Asynchronous JavaScript

CS572 Modern Web Applications Programming

Maharishi University of Management

Department of Computer Science

Assistant Professor Asaad Saad

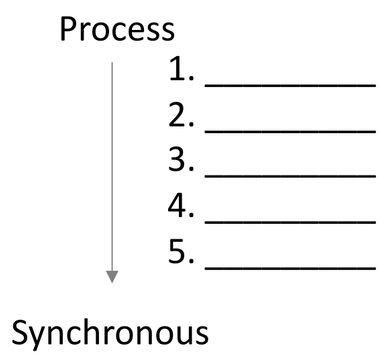
PREV NEXT

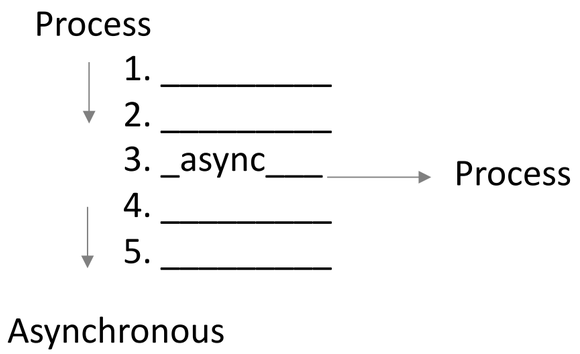
OUTLINE

Search...

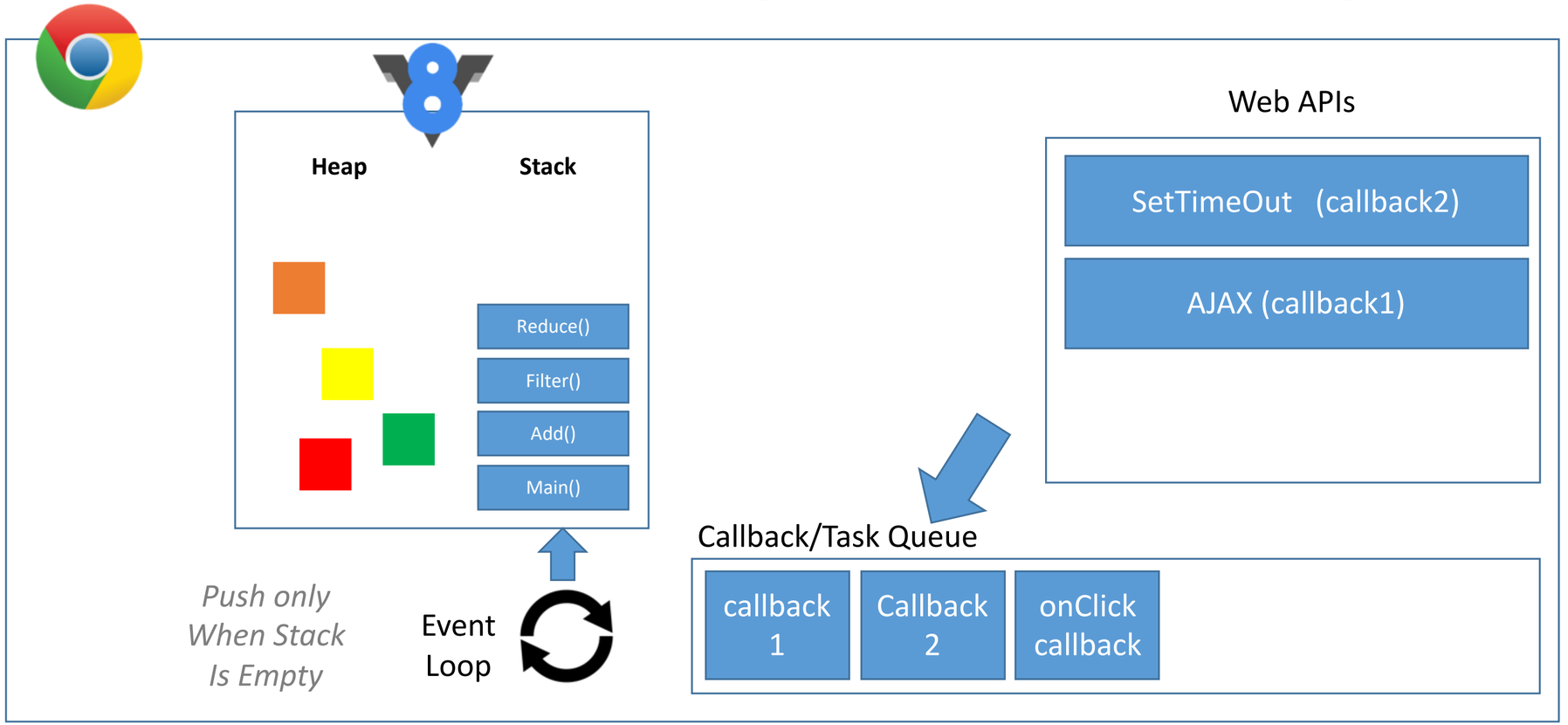
Synchronous and Asynchronous

- *Asynchronous* means more than one process running simultaneously.
- *Synchronous* means one process is executing at a time.
- **JavaScript/ V8 is Synchronous**





Chrome – Concurrency & the Event Loop



One thing at a time? Not really!

If you block the stack, browser can't do the render queue

The Boomerang Effect (Callback Hell)

As you may have noticed, asynchronous programming relies on callback functions that are usually passed as arguments. This can turn your code into "**callback spaghetti**", making it visually hard to track which context you are in. This style also makes debugging your application difficult, reducing even more the maintainability of your code.

```
async1(function(){
    async2(function(){
        async3(function(){
            async4(function(){
                ....
            });
        });
    });
});
```

Asynchronous Code

To write a nice asynchronous code away from the callback hell we can use one of these code structures:

- **Promises**
- **Async & Await**

PREV NEXT

The Promise Object

The **`Promise`** object is used for asynchronous computations. A **`Promise`** represents a value which may be available now, or in the future, or never.

```
new Promise( function(resolve, reject) {... resolve() ... reject() } );
```

A Promise has one of three states:

- **pending**: initial state, not fulfilled or rejected.
- **fulfilled**: meaning that the operation completed successfully.
- **rejected**: meaning that the operation failed.

A pending promise can either be fulfilled with a value, or rejected with a reason (error).

How Promises can make our code easy to read

The **`Promise`** object has two methods, **`then`** and **`catch`**. The methods will later be called depending on the state (fulfilled or rejected) of the Promise Object.

```
const postsPromise = fetch('http://mywebsite.com/API'); // returns Promise

postsPromise.then(data => data.json()) // After data is being received
            .then(data => { console.log(data) })
            .catch((err) => { console.error(err); }) // in case rejected
```

As the `Promise.prototype.then()` and `Promise.prototype.catch()` methods return promises, they can be chained.

Creating a promise

```
var giveMePizza = function(){
        return new Promise(function(resolve,reject){
                if(everythingWorks){
                        resolve("This is true"); // then() will be called
                } else {
                        reject("This is false"); // catch() will be called
                }
        })
}
giveMePizza()
        .then(data => console.log(data))
        .catch(err => console.error(err));
console.log('I will run immediately after calling giveMePizza()');
```

The callback from the `Promise` constructor gives us two parameters, `resolve` and `reject` functions, that will affect the state of the `Promise` object. If everything works, call `resolve`, otherwise call `reject`. Note that you can pass in values to `resolve` and `reject` which will be further passed on to the respective handlers, `then` and `catch`.

14

Fetch API

The Fetch API provides a JavaScript interface for accessing and manipulating requests and responses. It provides a global **`fetch()`** method that provides an easy, logical way to fetch resources asynchronously across the network.

This kind of functionality was previously achieved using **`XMLHttpRequest`**.

Fetch also provides a single logical place to define other HTTP-related concepts such as **`CORS`**.

Using Fetch

The **`fetch()`** method takes one mandatory argument, the path to the resource you want to fetch. It returns a Promise that resolves to the Response to that request.

```
fetch('http://example.com/movies.json')
   .then(response => response.json())
   .then(myJson => console.log(myJson));

fetch('http://example.com/answer',
      { method: 'POST', // *GET, POST, PUT, DELETE, etc.
        mode: 'cors',
        headers: {'Content-Type': 'application/json'},
        body: JSON.stringify({answer: 42})
      })
   .then(response => response.json());
   .catch(error => console.error(error));
```

Async/Await

Take advantage of the synchronous-looking code style.

`await` is always used before functions that return a **`Promise`**, If the awaited expression isn't a promise, it's casted into a promise.

The awaited function must be wrapped in another function marked with **`async`**.

For error handling, use **`try catch`** block.

Keep in mind that you should wrap await in try / catch so that you can capture and handle errors in awaited promises from within the `async` function.

Example - Promise

```
var Studied = true;
var willPassTheExam = function(){
        return new Promise(function(resolve, reject) {
        if (Studied) resolve('Pass');
        else reject(new Error('Fail'));
        })
};

var askMe = function () {
        willPassTheExam()
                .then(function(results){ console.log(results); })
                .catch(function (error) { console.log(error); });
};

askMe();
console.log('Finish')
```

Example - Promise

```
var Studied = true;
var willPassTheExam = function(){
        return new Promise(function(resolve, reject) {
        if (Studied) resolve('Pass');
        else reject(new Error('Fail'));
        })
};

var askMe = function () {
        willPassTheExam()
                .then(function(results){ console.log(results); })
                .catch(function (error) { console.log(error); });
};

askMe();
console.log('Finish')

// Finish
// Pass
```

Example – Async & Await

1

2

3

```
async function askMe() {
        try {
            console.log('Before taking the exam');
            let results = await willPassTheExam();
            console.log(results);
            console.log('After taking the exam');
        } catch (error) {
            console.log(error);
        }
}

askMe();
console.log('Finish')
```


PREV NEXT

Example – Async & Await

1

2

3

```
async function askMe() {
        try {
            console.log('Before taking the exam');
            let results = await willPassTheExam();
            console.log(results);
            console.log('After taking the exam');
        } catch (error) {
            console.log(error);
        }
}

askMe();
console.log('Finish')

// Before taking the exam
// Finish
// Pass
// After taking the exam
```


PREV NEXT

Reactive Programming

Reactive programming is programming with asynchronous data streams.

Rx gives us an amazing toolbox of functions to combine, create and filter any of those streams.

http://reactivex.io/

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.

To start using RxJS library in Node: `npm install rxjs`

To start using RxJS library in Browser: `<script src="https://unpkg.com/rxjs/bundles/rxjs.umd.min.js"></script>`

Everything can be a stream

Streams are cheap, We are able to create data streams of anything : variables, events, user inputs, properties, caches, data structures, etc.

You can listen to that stream and react accordingly.

Wildly used in Angular framework.

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?

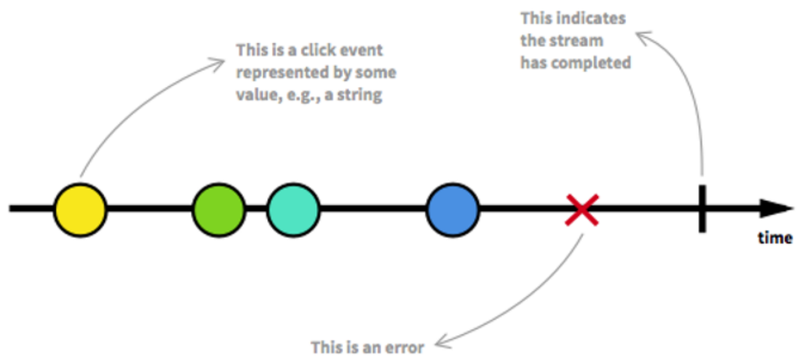
Reactive Programming raises the level of abstraction of your code so you can focus on the events that define the business logic, rather than having to constantly work with a large amount of implementation details.

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 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.

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.



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.

How Observables work?

We capture those 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.**

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**

André Staltz

31

PREV NEXT

Creating Simple Observable Example

```
const { Observable } = rxjs;

const obs$ = Observable.create(function (observer) {
   observer.next('CS572');
   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, or to receive error or completed notifications from the Observable, it must first subscribe to that Observable with this operator.

35

Converting Values to Observables

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

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

Converting Data Sets to Observables

```
const { from } = rxjs;
const { map, filter } = rxjs.operators;
const data = [
        { id: 0, name: 'Learning Node', topic: 'Node JS', },
        { id: 1, name: 'Learning MongoDB', topic: 'MongoDB', },
        { id: 2, name: 'Learning TypeScript', topic: 'TypeScript', }]

    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));
```


PREV NEXT

Subject Observable

```
const { Subject } = rxjs;

var myObservable$ = new Subject();

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

setTimeout(() => { myObservable$.next('CS472'); }, 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

```
<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>
```

scan() and reduce() Operators

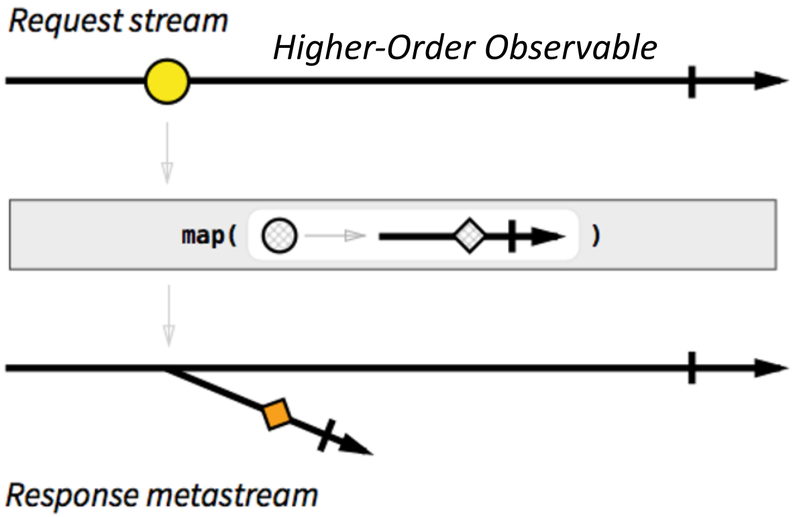
```
const observable$ = of(1,2,3,4,5);

observable$
        .pipe(
            reduce((total, currentValue) => { return total + currentValue; }, 0)
        )
        .subscribe( (value) => console.log(value) );
// 15

observable$
        .pipe(
            scan((total, currentValue) => { return total + currentValue; }, 0)
        )
        .subscribe( (value) => console.log(value) );
// 1, 3, 6, 10, 15
```


PREV NEXT

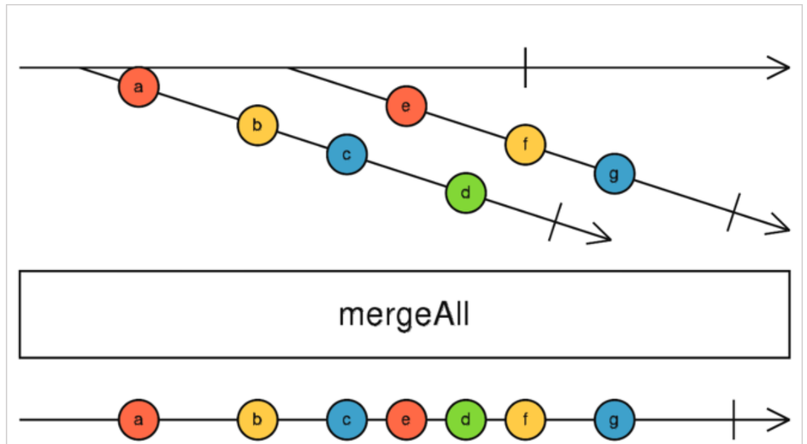
OUTLINE

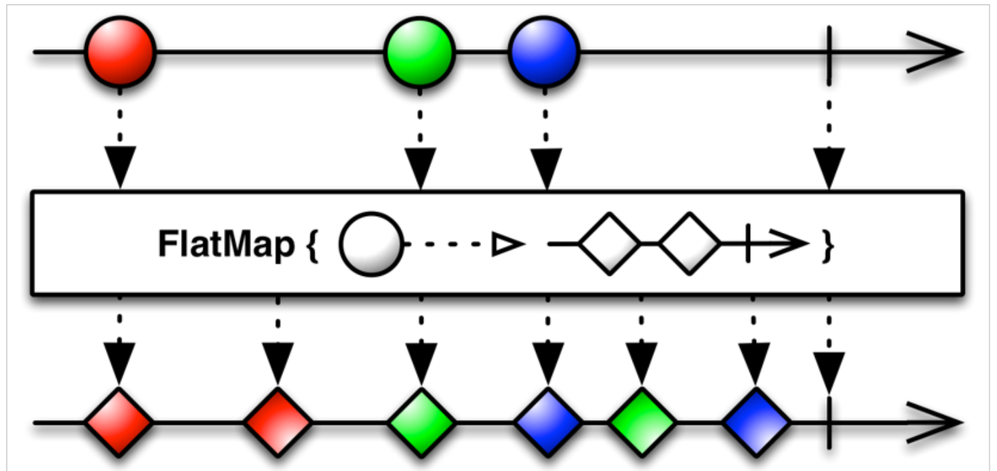
Metastream (Higher-Order Observable)

A metastream is a stream where each emitted value is yet another stream.



mergeAll() vs flatMap()





mergeAll subscribes to an Observable that emits Observables, also known as a higher-order Observable. Each time it observes one of these emitted inner Observables, it subscribes to that and delivers all the values from the inner Observable on the output Observable.

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.

mergeAll() vs 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(x => console.log(x));
```

Real-Life Example

```
const { just, from } = rxjs;
const { flatMap } = rxjs.operators;

const myStream$ = just('data.json')
    .pipe(
        flatMap( (requestUrl) => from(fetch(requestUrl)) );
)
.subscribe(response => {console.log(response)})
```

Caching

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/rxjs/6.5.2/rxjs.umd.min.js"></script>

<script>

const obs$ = rxjs.Observable.create(async (observer) => {
    const results = await fetch('https://randomuser.me/api')
    if (results.error) observer.error(results.error)
     else observer.next(results)
     observer.complete()
    })

    obs$.subscribe(null)
    obs$.subscribe(null)
</script>
```

Caching

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/rxjs/6.5.2/rxjs.umd.min.js"></script>

<script>
const { shareReplay } = rxjs.operators
const obs$ = rxjs.Observable.create(async (observer) => {
    const results = await fetch('https://randomuser.me/api')
    if (results.error) observer.error(results.error)
     else observer.next(results)
     observer.complete()
    }).pipe(shareReplay(1))

    obs$.subscribe(null)
    obs$.subscribe(null)
</script>
```