run if the condition function returns false. If this is not provided, it defaults to Rx.Observabe.Empty with the specified scheduler.

Returns

(Observable): The generated sequence.

Example

This uses and only then source

The next example uses an elseSource

Rx.Observable.interval(period, [scheduler])

Returns an observable sequence that produces a value after each period.

Arguments

1. `period` (*Number*): Period for producing the values in the resulting sequence (specified as an integer denoting milliseconds).
2. `[scheduler]` *(Scheduler=Rx.Scheduler.timeout)*: Scheduler to run the timer on. If not specified, Rx.Scheduler.timeout is used.

Returns

(*Observable*): An observable sequence that produces a value after each period.

Example

Rx.Observable.interval(period, [scheduler])

`Rx.Observable.isObservable(observable)`

```
Rx.Observable.return(value, [scheduler]) |
Rx.Observable.just(value, [scheduler])
```

Returns an observable sequence that contains a single element, using the specified scheduler to send out observer messages.

This is an alias for `just`.

Arguments

1. `value` (*Any*): Single element in the resulting observable sequence.
2. `[scheduler=Rx.Scheduler.immediate]` (*Scheduler*): Scheduler to send the single element on. If not specified, defaults to Scheduler.immediate.

Returns

(*Observable*): An observable sequence with the single element.

Example

Rx.Observable.merge([scheduler], ...args)

Merges all the observable sequences and Promises into a single observable sequence.

Arguments

1. `[scheduler]` *(Scheduler=Rx.Scheduler.timeout)*: Scheduler to run the timer on. If not specified, Rx.Scheduler.immediate is used.
2. `args` *(Array|arguments)*: Observable sequences to merge into a single sequence.

Returns

(`Observable`): An observable sequence that produces a value after each period.

Example

Rx.Observable.mergeDelayError(...args)

Flattens an Observable that emits Observables into one Observable, in a way that allows an Observer to receive all successfully emitted items from all of the source Observables without being interrupted by an error notification from one of them.

This behaves like `Observable.prototype.mergeAll` except that if any of the merged Observables notify of an error via the Observer's `onError`, `mergeDelayError` will refrain from propagating that error notification until all of the merged Observables have finished emitting items.

Arguments

1. `args` *(Array|arguments)*: Arguments or an array of Observable sequences to merge.

Returns

(`Observable`): An Observable that emits all of the items emitted by the Observables emitted by the Observable

Example

```
var source1 = Rx.Observable.of(1,2,3);
var source2 = Rx.Observable.throwError(new Error('woops'));
var source3 = Rx.Observable.of(4,5,6);

var source = Rx.Observable.mergeDelayError(source1, source2, source3);

var subscription = source.subscribe(
  function (x) {
    console.log('Next: %s', x);
  },
  function (err) {
    console.log('Error: %s', err);
  },
  function () {
    console.log('Completed');
  });

// => 1
// => 2
// => 3
// => 4
// => 5
// => 6
// => Error: Error: woops
```

`Rx.Observable.never()`

Returns a non-terminating observable sequence, which can be used to denote an infinite duration (e.g. when using reactive joins).

Returns

(`Observable`): An observable sequence whose observers will never get called.

Example

`Rx.Observable.never()`

`Rx.Observable.of(...args)`

Converts arguments to an observable sequence.

Arguments

1. `args` *(Arguments)*: A list of arguments to turn into an Observable sequence.

Returns

(`Observable`): The observable sequence whose elements are pulled from the given arguments.

Example

`Rx.Observable.ofWithScheduler([scheduler], ...args)`

Converts arguments to an observable sequence, using an optional scheduler to enumerate the arguments.

Arguments

1. `[scheduler]` *(Scheduler)*: An optional scheduler used to enumerate the arguments.
2. `args` *(Arguments)*: A list of arguments to turn into an Observable sequence.

Returns

(`Observable`): The observable sequence whose elements are pulled from the given arguments.

Example

Rx.Observable.onErrorResumeNext(...args)

Continues an observable sequence that is terminated normally or by an exception with the next observable sequence or Promise.

Arguments

1. `args` *(Array|arguments)*: Observable sequences to concatenate.

Returns

(`Observable`): An observable sequence that concatenates the source sequences, even if a sequence terminates exceptionally.

Example

Rx.Observable.pairs(obj, [scheduler])

Convert an object into an observable sequence of [key, value] pairs using an optional `Scheduler` to enumerate the object.

Arguments

1. `obj` *(Object)*: The object to inspect and turn into an Observable sequence.
2. `[scheduler]` (*Scheduler*): Scheduler to run the enumeration of the input sequence on. If not specified, defaults to `Rx.Scheduler.currentThread`

Returns

(*Observable*): An observable sequence of [key, value] pairs from the object.

Example

```javascript
// Using Standard JavaScript
var obj = {
  foo: 42,
  bar: 56,
  baz: 78
};

var source = Rx.Observable.pairs(obj);

var subscription = source.subscribe(
  function (x) {
    console.log('Next: %s', x);
  },
  function (err) {
    console.log('Error: %s', err);
  },
  function () {
    console.log('Completed');
  });

// => Next: ['foo', 42]
// => Next: ['bar', 56]
// => Next: ['baz', 78]
// => Completed
```

ES6 makes for an even nicer experience such as:

```
let obj = {
  foo: 42,
  bar: 56,
  baz: 78
};

let source = Rx.Observable.pairs(obj);

let subscription = source.subscribe(
  x => {
    var [key, value] = x;
    console.log('Key:', key, 'Value:', value);
  },
  err => {
    console.log('Error: %s', err);
  },
  => () {
    console.log('Completed');
  });

// => Key: 'foo' Value: 42
// => Key: 'bar' Value: 56
// => Key: 'baz' Value: 78
// => Completed
```

`Rx.Observable.range(start, count, [scheduler])`

Generates an observable sequence of integral numbers within a specified range, using the specified scheduler to send out observer messages.

Arguments

1. `start` (*Number*): The value of the first integer in the sequence.
2. `count` (*Number*): The number of sequential integers to generate.
3. `[scheduler=Rx.Scheduler.currentThread]` (*Scheduler*): Scheduler to run the generator loop on. If not specified, defaults to Scheduler.currentThread.

Returns

(*Observable*): An observable sequence that contains a range of sequential integral numbers.

Example

`Rx.Observable.repeat(value, [repeatCount], [scheduler])`

Generates an observable sequence that repeats the given element the specified number of times, using the specified scheduler to send out observer messages.

Arguments

1. `value` (*Any*): Element to repeat.
2. `[repeatCount=-1]` (*Number*):Number of times to repeat the element. If not specified, repeats indefinitely.
3. `[scheduler=Rx.Scheduler.immediate]` (*Scheduler*): Scheduler to run the producer loop on. If not specified, defaults to Scheduler.immediate.

Returns

(*Observable*): An observable sequence that repeats the given element the specified number of times.

Example

Rx.Observable.spawn(fn)

Spawns a generator function which allows for Promises, Observable sequences, Arrays, Objects, Generators and functions.

Arguments

1. `fn` (*Function*): The spawning function.

Returns

(*Observable*): An Observable with the final result.

Example

```javascript
var Rx = require('rx');

var thunk = function (val) {
  return function (cb) {
    cb(null, val);
  };
};

var spawned = Rx.Observable.spawn(function* () {
  var v = yield thunk(12);
  var w = yield [24];
  var x = yield Rx.Observable.just(42);
  var y = yield Rx.Observable.just(56);
  var z = yield Promise.resolve(78);
  return v + w[0] + x + y + z;
});

spawned.subscribe(
  function (x) { console.log('next %s', x); },
  function (e) { console.log('error %s', e); },
  function () { console.log('completed'); }
);

// => next 212
// => completed
```

`Rx.Observable.start(func, [scheduler], [context])`

Invokes the specified function asynchronously on the specified scheduler, surfacing the result through an observable sequence.

Arguments

1. `func` (*Function*): Function to run asynchronously.
2. `[scheduler=Rx.Scheduler.timeout]` (*Scheduler*): Scheduler to run the function on. If not specified, defaults to Scheduler.timeout.
3. `[context]` (*Any*): The context for the func parameter to be executed. If not specified, defaults to undefined.

Returns

(*Observable*): An observable sequence exposing the function's result value, or an exception.

Example

Rx.Observable.startAsync(functionAsync)

Invokes the asynchronous function, surfacing the result through an observable sequence.

Arguments

1. `functionAsync` (*Function*): Asynchronous function which returns a Promise to run.

Returns

(*Observable*): An observable sequence exposing the function's Promises's value or error.

Example

Rx.Observable.throw(exception, [scheduler])

Returns an observable sequence that terminates with an exception, using the specified scheduler to send out the single onError message.

Arguments

1. `dueTime` (*Any*): Absolute (specified as a Date object) or relative time (specified as an integer denoting milliseconds) at which to produce the first value.
2. `[scheduler=Rx.Scheduler.immediate]` (*Scheduler*): Scheduler to send the exceptional termination call on. If not specified, defaults to the immediate scheduler.

Returns

(*Observable*): The observable sequence that terminates exceptionally with the specified exception object.

Example

Rx.Observable.timer(dueTime, [period], [scheduler])

Returns an observable sequence that produces a value after dueTime has elapsed and then after each period. Note for `rx.lite.js` , only relative time is supported.

Arguments

1. `dueTime` *(Date|Number)*: Absolute (specified as a Date object) or relative time (specified as an integer denoting milliseconds) at which to produce the first value.
2. `[period|scheduler=Rx.Scheduler.timeout]` *(Number|Scheduler)*: Period to produce subsequent values (specified as an integer denoting milliseconds), or the scheduler to run the timer on. If not specified, the resulting timer is not recurring.
3. `[scheduler=Rx.Scheduler.timeout]` (*Scheduler*): Scheduler to run the timer on. If not specified, the timeout scheduler is used.

Returns

(*Observable*): An observable sequence that produces a value after due time has elapsed and then each period.

Example

`Rx.Observable.toAsync(func, [scheduler], [context])`

Converts the function into an asynchronous function. Each invocation of the resulting asynchronous function causes an invocation of the original synchronous function on the specified scheduler.

Arguments

1. `func` (*Function*): Function to convert to an asynchronous function.
2. `[scheduler=Rx.Scheduler.timeout]` (*Scheduler*): Scheduler to run the function on. If not specified, defaults to Scheduler.timeout.
3. `[context]` (*Any*): The context for the func parameter to be executed. If not specified, defaults to undefined.

Returns

(*Function*): Asynchronous function.

Example

`Rx.Observable.using(resourceFactory, observableFactory)`

Constructs an observable sequence that depends on a resource object, whose lifetime is tied to the resulting observable sequence's lifetime.

Arguments

1. `resourceFactory` (*Function*): Factory function to obtain a resource object.
2. `observableFactory` (*Scheduler*): Factory function to obtain an observable sequence that depends on the obtained resource.

Returns

(*Function*): An observable sequence whose lifetime controls the lifetime of the dependent resource object.

Example

`Rx.Observable.when(...args)`

A series of plans (specified as an Array of as a series of arguments) created by use of the Then operator on patterns.

Arguments

1. `args` (*arguments* | *Array*): A series of plans (specified as an Array of as a series of arguments) created by use of the then operator on patterns.

Returns

(*Observable*): Observable sequence with the results form matching several patterns.

Example

`Rx.Observable.while(condition, source)`

Repeats source as long as condition holds emulating a while loop. There is an alias for this method called 'whileDo' for browsers <IE9.

Arguments

1. `condition` (*Function*): The condition which determines if the source will be repeated.
2. `source` (*Observable*): The observable sequence that will be run if the condition function returns true.

Returns

(*Observable*): An observable sequence which is repeated as long as the condition holds.

Example

Rx.Observable.wrap(fn)

Wrap the given generator fn into a function that returns an Observable.

Arguments

1. `fn` (*Function*): A generator function to wrap.

Returns

(*Observable*): A function once executed, returns an Observable.

Example

```
var Rx = require('rx');

var fn = Rx.Observable.wrap(function* (val) {
  return yield Observable.just(val);
});

fn(42).subscribe(
  function (x) { console.log('next %s', x); },
  function (e) { console.log('error %s', e); },
  function () { console.log('completed'); }
);

// => next 42
// => completed
```

Rx.Observable.zip(...args, [resultSelector])

Merges the specified observable sequences or Promises into one observable sequence by using the selector function whenever all of the observable sequences have produced an element at a corresponding index. If the result selector function is omitted, a list with the elements of the observable sequences at corresponding indexes will be yielded.

Arguments

1. `args` (*Array* | *arguments*): Observable sources.
2. `[resultSelector]` *(Function)*: A function which takes the inputs at the specified index and combines them together. If omitted, a list with the elements of the observable sequences at corresponding indexes will be yielded.

Returns

(*Observable*): An observable sequence containing the result of combining elements of the sources using the specified result selector function.

Example

Without a result selector

```
var range = Rx.Observable.range(0, 5);

var source = Observable.zip(
  range,
  range.skip(1),
  range.skip(2)
);

var subscription = source.subscribe(
  function (x) {
    console.log('Next: %s', x);
  },
  function (err) {
    console.log('Error: %s', err);
  },
  function () {
    console.log('Completed');
  });

// => Next: 0,1,2
// => Next: 1,2,3
// => Next: 2,3,4
// => Completed
```

Using arguments

Using promises and Observables */

Observable Instance Methods

- amb
- and
- asObservable
- average
- buffer
- bufferWithCount
- bufferWithTime
- bufferWithTimeOrCount
- catch
- combineLatest
- concat
- concatAll
- concatMapObserver | selectConcatObserver
- controlled
- count
- debounce
- defaultIfEmpty
- delay
- delaySubscription
- dematerialize
- distinct
- distinctUntilChanged
- do | doAction | tap
- doOnCompleted | tapOnCompleted
- doOnError | tapOnError
- doOnNext | tapOnNext
- doWhile
- elementAt
- every
- expand
- extend | manySelect
- filter | where
- finally
- find
- findIndex
- first

- flatMap | selectMany
- flatMapConcat | concatMap
- flatMapFirst | selectSwitchFirst
- flatMapLatest
- flatMapObserver | selectManyObserver
- flatMapWithMaxConcurrent
- forkJoin
- groupBy
- groupByUntil
- groupJoin
- ignoreElements
- includes
- indexOf
- isEmpty
- join
- jortSort
- jortSortUntil
- last
- lastIndexOf
- let | letBind
- map | select
- materialize
- max
- maxBy
- merge
- mergeAll
- min
- minBy
- multicast
- observeOn
- onErrorResumeNext
- pairwise
- partition
- pausable
- pausableBuffered
- pipe
- pluck
- publish
- publishLast
- publishValue

- reduce
- repeat
- replay
- retry
- retryWhen
- scan
- sequenceEqual
- share
- shareReplay
- shareValue
- single
- singleInstance
- skip
- skipLast
- skipLastWithTime
- skipUntil
- skipUntilWithTime
- skipWhile
- skipWithTime
- slice
- some
- startWith
- subscribe | forEach
- subscribeOn
- subscribeOnCompleted
- subscribeOnError
- subscribeOnNext
- sum
- switch
- switchFirst
- take
- takeLast
- takeLastBuffer
- takeLastBufferWithTime
- takeLastWithTime
- takeUntil
- takeUntilWithTime
- takeWhile
- takeWithTime
- thenDo

- throttle
- throttleLatest | sample
- timeInterval
- timeout
- timestamp
- toArray
- toMap
- toPromise
- toSet
- transduce
- window
- windowWithCount
- windowWithTime
- windowWithTimeOrCount
- withLatestFrom
- zip
- zipIterable

Rx.Observable.prototype.amb(rightSource)

</rx-marbles>

Propagates the observable sequence or Promise that reacts first. "amb" stands for [ambiguous](#).

Arguments

1. `rightSource` (*Observable*): Second observable sequence.

Returns

(*Observable*): An observable sequence that surfaces either of the given sequences, whichever reacted first.

Example

Rx.Observable.prototype.and(rightSource)

Propagates the observable sequence that reacts first.

Arguments

1. `right` (*Observable*): Observable sequence to match with the current sequence.

Returns

(*Pattern*): Pattern object that matches when both observable sequences have an available value.

Example

`Rx.Observable.prototype.asObservable()`

Hides the identity of an observable sequence.

Returns

(*Observable*): An observable sequence that hides the identity of the source sequence.

Example

`Rx.Observable.prototype.average([selector], [thisArg])`

</rx-marbles>

Computes the average of an observable sequence of values that are in the sequence or obtained by invoking a transform function on each element of the input sequence if present.

Arguments

1. `[selector]` (*Function*): A transform function to apply to each element.
2. `[thisArg]` (*Any*): Object to use as `this` when executing `selector` .

Returns

(*Observable*): An observable sequence containing a single element with the average of the sequence of values.

Example

Without a selector

With a selector

Rx.Observable.prototype.buffer()

The `buffer` method periodically gathers items emitted by a source Observable into buffers, and emits these buffers as its own emissions.

Note that if the source Observable issues an `onError` notification, `buffer` will pass on this notification immediately without first emitting the buffer it is in the process of assembling, even if that buffer contains items that were emitted by the source Observable before it issued the error notification.

There are a number of ways with which you can regulate how `buffer` gathers items from the source Observable into buffers:

With buffer closing selector

```
Rx.Observable.prototype.buffer(bufferClosingSelector);
```

Returns an Observable that emits buffers of items it collects from the source `Observable`. The resulting `Observable` emits connected, non-overlapping buffers. It emits the current buffer and replaces it with a new buffer whenever the `Observable` produced by the specified `bufferClosingSelector` emits an item.

Arguments

1. `bufferClosingSelector` (*Function*): A function invoked to define the closing of each produced window.

Returns

(*Observable*): An observable sequence of windows.

Example

With buffer opening and buffer closing selector

```
Rx.Observable.prototype.buffer(bufferOpenings, bufferClosingSelector);
```

This version of `buffer` monitors an `Observable`, `bufferOpenings`, that emits Observable objects. Each time it observes such an emitted object, it creates a new bundle to begin collecting items emitted by the source Observable and it passes the `bufferOpenings` Observable into the `bufferClosingSelector` function. That function returns an `Observable`. `buffer` monitors that `Observable` and when it detects an emitted object, it closes its bundle and emits it as its own emission.

1. `bufferOpenings` (*Observable*): Observable sequence whose elements denote the creation of new windows.
2. `bufferClosingSelector` (*Function*): A function invoked to define the closing of each produced window.

Returns

(*Observable*): An observable sequence of windows.

Example

With boundaries

```
Rx.Observable.prototype.buffer(bufferBoundaries);
```

Returns an `Observable` that emits non-overlapping buffered items from the source `Observable` each time the specified boundary `Observable` emits an item.

Arguments

1. `bufferBoundaries` (*Observable*): Sequence of buffer boundary markers. The current buffer is closed and a new buffer is opened upon receiving a boundary marker.

Returns

(*Observable*): An observable sequence of windows.

Example

`Rx.Observable.prototype.bufferWithCount(count, [skip])`

Projects each element of an observable sequence into zero or more buffers which are produced based on element count information.

Arguments

1. `count` (*Function*): Length of each buffer.
2. `[skip]` (*Function*): Number of elements to skip between creation of consecutive buffers. If not provided, defaults to the count.

Returns

(*Observable*): An observable sequence of buffers.

Example

Without a skip

Using a skip

Rx.Observable.prototype.bufferWithTime(timeSpan, [timeShift | scheduler], [scheduler])

Projects each element of an observable sequence into zero or more buffers which are produced based on timing information.

Arguments

1. `timeSpan` (*Number*): Length of each buffer (specified as an integer denoting milliseconds).
2. `[timeShift]` (*Number*): Interval between creation of consecutive buffers (specified as an integer denoting milliseconds).
3. `[scheduler=Rx.Scheduler.timeout]` (*Scheduler*): Scheduler to run buffer timers on. If not specified, the timeout scheduler is used.

Returns

(*Observable*): An observable sequence of buffers.

Example

Without a skip

Using a skip

Rx.Observable.prototype.bufferWithTimeOrCount(timeSpan, count, [scheduler])

Projects each element of an observable sequence into a buffer that is completed when either it's full or a given amount of time has elapsed.

Arguments

1. `timeSpan` (*Number*): Maximum time length of a buffer.
2. `count` (*Number*): Maximum element count of a buffer.
3. `[scheduler=Rx.Scheduler.timeout]` (*Scheduler*): Scheduler to run buffer timers on. If not specified, the timeout scheduler is used.

Returns

(*Observable*): An observable sequence of buffers.

Example

`Rx.Observable.prototype.catch(second | handler)`

Continues an observable sequence that is terminated by an exception with the next observable sequence. There is an alias for this method `catchException` for browsers <IE9

Arguments

1. `second` (*Observable*): A second observable sequence used to produce results when an error occurred in the first sequence.
2. `handler` (*Function*): Exception handler function that returns an observable sequence given the error that occurred in the first sequence

Returns

(*Observable*): An observable sequence containing the first sequence's elements, followed by the elements of the handler sequence in case an exception occurred.

Example

Using a second observable

Using a handler function

`Rx.Observable.prototype.combineLatest(...args, [resultSelector])`

</rx-marbles>

Merges the specified observable sequences into one observable sequence by using the selector function whenever any of the observable sequences produces an element. This can be in the form of an argument list of observables or an array. If the result selector is omitted, a list with the elements will be yielded.

Arguments

1. `args` *(arguments | Array)*: An array or arguments of Observable sequences.
2. `[resultSelector]` (*Function*): Function to invoke whenever either of the sources produces an element. If omitted, a list with the elements will be yielded.

Returns

(*Observable*): An observable sequence containing the result of combining elements of the sources using the specified result selector function.

Examples

Example

Example

Rx.Observable.prototype.concat(...args)

</rx-marbles>

Concatenates all the observable sequences. This takes in either an array or variable arguments to concatenate.

Arguments

1. `args` (*arguments* | *Array*): An array or arguments of Observable sequences.

Returns

(*Observable*): An observable sequence that contains the elements of each given sequence, in sequential order.

Example

```javascript
/* Using Promises and Observable sequences */
var source1 = Rx.Observable.return(42);
var source2 = RSVP.Promise.resolve(56);

var source = Rx.Observable.concat(source1, source2);

var subscription = source.subscribe(
    function (x) {
        console.log('Next: ' + x);
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

// => Next: 42
// => Next: 56
// => Completed
```

Rx.Observable.prototype.concatAll()

Concatenates a sequence of observable sequences or promises into a single observable sequence.

Returns

(*Observable*): The observable sequence that merges the elements of the inner sequences.

Example

Rx.Observable.prototype.concatAll()

```
Rx.Observable.prototype.concatMapObserver(onNe
xt, onError, onCompleted, [thisArg]) .
 Rx.Observable.prototype.selectConcatObserver(o
nNext, onError, onCompleted, [thisArg])
```

Projects each notification of an observable sequence to an observable sequence and concats the resulting observable sequences into one observable sequence.

Arguments

1. `onNext` (*Function*): A transform function to apply to each element. The selector is called with the following information:
   i. the value of the element
   ii. the index of the element
2. `onError` (*Function*): A transform function to apply when an error occurs in the source sequence.
3. `onCompleted` (*Function*): A transform function to apply when the end of the source sequence is reached.
4. `[thisArg]` (*Any*): Object to use as `this` when executing the transform functions.

Returns

(*Observable*): An observable sequence whose elements are the result of invoking the one-to-many transform function corresponding to each notification in the input sequence.

Example

`Rx.Observable.prototype.controlled([enableQueue])`

Attaches a controller to the observable sequence with the ability to queue.

Arguments

1. `[enableQueue]` *(Boolean)*: Whether to enable queueing. If not specified, defaults to true.

Returns

(`Observable`): An observable sequence which can be used to request values from the sequence.

Example

`Rx.Observable.prototype.count([predicate], [thisArg])`

</rx-marbles>

Returns an observable sequence containing a value that represents how many elements in the specified observable sequence satisfy a condition if provided, else the count of items.

Arguments

1. `[predicate]` (*Function*): A function to test each element for a condition. The callback is called with the following information:
    i. the value of the element
    ii. the index of the element
    iii. the Observable object being subscribed
2. `[thisArg]` (*Any*): Object to use as `this` when executing `predicate`.

Returns

(*Observable*): An observable sequence containing a single element with a number that represents how many elements in the input sequence satisfy the condition in the predicate function if provided, else the count of items in the sequence.

Example

Without a predicate

With a predicate

**Rx.Observable.prototype.debounce(dueTime,
[scheduler])** ,
**Rx.Observable.prototype.throttle(dueTime,
[scheduler])**

Emits an item from the source Observable after a particular timespan has passed without the Observable omitting any other items.

--OR--

Ignores values from an observable sequence which are followed by another value within a computed debounced duration.

Arguments

If using a relative due time:

1. `dueTime` (*Number*): Duration of the throttle period for each value (specified as an integer denoting milliseconds).
2. `[scheduler]` (*Any*): Scheduler to run the throttle timers on. If not specified, the default scheduler is used.

If using the duration selector function:

1. `durationSelector` (*Function*): Selector function to retrieve a sequence indicating the throttle duration for each given element.

Returns

(*Observable*): The debounced sequence.

Example

```javascript
var times = [
    { value: 0, time: 100 },
    { value: 1, time: 600 },
    { value: 2, time: 400 },
    { value: 3, time: 700 },
    { value: 4, time: 200 }
];

// Delay each item by time and project value;
var source = Rx.Observable.from(times)
  .flatMap(function (item) {
    return Rx.Observable
      .of(item.value)
      .delay(item.time);
  })
  .debounce(500 /* ms */);

var subscription = source.subscribe(
  function (x) {
    console.log('Next: %s', x);
  },
  function (err) {
    console.log('Error: %s', err);
  },
  function () {
    console.log('Completed');
  });

// => Next: 3
// => Completed
```

```
/* Using Selector */
var array = [
  800,
  700,
  600,
  500
];

var source = Rx.Observable.for(
    array,
    function (x) { return Rx.Observable.timer(x) }
  )
  .map(function(x, i) { return i; })
  .debounce(function (x) { return Rx.Observable.timer(700); });

var subscription = source.subscribe(
  function (x) {
    console.log('Next: %s', x);
  },
  function (err) {
    console.log('Error: %s', err);
  },
  function () {
    console.log('Completed');
  });

// => Next: 0
// => Next: 3
// => Completed
```

`Rx.Observable.prototype.defaultIfEmpty([defaultValue])`

Returns the elements of the specified sequence or the specified value in a singleton sequence if the sequence is empty.

Arguments

1. `[defaultValue=null]` (*Any*): The value to return if the sequence is empty. If not provided, this defaults to null.

Returns

(*Observable*): An observable sequence that contains the specified default value if the source is empty; otherwise, the elements of the source itself.

Example

Without a default value

With a defaultValue

```
Rx.Observable.prototype.delay(dueTime,
[scheduler]) ,
 Rx.Observable.prototype.delay([subscriptionDel
ay], delayDurationSelector)
```

</rx-marbles>

Time shifts the observable sequence by dueTime. The relative time intervals between the values are preserved.

--OR--

Time shifts the observable sequence based on a subscription delay and a delay selector function for each element.

Arguments

For delays with an absolute or relative time:

1. `dueTime` *(Date | Number)*: Absolute (specified as a Date object) or relative time (specified as an integer denoting milliseconds) by which to shift the observable sequence.
2. `[scheduler]` (*Scheduler*): Scheduler to run the delay timers on. If not specified, the default scheduler is used.

For delays with a delay selector function:

1. `[subscriptionDelay]` (*Observable*): Sequence indicating the delay for the subscription to the source.
2. `delayDurationSelector` (*Function*): Selector function to retrieve a sequence indicating the delay for each given element.

Returns

(*Observable*): Time-shifted sequence.

Example

Using an absolute time to delay by a second

Using an relatove time to delay by a second

```
/* With subscriptionDelay */
var source = Rx.Observable
  .range(0, 3)
  .delay(
    Rx.Observable.timer(300),
    function (x) { return Rx.Observable.timer(x * 400); }
  )
  .timeInterval()
  .map(function (x) { return x.value + ':' + x.interval; });

var subscription = source.subscribe(
  function (x) {
    console.log('Next: %s', x);
  },
  function (err) {
    console.log('Error: %s', err);
  },
  function () {
    console.log('Completed');
  });


// => Next: 0:300
// => Next: 1:400
// => Next: 2:400
// => Completed
```

```
/* Without subscriptionDelay */
var source = Rx.Observable
  .range(0, 3)
  .delay(function (x) { return Rx.Observable.timer(x * 400); })
  .timeInterval()
  .map(function (x) { return x.value + ':' + x.interval; });

var subscription = source.subscribe(
  function (x) {
    console.log('Next: %s', x);
  },
  function (err) {
    console.log('Error: %s', err);
  },
  function () {
    console.log('Completed');
  });

// => Next: 0:0
// => Next: 1:400
// => Next: 2:400
// => Completed
```

Rx.Observable.prototype.delaySubscription(dueTime, [scheduler])

Time shifts the observable sequence by delaying the subscription with the specified relative time duration, using the specified scheduler to run timers.

This operator is more efficient than `delay` but postpones all side-effects of subscription and affects error propagation timing.

The side-effects of subscribing to the source sequence will be run on the specified scheduler. Observer callbacks will not be affected.

Arguments

1. `dueTime` (*Number | Date*): Relative (Number) or Absolute (Date) time shift of the subscription.
2. `[scheduler]` (*Scheduler*): Scheduler to run the subscription delay timer on. If not specified, the default scheduler is used.

Returns

(*Observable*): Time-shifted sequence.

Example

```javascript
var start = Date.now()

var source = Rx.Observable
  .range(0, 3)
  .delaySubscription(5000);

var subscription = source.subscribe(
  function (x) {
    console.log('Next: %s, %s', x, Date.now() - start);
  },
  function (err) {
    console.log('Error: %s', err);
  },
  function () {
    console.log('Completed');
  });

//=> Next: 0, 5001
//=> Next: 1, 5002
//=> Next: 2, 5003
//=> Completed
```

Rx.Observable.prototype.dematerialize()

Dematerializes the explicit notification values of an observable sequence as implicit notifications.

Returns

(*Observable*): An observable sequence exhibiting the behavior corresponding to the source sequence's notification values.

Example

Rx.Observable.prototype.dematerialize()

`Rx.Observable.prototype.distinct([keySelector], [keySerializer])`

</rx-marbles>

Returns an observable sequence that contains only distinct elements according to the keySelector and the comparer. Usage of this operator should be considered carefully due to the maintenance of an internal lookup structure which can grow large.

Arguments

1. `[keySelector]` (*Function*): A function to compute the comparison key for each element.
2. `[keySerializer]` (*Function*): Used to serialize the given object into a string for object comparison.

Returns

(*Observable*): An observable sequence only containing the distinct elements, based on a computed key value, from the source sequence.

Example

Without key selector

With key selector

Rx.Observable.prototype.distinctUntilChanged([keySelector], [comparer])

</rx-marbles>

Returns an observable sequence that contains only distinct contiguous elements according to the keySelector and the comparer.

Arguments

1. `[keySelector]` (*Function*): A function to compute the comparison key for each element.
2. `[comparer]` (*Function*): Equality comparer for computed key values. If not provided, defaults to an equality comparer function.

Returns

(*Observable*): An observable sequence only containing the distinct elements, based on a computed key value, from the source sequence.

Example

```
/* Without key selector */
var source = Rx.Observable.fromArray([
        42, 42, 24, 24
    ])
    .distinct();

var subscription = source.subscribe(
    function (x) {
        console.log('Next: ' + x.toString());
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

// => Next: 42
// => Next: 24
// => Completed

/* With key selector */
var source = Rx.Observable.fromArray([
        {value: 42}, {value: 24}, {value: 42}, {value: 24}
    ])
    .distinct(function (x) { return x.value; });

var subscription = source.subscribe(
    function (x) {
        console.log('Next: ' + x.toString());
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

// => Next: { value: 42 }
// => Next: { value: 24 }
// => Completed
```

```
Rx.Observable.prototype.do(observer |
[onNext], [onError], [onCompleted]) ,
 Rx.Observable.prototype.doAction(observer |
[onNext], [onError], [onCompleted]) ,
 Rx.Observable.prototype.tap(observer |
[onNext], [onError], [onCompleted])
```

Invokes an action for each element in the observable sequence and invokes an action upon graceful or exceptional termination of the observable sequence.

This method can be used for debugging, logging, etc. of query behavior by intercepting the message stream to run arbitrary actions for messages on the pipeline.

There is an alias to this method `doAction` for browsers <IE9 and `tap` as well.

Arguments

1. `[observer]` *(Observer)*: An observer to invoke for each element in the observable sequence.
2. `[onNext]` *(Function)*: Function to invoke for each element in the observable sequence.
3. `[onError]` *(Function)*: Function to invoke upon exceptional termination of the observable sequence. Used if only the first parameter is also a function.
4. `[oncompleted]` *(Function)*: Function to invoke upon graceful termination of the observable sequence. Used if only the first parameter is also a function.

Returns

(Observable): The source sequence with the side-effecting behavior applied.

Example

Using a function

Using an observer

```
Rx.Observable.prototype.doOnCompleted(onComple
ted, [thisArg]) ,
 Rx.Observable.prototype.tapOnCompleted(onCompl
eted, [thisArg])
```

Invokes an action upon graceful termination of the observable sequence.

This method can be used for debugging, logging, etc. of query behavior by intercepting the message stream to run arbitrary actions for messages on the pipeline.

Arguments

1. `oncompleted` (*Function*): Function to invoke upon graceful termination of the observable sequence.
2. [`thisArg`] *(Any)*: Object to use as this when executing callback.

Returns

(*Observable*): The source sequence with the side-effecting behavior applied.

Example

Using a function

Using a thisArg

```
Rx.Observable.prototype.doOnError(onError,
[thisArg]) ,
 Rx.Observable.prototype.tapOnError(onError,
[thisArg])
```

Invokes an action upon exceptional termination of the observable sequence.

This method can be used for debugging, logging, etc. of query behavior by intercepting the message stream to run arbitrary actions for messages on the pipeline.

Arguments

1. `onError` *(Function)*: Function to invoke upon exceptional termination of the observable sequence.
2. [`thisArg`] *(Any)*: Object to use as this when executing callback.

Returns

(Observable): The source sequence with the side-effecting behavior applied.

Example

Using a function

Using a thisArg

```
Rx.Observable.prototype.doOnNext(onNext,
[thisArg]) ,
 Rx.Observable.prototype.tapOnNext(onNext,
[thisArg])
```

Invokes an action for each element of the observable sequence.

This method can be used for debugging, logging, etc. of query behavior by intercepting the message stream to run arbitrary actions for messages on the pipeline.

Arguments

1. `onNext` (*Function*): Function to invoke for each element in the observable sequence.
2. [`thisArg`] *(Any)*: Object to use as this when executing callback.

Returns

(*Observable*): The source sequence with the side-effecting behavior applied.

Example

Using a function

Using a thisArg

`Rx.Observable.prototype.doWhile(condition)`

Repeats source as long as condition holds emulating a do while loop.

Arguments

1. `condition` (*Function*): The condition which determines if the source will be repeated.

Returns

(*Observable*): An observable sequence whose observers trigger an invocation of the given observable factory function.

Example

Rx.Observable.prototype.elementAt(index, [defaultValue])

</rx-marbles>

Returns the element at a specified index in a sequence. If the specified index is out of bounds and no default value was given `onError` will be called with an error, however, if a default value has been specified, that value will be returned via an `onNext` call.

Arguments

1. `index` (*Number*): The zero-based index of the element to retrieve.
2. `[defaultValue]` (*Any*): Default value if no such element exists.

Returns

(*Observable*): An observable sequence that produces the element at the specified position in the source sequence. If the specified index is out of bounds and no default value was given `onError` will be called with an error, however, if a default value has been specified, that value will be returned via an `onNext` call.

Example

Finds an index

Not found

```js
/* With default value */
var source = Rx.Observable.from([1,2,3,4])
  .elementAt(4, 42);

var subscription = source.subscribe(
  function (x) {
    console.log('Next: %s', x);
  },
  function (err) {
    console.log('Error: %s', err);
  },
  function () {
    console.log('Completed');
  });

// => Next: 42
// => Completed
```

/* With default value */
var source = Rx.Observable.from([1,2,3,4])

Rx.Observable.prototype.every(predicate, [thisArg])

</rx-marbles>

Determines whether all elements of an observable sequence satisfy a condition.

Arguments

1. `predicate` (*Function*): A function to test each element for a condition.
2. `[thisArg]` (*Function*): Object to use as this when executing callback.

Returns

(*Observable*): An observable sequence containing a single element determining whether all elements in the source sequence pass the test in the specified predicate.

Example

`Rx.Observable.prototype.expand(selector, [scheduler])`

Expands an observable sequence by recursively invoking selector.

Arguments

1. `selector` (*Function*): Selector function to invoke for each produced element, resulting in another sequence to which the selector will be invoked recursively again.
2. `[scheduler=Rx.Scheduler.immediate]` (*Scheduler*): Scheduler on which to perform the expansion. If not provided, this defaults to the immediate scheduler.

Returns

(*Observable*): An observable sequence containing a single element determining whether all elements in the source sequence pass the test in the specified predicate.

Example

```
Rx.Observable.prototype.manySelect(selector,
[scheduler]) ,
 Rx.Observable.prototype.extend(selector,
[scheduler])
```

Comonadic bind operator.

Arguments

1. `selector` (*Function*): A transform function to apply to each element.
2. `[scheduler=Rx.Scheduler.immediate]` (*Scheduler*): Scheduler used to execute the operation. If not specified, defaults to the `Rx.Scheduler.immediate` scheduler.

Returns

(*Observable*): An observable sequence which results from the comonadic bind operation.

Example

Rx.Observable.prototype.filter(predicate, [thisArg])

</rx-marbles>

Filters the elements of an observable sequence based on a predicate. This is an alias for the `where` method.

Arguments

1. `predicate` (*Function*): A function to test each source element for a condition. The callback is called with the following information:
   i. the value of the element
   ii. the index of the element
   iii. the Observable object being subscribed
2. `[thisArg]` (*Any*): Object to use as `this` when executing the predicate.

Returns

(*Observable*): An observable sequence that contains elements from the input sequence that satisfy the condition.

Example

Rx.Observable.prototype.finally(action)

Invokes a specified action after the source observable sequence terminates gracefully or exceptionally.

Arguments

1. `predicate` (*Function*): A function to test each source element for a condition; The callback is called with the following information:
     i. the value of the element
     ii. the index of the element
     iii. the Observable object being subscribed
2. `[thisArg]` (*Any*): Object to use as `this` when executing the predicate.

Returns

(*Observable*): An observable sequence that contains elements from the input sequence that satisfy the condition.

Example

Rx.Observable.prototype.find(predicate, [thisArg])

</rx-marbles>

Searches for an element that matches the conditions defined by the specified predicate, and returns the first occurrence within the entire Observable sequence.

Arguments

1. `predicate` (*Function*): A function to test each source element for a condition; The callback is called with the following information:
   i. the value of the element
   ii. the index of the element
   iii. the Observable object being subscribed
2. `[thisArg]` (*Any*): Object to use as `this` when executing the predicate.

Returns

(*Observable*): An Observable sequence with the first element that matches the conditions defined by the specified predicate, if found; otherwise, undefined.

Example

```
/* Not found */
var array = [1,2,3,4];

var source = Rx.Observable.from(array)
  .find(function (x, i, obs) { return x === 5; });

var subscription = source.subscribe(
  function (x) {
    console.log('Next: %s', x);
  },
  function (err) {
    console.log('Error: %s', err);
  },
  function () {
    console.log('Completed');
  });

// => Completed
```

Rx.Observable.prototype.findIndex(predicate, [thisArg])

</rx-marbles>

Searches for an element that matches the conditions defined by the specified predicate, and returns the first occurrence within the entire Observable sequence.

Arguments

1. `predicate` (*Function*): A function to test each source element for a condition; The callback is called with the following information:
    i. the value of the element
    ii. the index of the element
    iii. the Observable object being subscribed
2. `[thisArg]` (*Any*): Object to use as `this` when executing the predicate.

Returns

(*Observable*): An Observable sequence with the first element that matches the conditions defined by the specified predicate, if found; otherwise, undefined.

Example

```js
/* Not found */
var array = [1,2,3,4];

var source = Rx.Observable.from(array)
  .findIndex(function (x, i, obs) { return x === 5; });

var subscription = source.subscribe(
  function (x) {
    console.log('Next: %s', x);
  },
  function (err) {
    console.log('Error: %s', err);
  },
  function () {
    console.log('Completed');
  });

// => Next: -1
// => Completed
```

```
Rx.Observable.prototype.first([predicate],
[thisArg], [defaultValue]) .
Rx.Observable.prototype.first([settings])
```

</rx-marbles>

Returns the first element of an observable sequence that satisfies the condition in the predicate, or a default value if no such element exists. If no default value is given, then `onError` will be called.

Arguments

`Rx.Observable.prototype.first([predicate], [thisArg], [defaultValue])`

1. `[predicate]` (*Function*): A predicate function to evaluate for elements in the source sequence. The callback is called with the following information:
   i. the value of the element
   ii. the index of the element
   iii. the Observable object being subscribed
2. `[thisArg]` (*Any*): Object to use as `this` when executing the predicate.
3. `[defaultValue]` (*Any*): Default value if no such element exists.

`Rx.Observable.prototype.first([settings])`

1. `[settings]` (*Object*): An object with the following fields
   ○ `[predicate]` (*Function*): A predicate function to evaluate for elements in the source sequence. The callback is called with the following information:
      i. the value of the element
      ii. the index of the element
      iii. the Observable object being subscribed
   ○ `[thisArg]` (*Any*): Object to use as `this` when executing the predicate.
   ○ `[defaultValue]` (*Any*): Default value if no such element exists.

Returns

(*Observable*): An observable sequence that contains elements from the input sequence that satisfy the condition.

Example

No Match

Without a predicate

With a predicate

```
/* With a default value */
var source = Rx.Observable.range(0, 10)
  .first({
    predicate: function (x, idx, obs) { return x > 10; },
    defaultValue: 42
  });


var subscription = source.subscribe(
  function (x) {
    console.log('Next: %s', x);
  },
  function (err) {
    console.log('Error: %s', err);
  },
  function () {
    console.log('Completed');
  });

// => Next: 42
// => Completed
```

Rx.Observable.prototype.flatMap(selector, [resultSelector]) . Rx.Observable.prototype.selectMany(selector, [resultSelector])

One of the following:

Projects each element of an observable sequence to an observable sequence and merges the resulting observable sequences or Promises or array/iterable into one observable sequence.

```
source.selectMany(function (x, i) { return Rx.Observable.range(0, x); });
source.selectMany(function (x, i) { return Promise.resolve(x + 1); });
source.selectMany(function (x, i) { return [x, i]; });
```

Projects each element of an observable sequence or Promise to an observable sequence, invokes the result selector for the source element and each of the corresponding inner sequence's elements, and merges the results into one observable sequence.

```
source.selectMany(function (x, i) { return Rx.Observable.range(0, x); }, function (x, y,
source.selectMany(function (x, i) { return Promise.resolve(x + i); }, function (x, y, ix,
source.selectMany(function (x, i) { return [x, i]; }, function (x, y, ix, iy) { return x
```

Projects each element of the source observable sequence to the other observable sequence or Promise, or array/iterable and merges the resulting observable sequences into one observable sequence.

```
source.selectMany(Rx.Observable.of(1,2,3));
source.selectMany(Promise.resolve(42));
source.selectMany([1,2,3]);
```

Arguments

1. `selector` (*Function* | *Iterable* | *Promise*): An Object to project to the sequence or a transform function to apply to each element or an observable sequence to project each element from the source sequence onto. The selector is called with the following information:

   i. the value of the element

  ii. the index of the element

  iii. the Observable object being subscribed

2. `[resultSelector]` (*Function*): A transform function to apply to each element of the intermediate sequence. The resultSelector is called with the following information:

  i. the value of the outer element

  ii. the value of the inner element

  iii. the index of the outer element

  iv. the index of the inner element

Returns

(*Observable*): An observable sequence whose elements are the result of invoking the one-to-many transform function collectionSelector on each element of the input sequence and then mapping each of those sequence elements and their corresponding source element to a result element.

Example

```javascript
/* Using a promise */
var source = Rx.Observable.of(1,2,3,4)
  .selectMany(function (x, i) {
    return Promise.resolve(x + i);
  });

var subscription = source.subscribe(
  function (x) {
    console.log('Next: ' + x);
  },
  function (err) {
    console.log('Error: ' + err);
  },
  function () {
    console.log('Completed');
  });

// => Next: 1
// => Next: 3
// => Next: 5
// => Next: 7
// => Completed

/* Using an array */
Rx.Observable.of(2, 3, 5).selectMany(
  function(x) {
    // Return x^2, x^3 and x^4
    return [
```

```
      x * x,
      x * x * x,
      x * x * x * x
    ];
  },

  function(outer, inner, outerIndex, innerIndex) {
    return { outer : outer, inner : inner, outerIdx : outerIndex, innerIdx : innerIndex }
  }
).subscribe(
  function(next) {
    console.log(
      'Outer: ' + next.outer + ', Inner: ' + next.inner +
      ', InnerIndex: ' + next.innerIdx + ', OuterIndex: ' + next.outerIdx
    );
  },

  function() {
    console.log('Completed');
  }
);

//=> Outer: 2, Inner: 4, InnerIndex : 0, OuterIndex : 0
//=> Outer: 2, Inner: 8, InnerIndex : 1, OuterIndex : 0
//=> Outer: 2, Inner: 16, InnerIndex : 2, OuterIndex : 0
//=> Outer: 3, Inner: 9, InnerIndex : 0, OuterIndex : 1
//=> Outer: 3, Inner: 27, InnerIndex : 1, OuterIndex : 1
//=> Outer: 3, Inner: 81, InnerIndex : 2, OuterIndex : 1
//...etc
//=> Completed
```

Rx.Observable.prototype.concatMap(selector, [resultSelector]) . Rx.Observable.prototype.flatMapConcat(selector , [resultSelector])

This is an alias for the `flatMapConcat` method. This can be one of the following:

Projects each element of an observable sequence to an observable sequence and concatenates the resulting observable sequences or Promises or array/iterable into one observable sequence.

```
source.concatMap(function (x, i) { return Rx.Observable.range(0, x); });
source.concatMap(function (x, i) { return Promise.resolve(x + 1); });
source.concatMap(function (x, i) { return [x, i]; });
```

Projects each element of an observable sequence or Promise to an observable sequence, invokes the result selector for the source element and each of the corresponding inner sequence's elements, and concatenates the results into one observable sequence.

```
source.concatMap(function (x, i) { return Rx.Observable.range(0, x); }, function (x, y, i
source.concatMap(function (x, i) { return Promise.resolve(x + i); }, function (x, y, ix,
source.concatMap(function (x, i) { return [x, i];  }, function (x, y, ix, iy) { return x
```

Projects each element of the source observable sequence to the other observable sequence or Promise or array/iterable and merges the resulting observable sequences into one observable sequence.

```
source.concatMap(Rx.Observable.of(1,2,3));
source.concatMap(Promise.resolve(42));
source.concatMap([1,2,3]);
```

Arguments

1. `selector` (*Function* | *Iterable* | *Promise*): An Object to project to the sequence or a transform function to apply to each element or an observable sequence to project each element from the source sequence onto. The selector is called with the following information:

   i. the value of the element

      ii. the index of the element

     iii. the Observable object being subscribed

2. `[resultSelector]` *(Function)*: A transform function to apply to each element of the intermediate sequence. The resultSelector is called with the following information:

       i. the value of the outer element

      ii. the value of the inner element

     iii. the index of the outer element

     iv. the index of the inner element

3. `[thisArg]` *(Any)*: If `resultSelector` is not `Function`, Object to use as `this` when executing `selector`.

Returns

(Observable): An observable sequence whose elements are the result of invoking the one-to-many transform function collectionSelector on each element of the input sequence and then mapping each of those sequence elements and their corresponding source element to a result element.

Example

Using a promise

Using an array

```
var source = Rx.Observable.of(1,2,3)
  .concatMap(
    function (x, i) { return [x,i]; },
    function (x, y, ix, iy) { return x + y + ix + iy; }
  );

var subscription = source.subscribe(
    function (x) {
        console.log('Next: %s', x);
    },
    function (err) {
        console.log('Error: %s', err);
    },
    function () {
        console.log('Completed');
    });

// => Next: 2
// => Next: 2
// => Next: 5
// => Next: 5
// => Next: 8
// => Next: 8
// => Completed
```

```
 Rx.Observable.prototype.flatMapFirst(selector,
[thisArg]) , Rx.Observable.prototype.selectSwit
chFirst(selector, [thisArg])
```

Transform the items emitted by an Observable into Observables, and mirror those items emitted by the most-recently transformed Observable.

The `flatMapFirst` operator is similar to the `flatMap` and `concatMap` methods described above, however, rather than emitting all of the items emitted by all of the Observables that the operator generates by transforming items from the source `Observable`, `flatMapFirst` instead propagates the first `Observable` exclusively until it completes before it begins subscribes to the next `Observable`. Observables that come before the current `Observable` completes will be dropped and will not propagate.

Arguments

1. `selector` (*Function*): A transform function to apply to each source element. The callback has the following information:
   i. the value of the element
   ii. the index of the element
   iii. the Observable object being subscribed
2. `[thisArg]` (*Any*): Object to use as `this` when executing the predicate.

Returns

(*Observable*): An Observable sequence that is the result of concatenating non-overlapping items emitted by an Observable of Observables.

Example

```javascript
//Generate an event every 100 milliseconds
var source = Rx.Observable.generateWithRelativeTime(
    0,
    function(x) {return x < 5; },
    function(x) {return x + 1; },
    function(x) {return x; },
    function(x) {return 100; })
  .flatMapFirst(function(value) {
    //Observable takes 150 milliseconds to complete
    return Rx.Observable.timer(150).map(value);
  });

var subscription = source.subscribe(
  function (x) {
    console.log('Next: %d', x);
  },
  function (err) {
    console.log('Error: %s', err);
  },
  function () {
    console.log('Completed');
  });

// Next: 0
// Next: 2
// Next: 4
// Completed
```

`Rx.Observable.prototype.flatMapLatest(selector, [thisArg])`

Transform the items emitted by an Observable into Observables, and mirror those items emitted by the most-recently transformed Observable.

The `flatMapLatest` operator is similar to the `flatMap` and `concatMap` methods described above, however, rather than emitting all of the items emitted by all of the Observables that the operator generates by transforming items from the source `Observable`, `switchMap` instead emits items from each such transformed `Observable` only until the next such `Observable` is emitted, then it ignores the previous one and begins emitting items emitted by the new one.

Arguments

1. `selector` (*Function*): A transform function to apply to each source element. The callback has the following information:
    i. the value of the element
    ii. the index of the element
    iii. the Observable object being subscribed
2. `[thisArg]` (*Any*): Object to use as `this` when executing the predicate.

Returns

(*Observable*): An observable sequence which transforms the items emitted by an Observable into Observables, and mirror those items emitted by the most-recently transformed Observable.

Example

```
var source = Rx.Observable
  .range(1, 3)
  .flatMapLatest(function(x) {
    return Rx.Observable.from([x + 'a', x + 'b']);
  });

var subscription = source.subscribe(
  function (x) {
    console.log('Next: %s', x);
  },
  function (err) {
    console.log('Error: %s', err);
  },
  function () {
    console.log('Completed');
  });

// Next: 1a
// Next: 2a
// Next: 3a
// Next: 3b
// Completed
```

```
Rx.Observable.prototype.flatMapObserver(onNext
, onError, onCompleted, [thisArg]) ,
 Rx.Observable.prototype.selectManyObserver(onN
ext, onError, onCompleted, [thisArg])
```

Projects each notification of an observable sequence to an observable sequence and merges the resulting observable sequences into one observable sequence.

Arguments

1. `onNext` (*Function*): A transform function to apply to each element. The selector is called with the following information:
   i. the value of the element
   ii. the index of the element
2. `onError` (*Function*): A transform function to apply when an error occurs in the source sequence.
3. `onCompleted` (*Function*): A transform function to apply when the end of the source sequence is reached.
4. `[thisArg]` (*Any*): Object to use as `this` when executing the transform functions.

Returns

(*Observable*): An observable sequence whose elements are the result of invoking the one-to-many transform function corresponding to each notification in the input sequence.

Example

```
var source = Rx.Observable.range(1, 3)
    .flatMapObserver(
        function (x, i) {
            return Rx.Observable.repeat(x, i);
        },
        function (err) {
            return Rx.Observable.return(42);
        },
        function () {
            return Rx.Observable.empty();
        });

var subscription = source.subscribe(
    function (x) {
        console.log('Next: ' + x);
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

// => Next: 2
// => Next: 3
// => Next: 3
// => Completed
```

`Rx.Observable.prototype.flatMapWithMaxConcurrent(maxConcurrent, selector, [resultSelector], [thisArg])` .
`Rx.Observable.prototype.selectWithMaxConcurrent(maxConcurrent, selector, [resultSelector], [thisArg])`

This is an alias for the `selectWithMaxConcurrent` method. This can be one of the following:

Projects each element of the source observable sequence to the other observable sequence and merges the resulting observable sequences into one observable sequence with the given concurrency limit.

```
source.flatMapWithMaxConcurrent(10, function (x, i) { return Rx.Observable.range(0, x); }
source.flatMapWithMaxConcurrent(1, function (x, i) { return Promise.resolve(x + 1); });
source.flatMapWithMaxConcurrent(1, function (x, i) { return [x, i]; });
```

Projects each element of an observable sequence or Promise to an observable sequence, invokes the result selector for the source element and each of the corresponding inner sequence's elements, and merges the results into one observable sequence with the given concurrency limit.

```
source.flatMapWithMaxConcurrent(10, function (x, i) { return Rx.Observable.range(0, x); }
source.flatMapWithMaxConcurrent(1, function (x, i) { return Promise.resolve(x + i); }, fu
source.flatMapWithMaxConcurrent(1, function (x, i) { return [x, i];  }, function (x, y, i
```

Projects each element of the source observable sequence to the other observable sequence or Promise or array/iterable and merges the resulting observable sequences into one observable sequence with the given max concurrency limit.

```
source.flatMapWithMaxConcurrent(1, Rx.Observable.of(1,2,3));
source.flatMapWithMaxConcurrent(1, Promise.resolve(42));
source.flatMapWithMaxConcurrent(1, [1,2,3]);
```

Arguments

1. `maxConcurrent` (*Number*): Maximum number of inner observable sequences being subscribed to concurrently.

2. `selector` (*Function* | *Iterable* | *Promise*): An Object to project to the sequence or a transform function to apply to each element or an observable sequence to project each element from the source sequence onto. The selector is called with the following information:

    i. the value of the element
    ii. the index of the element
    iii. the Observable object being subscribed

3. `[resultSelector]` (*Function*): A transform function to apply to each element of the intermediate sequence. The resultSelector is called with the following information:

    i. the value of the outer element
    ii. the value of the inner element
    iii. the index of the outer element
    iv. the index of the inner element

4. `[thisArg]` (*Any*): If `resultSelector` is not `Function`, Object to use as `this` when executing `selector` .

Returns

(*Observable*): An observable sequence whose elements are the result of invoking the one-to-many transform function collectionSelector on each element of the input sequence and then mapping each of those sequence elements and their corresponding source element to a result element.

Example

```javascript
var source = Rx.Observable.range(0, 5)
  .flatMapWithMaxConcurrent(2, function (x, i) {
    return Rx.Observable
      .interval(100)
      .take(x).map(function() { return i; });
  });

var subscription = source.subscribe(
  function (x) {
    console.log('Next: %s', x);
  },
  function (err) {
    console.log('Error: %s', err);
  },
  function () {
    console.log('Completed');
  });
```

```
// => Next: 1
// => Next: 2
// => Next: 3
// => Next: 2
// => Next: 3
// => Next: 4
// => Next: 3
// => Next: 4
// => Next: 4
// => Next: 4
// => Completed

/* Using a promise */
var source = Rx.Observable.of(1,2,3,4)
  .flatMapWithMaxConcurrent(1, function (x, i) {
    return Promise.resolve(x + i);
  });

var subscription = source.subscribe(
  function (x) {
    console.log('Next: %s', x);
  },
  function (err) {
    console.log('Error: %s', err);
  },
  function () {
    console.log('Completed');
  });

// => Next: 1
// => Next: 3
// => Next: 5
// => Next: 7
// => Completed

/* Using an array */
var source = Rx.Observable.of(1,2,3)
  .flatMapWithMaxConcurrent(
    1,
    function (x, i) { return [x,i]; },
    function (x, y, ix, iy) { return x + y + ix + iy; }
  );

var subscription = source.subscribe(
  function (x) {
    console.log('Next: %s', x);
  },
  function (err) {
    console.log('Error: %s', err);
  },
  function () {
    console.log('Completed');
  });
```

```
// => Next: 2
// => Next: 2
// => Next: 5
// => Next: 5
// => Next: 8
// => Next: 8
// => Completed
```

`Rx.Observable.prototype.forkJoin(...args, [resultSelector])`

Runs all observable sequences in parallel and collect their last elements.

Arguments

1. `args` : *(Arguments | Array)* - An array or arguments of Observable sequences or Promises to collect the last elements for.
2. `resultSelector` : *(Function)* - The result selector from all the values produced. If not specified, `forkJoin` will return the results as an array.

Returns

(Observable): An observable sequence with an array collecting the last elements of all the input sequences or the result of the result selector if specified.

Example

```javascript
// Without a selector
var source = Rx.Observable.just(42).forkJoin(
  Rx.Observable.range(0, 3));

var subscription = source.subscribe(
  function (x) {
    console.log('Next: %s', x);
  },
  function (err) {
    console.log('Error: %s', err);
  },
  function () {
    console.log('Completed');
  });

// => Next: [42, 2]
// => Completed
```

`Rx.Observable.prototype.groupBy(keySelector, [elementSelector], [comparer])`

Groups the elements of an observable sequence according to a specified key selector function and comparer and selects the resulting elements by using a specified function.

Arguments

1. `keySelector` (*Function*): A function to extract the key for each element.
2. `[elementSelector]` (*Function*): A function to map each source element to an element in an observable group.
3. `[comparer]` (*Any*): Used to compare objects. If not specified, the default comparer is used.

Returns

(*Observable*): A sequence of observable groups, each of which corresponds to a unique key value, containing all elements that share that same key value.

Example

`Rx.Observable.prototype.groupByUntil(keySelector, [elementSelector], durationSelector, [comparer])`

Groups the elements of an observable sequence according to a specified key selector function and comparer and selects the resulting elements by using a specified function.

Arguments

1. `keySelector` (*Function*): A function to extract the key for each element.
2. `[elementSelector]` (*Function*): A function to map each source element to an element in an observable group.
3. `durationSelector` (*Function*): A function to signal the expiration of a group.
4. `[comparer]` (*Any*): Used to compare objects. If not specified, the default comparer is used.

Returns

(*Observable*): A sequence of observable groups, each of which corresponds to a unique key value, containing all elements that share that same key value.

If a group's lifetime expires, a new group with the same key value can be created once an element with such a key value is encoutered.

Example

Rx.Observable.prototype.groupJoin(right, leftDurationSelector, rightDurationSelector, resultSelector)

Correlates the elements of two sequences based on overlapping durations, and groups the results.

Arguments

1. `right` (*Observable*): The right observable sequence to join elements for.
2. `leftDurationSelector` (*Function*): A function to select the duration (expressed as an observable sequence) of each element of the left observable sequence, used to determine overlap.
3. `rightDurationSelector` (*Function*): A function to select the duration (expressed as an observable sequence) of each element of the right observable sequence, used to determine overlap.
4. `resultSelector` (*Any*): A function invoked to compute a result element for any element of the left sequence with overlapping elements from the right observable sequence. It has the following arguments
   i. (*Any*) An element of the left sequence.
   ii. (*Observable*) An observable sequence with elements from the right sequence that overlap with the left sequence's element.

Returns

(*Observable*): An observable sequence that contains result elements computed from source elements that have an overlapping duration.

Example

`Rx.Observable.prototype.ignoreElements()`

Ignores all elements in an observable sequence leaving only the termination messages.

Returns

(`Observable`): An empty observable sequence that signals termination, successful or exceptional, of the source sequence.

Example

Rx.Observable.prototype.includes(searchElement, [fromIndex])

Determines whether an observable sequence includes a specified element with an optional from index.

Arguments

1. `searchElement` (*Any*): The value to locate in the source sequence.
2. `[fromIndex]` (*Number*): The index to start the search. If not specified, defaults to 0.

Returns

(*Observable*): An observable sequence containing a single element determining whether the source sequence includes an element that has the specified value with an optional from index.

Example

```
/* Without an index */
var source = Rx.Observable.of(42)
  .includes(42);

var subscription = source.subscribe(
  function (x) {
    console.log('Next: %s', x);
  },
  function (err) {
    console.log('Error: %s', err);
  },
  function () {
      console.log('Completed');
  });

// => Next: true
// => Completed

/* With an index */
var source = Rx.Observable.of(1,2,3)
  .includes(2, 1);

var subscription = source.subscribe(
  function (x) {
    console.log('Next: %s', x);
  },
  function (err) {
    console.log('Error: %s', err);
  },
  function () {
    console.log('Completed');
  });

// => Next: true
// => Completed
```

`Rx.Observable.prototype.indexOf(searchElement, fromIndex)`

Returns the first index at which a given element can be found in the observable sequence, or -1 if it is not present.

Arguments

1. `searchElement` (*Any*): An element to locate in the array.
2. `fromIndex` *(Number)*: The index to start the search. If not specified, defaults to 0.

Returns

(*Observable*): An observable sequence containing the first index at which a given element can be found in the observable sequence, or -1 if it is not present.

Example

Rx.Observable.prototype.isEmpty()

Determines whether an observable sequence is empty.

Returns

(*Observable*): An observable sequence containing a single element determining whether the source sequence is empty.

Example

Not empty

Empty

Rx.Observable.prototype.isEmpty()

Rx.Observable.prototype.join(right, leftDurationSelector, rightDurationSelector, resultSelector)

Correlates the elements of two sequences based on overlapping durations.

Arguments

1. `right` (*Observable*): The right observable sequence to join elements for.
2. `leftDurationSelector` (*Function*): A function to select the duration (expressed as an observable sequence) of each element of the left observable sequence, used to determine overlap.
3. `rightDurationSelector` (*Function*): A function to select the duration (expressed as an observable sequence) of each element of the right observable sequence, used to determine overlap.
4. `resultSelector` (*Any*): A function invoked to compute a result element for any two overlapping elements of the left and right observable sequences. The parameters are as follows:
    i. (*Any*) Element from the left source for which the overlap occurs.
    ii. (*Any*) Element from the right source for which the overlap occurs.

Returns

(*Observable*): An observable sequence that contains result elements computed from source elements that have an overlapping duration.

Example

`Rx.Observable.prototype.last([predicate], [thisArg])`

</rx-marbles>

Returns the last element of an observable sequence that satisfies the condition in the predicate if specified, else the last element.

Arguments

1. `predicate` (*Function*): A predicate function to evaluate for elements in the source sequence. The callback is called with the following information:
   i. the value of the element
   ii. the index of the element
   iii. the Observable object being subscribed
2. `[thisArg]` (*Any*): Object to use as `this` when executing the predicate.

Returns

(*Observable*): Sequence containing the last element in the observable sequence that satisfies the condition in the predicate.

Example

No Match

Without predicate

With predicate

`Rx.Observable.prototype.let(func)`

Returns an observable sequence that is the result of invoking the selector on the source sequence, without sharing subscriptions.

This operator allows for a fluent style of writing queries that use the same sequence multiple times. There is an alias of `letBind` for browsers older than IE 9.

Arguments

1. `func` (*Function*): Selector function which can use the source sequence as many times as needed, without sharing subscriptions to the source sequence.

Returns

(*Observable*): An observable sequence that contains the elements of a sequence produced by multicasting the source sequence within a selector function.

Example

`Rx.Observable.prototype.map(selector, [thisArg])`

</rx-marbles>

Projects each element of an observable sequence into a new form by incorporating the element's index. This is an alias for the `select` method.

Arguments

1. `selector` (*Function*): Transform function to apply to each source element. The selector is called with the following information:
   i. the value of the element
   ii. the index of the element
   iii. the Observable object being subscribed
2. `[thisArg]` (*Any*): Object to use as `this` when executing the predicate.

Returns

(*Observable*): An observable sequence which results from the comonadic bind operation.

Example

`Rx.Observable.prototype.materialize()`

Materializes the implicit notifications of an observable sequence as explicit notification values.

Returns

(`Observable`): An observable sequence containing the materialized notification values from the source sequence.

Example

Rx.Observable.prototype.max([comparer])

</rx-marbles>

Returns the maximum value in an observable sequence according to the specified comparer.

Arguments

1. `[comparer]` (*Function*): Comparer used to compare elements.

Returns

(*Observable*): An observable sequence containing a single element with the maximum element in the source sequence.

Example

Without comparer

With a comparer

Rx.Observable.prototype.maxBy(keySelector, [comparer])

Returns the maximum value in an observable sequence according to the specified comparer.

Arguments

1. `keySelector` (*Function*): Key selector function.
2. `[comparer]` (*Function*): Comparer used to compare elements.

Returns

(*Observable*): An observable sequence containing a list of zero or more elements that have a maximum key value.

Example

`Rx.Observable.prototype.merge(maxConcurrent | other)`

</rx-marbles>

Merges an observable sequence of observable sequences into an observable sequence, limiting the number of concurrent subscriptions to inner sequences. Or merges two observable sequences into a single observable sequence.

Arguments

1. `maxConcurrent` *(Function)*: Maximum number of inner observable sequences being subscribed to concurrently.
2. `other` *(Observable)*: The second observable sequence to merge into the first.

Returns

(Observable): The observable sequence that merges the elements of the inner sequences.

Example

Merge two sequences

Use max concurrency

[`Rx.Observable.prototype.mergeAll()` ,

`Rx.Observable.prototype.mergeObservable()`](https://github.com/Reactive-Extensions/RxJS/blob/master/src/core/linq/observable/mergeall.js)

Merges an observable sequence of observable sequences into an observable sequence.

Returns

(*Observable*): The observable sequence that merges the elements of the inner sequences.

Example

Rx.Observable.prototype.min([comparer])

</rx-marbles>

Returns the minimum element in an observable sequence according to the optional comparer else a default greater than less than check.

Arguments

1. `[comparer]` (*Function*): Comparer used to compare elements.

Returns

(*Observable*): An observable sequence containing a single element with the minimum element in the source sequence.

Example

Without comparer

With a comparer

Rx.Observable.prototype.minBy(keySelector, [comparer])

Returns the elements in an observable sequence with the minimum key value according to the specified comparer.

Arguments

1. `keySelector` (*Function*): Key selector function.
2. `[comparer]` (*Function*): Comparer used to compare elements.

Returns

(*Observable*): An observable sequence containing a list of zero or more elements that have a minimum key value.

Example

Rx.Observable.prototype.multicast(subject | subjectSelector, [selector])

Multicasts the source sequence notifications through an instantiated subject into all uses of the sequence within a selector function. Each subscription to the resulting sequence causes a separate multicast invocation, exposing the sequence resulting from the selector function's invocation. For specializations with fixed subject types, see `publish` , `share` , `publishValue` , `shareValue` , `publishLast` , `replay` , and `shareReplay` .

Arguments

1. `subjectSelector` (*Function*): Factory function to create an intermediate subject through which the source sequence's elements will be multicast to the selector function.
2. `subject` *(Subject)*: Subject to push source elements into.
3. `[selector]` (*Function*): Optional selector function which can use the multicasted source sequence subject to the policies enforced by the created subject. Specified only if `subjectSelector` is provided.

Returns

(*Observable*): An observable sequence that contains the elements of a sequence produced by multicasting the source sequence within a selector function.

Example

Rx.Observable.prototype.observeOn(scheduler)

Wraps the source sequence in order to run its observer callbacks on the specified scheduler.

This only invokes observer callbacks on a scheduler. In case the subscription and/or unsubscription actions have side-effects that require to be run on a scheduler, use subscribeOn.

Arguments

1. `scheduler` (*Scheduler*): Scheduler to notify observers on.

Returns

(*Observable*): The source sequence whose observations happen on the specified scheduler.

Example

Rx.Observable.prototype.onErrorResumeNext(second)

Continues an observable sequence that is terminated normally or by an exception with the next observable sequence or Promise.

Arguments

1. `second` (*Observable* | *Promise*): Second observable sequence used to produce results after the first sequence terminates.

Returns

(*Observable*): An observable sequence that concatenates the first and second sequence, even if the first sequence terminates exceptionally.

Example

`Rx.Observable.prototype.pairwise()`

Triggers on the second and subsequent triggerings of the input observable. The Nth triggering of the input observable passes the arguments from the N-1th and Nth triggering as a pair.

Returns

(`Observable`): An observable that triggers on successive pairs of observations from the input observable as an array.

Example

Rx.Observable.prototype.partition(predicate, [thisArg])

Returns two observables which partition the observations of the source by the given function. The first will trigger observations for those values for which the predicate returns true. The second will trigger observations for those values where the predicate returns false. The predicate is executed once for each subscribed observer. Both also propagate all error observations arising from the source and each completes when the source completes.

Arguments

1. `predicate` (*Function*): Selector function to invoke for each produced element, resulting in another sequence to which the selector will be invoked recursively again. The callback is called with the following information:
   i. the value of the element
   ii. the index of the element
   iii. the Observable object being subscribed
2. `[thisArg]` (*Any*): Object to use as `this` when executing the predicate.

Returns

(*Array*): An array of observables. The first triggers when the predicate returns true, and the second triggers when the predicate returns false.

Example

Rx.Observable.prototype.pausable(pauser)

</rx-marbles>

Pauses the underlying observable sequence based upon the observable sequence which yields true/false. Note that this only works on hot observables.

Arguments

1. `pauser` *(Rx.Subject)*: The observable sequence used to pause the underlying sequence.

Returns

(`Observable`): The observable sequence which is paused based upon the pauser.

Example

`Rx.Observable.prototype.pausableBuffered(pauser)`

</rx-marbles>

Pauses the underlying observable sequence based upon the observable sequence which yields true/false, and yields the values that were buffered while paused. Note that this only works on hot observables.

Arguments

1. `pauser` *(Rx.Subject)*: The observable sequence used to pause the underlying sequence.

Returns

(`Observable`): The observable sequence which is paused based upon the pauser.

Example

`Rx.Observable.prototype.pipe(dest)`

Pipes the existing Observable sequence into a Node.js Stream.

Arguments

1. `dest` (*Stream*): dest The destination Node.js stream.

Returns

(*Stream*): The destination stream.

Example

Rx.Observable.prototype.pluck(property)

Projects each element of an observable sequence into a new form by incorporating the element's index.

Arguments

1. `property` (*String*): The property to pluck.

Returns

(*Observable*): Returns a new Observable sequence of property values.

Example

`Rx.Observable.prototype.publish([selector])`

Returns an observable sequence that is the result of invoking the selector on a connectable observable sequence that shares a single subscription to the underlying sequence.

This operator is a specialization of `multicast` using a regular `Rx.Subject`.

Arguments

1. `[selector]` (*Function*): Selector function which can use the multicasted source sequence as many times as needed, without causing multiple subscriptions to the source sequence. Subscribers to the given source will receive all notifications of the source from the time of the subscription on.

Returns

(ConnectableObservable): An observable sequence that contains the elements of a sequence produced by multicasting the source sequence within a selector function.

Example

Without publish

With publish

```
Rx.Observable.prototype.publishLast([selector])
```

Returns an observable sequence that is the result of invoking the selector on a connectable observable sequence that shares a single subscription to the underlying sequence containing only the last notification.

This operator is a specialization of `multicast` using a `Rx.AsyncSubject` .

Arguments

1. `[selector]` *(Function)*: Selector function which can use the multicasted source sequence as many times as needed, without causing multiple subscriptions to the source sequence. Subscribers to the given source will only receive the last notification of the source.

Returns

(ConnectableObservable): An observable sequence that contains the elements of a sequence produced by multicasting the source sequence within a selector function.

Example

`Rx.Observable.prototype.publishValue([selector])`

Returns an observable sequence that is the result of invoking the selector on a connectable observable sequence that shares a single subscription to the underlying sequence and starts with initialValue.

This operator is a specialization of `multicast` using a `Rx.BehaviorSubject` .

Arguments

1. `[selector]` (*Function*): Selector function which can use the multicasted source sequence as many times as needed, without causing multiple subscriptions to the source sequence. Subscribers to the given source will receive immediately receive the initial value, followed by all notifications of the source from the time of the subscription on.

Returns

(ConnectableObservable): An observable sequence that contains the elements of a sequence produced by multicasting the source sequence within a selector function.

Example

`Rx.Observable.prototype.reduce(accumulator, [seed])`

</rx-marbles>

Applies an accumulator function over an observable sequence, returning the result of the aggregation as a single element in the result sequence. The specified seed value is used as the initial accumulator value.

For aggregation behavior with incremental intermediate results, see the `scan` method.

Arguments

1. `accumulator` (*Function*): An accumulator function to be invoked on each element.
2. `[seed]` (*Any*): The initial accumulator value.

Returns

(*Observable*): An observable sequence containing a single element with the final accumulator value.

Example

Rx.Observable.prototype.repeat(repeatCount)

Repeats the observable sequence a specified number of times. If the repeat count is not specified, the sequence repeats indefinitely.

Arguments

1. `repeatCount` (*Number*): Number of times to repeat the sequence. If not provided, repeats the sequence indefinitely.

Returns

(*Observable*): The observable sequence producing the elements of the given sequence repeatedly.

Example

Rx.Observable.prototype.repeat(repeatCount)

Rx.Observable.prototype.replay([selector], [bufferSize], [window], [scheduler])

Returns an observable sequence that is the result of invoking the selector on a connectable observable sequence that shares a single subscription to the underlying sequence replaying notifications subject to a maximum time length for the replay buffer.

This operator is a specialization of `multicast` using a `Rx.ReplaySubject` .

Arguments

1. `[selector]` (*Function*): Selector function which can use the multicasted source sequence as many times as needed, without causing multiple subscriptions to the source sequence. Subscribers to the given source will receive all the notifications of the source subject to the specified replay buffer trimming policy.
2. `[bufferSize]` (*Number*): Maximum element count of the replay buffer.
3. `[window]` (*Number*): Maximum time length of the replay buffer in milliseconds.
4. `[scheduler]` (*Scheduler*): Scheduler where connected observers within the selector function will be invoked on.

Returns

(*Observable*): An observable sequence that contains the elements of a sequence produced by multicasting the source sequence within a selector function.

Example

`Rx.Observable.prototype.retry([retryCount])`

Projects each element of an observable sequence into a new form by incorporating the element's index. This is an alias for the `select` method.

Arguments

1. `[retryCount]` *(Number)*: Number of times to retry the sequence. If not provided, retry the sequence indefinitely.

Returns

(Observable): An observable sequence producing the elements of the given sequence repeatedly until it terminates successfully.

Example

Rx.Observable.prototype.retryWhen(notifier)

Repeats the source observable sequence on error when the notifier emits a next value. If the source observable errors and the notifier completes, it will complete the source sequence

Arguments

1. `notificationHandler` (*Function*): A handler that is passed an observable sequence of errors raised by the source observable and returns and observable that either continues, completes or errors. This behavior is then applied to the source observable.

Returns

(*Observable*): An observable sequence producing the elements of the given sequence repeatedly until it terminates successfully or is notified to error or complete.

Example: delayed retry

```javascript
var count = 0;

var source = Rx.Observable.interval(1000)
    .map(function(n) {
        if(n === 2) {
            throw 'ex';
        }
        return n;
    })
    .retryWhen(function(errors) {
        return errors.delay(200);
    })
    .take(6);

var subscription = source.subscribe(
    function (x) {
        console.log('Next: ' + x);
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

// => Next: 0
// => Next: 1
// 200 ms pass
// => Next: 0
// => Next: 1
// 200 ms pass
// => Next: 0
// => Next: 1
// => Error: 'ex'
```

Example: Erroring an observable after 2 failures

```
var count = 0;

var source = Rx.Observable.interval(1000)
    .map(function(n) {
        if(n === 2) {
            throw 'ex';
        }
        return n;
    })
    .retryWhen(function(errors) {
        return errors.scan(0, function(errorCount, err) {
            if(errorCount >= 2) {
                throw err;
            }
            return errorCount + 1;
        });
    });

var subscription = source.subscribe(
    function (x) {
        console.log('Next: ' + x);
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

// => Next: 0
// => Next: 1
// => Next: 0
// => Next: 1
// => Error: 'ex'
```

Example: Completing an observable after 2 failures

```
var count = 0;

var source = Rx.Observable.interval(1000)
    .map(function(n) {
        if(n === 2) {
            throw 'ex';
        }
        return n;
    })
    .retryWhen(function(errors) {
        return errors.scan(0, function(errorCount, err) {
            return errorCount + 1;
        }).takeWhile(function(errorCount) {
            return errorCount < 2;
        });
    });

var subscription = source.subscribe(
    function (x) {
        console.log('Next: ' + x);
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

// => Next: 0
// => Next: 1
// => Next: 0
// => Next: 1
// => Completed
```

An incrememntal back-off strategy for handling errors:

```
Rx.Observable.create(function (o) {
    console.log("subscribing");
    o.onError(new Error("always fails"));
}).retryWhen(function (attempts) {
    return Rx.Observable.range(1, 3).zip(attempts, function (i) { return i; }).flatMap(
        console.log("delay retry by " + i + " second(s)");
        return Rx.Observable.timer(i * 1000);
    });
}).subscribe();

/*
subscribing
delay retry by 1 second(s)
subscribing
delay retry by 2 second(s)
subscribing
delay retry by 3 second(s)
subscribing
*/
```

`Rx.Observable.prototype.scan([seed], accumulator)`

</rx-marbles>

Applies an accumulator function over an observable sequence and returns each intermediate result. The optional seed value is used as the initial accumulator value.

For aggregation behavior with no intermediate results, see `Rx.Observable.aggregate` .

Arguments

1. `[seed]` (*Any*): The initial accumulator value.
2. `accumulator` (*Function*): An accumulator function to be invoked on each element.

Returns

(*Observable*): An observable sequence which results from the comonadic bind operation.

Example

Without a seed

With a seed

Rx.Observable.prototype.sequenceEqual(second, [comparer])

</rx-marbles>

Determines whether two sequences are equal by comparing the elements pairwise using a specified equality comparer.

Arguments

1. `second` (*Observable* | *Promise* | *Array*): Second observable sequence, Promise or array to compare.
2. `[comparer]` (*Function*): Comparer used to compare elements of both sequences.

Returns

(*Observable*): An observable sequence that contains a single element which indicates whether both sequences are of equal length and their corresponding elements are equal according to the specified equality comparer.

Example

`Rx.Observable.prototype.share()`

Returns an observable sequence that shares a single subscription to the underlying sequence.

This operator is a specialization of `publish` which creates a subscription when the number of observers goes from zero to one, then shares that subscription with all subsequent observers until the number of observers returns to zero, at which point the subscription is disposed.

Returns

(`Observable`): An observable sequence that contains the elements of a sequence produced by multicasting the source sequence.

Example

Without share

With share

`Rx.Observable.prototype.shareReplay([bufferSize], [window], [scheduler])`

Returns an observable sequence that shares a single subscription to the underlying sequence replaying notifications subject to a maximum time length for the replay buffer.

This operator is a specialization of `replay` that connects to the connectable observable sequence when the number of observers goes from zero to one, and disconnects when there are no more observers.

Arguments

1. `[bufferSize]` (*Number*): Maximum element count of the replay buffer.
2. `[window]` (*Number*): Maximum time length of the replay buffer in milliseconds.
3. `[scheduler]` (*Scheduler*): Scheduler where connected observers within the selector function will be invoked on.

Returns

(*Observable*): An observable sequence that contains the elements of a sequence produced by multicasting the source sequence within a selector function.

Example

Rx.Observable.prototype.shareValue(value)

Returns an observable sequence that shares a single subscription to the underlying sequence and starts with initialValue.

This operator is a specialization of `publishValue` which creates a subscription when the number of observers goes from zero to one, then shares that subscription with all subsequent observers until the number of observers returns to zero, at which point the subscription is disposed.

Returns

(`Observable`): An observable sequence that contains the elements of a sequence produced by multicasting the source sequence.

Example

`Rx.Observable.prototype.single([predicate], [thisArg])`

Returns the only element of an observable sequence that satisfies the condition in the optional predicate, and reports an exception if there is not exactly one element in the observable sequence.

Arguments

1. `[predicate]` (*Function*): A predicate function to evaluate for elements in the source sequence. The callback is called with the following information:
    i. the value of the element
    ii. the index of the element
    iii. the Observable object being subscribed
2. `[thisArg]` (*Any*): Object to use as `this` when executing the predicate.

Returns

(*Observable*): Sequence containing the single element in the observable sequence that satisfies the condition in the predicate.

Example

No Match

Without a predicate

With a predicate

More than one match

`Observable.prototype.singleInstance()`

Returns a "cold" observable that becomes "hot" upon first subscription, and goes "cold" again when all subscriptions to it are disposed.

At first subscription to the returned observable, the source observable is subscribed to. That source subscription is then shared amongst each subsequent simultaneous subscription to the returned observable.

When all subscriptions to the returned observable have completed, the source observable subscription is disposed of.

The first subscription after disposal starts again, subscribing one time to the source observable, then sharing that subscription with each subsequent simultaneous subscription.

Returns

(`Observable`): An observable sequence that stays connected to the source as long as there is at least one subscription to the observable sequence.

Example

```javascript
var interval = Rx.Observable.interval(1000);

var source = interval
    .take(2)
    .doAction(function (x) {
        console.log('Side effect');
    });

var single = source.singleInstance();

// two simultaneous subscriptions, lasting 2 seconds
single.subscribe(createObserver('SourceA'));
single.subscribe(createObserver('SourceB'));

setTimeout(function(){
    // resubscribe two times again, more than 5 seconds later,
    // long after the original two subscriptions have ended
    single.subscribe(createObserver('SourceC'));
    single.subscribe(createObserver('SourceD'));
}, 5000);

function createObserver(tag) {
    return Rx.Observer.create(
        function (x) {
            console.log('Next: ' + tag + x);
        },
        function (err) {
            console.log('Error: ' + err);
        },
        function () {
            console.log('Completed: ' + tag);
        });
}

// => Side effect
// => Next: SourceA0
// => Next: SourceB0
// => Side effect
// => Next: SourceA1
// => Next: SourceB1
// => Completed: SourceA
// => Completed: SourceB
// => Side effect
// => Next: SourceC0
// => Next: SourceD0
// => Side effect
// => Next: SourceC1
// => Next: SourceD1
// => Completed: SourceC
// => Completed: SourceD
```

Rx.Observable.prototype.skip(count)

</rx-marbles>

Bypasses a specified number of elements in an observable sequence and then returns the remaining elements.

Arguments

1. `count` (*Number*): The number of elements to skip before returning the remaining elements.

Returns

(*Observable*): An observable sequence that contains the elements that occur after the specified index in the input sequence.

Example

`Rx.Observable.prototype.skipLast(count)`

</rx-marbles>

Bypasses a specified number of elements at the end of an observable sequence.

This operator accumulates a queue with a length enough to store the first `count` elements. As more elements are received, elements are taken from the front of the queue and produced on the result sequence. This causes elements to be delayed.

Arguments

1. `count` (*Number*): Number of elements to bypass at the end of the source sequence.

Returns

(*Observable*): An observable sequence containing the source sequence elements except for the bypassed ones at the end.

Example

`Rx.Observable.prototype.skipLastWithTime(duration)`

Bypasses a specified number of elements at the end of an observable sequence.

This operator accumulates a queue with a length enough to store the first `count` elements. As more elements are received, elements are taken from the front of the queue and produced on the result sequence. This causes elements to be delayed.

Arguments

1. `duration` (*Number*): Duration for skipping elements from the end of the sequence.
2. `[scheduler=Rx.Scheduler.timeout]` (*Scheduler*): Scheduler to run the timer on. If not specified, defaults to timeout scheduler.

Returns

(*Observable*): An observable sequence with the elements skipped during the specified duration from the end of the source sequence.

Example

Rx.Observable.prototype.skipUntil(other)

</rx-marbles>

Returns the values from the source observable sequence only after the other observable sequence produces a value.

Arguments

1. `other` (*Observable* | *Promise*): The observable sequence or Promise that triggers propagation of elements of the source sequence.

Returns

(*Observable*): An observable sequence containing the elements of the source sequence starting from the point the other sequence triggered propagation.

Example

`Rx.Observable.prototype.skipUntilWithTime(startTime, [scheduler])`

Skips elements from the observable source sequence until the specified start time, using the specified scheduler to run timers.

Errors produced by the source sequence are always forwarded to the result sequence, even if the error occurs before the start time.

Arguments

1. `startTime` *(Date* | *Number)*: Time to start taking elements from the source sequence. If this value is less than or equal to current time, no elements will be skipped.
2. `[ scheduler = Rx.Scheduler.timeout ]` *(Scheduler)*: Scheduler to run the timer on. If not specified, defaults to Rx.Scheduler.timeout.

Returns

(Observable): An observable sequence with the elements skipped until the specified start time.

Example

Rx.Observable.prototype.skipWhile([predicate], [thisArg])

Bypasses elements in an observable sequence as long as a specified condition is true and then returns the remaining elements.

Arguments

1. `predicate` (*Function*): A function to test each source element for a condition. The callback is called with the following information:
    i. the value of the element
    ii. the index of the element
    iii. the Observable object being subscribed
2. `[thisArg]` (*Any*): Object to use as this when executing callback.

Returns

(*Observable*): An observable sequence that contains the elements from the input sequence starting at the first element in the linear series that does not pass the test specified by predicate.

Example

Rx.Observable.prototype.some([predicate], [thisArg])

</rx-marbles>

Determines whether any element of an observable sequence satisfies a condition if present, else if any items are in the sequence. There is an alias to this method called `any`.

Arguments

1. `predicate` (*Function*): A function to test each source element for a condition. The callback is called with the following information:
    i. the value of the element
    ii. the index of the element
    iii. the Observable object being subscribed
2. `[thisArg]` (*Any*): Object to use as this when executing callback.

Returns

(*Observable*): An observable sequence containing a single element determining whether all elements in the source sequence pass the test in the specified predicate.

Example

`Rx.Observable.prototype.startWith([scheduler] ...args)`

</rx-marbles>

Prepends a sequence of values to an observable sequence with an optional scheduler and an argument list of values to prepend.

Arguments

1. `[scheduler]` (*Scheduler*): Scheduler to execute the function.
2. `args` *(arguments)*: Values to prepend to the observable sequence.

Returns

(*Observable*): The source sequence prepended with the specified values.

Example

[`Rx.Observable.prototype.subscribe([observer] | [onNext], [onError], [onCompleted])` ,

`Rx.Observable.prototype.forEach([observer] | [onNext], [onError], [onCompleted])`]
(https://github.com/Reactive-Extensions/RxJS/blob/master/src/core/observable.js)

Prepends a sequence of values to an observable sequence with an optional scheduler and an argument list of values to prepend.

Arguments

1. `[observer]` *(Observer)*: The object that is to receive notifications.
2. `[onNext]` *(Function)*: Function to invoke for each element in the observable sequence.
3. `[onError]` *(Function)*: Function to invoke upon exceptional termination of the observable sequence.
4. `[onCompleted]` *(Function)*: Function to invoke upon graceful termination of the observable sequence.

Returns

(Disposable): The source sequence whose subscriptions and unsubscriptions happen on the specified scheduler.

Example

With no arguments

With an observer

Using functions

`Rx.Observable.prototype.subscribeOn(scheduler)`

Wraps the source sequence in order to run its subscription and unsubscription logic on the specified scheduler.

This only performs the side-effects of subscription and unsubscription on the specified scheduler. In order to invoke observer callbacks on a scheduler, use `observeOn`.

Arguments

1. `scheduler` (*Scheduler*): Scheduler to notify observers on.

Returns

(*Observable*): The source sequence whose observations happen on the specified scheduler.

Example

`Rx.Observable.prototype.subscribeOnCompleted(onCompleted, [thisArg])`

Subscribes a function to invoke upon graceful termination of the observable sequence.

Arguments

1. `onCompleted` (*Function*): Function to invoke upon graceful termination of the observable sequence.
2. `[thisArg]` (*Any*): Object to use as this when executing callback.

Returns

(Disposable): The source sequence whose subscriptions and unsubscriptions happen on the specified scheduler.

Example

Using functions

With a thisArg

`Rx.Observable.prototype.subscribeOnError(onError, [thisArg])`

Subscribes a function to invoke upon exceptional termination of the observable sequence.

Arguments

1. `onError` *(Function)*: Function to invoke upon exceptional termination of the observable sequence.
2. `[thisArg]` *(Any)*: Object to use as this when executing callback.

Returns

(Disposable): The source sequence whose subscriptions and unsubscriptions happen on the specified scheduler.

Example

Using functions

With a thisArg

`Rx.Observable.prototype.subscribeOnNext(onNext, [thisArg])`

Subscribes a function to invoke for each element in the observable sequence.

Arguments

1. `onNext` (*Function*): Function to invoke for each element in the observable sequence.
2. `[thisArg]` (*Any*): Object to use as this when executing callback.

Returns

(Disposable): The source sequence whose subscriptions and unsubscriptions happen on the specified scheduler.

Example

Using functions

With a thisArg

`Rx.Observable.prototype.sum([keySelector], [thisArg])`

</rx-marbles>

Computes the sum of a sequence of values that are obtained by invoking an optional transform function on each element of the input sequence, else if not specified computes the sum on each item in the sequence.

Arguments

1. `[keySelector]` (*Scheduler*): A transform function to apply to each element. The callback is called with the following information:
     i. the value of the element
     ii. the index of the element
     iii. the Observable object being subscribed

Returns

(*Observable*): An observable sequence containing a single element with the sum of the values in the source sequence.

Example

Without a selector

With a selector

Rx.Observable.prototype.switch()

Transforms an observable sequence of observable sequences into an observable sequence producing values only from the most recent observable sequence. There is an alias for this method called `switchLatest` for browsers <IE9.

Returns

(*Observable*): The observable sequence that at any point in time produces the elements of the most recent inner observable sequence that has been received.

Example

Rx.Observable.prototype.take(count, [scheduler])

</rx-marbles>

Returns a specified number of contiguous elements from the start of an observable sequence, using the specified scheduler for the edge case of `take(0)` .

Arguments

1. `count` (*Number*): The number of elements to return.
2. `[schduler]` (*Scheduler*): Scheduler used to produce an onCompleted message in case `count` is set to 0.

Returns

(*Observable*): An observable sequence that contains the elements that occur after the specified index in the input sequence.

Example

Rx.Observable.prototype.takeLast(count)

</rx-marbles>

Returns a specified number of contiguous elements from the end of an observable sequence, using an optional scheduler to drain the queue.

This operator accumulates a buffer with a length enough to store elements count elements. Upon completion of the source sequence, this buffer is drained on the result sequence. This causes the elements to be delayed.

Arguments

1. `count` (*Number*): Number of elements to bypass at the end of the source sequence.

Returns

(*Observable*): An observable sequence containing the source sequence elements except for the bypassed ones at the end.

Example

`Rx.Observable.prototype.takeLastBuffer(count)`

Returns an array with the specified number of contiguous elements from the end of an observable sequence.

Arguments

1. `count` (*Number*): Number of elements to bypass at the end of the source sequence.

Returns

(*Observable*): An observable sequence containing a single array with the specified number of elements from the end of the source sequence.

Example

`Rx.Observable.prototype.takeLastBufferWithTime(duration, [scheduler])`

Returns an array with the elements within the specified duration from the end of the observable source sequence, using the specified scheduler to run timers.

This operator accumulates a queue with a length enough to store elements received during the initial duration window. As more elements are received, elements older than the specified duration are taken from the queue and produced on the result sequence. This causes elements to be delayed with duration.

Arguments

1. `duration` (*Number*): Duration for taking elements from the end of the sequence.
2. `[scheduler=Rx.Scheduler.timeout]` (*Scheduler*): Scheduler to run the timer on. If not specified, defaults to timeout scheduler.

Returns

(*Observable*): An observable sequence containing a single array with the elements taken during the specified duration from the end of the source sequence.

Example

Rx.Observable.prototype.takeLastWithTime(duration, [timeScheduler], [loopScheduler])

Returns elements within the specified duration from the end of the observable source sequence, using the specified schedulers to run timers and to drain the collected elements.

Arguments

1. `duration` (*Number*): Duration for taking elements from the end of the sequence.
2. `[timeScheduler=Rx.Scheduler.timeout]` (*Scheduler*): Scheduler to run the timer on. If not specified, defaults to timeout scheduler.
3. `[loopScheduler=Rx.Scheduler.currentThread]` (*Scheduler*): Scheduler to drain the collected elements. If not specified, defaults to current thread scheduler.

Returns

(*Observable*): An observable sequence with the elements taken during the specified duration from the end of the source sequence.

Example

Rx.Observable.prototype.takeUntil(other)

</rx-marbles>

Returns the values from the source observable sequence until the other observable sequence or Promise produces a value.

Arguments

1. `other` (*Observable* | *Promise*): Observable sequence or Promise that terminates propagation of elements of the source sequence.

Returns

(*Observable*): An observable sequence containing the elements of the source sequence up to the point the other sequence or Promise interrupted further propagation.

Example

`Rx.Observable.prototype.takeUntilWithTime(other)`

Returns the values from the source observable sequence until the other observable sequence produces a value.

Arguments

1. `endTime` *(Date | Number)*: Time to stop taking elements from the source sequence. If this value is less than or equal to the current time, the result stream will complete immediately.
2. [`scheduler`]*(Scheduler)*: Scheduler to run the timer on.

Returns

(Observable): An observable sequence with the elements taken until the specified end time.

Example

Rx.Observable.prototype.takeWhile(predicate, [thisArg])

Returns elements from an observable sequence as long as a specified condition is true.

Arguments

1. `predicate` (*Function*): A function to test each source element for a condition. The callback is called with the following information:
   i. the value of the element
   ii. the index of the element
   iii. the Observable object being subscribed
2. `[thisArg]` (*Any*): Object to use as this when executing callback.

Returns

(*Observable*): An observable sequence that contains the elements from the input sequence that occur before the element at which the test no longer passes.

Example

`Observable.prototype.thenDo(selector)`

Matches when the observable sequence has an available value and projects the value.

Arguments

1. `selector` *(Function)*: A function that will be invoked for values in the source sequence.

Returns

(Plan): Plan that produces the projected values, to be fed (with other plans) to the when operator.

`Rx.Observable.prototype.throttle(dueTime, [scheduler])`

Ignores values from an observable sequence which are followed by another value before dueTime.

Arguments

1. `dueTime` (*Number*): Duration of the throttle period for each value (specified as an integer denoting milliseconds).
2. `[scheduler=Rx.Scheduler.timeout]` (*Any*): Scheduler to run the throttle timers on. If not specified, the timeout scheduler is used.

Returns

(*Observable*): The throttled sequence.

Example

`Rx.Observable.prototype.timeInterval([scheduler])`

Records the time interval between consecutive values in an observable sequence.

Arguments

1. `[scheduler=Rx.Observable.timeout]` (*Scheduler*): Scheduler used to compute time intervals. If not specified, the timeout scheduler is used.

Returns

(*Observable*): An observable sequence with time interval information on values.

Example

Rx.Observable.prototype.timeout(dueTime, [other], [scheduler])

Returns the source observable sequence or the other observable sequence if dueTime elapses.

Arguments

1. `dueTime` *(Date | Number)*: Absolute (specified as a Date object) or relative time (specified as an integer denoting milliseconds) when a timeout occurs.
2. `[other]` *(Observable)*: Sequence or Promise to return in case of a timeout. If not specified, a timeout error throwing sequence will be used.
3. `[scheduler=Rx.Observable.timeout]` *(Scheduler)*: Scheduler to run the timeout timers on. If not specified, the timeout scheduler is used.

Returns

(Observable): An observable sequence with time interval information on values.

Example

With no other

With another

Rx.Observable.prototype.timestamp([scheduler])

Records the timestamp for each value in an observable sequence.

Arguments

1. `[scheduler=Rx.Observable.timeout]` (*Scheduler*): Scheduler used to compute timestamps. If not specified, the timeout scheduler is used.

Returns

(*Observable*): An observable sequence with timestamp information on values.

Example

Rx.Observable.prototype.toArray()

Creates a list from an observable sequence.

Returns

(*Observable*): An observable sequence containing a single element with a list containing all the elements of the source sequence.

Example

`Rx.Observable.prototype.toMap(keySelector, [elementSelector])`

Converts the observable sequence to a Map if it exists. Note that this only works in an ES6 environment or polyfilled.

Arguments

1. `keySelector` (*Function*): A function which produces the key for the Map.
2. `[elementSelector]` (*Function*): An optional function which produces the element for the Map. If not present, defaults to the value from the observable sequence.

Returns

(*Observable*): An observable sequence with a single value of a Map containing the values from the observable sequence.

Example

`Rx.Observable.prototype.toPromise(promiseCtor)`

Converts an existing observable sequence to an ES6 Compatible Promise.

Arguments

1. `promiseCtor` *(Function)*: The constructor of the promise. If not provided, it looks for it in Rx.config.Promise.

Returns

(Promise): An ES6 compatible promise with the last value from the observable sequence.

Example

```
var promise = Rx.Observable.return(42).toPromise(RSVP.Promise);

promise.then(console.log.bind(console));

// => 42
```

Example with config

```
Rx.config.Promise = RSVP.Promise;
var promise = Rx.Observable.return(42).toPromise();

promise.then(console.log.bind(console));

// => 42
```

`Rx.Observable.prototype.toSet()`

Creates an observable sequence with a single item of a Set created from the observable sequence. Note that this only works in an ES6 environment or polyfilled.

Returns

(`Observable`): An observable sequence containing a single element with a Set containing all the elements of the source sequence.

Example

`Rx.Observable.prototype.transduce(transducer)`

Executes a transducer to transform the observable sequence.

Transducers are composable algorithmic transformations. They are independent from the context of their input and output sources and specify only the essence of the transformation in terms of an individual element. Because transducers are decoupled from input or output sources, they can be used in many different processes such as Observable sequences. Transducers compose directly, without awareness of input or creation of intermediate aggregates.

Such examples of transducers libraries are transducers-js from Cognitect and transducers.js from James Long.

In order for this operator to work, it must the transducers library must follow the following contract:

```
return {
  init: function() {
    // Return the item
    return observer;
  },
  step: function(obs, input) {
    // Process next item
    return obs.onNext(input);
  },
  result: function(obs) {
    // Mark completion
    return obs.onCompleted();
  }
};
```

Arguments

1. `transducer` (*Transducer*): A transducer to execute.

Returns

(*Observable*): An observable sequence that results from the transducer execution.

Example

Below is an example using transducers-js.

```javascript
function even (x) { return x % 2 === 0; }
function mul10(x) { return x * 10; }

var t = transducers

var source = Rx.Observable.range(1, 5)
  .transduce(t.comp(t.filter(even), t.map(mul10)));

var subscription = source.subscribe(
  function (x) {
    console.log('Next: %s', x);
  },
  function (err) {
    console.log('Error: %s', err);
  },
  function () {
      console.log('Completed');
  });

// => Next: 20
// => Next: 40
// => Completed
```

Rx.Observable.prototype.window([windowOpenings], [windowBoundaries], windowClosingSelector)

Projects each element of an observable sequence into zero or more windows.

```
// With window closing selector
Rx.Observable.prototype.window(windowClosingSelector);

// With window opening and window closing selector
Rx.Observable.prototype.window(windowOpenings, windowClosingSelector);

// With boundaries
Rx.Observable.prototype.window(windowBoundaries);
```

Arguments

1. `[windowOpenings]` (*Observable*): Observable sequence whose elements denote the creation of new windows 2. `[windowBoundaries]` (*Observable*): Sequence of window boundary markers. The current window is closed and a new window is opened upon receiving a boundary marker.
2. `windowClosingSelector` (*Function*): A function invoked to define the closing of each produced window.

Returns

(*Observable*): An observable sequence of windows.

Example

With window boundaries

With window opening and window closing selector

With openings and closings

`Rx.Observable.prototype.windowWithCount(count, [skip])`

Projects each element of an observable sequence into zero or more windows which are produced based on element count information.

Arguments

1. `count` (*Function*): Length of each buffer.
2. `[skip]` (*Function*): Number of elements to skip between creation of consecutive windows. If not provided, defaults to the count.

Returns

(*Observable*): An observable sequence of windows.

Example

Without a skip

Using a skip

Rx.Observable.prototype.windowWithTime(timeSpan, [timeShift | scheduler])

Projects each element of an observable sequence into zero or more buffers which are produced based on timing information.

Arguments

1. `timeSpan` (*Number*): Length of each buffer (specified as an integer denoting milliseconds).
2. `[timeShift]` (*Number*): Interval between creation of consecutive buffers (specified as an integer denoting milliseconds).
3. `[scheduler=Rx.Scheduler.timeout]` (*Scheduler*): Scheduler to run buffer timers on. If not specified, the timeout scheduler is used.

Returns

(*Observable*): An observable sequence of buffers.

Example

Without a skip

Using a skip

Rx.Observable.prototype.windowWithTimeOrCount(timeSpan, count, [scheduler])

Projects each element of an observable sequence into a window that is completed when either it's full or a given amount of time has elapsed.

Arguments

1. `timeSpan` *(Number)*: Maximum time length of a window.
2. `count` *(Number)*: Maximum element count of a window.
3. `[scheduler=Rx.Scheduler.timeout]` *(Scheduler)*: Scheduler to run windows timers on. If not specified, the timeout scheduler is used.

Returns

(Observable): An observable sequence of windows.

Example

`Rx.Observable.prototype.withLatestFrom(...args, resultSelector)`

Merges the specified observable sequences into one observable sequence by using the selector function only when the source observable sequence (the instance) produces an element. The other observables can be in the form of an argument list of observables or an array.

Arguments

1. `args` *(arguments | Array)*: An array or arguments of Observable sequences.
2. `resultSelector` *(Function)*: Function to invoke when the instance source observable produces an element.

Returns

(Observable): An observable sequence containing the result of combining elements of the sources using the specified result selector function.

Example

```
/* Have staggering intervals */
var source1 = Rx.Observable.interval(140)
    .map(function (i) { return 'First: ' + i; });

var source2 = Rx.Observable.interval(50)
    .map(function (i) { return 'Second: ' + i; });

// When source1 emits a value, combine it with the latest emission from source2.
var source = source1.withLatestFrom(
    source2,
    function (s1, s2) { return s1 + ', ' + s2; }
).take(4);

var subscription = source.subscribe(
    function (x) {
        console.log('Next: ' + x.toString());
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

// => Next: First: 0, Second: 1
// => Next: First: 1, Second: 4
// => Next: First: 2, Second: 7
// => Next: First: 3, Second: 10
// => Completed
```

`Rx.Observable.prototype.zip(...args, [resultSelector])`

</rx-marbles>

Merges the specified observable sequences or Promises into one observable sequence by using the selector function whenever all of the observable sequences or an array have produced an element at a corresponding index.

The last element in the arguments must be a function to invoke for each series of elements at corresponding indexes in the sources.

Arguments

1. `args` (*Arguments* | *Array*): Arguments or an array of observable sequences.
2. `[resultSelector]` (*Any*): Function to invoke for each series of elements at corresponding indexes in the sources, used only if the first parameter is not an array.

Returns

(*Observable*): An observable sequence containing the result of combining elements of the sources using the specified result selector function.

Example

Using arguments

Using an array

Observer object

The Observer object provides support for push-style iteration over an observable sequence.

The Observer and Objects interfaces provide a generalized mechanism for push-based notification, also known as the observer design pattern. The Observable object represents the object that sends notifications (the provider); the Observer object represents the class that receives them (the observer).

- Observer methods
- Observer instance methods

Observer Methods

- create
- fromNotifier

Rx.Observer.create([onNext], [onError], [onCompleted])

Creates an observer from the specified `onNext` , `onError` , and `onCompleted` actions.

Arguments

1. `[onNext]` *(Function)*: Observer's onNext action implementation.
2. `[onError]` *(Function)*: Observer's onError action implementation.
3. `[onCompleted]` *(Function)*: Observer's onCompleted action implementation.

Returns

(Observer): The observer object implemented using the given actions.

Example

Location

- rx.js

`Rx.Observer.fromNotifier(handler)`

Creates an observer from a notification callback.

Arguments

1. `handler` *(Function)*: Function that handles a notification.

Returns

(Observer): The observer object that invokes the specified handler using a notification corresponding to each message it receives.

Example

Location

- rx.js

Observer Instance Methods

- asObserver
- checked
- notifyOn
- onCompleted
- onError
- onNext
- toNotifier

`Rx.Observer.prototype.asObserver()`

Hides the identity of an observer.

Returns

(Observer): An observer that hides the identity of the specified observer.

Example

Location

- rx.js

`Rx.Observer.prototype.checked()`

Checks access to the observer for grammar violations. This includes checking for multiple `onError` or `onCompleted` calls, as well as reentrancy in any of the observer methods.

If a violation is detected, an Error is thrown from the offending observer method call.

Returns

(Observer): An observer that checks callbacks invocations against the observer grammar and, if the checks pass, forwards those to the specified observer.

Example

Location

- rx.js

`Rx.Observer.prototype.checked()`

Rx.Observer.prototype.notifyOn(scheduler)

Schedules the invocation of observer methods on the given scheduler.

Arguments

1. `scheduler` *(Scheduler)*: Scheduler to schedule observer messages on.

Returns

(Observer): Observer whose messages are scheduled on the given scheduler.

Example

Location

- rx.js

Rx.Observer.prototype.onCompleted()

Notifies the observer of the end of the sequence.

Example

Location

- rx.js

Rx.Observer.prototype.onCompleted()

Rx.Observer.prototype.onError(error)

Notifies the observer that an exception has occurred.

Arguments

1. `error` *(Any)*: The error that has occurred.

Example

Location

- rx.js

`Rx.Observer.prototype.onNext(value)`

Notifies the observer of a new element in the sequence.

Arguments

1. `value` *(Any)*: Next element in the sequence.

Example

Location

- rx.js

Rx.Observer.prototype.toNotifier()

Creates a notification callback from an observer.

Returns

(Function): The function that forwards its input notification to the underlying observer.

Example

Location

- rx.js

Notification object

Represents a notification to an observer.

- Notification Methods
- Notification Instance Methods
- Notification Properties

Notification Methods

- `createOnCompleted`
- `createOnError`
- `createOnNext`

Rx.Notification.createOnNext(value)

Creates an object that represents an OnNext notification to an observer.

Arguments

1. `value` *(Any)*: The value contained in the notification.

Returns

(Notification): The OnNext notification containing the value.

Example

Location

- rx.js

Rx.Notification.createOnError(exception)

Creates an object that represents an OnError notification to an observer.

Arguments

1. `exception` *(Any)*: The exception contained in the notification.

Returns

(Notification): The OnError notification containing the exception.

Example

Location

- rx.js

Rx.Notification.createOnCompleted()

Creates an object that represents an OnCompleted notification to an observer.

Returns

(Notification): The OnCompleted notification.

Example

Location

- rx.js

Notification Instance Methods

- `accept`
- `toObservable`

Rx.Notification.prototype.accept([observer] | [onNext], [onError], [onCompleted])

Invokes the delegate corresponding to the notification or the observer's method corresponding to the notification and returns the produced result.

Arguments

1. `[observer]` *(Observer)*: Observer to invoke the notification on.
2. `[onNext]` *(Function)*: Function to invoke for an OnNext notification.
3. `[onError]` *(Function)*: Function to invoke for an OnError notification.
4. `[onError]` *(Function)*: Function to invoke for an OnCompleted notification.

Returns

(Any): Result produced by the observation.

Example

Using an observer

Using a function

Location

- rx.js

Rx.Notification.prototype.toObservable([scheduler])

Returns an observable sequence with a single notification.

Arguments

1. `[scheduler = Rx.Scheduler.immediate]` *(Scheduler)*: Scheduler to send out the notification calls on.

Returns

(Observable): The observable sequence that surfaces the behavior of the notification upon subscription.

Example

Without a scheduler

With a scheduler

Location

- rx.js

Notification Properties

- `exception`
- `hasValue`
- `kind`
- `value`

Rx.Notification.prototype.exception

Gets the exception from the OnError notification.

Returns

(Any): The Exception from the OnError notification.

Example

Location

- rx.js

Rx.Notification.prototype.exception

Rx.Notification.prototype.hasValue

Determines whether the Notification has a value. Returns `true` for OnNext Notifications, and `false` for OnError and OnCompleted Notifications.

Returns

(Bool): Returns `true` for OnNext Notifications, and `false` for OnError and OnCompleted Notifications.

Example

Location

- rx.js

`Rx.Notification.prototype.kind`

Gets the kind from the notification which denotes 'N' for OnNext, 'E' for OnError and 'C' for OnCompleted.

Returns

(String): The kind from the notification which denotes 'N' for OnNext, 'E' for OnError and 'C' for OnCompleted.

Example

Location

- rx.js

`Rx.Notification.prototype.kind`

`Rx.Notification.prototype.value`

Gets the value from the OnNext notification.

Returns

(Any): The value from the OnNext notification.

Example

Location

- rx.js

`Rx.Notification.prototype.value`

Subjects

- Rx.AsyncSubject
- Rx.BehaviorSubject
- Rx.ReplaySubject
- Rx.Subject

`Rx.AsyncSubject` class

Represents the result of an asynchronous operation. The last value before the OnCompleted notification, or the error received through OnError, is sent to all subscribed observers.

This class inherits both from the `Rx.Observable` and `Rx.Observer` classes.

Usage

The follow example shows caching on the last value produced when followed by an onCompleted notification which makes it available to all subscribers.

```javascript
var subject = new Rx.AsyncSubject();

var i = 0;
var handle = setInterval(function () {
    subject.onNext(i)
    if (++i > 3) {
        subject.onCompleted();
        clearInterval(handle);
    }
}, 500);

var subscription = subject.subscribe(
    function (x) {
        console.log('Next: ' + x.toString());
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

// => Next: 3
// => Completed
```

Location

- rx.js

`AsyncSubject Constructor`

- `constructor`

AsyncSubject Instance Methods

- `dispose`
- `hasObservers`

Inherited Classes

- `Rx.Observable`
- `Rx.Observer`

AsyncSubject Constructor

Rx.AsyncSubject()

Ⓢ

Creates a subject that can only receive one value and that value is cached for all future observations.

Example

```
var subject = new Rx.AsyncSubject();

subject.onNext(42);
subject.onCompleted();

var subscription = source.subscribe(
    function (x) {
        console.log('Next: ' + x);
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

// => 42
// => Completed
```

Location

- rx.js

AsyncSubject Instance Methods

`Rx.AsyncSubject.prototype.dispose()`

Ⓢ

Unsubscribe all observers and release resources.

Example

```javascript
var subject = new Rx.AsyncSubject();

var subscription = subject.subscribe(
    function (x) {
        console.log('Next: ' + x.toString());
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

subject.onNext(42);
subject.onCompleted();

// => Next: 42
// => Completed

subject.dispose();

try {
    subject.onNext(56);
} catch (e) {
    console.log(e.message);
}

// => Object has been disposed
```

Location

- rx.js

Rx.AsyncSubject.prototype.hasObservers()

\# ⓢ

Indicates whether the subject has observers subscribed to it.

Returns

(Boolean): Returns `true` if the AsyncSubject has observers, else `false` .

Example

```
var subject = new Rx.AsyncSubject();

console.log(subject.hasObservers());

// => false

var subscription = subject.subscribe(
    function (x) {
        console.log('Next: ' + x.toString());
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

console.log(subject.hasObservers());

// => true
```

Location

- rx.js

`Rx.BehaviorSubject` class

Represents a value that changes over time. Observers can subscribe to the subject to receive the last (or initial) value and all subsequent notifications.

This class inherits both from the `Rx.Observable` and `Rx.Observer` classes.

Usage

The follow example shows the basic usage of an `Rx.BehaviorSubject` class.

```javascript
/* Initialize with initial value of 42 */
var subject = new Rx.BehaviorSubject(42);

var subscription = subject.subscribe(
    function (x) {
        console.log('Next: ' + x.toString());
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

// => Next: 42

subject.onNext(56);
// => Next: 56

subject.onCompleted();
// => Completed
```

Location

- rx.binding.js

BehaviorSubject Constructor

- constructor

`BehaviorSubject Instance Methods`

- `dispose`
- `hasObservers`

Inherited Classes

- `Rx.Observable`
- `Rx.Observer`

BehaviorSubject Constructor

`Rx.BehaviorSubject(initialValue)`

[#](S)

Initializes a new instance of the `Rx.BehaviorSubject` class which creates a subject that caches its last value and starts with the specified value.

Arguments

1. `initialValue` *(Any)*: Initial value sent to observers when no other value has been received by the subject yet.

Example

```
var subject = new Rx.BehaviorSubject(56);

subject.onCompleted();

var subscription = source.subscribe(
    function (x) {
        console.log('Next: ' + x);
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

// => Next: 56

subject.onNext(42);
// => Next: 42

subject.onCompleted();
// => Completed
```

Location

= rx.binding.js

BehaviorSubject Instance Methods

`Rx.BehaviorSubject.prototype.dispose()`

Ⓢ

Unsubscribe all observers and release resources.

Example

```
var subject = new Rx.BehaviorSubject();

var subscription = subject.subscribe(
    function (x) {
        console.log('Next: ' + x.toString());
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

subject.onNext(42);
// => Next: 42

subject.onCompleted();
// => Completed

subject.dispose();

try {
    subject.onNext(56);
} catch (e) {
    console.log(e.message);
}

// => Object has been disposed
```

Location

= rx.binding.js

Rx.BehaviorSubject.prototype.hasObservers()

\# Ⓢ

Indicates whether the subject has observers subscribed to it.

Returns

(Boolean): Returns `true` if the Subject has observers, else `false` .

Example

```
var subject = new Rx.BehaviorSubject();

console.log(subject.hasObservers());

// => false

var subscription = subject.subscribe(
    function (x) {
        console.log('Next: ' + x.toString());
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

console.log(subject.hasObservers());

// => true
```

Location

= rx.binding.js

`Rx.ReplaySubject` class

Represents an object that is both an observable sequence as well as an observer. Each notification is broadcasted to all subscribed and future observers, subject to buffer trimming policies.

This class inherits both from the `Rx.Observable` and `Rx.Observer` classes.

Usage

The follow example shows the basic usage of an `Rx.ReplaySubject` class. Note that this only holds the past two items in the cache.

```javascript
var subject = new Rx.ReplaySubject(2 /* buffer size */);

subject.onNext('a');
subject.onNext('b');
subject.onNext('c');

var subscription = subject.subscribe(
    function (x) {
        console.log('Next: ' + x.toString());
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

// => Next: b
// => Next: c

subject.onNext('d');
// => Next: d
```

Location

- rx.binding.js

ReplaySubject Constructor

- `constructor`

ReplaySubject Instance Methods

- `dispose`
- `hasObservers`

Inherited Classes

- `Rx.Observable`
- `Rx.Observer`

ReplaySubject Constructor

```
Rx.ReplaySubject([bufferSize], [windowSize],
[scheduler])
```

\# Ⓢ

Initializes a new instance of the `Rx.ReplaySubject` class with the specified buffer size, window and scheduler.

Arguments

1. `[bufferSize = Number.MAX_VALUE]` *(Number)*: Maximum element count of the replay buffer.
2. `[windowSize = NUMBER.MAX_VALUE]` *(Number)*: Maximum time length of the replay buffer.
3. `[scheduler = Rx.Scheduler.currentThread]` *(Scheduler)*: Scheduler the observers are invoked on.

Example

```
var subject = new Rx.ReplaySubject(
    2 /* buffer size */,
    null /* unlimited time buffer */,
    Rx.Scheduler.timeout);

subject.onNext('a');
subject.onNext('b');
subject.onNext('c');

var subscription = subject.subscribe(
    function (x) {
        console.log('Next: ' + x.toString());
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

// => Next: b
// => Next: c

subject.onNext('d');
// => Next: d
```

Location

- rx.binding.js

ReplaySubject Instance Methods

```
Rx.ReplaySubject.prototype.dispose()
```

\# ⓢ

Unsubscribe all observers and release resources.

Example

```
var subject = new Rx.ReplaySubject();

var subscription = subject.subscribe(
    function (x) {
        console.log('Next: ' + x.toString());
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

subject.onNext(42);
// => Next: 42

subject.onCompleted();
// => Completed

subject.dispose();

try {
    subject.onNext(56);
} catch (e) {
    console.log(e.message);
}

// => Object has been disposed
```

Location

- rx.binding.js

Rx.ReplaySubject.prototype.hasObservers()

Ⓢ

Indicates whether the subject has observers subscribed to it.

Returns

(Boolean): Returns `true` if the Subject has observers, else `false`.

Example

```
var subject = new Rx.ReplaySubject();

console.log(subject.hasObservers());

// => false

var subscription = subject.subscribe(
    function (x) {
        console.log('Next: ' + x.toString());
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

console.log(subject.hasObservers());

// => true
```

Location

- rx.binding.js

`Rx.Subject` class

Represents an object that is both an observable sequence as well as an observer. Each notification is broadcasted to all subscribed observers.

This class inherits both from the `Rx.Observable` and `Rx.Observer` classes.

Usage

The follow example shows the basic usage of an Rx.Subject.

```javascript
var subject = new Rx.Subject();

var subscription = subject.subscribe(
    function (x) {
        console.log('Next: ' + x.toString());
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

subject.onNext(42);

// => Next: 42

subject.onNext(56);

// => Next: 56

subject.onCompleted();

// => Completed
```

Location

- rx.js

Subject Constructor

- constructor

Subject Class Methods

- create

Subject Instance Methods

- dispose
- hasObservers

Inherited Classes

- Rx.Observable
- Rx.Observer

Subject Constructor

Rx.Subject()

\# Ⓢ

Creates a subject.

Example

```
var subject = new Rx.Subject();

var subscription = source.subscribe(
    function (x) {
        console.log('Next: ' + x);
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

subject.onNext(42);
// => Next: 42

subject.onCompleted();
// => Completed
```

Location

- rx.js

Subject Class Methods

`Rx.Subject.create(observer, observable)`

Ⓢ

Creates a subject from the specified observer and observable.

Arguments

1. `observer` *(Observer)*: The observer used to send messages to the subject.
2. `observable` *(Observable)*: The observable used to subscribe to messages sent from the subject.

Returns

(Subject): Subject implemented using the given observer and observable.

Example

```javascript
/* Using a Web Worker to send and receive data via an Rx.Subject */

/* worker.js */

self.onmessage = function(e) {
    self.postMessage(e.data);
};

/* client.js */
var worker = new Worker('worker.js');

// Create observer to handle sending messages
var observer = Rx.Observer.create(
    function (data) {
        worker.postMessage(data);
    });

// Create observable to handle the messages
var observable = Rx.Observable.create(function (obs) {

    worker.onmessage = function (data) {
        obs.onNext(data);
    };

    worker.onerror = function (err) {
        obs.onError(err);
    };

    return function () {
        worker.close();
    };
});

var subject = Rx.Subject.create(observer, observable);

var subscription = subject.subscribe(
    function (x) {
        console.log('Next: ' + x);
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

subject.onNext(42);
// => Next: 42
```

Location

- rx.js

Subject Instance Methods

Rx.Subject.prototype.dispose()

Ⓢ

Unsubscribe all observers and release resources.

Example

```
var subject = new Rx.Subject();

var subscription = subject.subscribe(
    function (x) {
        console.log('Next: ' + x.toString());
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

subject.onNext(42);
// => Next: 42

subject.onCompleted();
// => Completed

subject.dispose();

try {
    subject.onNext(56);
} catch (e) {
    console.log(e.message);
}

// => Object has been disposed
```

Location

- rx.js

`Rx.Subject.prototype.hasObservers()`

Ⓢ

Indicates whether the subject has observers subscribed to it.

Returns

(Boolean): Returns `true` if the Subject has observers, else `false`.

Example

```javascript
var subject = new Rx.Subject();

console.log(subject.hasObservers());
// => false

var subscription = subject.subscribe(
    function (x) {
        console.log('Next: ' + x.toString());
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });

console.log(subject.hasObservers());
// => true
```

Location

- rx.js

Schedulers

- Rx.HistoricalScheduler
- Rx.Scheduler
- Rx.VirtualTimeScheduler

`Rx.HistoricalScheduler` class

Ⓢ

Provides a virtual time scheduler that uses a `Date` for absolute time and time spans for relative time. This inherits from the `Rx.VirtualTimeScheduler` class.

Usage

The following shows an example of using the `Rx.HistoricalScheduler` . This shows creating a minute's worth of data from January 1st, 1970.

```
// Initial data
var initialDate = 0;
var scheduler = new Rx.HistoricalScheduler(new Date(initialDate));

// Yield unto this subject
var s = new Rx.Subject();

// Some random data
function getData(time) {
  return Math.floor(Math.random() * (time + 1));
}

// Enqueue 1 minute's worth of data
while (initialDate <= 60000) {

  (function (i) {

    scheduler.scheduleWithAbsolute(i, function () {
      s.onNext({ value: getData(i), date: new Date(i) });
    });

  }(initialDate));

  initialDate += 10000;
}

// Subscription set
s.subscribe(function (x) {
  console.log('value: ', x.value);
  console.log('date: ', x.date.toGMTString());
});

// Run it
scheduler.start();

// => value: 0
// => date: Thu, 1 Jan 1970 00:00:00 UTC
// => value: 2013
// => date: Thu, 1 Jan 1970 00:00:10 UTC
// => value: 5896
// => date: Thu, 1 Jan 1970 00:00:20 UTC
// => value: 5415
// => date: Thu, 1 Jan 1970 00:00:30 UTC
// => value: 13411
// => date: Thu, 1 Jan 1970 00:00:40 UTC
// => value: 15518
// => date: Thu, 1 Jan 1970 00:00:50 UTC
// => value: 51076
// => date: Thu, 1 Jan 1970 00:01:00 UTC
```

Location

File:

- `/src/core/concurrency/historicalscheduler.js`

Dist:

- `rx.all.js`
- `rx.all.compat.js`
- `rx.virutaltime.js`

NPM Packages:

- `rx`

NuGet Packages:

- `RxJS-All`
- `RxJS-VirtualTime`

Unit Tests:

- `/tests/concurrency/historicalscheduler.js`

HistoricalScheduler Constructor

- `constructor`

Inherited Classes

- `Rx.VirtualScheduler`

HistoricalScheduler Constructor

Rx.HistoricalScheduler([initialClock], [comparer])

Ⓢ

Creates a new historical scheduler with the specified initial clock value.

Arguments

1. [`initialClock`] *(Function)*: Initial value for the clock.

2. [`comparer`] *(Function)*: Comparer to determine causality of events based on absolute time.

Example

```javascript
function comparer (x, y) {
  if (x > y) { return 1; }
  if (x < y) { return -1; }
  return 0;
}

var scheduler = new Rx.HistoricalScheduler(
  new Date(0),  /* initial clock of 0 */
  comparer      /* comparer for determining order */
);
```

`Rx.Scheduler` class

Provides a set of static methods to access commonly used schedulers and a base class for all schedulers.

Usage

The follow example shows the basic usage of an `Rx.Scheduler`.

```
var disposable = Rx.Scheduler.timeout.scheduleWithState(
    'world',
    function (x) {
        console.log('hello ' + x);
    }
);

// => hello world
```

Location

- rx.js

Scheduler Constructor

- `constructor`

Scheduler Instance Methods

- `catchException`
- `now`

Standard Scheduling

- `schedule`
- `scheduleWithState`
- `scheduleWithAbsolute`
- `scheduleWithAbsoluteAndState`
- `scheduleWithRelative`

- `scheduleWithRelativeAndState`

Recursive Scheduling

- `scheduleRecursive`
- `scheduleRecursiveWithState`
- `scheduleRecursiveWithAbsolute`
- `scheduleRecursiveWithAbsoluteAndState`
- `scheduleRecursiveWithRelative`
- `scheduleRecursiveWithRelativeAndState`

Periodic Scheduling

- `schedulePeriodic`
- `schedulePeriodicWithState`

`Scheduler Class Methods

- `normalize`

Scheduler Class Properties

- `currentThread`
- `immediate`
- `timeout`

Scheduler Constructor

```
Rx.Scheduler(now, schedule, scheduleRelative,
scheduleAbsolute)
```

\# Ⓢ

Initializes a new instance of the `Rx.Scheduler` . This is meant for Scheduler implementers and not normal usage.

Arguments

1. `now` *(Function)*: Function which gets the current time according to the local machine's

system clock.

2. `schedule` *(Function)*: Function to schedule an action immediately.

3. `scheduleRelative` *(Function)*: Function used to schedule an action in relative time.

4. `scheduleAbsolute` *(Function)*: Function used to schedule an action in absolute time.

Example

```javascript
// Used for scheduling immediately
function schedule(state, action) {
    var scheduler = this,
        disposable = new Rx.SingleAssignmentDisposable();

    var id = setTimeout(function () {
        if (!disposable.isDisposed) {
            disposable.setDisposable(action(scheduler, state));
        }
    }, 0);

    return new CompositeDisposable(disposable, disposableCreate(function () {
        clearMethod(id);
    }));
}

// Used for scheduling relative to now
function scheduleRelative(state, dueTime, action) {
    var scheduler = this,
        dt = Scheduler.normalize(dueTime);

    // Shortcut if already 0
    if (dt === 0) {
        return scheduler.scheduleWithState(state, action);
    }

    var disposable = new Rx.SingleAssignmentDisposable();
    var id = window.setTimeout(function () {
        if (!disposable.isDisposed) {
            disposable.setDisposable(action(scheduler, state));
        }
    }, dt);

    return new CompositeDisposable(disposable, disposableCreate(function () {
        window.clearTimeout(id);
    }));
}

// Used for scheduling in absolute time
function scheduleAbsolute(state, dueTime, action) {
    return this.scheduleWithRelativeAndState(state, dueTime - this.now(), action);
}
```

```
var timeoutScheduler = new Rx.Scheduler(
    Date.now,
    schedule,
    scheduleRelative,
    scheduleAbsolute
);

var handle = timeoutScheduler.schedule(function () {
    console.log('hello');
});


// => hello
```

Location

- rx.js

Scheduler Instance Methods

Rx.Scheduler.prototype.catch(handler)

\# Ⓢ

Returns a scheduler that wraps the original scheduler, adding exception handling for scheduled actions. There is an alias of `catchException` for browsers < IE9.

Arguments

1. `handler` *(Function)*: Handler that's run if an exception is caught. The exception will be rethrown if the handler returns `false`.

Returns

(Scheduler): Wrapper around the original scheduler, enforcing exception handling.

Example

```javascript
function inherits (ctor, superCtor) {
    ctor.super_ = superCtor;
    ctor.prototype = Object.create(superCtor.prototype, {
        constructor: {
            value: ctor,
            enumerable: false,
            writable: true,
            configurable: true
        }
    });
}


inherits (SchedulerError, Error);


function SchedulerError(message) {
    Error.call(this, message);
}


var scheduler = Rx.Scheduler.timeout;
var catchScheduler = scheduler.catchException(function (e) {
    return e instanceof SchedulerError;
});

// Throws no exception
var d1 = catchScheduler.schedule(function () {
    throw new SchedulerError('woops');
});

var d2 = catchScheduler.schedule(function () {
    throw new Error('woops');
});

// => Uncaught Error: woops
```

Location

- rx.js

`Rx.Scheduler.prototype.now()`

\# Ⓢ

Gets the current time according to the Scheduler implementation.

Returns

(Number): The current time according to the Scheduler implementation.

Example

```
var now = Rx.Scheduler.timeout.now();

console.log(now);
// => 1381806323143
```

Location

- rx.js

Standard Scheduling

`Rx.Scheduler.prototype.schedule(action)`

Ⓢ

Schedules an action to be executed.

Arguments

1. `action` *(Function)*: Action to execute.

Returns

(Disposable): The disposable object used to cancel the scheduled action (best effort).

Example

```
var disposable = Rx.Scheduler.immediate.schedule(function () {
    console.log('hello');
});

// => hello

// Tries to cancel but too late since it is immediate
disposable.dispose();
```

Location

- rx.js

`Rx.Scheduler.prototype.scheduleWithState(state, action)`

Ⓢ

Schedules an action to be executed with state.

Arguments

1. `state` *(Any)*: State passed to the action to be executed.
2. `action` *(Function)*: Action to execute.

Returns

(Disposable): The disposable object used to cancel the scheduled action (best effort).

Example

```
var disposable = Rx.Scheduler.immediate.scheduleWithState('world', function (x) {
    console.log('hello ' + x);
});

// => hello world

// Tries to cancel but too late since it is immediate
disposable.dispose();
```

Location

- rx.js

`Rx.Scheduler.prototype.scheduleWithAbsolute(duetime, action)`

Ⓢ

Schedules an action to be executed at the specified absolute due time. Note this only works with the built-in `Rx.Scheduler.timeout` scheduler, as the rest will throw an exception as the framework does not allow for blocking.

Arguments

1. `dueTime` *(Number)*: Absolute time at which to execute the action.
2. `action` *(Function)*: Action to execute.

Returns

(Disposable): The disposable object used to cancel the scheduled action (best effort).

Example

```
var disposable = Rx.Scheduler.timeout.scheduleWithAbsolute(
    Date.now() + 5000, /* 5 seconds in the future */
    function () {
        console.log('hello');
    }
);

// => hello
```

Location

- rx.js

Rx.Scheduler.prototype.scheduleWithAbsoluteAndState(state, duetime, action)

Ⓢ

Schedules an action to be executed at the specified absolute due time. Note this only works with the built-in `Rx.Scheduler.timeout` scheduler, as the rest will throw an exception as the framework does not allow for blocking.

Arguments

1. `state` *(Any)*: State passed to the action to be executed.
2. `dueTime` *(Number)*: Absolute time at which to execute the action.
3. `action` *(Function)*: Action to execute.

Returns

(Disposable): The disposable object used to cancel the scheduled action (best effort).

Example

```
var disposable = Rx.Scheduler.timeout.scheduleWithAbsolute(
    'world',
    Date.now() + 5000, /* 5 seconds in the future */
    function (x) {
        console.log('hello ' + x);
    }
);


// => hello world
```

Location

- rx.js

Rx.Scheduler.prototype.scheduleWithRelative(du etime, action)

Ⓢ

Schedules an action to be executed after the specified relative due time. Note this only works with the built-in `Rx.Scheduler.timeout` scheduler, as the rest will throw an exception as the framework does not allow for blocking.

Arguments

1. `dueTime` *(Number)*: Relative time at which to execute the action.
2. `action` *(Function)*: Action to execute.

Returns

(Disposable): The disposable object used to cancel the scheduled action (best effort).

Example

```
var disposable = Rx.Scheduler.timeout.scheduleWithRelative(
    5000, /* 5 seconds in the future */
    function () {
        console.log('hello');
    }
);


// => hello
```

Location

- rx.js

Rx.Scheduler.prototype.scheduleWithRelativeAndState(state, duetime, action)

\# Ⓢ

Schedules an action to be executed at the specified relative due time. Note this only works with the built-in `Rx.Scheduler.timeout` scheduler, as the rest will throw an exception as the framework does not allow for blocking.

Arguments

1. `state` *(Any)*: State passed to the action to be executed.
2. `dueTime` *(Number)*: Relative time at which to execute the action.
3. `action` *(Function)*: Action to execute.

Returns

(Disposable): The disposable object used to cancel the scheduled action (best effort).

Example

```
var disposable = Rx.Scheduler.timeout.scheduleWithAbsolute(
    'world',
    5000, /* 5 seconds in the future */
    function (x) {
        console.log('hello ' + x);
    }
);


// => hello world
```

Location

- rx.js

Recursive Scheduling

`Rx.Scheduler.prototype.scheduleRecursive(action)`

Ⓢ

Schedules an action to be executed recursively.

Arguments

1. `action` *(Function)*: Action to execute recursively. The parameter passed to the action is used to trigger recursive scheduling of the action.

Returns

(Disposable): The disposable object used to cancel the scheduled action (best effort).

Example

```
var i = 0;

var disposable = Rx.Scheduler.immediate.scheduleRecursive(
    function (self) {
        console.log(i);
        if (++i < 3) {
            self(i);
        }
    }
);

// => 0
// => 1
// => 2
```

Location

- rx.js

Rx.Scheduler.prototype.scheduleRecursiveWithState(state, action)

Ⓢ

Schedules an action to be executed with state.

Arguments

1. `state` *(Any)*: State passed to the action to be executed.
2. `action` *(Function)*: Action to execute recursively. The last parameter passed to the action is used to trigger recursive scheduling of the action, passing in recursive invocation state.

Returns

(Disposable): The disposable object used to cancel the scheduled action (best effort).

Example

```
var disposable = Rx.Scheduler.immediate.scheduleRecursiveWithState(
    0,
    function (i, self) {
        console.log(i);
        if (++i < 3) {
            self(i);
        }
    }
);

// => 0
// => 1
// => 2
```

Location

- rx.js

Rx.Scheduler.prototype.scheduleRecursiveWithAbsolute(duetime, action)

Ⓢ

Schedules an action to be executed recursively at a specified absolute due time. Note this only works with the built-in `Rx.Scheduler.timeout` scheduler, as the rest will throw an exception as the framework does not allow for blocking.

Arguments

1. `dueTime` *(Number)*: Absolute time at which to execute the action for the first time.
2. `action` *(Function)*: Action to execute recursively. The parameter passed to the action is used to trigger recursive scheduling of the action at the specified absolute time.

Returns

(Disposable): The disposable object used to cancel the scheduled action (best effort).

Example

```
var i = 0;

var disposable = Rx.Scheduler.timeout.scheduleRecursiveWithAbsolute(
    Date.now() + 5000, /* 5 seconds in the future */
    function (self) {
        console.log(i);
        if (++i < 3) {
            // Schedule mutliplied by a second by position
            self(Date.now() + (i * 1000));
        }
    }
);

// => 0
// => 1
// => 2
```

Location

- rx.js

`Rx.Scheduler.prototype.scheduleRecursiveWithAbsoluteAndState(state, duetime, action)`

Ⓢ

Schedules an action to be executed recursively at a specified absolute due time. Note this only works with the built-in `Rx.Scheduler.timeout` scheduler, as the rest will throw an exception as the framework does not allow for blocking.

Arguments

1. `state` *(Any)*: State passed to the action to be executed.
2. `dueTime` *(Number)*: Absolute time at which to execute the action for the first time.
3. `action` *(Function)*: Action to execute recursively. The last parameter passed to the action is used to trigger recursive scheduling of the action, passing in the recursive due time and invocation state.

Returns

(Disposable): The disposable object used to cancel the scheduled action (best effort).

Example

```
var disposable = Rx.Scheduler.timeout.scheduleRecursiveWithAbsoluteAndState(
    0,
    Date.now() + 5000, /* 5 seconds in the future */
    function (i, self) {
        console.log(i);
        if (++i < 3) {
            // Schedule mutliplied by a second by position
            self(i, Date.now() + (i * 1000));
        }
    }
);

// => 0
// => 1
// => 2
```

Location

- rx.js

Rx.Scheduler.prototype.scheduleRecursiveWithRelative(duetime, action)

\# Ⓢ

Schedules an action to be executed recursively at a specified relative due time. Note this only works with the built-in `Rx.Scheduler.timeout` scheduler, as the rest will throw an exception as the framework does not allow for blocking.

Arguments

1. `dueTime` *(Number)*: Relative time at which to execute the action for the first time.
2. `action` *(Function)*: Action to execute recursively. The parameter passed to the action is used to trigger recursive scheduling of the action at the specified relative time.

Returns

(Disposable): The disposable object used to cancel the scheduled action (best effort).

Example

```
var i = 0;

var disposable = Rx.Scheduler.timeout.scheduleRecursiveWithRelative(
    5000, /* 5 seconds in the future */
    function (self) {
        console.log(i);
        if (++i < 3) {
            // Schedule mutliplied by a second by position
            self(i * 1000);
        }
    }
);

// => 0
// => 1
// => 2
```

Location

- rx.js

Rx.Scheduler.prototype.scheduleRecursiveWithAbsoluteAndState(state, duetime, action)

Ⓢ

Schedules an action to be executed recursively at a specified relative due time. Note this only works with the built-in `Rx.Scheduler.timeout` scheduler, as the rest will throw an exception as the framework does not allow for blocking.

Arguments

1. `state` *(Any)*: State passed to the action to be executed.
2. `dueTime` *(Number)*: Relative time at which to execute the action for the first time.
3. `action` *(Function)*: Action to execute recursively. The last parameter passed to the action is used to trigger recursive scheduling of the action, passing in the recursive due time and invocation state.

Returns

(Disposable): The disposable object used to cancel the scheduled action (best effort).

Example

```
var disposable = Rx.Scheduler.timeout.scheduleRecursiveWithRelativeAndState(
    0,
    5000, /* 5 seconds in the future */
    function (i, self) {
        console.log(i);
        if (++i < 3) {
            // Schedule mutliplied by a second by position
            self(i, i * 1000);
        }
    }
);

// => 0
// => 1
// => 2
```

Location

- rx.js

Periodic Scheduling

`Rx.Scheduler.prototype.schedulePeriodic(period, action)`

Ⓢ

Schedules a periodic piece of work by dynamically discovering the scheduler's capabilities. The periodic task will be scheduled using `window.setInterval` for the base implementation.

Arguments

1. `period` *(Number)*: Period for running the work periodically in ms.
2. `action` *(Function)*: Action to execute.

Returns

(Disposable): The disposable object used to cancel the scheduled action (best effort).

Example

```
var i = 0;

var disposable = Rx.Scheduler.timeout.schedulePeriodic(
    1000, /* 1 second */
    function () {
        console.log(i);

        // After three times, dispose
        if (++i > 3) {
            disposable.dispose();
        }
});

// => 0
// => 1
// => 2
// => 3
```

Location

- rx.js

Rx.Scheduler.prototype.schedulePeriodicWithState(state, period, action)

Ⓢ

Schedules a periodic piece of work by dynamically discovering the scheduler's capabilities. The periodic task will be scheduled using `window.setInterval` for the base implementation.

Arguments

1. `state` *(Any)*: State passed to the action to be executed.
2. `period` *(Number)*: Period for running the work periodically in ms.
3. `action` *(Function)*: Action to execute.

Returns

(Disposable): The disposable object used to cancel the scheduled action (best effort).

Example

```
var disposable = Rx.Scheduler.timeout.schedulePeriodicWithState(
    0,
    1000, /* 1 second */
    function (i) {
        console.log(i);

        // After three times, dispose
        if (++i > 3) {
            disposable.dispose();
        }

        return i;
});

// => 0
// => 1
// => 2
// => 3
```

Location

- rx.js

Scheduler Class Methods

Rx.Scheduler.normalize(dueTime)

\# Ⓢ

Normalizes the specified time span value to a positive value.

Arguments

1. `dueTime` *(Number)*: The time span value to normalize.

Returns

(Number): The specified time span value if it is zero or positive; otherwise, 0

Example

```
var r1 = Rx.Scheduler.normalize(-1);
console.log(r1);
// => 0

var r2 = Rx.Scheduler.normalize(255);
console.log(r1);
// => 255
```

Location

- rx.js

Scheduler Class Properties

`Rx.Scheduler.currentThread`

\# Ⓢ

Gets a scheduler that schedules work as soon as possible on the current thread. This implementation does not support relative and absolute scheduling due to thread blocking required.

Example

```
var scheduler = Rx.Scheduler.currentThread;

var disposable = scheduler.scheduleWithState(
    'world',
    function (x) {
        console.log('hello ' + x);
    });

// => hello world
```

Location

- rx.js

`Rx.Scheduler.immediate`

Ⓢ

Gets a scheduler that schedules work immediately on the current thread.

Example

```javascript
var scheduler = Rx.Scheduler.immediate;

var disposable = scheduler.scheduleRecursiveWithState(
    0,
    function (x, self) {
        console.log(x);
        if (++x < 3) {
            self(x);
        }
    }
);

// => 0
// => 1
// => 2
```

Location

- rx.js

`Rx.Scheduler.timeout`

Ⓢ

Gets a scheduler that schedules work via a timed callback based upon platform.

For all schedule calls, it defaults to:

- Node.js: uses `setImmediate` for newer builds, and `process.nextTick` for older versions.
- Browser: depending on platform may use `setImmediate` , `MessageChannel` , `window.postMessage` and for older versions of IE, it will default to `script.onreadystatechanged` , else falls back to `window.setTimeout` .

For all relative and absolute scheduling, it defaults to using `window.setTimeout` .

Example

```
var scheduler = Rx.Scheduler.timeout;

var disposable = scheduler.scheduleWithState(
    0,
    function (x) {
        console.log(x);
    }
);

// => 0
```

Location

- rx.js

`Rx.VirtualTimeScheduler` class

Base class for providing scheduling in virtual time. This inherits from the `Rx.Scheduler` class.

Usage

The following shows an example of using the `Rx.VirtualTimeScheduler`. In order for this to work, you must implement the `add`, `toDateTimeOffset` and `toRelative` methods as described below.

```javascript
/* Comparer required for scheduling priority */
function comparer (x, y) {
    if (x > y) { return 1; }
    if (x < y) { return -1; }
    return 0;
}

var scheduler = new Rx.VirtualTimeScheduler(0, comparer);

/**
 * Adds a relative time value to an absolute time value.
 * @param {Any} absolute Absolute virtual time value.
 * @param {Any} relative Relative virtual time value to add.
 * @return {Any} Resulting absolute virtual time sum value.
 */
scheduler.add = function (absolute, relative) {
    return absolute + relative;
};

/**
 * Converts an absolute time to a number
 * @param {Number} The absolute time in ms
 * @returns {Number} The absolute time in ms
 */
scheduler.toDateTimeOffset = function (absolute) {
    return new Date(absolute).getTime();
};

/**
 * Converts the time span number/Date to a relative virtual time value.
 * @param {Number} timeSpan TimeSpan value to convert.
 * @return {Number} Corresponding relative virtual time value.
 */
scheduler.toRelative = function (timeSpan) {
    return timeSpan;
```

```
  };

  // Schedule some time
  scheduler.scheduleAbsolute(1, function () { console.log('foo'); });
  scheduler.scheduleAbsolute(2, function () { console.log('bar'); });
  scheduler.scheduleAbsolute(3, function () { scheduler.stop(); });

  // Start the scheduler
  scheduler.start();

  // => foo
  // => bar

  // Check the clock once stopped
  console.log(scheduler.now());
  // => 3

  console.log(scheduler.clock);
  // => 3
```

Location

- rx.virtualtime.js

VirtualTimeScheduler Constructor

- constructor

VirtualTimeScheduler Instance Methods

- advanceBy
- advanceTo
- scheduleAbsolute
- scheduleAbsoluteWithState
- scheduleRelative
- scheduleRelativeWithState
- sleep
- start
- stop

VirtualTimeScheduler Instance Properties

- isEnabled

VirtualTimeScheduler Protected Abstract Methods

- `add`
- `toDateTimeOffset`
- `toRelative`

VirtualTimeScheduler Protected Methods

- `getNext`

Inherited Classes

- `Rx.Scheduler`

VirtualTimeScheduler Constructor

Rx.VirtualTimeScheduler(initialClock, comparer)

Ⓢ

Creates a new virtual time scheduler with the specified initial clock value and absolute time comparer.

Arguments

1. `initialClock` *(Function)*: Initial value for the clock.
2. `comparer` *(Function)*: Comparer to determine causality of events based on absolute time.

Example

```
function comparer (x, y) {
    if (x > y) { return 1; }
    if (x < y) { return -1; }
    return 0;
}


var scheduler = new Rx.VirtualTimeScheduler(
    0,          /* initial clock of 0 */
    comparer    /* comparer for determining order */
);
```

Location

- rx.virtualtime.js

VirtualTimeScheduler Instance Methods

`Rx.VirtualTimeScheduler.prototype.advanceBy(time)`

Ⓢ

Advances the scheduler's clock by the specified relative time, running all work scheduled for that timespan.

Arguments

1. `time` *(Any)*: Relative time to advance the scheduler's clock by.

Example

```
var scheduler = new MyVirtualScheduler(
    200 /* initial time */
);

scheduler.scheduleAbsolute(250, function () {
    console.log('hello');
});

scheduler.advanceBy(300);
// => hello

console.log(scheduler.clock);
// => 500
```

Location

- rx.virtualtime.js

Rx.VirtualTimeScheduler.prototype.advanceTo(time)

Ⓢ

Advances the scheduler's clock to the specified time, running all work till that point.

Arguments

1. `time` *(Any)*: Absolute time to advance the scheduler's clock to.

Example

```
var scheduler = new MyVirtualScheduler(
    0 /* initial time */
);

scheduler.scheduleAbsolute(100, function () {
    console.log('hello');
});

scheduler.scheduleAbsolute(200, function () {
    console.log('world');
});

scheduler.advanceBy(300);
// => hello
// => world

console.log(scheduler.clock);
// => 300
```

Location

- rx.virtualtime.js

Rx.VirtualTimeScheduler.prototype.scheduleAbsolute(dueTime, action)

Ⓢ

Schedules an action to be executed at dueTime.

Arguments

1. `dueTime` *(Any)*: Absolute time at which to execute the action.
2. `action` *(Function)*: Action to be executed.

Returns

(Disposable): The disposable object used to cancel the scheduled action (best effort).

Example

```
var scheduler = new MyVirtualScheduler(
    0 /* initial time */
);

scheduler.scheduleAbsolute(100, function () {
    console.log('hello');
});

scheduler.scheduleAbsolute(200, function () {
    console.log('world');
});

scheduler.advanceBy(300);
// => hello
// => world

console.log(scheduler.clock);
// => 300
```

Location

- rx.virtualtime.js

Rx.VirtualTimeScheduler.prototype.scheduleAbsoluteWithState(state, dueTime, action)

Ⓢ

Schedules an action to be executed at dueTime.

Arguments

1. `state` : *(Any)*: State passed to the action to be executed.
2. `dueTime` *(Any)*: Absolute time at which to execute the action.
3. `action` *(Function)*: Action to be executed.

Returns

(Disposable): The disposable object used to cancel the scheduled action (best effort).

Example

```
var scheduler = new MyVirtualScheduler(
    0 /* initial time */
);

scheduler.scheduleAbsoluteWithState('world', 100, function (x) {
    console.log('hello ' + x);
});

scheduler.scheduleAbsoluteWithState(200, function () {
    console.log('goodnight ' + x);
}, 'moon');

scheduler.start();
// => hello world
// => goodnight moon

console.log(scheduler.clock);
// => 200
```

Location

- rx.virtualtime.js

Rx.VirtualTimeScheduler.prototype.scheduleRelative(dueTime, action)

\# Ⓢ

Schedules an action to be executed at dueTime.

Arguments

1. `dueTime` *(Any)*: Relative time after which to execute the action.
2. `action` *(Function)*: Action to be executed.

Returns

(Disposable): The disposable object used to cancel the scheduled action (best effort).

Example

```
var scheduler = new MyVirtualScheduler(
    100 /* initial time */
);

scheduler.scheduleRelative(100, function () {
    console.log('hello');
});

scheduler.scheduleRelative(200, function () {
    console.log('world');
});

scheduler.start();
// => hello
// => world

console.log(scheduler.clock);
// => 400
```

Location

- rx.virtualtime.js

Rx.VirtualTimeScheduler.prototype.scheduleRelativeWithState(state, dueTime, action)

Ⓢ

Schedules an action to be executed at dueTime.

Arguments

1. `state` : *(Any)*: State passed to the action to be executed.
2. `dueTime` *(Any)*: Relative time after which to execute the action.
3. `action` *(Function)*: Action to be executed.

Returns

(Disposable): The disposable object used to cancel the scheduled action (best effort).

Example

```
var scheduler = new MyVirtualScheduler(
    0 /* initial time */
);

scheduler.scheduleRelativeWithState('world', 100, function (x) {
    console.log('hello ' + x);
});

scheduler.scheduleRelativeWithState('moon', 200, function () {
    console.log('goodnight ' + x);
});

scheduler.start();
// => hello world
// => goodnight moon

console.log(scheduler.clock);
// => 300
```

Location

- rx.virtualtime.js

Rx.VirtualTimeScheduler.prototype.sleep(time)

Ⓢ

Advances the scheduler's clock by the specified relative time.

Arguments

1. `time` *(Any)*: Relative time to advance the scheduler's clock by.

Example

```
var scheduler = new MyVirtualScheduler(
    0 /* initial time */
);

scheduler.sleep(400);

console.log(scheduler.clock);
// => 400
```

Location

- rx.virtualtime.js

Rx.VirtualTimeScheduler.prototype.start()

\# Ⓢ

Starts the virtual time scheduler.

Example

```javascript
var scheduler = new MyVirtualScheduler(
    0 /* initial time */
);

scheduler.scheduleRelativeWithState('world', 100, function (x) {
    console.log('hello ' + x);
});

scheduler.scheduleRelativeWithState('moon', 200, function () {
    console.log('goodnight ' + x);
});

scheduler.start();
// => hello world
// => goodnight moon

console.log(scheduler.clock);
// => 400
```

Location

- rx.virtualtime.js

Rx.VirtualTimeScheduler.prototype.stop()

\# Ⓢ

Stops the virtual time scheduler.

Example

```
var scheduler = new MyVirtualScheduler(
    0 /* initial time */
);

scheduler.scheduleRelative(100, function () {
    console.log('hello world');
});

scheduler.scheduleRelative(100, function () {
    scheduler.stop();
});

scheduler.scheduleRelative(100, function () {
    console.log('hello world');
});

scheduler.start();
// => hello world
```

Location

- rx.virtualtime.js

VirtualTimeScheduler Abstract Protected Methods

`Rx.VirtualTimeScheduler.prototype.add(absolute, relative)`

\# Ⓢ

Adds a relative time value to an absolute time value. This method is used in several methods including `scheduleRelativeWithState` , `advanceBy` and `sleep` .

Arguments

1. `absolute` *(Any)*: Absolute virtual time value.
2. `relative` *(Any)*: Relative virtual time value.

Returns

(Any): Resulting absolute virtual time sum value.

Example

One possible implementation could be as simple as the following:

```
scheduler.add = function (absolute, relative) {
    return absolute + relative;
};
```

Location

* rx.virtualtime.js

Rx.VirtualTimeScheduler.prototype.toDateTimeOffset(absolute)

Ⓢ

Converts an absolute time to a number. This is used directly in the `now` method on the `Rx.Scheduler`

Arguments

1. `absolute` *(Any)*: The absolute time to convert.

Returns

(Number): The absolute time in ms.

Example

One possible implementation could be as simple as the following:

```
// String -> Number
scheduler.toDateTimeOffset = function (absolute) {
    return return absolute.length;
};
```

Location

* rx.virtualtime.js

Rx.VirtualTimeScheduler.prototype.toRelative(timeSpan)

Ⓢ

Converts the time span number/Date to a relative virtual time value.

Arguments

1. `timeSpan` *(Any)*: The time span number value to convert. This is used directly in `scheduleWithRelativeAndState` and `scheduleWithAbsoluteAndState` .

Returns

(Number): Corresponding relative virtual time value.

Example

One possible implementation could be as simple as the following:

```
// Number -> Number
scheduler.toRelative = function (timeSpan) {
    return timeSpan;
};
```

Location

- rx.virtualtime.js

Disposables

- Rx.CompositeDisposable
- Rx.Disposable
- Rx.RefCountDisposable
- Rx.SerialDisposable
- Rx.SingleAssignmentDisposable

`Rx.CompositeDisposable` class

Represents a group of disposable resources that are disposed together.

Usage

The follow example shows the basic usage of an Rx.CompositeDisposable.

```javascript
var d1 = Rx.Disposable.create(() => console.log('one'));

var d2 = Rx.Disposable.create(() => console.log('two'));

// Initialize with two disposables
var disposables = new Rx.CompositeDisposable(d1, d2);

disposables.dispose();
// => one
// => two
```

Location

- rx.js

CompositeDisposable Constructor

- constructor

CompositeDisposable Instance Methods

- add
- clear
- contains
- dispose
- remove
- toArray

CompositeDisposable Instance Properties

- `isDisposed`
- `length`

CompositeDisposable Constructor

`Rx.CompositeDisposable(...args)`

\# Ⓢ

Initializes a new instance of the `Rx.CompositeDisposable` class from a group of disposables.

Arguments

1. `args` *(Array|arguments)*: Disposables that will be disposed together.

Example

```
var d1 = Rx.Disposable.create(() => console.log('one'));

var d2 = Rx.Disposable.create(() => console.log('two'));

// Initialize with two disposables
var disposables = new Rx.CompositeDisposable(d1, d2);

disposables.dispose();
// => one
// => two
```

Location

- rx.js

CompositeDisposable Instance Methods

`Rx.CompositeDisposable.prototype.add(item)`

\# Ⓢ

Adds a disposable to the CompositeDisposable or disposes the disposable if the CompositeDisposable is disposed.

Arguments

1. `item` *(Disposable)*: Disposable to add.

Example

```
var disposables = new Rx.CompositeDisposable();

var d1 = Rx.Disposable.create(() => console.log('one'));

disposables.add(d1);

disposables.dispose();
// => one
```

Location

- rx.js

Rx.CompositeDisposable.prototype.clear()

Ⓢ

Removes and disposes all disposables from the CompositeDisposable, but does not dispose the CompositeDisposable.

Example

```
var d1 = Rx.Disposable.create(() => console.log('one'));

var disposables = new Rx.CompositeDisposable(d1);

console.log(disposables.length);
// => 1

disposables.clear();
// => one

console.log(disposables.length);
// => 0
```

Location

- rx.js

Rx.CompositeDisposable.prototype.contains(item)

Ⓢ

Determines whether the CompositeDisposable contains a specific disposable.

Arguments

1. `item` *(Disposable)*: Disposable to search for.

Returns

(Boolean): `true` if the disposable was found; otherwise, `false`.

Example

```javascript
var disposables = new Rx.CompositeDisposable();

var d1 = Rx.Disposable.create(() => console.log('one'));

disposables.add(d1);

console.log(disposables.contains(d1));
// => true
```

Location

- rx.js

Rx.CompositeDisposable.prototype.dispose()

Ⓢ

Disposes all disposables in the group and removes them from the group.

Example

```
var d1 = Rx.Disposable.create(() => console.log('one'));

var d2 = Rx.Disposable.create(() => console.log('two'));

var disposables = new Rx.CompositeDisposable(d1, d2);

disposables.dispose();
// => one
// => two

console.log(disposables.length);
// => 0
```

Location

- rx.js

Rx.CompositeDisposable.prototype.remove(item)

Ⓢ

Removes and disposes the first occurrence of a disposable from the CompositeDisposable.

Arguments

1. `item` *(Disposable)*: Disposable to remove.

Returns

(Boolean): `true` if the disposable was found and disposed; otherwise, `false`.

Example

```
var disposables = new Rx.CompositeDisposable();

var d1 = Rx.Disposable.create(() => console.log('one'));

disposables.add(d1);

console.log(disposables.remove(d1));
// => true
```

Location

- rx.js

`Rx.CompositeDisposable.prototype.rxcompositedisposableprototypetoarray()`

Ⓢ

Converts the existing CompositeDisposable to an array of disposables

Returns

(Array): An array of disposable objects.

Example

```
var d1 = Rx.Disposable.create(() => console.log('one'));

var d2 = Rx.Disposable.create(() => console.log('two'));

var disposables = new Rx.CompositeDisposable(d1, d2);

var array = disposables.toArray();

console.log(array.length);
// => 2
```

Location

- rx.js

CompositeDisposable Instance Properties

`isDisposed`

Ⓢ

Gets a value that indicates whether the object is disposed.

Example

```
var disposables = new Rx.CompositeDisposable();

var d1 = Rx.Disposable.create(() => console.log('disposed'));

disposables.add(d1);

console.log(disposables.isDisposed);
// => false

disposables.dispose();
// => disposed

console.log(disposables.isDisposed);
// => true
```

Location

- rx.js

`length`

Ⓢ

Gets the number of disposables in the CompositeDisposable.

Example

```
var disposables = new Rx.CompositeDisposable();

var d1 = Rx.Disposable.create(() => console.log('disposed'));

disposables.add(d1);

console.log(disposables.length);
// => 1

disposables.dispose();
// => disposed

console.log(disposables.length);
// => 0
```

Location

- rx.js

`Rx.Disposable` class

Provides a set of static methods for creating Disposables, which defines a method to release allocated resources.

Usage

The follow example shows the basic usage of an `Rx.Disposable`.

```javascript
var disposable = Rx.Disposable.create(() => console.log('disposed'));

disposable.dispose();
// => disposed
```

Location

- rx.js

Disposable Class Methods

- `create`

Disposable Class Properties

- `empty`

Disposable Instance Methods

- `dispose`

Class Methods

`Rx.Disposable.create(action)`

Ⓢ

Creates a disposable object that invokes the specified action when disposed.

Arguments

1. `action` *(Function)*: Function to run during the first call to `dispose`. The action is guaranteed to be run at most once.

Returns

(Disposable): The disposable object that runs the given action upon disposal.

Example

```
var disposable = Rx.Disposable.create(() => console.log('disposed'));

disposable.dispose();
// => disposed
```

Location

- rx.js

Disposable Class Properties

`Rx.Disposable.empty`

\# Ⓢ

Gets the disposable that does nothing when disposed.

Returns

(Disposable): The disposable that does nothing when disposed.

Example

```
var disposable = Rx.Disposable.empty;

disposable.dispose(); // Does nothing
```

Location

- rx.js

Disposable Instance Methods

`Rx.Disposable.prototype.dispose()`

Ⓢ

Performs the task of cleaning up resources.

Example

```
var disposable = Rx.Disposable.create(() => console.log('disposed'));

disposable.dispose();
// => disposed
```

Location

- rx.js

`Rx.RefCountDisposable` class

Represents a disposable resource that only disposes its underlying disposable resource when all `getDisposable` dependent disposable objects have been disposed.

Usage

The follow example shows the basic usage of an `Rx.RefCountDisposable` .

```
var disposable = Rx.Disposable.create(() => console.log('disposed'));

var refCountDisposable = new Rx.RefCountDisposable(disposable);

// Try disposing before the underlying is disposed
refCountDisposable.dispose();

console.log(refCountDisposable.isDisposed);
// => false

// Dispose the underlying disposable
disposable.dispose();
// => disposed

// Now dispose the primary
refCountDisposable.dispose();

console.log(refCountDisposable.isDisposed);
// => true
```

Location

- rx.js

RefCountDisposable Constructor

- constructor

RefCountDisposable Instance Methods

- dispose

- getDisposable

RefCountDisposable Instance Properties

- isDisposed

RefCountDisposable Constructor

Rx.RefCountDisposable(disposable)

\# Ⓢ

Initializes a new instance of the `Rx.RefCountDisposable` class with the specified disposable

Arguments

1. `disposable` *(Disposable)*: Underlying disposable.

Example

```
var disposable = Rx.Disposable.create(() => console.log('disposed'));

var refCountDisposable = new Rx.RefCountDisposable(disposable);

console.log(refCountDisposable.isDisposed);
// => false
```

Location

- rx.js

RefCountDisposable Instance Methods

Rx.RefCountDisposable.prototype.dispose()

\# Ⓢ

Disposes the underlying disposable only when all dependent disposables have been disposed.

Example

```
var disposable = Rx.Disposable.create(() => console.log('disposed'));

var refCountDisposable = new Rx.RefCountDisposable(disposable);

// Try disposing before the underlying is disposed
refCountDisposable.dispose();

console.log(refCountDisposable.isDisposed);
// => false

// Dispose the underlying disposable
disposable.dispose();
// => disposed

// Now dispose the primary
refCountDisposable.dispose();

console.log(refCountDisposable.isDisposed);
// => true
```

Location

- rx.js

Rx.RefCountDisposable.prototype.getDisposable()

Ⓢ

Returns a dependent disposable that when disposed decreases the refcount on the underlying disposable.

Returns

(Disposable): A dependent disposable contributing to the reference count that manages the underlying disposable's lifetime.

Example

```
var disposable = Rx.Disposable.create(() => console.log('disposed'));

var refCountDisposable = new Rx.RefCountDisposable(disposable);

var d = refCountDisposable.getDisposable();

console.log(d === disposable);
// => false

// Clean up disposables
disposable.dispose();
d.dispose();

// Now try to dispose the main
refCountDisposable.dispose();

console.log(refCountDisposable.isDisposed);
// => true
```

Location

- rx.js

RefCountDisposable Instance Properties

`isDisposed`

ⓢ

Gets a value that indicates whether the object is disposed.

Example

```
var disposable = Rx.Disposable.create(() => console.log('disposed'));

var refCountDisposable = new Rx.RefCountDisposable(disposable);

disposable.dispose();

console.log(refCountDisposable.isDisposed);
// => false

refCountDisposable.dispose();
// => disposed

console.log(refCountDisposable.isDisposed);
// => true
```

Location

- rx.js

`Rx.SerialDisposable` class

Represents a disposable resource whose underlying disposable resource can be replaced by another disposable resource, causing automatic disposal of the previous underlying disposable resource.

Usage

The follow example shows the basic usage of an Rx.SerialDisposable.

```
var serialDisposable = new Rx.SerialDisposable();

var d1 = Rx.Disposable.create(() => console.log('one'));

serialDisposable.setDisposable(d1);

var d2 = Rx.Disposable.create(() => console.log('two'));

serialDisposable.setDisposable(d2);
// => one

serialDisposable.dispose();
// = two
```

Location

- rx.js

SerialDisposable Constructor

- constructor

SerialDisposable Instance Methods

- dispose
- getDisposable
- setDisposable

SerialDisposable Instance Properties

- `isDisposed`

SerialDisposable Constructor

Rx.SerialDisposable()

\# Ⓢ

Initializes a new instance of the `Rx.SerialDisposable` class.

Example

```
var serialDisposable = new Rx.SerialDisposable();

console.log(serialDisposable.isDisposed);
// => false
```

Location

- rx.js

SerialDisposable Instance Methods

Rx.SerialDisposable.prototype.dispose()

\# Ⓢ

Disposes the underlying disposable as well as all future replacements.

Example

```
var serialDisposable = new Rx.SerialDisposable();

var d1 = Rx.Disposable.create(() => console.log('one'));

serialDisposable.setDisposable(disposable);

serialDisposable.dispose();
// => one

var d2 = Rx.Disposable.create(() => console.log('two'));

// => two
```

Location

- rx.js

Rx.SerialDisposable.prototype.getDisposable()

Ⓢ

Gets the underlying disposable.

Returns

(Disposable): The underlying disposable.

Example

```
var serialDisposable = new Rx.SerialDisposable();

var disposable = Rx.Disposable.create(() => console.log('disposed'));

serialDisposable.setDisposable(disposable);

var d = serialDisposable.getDisposable();

console.log(d === disposable);
```

Location

- rx.js

`Rx.SerialDisposable.prototype.setDisposable(value)`

\# Ⓢ

Sets the underlying disposable.

Arguments

1. `value` *(Disposable)*: The new underlying disposable.

Example

```javascript
var serialDisposable = new Rx.SerialDisposable();

var d1 = Rx.Disposable.create(() => console.log('one'));

serialDisposable.setDisposable(d1);

serialDisposable.dispose();
// => one

var d2 = Rx.Disposable.create(() => console.log('two'));

serialDisposable.setDisposable(d2);
// => two
```

Location

- rx.js

SerialDisposable Instance Properties

`isDisposed`

\# Ⓢ

Gets a value that indicates whether the object is disposed.

Example

```
var serialDisposable = new Rx.SerialDisposable();

var disposable = Rx.Disposable.create(() => console.log('disposed'));

serialDisposable.setDisposable(disposable);

console.log(serialDisposable.isDisposed);
// => false

serialDisposable.dispose();
// => disposed

console.log(serialDisposable.isDisposed);
// => true
```

Location

- rx.js

`Rx.SingleAssignmentDisposable` class

Represents a disposable resource which only allows a single assignment of its underlying disposable resource. If an underlying disposable resource has already been set, future attempts to set the underlying disposable resource will throw an Error.

Usage

The follow example shows the basic usage of an Rx.SingleAssignmentDisposable.

```javascript
var singleDisposable = new Rx.SingleAssignmentDisposable();

var disposable = Rx.Disposable.create(() => console.log('disposed'));

singleDisposable.setDisposable(disposable);

singleDisposable.dispose();
// => disposed
```

Location

- rx.js

SingleAssignmentDisposable Constructor

- `constructor`

SingleAssignmentDisposable Instance Methods

- `dispose`
- `getDisposable`
- `setDisposable`

SingleAssignmentDisposable Instance Properties

- `isDisposed`

SingleAssignmentDisposable Constructor

```
Rx.SingleAssignmentDisposable()
```

\# ⓢ

Initializes a new instance of the `Rx.SingleAssignmentDisposable` class.

Example

```javascript
var singleDisposable = new Rx.SingleAssignmentDisposable();

console.log(singleDisposable.isDisposed);
// => false
```

Location

- rx.js

SingleAssignmentDisposable Instance Methods

```
Rx.SingleAssignmentDisposable.prototype.dispose()
```

\# ⓢ

Disposes the underlying disposable.

Example

```
var singleDisposable = new Rx.SingleAssignmentDisposable();

var disposable = Rx.Disposable.create(() => console.log('disposed'));

singleDisposable.setDisposable(disposable);

console.log(singleDisposable.isDisposed);
// => false

singleDisposable.dispose();
// => disposed

console.log(singleDisposable.isDisposed);
// => true
```

Location

- rx.js

Rx.SingleAssignmentDisposable.prototype.getDisposable()

Ⓢ

Gets the underlying disposable. After disposal, the result of getting this method is undefined.

Returns

(Disposable): The underlying disposable.

Example

```
var singleDisposable = new Rx.SingleAssignmentDisposable();

var disposable = Rx.Disposable.create(() => console.log('disposed'));

singleDisposable.setDisposable(disposable);

var d = singleDisposable.getDisposable();

console.log(d === disposable);
```

Location

- rx.js

`Rx.SingleAssignmentDisposable.prototype.setDisposable(value)`

Ⓢ

Sets the underlying disposable.

Arguments

1. `value` *(Disposable)*: The new underlying disposable.

Example

```
var singleDisposable = new Rx.SingleAssignmentDisposable();

var d1 = Rx.Disposable.create(() => console.log('one'));

singleDisposable.setDisposable(d1);

var d2 = Rx.Disposable.create(() => console.log('two'));

try {
    singleDisposable.setDisposable(d2);
} catch (e) {
    console.log(e.message);
}

// => Disposable has already been assigned
```

Location

- rx.js

SingleAssignmentDisposable Instance Properties

`isDisposed`

Ⓢ

Gets a value that indicates whether the object is disposed.

Example

```javascript
var singleDisposable = new Rx.SingleAssignmentDisposable();

var disposable = Rx.Disposable.create(() => console.log('disposed'));

singleDisposable.setDisposable(disposable);

console.log(singleDisposable.isDisposed);
// => false

singleDisposable.dispose();
// => disposed

console.log(singleDisposable.isDisposed);
// => true
```

Location

- rx.js

Testing

`Rx.ReactiveTest` class

This class contains test utility methods such as create notifications for testing purposes.

Location

- rx.testing.js

ReactiveTest Class Methods

- onCompleted
- onError
- onNext
- subscribe

ReactiveTest Class Fields

- created
- disposed
- subscribed

ReactiveTest Class Methods

`Rx.ReactiveTest.onCompleted(ticks)`

\# Ⓢ

Factory method for an OnCompleted notification record at a given time.

Arguments

1. `ticks` *(Number)*: Recorded virtual time the OnCompleted notification occurs.

Returns

(Recorded): OnCompleted notification.

Example

```
var onCompleted = Rx.ReactiveTest.onCompleted;

var scheduler = new Rx.TestScheduler();

var xs = scheduler.createHotObservable(
    onCompleted(260)
);

var res = scheduler.startWithCreate(function () {
    return xs.map(function (x) { return x; });
});

// Write custom assertion
collectionAssert(res, [
    onCompleted(260)
]);
```

Location

- rx.testing.js

Rx.ReactiveTest.onError(ticks, exception)

Ⓢ

Factory method for an OnError notification record at a given time with a given error.

Arguments

1. `ticks` *(Number)*: Recorded virtual time the OnError notification occurs.
2. `exception` *(Error | Function)*: Recorded exception stored in the OnError notification or a predicate

Returns

(Recorded): Recorded OnError notification.

Example

```
var ex = new Error('woops');

var onError = Rx.ReactiveTest.onError;

var scheduler = new Rx.TestScheduler();

var xs = scheduler.createHotObservable(
    onError(201, ex)
);

var res = scheduler.startWithCreate(function () {
    return xs.map(function (x) { return x; });
});

// Write custom assertion
collectionAssert(res, [

    // Using a predicate
    onError(201, function (e) { return e.message === 'woops'; })
]);
```

Location

- rx.testing.js

Rx.ReactiveTest.onNext(ticks, value)

\# Ⓢ

Factory method for an OnNext notification record at a given time with a given error.

Arguments

1. `ticks` *(Number)*: Recorded virtual time the OnNext notification occurs.
2. `value` *(Any | Function)*: Recorded exception stored in the OnNext notification or a predicate

Returns

(Recorded): Recorded OnNext notification.

Example

```
var ex = new Error('woops');

var onNext = Rx.ReactiveTest.onNext;

var scheduler = new Rx.TestScheduler();

var xs = scheduler.createHotObservable(
    onNext(201, 42)
);

var res = scheduler.startWithCreate(function () {
    return xs.map(function (x) { return x; });
});

// Write custom assertion
collectionAssert(res, [

    // Using a predicate
    onNext(201, function (x) { return x === 42; })
]);
```

Location

- rx.testing.js

ReactiveTest Class Fields

Rx.ReactiveTest.created

Ⓢ

Default virtual time used for creation of observable sequences in unit tests. This has a value of `100` .

Example

```
var scheduler = new Rx.TestScheduler();

var xs = scheduler.createHotObservable(
    Rx.ReactiveTest.onNext(201, 42),
    Rx.ReactiveTest.onNext(202, 56),
    Rx.ReactiveTest.onCompleted(203)
);

var res = scheduler.startWithTiming(
    function () { return xs.map(funxtion (x) { return x; })},
    Rx.ReactiveTest.created,
    Rx.ReactiveTest.subscribed,
    Rx.ReactiveTest.disposed
);
```

Location

- rx.testing.js

`Rx.ReactiveTest.disposed`

ⓢ

Default virtual time used to dispose subscriptions in unit tests. This has a value of `1000` .

Example

```
var scheduler = new Rx.TestScheduler();

var xs = scheduler.createHotObservable(
    Rx.ReactiveTest.onNext(201, 42),
    Rx.ReactiveTest.onNext(202, 56),
    Rx.ReactiveTest.onCompleted(203)
);

var res = scheduler.startWithTiming(
    function () { return xs.map(funxtion (x) { return x; })},
    Rx.ReactiveTest.created,
    Rx.ReactiveTest.subscribed,
    Rx.ReactiveTest.disposed
);
```

Location

- rx.testing.js

`Rx.ReactiveTest.subscribed`

\# Ⓢ

Default virtual time used to subscribe to observable sequences in unit tests. This has a value of `200` .

Example

```
var scheduler = new Rx.TestScheduler();

var xs = scheduler.createHotObservable(
    Rx.ReactiveTest.onNext(201, 42),
    Rx.ReactiveTest.onNext(202, 56),
    Rx.ReactiveTest.onCompleted(203)
);

var res = scheduler.startWithTiming(
    function () { return xs.map(funxtion (x) { return x; })},
    Rx.ReactiveTest.created,
    Rx.ReactiveTest.subscribed,
    Rx.ReactiveTest.disposed
);
```

Location

- rx.testing.js

`Rx.Recorded` class

Record of a value including the virtual time it was produced on.

Location

- rx.testing.js

Recorded Constructor

- `constructor`

Recorded Instance Methods

- `equals`
- `toString`

Recorded Instance Properties

- `time`
- `value`

Recorded Constructor

`Rx.Recorded(time, value, [comparer])`

\# Ⓢ

Creates a new object recording the production of the specified value at the given virtual time.

Arguments

1. `time` *(Number)*: Virtual time the value was produced on.
2. `value` *(Any)*: Value that was produced
3. `[comparer]` *(Function)*: Optional comparer function.

Example

```
var recorded = new Rx.Recorded(200, 'value');

console.log(recorded.time);
// => 200

console.log(recorded.value);
// => value
```

Location

- rx.js

Recorded Instance Methods

`Rx.Recorded.prototype.equals(other)`

\# Ⓢ

Checks whether the given recorded object is equal to the current instance.

Arguments

1. `other` *(Recorded)*: Recorded object to check for equality.

Returns

(Boolean): Returns `true` if the Recorded equals the other, else `false`.

Example

```
var r1 = new Recorded(201, 'foo');
var r2 = new Recorded(201, 'bar');
var r3 = new Recorded(201, 'foo');

console.log(r1.equals(r2));
// => false

console.log(r1.equals(r3));
// => true
```

Location

- rx.testing.js

`Rx.Recorded.prototype.toString()`

\# Ⓢ

Returns a string representation of the current Recorded value.

Returns

(String): String representation of the current Recorded value.

Example

```
var r1 = new Recorded(201, 'foo');

console.log(r1.toString());
// => foo@201
```

Location

- rx.testing.js

Recorded Instance Properties

`time`

\# Ⓢ

Gets the virtual time the value was produced on.

Returns

(Number): The virtual time the value was produced on.

Example

```
var r1 = new Recorded(201, 'foo');

console.log(r1.time);
// => 201
```

Location

- rx.testing.js

`value`

Ⓢ

Gets the recorded value.

Returns

(Number): The recorded value.

Example

```
var r1 = new Recorded(201, 'foo');

console.log(r1.value;
// => foo
```

Location

- rx.testing.js

`Rx.Subscription` class

Records information about subscriptions to and unsubscriptions from observable sequences.

Location

- rx.testing.js

`Subscription Constructor`

- `constructor`

`Subscription Instance Methods`

- `equals`
- `toString`

`Subscription Instance Properties`

- `subscribe`
- `unsubscribe`

Subscription Constructor

`Rx.Subscription(subscribe, unsubscribe)`

\# Ⓢ

Creates a new subscription object with the given virtual subscription and unsubscription time.

Arguments

1. `subscribe` *(Number)*: Virtual time at which the subscription occurred.
2. `[unsubscribe = Number.MAX_VALUE]` *(Number)*: Virtual time at which the unsubscription occurred.

Example

```
var subscription = new Rx.Subscription(200, 1000);

console.log(recorded.time);
// => 200

console.log(recorded.unsubscribe);
// => 1000
```

Location

- rx.testing.js

Subscription Instance Methods

Rx.Subscription.prototype.equals(other)

ⓢ

Checks whether the given subscription is equal to the current instance.

Arguments

1. `other` *(Subscription)*: Subscription object to check for equality.

Returns

(Boolean): Returns `true` if the Subscription equals the other, else `false`.

Example

```
var s1 = new Subscription(201, 500);
var s2 = new Subscription(201);
var s3 = new Subscription(201, 500);

console.log(s1.equals(s2));
// => false

console.log(s1.equals(s3));
// => true
```

Location

- rx.testing.js

`Rx.Subscription.prototype.toString()`

Ⓢ

Returns a string representation of the current Subscription value.

Returns

(String): String representation of the current Subscription value.

Example

```
var s1 = new Subscription(201);

console.log(s1.toString());
// => (201, Infinite)

var s2 = new Subscription(201, 1000);
console.log(s2.toString());
// => (201, 1000)
```

Location

- rx.testing.js

Subscription Instance Properties

`subscribe`

Ⓢ

Gets the subscription virtual time.

Returns

(Number): The subscription virtual time.

Example

```
var s1 = new Subscription(201);

console.log(s1.subscribe);
// => 201
```

Location

- rx.testing.js

`unsubscribe`

Ⓢ

Gets the unsubscription virtual time.

Returns

(Number): The unsubscription virtual time.

Example

```
var s1 = new Subscription(201, 500);

console.log(r1.unsubscribe);
// => foo
```

Location

- rx.testing.js

`Rx.TestScheduler` class

Virtual time scheduler used for testing applications and libraries built using Reactive Extensions. This inherits from the `Rx.TestScheduler` class.

Usage

The following shows an example of using the `Rx.TestScheduler`. In order to make the end comparisons work, you must implement a collection assert, for example here using QUnit.

```javascript
function createMessage(actual, expected) {
    return 'Expected: [' + expected.toString() + ']\r\nActual: [' + actual.toString() + '
}

// Using QUnit testing for assertions
var collectionAssert = {
    assertEqual: function (expected, actual) {
        var comparer = Rx.Internals.isEqual,
            isOk = true;

        if (expected.length !== actual.length) {
            ok(false, 'Not equal length. Expected: ' + expected.length + ' Actual: ' + ac
            return;
        }

        for(var i = 0, len = expected.length; i < len; i++) {
            isOk = comparer(expected[i], actual[i]);
            if (!isOk) {
                break;
            }
        }

        ok(isOk, createMessage(expected, actual));
    }
};

var onNext = Rx.ReactiveTest.onNext,
    onCompleted = Rx.ReactiveTest.onCompleted,
    subscribe = Rx.ReactiveTest.subscribe;

var scheduler = new Rx.TestScheduler();

// Create hot observable which will start firing
var xs = scheduler.createHotObservable(
    onNext(150, 1),
    onNext(210, 2),
```

```
        onNext(220, 3),
        onCompleted(230)
    );

    // Note we'll start at 200 for subscribe, hence missing the 150 mark
    var res = scheduler.startWithCreate(function () {
        return xs.map(function (x) { return x * x });
    });

    // Implement collection assertion
    collectionAssert.assertEqual(res.messages, [
        onNext(210, 4),
        onNext(220, 9),
        onCompleted(230)
    ]);

    // Check for subscribe/unsubscribe
    collectionAssert.assertEqual(xs.subscriptions, [
        subscribe(200, 230)
    ]);
```

Location

- rx.testing.js

TestScheduler Constructor

- constructor

TestScheduler Instance Methods

- createColdObservable
- createHotObservable
- createObserver
- startWithCreate
- startWithDispose
- startWithTiming

Inherited Classes

- Rx.VirtualTimeScheduler

TestScheduler Constructor

`Rx.TestScheduler()`

\# Ⓢ

Creates a new virtual time test scheduler.

Example

```javascript
var onNext = Rx.ReactiveTest.onNext,
    onCompleted = Rx.ReactiveTest.onCompleted;

var scheduler = new Rx.TestScheduler();

// Create hot observable which will start firing
var xs = scheduler.createHotObservable(
    onNext(150, 1),
    onNext(210, 2),
    onNext(220, 3),
    onCompleted(230)
);

// Note we'll start at 200 for subscribe, hence missing the 150 mark
var res = scheduler.startWithCreate(function () {
    return xs.map(function (x) { return x * x });
});

// Implement collection assertion
collectionAssert.assertEqual(res.messages, [
    onNext(210, 4),
    onNext(220, 9),
    onCompleted(230)
]);

// Check for subscribe/unsubscribe
collectionAssert.assertEqual(xs.subscriptions, [
    subscribe(200, 230)
]);
```

Location

- rx.testing.js

TestScheduler Instance Methods

 `Rx.TestScheduler.prototype.createColdObservable(...args)`

Ⓢ

Creates a cold observable using the specified timestamped notification messages.

Arguments

1. `args` *(Arguments)*: An arguments array of Recorded objects from `Rx.ReactiveTest.onNext` , `Rx.ReactiveTest.onError` , and `Rx.ReactiveTest.onCompleted` methods.

Returns

(Observable): Cold observable sequence that can be used to assert the timing of subscriptions and notifications.

Example

```
var onNext = Rx.ReactiveTest.onNext,
    onCompleted = Rx.ReactiveTest.onCompleted;

var scheduler = new Rx.TestScheduler();

// Create cold observable with offset from subscribe time
var xs = scheduler.createColdObservable(
    onNext(150, 1),
    onNext(200, 2),
    onNext(250, 3),
    onCompleted(300)
);

// Note we'll start at 200 for subscribe
var res = scheduler.startWithCreate(function () {
    return xs.filter(function (x) { return x % 2 === 0; });
});

// Implement collection assertion
collectionAssert.assertEqual(res.messages, [
    onNext(400, 2),
    onCompleted(500)
]);

// Check for subscribe/unsubscribe
collectionAssert.assertEqual(xs.subscriptions, [
    subscribe(200, 500)
]);
```

Location

- rx.testing.js

`Rx.TestScheduler.prototype.createHotObservable(...args)`

\# Ⓢ

Creates a hot observable using the specified timestamped notification messages.

Arguments

1. `args` *(Arguments)*: An arguments array of Recorded objects from `Rx.ReactiveTest.onNext` , `Rx.ReactiveTest.onError` , and `Rx.ReactiveTest.onCompleted` methods.

Returns

(Observable): Hot observable sequence that can be used to assert the timing of subscriptions and notifications.

Example

```javascript
var onNext = Rx.ReactiveTest.onNext,
    onCompleted = Rx.ReactiveTest.onCompleted;

var scheduler = new Rx.TestScheduler();

// Create hot observable which will start firing
var xs = scheduler.createHotObservable(
    onNext(150, 1),
    onNext(210, 2),
    onNext(220, 3),
    onCompleted(230)
);

// Note we'll start at 200 for subscribe, hence missing the 150 mark
var res = scheduler.startWithCreate(function () {
    return xs.map(function (x) { return x * x });
});

// Implement collection assertion
collectionAssert.assertEqual(res.messages, [
    onNext(210, 4),
    onNext(220, 9),
    onCompleted(230)
]);

// Check for subscribe/unsubscribe
collectionAssert.assertEqual(xs.subscriptions, [
    subscribe(200, 230)
]);
```

Location

- rx.testing.js

Rx.TestScheduler.prototype.createObserver()

Ⓢ

Creates an observer that records received notification messages and timestamps those.

Returns

(Observer): Observer that can be used to assert the timing of received notifications.

Example

```
var onNext = Rx.ReactiveTest.onNext;

var scheduler = new Rx.TestScheduler();

var d = new Rx.SerialDisposable();

var xs = Rx.Observable.return(42, scheduler);

var res = scheduler.createObserver();

scheduler.scheduleAbsolute(100, function () {
    d.setDisposable(xs.subscribe(
        function (x) {
            d.dispose();
            res.onNext(x);
        },
        res.onError.bind(res),
        res.onCompleted.bind(res)
    );
});

scheduler.start();

collectionAssert.assertEqual(res.messages, [
    onNext(101, 42)
]);
```

Location

- rx.testing.js

Rx.TestScheduler.prototype.startWithCreate(create)

Ⓢ

Starts the test scheduler and uses default virtual times to invoke the factory function, to subscribe to the resulting sequence, and to dispose the subscription.

Arguments

1. `create` *(Function)*: Factory method to create an observable sequence.

Returns

(Observer): Observer with timestamped recordings of notification messages that were received during the virtual time window when the subscription to the source sequence was active.

Example

```javascript
var onNext = Rx.ReactiveTest.onNext,
    onCompleted = Rx.ReactiveTest.onCompleted;

var scheduler = new Rx.TestScheduler();

// Create cold observable with offset from subscribe time
var xs = scheduler.createColdObservable(
    onNext(150, 1),
    onNext(200, 2),
    onNext(250, 3),
    onCompleted(300)
);

// Note we'll start at 200 for subscribe
var res = scheduler.startWithCreate(function () {
    return xs.filter(function (x) { return x % 2 === 0; });
});

// Implement collection assertion
collectionAssert.assertEqual(res.messages, [
    onNext(400, 2),
    onCompleted(500)
]);

// Check for subscribe/unsubscribe
collectionAssert.assertEqual(xs.subscriptions, [
    subscribe(200, 500)
]);
```

Location

- rx.testing.js

`Rx.TestScheduler.prototype.startWithDispose(create, disposed)`

Ⓢ

Starts the test scheduler and uses the specified virtual time to dispose the subscription to the sequence obtained through the factory function. Default virtual times are used for factory invocation and sequence subscription.

Arguments

1. `create` *(Function)*: Factory method to create an observable sequence.
2. `disposed` *(Number)*: Virtual time at which to dispose the subscription.

Returns

(Observer): Observer with timestamped recordings of notification messages that were received during the virtual time window when the subscription to the source sequence was active.

Example

```
var onNext = Rx.ReactiveTest.onNext,
    onCompleted = Rx.ReactiveTest.onCompleted;

var scheduler = new Rx.TestScheduler();

// Create hot observable which will start firing
var xs = scheduler.createHotObservable(
    onNext(150, 1),
    onNext(210, 2),
    onNext(220, 3),
    onCompleted(230)
);

// Note we'll start at 200 for subscribe, hence missing the 150 mark
var res = scheduler.startWithDispose(
    function () {
        return xs.map(function (x) { return x * x });
    },
    215 /* Dispose at 215 */
);

// Implement collection assertion
collectionAssert.assertEqual(res.messages, [
    onNext(210, 4),
    onCompleted(215)
]);

// Check for subscribe/unsubscribe
collectionAssert.assertEqual(xs.subscriptions, [
    subscribe(200, 215)
]);
```

Location

- rx.testing.js

Rx.TestScheduler.prototype.startWithTiming(create, created, subscribed, disposed)

\# Ⓢ

Starts the test scheduler and uses the specified virtual times to invoke the factory function, subscribe to the resulting sequence, and dispose the subscription.

Arguments

1. `create` *(Function)*: Factory method to create an observable sequence.
2. `created` *(Number)*: Virtual time at which to invoke the factory to create an observable sequence.
3. `subscribed` *(Number)*: Virtual time at which to subscribe to the created observable sequence.
4. `disposed` *(Number)*: Virtual time at which to dispose the subscription.

Returns

(Observer): Observer with timestamped recordings of notification messages that were received during the virtual time window when the subscription to the source sequence was active.

Example

```
var onNext = Rx.ReactiveTest.onNext,
    onCompleted = Rx.ReactiveTest.onCompleted;

var scheduler = new Rx.TestScheduler();

// Create hot observable which will start firing
var xs = scheduler.createHotObservable(
    onNext(150, 1),
    onNext(210, 2),
    onNext(260, 3),
    onNext(310, 4),
    onCompleted(360)
);

// Note we'll start at 200 for subscribe, hence missing the 150 mark
var res = scheduler.startWithTiming(
    function () {
        return xs.map(function (x) { return x * x });
    },
    100, /* Create at 100    */
    200, /* Subscribe at 200 */
    300  /* Dispose at 300   */
);

// Implement collection assertion
collectionAssert.assertEqual(res.messages, [
    onNext(210, 4),
    onNext(260, 9),
    onCompleted(300)
]);

// Check for subscribe/unsubscribe
collectionAssert.assertEqual(xs.subscriptions, [
    subscribe(200, 300)
]);
```

Location

- rx.testing.js

RxJS Bindings

- DOM
- jQuery
- AngularJS
- Facebook React
- Ractive.js
- NodeJS

`DOM` *(plugin* `RxJS-DOM` *)*

Reactive Extensions (Rx) is a library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators.

Data sequences can take many forms, such as a stream of data from a file or web service, web services requests, system notifications, or a series of events such as user input.

Reactive Extensions represents all these data sequences as observable sequences. An application can subscribe to these observable sequences to receive asynchronous notifications as new data arrive.

This library provides bridges to common DOM related features such as events, Ajax requests, JSONP requests, and HTML5 features like WebSockets, Web Workers, Geolocation, MutationObservers and more.

Reactive Extensions Binding for the DOM (RxJS-DOM) API

This section contains the reference documentation for the Reactive Extensions for the DOM class library.

Ajax

- `Rx.DOM.Request.ajax`
- `Rx.DOM.Request.ajaxCold`
- `Rx.DOM.Request.get`
- `Rx.DOM.Request.getJSON`
- `Rx.DOM.Request.post`

JSONP

- `Rx.DOM.Request.jsonpRequest`
- `Rx.DOM.Request.jsonpRequestCold`

Web Sockets

- `Rx.DOM.fromWebSocket`

Web Workers

- `Rx.DOM.fromWebWorker`

Mutation Observers

- `Rx.DOM.fromMutationObserver`

Geolocation

- `Rx.DOM.Geolocation.getCurrentPosition`
- `Rx.DOM.Geolocation.watchPosition`

Schedulers

- `Rx.Schedulers.requestAnimationFrame`
- `Rx.Schedulers.mutationObserver`

Example

Ajax

- `Rx.DOM.Request.ajax`
- `Rx.DOM.Request.ajaxCold`
- `Rx.DOM.Request.get`
- `Rx.DOM.Request.getJSON`
- `Rx.DOM.Request.post`

- `Rx.DOM.Request.ajax`

`Rx.DOM.Request.ajax(url | settings)`

#Ⓢ [Ⓣ][1]

Creates a hot observable for an Ajax request with either a settings object with url, headers, etc or a string for a URL.

Arguments

1. `url` *(String)*: A string of the URL to make the Ajax call.
2. `settings` *(Object)*: An object with the following properties

```
- `url` *(String)*: URL of the request
  - `method` *(String)*: Method of the request, such as GET, POST, PUT, PATCH, DELET
  - `async` *(Boolean)*: Whether the request is async
  - `headers` *(Object)*: Optional headers
```

Returns

(Observable): An observable sequence containing the `XMLHttpRequest` .

Example

The following example uses a simple URL to retrieve a list of products.

```javascript
Rx.DOM.Request.ajax('/products')
    .subscribe(
        function (xhr) {

            var products = JSON.parse(xhr.responseText);

            products.forEach(function (product) {
                console.log(product);
            });
        },
        function (error) {
            // Log the error
        }
    );
```

`Rx.DOM.Request.ajaxCold(url | settings)`

#Ⓢ [Ⓣ][1]

Creates a cold observable for an Ajax request with either a settings object with url, headers, etc or a string for a URL.

Syntax

```
// Using string URL
Rx.DOM.Request.ajaxCold(url);


// Using settings object
Rx.DOM.Request.ajaxCold(settings);
```

Arguments

1. `url` *(String)*: A string of the URL to make the Ajax call.
2. `settings` *(Object)*: An object with the following properties

   ```
   - `url` *(String)*: URL of the request
     - `method` *(String)*: Method of the request, such as GET, POST, PUT, PATCH, DELET
     - `async` *(Boolean)*: Whether the request is async
     - `headers` *(Object)*: Optional headers
   ```

Returns

(Observable): An observable sequence containing the `XMLHttpRequest`.

Example

The following example uses a simple URL to retrieve a list of products.

```
Rx.DOM.Request.ajaxCold('/products')
    .subscribe(
        function (xhr) {

            var products = JSON.parse(xhr.responseText);

            products.forEach(function (product) {
                console.log(product);
            });
        },
        function (error) {
            // Log the error
        }
    );
```

```
Rx.DOM.Request.ajaxCold('/products')
    .subscribe(
        function (xhr) {
```

`Rx.DOM.Request.get(url)`

#ⓢ [Ⓣ][1]

Creates an observable sequence from an Ajax GET Request with the body. This method is just shorthand for the `Rx.DOM.Request.ajax` method with the GET method.

Arguments

1. `url` *(String)*: A string of the URL to make the Ajax call.

Returns

(Observable): The observable sequence which contains the response from the Ajax GET.

Example

```javascript
Rx.DOM.Request.get('/products')
    .subscribe(
        function (xhr) {
            var text = xhr.responseText;
            console.log(text);
        },
        function (err) {
            // Log the error
        }
    );
```

`Rx.DOM.Request.getJSON(url)`

#Ⓢ [Ⓣ][1]

Creates an observable sequence from JSON from an Ajax request.

Arguments

1. `url` *(String)*: A string of the URL to make the Ajax call.

Returns

(Observable): The observable sequence which contains the parsed JSON.

Example

```
Rx.DOM.Request.get('/products')
    .subscribe(
        function (data) {
            // Log data length
            console.log(data.length);
        },
        function (err) {
            // Log the error
        }
    );
```

Rx.DOM.Request.post(url, [body])

#Ⓢ [Ⓣ][1]

Creates an observable sequence from an Ajax POST Request with the body. This method is just shorthand for the `Rx.DOM.Request.ajax` method with the POST method.

Syntax

```
Rx.DOM.Request.post(url, body);
```

Arguments

1. `url` *(String)*: A string of the URL to make the Ajax call.
2. `[body]` *(Object)*: The body to post

Returns

(Observable): The observable sequence which contains the response from the Ajax POST.

Example

```
Rx.DOM.Request.post('/test')
    .subscribe(
        function (xhr) {
            console.log(xhr.responseText);
        },
        function (err) {
            // Log the error
        }
    );
```

JSONP

- `Rx.DOM.Request.jsonpRequest`
- `Rx.DOM.Request.jsonpRequestCold`

`Rx.DOM.Request.jsonpRequest(url | settings)`

#Ⓢ [Ⓣ][1]

Creates a hot observable JSONP Request with the specified settings or a string URL. **Note when using the method with a URL, it must contain JSONPRequest=?.**

Syntax

This method has two versions, one with a string URL, the other with a settings object.

```
// With a string URL
Rx.DOM.Request.jsonpRequest(url);

// With a settings object
Rx.DOM.Request.jsonpRequest(settings);
```

Arguments

1. `url` *(String)*: A string of the URL to make the JSONP call.
2. `settings` *(Object)*: An object with the following properties:

   ```
     - `url` *(String)*: URL of the request
       - `jsonp` *(String)*: The named callback parameter for the JSONP call
   ```

Returns

(Observable): A hot observable containing the results from the JSONP call.

Example

The following example uses a simple URL to retrieve a list of entries from Wikipedia.

```
var url = 'http://en.wikipedia.org/w/api.php?action=opensearch'
    + '&format=json'
    + '&search=reactive';

Rx.DOM.Request.jsonpRequest(url)
    .subscribe(
        function (data) {
            data[1].forEach(function (item) {
                console.log(item);
            });
        },
        function (error) {
            // Log the error
        }
    );
```

`Rx.DOM.Request.jsonpRequestCold(url | settings)`

#Ⓢ [Ⓣ][1]

Creates a cold observable JSONP Request with the specified settings or a string URL. **Note when using the method with a URL, it must contain JSONPRequest=?.**

Syntax

This method has two versions, one with a string URL, the other with a settings object.

```
// With a string URL
Rx.DOM.Request.jsonpRequestCold(url);

// With a settings object
Rx.DOM.Request.jsonpRequestCold(settings);
```

Arguments

1. `url` *(String)*: A string of the URL to make the JSONP call.
2. `settings` *(Object)*: An object with the following properties:

   ```
   - `url` *(String)*: URL of the request
     - `jsonp` *(String)*: The named callback parameter for the JSONP call
   ```

Returns

(Observable): A cold observable containing the results from the JSONP call.

Example

The following example uses a simple URL to retrieve a list of entries from Wikipedia.

```
var url = 'http://en.wikipedia.org/w/api.php?action=opensearch'
    + '&format=json'
    + '&search=reactive';

Rx.DOM.Request.jsonpRequestCold(url)
    .subscribe(
        function (data) {
            data[1].forEach(function (item) {
                console.log(item);
            });
        },
        function (error) {
            // Log the error
        }
    );
```

Web Sockets

- `Rx.DOM.fromWebSocket`

Rx.DOM.fromWebSocket(url, protocol, [observerOrOnNext])

#Ⓢ [⊤][1]

Creates a WebSocket Subject with a given URL, protocol and an optional observer for the open event.

Arguments

1. `url` *(String)*: The URL of the WebSocket.
2. `protocol` *(String)*: The protocol of the WebSocket.
3. `[observerOrOnNext]` *(Rx.Observer|Function)*: An optional Observer or onNext function to capture the open event.

Returns

(Subject): A Subject which wraps a WebSocket.

Example

```javascript
// Using a function for the open
var socket = Rx.DOM.fromWebSocket(
    'http://localhost:8080',
    'protocol',
    function (e) {
        console.log('Opening');
    })

socket.subscribe(function (next) {
    console.log('Received data: ' + next);
});

socket.onNext('data');

// Using an observer for the open
var observer = Rx.Observer.create(function (e) {
    console.log('Opening');
});

var socket = Rx.DOM.fromWebSocket(
    'http://localhost:8080', 'protocol', observer)

socket.subscribe(function (next) {
    console.log('Received data: ' + next);
});

socket.onNext('data');
```

Web Workers

- `Rx.DOM.fromWebWorker`

`Rx.DOM.fromWebWorker(url)`

#Ⓢ [Ⓣ][1]

Creates a Web Worker with a given URL as a Subject.

Syntax

```
Rx.DOM.fromWebWorker(url);
```

Arguments

1. `url` *(String)*: The URL of the Web Worker.

Returns

(Subject): A Subject which wraps a Web Worker.

Example

```javascript
var worker = Rx.DOM.fromWebWorker('worker.js');

worker.subscribe(function (e) {
    console.log(e.data);
});

worker.onNext('some data');
```

Mutation Observers

- `Rx.DOM.fromMutationObserver`

Rx.DOM.fromMutationObserver(target, options)

#Ⓢ [Ⓣ][1]

Creates an observable sequence from a `MutationObserver` . The `MutationObserver` provides developers a way to react to changes in a DOM. This requires `MutationObserver` to be supported in your browser/JavaScript runtime.

Arguments

1. `target` *(Node)*: The Node on which to obserave DOM mutations.
2. `options` *(MutationObserverInit)*: A `MutationObserverInit` object, specifies which DOM mutations should be reported.

Returns

(Observable): An observable sequence which contains mutations on the given DOM target.

Example

```javascript
var foo = document.getElementById('foo');

var obs = Rx.DOM.fromMutationObserver(foo, {
    attributes: true,
    childList: true,
    characterData: true,
    attributeFilter: ["id", "dir"]
});

foo.dir = 'rtl';

// Listen for mutations
obs.subscribe(function (mutations) {
    mutations.forEach(function(mutation) {
        console.log("Type of mutation: " + mutation.type);

        if ("attributes" === mutation.type) {
            console.log("Old attribute value: " + mutationRecord.oldValue);
        }
    });
});
```

Geolocation

- `Rx.DOM.Geolocation.getCurrentPosition`
- `Rx.DOM.Geolocation.watchPosition`

`Rx.DOM.Geolocation.getCurrentPosition([geolocationOptions])`

#Ⓢ [Ⓣ][1]

Obtains the geographic position, in terms of latitude and longitude coordinates, of the device.

Arguments

1. `[geolocationOptions]` *(Object)*: An object literal to specify one or more of the following attributes and desired values:
   - enableHighAccuracy: Specify true to obtain the most accurate position possible, or false to optimize in favor of performance and power consumption.
   - timeout: An Integer value that indicates the time, in milliseconds, allowed for obtaining the position. If timeout is Infinity, (the default value) the location request will not time out. If timeout is zero (0) or negative, the results depend on the behavior of the location provider.
   - maximumAge: An Integer value indicating the maximum age, in milliseconds, of cached position information. If maximumAge is non-zero, and a cached position that is no older than maximumAge is available, the cached position is used instead of obtaining an updated location. If maximumAge is zero (0), watchPosition always tries to obtain an updated position, even if a cached position is already available. If maximumAge is Infinity, any cached position is used, regardless of its age, and watchPosition only tries to obtain an updated position if no cached position data exists.

Returns

(Observable): An observable sequence with the current geographical location of the device running the client.

Example

```javascript
var source = Rx.DOM.Geolocation.getCurrentPosition();

var subscription = source.subscribe(
    function (pos) {
        console.log('Next:' + position.coords.latitude + ',' + position.coords.longitude)
    },
    function (err) {
        var message = '';
        switch (err.code) {
            case err.PERMISSION_DENIED:
                message = 'Permission denied';
                break;
            case err.POSITION_UNAVAILABLE:
                message = 'Position unavailable';
                break;
            case err.PERMISSION_DENIED_TIMEOUT:
                message = 'Position timeout';
                break;
        }
        console.log('Error: ' + message);
    },
    function () {
        console.log('Completed');
    });
```

```
 Rx.DOM.Geolocation.watchPosition([geolocationO
ptions])
```

#Ⓢ [Ⓣ][1]

Begins listening for updates to the current geographical location of the device running the client.

Arguments

1. `[geolocationOptions]` *(Object)*: An object literal to specify one or more of the following attributes and desired values:
   - enableHighAccuracy: Specify true to obtain the most accurate position possible, or false to optimize in favor of performance and power consumption.
   - timeout: An Integer value that indicates the time, in milliseconds, allowed for obtaining the position. If timeout is Infinity, (the default value) the location request will not time out. If timeout is zero (0) or negative, the results depend on the behavior of the location provider.
   - maximumAge: An Integer value indicating the maximum age, in milliseconds, of cached position information. If maximumAge is non-zero, and a cached position that is no older than maximumAge is available, the cached position is used instead of obtaining an updated location. If maximumAge is zero (0), watchPosition always tries to obtain an updated position, even if a cached position is already available. If maximumAge is Infinity, any cached position is used, regardless of its age, and watchPosition only tries to obtain an updated position if no cached position data exists.

Returns

(Observable): An observable sequence with the current geographical location of the device running the client.

Example

```
var source = Rx.DOM.Geolocation.watchPosition();

var subscription = source.subscribe(
    function (pos) {
        console.log('Next:' + position.coords.latitude + ',' + position.coords.longitude)
    },
    function (err) {
        var message = '';
        switch (err.code) {
            case err.PERMISSION_DENIED:
                message = 'Permission denied';
                break;
            case err.POSITION_UNAVAILABLE:
                message = 'Position unavailable';
                break;
            case err.PERMISSION_DENIED_TIMEOUT:
                message = 'Position timeout';
                break;
        }
        console.log('Error: ' + message);
    },
    function () {
        console.log('Completed');
    });
```

Schedulers

- `Rx.Schedulers.requestAnimationFrame`
- `Rx.Schedulers.mutationObserver`

- `Rx.Schedulers.requestAnimationFrame`
- `Rx.Schedulers.mutationObserver`

Rx.Scheduler.requestAnimationFrame

Gets a scheduler that schedules schedules work on the `window.requestAnimationFrame` for immediate actions.

Example

Rx.Scheduler.mutationObserver

Gets a scheduler that schedules schedules work on the `window.MutationObserver` for immediate actions.

Example

jQuery (*plugin* `rxjs-jquery`)

Example

`AngularJS` *(plugin `rx.angular.js`)*

Reactive Extensions (Rx) is a library for composing asynchronous and event-based programs using observable sequences and Array#extras style operators.

Data sequences can take many forms, such as a stream of data from a file or web service, web services requests, system notifications, or a series of events such as user input.

Reactive Extensions represents all these data sequences as observable sequences. An application can subscribe to these observable sequences to receive asynchronous notifications as new data arrive.

This library provides bridges to the popular Angular JS library.

Reactive Extensions Binding for the AngularJS API

This section contains the reference documentation for the Reactive Extensions for AngularJS library.

Factories:

- `rx`
- `observeOnScope`

Observable Methods:

- `safeApply`

`$rootScope` Methods:

- `$createObservableFunction`
- `$toObservable`
- `$eventToObservable`

Factories:

- `rx`
- `observeOnScope`

rx

Ⓢ

Creates a factory for using RxJS.

Returns

(Rx): The root of RxJS

Example

```javascript
angular.module('example', ['rx'])
    .controller('AppCtrl', function($scope, rx) {

        $scope.counter = 0;

        rx.Observable.interval(1000)
            .safeApply(
                $scope,
                function (x) {
                    $scope.counter = x;
                })
            .subscribe();

    });
```

Location

File:

- /src/factory.js

Dist:

- rx.angular.js

observeOnScope(scope, watchExpression, [objectEquality])

Ⓢ

Creates a factory which allows the user to observe a property on a given scope to check for old and new values.

Arguments

1. `scope` *(Scope)*: The scope to apply the watch function.
2. `watchExpression` : Expression that is evaluated on each `$digest` cycle. A change in the return value triggers a call to the listener.
   - `string` : Evaluated as expression
   - `function(scope)` : called with current scope as a parameter.
3. `[objectEquality]` : *(Function)*: Compare object for equality rather than for reference.

Returns

(Rx): The root of RxJS

Example

```
angular.module('observeOnScopeApp', ['rx'])
    .controller('AppCtrl', function($scope, observeOnScope) {

        observeOnScope($scope, 'name').subscribe(function(change) {
            $scope.observedChange = change;
            $scope.newValue = change.newValue;
            $scope.oldValue = change.oldValue;
        });
    });
```

Location

File:

- /src/observeronscope.js

Dist:

- rx.angular.js

Observable Methods:

- `safeApply`

rx

Ⓢ

Creates a factory for using RxJS.

Returns

(Rx): The root of RxJS

Example

```javascript
angular.module('example', ['rx'])
    .controller('AppCtrl', function($scope, rx) {

        $scope.counter = 0;

        rx.Observable.interval(1000)
            .safeApply(
                $scope,
                function (x) {
                    $scope.counter = x;
                })
            .subscribe();

    });
```

Location

File:

- /src/factory.js

Dist:

- rx.angular.js

`$rootScope` Methods:

- `$createObservableFunction`
- `$toObservable`
- `$eventToObservable`

- `$createObservableFunction`

$toObservable

\# Ⓢ

$eventToObservable

\# Ⓢ

`$createObservableFunction`

\# Ⓢ

Example

Facebook React *(plugin* *RxLifecycleMixin* *)*

Ractive.js *(plugin* `ractive-adaptors-rxjs` *)*

Example

Ractive.js *(plugin* `ractive-adaptors-rxjs` *)*

Node.js

Callback Handlers

- fromCallback
- fromNodeCallback

fromCallback

```
 Rx.Node.fromCallback(func, [scheduler],
[context])
```

\# Ⓢ

Deprecated in favor of `Rx.Observable.fromCallback` **in rx.async.js.**

Converts a callback function to an observable sequence.

Arguments

1. `func` *(Function)*: Callback function
2. `[scheduler = Rx.Scheduler.timeout]` *(Scheduler)*: Scheduler used to execute the callback.
3. `[context]` *(Any)*: The context to execute the callback.

Returns

(Function): Function, when called with arguments, creates an Observable sequence from the callback.

Example

```javascript
var fs = require('fs');
var Rx = require('Rx');

// Wrap exists
var exists = Rx.Node.fromCallback(fs.exists);

// Call exists
var source = exists('/etc/passwd');

var observer = Rx.Observer.create(
    function (x) {
        console.log('Next: ' + x);
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    }
);

var subscription = source.subscribe(observer);

// => Next: true
// => Completed
```

Location

- rx.node.js

fromNodeCallback

```
 Rx.Node.fromNodeCallback(func, [scheduler],
[context])
```

ⓢ

Deprecated in favor of `Rx.Observable.fromNodeCallback` **in rx.async.js.**

Converts a Node.js callback style function to an observable sequence. This must be in function (err, ...) format.

Arguments

1. `func` *(Function)*: Callback function which must be in function (err, ...) format.
2. `[scheduler = Rx.Scheduler.timeout]` *(Scheduler)*: Scheduler used to execute the callback.
3. `[context]` *(Any)*: The context to execute the callback.

Returns

(Function): An function which when applied, returns an observable sequence with the callback arguments as an array.

Example

```javascript
var fs = require('fs');
var Rx = require('Rx');

var source = Rx.Node.fromNodeCallback(fs.stat)('file.txt');

var observer = Rx.Observer.create(
    function (x) {
        var stat = x[0];
        console.log('Next: ' + stat.isFile());
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    }
);

var subscription = source.subscribe(observer);

// => Next: true
// => Completed
```

Location

- rx.node.js

Event Handlers

- fromEvent
- toEventEmitter

fromEvent

```
Rx.Node.fromEvent(eventEmitter, eventName)
```

Ⓢ

Handles an event from the given EventEmitter as an observable sequence.

Arguments

1. `eventEmitter` *(EventEmitter)*: The EventEmitter to subscribe to the given event.
2. `eventName` *(String)*: The event name to subscribe.

Returns

(Observable): An observable sequence generated from the named event from the given EventEmitter.

Example

```javascript
var EventEmitter = require('events').EventEmitter;
var Rx = require('Rx');

var emitter = new EventEmitter();

var source = Rx.Node.fromEvent(emitter, 'data');

var observer = Rx.Observer.create(
    function (x) {
        console.log('Next: ' + x[0]);
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    }
);

var subscription = source.subscribe(observer);

emitter.emit('data', 'foo');

// => Next: foo
```

Location

- rx.node.js

toEventEmitter

`Rx.Node.toEventEmitter(observable, eventName)`

Ⓢ

Converts the given observable sequence to an event emitter with the given event name. The errors are handled on the 'error' event and completion on the 'end' event.

Arguments

1. `observable` *(Obsesrvable)*: The observable sequence to convert to an EventEmitter.
2. `eventName` *(String)*: The event name to subscribe.

Returns

(EventEmitter): An EventEmitter which emits the given eventName for each onNext call in addition to 'error' and 'end' events.

Example

```javascript
var Rx = require('Rx');

var source = Rx.Observable.return(42);

var emitter = Rx.Node.toEventEmitter(source, 'data');

emitter.on('data', function (data) {
    console.log('Data: ' + data);
});

emitter.on('end', function () {
    console.log('End');
});

// Ensure to call publish to fire events from the observable
emitter.publish();

// => Data: 42
// => End
```

Location

- rx.node.js

Stream Handlers

- fromStream
- writeToStream

fromStream

```
Rx.Node.fromStream(stream)
```

\# Ⓢ

Converts a flowing stream to an Observable sequence.

Arguments

1. `stream` *(Stream)*: A stream to convert to a observable sequence.

Returns

(Observable): An observable sequence which fires on each 'data' event as well as handling 'error' and 'end' events.

Example

```javascript
var Rx = require('rx');

var subscription = Rx.Node.fromStream(process.stdin)
    .subscribe(function (x) { console.log(x); });

// => r<Buffer 72>
// => x<Buffer 78>
```

Location

- rx.node.js

fromReadableStream

`Rx.Node.fromReadableStream(stream)`

Ⓢ

Converts a flowing readable stream to an Observable sequence.

Arguments

1. `stream` *(Stream)*: A stream to convert to a observable sequence.

Returns

(Observable): An observable sequence which fires on each 'data' event as well as handling 'error' and 'end' events.

Example

```js
var Rx = require('rx');

var subscription = Rx.Node.fromReadableStream(process.stdin)
    .subscribe(function (x) { console.log(x); });

// => r<Buffer 72>
// => x<Buffer 78>
```

Location

- rx.node.js

fromWritableStream

```
Rx.Node.fromWritableStream(stream)
```

ⓢ

Converts a flowing writeable stream to an Observable sequence.

Arguments

1. `stream` *(Stream)*: A stream to convert to a observable sequence.

Returns

(Observable): An observable sequence which fires on each 'data' event as well as handling 'error' and 'finish' events.

Example

```javascript
var Rx = require('rx');

var subscription = Rx.Node.fromWritableStream(process.stdout)
    .subscribe(function (x) { console.log(x); });

// => r<Buffer 72>
// => x<Buffer 78>
```

Location

- rx.node.js

fromTransformStream

Rx.Node.fromTransformStream(stream)

Ⓢ

Converts a flowing transform stream to an Observable sequence.

Arguments

1. `stream` *(Stream)*: A stream to convert to a observable sequence.

Returns

(Observable): An observable sequence which fires on each 'data' event as well as handling 'error' and 'finish' events.

Example

```javascript
var Rx = require('rx');

var subscription = Rx.Node.fromTransformStream(getTransformStreamSomehow());
```

Location

- rx.node.js

writeToStream

```
Rx.Node.writeToStream(observable, stream,
[encoding])
```

Ⓢ

Writes an observable sequence to a stream.

Arguments

1. `observable` *(Obsesrvable)*: Observable sequence to write to a stream.
2. `stream` *(Stream)*: The stream to write to.
3. `[encoding]` *(String)*: The encoding of the item to write.

Returns

(Disposable): The subscription handle.

Example

```javascript
var Rx = require('Rx');

var source = Rx.Observable.range(0, 5);

var subscription = Rx.Node.writeToStream(source, process.stdout, 'utf8');

// => 01234
```

Location

- rx.node.js

Resources

- Articles
- Similar libraries
- Presentations
- Video

Articles

Rx

- Reactive Programming and MVC
- Visualizing Reactive Streams: Hot and Cold Observables
- Using RxJS for data flow instead of Flux with React
- 2 minute introduction to Rx
- Functional Reactive React.js
- Разработка javascript приложений на базе Rx.js и React.js (RxReact) (ru)
- FRP using RxJS and Firebase
- Reactive Components with RxJs filter-scan-map
- Rx（Reactive Extensions）とVirtual DOMで作るリアクティブなUI、およびMVARアーキテクチャについて
- Reactive (in RxJS) and Non-Reactive (in Scala) Snakes And Ladders
- How to make a godforsaken dynamic width scroll table with React with RxJS
- Using RxJs instead of flux with react
- Writing an App
- Android архитектура клиент-серверного приложения (ru)
- Building a Model Layer with RxJS
- React + RxJS: Live Search
- Advanced RxJava
- Angular — Introduction to Reactive Extensions (RxJS)
- Introducing Rx (Linq to Events)
- Как сделать самоподгружаемый RecyclerView с помощью RxJava (ru)
- Git Concurrency in GitHub Desktop
- Loading data from multiple sources with RxJava
- Hot and Cold Observables
- To Use Subject Or Not To Use Subject?
- Deferring Observable code until subscription in RxJava
- Retrofit 2.0: The biggest update yet on the best HTTP Client Library for Android
- Linq Over Events — Playing with the Rx Framework
- Drag and Drop using Reactive Extensions for Javascript
- Reactive Extensions for Javascript – Observable vs ConnectableObservable
- Reactive Extensions for Javascript – Causing side effects with Do
- Reactive Extensions for Javascript – Wikipedia lookup demo
- Reactive Extensions for Javascript – Multiple observers for multiple observables

rx-book

- Rx framework — IObservable.Retry
- Rx framework — IObservable.Repeat
- Rx framework — IObservable.Catch and IObservable.OnErrorResumeNext
- Rx framework — IObservable.Concat
- Rx framework — IObservable.CombineLatest
- Rx framework — IObservable.Merge
- RxJava Observable tranformation: concatMap() vs flatMap()
- RxJava: defer execution of a function with new operator fromCallable()
- Введение в RxJava, часть первая – Вступление: Почему Rx? (ru)
- Введение в RxJava, часть первая – Вступление: Ключевые типы (ru)
- Как сделать самоподгружаемый RecyclerView с помощью RxJava (ru)
- Reactive Extensions. Api client with Cache-Aside & Refresh-Ahead strategy. Part 1.
- Reactive Extensions. Api client with Cache-Aside & Refresh-Ahead strategy. Part 2.
- Rxで指定した条件を満たすまでbufferする
- Reactive Programming with RxJava in Android
- Adding Retrofit To Kotlin And RxJava Android Application
- Build Android UI In Code Painlessly With Anko And Kotlin
- Porting RxJava Android Application To Kotlin
- Android From Scratch - Bare Minimum Android Project With Kotlin
- Reactive Programming with RxJava in Android
- What's the RxJava equivalent of the Observable.expand() in RxJs?
- How can I do a recursive read-dir with RxJs (or any FRP lib)?
- Functional Reactive Programming on Android With RxJava
- Introducing Rx (Linq to Events)
- Rx: Materialize() – Stream Notifcations
- Angular services with RxJS; decrapify your services
- Sharing in RX: Publish, Replay, and Multicast
- Many articles about Rx
- Blog about Rx
- Dave Sexton's Blog
- Modelling Market Data Services in Rx with MergeSubject
- neue.cc blog
- Reactive (Dataflow) Programming in Java and Clojure with Quasar and Pulsar
- Comparing Core Async and Rx by Example
- Mike Taulty's Blog
- Event stream manipulation using Rx – Part 1
- Event stream manipulation using Rx – Part 2
- Functional Reactive Programming on Android With RxJava
- Isolated Storage as Cache using Reactive Extensions
- Cache as Fallback using Reactive Extensions

- [Some material about Rx (http://www.minddriven.de)](http://www.minddriven.de)
- [Financial Charts & Reactive Extensions](#)
- [Angular 2 Observable Data Services](#)
- [Game of Life in RxJS](#)
- [Rx glitches aren't actually a problem](#)
- [The Essence of LINQ – MinLINQ](#)
- [Duality of IEnumerable/IObservable](#)
- [Introduction to the Reactive frmawork - Part II](#)
- [Official blog](#)
- [Your Mouse is a Database](#)
- [Going Reactive: An Architectural Journey](#)
- [Android MVP, Retrofit & Rx](#)
- [Managing state in Angular 2 using RxJs](#)
- [Experimentation with RxJava](#)
- [Jugando con decoradores de ES6 y RxJS: Streamify](#)
- [Пагинация списков в Android с RxJava. Часть II](#)
- [Fluorine: Flux beyond Redux with RxJS](#)
- [RxJS is great. So why have I moved on?](#)
- [Functional Reactive Programming with RxSwift](#)
- [Experimentation with Rx](#)

Falcor

- [Falcor for Android](#)

F

- [Nice F# Syntax for Rx (Reactive Extensions)](#)
- [Reactive programming (I.) - First class events in F#](#)
- [Reactive programming (II.) - Introducing Reactive LINQ](#)
- [Reactive Programming (III.) - Useful Reactive LINQ Operators](#)
- [Reactive Programming (IV.) - Developing a game in Reactive LINQ](#)
- [Tutorial: Functional Reactive Programming in F# and WPF (Part 1)](#)
- [Reactive Programming](#)
- [Tutorial: Using F# with Unity3D](#)

Elm

- A simple introduction to using Elm ports
- Making a scroll table with Elm
- Elm: Signals, Mailboxes & Ports (Video Tutorial with Mike Clark)
- Elm – fun with L-System (Part 1)
- Many articles about Elm
- Elm Architecture in PureScript (Elm vs Purescript)
- Learning FP the hard way: Experiences on the Elm language
- Elm + Electron

Cycle

- A Dead-Simple Todo List with RxJS
- How to use Cycle.js to create a scroll-table
- Mounting D3 Datamaps to containers in Cycle.js using drivers
- How to make a Cycle.js + Elm hybrid Etch-a-Sketch app
- Plug and Play All Your Observable Streams With Cycle.js
- Detect touch gestures with Cycle.js and Hammer.js
- Why I chose to use Cycle.js

Bacon

- Functional Reactive Programming in JavaScript
- Functional Reactive Programming in JavaScript
- Javascript generators and functional reactive programming
- Writing functional applications with Functional Reactive Programming and Bacon.js
- Functional Reactive Programming with Bacon.js
- Implementing Snake in Bacon.js
- Reactive UIs with React & Bacon
- Create an Infinite Scroll List with Bacon.js
- Bacon.js + Node.js + MongoDB: Functional Reactive Programming on the Server
- Flux inspired reactive data flow using React and Bacon.js
- Good bye Flux, welcome Bacon/Rx?
- Isomorphic apps = normal React+FRP apps
- Building a Pacman Game With Bacon.js
- Reactive ReactJS: improving data flow using reactive streams
- Event Driven and Functional Reactive Programming, Step by Step
- Functional Reactive Programming with Bacon.js and d3
- Bacon.js blog
- Functional Reactive Programming with Bacon.js

- Bacon.js for Dummies
- FRP (functional reactive programming) на Bacon.js (ru)
- Functional Reactive Applications

Kefir

- Learning to think in frp: my experience coding a game with Kefir.js
- Courier: a reactive React client architecture
- Reactive programming with Kefir.js and React

Mixed

- Functional reactive programming in JavaScript
- Functional reactive programming concepts in javascript on top of Backbone
- Функциональное реактивное программирование (ru)
- Reactive Programming
- Articles about FRP
- Abstracting away the grunt work with Plumber
- Understanding the functional revolution in front-end applications
- Promises + FP = Beautiful Streams
- Articles about FRP
- Flatmap that shit!
- CSP vs. FRP
- Containers Are Dead. Long Live Observable Combinators
- Combining Flux and FRP: Hello ffux!
- Good bye Flux, welcome Bacon/Rx?
- Isomorphic apps = normal React+FRP apps
- A simpler web architecture using Flux, CSP, and FRP concepts
- Functional Reactive Programming in reactive-banana

Unity

- Многопоточность в Unity средствами реактивных расширений (ru)

Flyd

- Fluid configuration of d3.js bandlines with FRP

Similar libraries which were inspired by RxJS

Libraries

- **Bacon** - A small functional reactive programming lib for JavaScript
- **Kefir** - Is a Reactive Programming library for JavaScript inspired by Bacon.js and RxJS, with focus on high performance and low memory usage
- **Streamly** - A very small functional reactive programming lib for JavaScript.
- **Flyd** - The minimalistic but powerful, modular, functional reactive programming library in JavaScript
- **Asyncplify** - FRP (functional reactive programming) library for Javascript
- **RxJS(ES6)** - A reactive programming library for JavaScript
- **RxJS-2016**
- **RxJS(Modular)** - Modular version of RxJS
- **Most** - Monadic reactive streams
- **Reactor.js** - Javascript reactive programming library
- **Reactive.js** - A library for programming reactively in Javascript.
- **Sig** - High-level reactive-style programming in javascript
- **F** - Dead simple library for functional and reactive programming in JavaScript
- **Model** - A functional reactive model library for interactive data visualization.
- **Reactive Model** - A library for reactive models.
- **Proactjs** - Reactive JavaScript Object Properties
- **Axos** - High-performance multi-paradigm async programming
- **es-observable** - Observables for ECMAScript
- **MOBservable** - Library to create reactive data structures and functions
- **DYReactive** - Reactive programming library
- **adtstream** - Streams built using Algebraic Data Types (ADT) and Pure FP (Haskell like) code
- **Radioactive** - The Missing Native Functional Reactive API for Javascript
- **Frampton** - A functional reactive library
- **havelock** - Functional Reactive State for JavaScript and TypeScript
- **timeengine** - Time Engine is the True Functional Reactive Programming Library from the Point of View of Physics, our World and Time
- **frp.js** - Functional Reactive Programming primitives for JavaScript
- **fairmont-reactive** - Functional reactive programming in JavaScript and CoffeeScript
- **paunkie** - No non-sense Functional Reactive Programming Library

- **mom** - Minimalistic reactive javascript framework (mori + snabbdom)
- **Apostate** - Reactive state management
- **cellx** - Ultra-fast implementation of reactivity for javascript
- **dgelong** - A JavaScript Implementation of Useful First-class Citizens
- **dv.js** - Dynamic Value. Basic FRP with Javascript
- **Highland** - The high-level streams library for Node.js and the browser
- **Asynquence** - Promise-style async sequence flow-control
- **furp** - FRP with LiveScript
- **derivablejs** - Functional Reactive State for JavaScript and TypeScript
- **happen.js** - Functional Reactive & Dataflow programming in Javascript done right, done fast
- **quiver-signal** - FRP signal type for Quiver front end
- **frpjs** - Functional Reactive Programming Library for JavaScript
- **nexus** - An FRP cell application framework

Frameworks

- **Yolk** - A library for building asynchronous user interfaces
- **Reflex** - Functional reactive UI library
- **Zorium** - The CoffeeScript Web Framework
- **Cycle** - A fully reactive JavaScript framework for Human-Computer Interaction
- **WebRx** - WebRx is a browser-based MVVM-Framework that combines functional-reactive programming with declarative Data-Binding, Templating and Client-Side Routing.
- **Falcor** - A JavaScript library for efficient data fetching
- **jsblocks** - Better MV-ish Framework (doesn't use Rx but use frp)
- **reaxtor** - Falcor + Rx + virtual-dom

Resources

- https://github.com/christianramsey/rx-react-flux
- https://github.com/LeeCampbell/RxCookbook
- https://github.com/zsoltk/RxAndroidLibs
- http://elm-lang.org/guide/reactivity
- https://github.com/vagmi/jqconf-elm-talk
- https://blogs.endjin.com/2014/04/event-stream-manipulation-using-rx-part-1
- http://blog.webrunners.de/series/reactive-extensions
- http://www.introtorx.com
- https://github.com/riiid/rxjs-dash

- http://leecampbell.blogspot.com/2010/05/intro-to-rx.html
- https://github.com/pinakinmistry/workshops/tree/gh-pages/frjs2
- https://github.com/annamalai-saro/Rx.js-REPL
- https://github.com/RWTreichel/frp-demo
- https://github.com/d6u/reactdc-rxjs-talk

Most

React

- https://github.com/izaakschroeder/afflux
- https://github.com/roman01la/most-react

Other

- https://github.com/s2js/fabulous
- https://github.com/TylorS/most-subject
- https://github.com/axefrog/most-replay

Sodium

- https://github.com/SodiumFRP/sodium
- https://github.com/JamesTimms/FRPGameEngine
- https://github.com/SodiumFRP/sodium-kotlin

Jnoid

- https://github.com/it-vecherka/jnoid

Lazychain

- https://github.com/falsandtru/lazychain

Java

- https://github.com/skimarxall/RxFlux
- https://gist.github.com/cesarferreira/510aa2456dc0879f559f
- https://github.com/eleventigers/awesome-rxjava

- https://github.com/ReactiveX/RxJava
- https://github.com/davidmoten/rxjava-extras
- https://github.com/davidmoten/rxjava-jdbc
- https://github.com/darylteo/rxjava-promises
- https://github.com/ReactiveX/RxNetty
- https://github.com/fabioCollini/IntroToRetrofitRxJava
- https://github.com/Froussios/Intro-To-RxJava (learning material)
- https://github.com/jmhofer/rxjava-libgdx
- https://github.com/meez/mod-rxjava
- https://github.com/samuelgruetter/rx-playground
- https://github.com/vert-x/mod-rxvertx
- https://github.com/mox601/rx-java-spike
- https://github.com/vert-x3/vertx-rx
- https://github.com/pk11/rxnetty-router
- https://github.com/threerings/react
- https://github.com/ReactiveX/RxSwing
- https://github.com/lzyzsd/Awesome-RxJava
- https://github.com/Gridstone/RxStore
- https://github.com/f2prateek/rx-receivers
- https://github.com/f2prateek/rx-preferences
- https://github.com/ReactiveX/RxJavaString
- https://github.com/ReactiveX/RxJavaAsyncUtil
- https://github.com/mutexkid/rxjava-koans
- https://github.com/helmbold/rxfilewatcher
- https://github.com/Marchuck/RxRealm
- https://github.com/Reyurnible/RxAnimation

Android

- https://github.com/ReactiveX/RxAndroid
- https://github.com/tehmou/rx-android-architecture
- https://github.com/mcharmas/Android-ReactiveLocation
- https://github.com/kaushikgopal/Android-RxJava (should be read first)
- https://github.com/JakeWharton/RxBinding
- https://github.com/vyshane/rex-weather
- https://github.com/Naturally-Being/G-Rex
- https://github.com/novoda/rxpresso
- https://github.com/erickok/RxCupboard
- https://github.com/ogaclejapan/RxBinding

- https://github.com/kboyarshinov/realm-rxjava-example
- https://github.com/exallium/RxRecyclerView
- https://github.com/richardradics/RxAndroidBootstrap
- https://github.com/square/retrofit
- https://github.com/futurice/android-best-practices
- https://github.com/andrewhr/rxjava-android-example
- https://github.com/aurae/RxBonjour
- https://github.com/square/sqlbrite
- https://github.com/pushtorefresh/storio
- https://github.com/wverdese/rxgithub
- https://github.com/kgmyshin/Observer-Pattern-to-Rx
- https://github.com/mattlogan/RxJavaImageSearch
- https://github.com/danosipov/rxdownloader
- https://github.com/jejking/RxOsmParser
- https://github.com/tehmou/RxMapsAndroid
- https://github.com/colintheshots/RxRetrofitAutoCompleteExample
- https://github.com/nightbear1009/SmallDaggerRetrofitRxjavaSample
- https://github.com/drteeth/android-rx-mvvm
- https://github.com/abufathan/RxRobo
- https://github.com/asufana/rxjava-github-api
- https://github.com/konmik/RxQuery
- https://github.com/RxTUDelft/RxJavaGames
- https://github.com/kboyarshinov/realm-rxjava-example
- https://github.com/android10/Android-ReactiveProgramming
- https://github.com/mdrabic/android-yolo-rxsensors
- https://github.com/fedestyla/RetrofitWithRx
- https://github.com/michaelworth/RxTwitter
- https://github.com/miomao656/RxMeetUp
- https://github.com/evant/rxloader
- https://github.com/t28hub/RxWeather
- https://github.com/GDG-Korea/HelloRx
- https://github.com/talenguyen/rxflux
- https://github.com/kaushikgopal/RxJava-Android-Samples
- https://github.com/ewintory/udacity-popular-movies
- https://github.com/trello/RxLifecycle
- https://github.com/jacek-marchwicki/rx-java-extensions
- https://github.com/ivacf/archi
- https://github.com/ekamp/RxCurrently
- https://github.com/konmik/nucleus
- https://github.com/Gitteroid/GitterRxJavaSDK

- https://github.com/reark/reark
- https://github.com/3pillarlabs/rx_android_workshop
- https://github.com/aaronzirbes/rx-java-playground
- https://github.com/mulab/rx-realm
- https://github.com/RefineriaWeb/base_app_android
- https://github.com/yongjhih/RxBolts
- https://github.com/LiveTyping/u2020-mvp

Tutorials

- https://github.com/jhusain/learnrxjava

Scala

- https://github.com/LoyolaChicagoCode/clickcounter-android-rxscala
- https://github.com/Applied-Duality/RxGame
- https://github.com/lukiano/rxspray
- https://github.com/jmhofer/activator-libgdx-rxscala
- https://github.com/jmhofer/rxjava-akka
- https://github.com/plasmaconduit/conveyance
- https://github.com/MetaStack-pl/MetaRx
- https://github.com/monifu/monifu
- https://github.com/raimohanska/francis
- https://github.com/cjdev/kafka-rx
- https://github.com/evgenynacu/android-frp
- https://github.com/widok/widok

Groovy

- https://github.com/sys1yagi/rxandroid-usecase

Kotlin

- https://github.com/ReactiveX/RxKotlin
- https://github.com/cy6erGn0m/kotlin-rxokhttp-websocket
- https://github.com/griffio/kotlin-rx-android

- https://github.com/stepango/kotlin_rxbinding_android_demo
- https://github.com/kittinunf/RxMovieKotlin
- https://github.com/benoitletondor/Rx4Square
- https://github.com/importre/popular
- https://github.com/satorufujiwara/kotlin-android-example
- https://github.com/ian-ellis/kotlin-mvvm

.NET

- https://github.com/TORISOUP/PhotonRx
- https://github.com/ufcpp/AwaitableUniRx
- https://github.com/mavriel/UniRxForNGUI
- https://github.com/TORISOUP/UniRxExamples
- https://github.com/runceel/UniRxProperty
- https://github.com/keijiro/Reaktion
- http://forum.unity3d.com/threads/unirx-reactive-extensions-for-unity.248535
- https://github.com/Reactive-Extensions/Rx.NET
- https://github.com/neuecc/UniRx (Unity)
- https://github.com/zillakot/reactive-core (Unity)
- https://github.com/TORISOUP/UniRxWorkBook (Unity)
- https://github.com/waynebaby/MVVM-Sidekick
- https://github.com/niik/RxSpy
- https://github.com/atsushieno/mono-reactive
- https://github.com/inter8ection/Obvs
- https://github.com/shiftkey/Reactive.EventAggregator
- https://github.com/inconspicuous-creations/Inconspicuous.Framework
- https://github.com/RxDave/Rxx
- https://github.com/Reactive-Extensions/RxToBand
- https://github.com/MarcusKohnert/Kinect.ReactiveV2
- https://github.com/ugaya40/RxFlow
- https://github.com/reactiveui/ReactiveUI
- https://github.com/AdaptiveConsulting/ReactiveTrader
- https://github.com/runceel/ReactiveProperty
- https://github.com/atsushieno/mono-reactive
- https://github.com/shiftkey/ReactiveGit
- https://github.com/sh1989/ReactiveFlickr
- https://github.com/terrehbyte/UnityRxResearch
- https://github.com/TORISOUP/UniRxExamples
- https://github.com/Froussios/New-Intro-To-Rx

- https://github.com/rbirkby/RxMovies
- http://reactiveoauth.codeplex.com
- https://github.com/runceel/UniRxProperty
- https://github.com/LeeCampbell/RxCookbook
- https://github.com/bordoley/RxApp
- https://github.com/rob-blackbourn/JetBlack.Network
- https://github.com/TheAngryByrd/RxSandbox
- https://github.com/JuliusSweetland/OptiKey
- https://github.com/envicase/ReactiveMvvm
- https://github.com/JohanLarsson/Gu.Reactive
- https://github.com/tamirdresher/RxInAction
- https://github.com/ApocalypticOctopus/Rx.Community.Extensions
- https://github.com/akarnokd/RxReactiveStreamsCSharp
- https://github.com/kswoll/sexy-react
- https://github.com/ExRam/ExRam.ReactiveCollections
- https://github.com/Anna-Rx/Anna
- https://github.com/akarnokd/RxAdvancedFlow

C++

- https://github.com/Reactive-Extensions/RxCpp

Python

- https://github.com/ReactiveX/RxPY

Ruby

- https://github.com/ReactiveX/RxRuby

Swift

- https://github.com/ReactiveX/RxSwift
- https://github.com/ReSwift/ReduxRx

FSharp

- https://github.com/fsprojects/FSharp.Control.Reactive
- http://rxbuilder.codeplex.com
- http://fsreactive.codeplex.com

PHP

- https://github.com/asm89/Rx.PHP

Dart

- https://github.com/theburningmonk/stream_ext
- https://github.com/frankpepermans/rxdart

Rust

- https://github.com/owickstrom/rust-rx
- https://github.com/ReactiveX/RxRust

Haskell

- https://github.com/raimohanska/rx-haskell

Lua

- https://github.com/bjornbytes/RxLua

Elixir

- https://github.com/alfert/reaxive

Nim

- https://github.com/andreaferretti/rxnim

MObservable

- https://github.com/dschalk/mobservable-monads
- https://github.com/survivejs/mobservable-demo

Other

- https://github.com/kalcifer/absolem
- https://github.com/f5io/ohflux

Haskell

- https://github.com/ivanperez-keera/Yampa
- https://github.com/raimohanska/reactive-bacon

F

- https://github.com/intellifactory/reactive

Bacon - A small functional reactive programming lib for JavaScript

Angular

- https://github.com/lauripiispanen/angular-bacon
- https://github.com/rap1ds/angular-bacon
- https://github.com/xgrommx/angular-bacon-bridge
- https://github.com/javamonn/todo-angular-flux
- https://github.com/bangjs/bangjs
- https://github.com/javamonn/todo-angular-flux
- https://github.com/simon-engledew/bacon-angular-draggables

React

- https://github.com/jamesmacaulay/react-bacon
- https://github.com/niklasvh/react-bacon-flux-poc
- https://github.com/mattsenior/react-bacon-mori
- https://github.com/Willmo36/FluxBacon
- https://github.com/theadam/conflux.js
- https://gist.github.com/rattrayalex/dee40d86813bcaa9de80
- https://github.com/ethagnawl/Hello-FRP
- https://github.com/idmitriev/hypr
- https://github.com/milankinen/react-bacon-isomorphic
- https://github.com/skellyb/fluxstream
- https://github.com/milankinen/react-bacon-todomvc
- https://github.com/sdkennedy/baconflux
- https://github.com/rikukissa/bacon-flux-react
- https://github.com/checkraiser/frp-test
- https://github.com/elsehow/frplux
- https://github.com/miciek/web-snake-react-bacon
- https://github.com/milankinen/megablob
- https://github.com/VesaKarvonen/atomi-poc
- https://github.com/aparticka/redux-bacon
- https://github.com/aparticka/react-bacon-component

- https://github.com/spkerkela/mmh-js
- https://github.com/polytypic/bacon.react

Other

- https://github.com/erlikh/baconjs-experiments
- https://github.com/josepot/bacon-snake
- https://github.com/andrewhao/rxlocation
- https://github.com/moredip/reactive-colors
- https://github.com/ahomu/Talkie
- https://github.com/plemarquand/bacon-unsub
- https://github.com/eamodeorubio/reactive-baconjs
- https://github.com/puppybits/Bacon.CursorProperty.js
- https://github.com/suominentoni/bacon-ui-examples
- https://github.com/rikukissa/flappy-bird
- https://github.com/rbelouin/bacon-routes
- https://github.com/simon-engledew/bacon-angular-draggables
- https://github.com/CRogers/baconsnake
- https://github.com/josepot/bacon-snake
- https://github.com/philipnilsson/Snake-Bacon
- https://github.com/DanGorst/frp-with-bacon
- https://github.com/StreakYC/bacon-cast
- https://github.com/SamiJ/match3bacons
- https://github.com/Nyamazing/bacon-trans
- https://github.com/ericponto/bacon-for-breakfast-midwestjs
- https://github.com/erlikh/baconjs-experiments
- https://github.com/laurilehmijoki/learn-reactive-programming-with-bacon.js

Tutorials

- https://github.com/mikaelbr/bacon-love

Build tools

- https://github.com/sighjs/sigh

Cycle

- https://github.com/Widdershin/cycle-live-todo
- https://github.com/ohanhi/hyperscript-helpers
- https://github.com/benji6/cycle-audio-graph
- https://github.com/Mercateo/component-check
- https://github.com/Cmdv/cycle-universal-boilerplate
- https://github.com/bouzuya/cycle-todo
- https://github.com/TylorS/cycle-starter
- https://github.com/CyclicMaterials
- https://github.com/cyclejs/cycle-react-native
- https://gist.github.com/dmitriid/53789c56160d4d754ab5
- https://github.com/ivan-kleshnin/cycle-ultimate
- https://github.com/benji6/tic-tac-toe-cycle
- https://github.com/eschwartz/cycle-pong
- https://github.com/xgrommx/cycle-drag-and-drop
- https://github.com/pH200/cycle-react
- https://github.com/cyclejs/cycle-examples
- https://github.com/erykpiast/cyclejs-stream
- https://github.com/schrepfler/rx-helloworld-cycle
- https://github.com/phadej/graafi
- https://github.com/erykpiast/autocompleted-select
- https://github.com/erykpiast/cyclejs-examples
- https://github.com/erykpiast/cyclejs-mock
- https://github.com/erykpiast/cyclejs-group
- https://github.com/justinwoo/cycle-scroll-table
- https://github.com/rikukissa/ideas
- https://github.com/cyclejs/cycle-http-driver
- https://github.com/peterjoel/frpreact
- https://github.com/ivan-kleshnin/cyclejs-examples
- https://github.com/ivan-kleshnin/tetris-cyclejs
- https://github.com/ATempsch/cyclejs-boilerplate
- https://github.com/leoasis/cycle-poc
- https://github.com/bcoop713/cycrylic
- https://github.com/MadcapJake/earl-hyperscript
- https://github.com/cgeorg/todomvp
- https://github.com/Oliv75/stack-frp
- https://github.com/killercup/cycle-webpack-starter

- https://github.com/trlim/cycle-sockjs
- https://github.com/trlim/cycle-engine.io
- https://github.com/cgeorg/cycle-socket.io
- https://github.com/cyclejs/cycle-dom
- https://github.com/cyclejs/cycle-core
- https://github.com/cyclejs/todomvc-cycle
- https://github.com/cyclejs/cycle-jsonp-driver
- https://gist.github.com/staltz/b9850c4e620946edf14e
- https://github.com/cyclejs/rx-injectable-observable
- https://github.com/m0ru/cyclejs-todomvc
- https://github.com/tylors/cycle-director
- https://gist.github.com/staltz/e6fc5b2b19207bb4f7da
- https://github.com/axefrog/cycle-router5
- https://github.com/Mercateo/cycle-counter
- https://github.com/bkase/drag-drop-in-cyclejs
- https://github.com/vic/awesome-cyclejs
- https://github.com/Widdershin/cycle-time-travel
- https://github.com/CyclicMaterials/molecule-input
- https://github.com/andreloureiro/cyclejs-starter
- http://staltz.com/random-namespacing-in-cyclejs.html
- https://github.com/augustblack/cyclejs-autocomplete-multi
- https://github.com/survivejs/cycle-starter
- https://github.com/eschwartz/rxtodo
- https://github.com/bobvanderlinden/cycle-remote
- https://github.com/ckknight/fallout-character-sheet
- https://github.com/WorldMaker/cycle-history
- https://github.com/TylorS/cycle-history
- https://github.com/futurice/power-ui
- https://github.com/Cmdv/cycle-experiment
- https://github.com/takashi/cycle-markdown-driver
- https://github.com/r7kamura/cycle-fetcher-driver
- https://github.com/r7kamura/cycle-history-driver
- https://github.com/MarcCloud/magic-cart
- https://github.com/foxdonut/cycle-todolist
- https://github.com/staltz/flux-challenge/tree/master/submissions/staltz
- https://github.com/TylorS/cycle-snabbdom
- https://github.com/collardeau/kairos
- https://github.com/secobarbital/iouo.me
- https://github.com/secobarbital/cycle-vtree-switcher
- https://github.com/secobarbital/cycle-hashchange-driver

- https://github.com/secobarbital/cycle-pushstate-driver
- https://github.com/justinwoo/housing-costs-maps
- https://gist.github.com/dmitriid/baa371845e617a2dc18e
- https://github.com/justinwoo/cycle-elm-etch-sketch
- https://github.com/CyclicMaterials/cycle-hammer-driver
- https://github.com/Widdershin/cycle-hot-reloading-example
- https://gist.github.com/staltz/f460cc137fdef0181df3
- https://github.com/Cmdv/cycle-webpack-boilerplate
- https://github.com/kibin/cycle-example-who-to-follow
- https://github.com/cgeorg/cycle-socket.io
- https://github.com/spudly/cyclejs-todo-app
- https://github.com/motorcyclejs/core
- https://github.com/futurice/op-hackathon-templates
- https://github.com/class-ideas/cyclejs-hangman
- https://github.com/TylorS/motorcycle-native-poc

Elm - the best of functional programming in your browser

- http://elm-lang.org
- https://github.com/isRuslan/awesome-elm
- https://github.com/evancz/elm-html/tree/master/examples
- https://github.com/evancz/elm-todomvc
- https://github.com/grzegorzbalcerek/elm-by-example
- http://helm-engine.org
- https://github.com/dimchansky/elmnim
- https://github.com/evancz/elm-examples
- https://github.com/swistak35/uwr_elmgame
- https://github.com/rtfeldman/mloc-2015-elm-workshop
- https://github.com/TheSeamau5/elm-checkerboardgrid-tutorial
- https://github.com/evancz/TodoFRP
- https://github.com/mgold/elm-animation
- https://github.com/Morendil/FunFun
- https://github.com/mikegehard/buzzword-bingo-elm
- https://github.com/evancz/elm-architecture-tutorial
- https://github.com/evancz/elm-effects
- https://github.com/TheSeamau5/elm-undo-redo
- https://github.com/giisyu/elm-signalCatalog
- https://github.com/giisyu/svgToElm
- https://github.com/stephenbalaban/Gravity
- https://github.com/rtfeldman/dreamwriter
- https://github.com/evancz/elm-html-and-js
- https://www.classes.cs.uchicago.edu/current/22300-1/lectures
- https://github.com/dkarter/elm-bingo
- https://github.com/TheSeamau5/elm-undo-redo
- https://github.com/justinwoo/elm-etch-sketch
- https://github.com/netzwerg/elm-playground
- https://github.com/justinwoo/phoenix-no-brunch-webpack
- https://github.com/justinwoo/elm-simple-ports-example
- https://github.com/seliopou/elm-d3
- https://gist.github.com/evancz/2b2ba366cae1887fe621
- http://elm-lang.org/examples/cube
- https://github.com/TheSeamau5/simple-animations
- https://github.com/tennety/elm-game-of-life

- https://github.com/yasuyuky/elmtrn
- https://github.com/ohanhi/elm-native
- https://github.com/eeue56/elm-http-server
- https://github.com/Nazanin1369/elm-memoryGame
- https://github.com/eeue56/elm-http-server
- https://github.com/ohanhi/elm-native
- https://github.com/w0rm/elm-flatris (flat tetris game)
- https://github.com/artuuge/elm-collisions
- https://github.com/davidwatkins73/elm-reactive
- https://github.com/evancz/automaton
- https://github.com/vilterp/lemur
- https://github.com/pykello/alien-panic
- https://github.com/KualiCo/ReConfigurator
- https://github.com/VerbalExpressions/elm-verbal-expressions

Resources

- http://planet.elm-lang.org

Flyd - The minimalistic but powerful, modular, functional reactive programming library in JavaScript

React

- https://github.com/theadam/react-flyd
- https://github.com/gilbox/elegant-react
- https://github.com/gilbox/elegant-react-og
- https://github.com/gilbox/elegant-react-hot-demo
- https://github.com/gilbox/react-repl
- https://github.com/alexeygolev/elm-architecture-using-ramda-and-flyd
- https://github.com/ccorcos/elmish
- https://github.com/winitzki/flyd-react-sample
- https://github.com/winitzki/flyd-react-minesweeper
- https://github.com/winitzki/flyd-react-es6
- https://github.com/jwoudenberg/flyd-lift-react

Tutorials

- https://versal.com/c/xnedrc/learn/lessons/1

Other

- https://github.com/exound/lmount
- https://github.com/jayrbolton/vvvview
- https://github.com/paldepind/functional-frontend-architecture
- https://github.com/raine/flyd-playground
- https://github.com/ybybzj/r-stream
- https://github.com/somata/somata-flyd
- https://github.com/paldepind/flyd-aftersilence
- https://github.com/raine/flyd-keyboard
- https://github.com/ThomWright/flyd-connect

- https://github.com/ThomWright/flyd-cacheUntil
- https://github.com/ThomWright/flyd-onAnimationFrame
- https://github.com/ThomWright/flyd-timeInterval
- https://github.com/paldepind/flyd-switchlatest
- https://github.com/raine/flyd-droprepeats
- https://github.com/rektide/flyd2asyncgen
- https://github.com/paldepind/flyd-forwardto
- https://github.com/paldepind/flyd-filter
- https://github.com/paldepind/flyd-obj
- https://github.com/paldepind/flyd-every
- https://github.com/paldepind/flyd-flatmaplatest
- https://github.com/paldepind/flyd-scanmerge
- https://github.com/paldepind/flyd-takeuntil
- https://github.com/paldepind/flyd-lift
- https://github.com/paldepind/flyd-inlast
- https://github.com/paldepind/flyd-sampleon
- https://github.com/paldepind/flyd-keepwhen
- https://github.com/paldepind/flyd-flatmap

Kefir - Is a Reactive Programming library for JavaScript inspired by Bacon.js and RxJS, with focus on high performance and low memory usage

Angular

- https://github.com/xgrommx/angular-kefir
- https://github.com/mazelab/angular-kefir

Mithril

- https://github.com/oren/mithril-frp
- https://github.com/pixelpicosean/mithril-level-editor

React

- https://github.com/iofjuupasli/kefir-react
- https://github.com/Willmo36/FluxKefir
- https://github.com/lapanoid/react-kefir-flux
- https://github.com/pqr/react-flux-kefir-es6-todomvc-example
- https://github.com/gnapse/keflux
- https://github.com/f5io/react-kefir-todomvc
- https://github.com/gnapse/kefir-flux-todo
- https://github.com/kiloe/arla-todomvc
- https://github.com/lapanoid/react-kefir-flux-pouch-todo
- https://github.com/lapanoid/serialize-flux
- https://github.com/ahomu/Loxe
- https://github.com/saikocat/missing-frp-learn
- https://github.com/winitzki/kefir-react-es6
- https://github.com/iofjuupasli/kefir-react-example

Deku

- https://github.com/Artesonraju/deku-kefir-personal

Other

- https://github.com/elsehow/kefir-get
- https://github.com/systeminsights/kefir-contrib-crud
- https://github.com/pixelpicosean/panda-reactive
- https://github.com/eaglus/frp-component-vdom
- https://github.com/elsehow/notational-momentum
- https://github.com/gdeb/simple-blog-app
- https://github.com/AgentME/kefir-bus
- https://github.com/myndzi/kefir-node-stream
- https://github.com/ahomu/kefir.combinetemplate
- https://github.com/AgentME/kefir-stopper
- https://github.com/StreakYC/kefir-cast
- https://github.com/markmarijnissen/kefir-webworker
- https://github.com/beckyconning/tcomb-kefir
- https://github.com/dzdrazil/stream-router
- https://github.com/fixplz/up
- https://github.com/fixplz/kefir-store
- https://github.com/redpelicans/formo
- https://github.com/somata/somata-kefir
- https://github.com/iofjuupasli/kefir-bus-property
- https://github.com/iofjuupasli/kefir-proxy
- https://github.com/iofjuupasli/kefir-cron

Dialects

Purescript

- https://github.com/philopon/purescript-kefir
- https://github.com/EugeneN/pureGoL

RxJS - The Reactive Extensions for JavaScript

Angular

- **Standalone plugins**

    - **rx.angular.js** - Reactive Extensions Bindings for AngularJS
    - **ngMagicStream** - Models as Streams in Angular using RxJS
    - **reactiveproperty-angular** - ReactiveProperty for AngularJS
    - **angular-observe** - Angular directive for representing the different states of an Observable
    - **store** - RxJS powered Redux for Angular2 apps

- **Examples**

    - https://github.com/fxck/ng2-rx-redux (angular 2 + rx5 + redux)
    - https://github.com/cvuorinen/reactive-life
    - https://github.com/mjw56/ng2-rx (angular 2)
    - https://github.com/angular-class/angular2-reactive-starter (angular 2)
    - https://github.com/ninya-io/ninya.io
    - https://github.com/mikkark/TheCarGame
    - https://github.com/SekibOmazic/angular2-rx
    - https://github.com/xgrommx/angular-frp
    - https://github.com/mflorence99/frp-form
    - https://github.com/micmarsh/rx-coins
    - https://github.com/abenedykt/rxjs_angular
    - https://github.com/FreeCodeCamp/freecodecamp
    - http://plnkr.co/edit/kaOmmXVaBJdUbGUtvDIM
    - https://github.com/rolaveric/angular-falcor
    - http://plnkr.co/edit/wj4IRtAwNMry8ToNzLrm?p=info (angular 2 + redux + rxjs)
    - https://github.com/ng-book/angular2-rxjs-chat (angular 2)
    - http://plnkr.co/edit/WT6JK3o1XZ8PuWJcPVV2?p=preview (angular 2)
    - http://plnkr.co/edit/pUfJ0b3Kay59PfkPEBlo?p=preview (angular 2 + redux)
    - https://github.com/r-park/todo-angular-rx (angular 1.5)
    - https://github.com/veyo-care/angular-2-redux

- https://github.com/ariesxje/ng2-rxjs-tryout

React

- ## Standalone plugins

  - ### React

    - **rx-react** - ReactJS bindings for RxJS
    - **react-bacon-component** - Utility for attaching Bacon.js to React components
    - **react-observable-mixin** - Container like observable mixin for ReactJS
    - **retransmitter** - Async-friendly stateful React containers
    - **rx-recompose** - RxJS utilities for Recompose
    - **react-combinators** - Seamless combination of React and reactive programming
    - **react-rx-subscriber** - React component enhancer to easily subscribe to Rx Observables

  - ### Flux

    - **rx-flux** - A Flux architecture implementation based on RxJS
    - **River** - Reactive stream composer for React application
    - **redux-core** - Minimal Redux
    - **rx-redux** - A reimplementation of redux using RxJS
    - **thisless-react** - Reactive React application flow with React, RxJS and Redux
    - **Flow** - A Reactive Extension flux like library based on Rx
    - **flux-rx** - Flux components which can be used to implement Flux within a React.js application. Combines Rx.JS and Immutable.JS for Dispatcher/Store implementations
    - **realm** - A boilerplate for creating scalable apps with react, rxjs and a little bit of magic. Heavily inspired by flux, redux and elm architecture
    - **scanux** - A simple Flux implementation
    - **rx-store** - Reactive JS library for managing state
    - **gleedux** - Bringing joy and happiness to Redux using RxJS
    - **luna** - a reactive redux library written using Rxjs with built-in thunk and async action support
    - **fluorine** - State Accumulation from a single stream of actions
    - **luxor** - Flux-ish action/store framework. Built with Typescript and Rx.js

- ## Examples

- https://github.com/incrediblesound/RxReact-ToDo
- https://github.com/mattpodwysocki/rx-fluxish
- https://github.com/zxbodya/rx-react-container
- https://github.com/almasakchabayev/react-rx-falcor
- https://github.com/winsonwq/erc
- https://github.com/foxdonut/adventures-reactive-web-dev
- https://github.com/jamesknelson/numbat-ui-example
- https://github.com/yanglinz/reddio
- https://github.com/brabadu/tanok Elm architecture
- https://gist.github.com/sectore/a8147ce68fc3966ce142
- https://github.com/bordoley/RelaX
- https://github.com/weslleyaraujo/react-flux-puzzle
- https://github.com/bennyhobart/rxjs-react-todomvc
- https://github.com/foxdonut/react-rxjs-todolist
- https://github.com/mauriciosoares/todomvc-react-rxjs
- https://github.com/findjashua/react-elm
- https://github.com/mauriciosoares/rxjs-react-example
- https://github.com/jschr/frp-pong
- https://github.com/justinwoo/react-14-rx-fake-elm-arch
- https://github.com/jas-chen/react-reactive-class
- https://github.com/jcouyang/react-falcor
- https://github.com/yevk/react-rx-pipeline-prototype
- https://github.com/fdecampredon/react-rxjs-todomvc
- https://github.com/AlexMost/RxReact
- https://github.com/joelhooks/react-rxjs-angular-di-todomvc
- https://github.com/footballradar/rx-todomvc
- https://github.com/Willmo36/FluxRx
- https://github.com/r3dm/thundercats
- https://github.com/codesuki/reactive-flux
- https://github.com/cell303/flurx
- https://github.com/andreypopp/react-async
- https://github.com/justinwoo/react-rxjs-flow
- https://github.com/justinwoo/react-rxjs-flow/tree/dynamic-width-scroll-table
- https://github.com/kmcclosk/reactjs-rxjs-example.git
- https://github.com/fdecampredon/rx-react-example
- https://github.com/winsonwq/RR
- https://gist.github.com/zxbodya/20c63681d45a049df3fc
- https://github.com/erikschlegel/sails-rx-react-js
- https://github.com/hugobessaa/rx-react-pinch
- https://github.com/ahomu/hn-react-rxjs.git

- https://github.com/holsee/ReactiveReact
- https://github.com/joelhooks/reactive-sandbox
- https://github.com/zxbodya/reactive-widgets
- https://github.com/justinwoo/react-rxjs-scroll
- https://github.com/eliseumds/react-autocomplete
- https://github.com/doctolib/floux
- https://github.com/ahomu/Loxe
- https://github.com/justinwoo/react-fixed-scroll-element
- https://github.com/hugobessaa/cubactive
- https://github.com/r3dm/r3dm.com
- https://github.com/Mosho1/react-rx
- https://gist.github.com/nicolashery/6dec8b7b3c9271ff18c9
- https://github.com/Cmdv/React-RxJS
- https://github.com/jamesknelson/maxim
- https://github.com/bobiblazeski/rio
- https://github.com/gyzerok/rxflux
- https://github.com/stepennwolf/rx-todo
- https://github.com/oscarduignan/react-rxjs-elasticsearch-faceted-search-example
- https://github.com/vslinko/observable-components-test
- https://github.com/usco/Jam
- https://github.com/gah-boh/trendyjs-cal
- https://github.com/xgrommx/react-rx-flux
- https://github.com/shovon/todomvc-react-rxjs
- https://github.com/milankinen/ffux (interesting implementation)
- https://github.com/terpiljenya/react-rxjs-list
- https://github.com/dqdinh/flux-chat-with-bacon
- https://github.com/dqdinh/flux-chat-example-rxjs
- https://github.com/canalplus/rx-player
- https://github.com/acdlite/redux-rx
- https://github.com/acdlite/react-rx-component
- https://github.com/gilbox/react-spark-scroll
- https://github.com/oscarduignan/koa-react-router-rxjs-example
- https://github.com/flipflops/observable-store
- https://github.com/pstoica/hot-reload-rxjs
- https://github.com/mgonto/reactive-frontend-oscon
- https://github.com/skellyb/fluxstream
- https://github.com/ThomasBrekelmans/stratego
- https://github.com/mrspeaker/pathe
- https://github.com/kamalmarhubi/todomvc-es6-react-rxjs-elm-architecture
- https://github.com/newtriks/react-rx-immutable-example

- https://github.com/jtangelder/rxApp

MongoDB

- https://github.com/data-avail/da-mongo-rx

PostgreSQL

- https://github.com/jadbox/rxjs-massive

Meteor

- https://github.com/bobiblazeski/rxjs-react

Node

- https://github.com/Reactive-Extensions/rxjs-node

Ember

- https://github.com/blesh/RxEmber

Can

- https://github.com/canjs/can.rx

Mercury

- https://github.com/pH200/mercury-rx

Mithril

- https://github.com/ariutta/mithril-rxjs-demos

Knockout

- https://github.com/Igorbek/knockout.rx

Polymer

- https://gist.github.com/paulcbetts/2877b05a57d0a4e0ab54

Riot

- http://codepen.io/filippomangione/pen/JYOZBY

Vue

- https://github.com/vuejs/vue-rx

Pub/Sub

- https://github.com/rxmqjs/rxmq.js
- https://github.com/mchmielarski/busbup

WebRx

- https://github.com/patsissons/WebRx.React/tree/develop

Yolk - A library for building asynchronous user interfaces.

- https://github.com/yolkjs/yolk

Visualizing streams

- https://github.com/staltz/rxmarbles
- https://github.com/jaredly/rxvision
- https://github.com/justinwoo/rx-lightbulbs
- http://jsbin.com/hovegi/1/edit?js,output
- https://jsfiddle.net/y63p8b9b/light
- https://github.com/amilosevic/rxdiag
- https://github.com/grisendo/Percussion
- https://github.com/Astn/rxjs-marble-diagram
- https://github.com/iamssen/rxjs-marble-renderer

Dialects

Purescript

- https://github.com/anttih/purescript-rx
- https://github.com/bodil/purescript-signal
- https://gist.github.com/roman/98c0999ba003b6a8f6c5

Tutorials

- https://github.com/staltz/rxjs-training
- https://gist.github.com/staltz/868e7e9bc2a7b8c1f754
- https://github.com/channikhabra/yarr/blob/master/complete-tutorial.org
- http://jhusain.github.io/learnrx
- https://egghead.io/technologies/rx
- http://rxwiki.wikidot.com
- http://www.introtorx.com
- https://github.com/erwanthomas/learnrx-bdd
- Introduction to the Reactive Extensions for JavaScript
- https://github.com/supersymme3/learnrx-exercises
- https://github.com/JulienMoumne/rx-training-games

Other

- https://github.com/alexvictoor/RxJsKata
- https://github.com/kt3kstudio/long-dream-core
- https://github.com/fdecampredon/react-observe-todomvc
- https://github.com/trxcllnt/rxjs-easing
- https://github.com/staltz/rxtween
- https://github.com/blesh/RxSocketSubject
- https://github.com/paulpdaniels/rx.wamp
- https://github.com/jhusain/compositional-functions
- https://github.com/bodil/erx
- https://github.com/stomita/rxpression
- https://github.com/alekspolitov/ReactAngularRxJs
- https://github.com/brettshollenberger/rx_maze
- https://github.com/johntron/rxjs-d3-example
- https://github.com/kriskowal/gtor
- https://github.com/bleathem/locationforcelayout
- https://github.com/blesh/multi-target-build-scaffold
- https://github.com/huang47/rx-event-manager
- https://github.com/trxcllnt/rxjs-extras
- https://github.com/ahomu/rx.observable.combinetemplate
- https://github.com/myhau/approximation
- https://github.com/cbelden/rx-pagination
- https://github.com/Reactive-Extensions/FutureJS
- https://github.com/jtwebman/RxExpressJS
- https://github.com/mgonto/reactive-all-the-things-talk
- https://github.com/trxcllnt/rxjs-gestures
- https://github.com/jwooley/RxSamples
- https://github.com/ericelliott/irecord
- https://github.com/lipsmack/router-rx
- https://github.com/nemanja-stanarevic/firebase-promisified
- https://github.com/yamalight/postal.observe
- https://github.com/hlmerscher/pong-rxjs
- https://github.com/cbelden/RxJS-tic-tac-toe
- https://github.com/theefer/talk-reactive-loop
- https://github.com/wikp/inka.js
- https://github.com/ryanj/hexboard
- http://github.com/zenparsing/es-observable
- https://bitbucket.org/iyegoroff/rx-interactions
- https://github.com/blesh/rx-talk-may-2015
- https://github.com/supershabam/rxjs-mm
- https://github.com/JosephMoniz/rx-request-handler

- https://github.com/msscully/NNRx
- https://github.com/cwharris/rxjs-minesweeper
- https://github.com/cwharris/rxjs-tictactoe
- https://github.com/cwharris/rxjs-graphsweeper
- https://github.com/cwharris/rxjs-checkers
- https://github.com/cwharris/rxjs-graphtoy
- https://github.com/pstoica/rx-music
- https://github.com/cwharris/rxjs-splash
- https://github.com/trxcllnt/rx-node-template
- https://github.com/logicalguess/rx-state-machine
- https://github.com/trxcllnt/rxjs-fs
- https://github.com/JosephMoniz/rx-map
- https://github.com/TimPapler/rx-mongoose
- https://github.com/yamalight/superagent-rx
- https://github.com/robwormald/rx-slack
- https://github.com/itechdom/rx-examples
- https://github.com/robwormald/rx-router
- https://github.com/channikhabra/frp-with-rxjs-jschannel-conf
- https://github.com/alexander-matsievsky/circuitry.ts
- https://github.com/chcokr/firebase-rx
- https://github.com/data-avail/da-rabbitmq-rx
- https://github.com/staltz/combineLatestObj
- **routerx** - A stupid simple functional reactive router built for use with Rx.js apps
- https://github.com/hpinsley/RxPlay
- https://github.com/zhangchiqing/milestones
- https://github.com/tategakibunko/inga
- https://github.com/erikschlegel/RxBing
- https://github.com/tomlarkworthy/animaxe (Animation with Rx)
- https://github.com/nramadas/Storable
- https://github.com/stylelab-io/event-emitter-rx
- https://gist.github.com/mattpodwysocki/9aeb5ce339cdb38a5dc2
- https://github.com/AlexMost/morse
- https://github.com/ziaukhan/learn-rxjs-typescript
- https://github.com/christyharagan/koa-rx-router
- https://github.com/rajeshsegu/request-rx
- https://github.com/achydenius/rxjs-web-midi
- https://github.com/itechdom/rx-examples
- https://github.com/gchudnov/rx-image
- https://github.com/akauppi/svg.rx.js
- https://github.com/psycotica0/bdd_game

- https://github.com/Widdershin/rx-undoable
- https://github.com/doctolib/rx
- https://github.com/ospatil/rx-textsearch
- https://github.com/ajbertra91/footballCardGame
- https://github.com/alexlrobertson/understanding-rx
- https://github.com/JayKan/RxJS-Playground
- https://github.com/gschambers/router-rx
- **cv-decorators** - List of ES6 decorators
- https://github.com/codeflyer/hack-rxjs
- https://github.com/petuomin/advent-of-code-rxjs
- https://github.com/Widdershin/rx-audible
- https://github.com/jeremymwells/rxstream
- https://github.com/jadbox/rxjs-cluster

Build tools

- https://github.com/plumberjs/plumber

Presentations

Rx

- Reactive Extensions for JavaScript
- Add Some Fun to Your Functional Programming With RXJS
- Rethink Async With RXJS
- Intro to Reactive Extensions
- Reactive Programming - Curing your asynchronous programming blues
- Reactive Extensions(Rx) - Rx = Observables + LINQ + Schedulers.
- Functional Reactive Programming with RxJS
- Functional Reactive Programming / Compositional Event Systems
- Reactive programming in javascript
- What if the user was a function?
- Reactive game development for the discerning hipster
- Functional Reactive Programming
- Functional Reactive Programming
- RxJS and Reactive Programming - Modern Web UI - May 2015
- The Reactive Loop
- Infinite scrolling with angular & rxjs
- Reactive programming with Rx.js
- Reactive functions with RxJS
- Raw and uncut slideshow software used for the Functional Reactive Programming uncon talk at DPC 2015
- Dealing with streams using RxJs
- ES2016 Observable
- Reactive Programming - Curing your asynchronous programming blues
- Functional Reactive Programming in Javascript with RxJS : JsChannel Conf 2015
- Reactive frontend
- Functional Reactive Programming for front-end developers
- Building fully reactive user interfaces with Rx Observables and Yolk.js
- Going Reactive with React
- Rx.js make async programming simpler
- Tanok - Elm Arhictecture in JS
- Reactive Programming in JavaScript by Ayumu Sato
- RxJS Evolved
- Pimp your Redux app with RxJS

- Fully Reactive User Interfaces
- A Playful Introduction to Rx
- Reactive Programming with Rx (*)
- Mini training - Reactive Extensions (Rx) (*)

Bacon

- Reactive Programming in Practice – (Using Bacon JS)
- Functional Reactive Programming with Bacon.js
- Reactive Programming in practice (using Bacon js)
- Functional Reactive Programming and Bacon.js
- FrontendLab: Programming UI with FRP and Bacon js(ru)
- Codemotion 2015 - Functional Reactive Programming: FP, Javascript con extra de bacon

Other

- Functional Programming Patterns
- Functional Reactive Programming
- Functional Reactive Programming in javascript
- Cycle.js an honestly reactive framework
- Functional Reactive Programming in JavaScript
- Let it Flow - Introduction to Functional Reactive Programming
- OMG Streams
- FRP and you
- Functional Reactive Programming / Compositional Event Systems
- Reactive design: languages, and paradigms
- A journey in the world of Reactive Programming with JavaScript (*)
- Reactive Programming

Cycle

- From jQuery to Cycle
- Et si l'utilisateur était une fonction ?

RxJava

- Applying RxJava to Existing Applications at Philly ETE 2015
- FRP on Android
- Vert.x and RxJava
- Reactive Programming with RxJava
- Exploring the possibilities of Unidirectional Data Flow Architectures on Android
- Demystifying RxJava Subscribers (Øredev 2015)
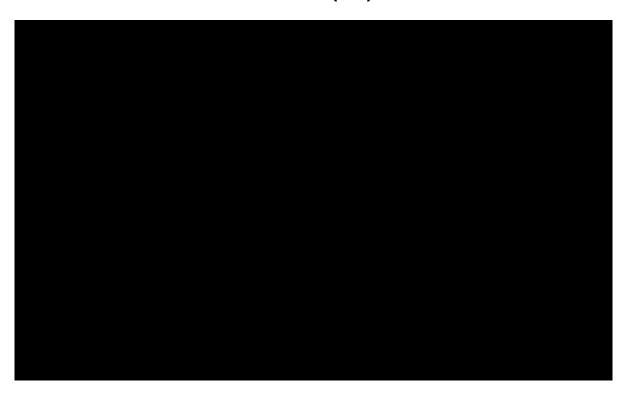
Elm

- Elm kyivfprog 2015

Video
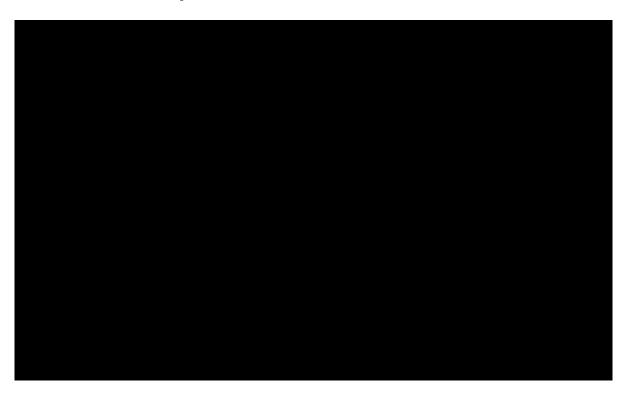
- **Expert to Expert: Brian Beckman and Erik Meijer - Inside the .NET Reactive Framework (Rx)**

- **Matthew Podwysocki - Streaming and event-based programming using FRP and RxJS (FutureJS 2014)**

- **Netflix JavaScript Talks - Falcor**

- **Netflix JavaScript Talks - Version 7: The Evolution of JavaScript**
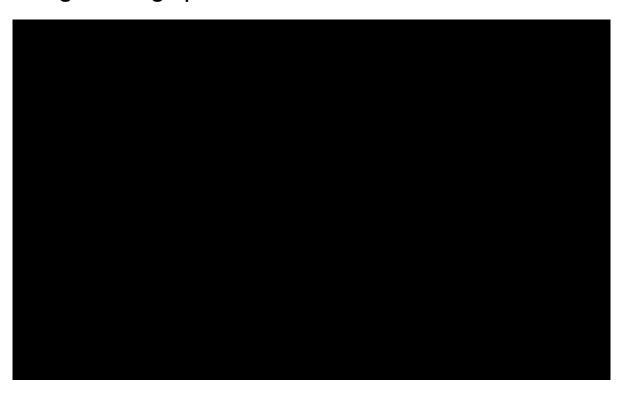
- **Netflix JavaScript Talks - Async JavaScript with Reactive Extensions**

- **Keynote - JSON Graph: Reactive REST at Netflix**
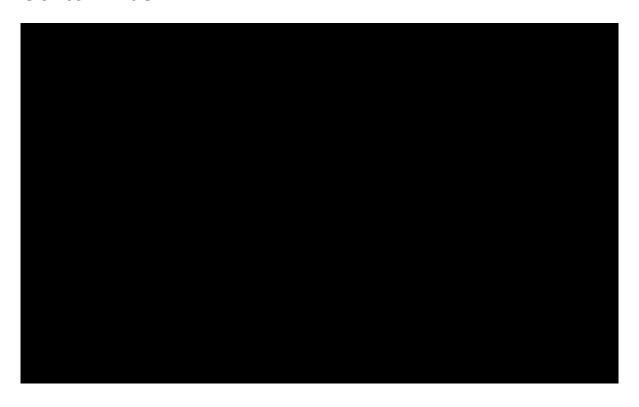
- **Matthew Podwysocki: Async and Streaming JS - Are we doing it wrong? | JSConf EU 2014**

- **Philly ETE 2014 #39 - Reactive Javascript at Netflix - Jafar Husain & Matthew Podwysocki**

- **Reactive all the Things - Ben Lesh and Martin Gontovnikas**

- **Ian Hansen - RxJS for Metrics and Music**

- **Getting the most out of $http and Intro to rxjs (and the angular plugin)**
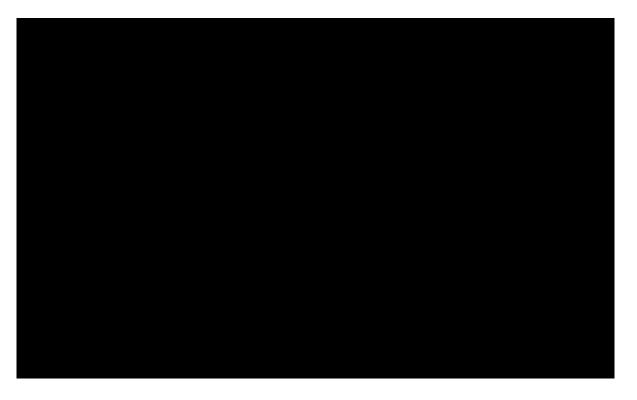
- **MountainWest JavaScript 2014 - Adding Even More Fun to Functional Programming With RXJS**
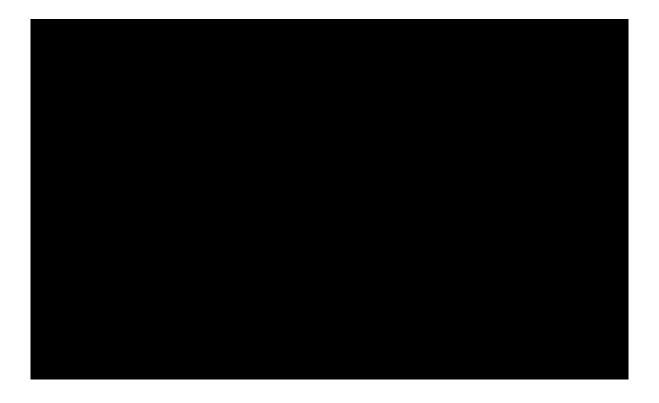
- **RxJS**
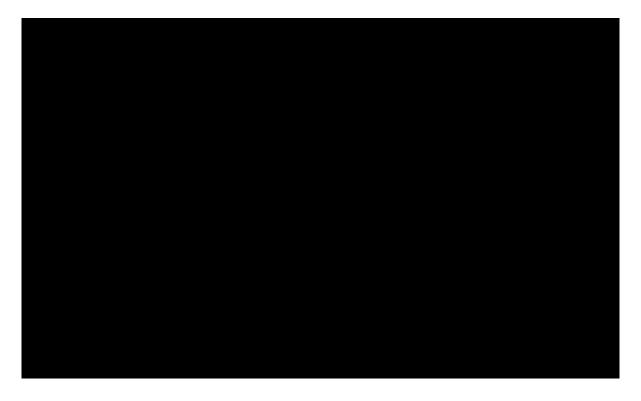
- **CascadiaJS 2012 - Matthew Podwysocki**

- **Netflix JavaScript Talks - Async JavaScript with Reactive Extensions**

- **MountainWest JavaScript 2014 - Asynchronous JavaScript at Netflix by Jafar Husain**

- **HTML5DevConf Jafar Husain, Netflix: Asyncronous JavaScript at Netflix**

- **Bodil Stokke: Reactive Game Development For The Discerning Hipster [JSConf2014]**
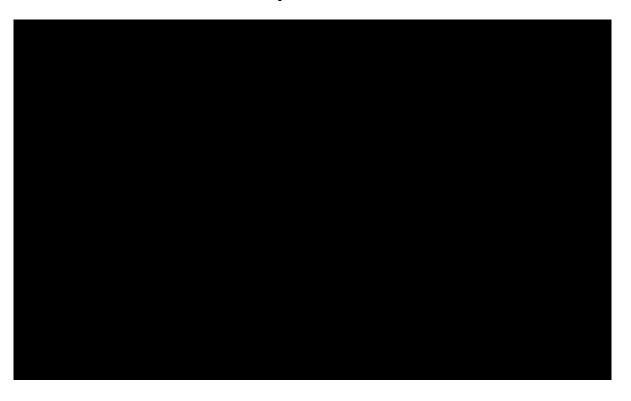
- **A Playful Introduction to Rx by Erik Meijer**

- **HTML5DevConf: Jafar Husain, Netflix: Version 7: The Evolution of the JavaScript**
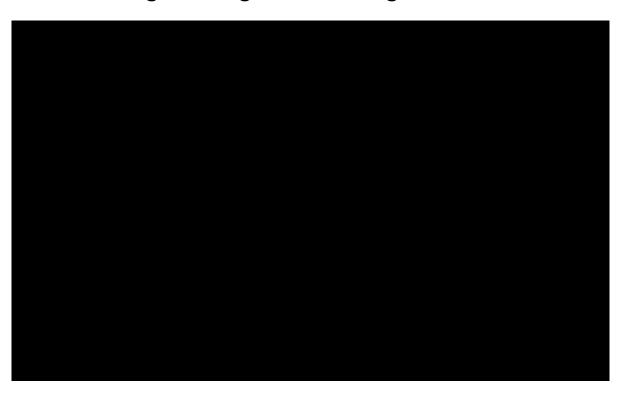
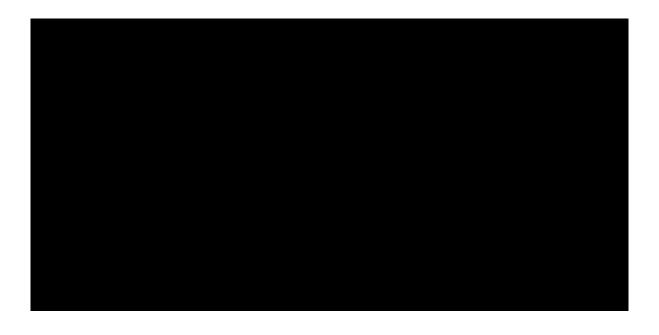- **Async Programming at Netflix by Matthew Podwysocki at Web Rebels 2015**

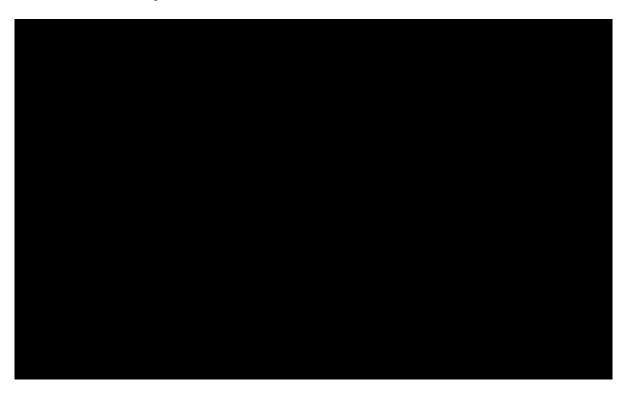- **Reactive Programming for Java Programmers**

- **Asynchronous JavaScript at Netflix by Matthew Podwysowski at JSConf Budapest 2015**
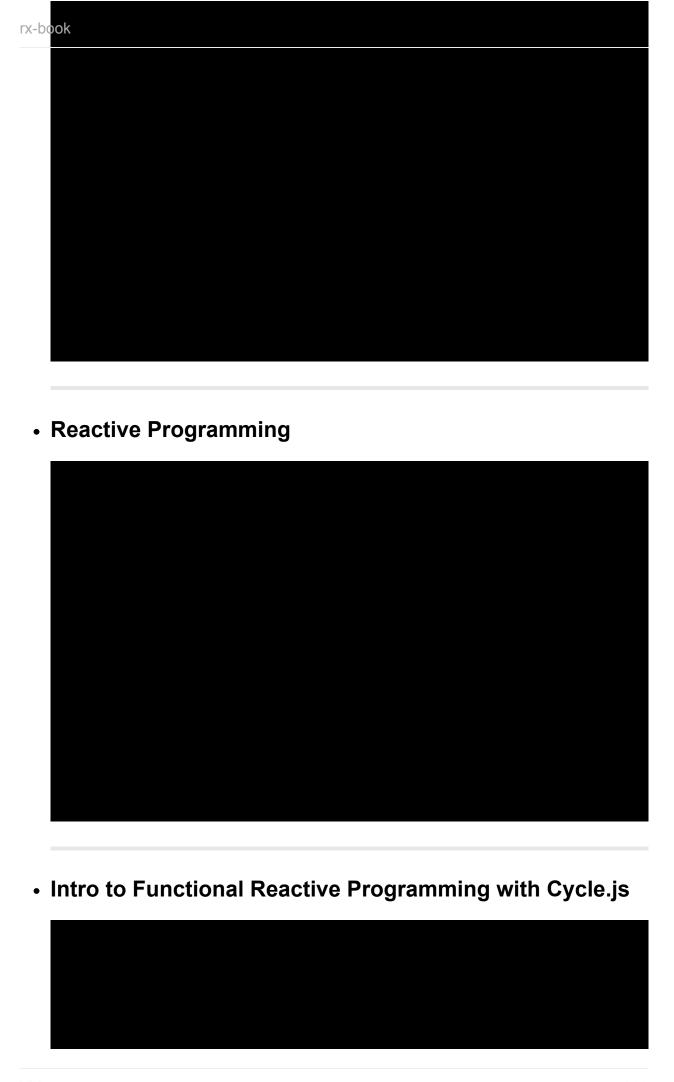
- **What if the user was a function? by Andre Staltz at JSConf Budapest 2015**

- **Async and Streaming JavaScript, We're All Doing it Wrong! - Matthew Podwysocki**

- **Reactive Programming**

- **Intro to Functional Reactive Programming with Cycle.js**

- **React 2014 : Erik Meijer - What does it mean to be Reactive?**

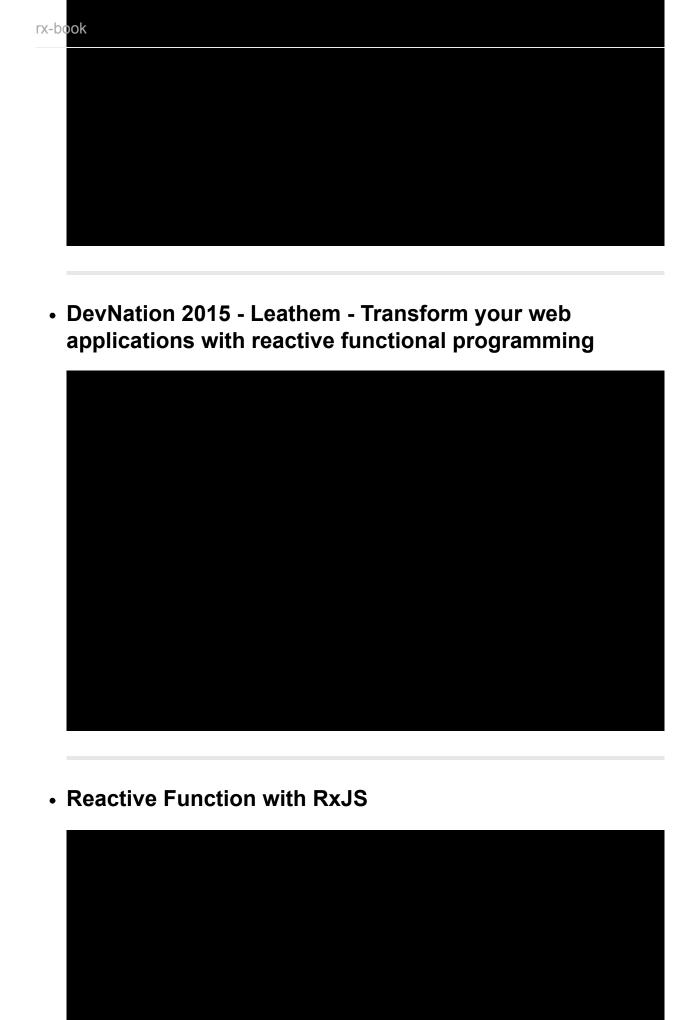- **Evan Czaplicki - Let's be mainstream! User focused design in Elm - Curry On**

Video

- **Functional Reactive Programming in Elm**

- **John P Mayer - Delightful WebGL in Elm**

- **DevNation 2015 - Leathem - Transform your web applications with reactive functional programming**

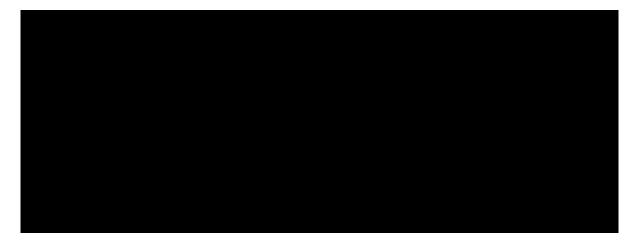- **Reactive Function with RxJS**

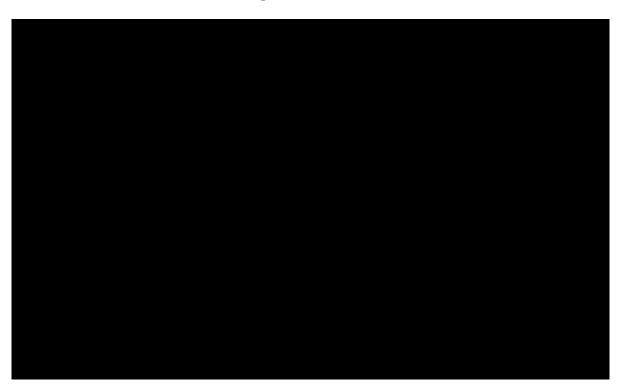- **Functional Reactive Programming in your browser by Vagmi Mudumbai**

- **Erik Hinton on Functional Reactive Programming (TimesOpen: Reactive Programming)**
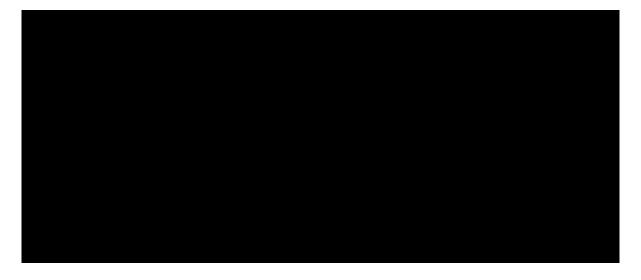
- **Reactive Streams in Angular 1 and 2 Ben Lesh**

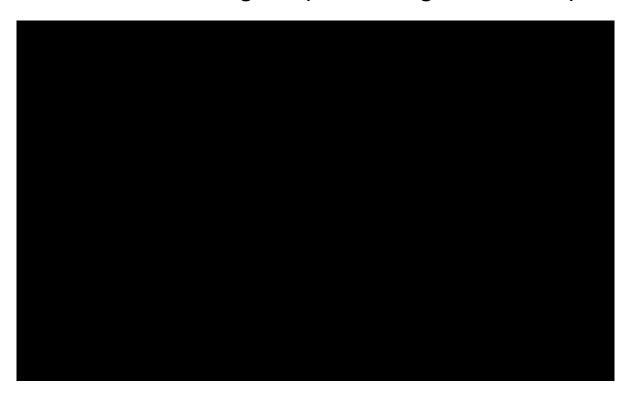- **"Controlling Time and Space: understanding the many formulations of FRP" by Evan Czaplicki**

- **RxJS used to build a game (enhanced game features)**
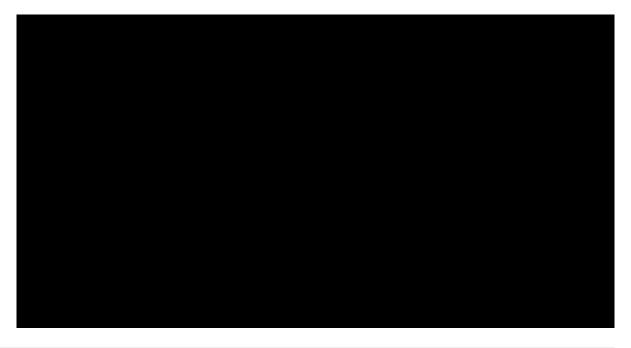
- **The functional reactive frontend**

- **.NET Conf UY v2015 - Building Reactive Services using Functional Programming by Rachel Reese**

- **Lambda Jam 2015 - Conal Elliott - The Essence and Origins of Functional Reactive Programming**

- **Reactive Programming**

- **I have a stream – Einsichten in Reactive Programming**

- **Reactive.community: Ben Christensen, Reactive Extensions (Rx) at Netflix**

- **An Introduction to Functional Reactive Programming using [Sir Francis]Bacon.js**

- **Real-Time Deadlines in Functional-Reactive Programming**
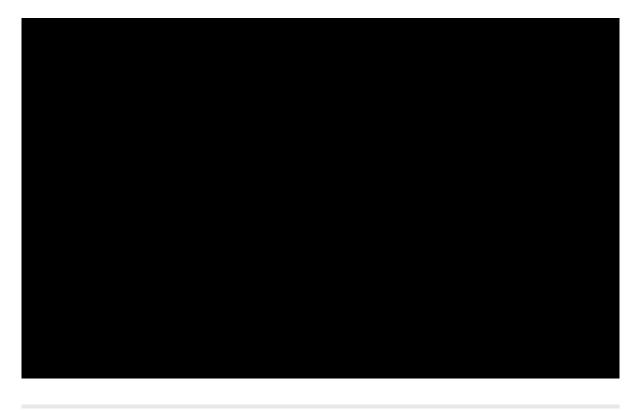
- **Akka.NET with F# Getting functional with Reactive systems - Russ Cam (@forloop)**
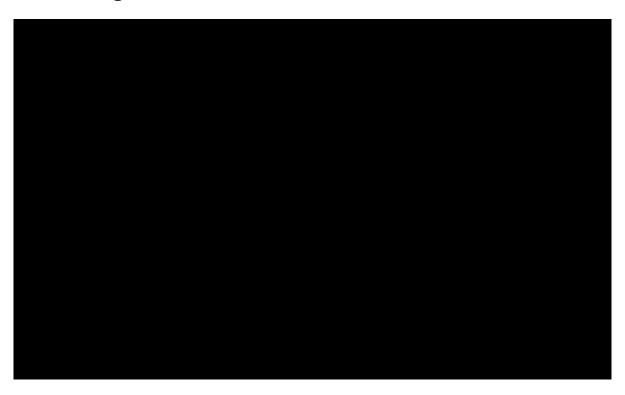
- **Bacon.js for Breakfast: An intro to functional reactive programming**

Video   674

- **Angular 2 Data Flow**

- **Angular 2 Data Flow – Jeff Cross, Rob Wormald and Alex Rickabaugh**
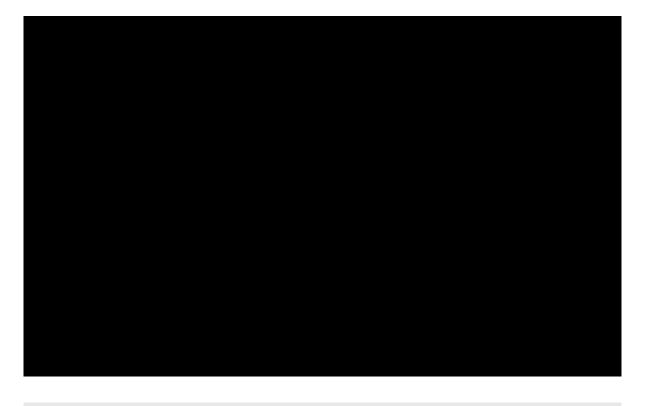
- **Philip Roberts: Enemy of the state: An introduction to Functional Reactive Programming and Bacon.js**

- **Wellington FP - Why Functional Reactive Programming FRP?**
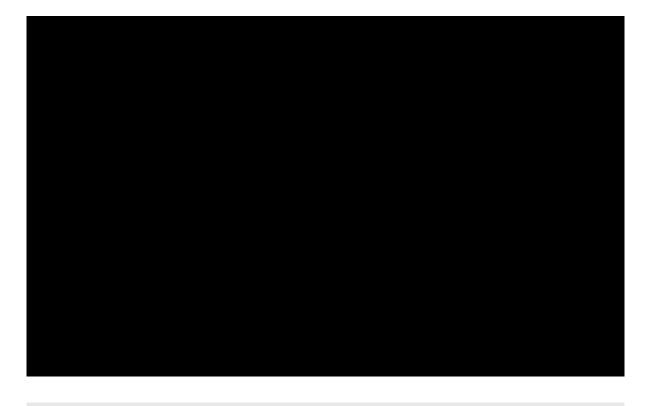
- **Jeffrey Van Gogh: The Reactive Extensions for JavaScript**

- **Bacon.js - Functional Reactive Programming - Thomas Holmes - ATLRUG 2014**

- **Functional Reactive Programming with RxJava • Ben Christensen**

- **Juha Paananen. FRP with Bacon.js**

- **"PureScript (Maybe This Time We Get JavaScript Right)" by Bodil Stokke**

- **PolyConf 15: Post FRP Frontend Programming / Bodil Stokke**

rx-book

RxJS Recipes

Error Handling

There are many ways of handling errors in RxJS. This will cover a number of recipes when handling errors:

- Incremental retry with `retryWhen`

Incremental retry with `retryWhen`

This example will show how you can back off a retry for a number of seconds between tries instead of retrying them all at once.

Example

Recipes   680

Which Operator do I use?

- Creation Operators
- Instance Operators

Which Operator to Use? - Creation Operators

Use this page to find the creation operator implemented by the `Observable` type that fits your needs:

Static methods

I want to create a new sequence	using custom logic			Observable.create
		that works like a for-loop		Observable.generate
			and emits the values over time	Observable.generateW
				Observable.generateW
	that returns a value			Observable.return/just
		multiple times		Observable.repeat
	that throws an error			Observable.throw
	that completes			Observable.empty
	that never does anything			Observable.never
	from an event			Observable.fromEvent
		that uses custom functions to add and remove event handlers		Observable.fromEventF
	from an [ES6 Promise](#)			Observable.fromPromis
	that iterates	over the values in an array		Observable.fromArray
			of object key/values	Observable.pairs
		of asynchronous elements	Observable.for	
		over values in a numeric range		Observable.range
		over the values in an iterable, array or array-like object		Observable.from
		over arguments		Observable.of
	that emits values on a timer			Observable.interval
		with an optional initial delay		Observable.timer
			on a specific	Observable.start

	arguments		scheduler	Observable.start
			asynchronously	Observable.startAsync
	decided at subscribe-time	based on a boolean condition		Observable.if
		from a pre-set set of sequences		Observable.case
		using custom logic		Observable.defer
			that depends on a resource	Observable.using
I want to wrap a function			and yield the result in a sequence	Observable.toAsync
	which accepts a callback			Observable.fromCallba
	which accepts a Node.js callback			Observable.fromNodeC
I want to combine multiple sequences	and only receive values from the sequence that yields a value first			Observable.amb
	and be notified when all of them have finished			Observable.forkJoin
	and output the values from all of them			Observable.merge
	in order	reusing the latest value when unchanged		Observable.combineLa
		using each value only once		Observable.zip
	by subscribing to each in order	when the previous sequence completes		Observable.concat
		when the previous sequence errors		Observable.catch
		regardless of whether the previous sequence completes or errors		Observable.onErrorRes
	by responding to different combinations of values (join calculus)			Observable.when

See Also

Reference

- `Observable`

Concepts

- Querying Observable Sequences
- Operators By Category

Which Operator to Use? - Instance Operators

Use this page to find the instance operator implemented by the `Observable` type that fits your needs:

Instance methods

	I want to change each value		map/select
	I want to pull a property off each value		pluck
	I want to be notified of values without affecting them		do/tap doOnNext/tapOnNext doOnError/tapOnError doOnCompleted/tapOnCor
	I want to include values	based on custom logic		filter/where
		from the start of the sequence		take
			based on custom logic	takeWhile
		from the end of the sequence		takeLast
		until another sequence emits a value or completes		takeUntil
	I want to ignore values	altogether		ignoreElements
		from the start of the sequence		skip
			based on custom logic	skipWhile
		from the end of the sequence		skipLast
		until another sequence emits a value		skipUntil
		that have the same value as the previous		distinctUntilChanged
		that occur too frequently		throttle
		the sum	of its values	sum
		the average		average

	I want to compute	using custom logic	and only output the final value	aggregate reduce
			and output the values as they are calculated	scan
	I want to wrap its messages with metadata	that describes each message		materialize
		that includes the time past since the last value		timeInterval
		that includes a timestamp		timestamp
	after a period of inactivity	I want to throw an error		timeout
		I want to switch to another sequence		timeout
	I want ensure there is only one value	and throw an error if there are more or less than one value		single
		and use the default value if there are no values		singleOrDefault
	I want to only take the first value	and throw an error if there are no values		first
		and use the default value if there are no values		firstOrDefault
		within a time period		sample
	I want to only take the last value	and error if there are no values		last
		and use the default value if there are no values		lastOrDefault
	I want to know how many values it contains			count
	I want to know if it includes a value			contains
	I want to know if a condition is satisfied	by any of its values		any/some
		by all of its values		all/every
	I want to delay messages by a specific amount of time			delay
			based on custom logic	delayWithSelector

Using an existing sequence	I want to group the values	until the sequence completes		toArray toMap toSet
		using custom logic	as arrays	buffer
			as sequences	window
		in batches of a particular size	as arrays	bufferWithCount
			as sequences	windowWithCount
		based on time	as arrays	bufferWithTime
			as sequences	windowWithTime
		based on time or count, whichever happens first	as arrays	bufferWithTimeOrCount
			as sequences	windowWithTimeOrCount
		based on a key	until the sequence completes	groupBy
			and control the lifetime of each group	groupByUntil
	I want to start a new sequence for each value	and emit the values from all sequences in parallel		flatMap/selectMany
		and emit the values from each sequence in order		concatMap/selectConcat
		and cancel the previous sequence when a new value arrives		flatMapLatest/selectSwitch
		and recursively start a new sequence for each new value		expand
		and emit values from all sequences depending for onNext, onError, and onCompleted in parallel		flatMapObserver/selectMar
		and emit values from all sequences depending for onNext, onError, and		concatMapObserver/select

		onCompleted in order	
	I want to combine it with another	And be notified when both have completed	forkJoin
	I want to perform complex operations without breaking the fluent calls		let
	I want to share a subscription between multiple subscribers	using a specific subject implementation	multicast
			publish share
		and suppy the last value to future subscribers	publishLast shareLast
		and replay a default or the latest value to future subscribers	publishValue shareValue
		and replay *n* number of values to future subscribers	replay shareReplay
	when an error occurs	I want to re-subscribe	retry
		I want to start a new sequence	catch
		that depends on the error	catch
	when it completes	I want to re-subscribe	repeat
		I want to start a new sequence	concat
	when it completes or errors	I want to start a new sequence	onErrorResumeNext
	when it completes, errors or unsubscribes	I want to execute a function	finally
	I want to change the scheduler that routes	calls to subscribe	subscribeOn
		messages	observeOn
	I want to decide which to receive values from	based on which one has values first	amb
	I want to determine if their values are equal		sequenceEqual

Using two sequences	I want to combine their values	only when the first sequence emits, using the latest value from each		withLatestFrom
		in order	reusing the latest value when unchanged	combineLatest
			using each value only once	zip
		that share overlapping "lifetime" that I choose	and be notified for each combination	join
			and be given a sequence of "rights" for each "left"	groupJoin
	I want to include values from both			merge

See Also

Reference

- `Observable`

Concepts

- Querying Observable Sequences
- Operators By Category