FRONT-END

LECTURE VI/VI

SCHEDULE

Lecture IV (today)

I. HomeWork

   A. Anyone?

II. History

   A. Callbacks

   B. Promises

   C. Async-await

Homework

➢ Continue working on Feature

Homework

VI/VI

The history of Async JS

VI/VI

Callbacks - Promises - Async/await

synopsis

In order to make websites / apps dynamic we need to fetch data from an external source.

Fetching data from an external source takes time. In order to deal with these “asynchronous events” several patterns were developed:

Callbacks / Promises / Async-await are the 3 most popular patterns

Callbacks

Synopsis

In order to understand callbacks we need to understand functions

We know how to create a function;

```
function add (x, y) {
  return x + y;
}
```

Callbacks

SYNOPSIS

We also know how to create a reference to the function and store that reference inside of a variable. Say I want two references to the function, one for my father and one for my mother; I can do the following:

```
function add (x, y) {
  return x + y;
}

const mom  = add;
mom(5, 10);

const dad = add;
dad(5, 20);
```

Callbacks

SYNOPSIS

We can create as many references to the function add as we want.

Now, we can also create an addFive function which takes in 2 params, first being a number to add ‘5’ to, and second being a reference to a function. And by reference to a function I mean e.g mom or dad. Since mom and dad refer to the add function.

```
function add (x, y) {
  return x + y;
}

const mom  = add;
mom(5, 10);

const dad = add;
dad(5, 20);
```

Callbacks

Synopsis

That being said; we can now write the following logic:

```
function add (x, y) {
  return x + y;
}

function addFive (x, referencedFunction) {
  return referencedFunction (5, x);
}

addFive(10, add);
```

So what we are actually doing is passing a function as an argument to another function.

Making the addFive() function a:

higherOrderFunction

the argument being the function - in this case the add function,

is referred to being a **callback**

Callbacks

SYNOPSIS

So in essence;

equals

```
function add (x, y) {
  return x + y;
}

function addFive (x, referencedFunction) {
  return referencedFunction (5, x);
}

addFive(10, add);
```

```
function add (x, y) {
  return x + y;
}

function higherOrderFunction (x, callback) {
  return callback (5, x);
}

higherOrderFunction(10, add);
```

Callbacks

synopsis

Takeaways:

- Any function that receives another function as its argument is a higher order function
- The function you are passing in as the argument is a callback function

Callbacks

synopsis

Callbacks are everywhere;

```
const button = document.querySelector('.btn');
// addEventListener is a Higher Order Function
button.addEventListener('click', function() {
  console.log('Im a callback!');
});
```

Callbacks allow us to delay the execution of a function. In the above example we delay the execution of the function that runs console.log(‘Im a callback!’) until some event occurs.

Callbacks

synopsis

With the knowledge of being able to delay functions until specific events occur - we have a great use case for fetching external data.

We can now say; fetch the data - when the data is retrieved, run the following logic. And logic meaning, when referring back to the previous example; the console.log(“im a callback”)

Callbacks

synopsis

So callbacks are life.. right!?

Not.

Welcome to callback hell.

Callbacks

synopsis

Callback hell
source

Callbacks

synopsis

Downsides..

- callback hell: We want to modularize our code - make it more “manageable”

- inversion of control (passing a callback to a third party lib still requires the third party lib to call our callback function.. What if this doesn't happen?)

  well.. then our code breaks.

Callbacks

synopsis

Recap:

You call a restaurant - they are full. You leave your number and the restaurant tells you they will call whenever a spot opens up.

But what if they never - call back? (pun intended)

Promises

SYNOPSIS

In order to deal with the “loss of control” with respect to callbacks the promise pattern was developed.

Promises consist of three states:

- pending (default state)
- fulfilled (when your promise is ready)
- rejected (when something goes wrong)

This way we are back in control.. Promises were invented to make the complexity of making asynchronous requests more manageable

Promises

synopsis

But how do I create a promise?

How do I change the state of a promise?

How do I listen for status changes of a promise?!

Promises

synopsis

Creating a promise.

You can do so by creating a new instance of the promise constructor

```
const promise = new Promise();
```

Promises

synopsis

The promise constructor takes in 1 argument, namely; a callback. The callback takes in two arguments; a callback function named resolve and a callback function called reject.

```
const promise = new Promise((resolve, reject) => {
  // write logic
});
```

Promises

synopsis

Whenever the callback function Resolve is called, it is going to change the status of the promise to fulfilled.

When Reject is called, it is going to change the status of the promise to reject.

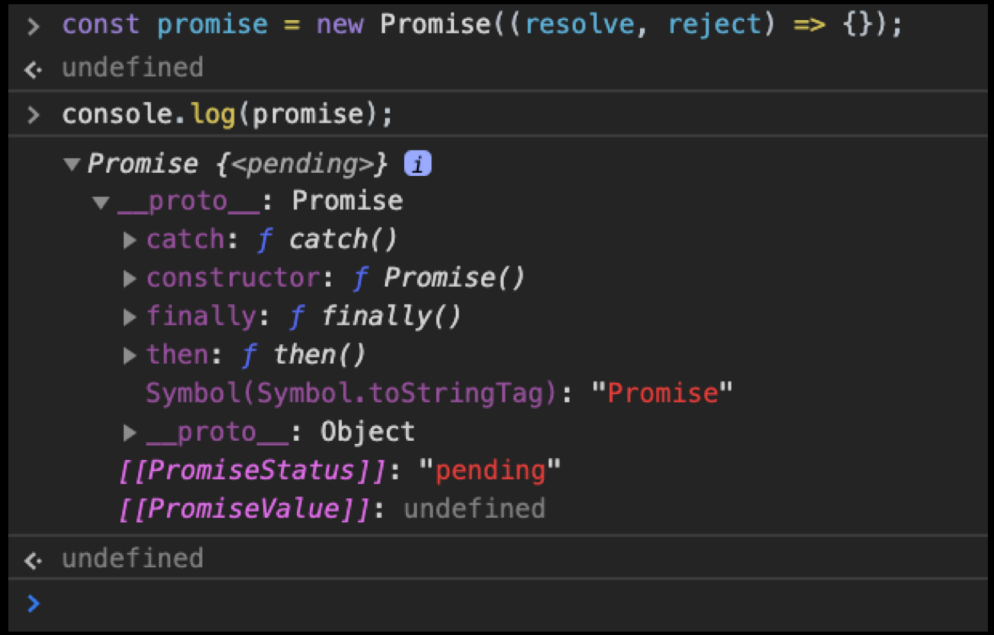
```
const promise = new Promise((resolve, reject) => {
  // write logic
});
```

Promises

SYNOPSIS

Now we just need to know how to listen for these status changes..

In order to understand what's going on - we need to understand what promises do under the hood.



Promises

synopsis

What we see here is that on the prototype of the Promise constructor several methods are at our disposal.

To answer our question of; “how to listen for a status change” we require the catch() and then() methods.

```
> const promise = new Promise((resolve, reject) => {});
<- undefined
> console.log(promise);
  ▼ Promise {<pending>} i
    ▼ __proto__: Promise
      ▶ catch: ƒ catch()
      ▶ constructor: ƒ Promise()
      ▶ finally: ƒ finally()
      ▶ then: ƒ then()
        Symbol(Symbol.toStringTag): "Promise"
      ▶ __proto__: Object
      [[PromiseStatus]]: "pending"
      [[PromiseValue]]: undefined
<- undefined
>
```

Promises

synopsis

Whenever the callback “resolve” is being called, the .then() function on the promise prototype will be called. So our logic for our function success - can be placed inside of the .then() function

Whenever the callback “reject” is being called, the .catch() function on the promise prototype will be called. So our logic for when function error can be placed inside of the .catch function.

In order to display this we can write the following code:

Promises

Synopsis

```
function onSuccess () {
  console.log('Our function was a success!');
}

function onError () {
  console.log('Our code went to 💩');
}

const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve() // fulfilled promise
  }, 2000);
});

promise.then(onSuccess);
promise.catch(onError);
```

Here we resolve our promise, meaning we can write our promise.then() logic and call our onSuccess function.

The same goes for the case when we reject our promise, promise.catch() will then be called with the onError value.

Promises

synopsis

This allows for a more modular approach and understandable code orientation.

Important to know is that whenever .then() or .catch() is invoked - it is going to wrap whatever you are returning inside of a new promise, which allows us to again .then() or .catch() on the outcome. This concept is called promise chaining and looks like this:

Promises

synopsis

```
function logA () {
  console.log('Im A');
}

function logB () {
  console.log('Im B');
}

function logC () {
  console.log('Im C');
}

function throwTheErr () {
  throw new Error();
}

function logErr () {
  console.log('Our code when to 💩 ');
}
```

```
function myPromise () {
  return new Promise((resolve, reject) => {
    setTimeout(function() {
      resolve();
    }, 2000);
  });
}

myPromise()
  .then(logA)
  .then(logB)
  .then(logC)
  .then(throwTheErr)
  .catch(logErr);
```

Promises

SYNOPSIS

Okok. Im convinced; promises are life.

Promises

Or…. are they?

synopsis

Async-await

SYNOPSIS

Benefits of async-await: we could write our asynchronous code as if it were synchronous!!!!11!111!111!!!11!!! L33tH4x0rsz

This kind of makes us invincible.

source

Async-await

SYNOPSIS

So in essence we could do the following:

```
async function getMehData() {
  const userData = await getUserData();
  const weather = await getCurrentWeather();

  return {
    user: userData,
    degrees: weather.currentDegrees
  }
}
```

We tell the JS interpreter it is dealing with an async function - and then it can just await the async processes.

Async-await

synopsis

Async functions always return us a promise.

Async await is sugar syntax on top of promises.

So how do we handle errors? Well one way to do so is by placing our code inside of a try catch block;

Async-await

SYNOPSIS

```
async function getMehData() {
  try {
    const userData = await getUserData();
    const weather = await getCurrentWeather();

    return {
      user: userData,
      degrees: weather.currentDegrees
    }
  } catch(err) {
    console.log('Our code went to 💩');
  }
}
```

Advanced:
Another way would be to call .catch() on getMehData since all async functions return a promise..

But then we would have to be in control of invoking the function

SYNOPSIS

That was a lot.. sorry.

Now go be people. Go be.

source

SYNOPSIS

Homework

- Continue working on Feature

EXIT;

see you Next();