Context for today

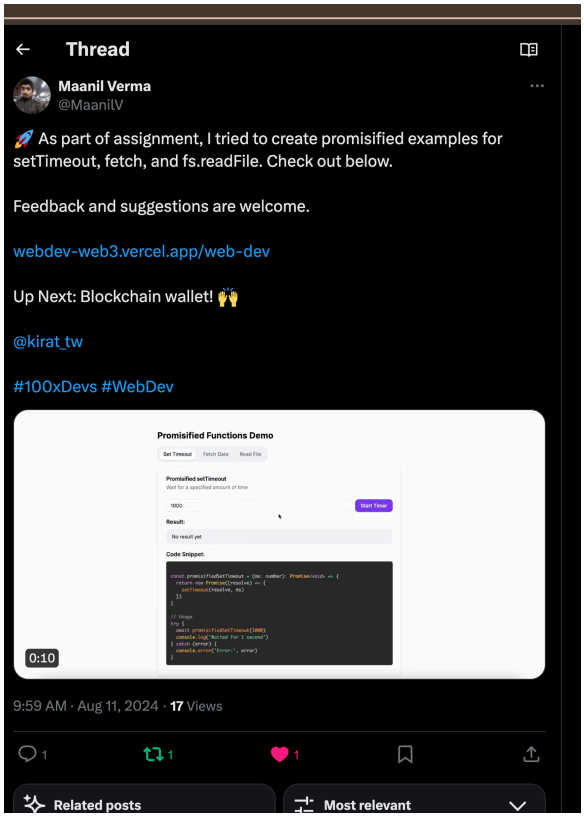
`Aaj ghode khuleinge.`

Bounty

Bounty $50

https://webdev-web3.vercel.app/web-dev

https://x.com/MaanilV/status/1822490498252337262



What we're doing today

1. Classes in JS
2. Revise callbacks
3. Callback hell
4. Promises
5. Async await

Assignments

Will release a video on how to solve them

https://github.com/100xdevs-cohort-3/assignments/

Assignments

Classes in JS

Primitive types

1. number
2. string
3. boolean

Complex types

1. Objects
2. Arrays

Classes

In JavaScript, classes are a way to define blueprints for creating objects (these objects are different from the objects defined in the last section).

For example

```
class Rectangle {
   constructor(width, height, color) {
        this.width = width;
        this.height = height;
        this.color = color;
   }

   area() {
       const area = this.width * this.height;
         return area;
   }

   paint() {
            console.log(`Painting with color ${this.color}`);
   }

}
```

```
const rect = new Rectangle(2, 4)
const area = rect.area();
console.log(area)
```

Key Concepts

1. **Class Declaration**:
    - You declare a class using the `class` keyword.
    - Inside a class, you define properties (variables) and methods (functions) that will belong to the objects created from this class.
2. **Constructor**:
    - A special method inside the class that is called when you create an instance (an object) of the class.
    - It's used to initialize the properties of the object.
3. **Methods**:
    - Functions that are defined inside the class and can be used by all instances of the class.
4. **Inheritance**:
    - Classes can inherit properties and methods from other classes, allowing you to create a new class based on an existing one.
5. **Static Methods**:
    - Methods that belong to the class itself, not to instances of the class. You call them directly on the class.
6. **Getters and Setters**:
    - Special methods that allow you to define how properties are accessed and modified.

Inheritance in classes

Inheritance in JavaScript classes allows one class to inherit properties and methods from another class. This mechanism enables code reuse, making it easier to create new classes that are based on existing ones, without having to duplicate code.

Assignment #1 - Create a Circle class

```
class Circle {
   constructor(radius, color) {
       this.radius = radius;
       this.color = color;
   }

   area() {
      const area = this.radius * this.radius * Math.PI;
       return area;
   }

   paint() {
            console.log(`Painting with color ${this.color}`);
   }

}

const circle = new Circle(2, "red")
const area = circle.area();
console.log(area)
```

💡 Can you see there is code repetition here and in the Rectangle class?

Assignment #2 - Create a base shape class

Base class

```
class Shape {
    constructor(color) {
```

```
        this.color = color;
    }

    paint() {
            console.log(`Painting with color ${this.color}`);
    }

    area() {
        throw new Error('The area method must be implemented in the subclass');
    }

    getDescription() {
        return `A shape with color ${this.color}`;
    }
}
```

Rectangle class

```
class Rectangle extends Shape {
    constructor(width, height, color) {
        super(color);  // Call the parent class constructor to set the color
        this.width = width;
        this.height = height;
    }

    area() {
        return this.width * this.height;
    }

    getDescription() {
        return `A rectangle with width ${this.width}, height ${this.height}, and
    }
}
```

Circle class

```
class Circle extends Shape {
    constructor(radius, color) {
        super(color);  // Call the parent class constructor to set the color
        this.radius = radius;
    }
```

```
    area() {
        return Math.PI * this.radius * this.radius;
    }

    getDescription() {
        return `A circle with radius ${this.radius} and color ${this.color}`;
    }
}
```

Try playing with it

```
const circle = new Circle(20);
console.log(circle.area());
```

Some more classes

Date

```
const now = new Date(); // Current date and time
console.log(now.toISOString()); // Outputs the date in ISO format
```

Maps

```
const map = new Map();
map.set('name', 'Alice');
map.set('age', 30);
console.log(map.get('name'));
```

Promise class

💡 Calling a promise is easy, defining your own promise is where things get hard

A **Promise** in JavaScript is an object that represents the `eventual completion` (or failure) of an asynchronous operation and its resulting value. Promises are used to handle asynchronous operations more effectively than traditional callback functions, providing a cleaner and more manageable way to deal with code that executes asynchronously, such as API calls, file I/O, or timers.

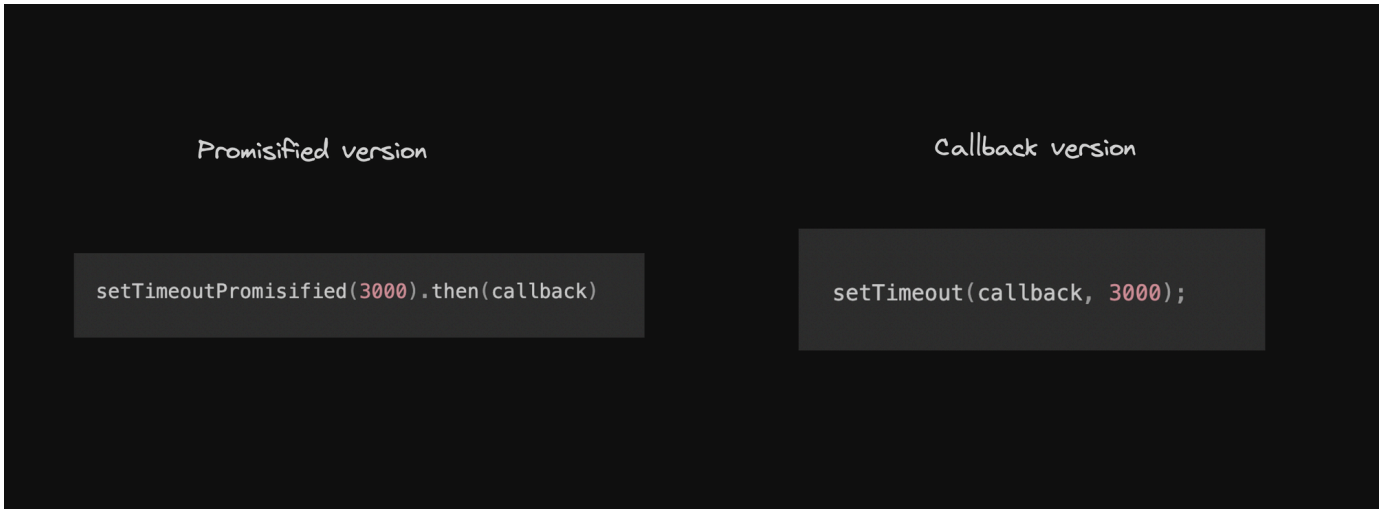
Using a function that returns a promise

Ignore the function definition of `setTimeoutPromisifed` for now

```
function setTimeoutPromisified(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

function callback() {
    console.log("3 seconds have passed");
}

setTimeoutPromisified(3000).then(callback)
```



Callback hell

Q: Write code that

1. logs `hi` after 1 second
2. logs `hello` 3 seconds after `step 1`
3. logs `hello there` 5 seconds after `step 2`

▼ Solution (has callback hell)

```
setTimeout(function () {
  console.log("hi");
  setTimeout(function () {
    console.log("hello");

    setTimeout(function () {
      console.log("hello there");
    }, 5000);
  }, 3000);
}, 1000);
```

▼ Alt solution (doesnt really have callback hell)

```
function step3Done() {
  console.log("hello there");
}

function step2Done() {
  console.log("hello");
  setTimeout(step3Done, 5000);
}

function step1Done() {
  console.log("hi");
  setTimeout(step2Done, 3000);
}

setTimeout(step1Done, 1000);
```

Promisified version

Now use the `promisified` version we saw in the last slide

```
function setTimeoutPromisified(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}
```

▼ Solution #1 (has callback hell)

```
function setTimeoutPromisified(ms) {
  return new Promise((resolve) => setTimeout(resolve, ms));
}

setTimeoutPromisified(1000).then(function () {
  console.log("hi");
  setTimeoutPromisified(3000).then(function () {
    console.log("hello");
    setTimeoutPromisified(5000).then(function () {
      console.log("hello there");
    });
  });
});
```

▼ Alt solution

```
setTimeoutPromisified(1000)
  .then(function () {
    console.log("hi");
    return setTimeoutPromisified(3000);
  })
  .then(function () {
    console.log("hello");
    return setTimeoutPromisified(5000);
  })
  .then(function () {
    console.log("hello there");
  });
```

Async await syntax

The `async` and `await` syntax in JavaScript provides a way to write asynchronous code that looks and behaves like synchronous code, making it easier to read and maintain.

It builds on top of Promises and allows you to avoid chaining `.then()` and `.catch()` methods while still working with asynchronous operations.

`async/await` is essentially syntactic sugar on top of Promises.

Assignment

Write code that

1. logs `hi` after 1 second
2. logs `hello` 3 seconds after `step 1`
3. logs `hello there` 5 seconds after `step 2`

```
function setTimeoutPromisified(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

async function solve() {
    await setTimeoutPromisified(1000);
    console.log("hi");
    await setTimeoutPromisified(3000);
    console.log("hello");
    await setTimeoutPromisified(5000);
    console.log("hi there");
}

solve();
```

Things to keep in mind

1. You can only call `await` inside a function if that function is `async`

2. You cant have a `top level await`

Defining your own async function

Q: Write a function that

1. Reads the contents of a file
2. Trims the extra space from the left and right
3. Writes it back to the file

1. Callback approach

In the callback approach, the function signature should look something like this -

```
function onDone() {
    console.log("file has been cleaned");
}
cleanFile("a.txt", onDone)
```

▼ Solution

```
const fs = require("fs");
function cleanFile(filePath, cb) {
  fs.readFile(filePath, "utf-8", function (err, data) {
    data = data.trim();
    fs.writeFile(filePath, data, function () {
      cb();
    });
  });
}

function onDone() {
  console.log("file has been cleaned");
}
cleanFile("a.txt", onDone);
```

2. Promisified approach

In the promisified approach, the function signature should look something like this -

```
async function main() {
   await cleanFile("a.txt")
   console.log("Done cleaning file");
}

main();
```

▼ Solution

```
const fs = require("fs");
function cleanFile(filePath, cb) {
  return new Promise(function (resolve) {
    fs.readFile(filePath, "utf-8", function (err, data) {
      data = data.trim();
      fs.writeFile(filePath, data, function () {
        resolve();
      });
    });
  });
}

async function main() {
  await cleanFile("a.txt");
  console.log("Done cleaning file");
}

main();
```

err first callback vs rejects in promises

Callbacks

`fs.readFile` function used an `err first callback` approach to propagate back errors

```
const fs = require("fs")
function afterDone(err, data) {
  if (err) {
    console.log("Error while reading file");
  } else {
    console.log(data)
  }
}

fs.readFile("a.txt", "utf-8", afterDone);
```

Promises

Promises use the `reject` argument to propagate errors

```
const fs = require("fs");

function readFilePromisified(filePath) {
  return new Promise(function (resolve, reject) {
    fs.readFile(filePath, "utf-8", function (err, data) {
      if (err) {
        reject("Error while reading file");
      } else {
        resolve(data);
      }
    });
  });
}

function onDone(data) {
  console.log(data);
}
```

```
function onError(err) {
  console.log("Error: " + err);
}

readFilePromisified("a.txt").then(onDone).catch(onError);
```