

RxJS

Reactive Programming

CS445 Modern Asynchronous Programming

Maharishi University of Management

Department of Computer Science

Teaching Faculty: Assistant Professor Umur Inan

Prepared by: Associate Professor Asaad Saad

Reactive Programming

Reactive programming is programming with **asynchronous data streams**.



<http://reactivex.io/>

The ReactiveX Observable model allows you to treat streams of asynchronous events with the same **sort of simple, composable operations** that you use for **collections of data** items like arrays. It frees you from **tangled webs of callbacks**, and thereby makes your code more readable and less prone to bugs.

To start using RxJS library in Browser: `<script src="https://cdnjs.cloudflare.com/ajax/libs/rxjs/6.6.7/rxjs.umd.min.js"></script>`

Everything can be a stream

Streams are **cheap**, We can create data streams of **anything**: values, events, user inputs, properties, arrays, caches, data structures, etc.

You can **listen to the stream** and **react** accordingly.

Wildly used in Angular and React.



Rx* library family is widely available for most languages and platforms (.NET, Java, Scala, Clojure, JavaScript, Ruby, Python, C++, Objective-C/Cocoa, Groovy, etc).

Why Reactive Programming?

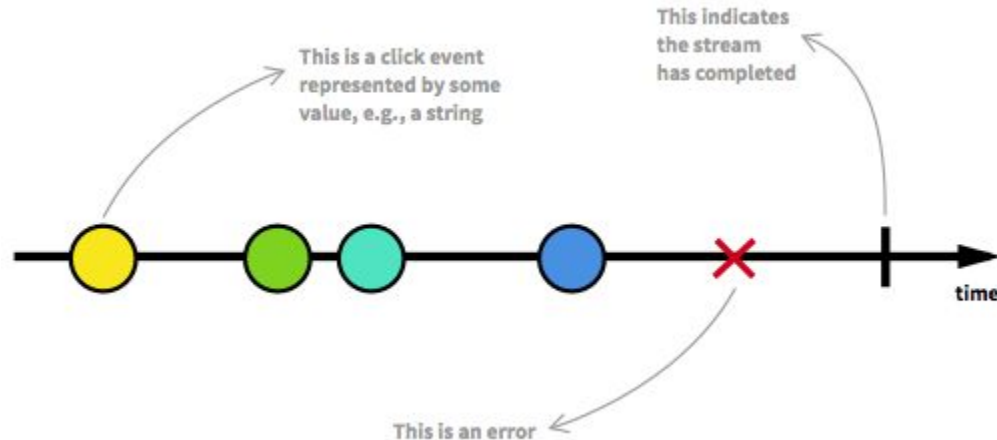
Reactive Programming **raises** the level of abstraction of your code so you can focus on the events that define the **business logic**, rather than having to constantly work with a **large amount of implementation details**.

The benefit is more evident in modern web apps and mobile apps that are highly interactive with a multitude of UI events related to data events.

Apps nowadays have an abundance of real-time events of every kind that enable a highly interactive experience to the user. We need tools for properly dealing with that.

Streams

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.



How Observables work?

We capture those emitted events 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.

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**.

Intro to Observables

Observables are **LAZY event streams** which can emit zero or more events, and may or may not finish.

Some key differences between promises and observable are:

- Observables handle **multiple values over time**
- Observables can be **cancelled**
- Observables can be **retried**

Creating Simple Observable Example

```
const { Observable } = rxjs;

const obs$ = Observable.create(function (observer) {
  observer.next('CS445');
  setTimeout(() => { observer.complete(); }, 3000);
});

const subscription = obs$.subscribe(
  function (x) { console.log(`Value: ${x}`); },
  function (err) { console.error(err); },
  function () { console.info(`Done`); }
);
```

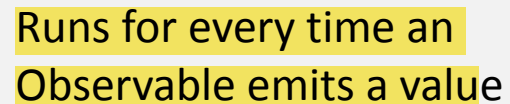
The **subscribe** operator is the glue that connects an observer to an observable. In order for an observer to see the items being emitted by an observable, it must first subscribe to that observable.

Example

```
var bar = Observable.create(function (observer) {  
  observer.next('Hello');  
  setTimeout(function () {  
    observer.next('world'); }, 1000);  
});
```

```
console.log('before');  
bar.subscribe(function (x) { console.log(x); });  
console.log('after');
```

```
// before  
// Hello  
// after  
// World
```



Runs for every time an
Observable emits a value

Converting Values to Observables

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

```
console.log('Start')  
of(1, 2, 3)  
  .pipe(  
    map(n => n + 3),  
    filter(n => n >= 5)  
  )  
  .subscribe(  
    x => console.log(x),  
    null,  
    () => console.log('done')  
  )  
console.log('End')
```

Converting Data Sets to Observables

```
const { from } = rxjs;
const { map, filter } = rxjs.operators;
const data = [
  { id: 0, name: 'Learning Promises' },
  { id: 1, name: 'Learning Async/Await' },
  { id: 2, name: 'Learning Observables' } ]

from(data)
  .pipe(map(obj => ({ msg: `${obj.name} is awesome!` })))
  .subscribe(obj => console.log(obj.msg))
```

Converting Events to Observables

```
const { fromEvent } = rxjs;  
const obs$ = fromEvent(document, 'click');  
  
obs$.subscribe(e => console.log(e));  
obs$.subscribe(e => console.log(e.target));
```

Converting Promise to Observables

```
const { from } = rxjs;
```

```
let myPromise = new Promise((resolve, reject) => {  
    setTimeout(function () { resolve("Success!"); }, 2000);  
});
```

```
// myPromise.then(e => console.log(e))  
from(myPromise).subscribe(e => console.log(e));
```

Subject

```
const { Subject } = rxjs;

var myObservable$ = new Subject();

myObservable$.subscribe(
  value => console.log(value)
);

setTimeout(() => { myObservable$.next('CS445'); }, 3000);
```

Behavior Subject

```
<button>Click me</button> <div></div>
```

```
<script>
```

```
  const button = document.querySelector('button');
```

```
  const div = document.querySelector('div');
```

```
  const { Subject, BehaviorSubject } = rxjs;
```

```
  const myObservable$ = new BehaviorSubject('Not clicked');
```

```
  // var myObservable$ = new Subject();
```

```
  myObservable$.subscribe(value => div.textContent = value);
```

```
  button.addEventListener('click', () =>
```

```
    myObservable$.next('Clicked!'));
```

```
</script>
```

Observable Operators

```
const input = fromEvent(document.querySelector('input'), 'input');
```

Filter out target values less than 3 characters long

```
input.filter(event => event.target.value.length > 2)
```

```
.map(event => event.target.value)
```

```
.subscribe(value => console.log(value));
```

Delay the events

```
input.delay(1000)
```

```
.map(event => event.target.value)
```

```
.subscribe(value => console.log(value));
```

*filter & map should be inside
RCE*

debounceTime()

```
<input/>
```

```
<script>
```

```
  const { fromEvent } = rxjs;
```

```
  const { map, debounceTime } = rxjs.operators;
```

```
  const input$ = fromEvent(document.querySelector('input'), 'input');
```

```
  input$
```

```
    .pipe( // When silence of 200ms => Emit
```

```
      debounceTime(200),
```

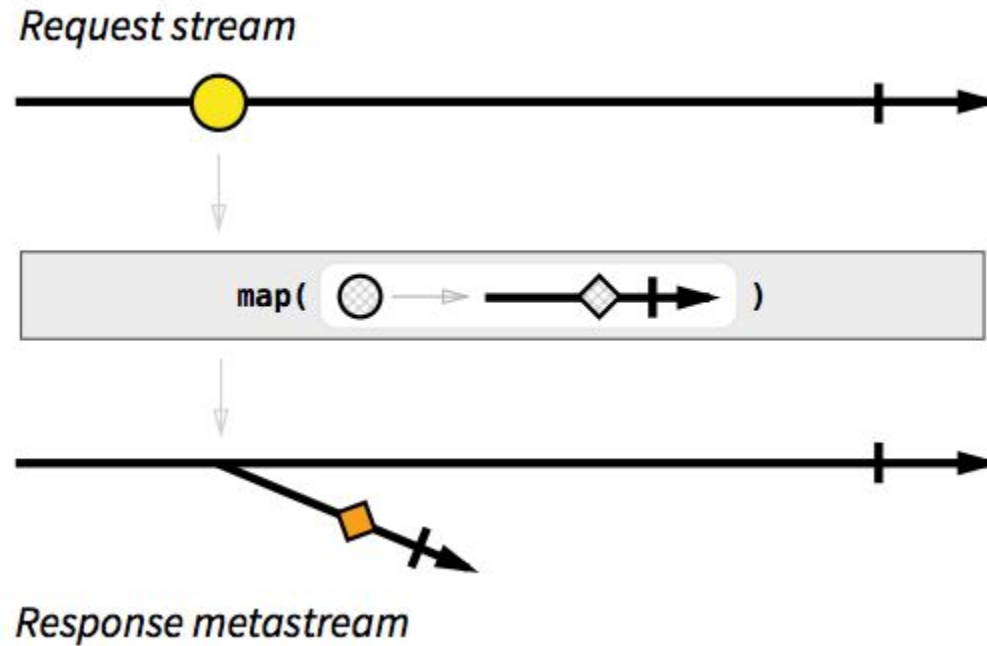
```
      map(event => event.target.value),
```

```
    ).subscribe(value => console.log(`Check if user exists: ${value}`));
```

```
</script>
```

Higher-Order Observable

A higher-order observable is a stream that returns another stream.



mergeAll()

It subscribes to an Observable that emits Observables (higher-order Observable). Each time it observes one of these emitted inner Observables, it subscribes to it and delivers all the values from the inner Observable on the main Observable.

mergeAll()

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

```
interval(100) // ms  
  .pipe(  
    take(10),  
    map(x => of(1, 2, 3)),  
    mergeAll()  
  )  
  .subscribe(console.log);
```

flatMap()

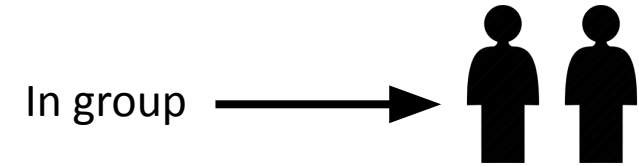
The `flatMap` operator is useful when you have an Observable that emits a series of items that themselves have Observable members or are in other ways transformable into Observables, so that you can create a new Observable that emits the complete collection of items emitted by the sub-Observables of these items.

flatMap()

```
const { interval } = rxjs;
const { map, flatMap, mergeAll, take } = rxjs.operators;

interval(100) // ms
  .pipe(
    take(10),
    // map(x => of(1, 2, 3)),
    // mergeAll()
    flatMap(x => of(1, 2, 3)),
  )
  .subscribe(console.log);
```

Exercise



1. Define a **removeNum** function that will work asynchronously on every **Array** object.
The function accepts one argument: a number.
The function returns a new array after removing all instances of the passed number.
2. Explain how does this function affect the event-loop?

```
console.log('Start');  
console.log([1, 3, 4, 2, 1, 5].removeNum(1));  
console.log('Finish');
```

```
Start  
Finish  
[3, 4, 2, 5]
```

Exercise

Write a method **removeDuplicates()** to remove all the duplicate numbers from an array.

```
[4,1,5,7,2,3,1,4,6,5,2].removeDuplicates(); // [4, 1, 5, 7, 2, 3, 6]
```

Solve the exercise above in 3 different ways using:

- Promises
- Async/Await
- Observables

Exercise





Create a webpage that displays the world cup results for a certain team.

Use Fetch API to send a call to

<https://raw.githubusercontent.com/lsv/fifa-worldcup-2018/master/data.json>

Create a service layer to work with this object (higher-order functions).

Create a form (use observables to listen to user input) to display full details about the team's group as following:

Group A					
	Played	Wins - Draws - Lost	Goals	Goal difference	Points
 Russia	2	2 - 0 - 0	8 - 1	7	6
 Uruguay	2	2 - 0 - 0	2 - 0	2	6
 Egypt	2	0 - 0 - 2	1 - 4	-3	0
 Saudi Arabia	2	0 - 0 - 2	0 - 6	-6	0

Exercise: Parsing JSON

Suppose we have a service <http://jsonplaceholder.typicode.com> about blogs.

- Write a page that processes this JSON blog data.
 - To display user information [/users/1](#)
 - Display all posts from selected user [/posts?userId=1](#)
 - Display all comments from selected post [/comments?postId=1](#)
- Create one page with input form to take userId from the browser
- Display user name and email and address and all posts belongs to this userId
- For every post, you need to show a button (show comments) once clicked you need to show all comments for the specific post.
- The trick is to hide postId in data- attribute in each post. (HTML5 compatible)
- To display the comments you need to create the output comment elements and append them to the DOM (faster, memory efficient)
- You may need to use `$(DOM).on()` to attach event handler with functions to newly created elements

```
$(staticAncestors).on(eventName, dynamicChild,  
function() {});  
$(document).on( eventName, selector, function(){} );
```