CONTENTS

REACTIVE PROGRAMMING IN JAVASCRIPT WITH

RxJS

BY LUIS ATENCIO

INTRODUCTION

As experienced JavaScript developers, we're used to dealing with a certain level of asynchronous computations in our code. We're constantly asked to process user input, fetch data from remote locations, or run long-running computations simultaneously, all without halting the browser. Arguably, these are not trivial tasks and certainly require that we learn to step away from the synchronous computation paradigm, and step into a model where time and latency become key issues. For simple applications, using JavaScript's main event system directly or even wrapped with the help of jQuery libraries is common practice. However, without the proper paradigm in place, scaling the simple code that solves these asynchronous problems to a richer and feature-complete app that meets the needs of the modern day web user, is still difficult. What we find is that our application code becomes tangled, hard to maintain, and hard to test. The problem is that asynchronous computations are inherently difficult to manage, and RxJS solves this problem.

THE PROBLEM RXJS SOLVES

One of the most important goals of any application is to remain responsive at all times. This means that it's unacceptable for an application to halt while processing user input or fetching some additional data from the server via AJAX. Generally speaking, the main issue is that IO operations (reading from disk or from the network) are much slower than executing instructions on the CPU. This applies both to the client as well as the server. Let's focus on the client. In JavaScript, the solution has always been to take advantage of the browser's multiple connections and use callbacks to spawn a separate process that takes care of some long-running task. Once the task terminates, the JavaScript runtime will invoke the callback with the data. This a form of *inversion of control*, as the control of the program is not driven by you (because you can't predict when a certain process will complete), but rather under the responsibility of the runtime to give it back to you. While very useful for small applications, the use of callbacks gets messy with much richer that need to handle an influx of data coming from the user as well as remote HTTP calls. We've all been through this: as soon as you need multiple pieces of data you begin to fall into the popular "pyramid of doom" or callback hell.

```
makeHttpCall('/items',
   items => {
      for (itemId of items) {
         makeHttpCall('/items/${itemId}/info',
           itemInfo => {
              makeHttpCall('/items/${itemInfo.pic}',
                img => {
                    showImg(img);
              });
           });
      }
});
beginUiRendering();
```

This code has multiple issues. Once of them is style. As you pile more logic into these nested callback functions, this code becomes more complex and harder to reason about. A more subtle issue is created by the `for` loop. A `for` loop is a synchronous control flow statement that doesn't work well with asynchronous calls that have latency, which could lead to very strange bugs.

Historically, this has been a very big problem for JavaScript developers, so the language introduced Promises in ES6. Promises help shoulder some of these issues by providing a nice fluent interface that captures the notion of time and exposes a continuity method called `then()`. The same code above becomes:

```
makeHttpCall('/items')
    .then(itemId => makeHttpCall('/items/${itemId}/info'))
    .then(itemInfo => makeHttpCall('/items/${itemInfo}.pic}'))
    .then(showImg);
```

Certainly this is a step in the right direction. The sheer mental load of *reading* this code has reduced dramatically. But promises have some limitations, as they are really efficient for working with single value (or single error) events. What about handling user input where there's a constant flow of data? Promises are also insufficient to handle events because they lack semantics for event cancellation, disposal, retries, etc. **Enter RxJS.**

HELLO RXJS

RxJS is a library that directly targets problems of an asynchronous nature. Originating from the Reactive Extensions project, it brings the best concepts from the Observer pattern and functional programming together. The Observer pattern is used to provide a proven design based on *Producers* (the creators of event) and *Consumers* (the observers listening for said events), which offers a nice separation of concerns.

Moreover, functional programming concepts such as declarative programming, immutable data structures, and fluent method chaining—to name a few—enable you to write very expressive and easy to reason about code (bye-bye, callbacks).

For an overview of functional programming concepts, please read the Functional Programming in JavaScript Refcard.

If you're familiar with functional programming in JavaScript, you can think of RxJS as the "Underscore.js of asynchronous programming."

UNDERSTANDING STREAMS

RxJS introduces an overarching data type called the *stream*. Streams are nothing more than a sequence of events over time. Streams can be used to process any type of event such as mouse clicks, key presses, bits of network data, etc. You can think of streams as variables with the ability to react to changes emitted from the data they point to.

Variables and streams are both dynamic, but behave a bit differently; in order to understand this, let's look at a simple example. Consider the following simple arithmetic:

```
var a = 2;
var b = 4;
var c = a + b;
console.log(c); //-> 6

a = 10;  // reassign a
console.log(c); //-> still 6
```

Even though variable `a` changed to 10, the values of the other dependent variables remain the same and do not propagate through—by design. This is where the main difference is. A change in an event always gets propagated from the source of the event (producers) down to any parts that are listening (consumers). Hypothetically speaking, if these variables were to behave like streams, the following would occur:

```
var A$ = 2;
var B$ = 4;
var C$ = A$ + B$;
console.log(C$); //-> 6

A$ = 10;
console.log(C$); //->  16
```

As you can see, streams allow you to specify the dynamic behavior of a value declaratively (As a convention, I like to use the `$` symbol in front of stream variables). In other words, `C$` is a specification that concatenates (or adds) the values of streams `A$` and `B$`. As soon as a new value is pushed onto `A$`, `C$` immediately reacts to the change printing the new value 16. Now, this is a very contrived example and far from actual syntax, but it serves to explain how programming with streams differs from variables.

Now let's begin learning some RxJS.

THE OBSERVABLE

Perhaps the most important part of the RxJS library is the declaration of the Observable type. Observables are used to wrap a piece of data (button clicks, key presses, mouse movement, numbers, strings, or arrays) and decorate it with stream-like qualities. The simplest observable you can create is one with a single value, for instance:

```
var streamA$ = Rx.Observable.of(2);
```

Let's revisit the example above, this time using real RxJS syntax. I'll show you some new APIs that I'll talk about more in a bit:

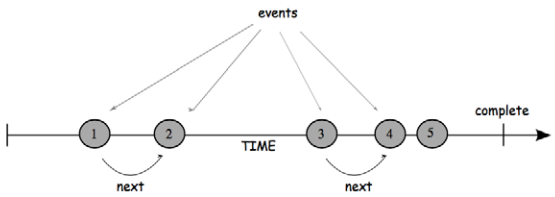
```
const streamA$ = Rx.Observable.of(2);
const streamB$ = Rx.Observable.of(4);
const streamC$ = Rx.Observable.concat(streamA$, streamB$)
  .reduce((x, y) => x  + y);

streamC$.subscribe(console.log); // prints 6
```

Running this example prints the value 6. Unlike the pseudo code above, I can't really reassign the value of a stream after its been declared because that would just create a new stream–it's an immutable data type. As a convention, since it's all immutable I can safely use the ES6 `const` keyword to make my code even more robust.

In order to push new values, you will need to modify the declaration of `streamA$`:

```
const streamA$ = Rx.Observable.of(2, 10)
...
streamC$.subscribe(console.log); // prints 16
```

Now, subscribing to `streamC$` would generate 16. Like I mentioned before, streams are just sequences of events distributed over time. This is often visualized using a marble diagram:



CREATING OBSERVABLES

Observables can be created from a variety of different sources. Here are the more common ones to use:

SOURCE	OBSERVABLE (STATIC) METHOD
of(arg)	Converts arguments to an observable sequence
from(iterable)	Converts arguments to an observable sequence
fromPromise(promise)	Converts a Promises/A+ spec-compliant Promise and/or ES2015-compliant Promise to an observable sequence
fromEvent(element, eventName)	Creates an observable sequence by adding an event listener to the matching DOMElement, jQuery element, Zepto Element, Angular element, Ember.js element, or EventEmitter

The other noticeable difference when programming with streams is the subscription step. Observables are lazy data types, which means that nothing will actually run (no events emitted, for that matter) until a subscriber `streamC$.subscribe(...)` is attached. This subscription mechanism is handled by `Observer`.

THE OBSERVER

Observers represent the consumer side of the model and are in charge of reacting to values produced or emitted by the corresponding Observables. The Observer is a very simple API based on the Iterator pattern that defines a `next()` function. This function is called on every event that's pushed onto an observable. Behind the scenes, the shorthand subscription above `streamC$.subscribe(console.log)` creates an Observer object behind the scenes that looks like this:

```
const observer = Rx.Observer.create(
    function next(val) {
        console.log(val);
    },
    function error(err) {
        ; // fired in case an exception occurs
    },
    function complete() {
        ; // fired when all events have been processed
    }
);
```

Observers also specify an API to handle any errors, as well as a means to signal that all the events have been processed. All of the Observer methods are optional, you can subscribe to an observable with just `.subscribe()`, but the most common approach is to at least provide a single function which will be mapped to `next()`. Within the subscription is where you would typically perform any effectful computations like writing to a file, logging to the console, appending to the DOM, or whatever tasks you're required you to do.

THE SUBSCRIPTION

Subscribing to an Observable returns a Subscription object, which you can you use to *unsubscribe* or dispose of the stream at ant point in time. This mechanism is really nice because it fills in the gap of the native JavaScript system, where cancelling events and disposing of them correctly has always been problematic.

To show this, I'll create an observable to listen for all click events:

```
const mouseClicks = Rx.Observable.fromEvent(document, 'click');
const subscription = mouseClicks.subscribe(...);
subscription.unsubscribe();
```

Obviously, this represents an infinite stream of clicks (the completed

signal will never actually fire). If I want to stop listening for events, I can simply call the `unsubscribe()` method on the Subscription instance returned from the Observable. This will also properly clean up and dispose of any event handlers and temporary objects created.

Now that you know how to create and destroy streams, let's take a look at a how to use them to support any problem domain you're tackling. I'll stick to using numbers as my domain model to illustrate these APIs, but of course you can use them to supply the business logic of any domain.

SEQUENCING WITH STREAMS

At the heart of RxJS is to provide a *unified programming model* to handle any type of data, whether it's synchronous (like an array) or asynchronous (remote HTTP call). RxJS uses a simple, familiar API based on the functional programming extensions added to JavaScript arrays (known as the Array#extras) with functions: `map`, `filter`, and `reduce`.

NAME	DESCRIPTION
map(fn)	Projects each element of an observable sequence into a new form
filter(predicate)	Filters the elements of an observable sequence based on a predicate
reduce(accumulator, [seed])	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

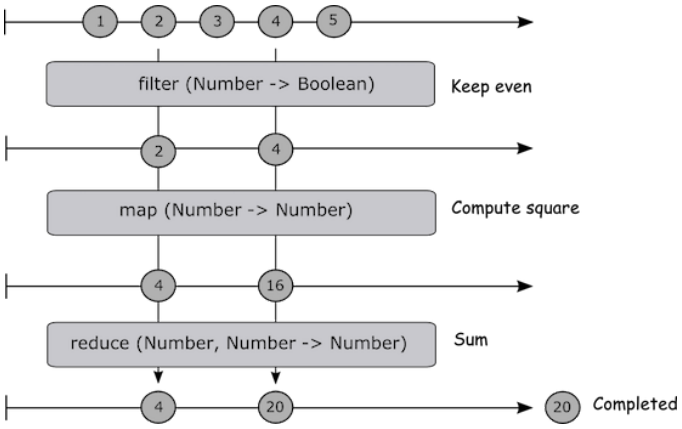
I'll begin with arrays. Given an array of numbers from 1 – 5, I will filter out odd numbers, compute their square, and sum them. Using traditional imperative code, this will require the use of at least a loop and a conditional statement. Using the functional Array methods I get:

```
var arr = [1, 2, 3, 4, 5];
var result =  arr
     .filter(x => (x % 2) === 0)
     .map(x => x * x)
     .reduce((a, b) => a + b, 0);
console.log(result);  //-> 20
```

Now, with very minor changes, this can work with Observable instance methods just as easily. Just like with arrays, the operators called on the Observable receive data from its preceding operator. The goal is to transform the input and apply business logic as needed within the realms of the Observable operators.

```
Rx.Observable.from(arr)
     .filter(x => (x % 2) === 0)
     .map(x => x * x)
     .reduce((a, b) => a + b)
     .subscribe(console.log); //-> 20
```

You can visualize what's happening behind the scenes with this diagram:



Arrays are predictable streams because they are all in memory at the time of consumption. Regardless, if these numbers were computed as the result of an HTTP call (perhaps wrapped in a Promise), the same code would still hold:

```
Rx.Observable.fromPromise(makeHttpCall('/numbers'))
     .filter(x => (x % 2) === 0)
     .map(x => x * x)
     .reduce((a, b) => a + b)
     .subscribe(console.log); //-> 20
```

Alternatively, of course, I can create independent side-effect-free functions that can be injected into the Observable sequence. This allows your code to look even more declarative:

```
const even = x => (x % 2) === 0;
const square = x => x * x;       // represent your business logic
const sum = (a, b) => a + b;

Rx.Observable.fromPromise(makeHttpCall('/numbers'))
     .filter(even)
     .map(square)
     .reduce(sum)
     .subscribe(console.log); //-> 20
```

This is the beauty of RxJS: a single programming model can support all cases. Additionally, observables allow you to sequence and chain operators together very fluently while abstracting the problem of latency and wait-time away from your code. Notice that this way of writing code also eliminates the complexity involved in looping and conditional statements in favor of higher-order functions.

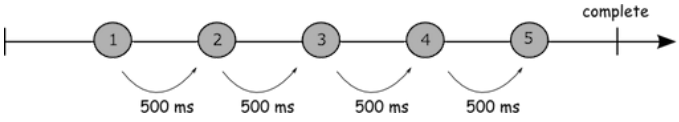
DEALING WITH TIME

Understanding how RxJS effectively deals with the time and latency can be done via its time-based operators. Here's a short list of the most common ones used to create Observables with time in them:

NAME	DESCRIPTION
Rx.Observable.interval(period)	Returns an observable sequence that produces a value after each period
Rx.Observable.timer(dueTime)	Returns an observable sequence that produces a value after dueTime has elapsed and then after each period

These are very nice for simulating timed events:

```
const source$ = Rx.Observable.interval(500)
  .take(5)
  .map(num => ++num);

source$.subscribe(
  (num) => console.log(num),
  (err) => console.err('Error:  ${err}'),
  ()    => console.log('Completed'));
```



Using `interval(500)` will emit values every half a second. Because this is an infinite stream, I'm taking only the first 5 events, converting the Observable into a finite stream that will actually send a completed signal.

```
1  // separated by 500 ms
2
3
4
5
"Completed"
```

HANDLING USER INPUT

You can also use Observables to interact with the DOM. Using `Rx.Observable.fromEvent` I can listen for any DOM events like mouse clicks, key presses, input changes, etc. Here's a quick example:

```
const link = document.querySelector('#go');

const extract = (event, attr) => event.currentTarget.getAttribute(attr);

const clickStream$ = Rx.Observable
  .fromEvent(link, 'click')
  .map(event => extract(event, 'href'));

clickStream$.subscribe(
  href => {
    if(href) {
       if(confirm('Navigate away from this page?')) {
          location.assign(href);
       }
    }
  }
);
```

I can handle clicks in this case and perform any action within the observer. The `map` operator is used here to transform the incoming click event, extract the underlying element, and read its `href` attribute.

HANDLING ASYNCHRONOUS CALLS

Handling user input is not the only type of asynchronous actions you can work with. RxJS also nicely integrates with the ES6 Promise API to fetch remote data. Suppose I need to fetch users from GitHub and extract their login names. The power of RxJS lets me do all of this with just 5 lines of code by taking advantage of lambda expressions.

```
Rx.Observable.of('http://api.github.com/users')
  .flatMap(url => Rx.Observable.fromPromise(makeHttpCall(url)))
  .concatMap(response => Rx.Observable.from(response))
  .map(user => user.login)
  .subscribe(console.log);
```

This code introduces a couple of new artifacts, which I'll explain. First of all, I start with an Observable wrapping GitHub's `users` REST API. I `flatMap` the `makeHttpCall` function over that URL, which returns a promisified AJAX call. At this point, RxJS will attempt to resolve the promise and wait for its resolution. Upon completion, the response (an array containing user objects) from GitHub is mapped to a function that wraps the single array output back into an Observable, so that I can continue to apply further operations on that data. Lastly, I map a simple function to extract the `login` attribute over that data.

MAP VS FLATMAP

As I said before, the `map` function on Observables applies a function onto the data within it, and returns a new Observable containing that result. The function you call can return any type—even another Observable. In the example above, I pass in a lambda expression that takes a URL and returns an Observable made from the result of a promise:

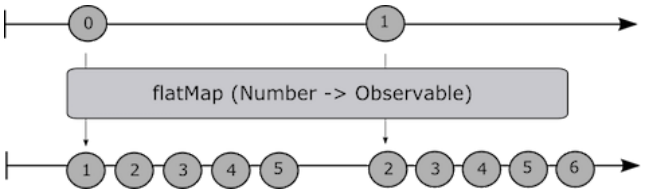
```
url => Rx.Observable.fromPromise(makeHttpCall(url))
```

Mapping this function would yield an observable of observables (this is a very common scenario in functional programming when working with data types called Monads). What we need is to be able to map the function and flatten or join the result back into a single Observable—like peeling back an onion. This is where `flatMap` comes in. The function above returns an `Rx.Observable.fromPromise(...)`, so I need to flatten this result. As a general rule of thumb, use `flatMap` when you project an existing observable onto another. Let's look at a simpler example that's a bit easier to understand:

```
Rx.Observable
    .interval(500)
    .flatMap(function (x) {
        return Rx.Observable.range(x + 1, 5);
    })
    .subscribe(console.log);
```

This code will generate a window of 5 consecutive numbers every half a second. First, it will print numbers 1-5, then 2-6, 3-7, and so on.

This can be represented visually like this:



DISPOSING OF AN OBSERVABLE SEQUENCE

Recall earlier, I mentioned that one of the main benefits of RxJS's abstraction over JavaScript's event system is the ability to dispose or cancel events. This responsibility lies with the Observer, which gives you the opportunity to perform your own cleanup. This is done via the Subscription instance you obtain by subscribing to the Observable.

```
const source$ = Rx.Observable.create(observer => {
    var num = 0;
    const id = setInterval(() => {
      observer.next('Next ${num++}');
    }, 1000);

    return () => {
       clearInterval(id);  // disposal handler
    }
});

const subscription = source$.subscribe(console.log);

setTimeout(function () {
    subscription.unsubscribe();
}, 7000);
```

This code creates a simple Observable. But this time, instead of wrapping over a data source such as an event or an AJAX call, I decided to create my own custom event that emits numbers in one second intervals indefinitely. Providing my own custom event also allows me to define its disposal routine, which is done by the function returned to the subscription.

RxJS maps this action to the `Subscription.unsubcribe()` method. In this case, my cleanup action consists of just clearing the `interval` function. Instead of printing numbers indefinitely, after 7 seconds, I dispose of the Observable, causing the interval to cease emitting new numbers.

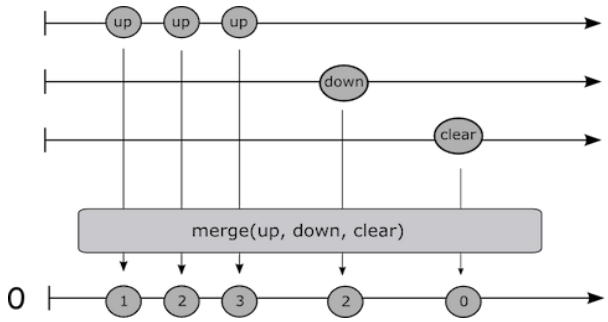
COMBINING STREAMS

While Observables might seem heavyweight, they're actually very cheap to create and dispose of. Just like variables, they can be combined, added, ANDed together, etc. Let's begin with merging observables.

MERGING MULTIPLE STREAMS

The merge method combines any Observable sequences into a single one. What's impressive about this operator is that event sequences emitted over time by multiple streams are combined in the correct order. For instance, consider a simple HTML widget with three buttons to perform three actions on a counter: up, down, and clear.

```
// buttons
const upButton = document.querySelector('#up');
const downButton = document.querySelector('#down');
const clearButton = document.querySelector('#clear');

// counter
const total = document.querySelector('#total');

// helper function used to create a click stream from an HTML element
const buttonStreamWith = (elem, val) =>
   Rx.Observable
    .fromEvent(elem, 'click')
    .map(() => val);

// wrap streams over the buttons
const up$ = buttonStreamWith(upButton, 1);
const down$ = buttonStreamWith(downButton, -1);
const clear$ = buttonStreamWith(clearButton, null);

// subscribe to all and update counter accordingly
Rx.Observable.merge(up$, down$, clear$)
  .subscribe(
    value => {
      if(!value) {
        total.innerHTML = 0;
      }
      else {
        var current = parseInt(total.innerHTML);
        total.innerHTML  = current + value;
      }
    }
);
```



Other means of combining streams can be done via the `concat()` and `concatAll()` operators.

COMBINING ONE STREAM WITH ANOTHER

The `withLatestFrom` operator is very useful because it allows you to merge an observable sequence into another by using a selector function, only when the source observable sequence produces an element. To show this, suppose I want to print out the list of GitHub users, one every second. Intuitively, I need to combine a time-based stream with an HTTP stream.

```
const request$ =
    Rx.Observable.of('http://api.github.com/users');

const response$ = request$
  .flatMap(url => Rx.Observable.fromPromise(makeHttpCall(url)));

Rx.Observable.interval(1000)
  .withLatestFrom(response$, (i, users) => users[i])
  .subscribe(user => console.log(user.login));
```

BUFFERING

As I mentioned earlier, streams are stateless data structures, which means state is never held within them but is immediately flushed from the producers to the consumers. However, once in a while it's important to be able to temporarily store some data and be able to make decisions based on it. One example that comes to mind is tracking double-clicks on an element. How can you detect a second click action without storing the first one? For this, we can use buffering. There are multiple cases:

BUFFERING FOR A CERTAIN AMOUNT OF TIME

You can temporarily hold a fixed amount of data into internal arrays that get emitted as a whole once the count threshold is met.

```
Rx.Observable.range(1, 9)
  .bufferCount(3)
  .subscribe(console.log);
  //-> prints [1, 2, 3]
  //->        [4, 5, 6]
  //->        [7, 8, 9]
```

BUFFERING DATA BASED ON TIME

You can also buffer for a predefined period of time. To show this I'll create a simple function that simulates sending emails every second from a set of available email addresses. If I send emails every second, and buffer for, say, 5 seconds, then buffering will emit a group of emails once the buffered time has elapsed:

```
// email users
const addresses = [
     "erik.meijer@dzone.com",
     "matthew.podwysocki@dzone.com",
     "paul.daniels@dzone.com",
     "igor.oleinikov@dzone.com",
     "tavis.rudd@dzone.com",
     "david.driscoll@dzone.com"
    ];

// simulates the arrival of emails every second
// wrapped in an observable
const sendEmails = addresses => {
  return Rx.Observable.interval(1000).map(i => addresses[i]);
};

sendEmails(addresses)
  .buffer(Rx.Observable.timer(5000))
  .forEach(group => console.log('Received emails from: ' + group));
  //-> prints
  Received emails from: erik.meijer@dzone.com,
                        matthew.podwysocki@dzone.com,
                        paul.daniels@dzone.com,
                        igor.oleinikov@dzone.com
```

ERROR HANDLING

Up until now, you've learned different types of asynchronous operations on streams, whether they be on DOM events or fetching remote data. But none of the examples so far have shown you what happens when there's an error or an exception in the stream pipeline. DOM events are very easy to work with because they won't actually throw errors. The same can't be said about AJAX calls. When not dealing with simple arrays, streams are actually highly unpredictable and you must be prepared for the worst.

With errors, if you are passing your own observer, you need to try `catch` inside and call `observer.onError(error)`; This will allow you to catch the error , handle it, and also dispose.

Alternatively, you can use `.onErrorResumeNext`.

CATCH

The good news is that you can continue to use a good 'ol `catch` block (now an operator) just like you're used to. To show this I'll artificially create an error as soon as a stream sees the value 5.

```
Rx.Observable.of(1, 2, 3, 4, 5)
  .map(num => {
        if(num === 5) {
          throw 'I hate 5!';
        }
        return num;
      })
  .subscribe(
    x => console.log(x),
    error => console.log('Error! ' + error)
  );
  // prints 1
  //        2
  //        3
  //        4
  //        "Error! I hate 5!"
```

As soon as the condition is met, the exception is thrown and propagated all the way down to the Observer subscribed to this stream. You might want to gracefully catch the exception and provide a friendly message:

```
Rx.Observable.of(1, 2, 3, 4, 5)
  .map(num => {
        if(num === 5) {
          throw 'I hate 5!';
        }
        return num;
      })
  .catch(err => Rx.Observable.of('Sorry. ${err}'))
  .subscribe(
    x => console.log(x),
    error => console.log('Error! ' + error)
  );
  // prints 1
  //        2
  //        3
  //        4
  //        "Sorry. I hate 5!"
```

The `catch` operator allows you to handle the error so that it doesn't get propagated down to any observers attached to this stream. This operator expects another Observable to carry the baton forward, so you can use this to suggest some default value in case of errors. Notice that, this time, the error function on the observer never actually fired!

Now, if we do want to signal an unrecoverable condition, you can catch and throw the error. Within the catch block, this code will actually cause the exception to fire. I would caution against this as throwing exceptions is a side effect that will ripple through and is expected to be handled somewhere else. This should be used sparingly in truly critical conditions.

```
...
.catch(() => Rx.Observable.throw('System error: Can't go any
  further!'))
```

Another option is to attempt a `retry`.

RETRIES

With observables, you can retry the previous operation for a certain number of times, before the failure is fired.

```
Rx.Observable.of(1, 2, 3, 4, 5)
  .map(num => {
        if(num % 2 === 0) {
          throw 'I hate even numbers!'
        }
        return num;
      })
  .retry(3)
  .catch(err => Rx.Observable.throw('After 3 attempts. Fail!'))
  .subscribe(
    x => console.log('Found ' + x),
    error => console.log('Error! ' + error)
  );
```

.FINALLY()

As JavaScript developers, our code deals with many events or asynchronous computations all the time. This can get complicated quickly as we build comprehensive UIs or state-machines that need to react and keep responsive in the face of failures. RxJS truly embodies two of the most important principles of the Reactive Manifesto, which are Responsive and Resilient.

Moreover, RxJS makes these computations first-class citizens of the language and offers a state of the art event system for JavaScript. This provides a unified computing model that allows for readable and composable APIs to deal with these asynchronous computations, abstracting out the nitty gritty details of latency and wait time.

ADDITIONAL RESOURCES

- **RxMarbles**—Interactive diagrams of Rx Observables
- **ReactiveX**—An API for Asynchronous Programming With Observable Streams
- **Callback Hell**—A Guide to Writing Asynchronous JavaScript Programs
- **RxJS GitBook**
- **JS Promise Documentation on Mozilla Developer Network**
- **The Introduction to Reactive Programming You've Been Missing** by André Staltz
- **RxJS Design Guidelines**

ABOUT THE AUTHOR

LUIS ATENCIO (@luijar) is a Staff Software Engineer for Citrix Systems in Ft. Lauderdale, FL. He has a B.S. and an M.S. in Computer Science. He works full time developing and architecting web applications leveraging both Java, PHP, and JavaScript platforms. Luis is also very involved in the community and has presented on several occasions at conferences and local meet-ups. When he is not coding, Luis writes a developer blog at luisatencio.net focused on software engineering as well as several magazine articles for PHPArch magazine and DZone Refcards. Luis is also the author of *Functional Programming in JavaScript* (manning.com/atencio), *Functional PHP* (leanpub.com/functional-php), as well as co-author for *RxJS in Action* (Manning 2016).

RECOMMENDED BOOK RxJS is deeply inspired by the principles of functional programming. *Functional Programming in JavaScript* teaches JS developers functional techniques that will improve extensibility, modularity, reusability, testability, and performance. Through concrete examples and jargon-free explanations, this book shows you how to apply functional programming to real-life JavaScript development tasks. The book includes insightful comparisons to object-oriented or imperative programming, allowing you to ease into functional design. Moreover, you'll learn a repertoire of techniques and design patterns including function chaining and pipelining, recursion, currying, binding, functional composition, lazy evaluation, fluent error handling, memoization, and much more. By the end of the book, you'll think about application design in a fresh new way.

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513

888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795
9 781936 502776

VERSION 1.0