RxJS 6
Reactive Programming

Rujuan Xing

Maharishi University of Management - Fairfield, Iowa

Why do we need Reactive Programming?

- JS is lack of threads
- Asynchronous code using callbacks, promises and events, they all have drawbacks:
- Callback Functions: a function A passed as a parameter to another function B that performs an asynchronous operations
  - Callback hell
  - Callbacks can run more than once
  - Callbacks change error semantics – break traditional `try/catch` mechanism
  - Concurrency gets increasing complicated

Why do we need Reactive Programming? (cont.)

- Promise
  - Only ever yield a single value.
  - Useless for handling recurrent events such as mouse clicks, steams of data coming from server, etc
- Event Emitters
  - Events force side effects. – ignore return values
  - Events are not first-class values.
    - a series of click events can’t be passed as a parameter or manipulated as the sequence it actually is.
  - It’s easy to miss events if we start listening too late.

What’s Reactive?

- Spreadsheets Are Reactive
  - We have a value in cell B2. We can reference it in other cells.
  - Whenever we change B2, every cell depending on B2 will automatically update its own value

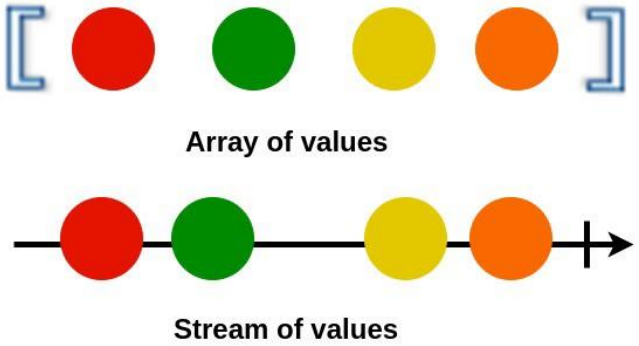
	A	B	C	D	E	F
1	Designation	Quantity	Price $	Total $	Tax rate:	12%
2	PC	1	500	500	Average price:	290
3	Monitor	2	250	500	Nb of items :	4
4	Desk	1	120	120	Avg price / items:	72
5			TOTAL $	1120		
6			VAT	134.4		
7						
8						

What is Reactive Extension and RxJS?

- Reactive Programming is an asynchronous programming paradigm concerned with data streams and the propagation of the change.
- Reactive Extensions or Rx
  - A reactive programming model originally created at Microsoft that allows developers to easily compose asynchronous stream of data.
  - Provides a common interface to combine and transform data from wildly different sources, such as filesystem operations, user interaction, etc.
- RxJS is a JavaScript implementation of the Reactive Extensions, or Rx.
  - A library for composing asynchronous and event-based programs using observable sequences.
  - Provides `Observable`, `Observer`, `Subjects` and operators like `map`, `filter`, `reduce` to allow handling asynchronous events as collections.
- Seeing your program as sequences of data is key to understanding reactive programming.

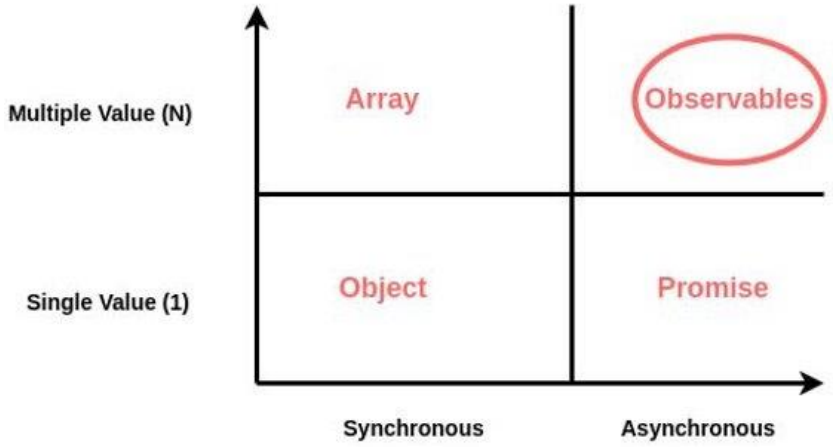
Stream

- A stream is a sequence of ongoing events ordered in time.
  - anything like user inputs, button clicks or data structures.
- Stream emit three things during its timeline:
  - a value, an error, and complete signal.
  - We have to catch this asynchronous event and execute functions accordingly.



Observables

- Observables are lazy Push collections of multiple values.
- An Observable is basically a function that can return a stream of values to an observer over time, this can either be synchronously or asynchronously. The data values returned can go from zero to an infinite range of values.



	SINGLE	MULTIPLE
Pull	Function	Iterator
Push	Promise	Observable

- In Pull systems, the Consumer determines when it receives data from the data Producer. The Producer itself is unaware of when the data will be delivered to the Consumer.
- In Push systems, the Producer determines when to send data to the Consumer. The Consumer is unaware of when it will receive that data. It’s like the saying, “Don’t call us; we’ll call you."

Observable example - create observable that emits click events

- HTML file

```
<input id="search" type="text" />
<script src="https://cdnjs.cloudflare.com/ajax/libs/rxjs/6.6.7/rxjs.umd.js"></script>
<script src="rxjs-fromevent.js"></script>
```

- JS file

```
const { fromEvent } = rxjs;

const node = document.getElementById("search");
//create observable that emits click events
const inputObs = fromEvent(node, 'input');

inputObs.subscribe({
    next: event => console.log(`You just typed ${event.target.value}!`),
    error: err => console.log(`Oops... ${err}`),
    complete: () => console.log(`Complete!`),
});
```

What is an Observer?

- An Observer is a consumer of values delivered by an Observable.
- Observers are simply a set of callbacks: `next`, `error`, and `complete`.

```
const observer = {
    next: x => console.log('Observer got a next value: ' + x),
    error: err => console.error('Observer got an error: ' + err),
    complete: () => console.log('Observer got a complete notification'),
};
```

- To use the Observer, provide it to the subscribe of an Observable:

```
observable.subscribe(observer);
```

Observable Lifecycle

- **Creation**

```
// Create an observable that emits 'Hello' and 'World'
const hello = Observable.create(function(observer) {
    observer.next('Hello');
    observer.next('World');
    observer.complete();
});
```

- **Subscription**

```
 hello.subscribe({
    next: x => console.log('Observer got a next value: ' + x),
    error: err => console.error('Observer got an error: ' + err),
    complete: () => console.log('Observer got a complete notification'),
});
```

- **Execution**: At least 1 Observer subscribe the Observable.

Observable Lifecycle (cont.) - Subscription

- A Subscription is an object that represents a disposable resource, usually the execution of an Observable.
  - Method `unsubscribe` - takes no argument
    - *release resources or cancel Observable executions.*

- **Lifecycle: Destruction – using Subscription**

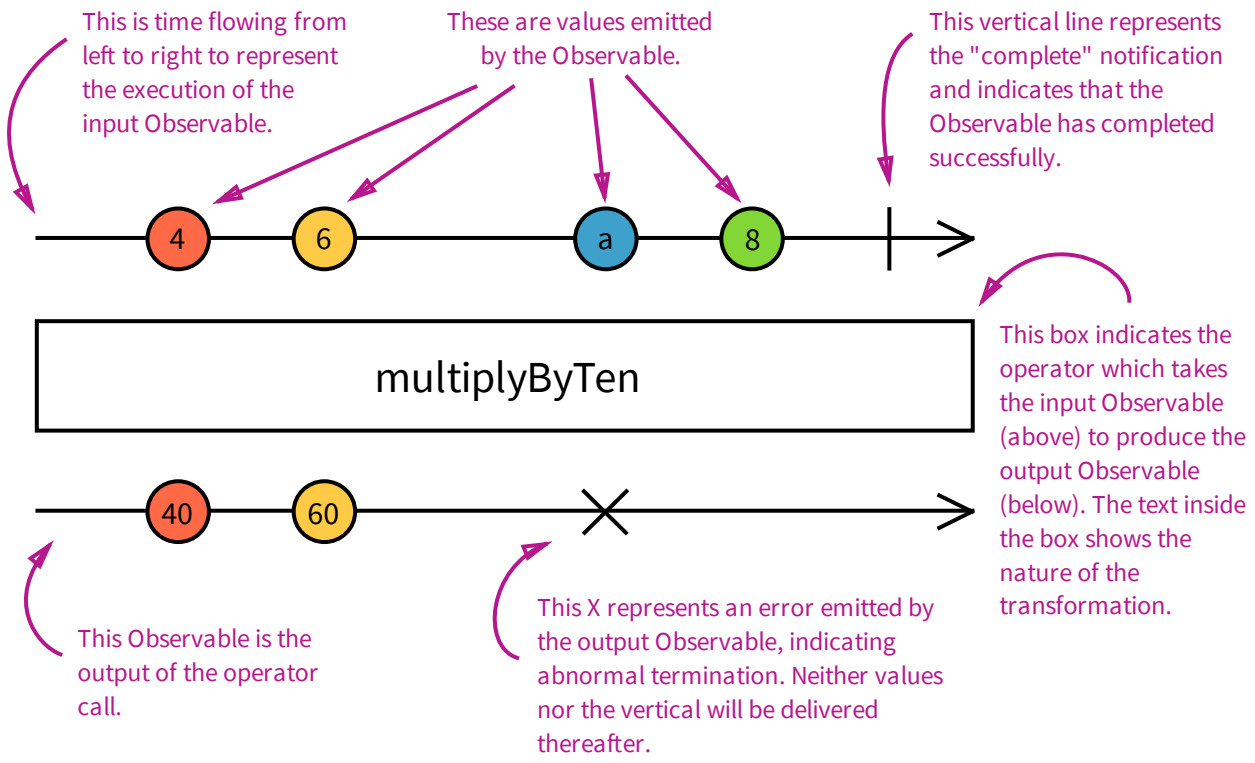
```
const { interval } = rxjs;

//emit value in sequence every 1 second
const observable = interval(1000);
//output: 0,1,2,3,4,5....
const subscription = observable.subscribe(x => console.log(x));

// This cancels the ongoing Observable execution which
// was started by calling subscribe with an Observer.
setTimeout(() => subscription.unsubscribe(), 5000);
```

Operators

- Operators are functions.
- Two kinds of operators:
  - Pipeable Operator
    - a function that takes an Observable as its input and returns another Observable
    - Can be piped to Observables using the syntax `observableInstance.pipe(operator())`
    - `filter(...), map(…), etc`
  - Creation Operators
    - can be used to create an Observable
    - `from(…), fromEvent(…), interval(…), create(…), etc`



Operators – filter, map

```
const { from } = rxjs;
const { map, filter } = rxjs.operators;

console.log('Start');
from([1, 2, 3])
    .pipe(
        map(n => n + 3),
        filter(n => n >= 5)
    )
    .subscribe(
        x => console.log(x),
        error => console.error(error),
        () => console.log('done')
    );
console.log('End');
```

Operators - reduce

```
const { of } = rxjs;

const { reduce } = rxjs.operators;


const source = of(1, 2, 3, 4);

const example = source.pipe(reduce((acc, val) => acc + val));

//output: Sum: 10'

const subscribe = example.subscribe(val => console.log('Sum:', val));
```

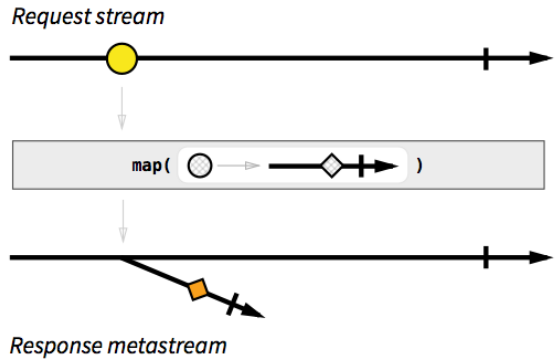
Higher-Order Observables

- Observables *of* Observables
  - For example, imagine you had an Observable emitting strings that were the URLs of files you wanted to see. The code might look like this:

```
const fileObservable = urlObservable.pipe(

    map(url => http.get(url)),

);
```



  - `http.get()` returns an Observable (of string or string arrays probably) for each individual URL. Now you have an Observables of Observables, a higher-order Observable.

mergeAll()

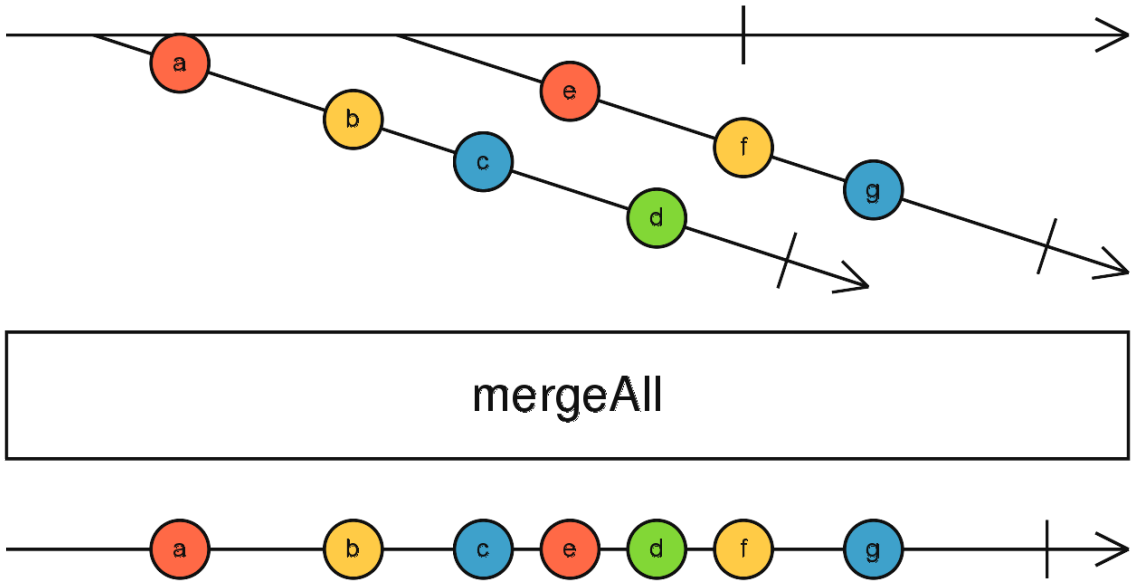
- *Flattens an Observable-of-Observables.*

```
const { of } = rxjs;
const { map, mergeAll } = rxjs.operators;

const myPromise = val =>
    new Promise(resolve =>
          setTimeout(() => resolve(`Result: ${val}`)
          2000));
//emit 1,2,3
const source = of(1, 2, 3);

const example = source.pipe(
    //map each value to promise
    map(val => myPromise(val)),
    //emit result from source
    mergeAll()
);

const subscribe = example.subscribe(val => console.log(val));
```



Subject

- A special type of Observable that allows values to be multicasted to many Observers.
- *Subjects are like EventEmitters: they maintain a registry of many listeners.*
- Internally to the Subject, `subscribe` does not invoke a new execution that delivers values. It simply registers the given Observer in a list of Observers, similarly to how `addListener` usually works in other libraries and languages.

```
const { Subject } = rxjs;

const subject = new Subject();

subject.subscribe({
    next: (v) => console.log(`observerA: ${v}`)
});
subject.subscribe({
    next: (v) => console.log(`observerB: ${v}`)
});

subject.next(1);
subject.next(2);
```

BehaviorSubject

- Stores the latest value emitted to its consumers, and whenever a new Observer subscribes, it will immediately receive the "current value" from the `BehaviorSubject`.

```
const { BehaviorSubject } = rxjs;;
const subject = new BehaviorSubject(0); // 0 is the initial value

subject.subscribe({
    next: (v) => console.log(`observerA: ${v}`)
});

subject.next(1);
subject.next(2);

subject.subscribe({
    next: (v) => console.log(`observerB: ${v}`)
});

subject.next(3);
```

References

- https://rxjs-dev.firebaseapp.com/guide/overview
- https://www.learnrxjs.io/