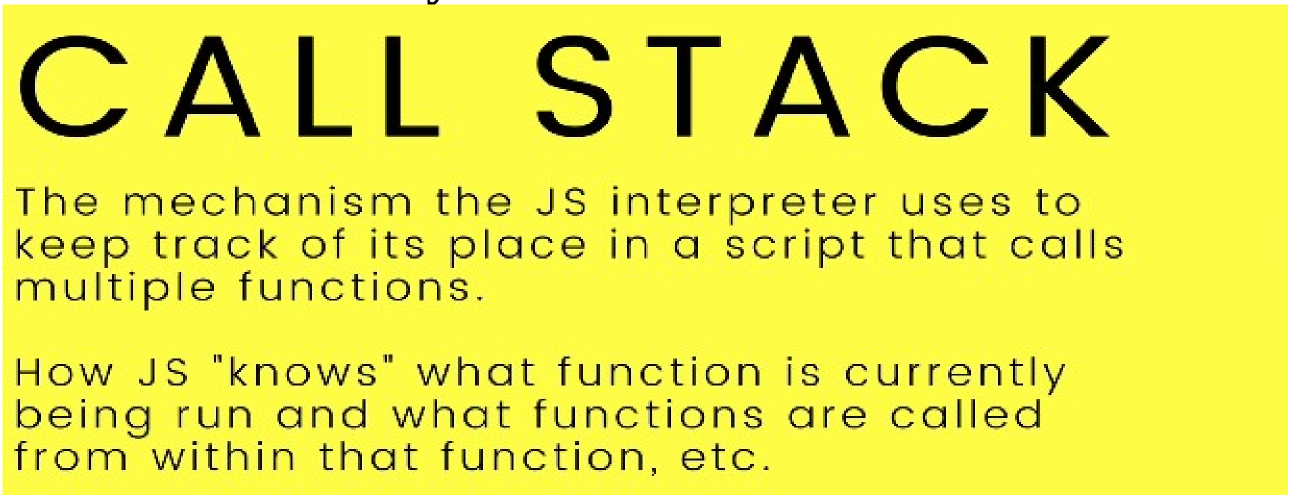
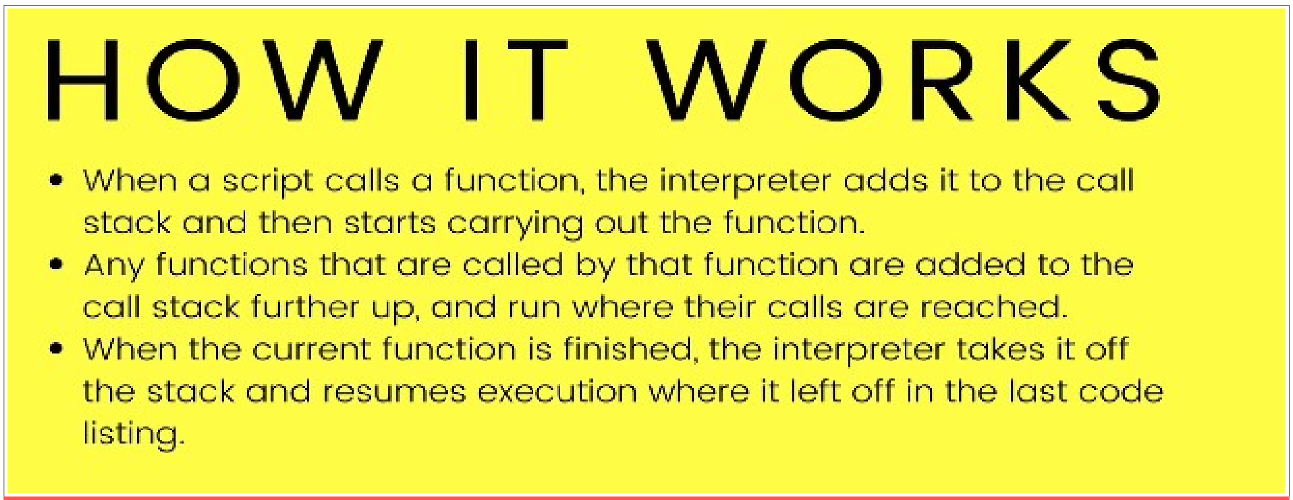
Asynchronous Js



CALL STACK

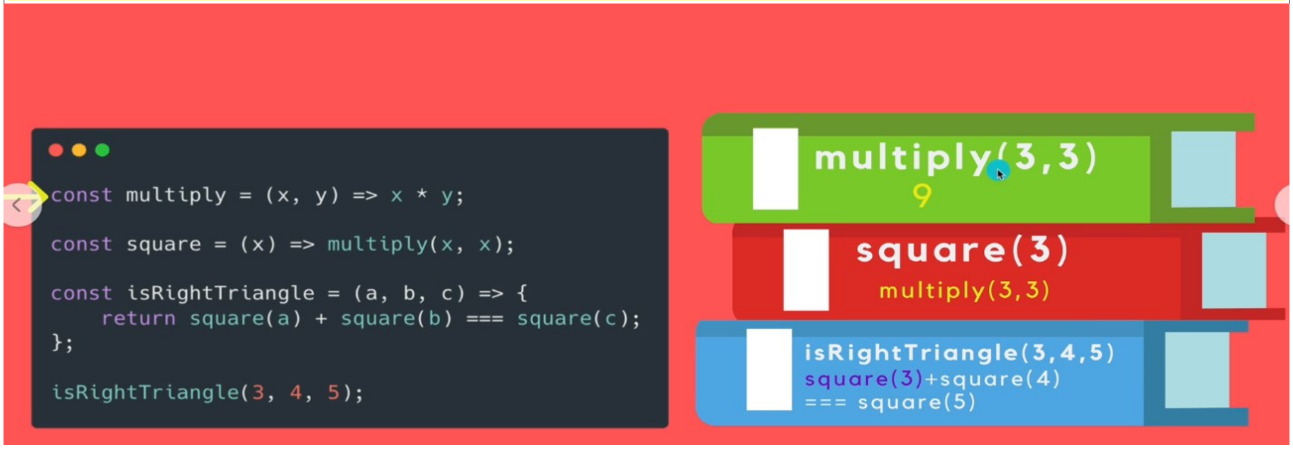
The mechanism the JS interpreter uses to keep track of its place in a script that calls multiple functions.

How JS "knows" what function is currently being run and what functions are called from within that function, etc.

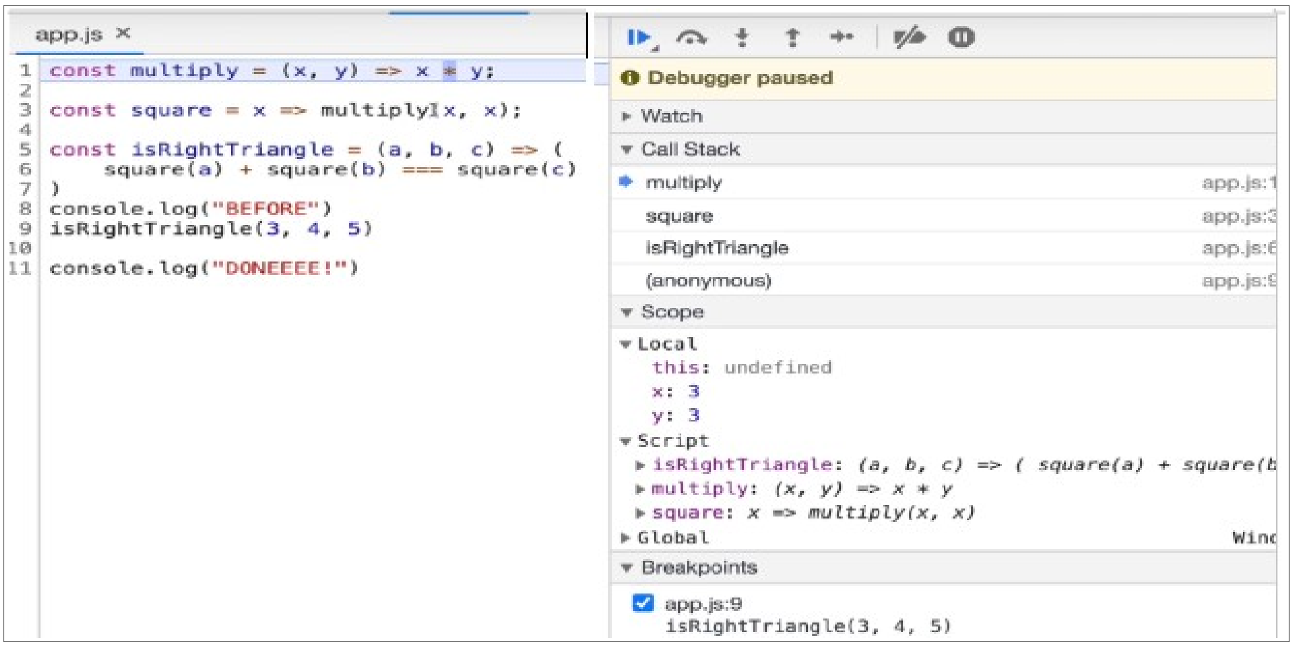


HOW IT WORKS

- When a script calls a function, the interpreter adds it to the call stack and then starts carrying out the function.
- Any functions that are called by that function are added to the call stack further up, and run where their calls are reached.
- When the current function is finished, the interpreter takes it off the stack and resumes execution where it left off in the last code listing.



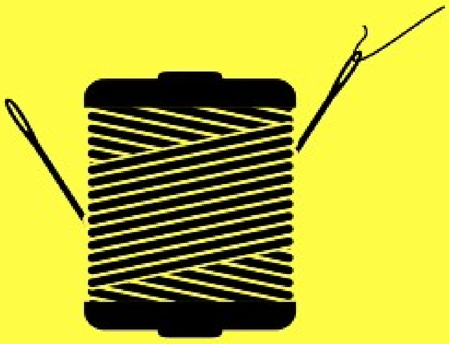
```
const multiply = (x, y) => x * y;

const square = (x) => multiply(x, x);

const isRightTriangle = (a, b, c) => {
    return square(a) + square(b) === square(c);
};

isRightTriangle(3, 4, 5);
```



```
app.js ×
const multiply = (x, y) => x * y;

const square = x => multiply(x, x);

const isRightTriangle = (a, b, c) => (
    square(a) + square(b) === square(c)
)
console.log("BEFORE")
isRightTriangle(3, 4, 5)

console.log("DONEEEE!")
```

```
Debugger paused
▸ Watch
▾ Call Stack
➡ multiply                                   app.js:1
  square                                     app.js:3
  isRightTriangle                            app.js:6
  (anonymous)                                app.js:9
▾ Scope
▾ Local
    this: undefined
    x: 3
    y: 3
▾ Script
  ▸ isRightTriangle: (a, b, c) => ( square(a) + square(b
  ▸ multiply: (x, y) => x * y
  ▸ square: x => multiply(x, x)
▸ Global                                     Winc
▾ Breakpoints
☑ app.js:9
   isRightTriangle(3, 4, 5)
```

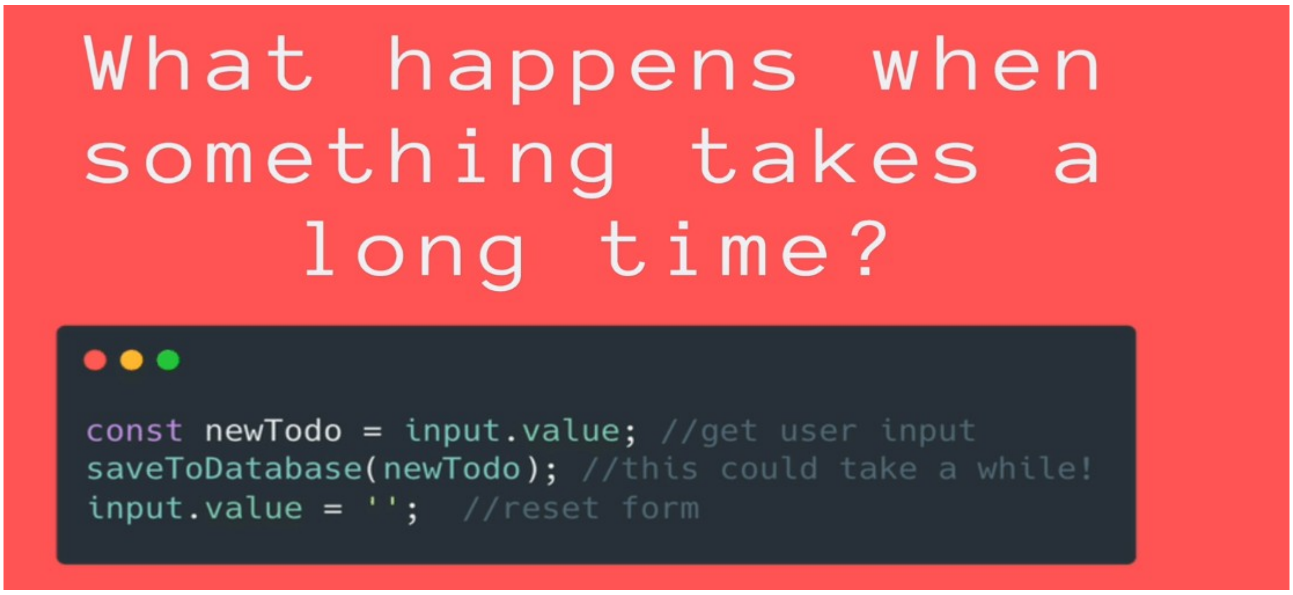
JS IS SINGLE THREADED



WHAT DOES THAT MEAN?

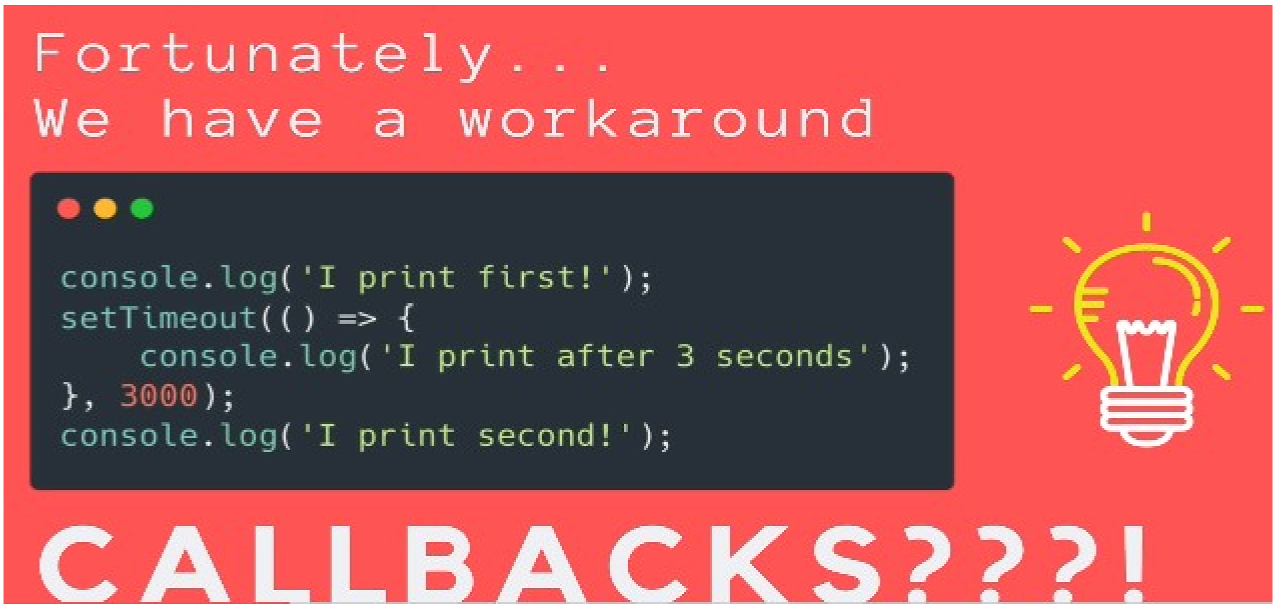
At any given point in time, that single JS thread is running at most one line of JS code.

IS OUR APP GOING TO GRIND TO A HALT?



What happens when something takes a long time?

```
const newTodo = input.value; //get user input
saveToDatabase(newTodo); //this could take a while!
input.value = '';  //reset form
```



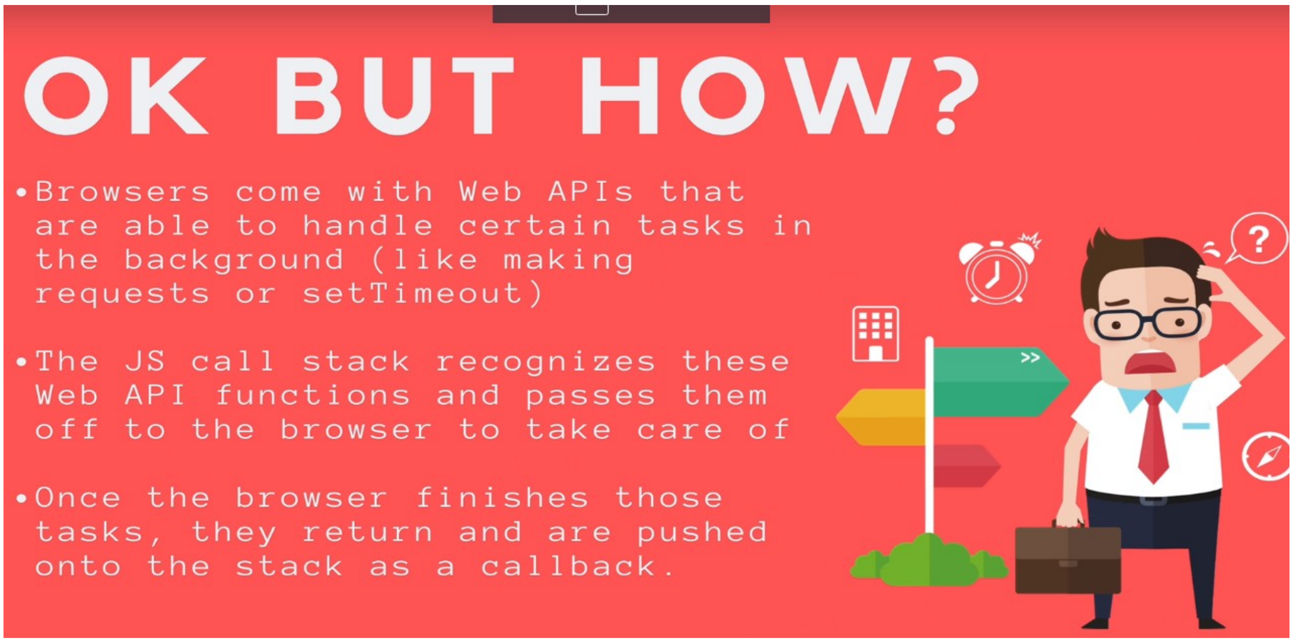
Fortunately...
We have a workaround

```
console.log('I print first!');
setTimeout(() => {
    console.log('I print after 3 seconds');
}, 3000);
console.log('I print second!');
```

CALLBACKS???!

setTimeout is a way of running a code after a delay

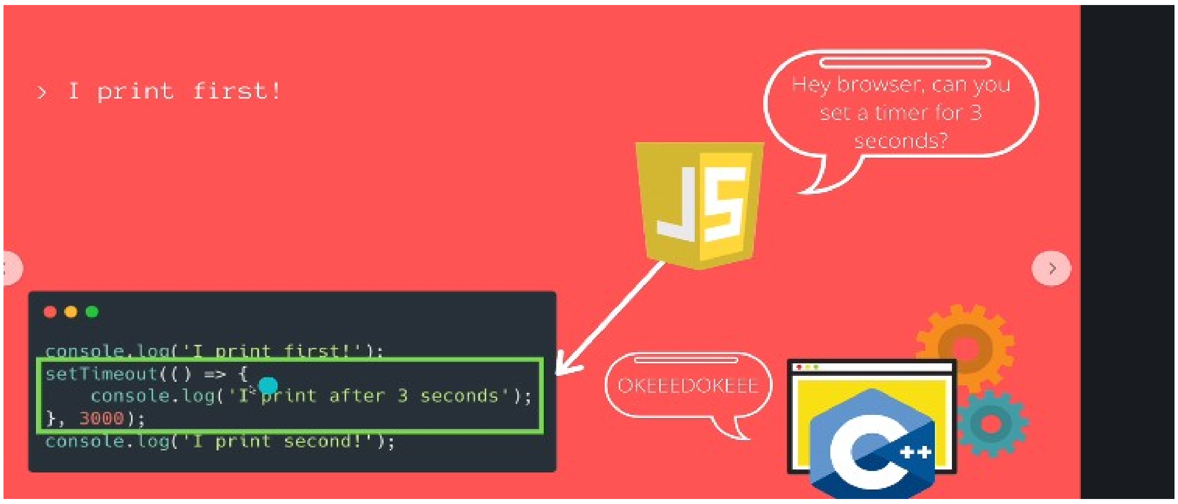
If js is single threaded shouldnt it setTimeout just stop everything for 3s
Ans:- The browser does the work
Your browser is written typically in C++, it can do things that js can’t do.
 What happen is js, hands off certain task to the browser to take care of.
Browser comes with something called web Api.



OK BUT HOW?

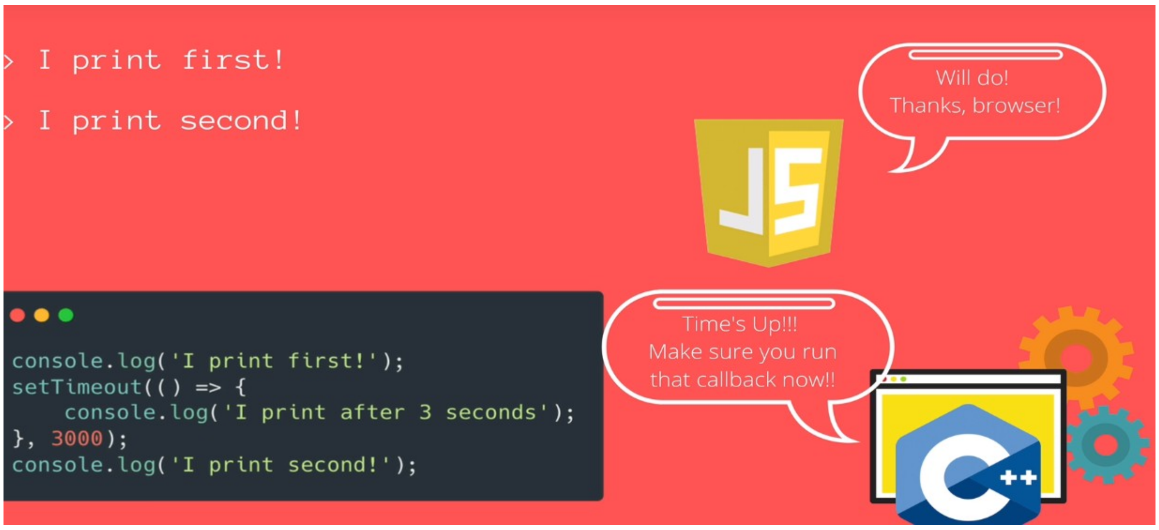
- Browsers come with Web APIs that are able to handle certain tasks in the background (like making requests or setTimeout)
- The JS call stack recognizes these Web API functions and passes them off to the browser to take care of
- Once the browser finishes those tasks, they return and are pushed onto the stack as a callback.

Video:js asyn



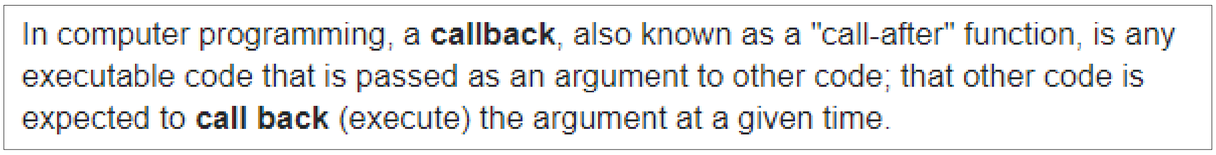
```
> I print first!
```

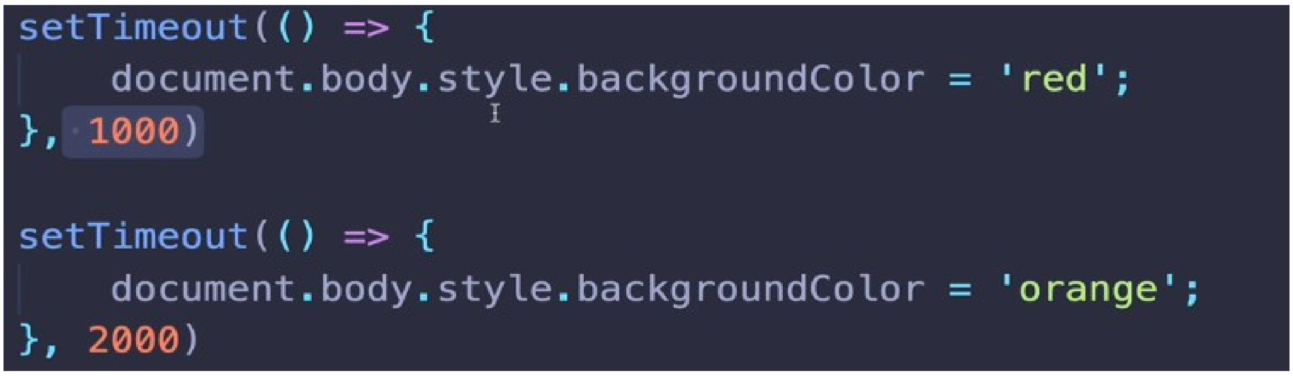
```
console.log('I print first!');
setTimeout(() => {
    console.log('I print after 3 seconds');
}, 3000);
console.log('I print second!');
```


```
> I print first!
> I print second!
```

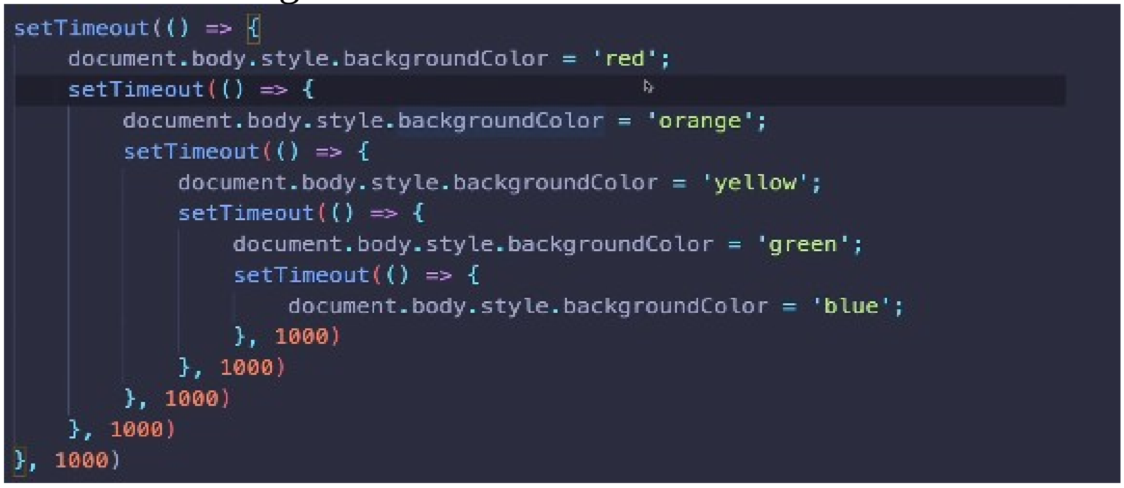
```
console.log('I print first!');
setTimeout(() => {
    console.log('I print after 3 seconds');
}, 3000);
console.log('I print second!');
```

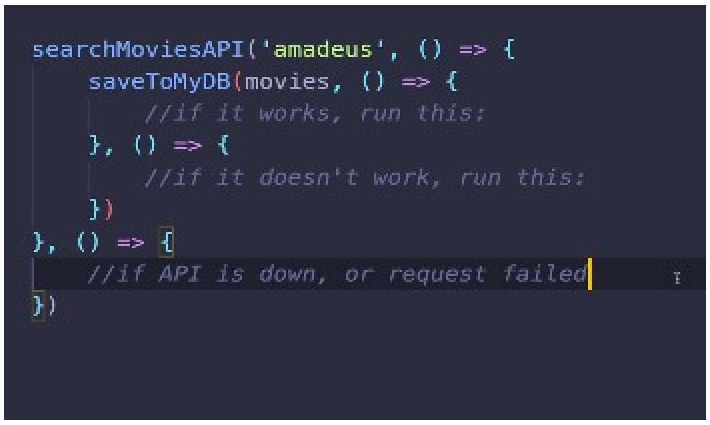
The whole mechanism it works, is just 2 things:
1) The browser comes with function, this thing that browser can do for us aka for js
2)


In computer programming, a **callback**, also known as a "call-after" function, is any executable code that is passed as an argument to other code; that other code is expected to **call back** (execute) the argument at a given time.


```
setTimeout(() => {
    document.body.style.backgroundColor = 'red';
}, 1000)

setTimeout(() => {
    document.body.style.backgroundColor = 'orange';
}, 2000)
```

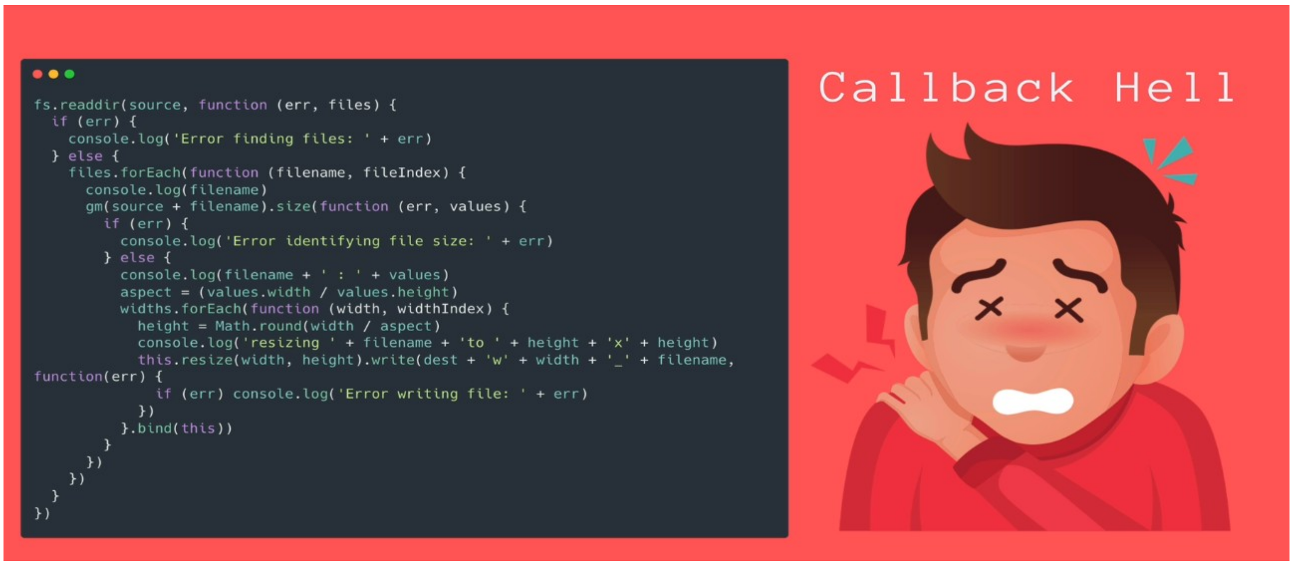
red took 1s.orange run 1s later.


```
setTimeout(() => {
    document.body.style.backgroundColor = 'red';
    setTimeout(() => {
        document.body.style.backgroundColor = 'orange';
        setTimeout(() => {
            document.body.style.backgroundColor = 'yellow';
            setTimeout(() => {
                document.body.style.backgroundColor = 'green';
                setTimeout(() => {
                    document.body.style.backgroundColor = 'blue';
                }, 1000)
            }, 1000)
        }, 1000)
    }, 1000)
}, 1000)
```



```
searchMoviesAPI('amadeus', () => {
    saveToMyDB(movies, () => {
        //if it works, run this:
    }, () => {
        //if it doesn't work, run this:
    })
}, () => {
    //if API is down, or request failed
})
```

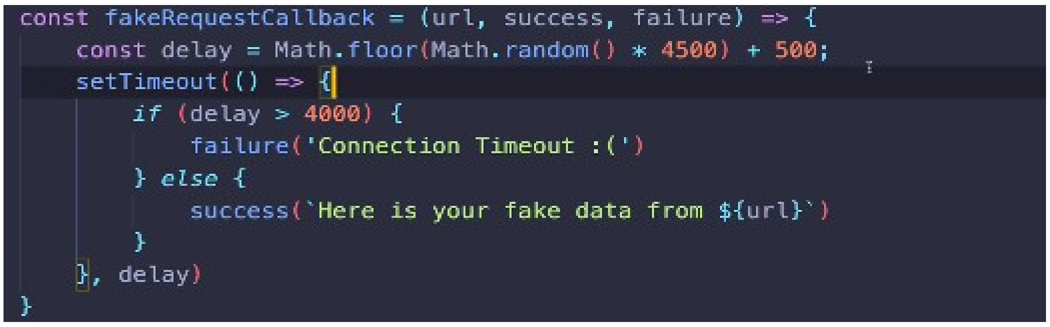
Asynchronous client is when u make a request, say whenever you ready whether you failed or success, just call this function – i am gonna give you a handle aka callback & i am go and do my job.
So there is a thread executing and doing its thing in background.



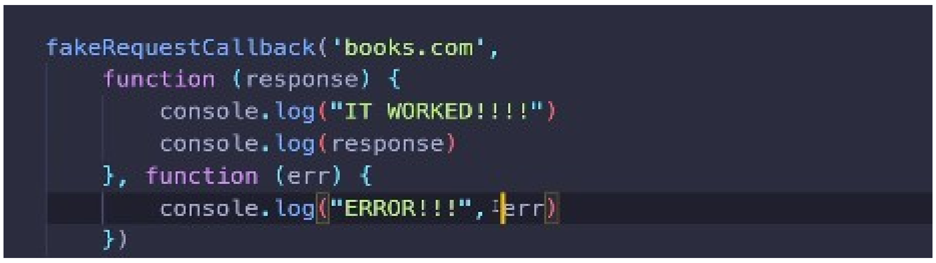
```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename,
function(err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }.bind(this))
        }
      })
    })
  }
})
```

MDN:PROMISE

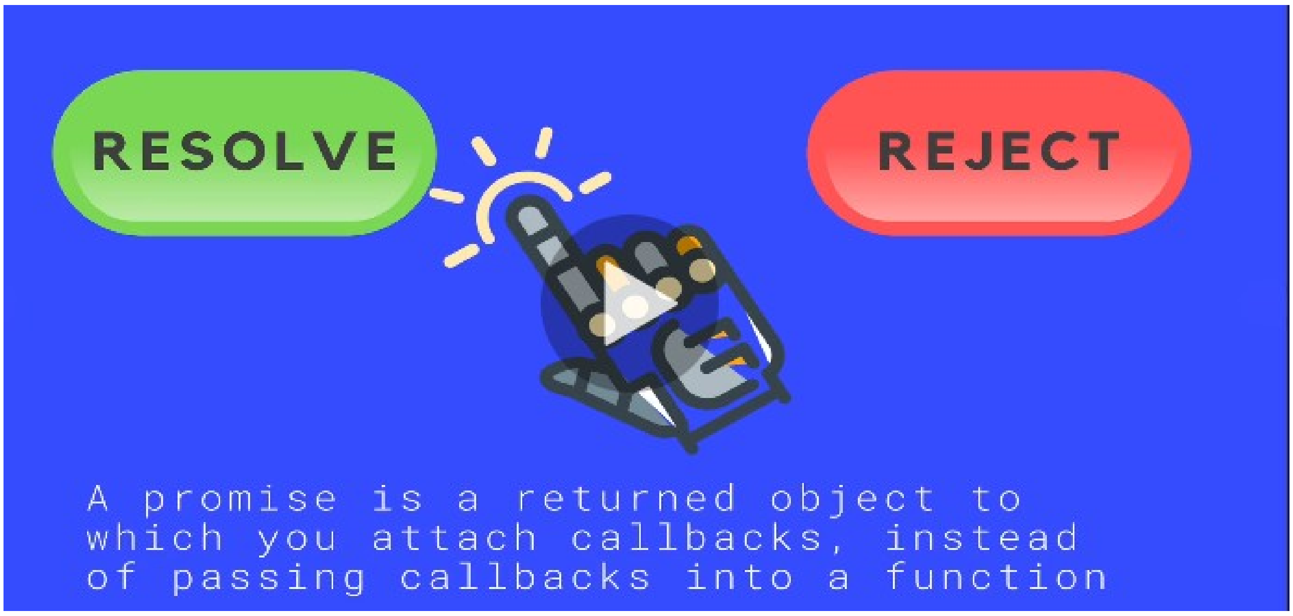
A Promise is an object representing the eventual completion or failure of an asynchronous operation



```
const fakeRequestCallback = (url, success, failure) => {
    const delay = Math.floor(Math.random() * 4500) + 500;
    setTimeout(() => {
        if (delay > 4000) {
            failure('Connection Timeout :(')
        } else {
            success(`Here is your fake data from ${url}`)
        }
    }, delay)
}
```

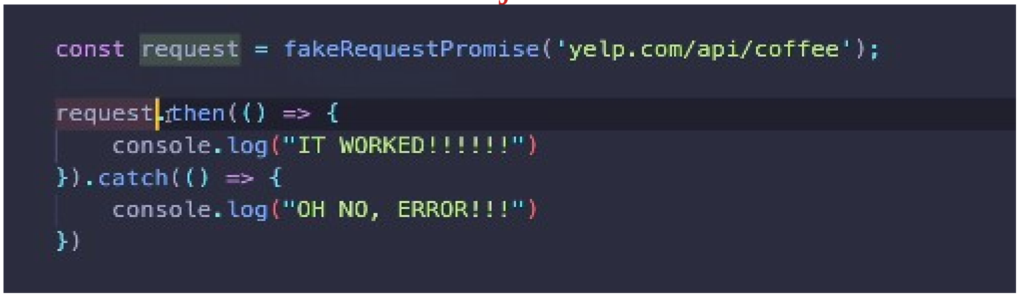


```
fakeRequestCallback('books.com',
    function (response) {
        console.log("IT WORKED!!!!")
        console.log(response)
    }, function (err) {
        console.log("ERROR!!!", err)
    })
```

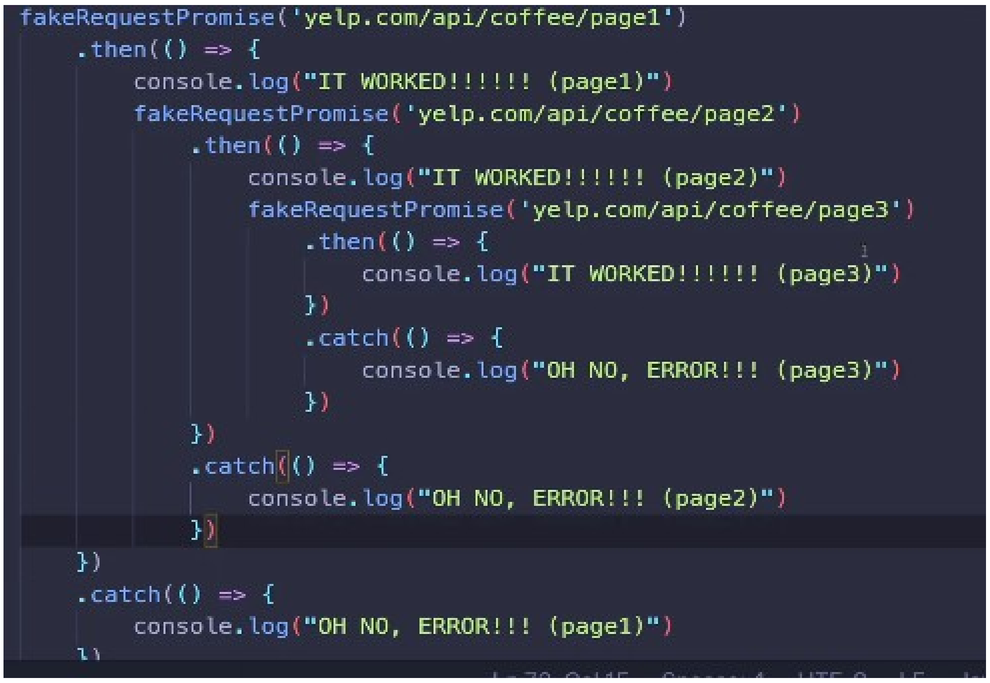


A promise is a returned object to which you attach callbacks, instead of passing callbacks into a function

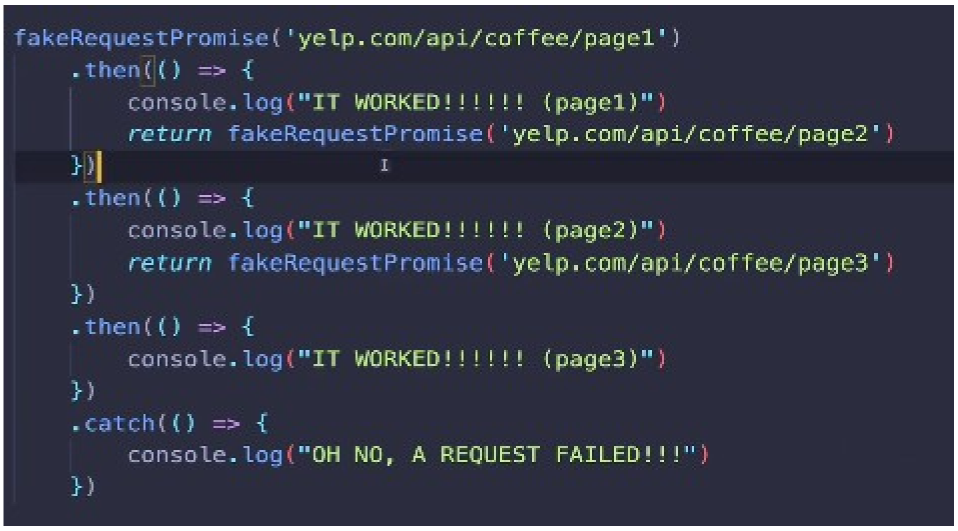
Asynchronous
so pending
and then either resolved or rejected.



```
const request = fakeRequestPromise('yelp.com/api/coffee');

request.then(() => {
    console.log("IT WORKED!!!!!!")
}).catch(() => {
    console.log("OH NO, ERROR!!!")
})
```

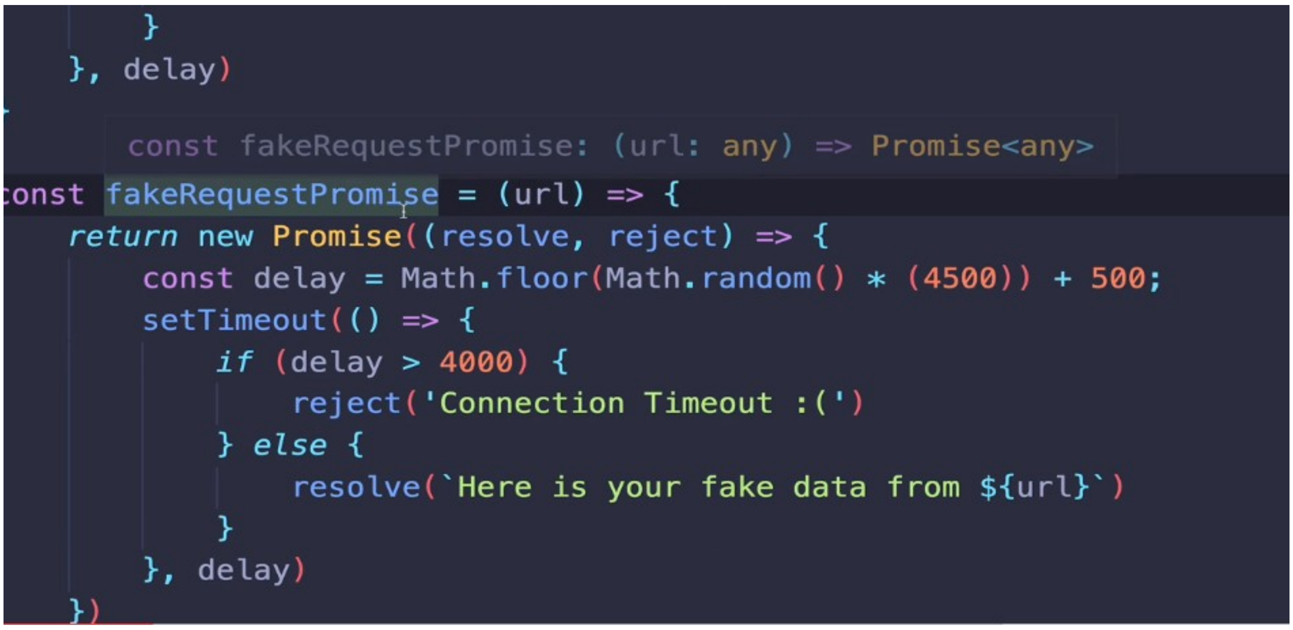


```
fakeRequestPromise('yelp.com/api/coffee/page1')
    .then(() => {
        console.log("IT WORKED!!!!!! (page1)")
        fakeRequestPromise('yelp.com/api/coffee/page2')
            .then(() => {
                console.log("IT WORKED!!!!!! (page2)")
                fakeRequestPromise('yelp.com/api/coffee/page3')
                    .then(() => {
                        console.log("IT WORKED!!!!!! (page3)")
                    })
                    .catch(() => {
                        console.log("OH NO, ERROR!!! (page3)")
                    })
            })
            .catch(() => {
                console.log("OH NO, ERROR!!! (page2)")
            })
    })
    .catch(() => {
        console.log("OH NO, ERROR!!! (page1)")
    })
```

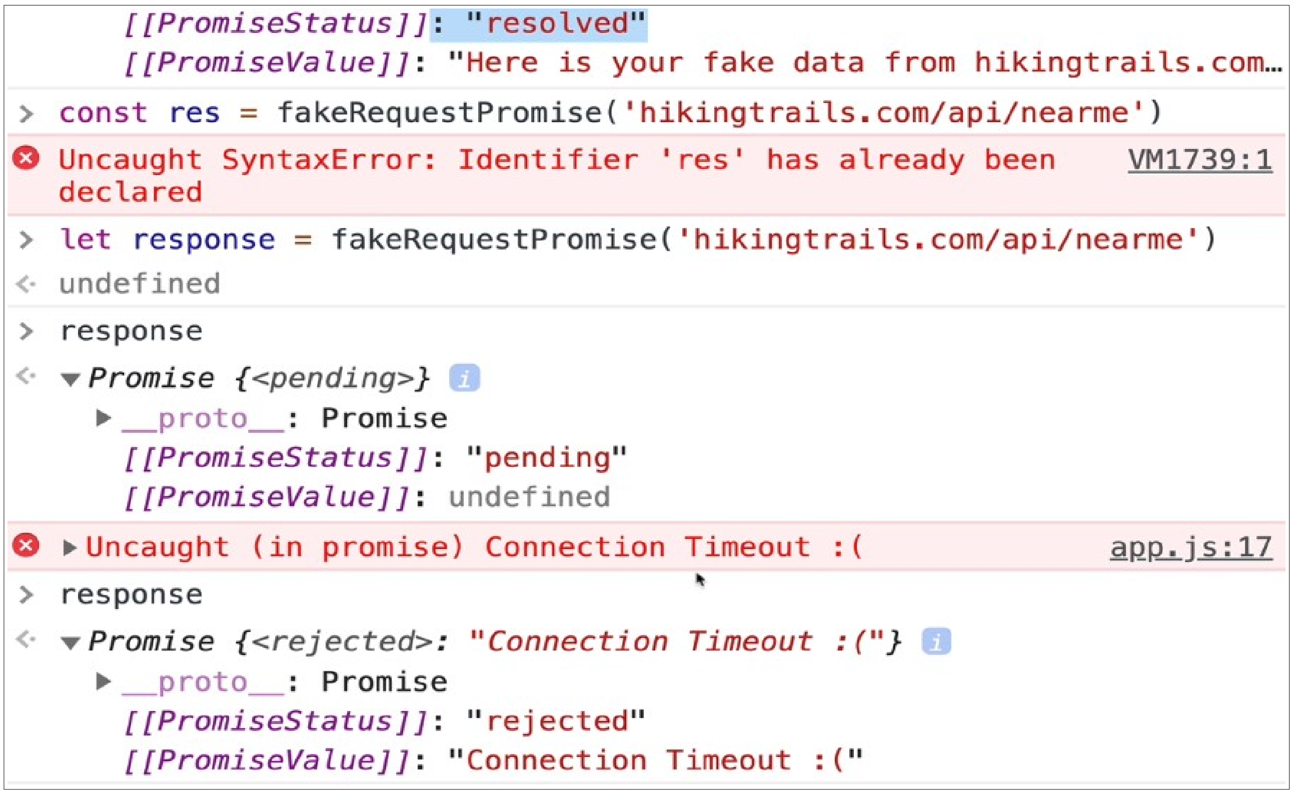


```
fakeRequestPromise('yelp.com/api/coffee/page1')
    .then(() => {
        console.log("IT WORKED!!!!!! (page1)")
        return fakeRequestPromise('yelp.com/api/coffee/page2')
    })
    .then(() => {
        console.log("IT WORKED!!!!!! (page2)")
        return fakeRequestPromise('yelp.com/api/coffee/page3')
    })
    .then(() => {
        console.log("IT WORKED!!!!!! (page3)")
    })
    .catch(() => {
        console.log("OH NO, A REQUEST FAILED!!!")
    })
```

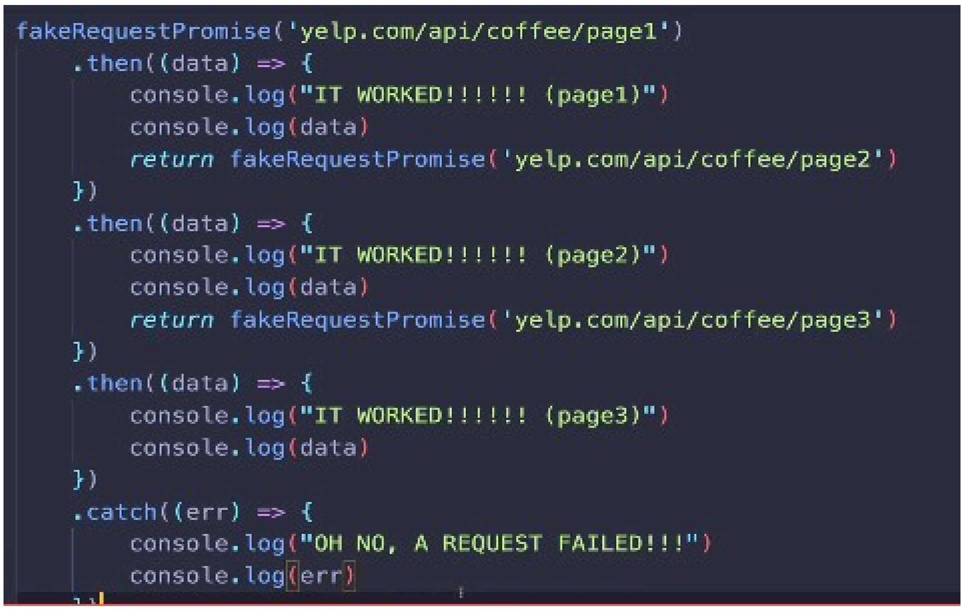
whats extra nice you can catch failure with a single catch().
So we can make no of sequential request.That only if the first one works then the second one and so on.



```
        }
    }, delay)

      const fakeRequestPromise: (url: any) => Promise<any>
onst fakeRequestPromise = (url) => {
    return new Promise((resolve, reject) => {
        const delay = Math.floor(Math.random() * (4500)) + 500;
        setTimeout(() => {
            if (delay > 4000) {
                reject('Connection Timeout :(')
            } else {
                resolve(`Here is your fake data from ${url}`)
            }
        }, delay)
    })
```



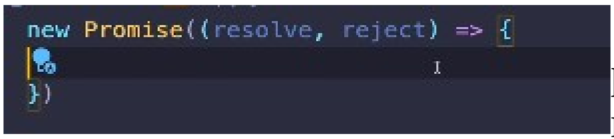
```
    [[PromiseStatus]]: "resolved"
    [[PromiseValue]]: "Here is your fake data from hikingtrails.com…
> const res = fakeRequestPromise('hikingtrails.com/api/nearme')
⊗ Uncaught SyntaxError: Identifier 'res' has already been     VM1739:1
  declared
> let response = fakeRequestPromise('hikingtrails.com/api/nearme')
<· undefined
> response
<· ▼Promise {<pending>} i
    ▶__proto__: Promise
    [[PromiseStatus]]: "pending"
    [[PromiseValue]]: undefined
⊗ ▶Uncaught (in promise) Connection Timeout :(               app.js:17
> response
<· ▼Promise {<rejected>: "Connection Timeout :("} i
    ▶__proto__: Promise
    [[PromiseStatus]]: "rejected"
    [[PromiseValue]]: "Connection Timeout :("
```



```
fakeRequestPromise('yelp.com/api/coffee/page1')
    .then((data) => {
        console.log("IT WORKED!!!!!! (page1)")
        console.log(data)
        return fakeRequestPromise('yelp.com/api/coffee/page2')
    })
    .then((data) => {
        console.log("IT WORKED!!!!!! (page2)")
        console.log(data)
        return fakeRequestPromise('yelp.com/api/coffee/page3')
    })
    .then((data) => {
        console.log("IT WORKED!!!!!! (page3)")
        console.log(data)
    })
    .catch((err) => {
        console.log("OH NO, A REQUEST FAILED!!!")
        console.log(err)
```

A promise can be rejected or resolved with a value passed to it.

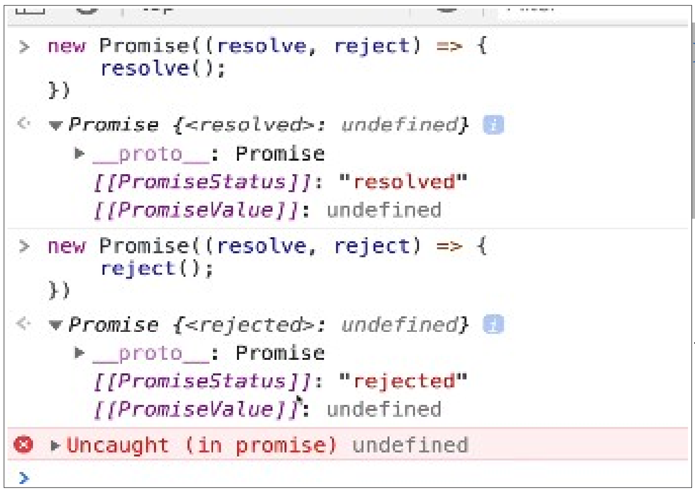
Creating our promises



```
new Promise((resolve, reject) => {

})
```

A promise expects a function with two parameters that is called resolve or reject.

These 2 parameters are actually function that we can call inside our promise.

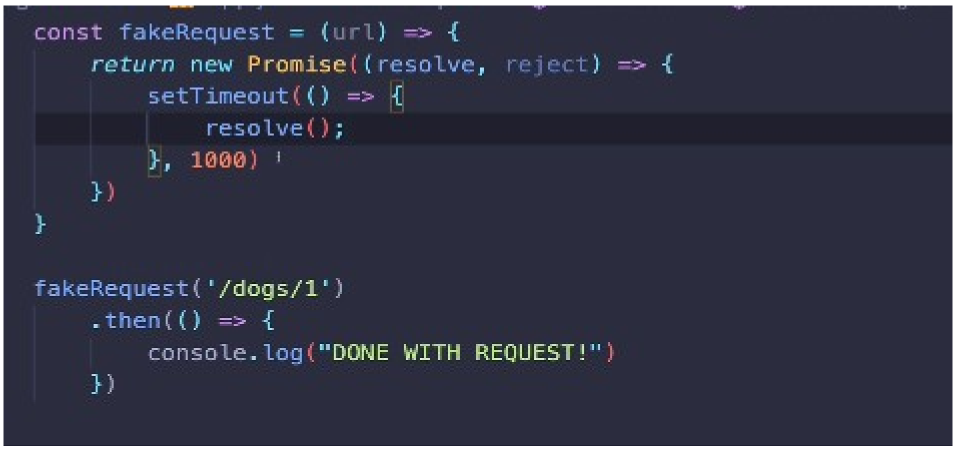


```
> new Promise((resolve, reject) => {
      resolve();
  })
<· ▼Promise {<resolved>: undefined}
    ▶__proto__: Promise
     [[PromiseStatus]]: "resolved"
     [[PromiseValue]]: undefined
> new Promise((resolve, reject) => {
      reject();
  })
<· ▼Promise {<rejected>: undefined}
    ▶__proto__: Promise
     [[PromiseStatus]]: "rejected"
     [[PromiseValue]]: undefined
⊗ ▶Uncaught (in promise) undefined
>
```

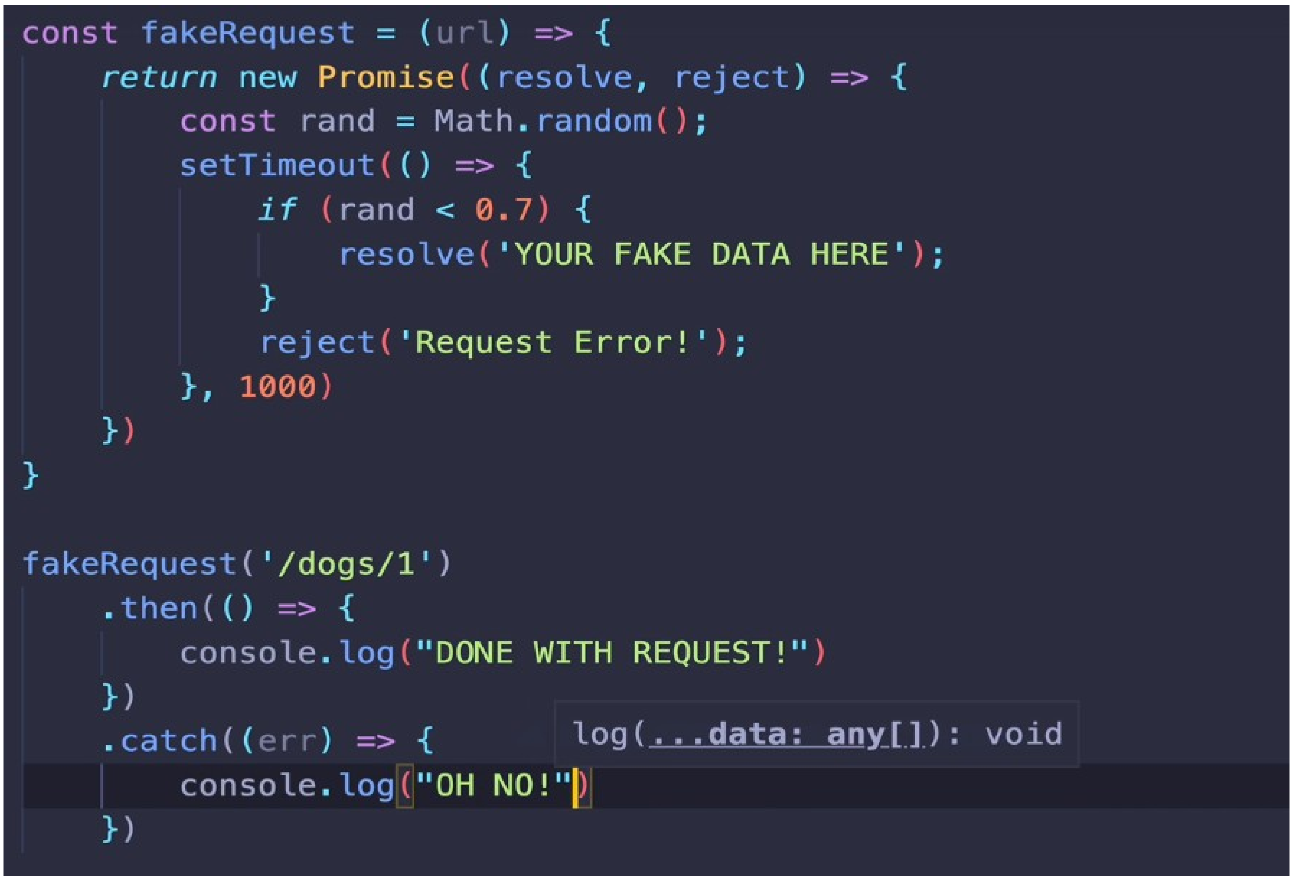
```
new Promise (
        (resolve,reject)=>{
                reject()
})
```

<<<

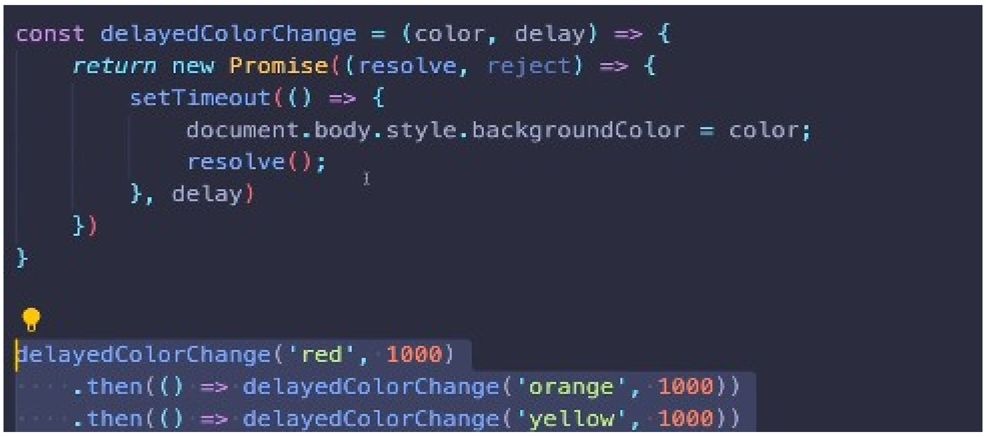
If any parameter isn't called promiseStatus: "pending"



```
const fakeRequest = (url) => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve();
        }, 1000)
    })
}

fakeRequest('/dogs/1')
    .then(() => {
        console.log("DONE WITH REQUEST!")
    })
```



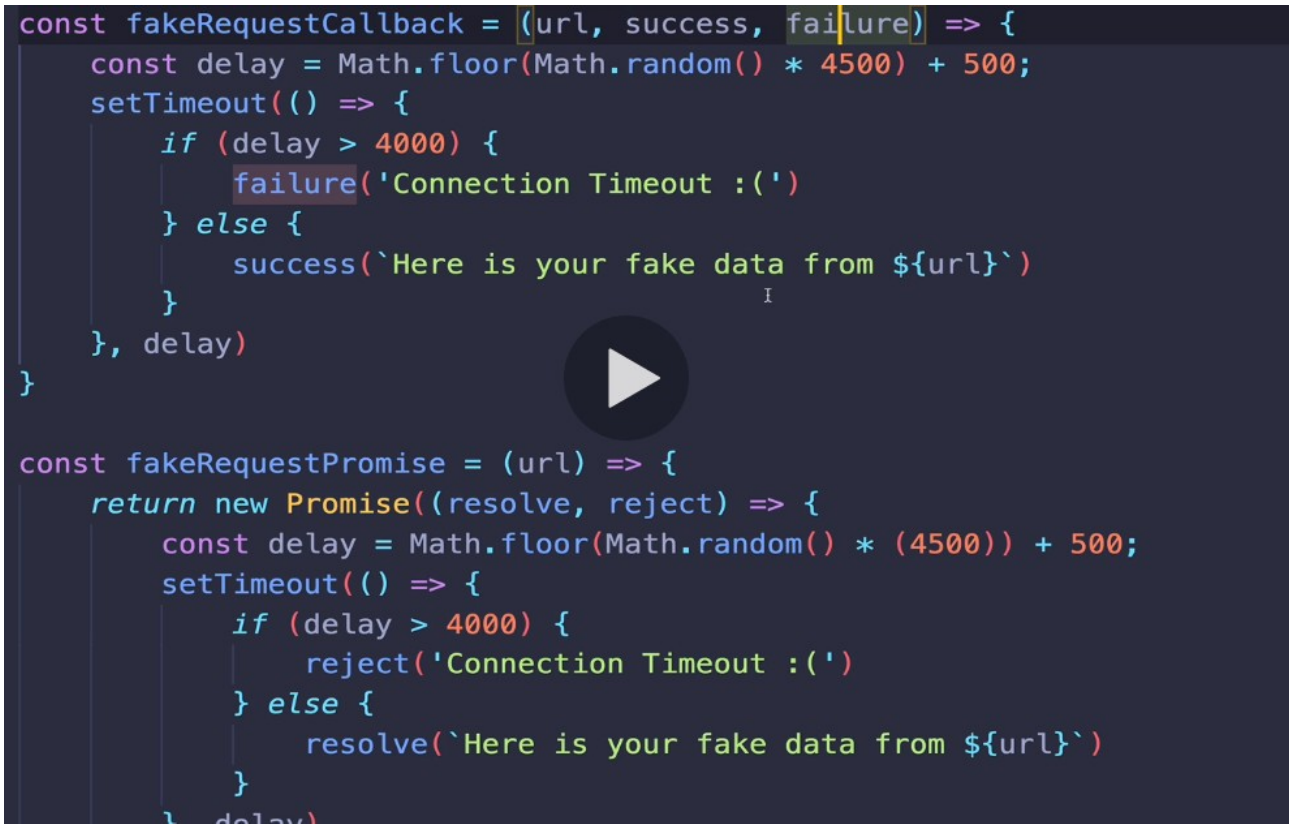
```
const fakeRequest = (url) => {
    return new Promise((resolve, reject) => {
        const rand = Math.random();
        setTimeout(() => {
            if (rand < 0.7) {
                resolve('YOUR FAKE DATA HERE');
            }
            reject('Request Error!');
        }, 1000)
    })
}

fakeRequest('/dogs/1')
    .then(() => {
        console.log("DONE WITH REQUEST!")
    })
    .catch((err) => {          log(...data: any[]): void
        console.log("OH NO!")
    })
```



```
const delayedColorChange = (color, delay) => {
    return new Promise(resolve, reject) => {
        setTimeout(() => {
            document.body.style.backgroundColor = color;
            resolve();
        }, delay)
    })
}

delayedColorChange('red', 1000)
    .then(() => delayedColorChange('orange', 1000))
    .then(() => delayedColorChange('yellow', 1000))
```

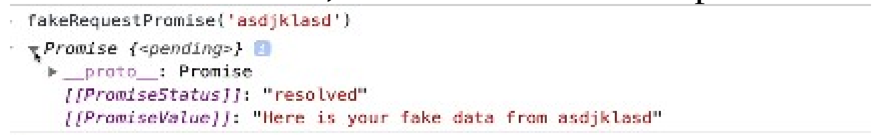
working with them when promises are returned to us, where we calling .then/.catch and passing callback, but the promises will be working with are coming from some library or some func we didn’t write.

Promises Restart



```
const fakeRequestCallback = (url, success, failure) => {
    const delay = Math.floor(Math.random() * 4500) + 500;
    setTimeout(() => {
        if (delay > 4000) {
            failure('Connection Timeout :(')
        } else {
            success(`Here is your fake data from ${url}`)
        }
    }, delay)
}

const fakeRequestPromise = (url) => {
    return new Promise((resolve, reject) => {
        const delay = Math.floor(Math.random() * (4500)) + 500;
        setTimeout(() => {
            if (delay > 4000) {
                reject('Connection Timeout :(')
            } else {
                resolve(`Here is your fake data from ${url}`)
            }
```

Well, you noticed something it doesn’t expect success or failure. It doesn’t expect callbacks at all.
If You call the func, the return value is a promise.

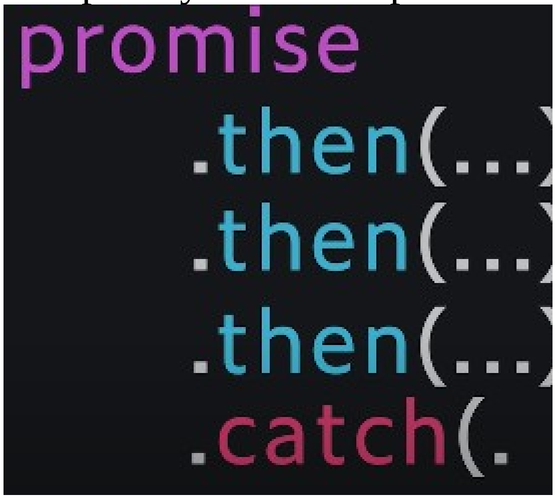


```
fakeRequestPromise('asdjklasd')
▼Promise {<pending>}
  ▶ __proto__: Promise
    [[PromiseStatus]]: "resolved"
    [[PromiseValue]]: "Here is your fake data from asdjklasd"
```

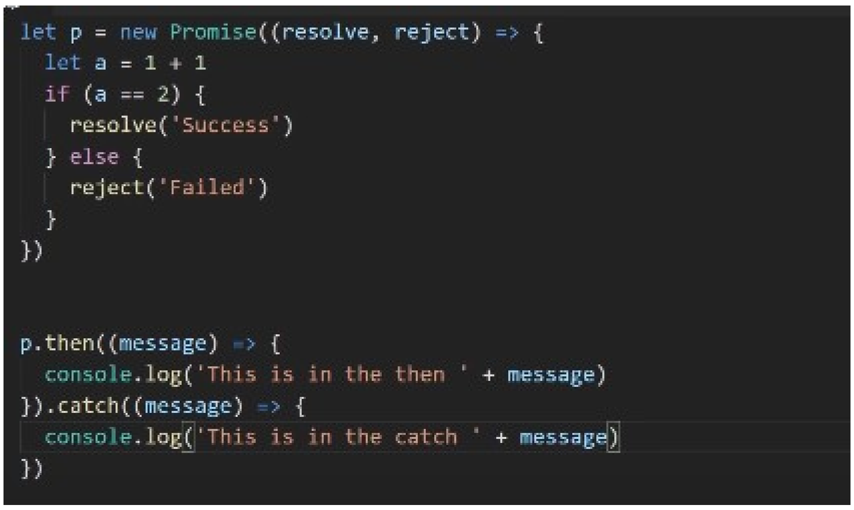
A pending promise can either be *fulfilled* with a value or *rejected* with a reason (error).

A promise represents a value that is unknown now that may become known in the future. I.O.W a Asychronous value.
For e.g
when you request a ride hailing, the driver makes a promise to pick you up.While you waiting the ride is pending.
In future all goes to plan, the driver will resolve to pick you up & take you somewhere.At which point your ride is been fulfilled. In some cases, the driver may reject your ride that means you need to catch one somewhere else.Either way the original request is finally settled.
As a Developer you wanna create a promise to represent a asynchronous value. But more often than not you will be consuming promises to use the result of asynchronous operation in your code.

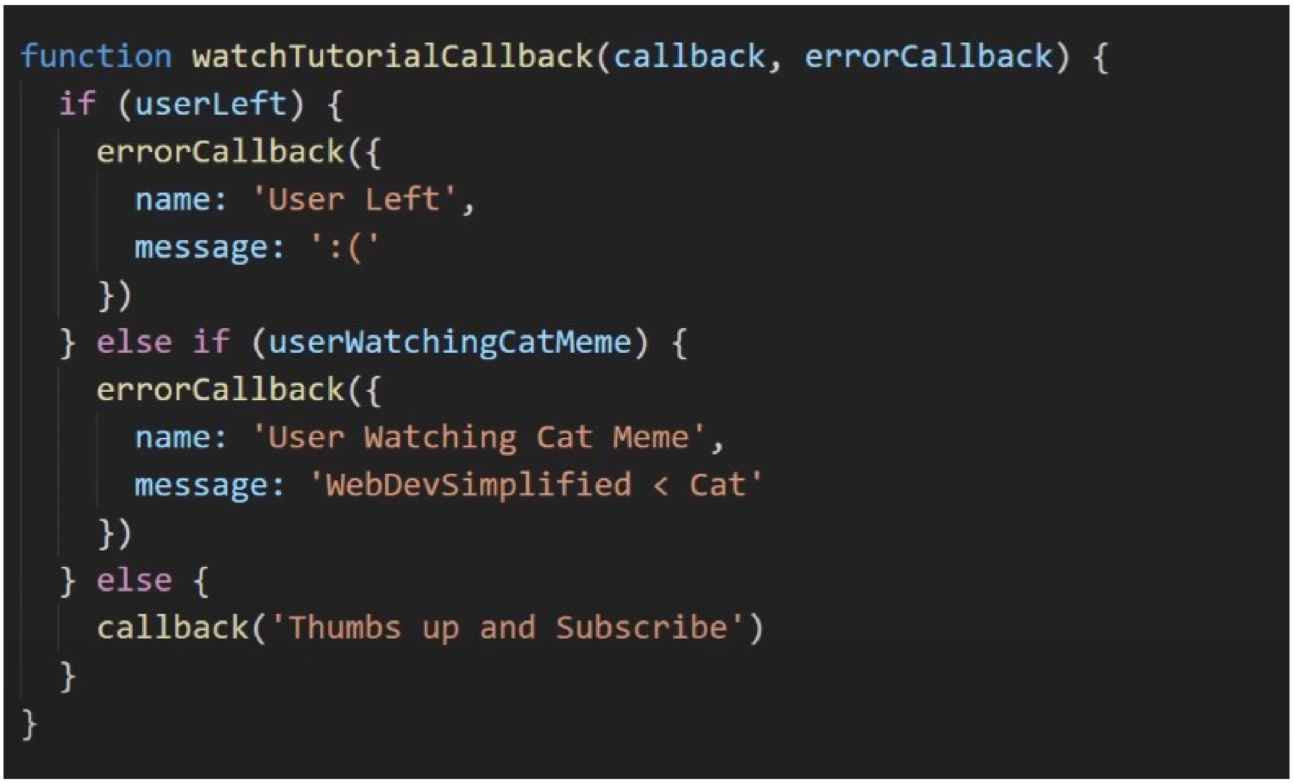
Promise create
At start, the promise is in pending state, it’s your job to define a callback func, call the executer to define when to resolve or reject the promise.
This is where you kick off your asynchronous work.
The other side, the consumer of a promise has called it’s ‘then’ method
Then: function that handles fulfillment
It’s waiting for it’s asynchronous value to be fulfilled.
when that happend we will call “then” this func with the value as an arg.
We fulfill the promise by calling resolve.
There is always a possibility of exeception in that case, we can reject the promise and send the error back to the consumer. Which can use the catch method and a entirely diff func for handling exception.
If you want to run the code no matter what, you can use finally() to handle both possibility.
All these method return promises they can be chain together to handle multiple asynchrnous operations in a row.



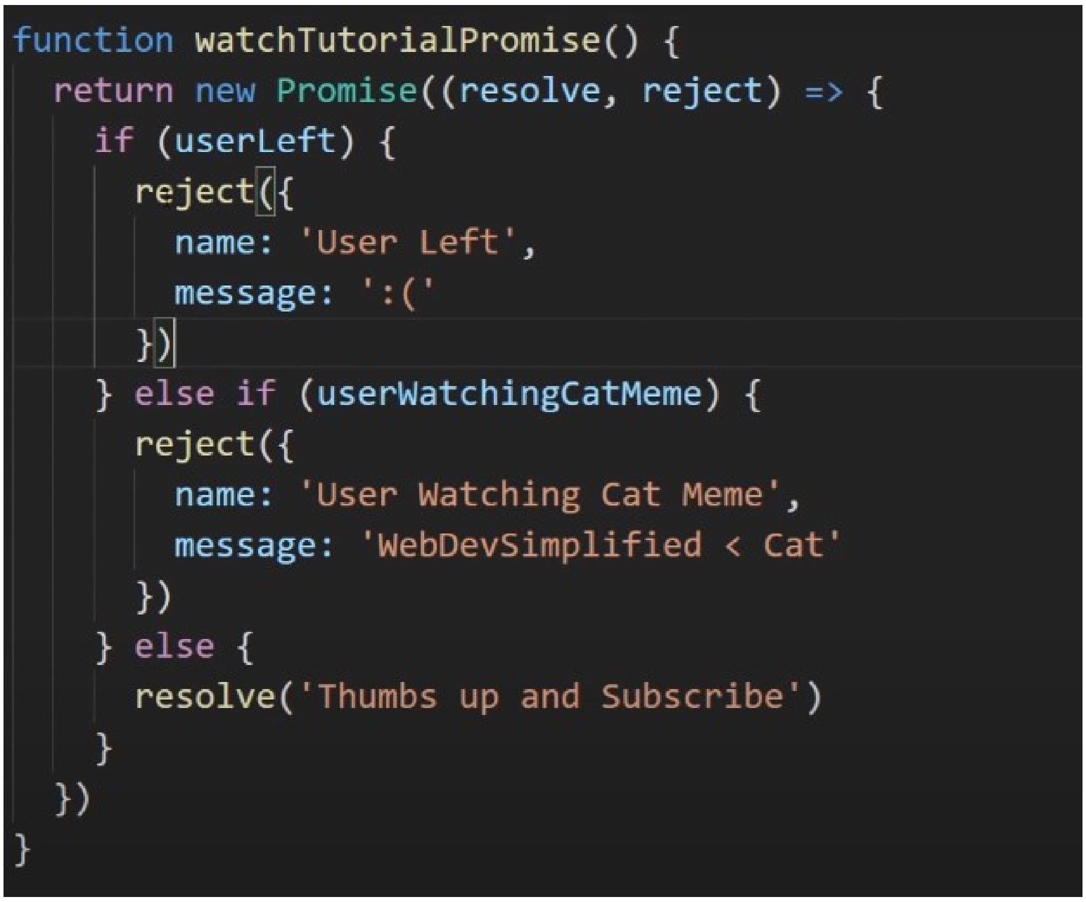
```
promise
    .then(...
    .then(...
    .then(...
    .catch(.
```



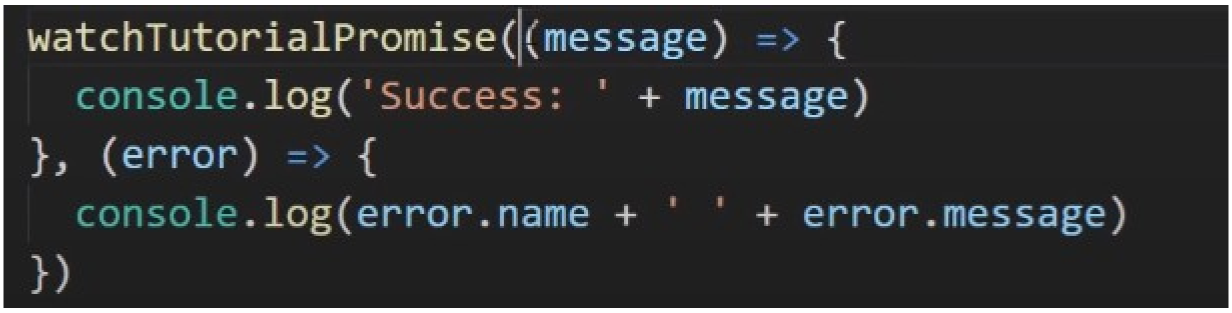
```
let p = new Promise((resolve, reject) => {
  let a = 1 + 1
  if (a == 2) {
    resolve('Success')
  } else {
    reject('Failed')
  }
})


p.then((message) => {
  console.log('This is in the then ' + message)
}).catch((message) => {
  console.log('This is in the catch ' + message)
})
```

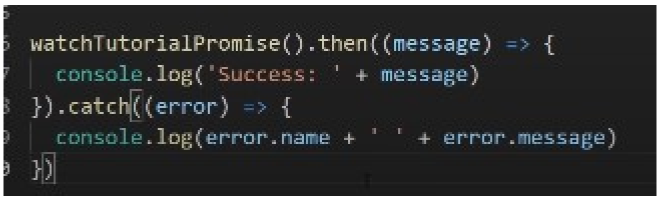


```
function watchTutorialCallback(callback, errorCallback) {
  if (userLeft) {
    errorCallback({
      name: 'User Left',
      message: ':('
    })
  } else if (userWatchingCatMeme) {
    errorCallback({
      name: 'User Watching Cat Meme',
      message: 'WebDevSimplified < Cat'
    })
  } else {
    callback('Thumbs up and Subscribe')
  }
}
```

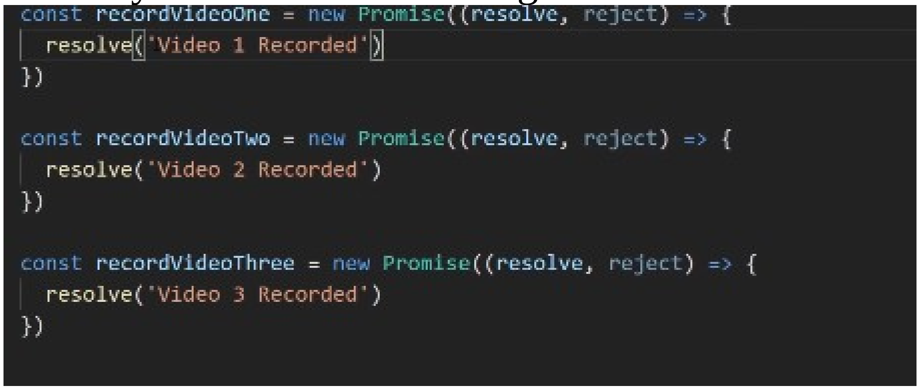


```
function watchTutorialPromise() {
  return new Promise((resolve, reject) => {
    if (userLeft) {
      reject({
        name: 'User Left',
        message: ':('
      })
    } else if (userWatchingCatMeme) {
      reject({
        name: 'User Watching Cat Meme',
        message: 'WebDevSimplified < Cat'
      })
    } else {
      resolve('Thumbs up and Subscribe')
    }
  })
}
```

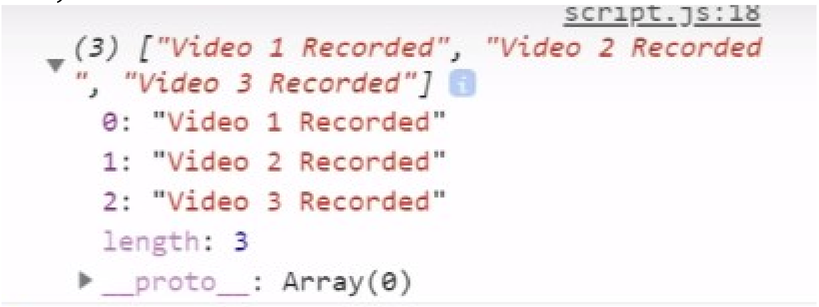

```
watchTutorialPromise((message) => {
  console.log('Success: ' + message)
}, (error) => {
  console.log(error.name + ' ' + error.message)
})
```


```
watchTutorialPromise().then((message) => {
  console.log('Success: ' + message)
}).catch((error) => {
  console.log(error.name + ' ' + error.message)
})
```

Lets say i wanna do something after i recorded all 3 videos

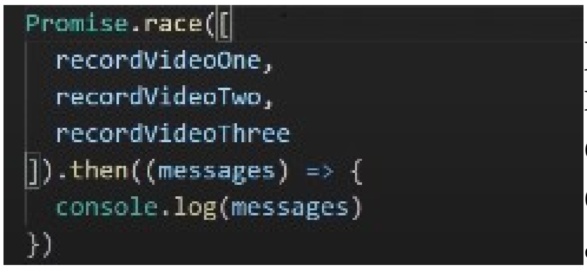

```
const recordVideoOne = new Promise((resolve, reject) => {
  resolve('Video 1 Recorded')
})

const recordVideoTwo = new Promise((resolve, reject) => {
  resolve('Video 2 Recorded')
})

const recordVideoThree = new Promise((resolve, reject) => {
  resolve('Video 3 Recorded')
})
```

To run all 3 together in parallel, i mean we dont have to wait for one to start next.
So,


```
                                              script.js:18
 ▼ (3) ["Video 1 Recorded", "Video 2 Recorded
   ", "Video 3 Recorded"]
     0: "Video 1 Recorded"
     1: "Video 2 Recorded"
     2: "Video 3 Recorded"
     length: 3
   ▶ __proto__: Array(0)
```

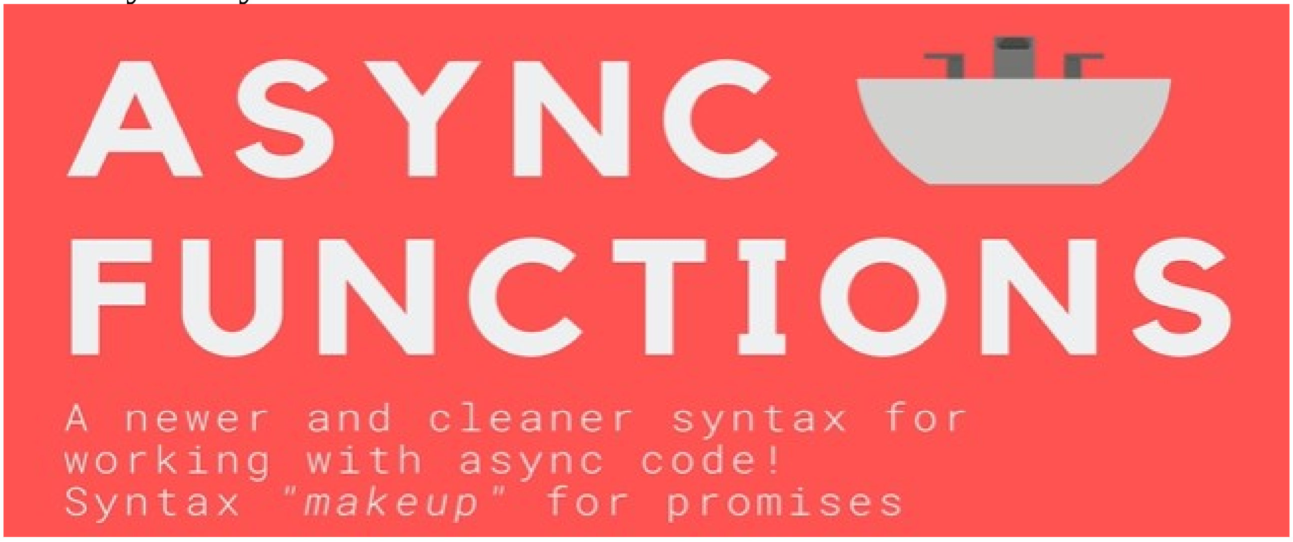
promise.all

All three Asynchronous func runs paralled. If the first one is slower than other2, the other dont need to wait for the 1st one to finish.


```
Promise.race([
  recordVideoOne,
  recordVideoTwo,
  recordVideoThree
]).then((messages) => {
  console.log(messages)
})
```

promise.race is like promise.all but it returns as soon as first one9(any) is completed, instead of waiting for everyone to complete.So it will return a single value in fulfilled callback(.then)

The Async keyword

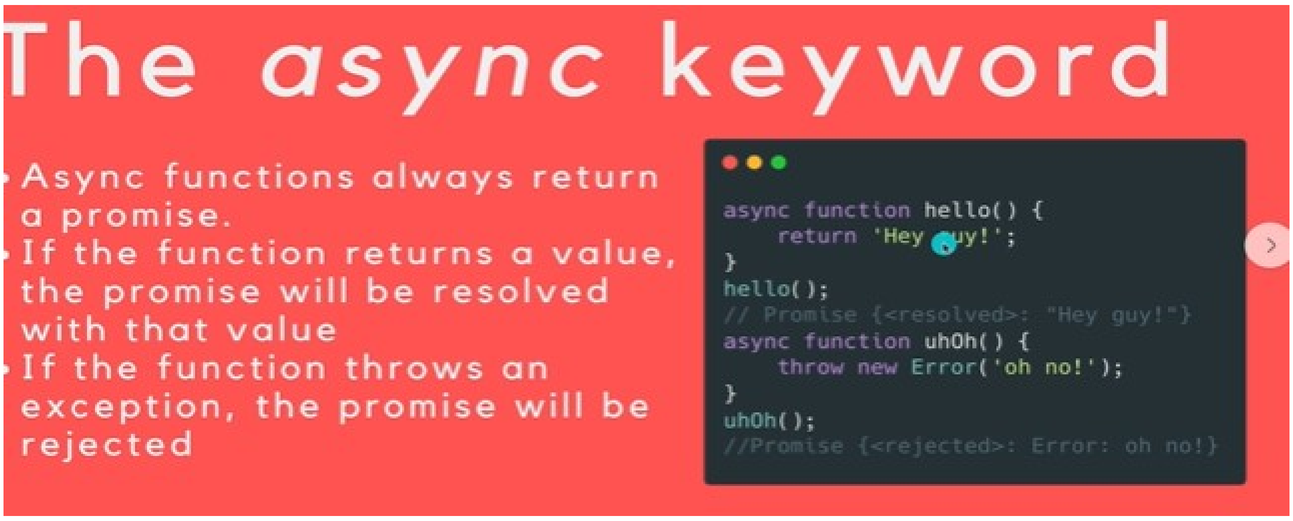


ASYNC FUNCTIONS

A newer and cleaner syntax for working with async code!
Syntax *"makeup"* for promises

2 PIECES

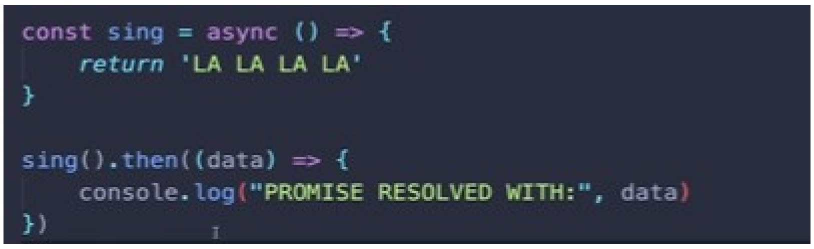
- ASYNC
- AWAIT



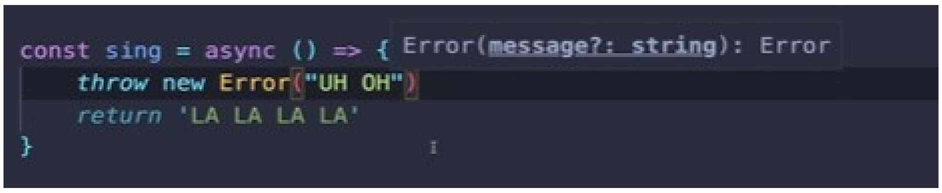
The *async* keyword

- Async functions always return a promise.
- If the function returns a value, the promise will be resolved with that value
- If the function throws an exception, the promise will be rejected

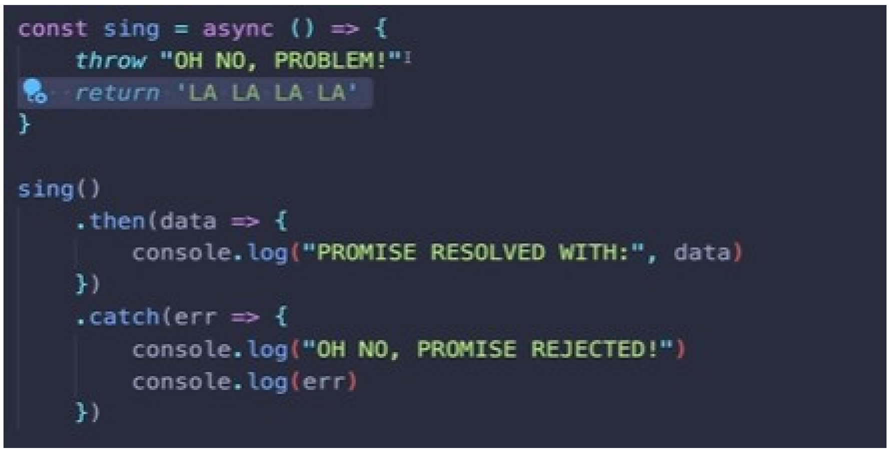
```
async function hello() {
    return 'Hey guy!';
}
hello();
// Promise {<resolved>: "Hey guy!"}
async function uhOh() {
    throw new Error('oh no!');
}
uhOh();
//Promise {<rejected>: Error: oh no!}
```



```
const sing = async () => {
    return 'LA LA LA LA'
}

sing().then((data) => {
    console.log("PROMISE RESOLVED WITH:", data)
})
```

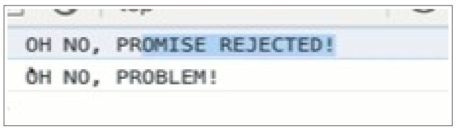


```
const sing = async () => { Error(message?: string): Error
    throw new Error("UH OH")
    return 'LA LA LA LA'
}
```

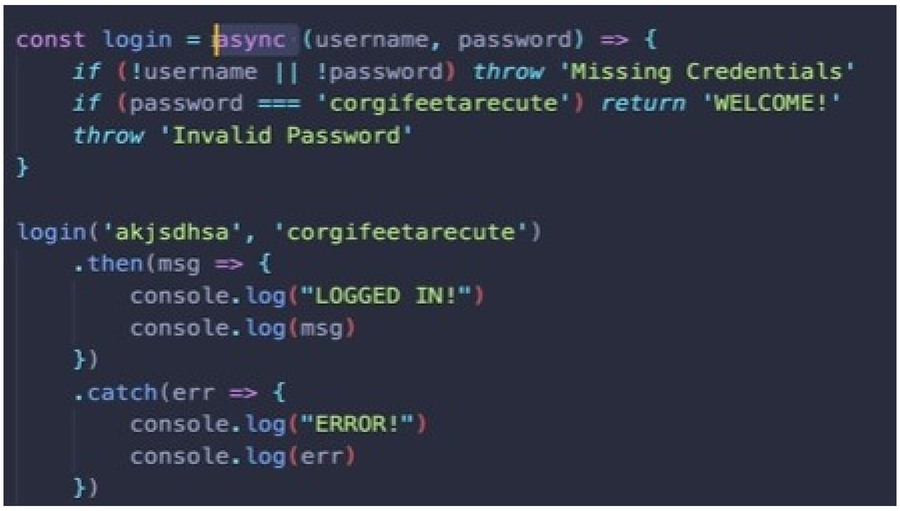


```
const sing = async () => {
    throw "OH NO, PROBLEM!"
    return 'LA LA LA LA'
}

sing()
    .then(data => {
        console.log("PROMISE RESOLVED WITH:", data)
    })
    .catch(err => {
        console.log("OH NO, PROMISE REJECTED!")
        console.log(err)
    })
```

>>>

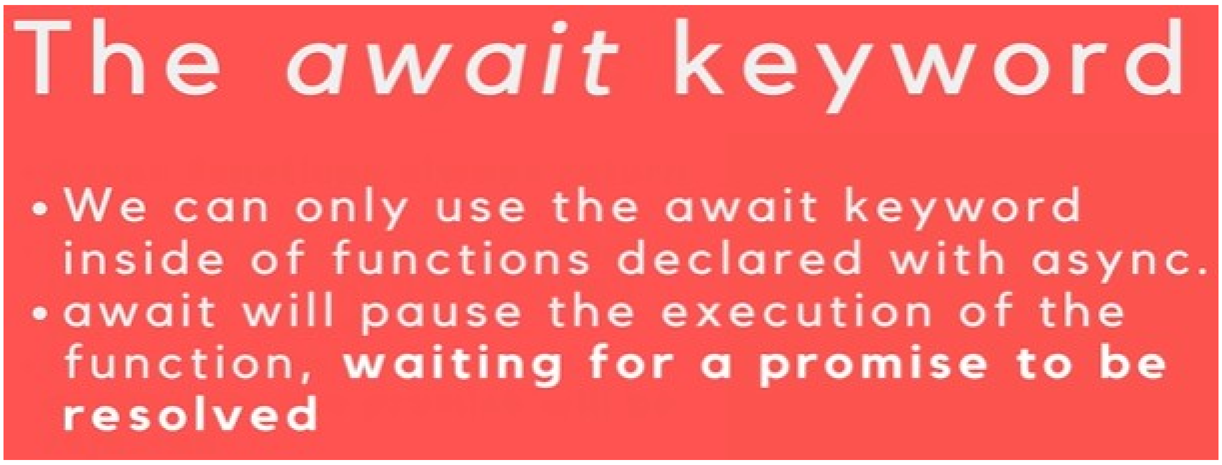


```
OH NO, PROMISE REJECTED!
OH NO, PROBLEM!
```



```
const login = async (username, password) => {
    if (!username || !password) throw 'Missing Credentials'
    if (password === 'corgifeetarecute') return 'WELCOME!'
    throw 'Invalid Password'
}

login('akjsdhsa', 'corgifeetarecute')
    .then(msg => {
        console.log("LOGGED IN!")
        console.log(msg)
    })
    .catch(err => {
        console.log("ERROR!")
        console.log(err)
    })
```

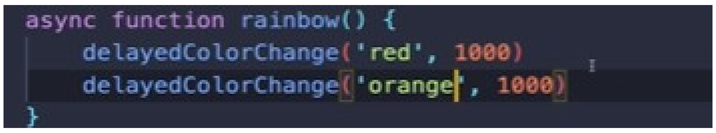
2 PIECES
• ASYNC
• AWAIT



The *await* keyword

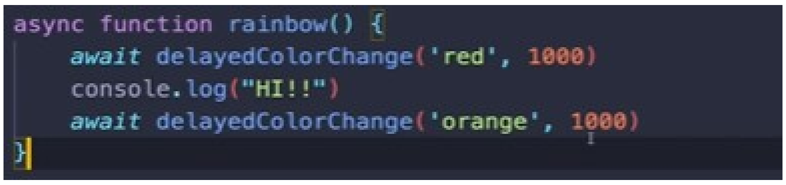
- We can only use the await keyword inside of functions declared with async.
- await will pause the execution of the function, **waiting for a promise to be resolved**

Above, await is by far common use with async.



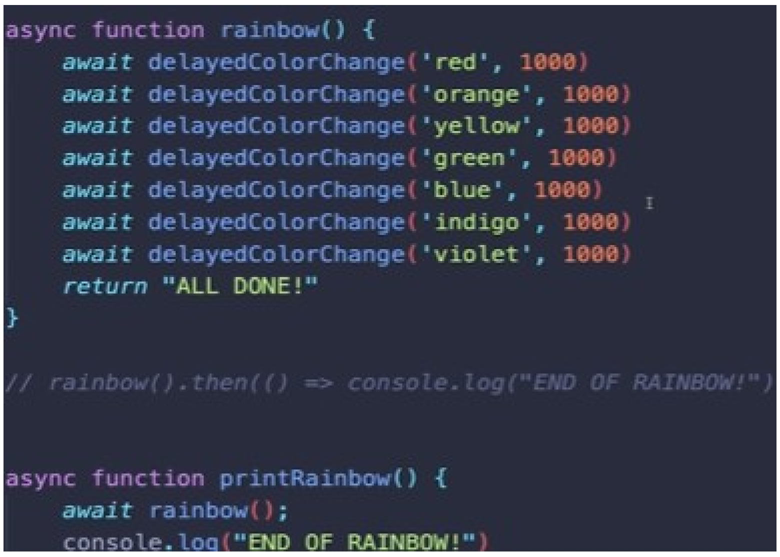
```
async function rainbow() {
    delayedColorChange('red', 1000)
    delayedColorChange('orange', 1000)
}
```

Above wont work, cuz both happen at the same time.

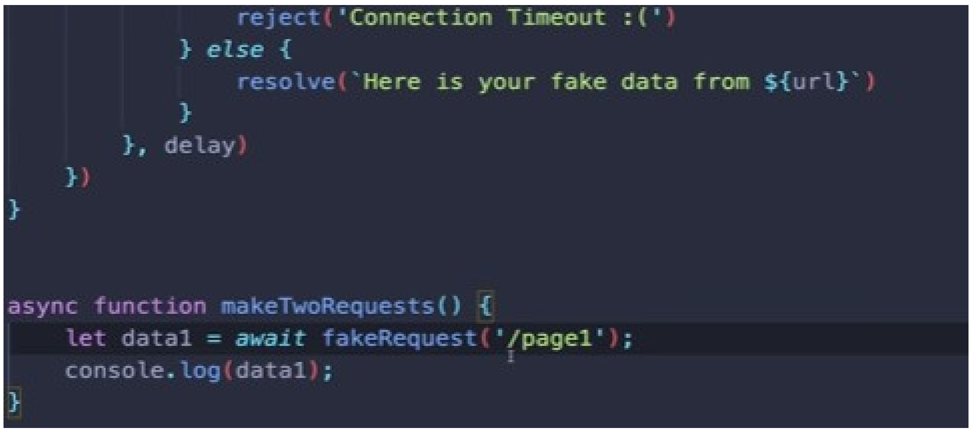


```
async function rainbow() {
    await delayedColorChange('red', 1000)
    console.log("HI!!")
    await delayedColorChange('orange', 1000)
}
```

If i we use await keyword, its gonna wait for a promise to be resolved.

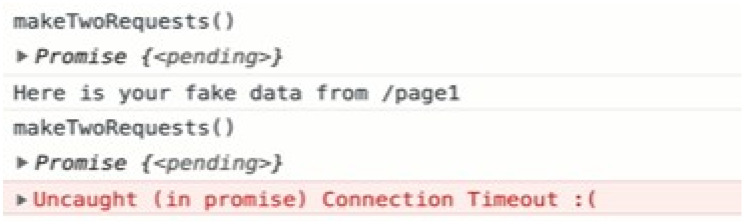


```
async function rainbow() {
    await delayedColorChange('red', 1000)
    await delayedColorChange('orange', 1000)
    await delayedColorChange('yellow', 1000)
    await delayedColorChange('green', 1000)
    await delayedColorChange('blue', 1000)
    await delayedColorChange('indigo', 1000)
    await delayedColorChange('violet', 1000)
    return "ALL DONE!"
}

// rainbow().then(() => console.log("END OF RAINBOW!"))


async function printRainbow() {
    await rainbow();
    console.log("END OF RAINBOW!")
```

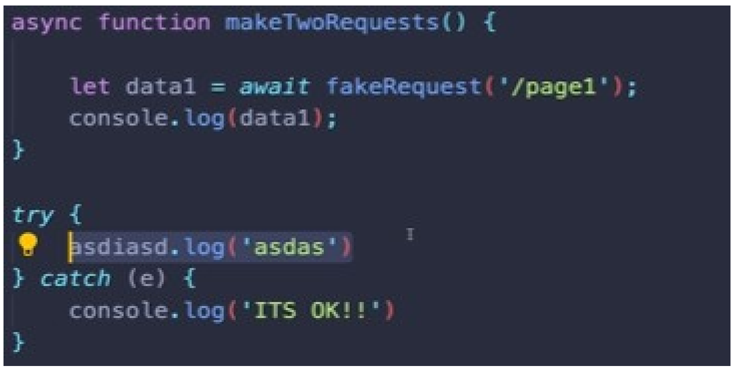


```
                reject('Connection Timeout :(')
            } else {
                resolve(`Here is your fake data from ${url}`)
            }
        }, delay)
    })
}


async function makeTwoRequests() {
    let data1 = await fakeRequest('/page1');
    console.log(data1);
}
```

>>>



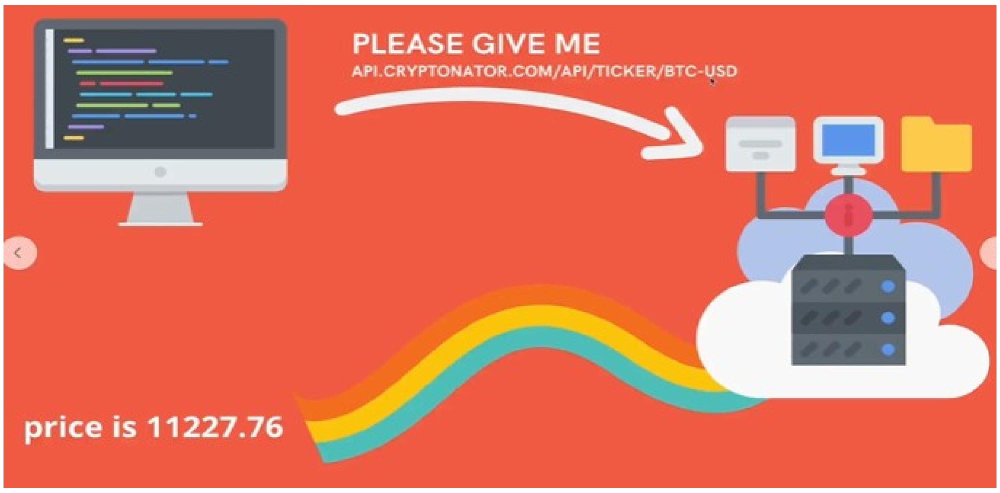
```
makeTwoRequests()
▸ Promise {<pending>}
Here is your fake data from /page1
makeTwoRequests()
▸ Promise {<pending>}
▸ Uncaught (in promise) Connection Timeout :(
```



```
async function makeTwoRequests() {

    let data1 = await fakeRequest('/page1');
    console.log(data1);
}

try {
    asdiasd.log('asdas')
} catch (e) {
    console.log('ITS OK!!')
}
```

AJAX and API

Making request to load information or to send information to save something basically behind the scenes on a given website or on a given application, seamlessly behind the scenes interacting with a server somewhere.
In Reddit, you scroll down, more and more content is loaded and displayed .Infinite scroll
In devtools > network>
you'll see any new request has been made on this page or any new info that has been loaded.
Alot of request has been made to reddit server to load new post.
Ajax refers to making request on page while it's already been loaded or after it's been loaded, making request behind the scenes.
It allow us to do things that users in the past would have to click around and you know go to a new page.
For e.g i can do live search.
Whats happening is as i am typing request are being made to a server somewhere.
Alot of time when we are working with js what we want is just data.
So if i want an app that would show you the current bitcoin price every 2mins or 30s whatever. I dont have to refresh the page. I Could use a setTimeout or make a request to this Api below. This is an endpoint that i can make request on, it response back in json
Using js we can load data or fetch info or send data to save progress behind the scene without refresh.



Intro to Api

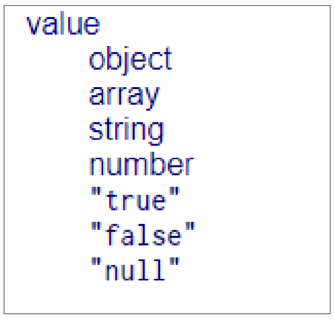
We know the request response cycle and for typical webpage we are loading html+css+js .
When we make request using js, when we are making ajax request. We are looking for barebone of info. We dont want the html or Css or Js.
We just want the data.This is where API comes in.
It refers to any interface where one software communicates with another software.
Web Api are interface that are Web based.
Companies with api have endpoints, this endpoint response with some info for code or other peice of software to consume. A webapi is a portal to diff application or db somewhere or dataset.
A bitcoin app, every 2min to get new price of bitcoin and display it. Using js update the dom and selcect the element and update the current price.
Api is an interface for our applications.

Json.org **json_formater** **json_mdn**

JSON
• Java
• Script
• Object
• Notation

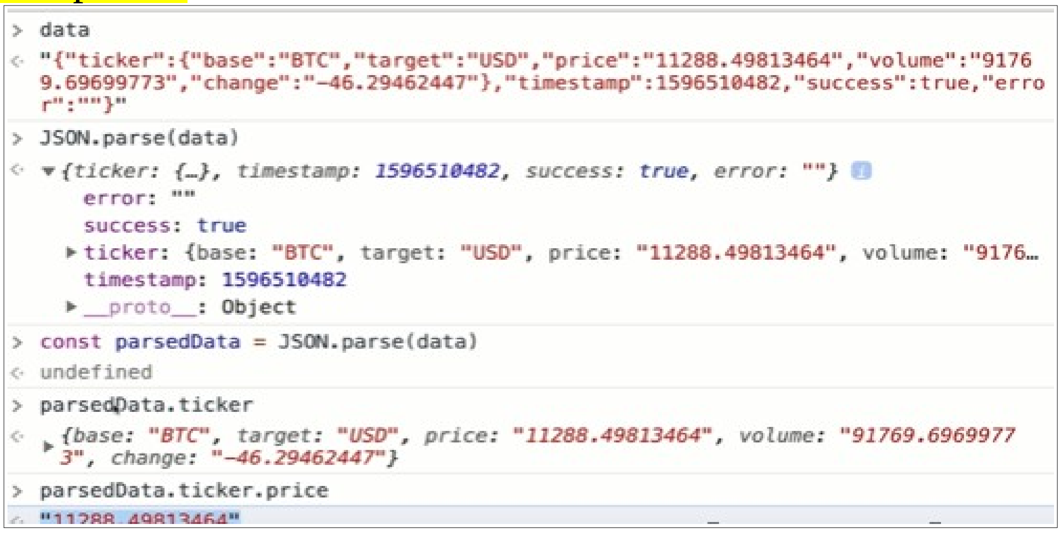
```
{
  "squadName": "Super hero squad",
  "homeTown": "Metro City",
  "formed": 2016,
  "secretBase": "Super tower",
  "active": true,
  "members": [
    "Molecule Man",
      "Madame Uppercut",
      "Eternal Flame"
  ]
}
```

Note:
-every key has to be a double quoted string.
-Json doesn't have undefined, undefined turn into a String.



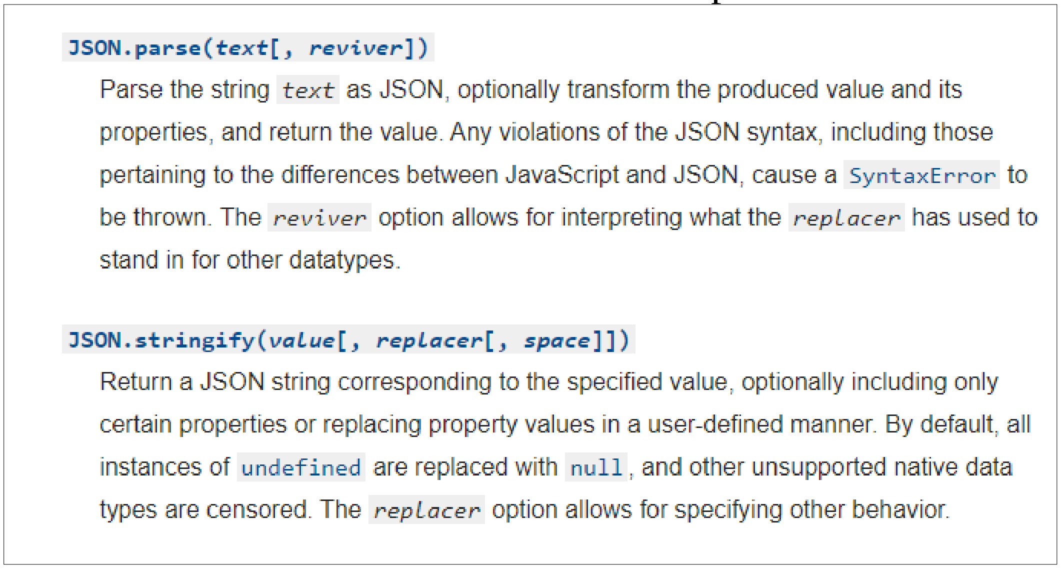
```
value
    object
    array
    string
    number
    "true"
    "false"
    "null"
```

-Json works with other lang such as python, ruby.... and not just js. We still work with api that is json api. They all have their own way of parsing json and turning into python code or js code..
-When we make request to an Api what we get is a giant string.
Json parser



```
> data
< "{"ticker":{"base":"BTC","target":"USD","price":"11288.49813464","volume":"9176
  9.69699773","change":"-46.29462447"},"timestamp":1596510482,"success":true,"erro
  r":""}"
> JSON.parse(data)
< ▾{ticker: {…}, timestamp: 1596510482, success: true, error: ""}
      error: ""
      success: true
    ▸ticker: {base: "BTC", target: "USD", price: "11288.49813464", volume: "9176…
      timestamp: 1596510482
    ▸__proto__: Object
> const parsedData = JSON.parse(data)
< undefined
> parsedData.ticker
< ▸{base: "BTC", target: "USD", price: "11288.49813464", volume: "91769.6969977
   3", change: "-46.29462447"}
> parsedData.ticker.price
< "11288.49813464"
```

-If we have info that is in js object and we wanna turn it into json. This is useful when we want to send info to an api

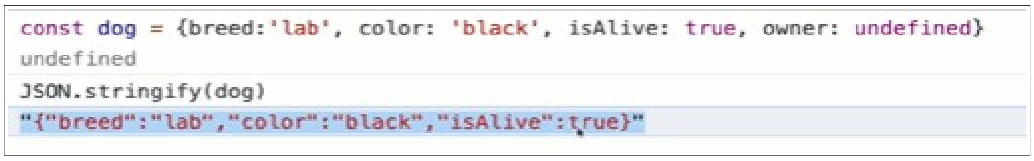


`JSON.parse(text[, reviver])`

Parse the string `text` as JSON, optionally transform the produced value and its properties, and return the value. Any violations of the JSON syntax, including those pertaining to the differences between JavaScript and JSON, cause a `SyntaxError` to be thrown. The `reviver` option allows for interpreting what the `replacer` has used to stand in for other datatypes.

`JSON.stringify(value[, replacer[, space]])`

Return a JSON string corresponding to the specified value, optionally including only certain properties or replacing property values in a user-defined manner. By default, all instances of `undefined` are replaced with `null`, and other unsupported native data types are censored. The `replacer` option allows for specifying other behavior.

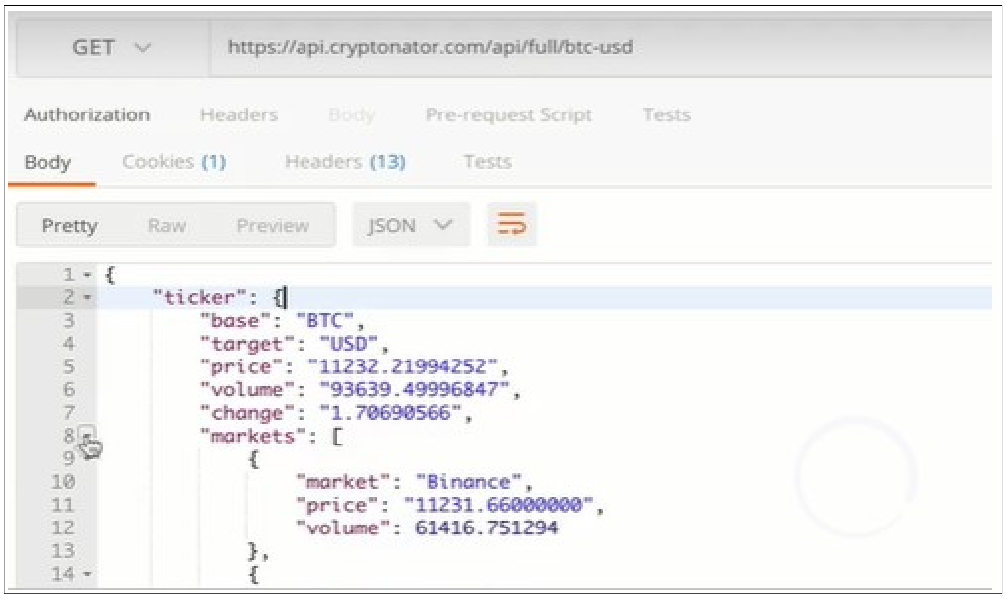


```
const dog = {breed:'lab', color: 'black', isAlive: true, owner: undefined}
undefined
JSON.stringify(dog)
"{"breed":"lab","color":"black","isAlive":true}"
```

Postman

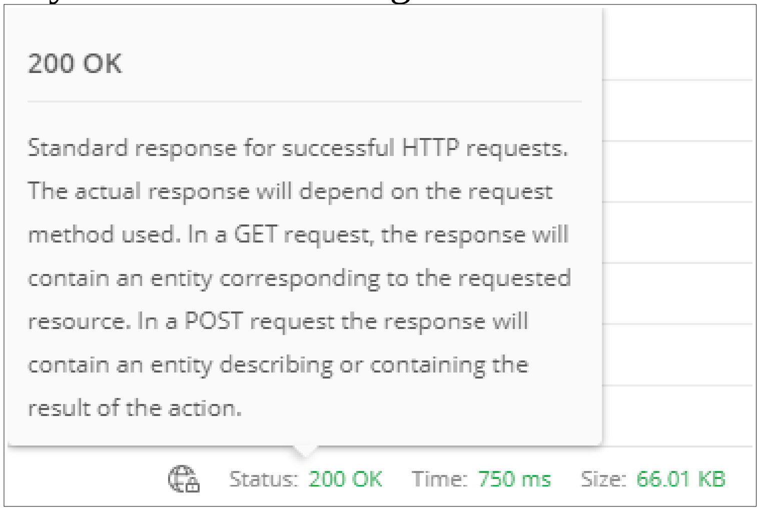
You can do by going through devtools>network.
We can see the response and header but this isn't best way.
Doesn't allow save request and easily modify or redo request.
Post man is a great tool for testing api.
-So much what you do when you work with an Api is reading the docs and tesitng it out and verifying things work as the way you expect and then writing code to make those request.

Http methods:
Get: retrieving or getting info or asking for smthg.
Post: request which we use for sending data that we intent to somehow impact the server-side (insert,update)



```
{
    "ticker": {
        "base": "BTC",
        "target": "USD",
        "price": "11232.21994252",
        "volume": "93639.49996847",
        "change": "1.70690566",
        "markets": [
            {
                "market": "Binance",
                "price": "11231.66000000",
                "volume": 61416.751294
            },
            {
```

Body is the intended msg and status code

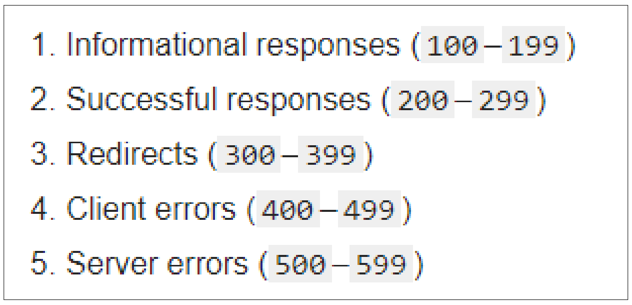


200 OK

Standard response for successful HTTP requests. The actual response will depend on the request method used. In a GET request, the response will contain an entity corresponding to the requested resource. In a POST request the response will contain an entity describing or containing the result of the action.

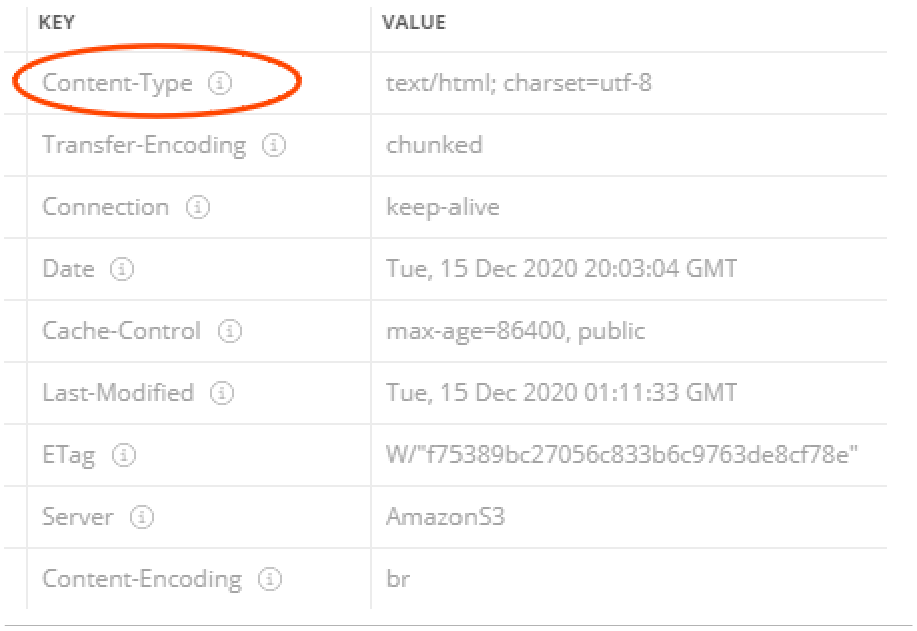
Status: 200 OK Time: 750 ms Size: 66.01 KB

[HTTP Status code]()



1. Informational responses (`100` – `199`)
2. Successful responses (`200` – `299`)
3. Redirects (`300` – `399`)
4. Client errors (`400` – `499`)
5. Server errors (`500` – `599`)

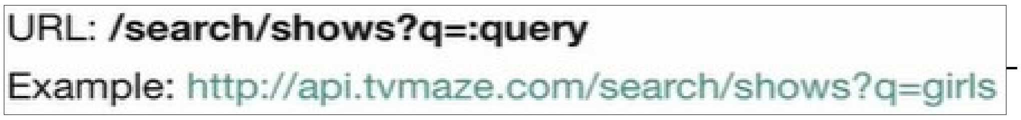
Headers: are bunch of key-value pairs. They are kind of metadata for the response and also for the request.



KEY	VALUE
Content-Type	text/html; charset=utf-8
Transfer-Encoding	chunked
Connection	keep-alive
Date	Tue, 15 Dec 2020 20:03:04 GMT
Cache-Control	max-age=86400, public
Last-Modified	Tue, 15 Dec 2020 01:11:33 GMT
ETag	W/"f75389bc27056c833b6c9763de8cf78e"
Server	AmazonS3
Content-Encoding	br

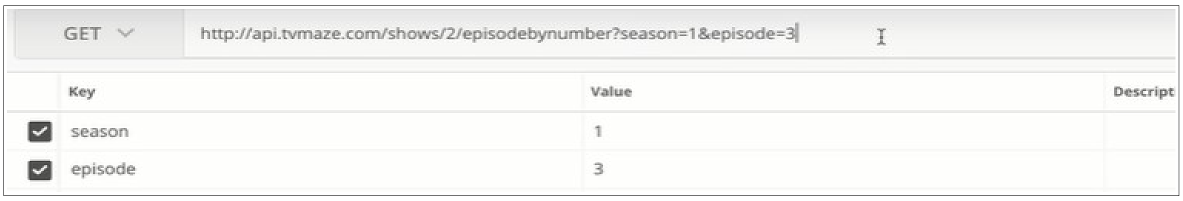
Content-Type:text/html	for web resource
Content-Type:application/json	response from an api

Query String and headers


URL: **/search/shows?q=:query**
Example: http://api.tvmaze.com/search/shows?q=girls

Whenever you see a colon in a URL in Api documentation, it’s a way of saying this is a variable, It’s something that you provide.
?q=<something> <!-- query string

/search/shows which is relative to base url.

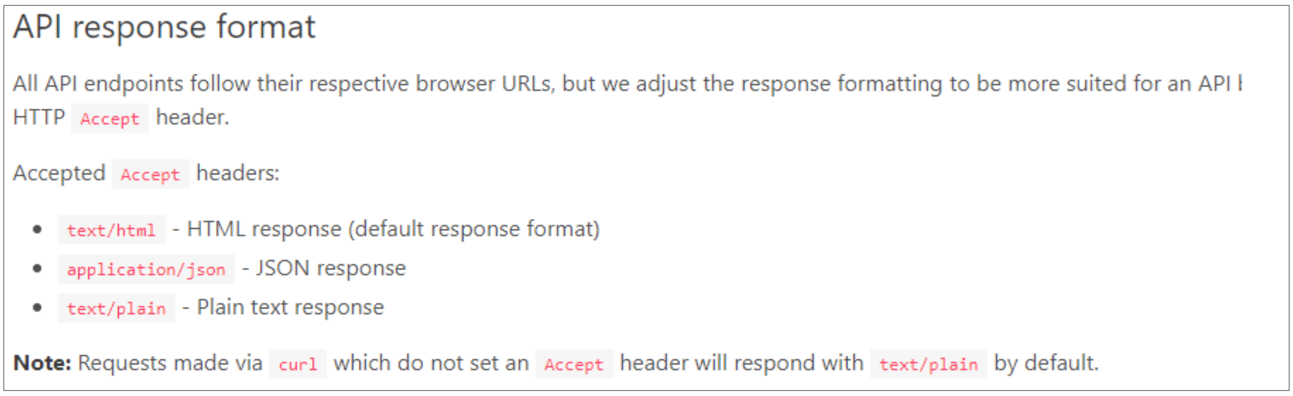

GET http://api.tvmaze.com/shows/2/episodebynumber?season=1&episode=3

	Key	Value	Descript
✓	season	1	
✓	episode	3	

The Api is listening for query string for e.g season it works on. Else which it doesn’t works on, are ignored.

icanhazdadjoke.com/api

Website


API response format

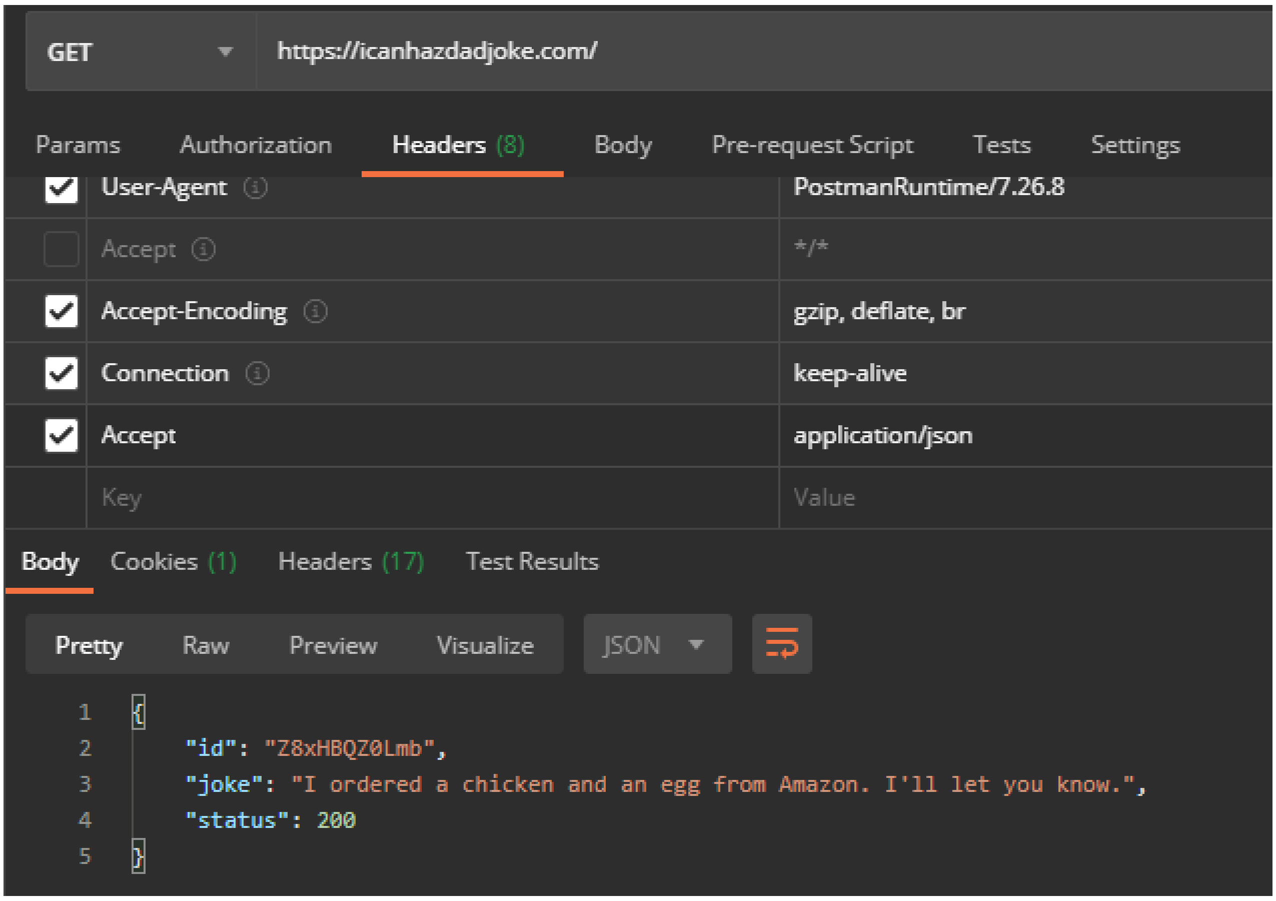
All API endpoints follow their respective browser URLs, but we adjust the response formatting to be more suited for an API l
HTTP `Accept` header.

Accepted `Accept` headers:

- `text/html` - HTML response (default response format)
- `application/json` - JSON response
- `text/plain` - Plain text response

Note: Requests made via `curl` which do not set an `Accept` header will respond with `text/plain` by default.

The default it’s gonna response with is html.
Here, we can send our own header

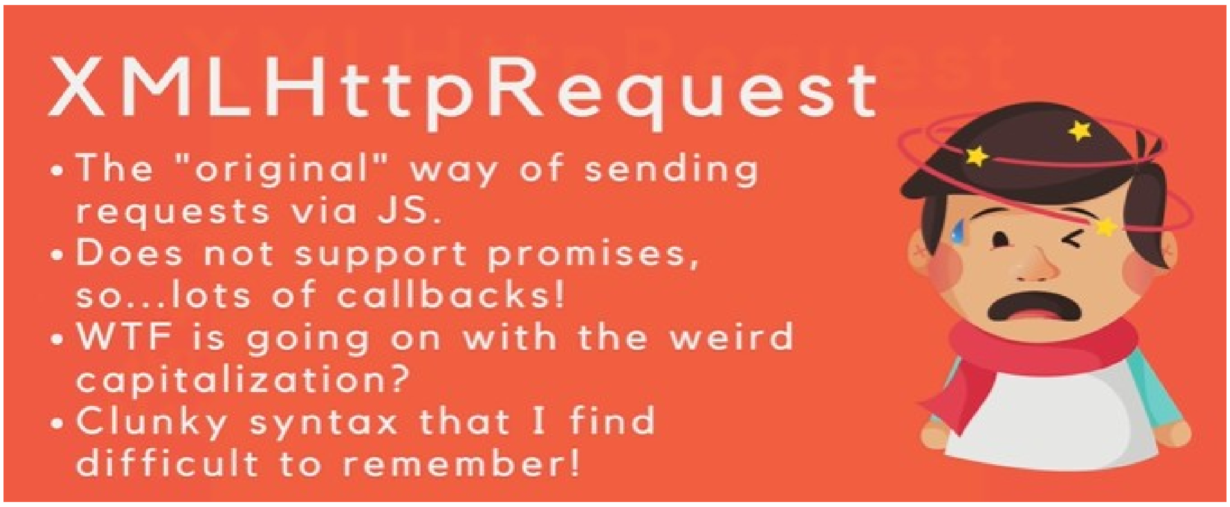


	Key	Value
✓	User-Agent	PostmanRuntime/7.26.8
	Accept	*/*
✓	Accept-Encoding	gzip, deflate, br
✓	Connection	keep-alive
✓	Accept	application/json
	Key	Value

```
{
    "id": "Z8xHBQZ0Lmb",
    "joke": "I ordered a chicken and an egg from Amazon. I'll let you know.",
    "status": 200
}
```

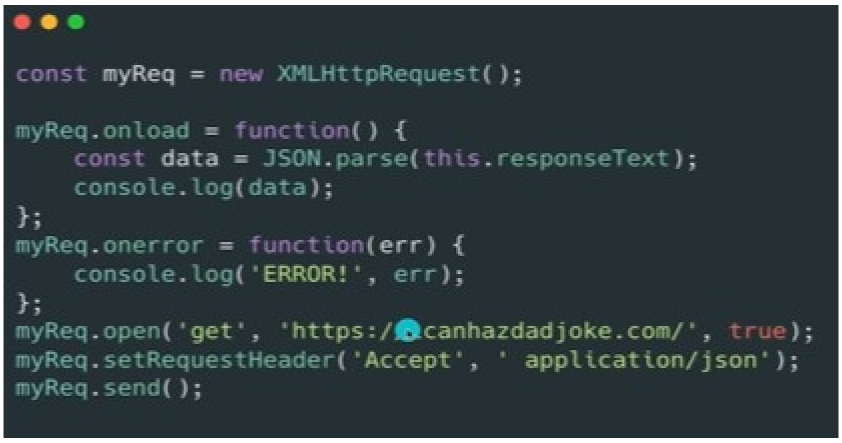
put the endpoint in “enter request url” not

icanhazdadjoke.com/api

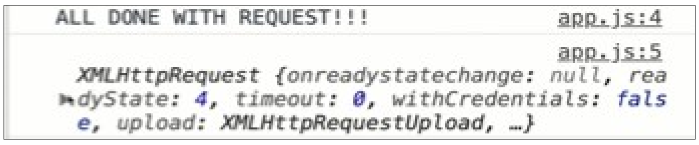


XMLHttpRequest

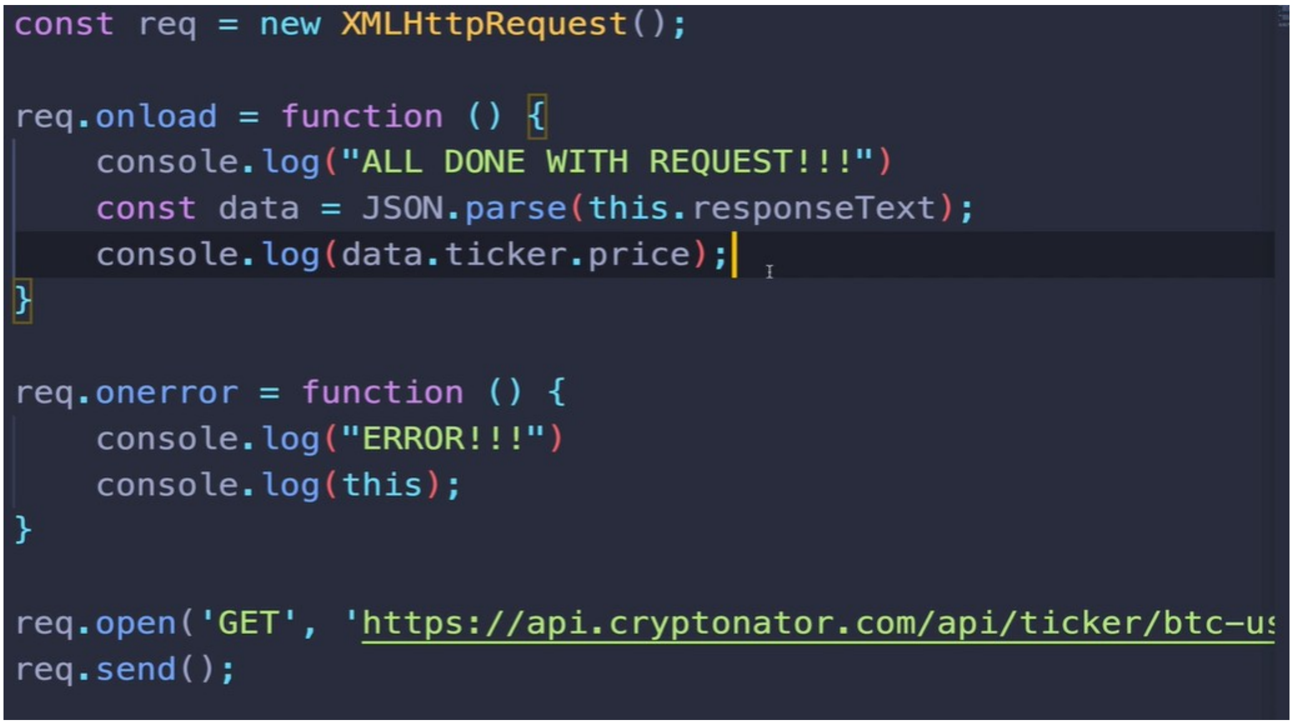
- The "original" way of sending requests via JS.
- Does not support promises, so...lots of callbacks!
- WTF is going on with the weird capitalization?
- Clunky syntax that I find difficult to remember!



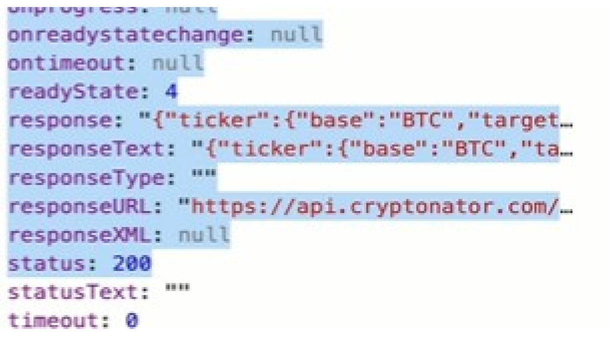
```
const myReq = new XMLHttpRequest();

myReq.onload = function() {
    const data = JSON.parse(this.responseText);
    console.log(data);
};
myReq.onerror = function(err) {
    console.log('ERROR!', err);
};
myReq.open('get', 'https://icanhazdadjoke.com/', true);
myReq.setRequestHeader('Accept', ' application/json');
myReq.send();
```



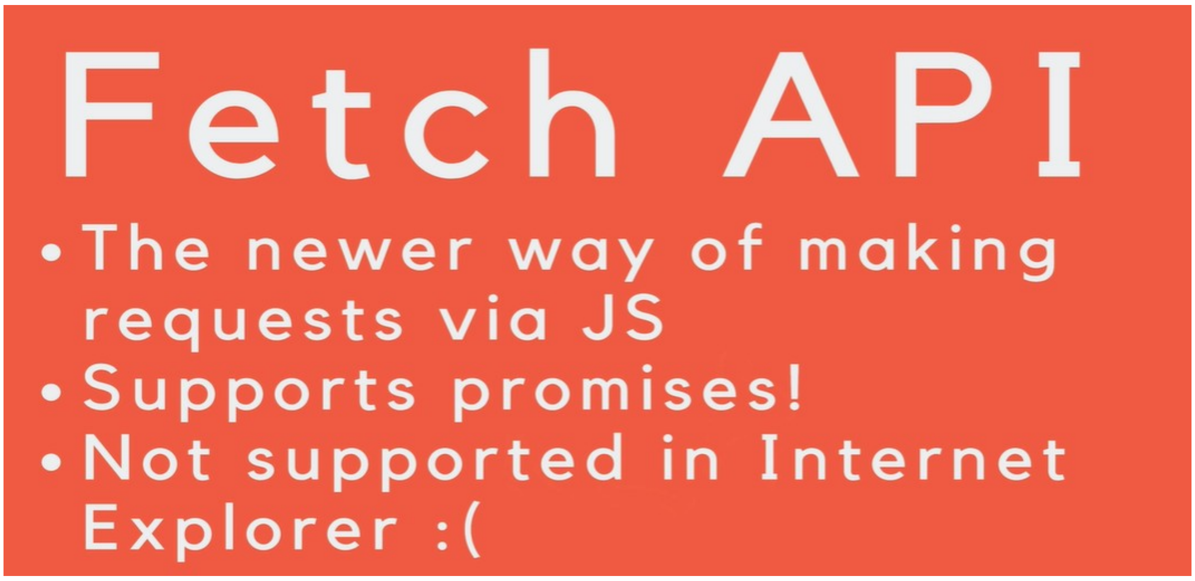
```
ALL DONE WITH REQUEST!!!                          app.js:4
                                                  app.js:5
  XMLHttpRequest {onreadystatechange: null, rea
▸dyState: 4, timeout: 0, withCredentials: fals
  e, upload: XMLHttpRequestUpload, …}
```



```
const req = new XMLHttpRequest();

req.onload = function () {
    console.log("ALL DONE WITH REQUEST!!!")
    const data = JSON.parse(this.responseText);
    console.log(data.ticker.price);
}

req.onerror = function () {
    console.log("ERROR!!!")
    console.log(this);
}

req.open('GET', 'https://api.cryptonator.com/api/ticker/btc-us
req.send();
```

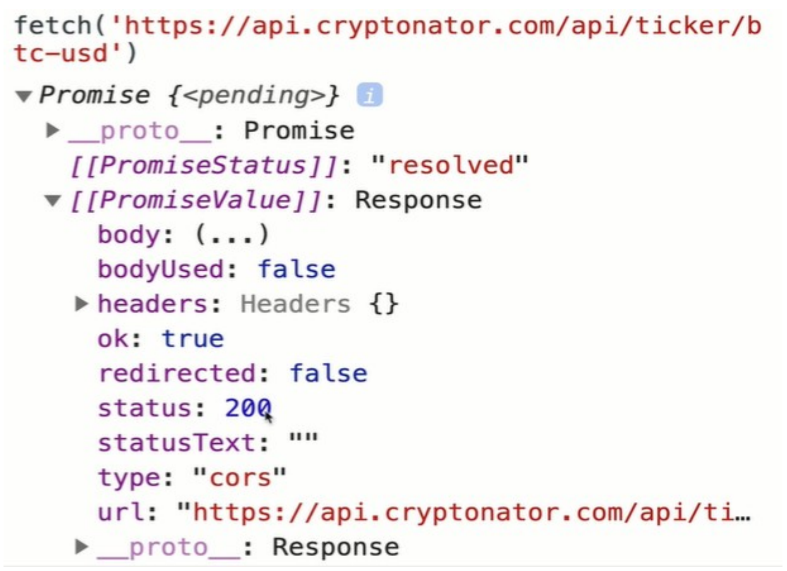


```
onprogress: null
onreadystatechange: null
ontimeout: null
readyState: 4
response: "{"ticker":{"base":"BTC","target…
responseText: "{"ticker":{"base":"BTC","ta…
responseType: ""
responseURL: "https://api.cryptonator.com/…
responseXML: null
status: 200
statusText: ""
timeout: 0
```

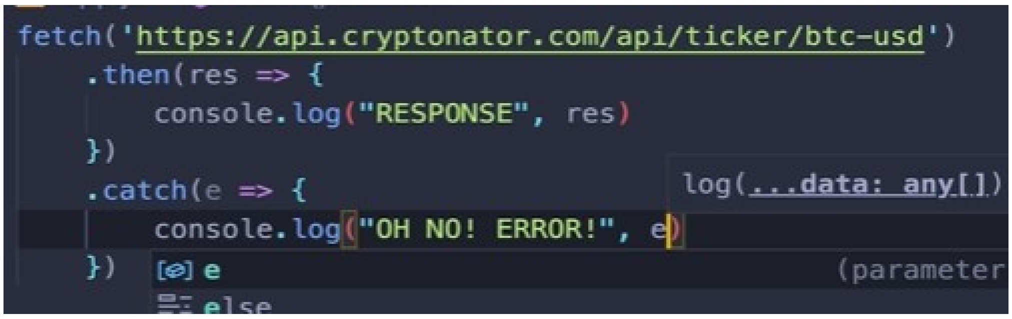


Fetch API

- The newer way of making requests via JS
- Supports promises!
- Not supported in Internet Explorer :(

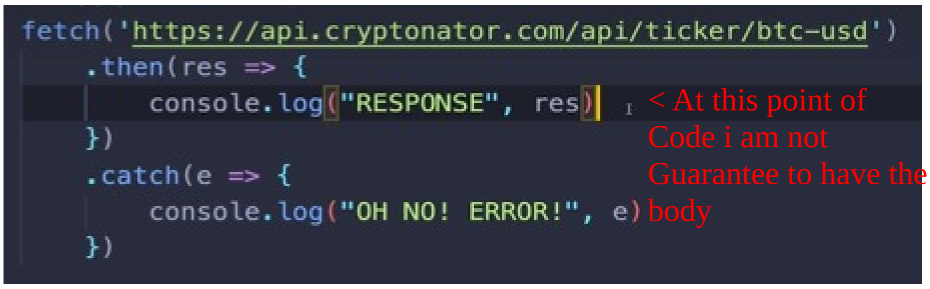


```
fetch('https://api.cryptonator.com/api/ticker/b
tc-usd')
▼Promise {<pending>}
  ▶__proto__: Promise
   [[PromiseStatus]]: "resolved"
  ▼[[PromiseValue]]: Response
     body: (...)
     bodyUsed: false
    ▶headers: Headers {}
     ok: true
     redirected: false
     status: 200
     statusText: ""
     type: "cors"
     url: "https://api.cryptonator.com/api/ti…
    ▶__proto__: Response
```



```
fetch('https://api.cryptonator.com/api/ticker/btc-usd')
    .then(res => {
        console.log("RESPONSE", res)
    })
    .catch(e => {                              log(...data: any[])
        console.log("OH NO! ERROR!", e)
    })  [⊘] e                                         (parameter
        else
```

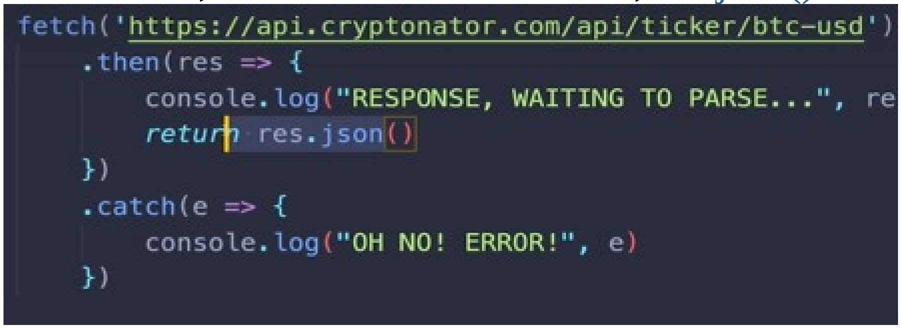
fetch returns a promise.
Fetch Api is promise based so we dont have to worry abt callback hell.
- fetch is gonna resolve the promise, triggering the .then as soon as it recieves the header coming from the Api. So it doesn't wait all of the data is back, the body is back.
-Information is send in streams, it takes time, all doesn't arrive at once.
-As soon as it get few bits of headers, fetch is gonna resolve the promise.



```
fetch('https://api.cryptonator.com/api/ticker/btc-usd')
    .then(res => {
        console.log("RESPONSE", res)
    })
    .catch(e => {
        console.log("OH NO! ERROR!", e)
    })
```

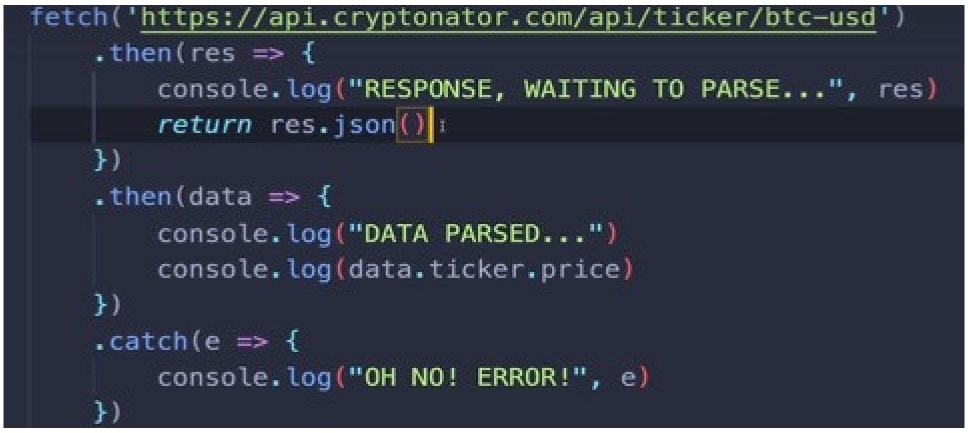
< At this point of Code i am not Guarantee to have the body

That is where, we use a second method, res.json()



```
fetch('https://api.cryptonator.com/api/ticker/btc-usd')
    .then(res => {
        console.log("RESPONSE, WAITING TO PARSE...", re
        return res.json()
    })
    .catch(e => {
        console.log("OH NO! ERROR!", e)
    })
```

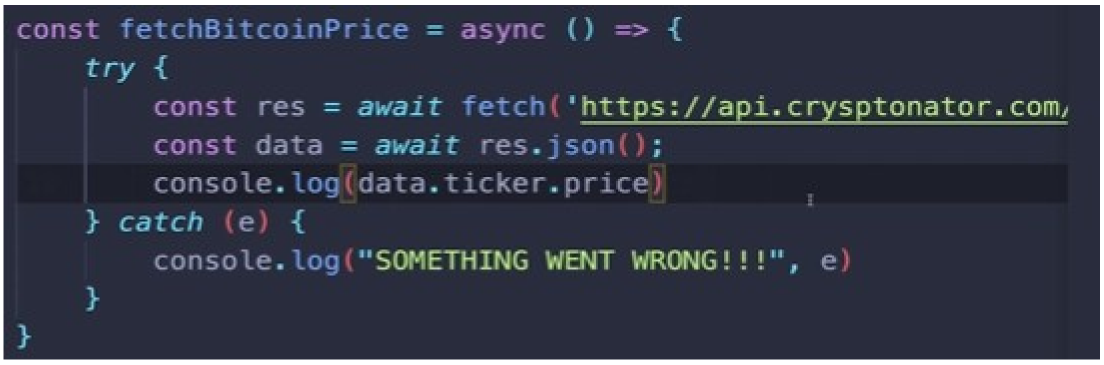
res.json() is another promise and we have to wait for it to finish.
fetch returns a promise cuz it is sychronous it's waiting for the body to arrive.



```
fetch('https://api.cryptonator.com/api/ticker/btc-usd')
    .then(res => {
        console.log("RESPONSE, WAITING TO PARSE...", res)
        return res.json()
    })
    .then(data => {
        console.log("DATA PARSED...")
        console.log(data.ticker.price)
    })
    .catch(e => {
        console.log("OH NO! ERROR!", e)
    })
```

res.json returns a promise.
If you return res.json, then you can chain on.

Above was using promises and below was using async-await



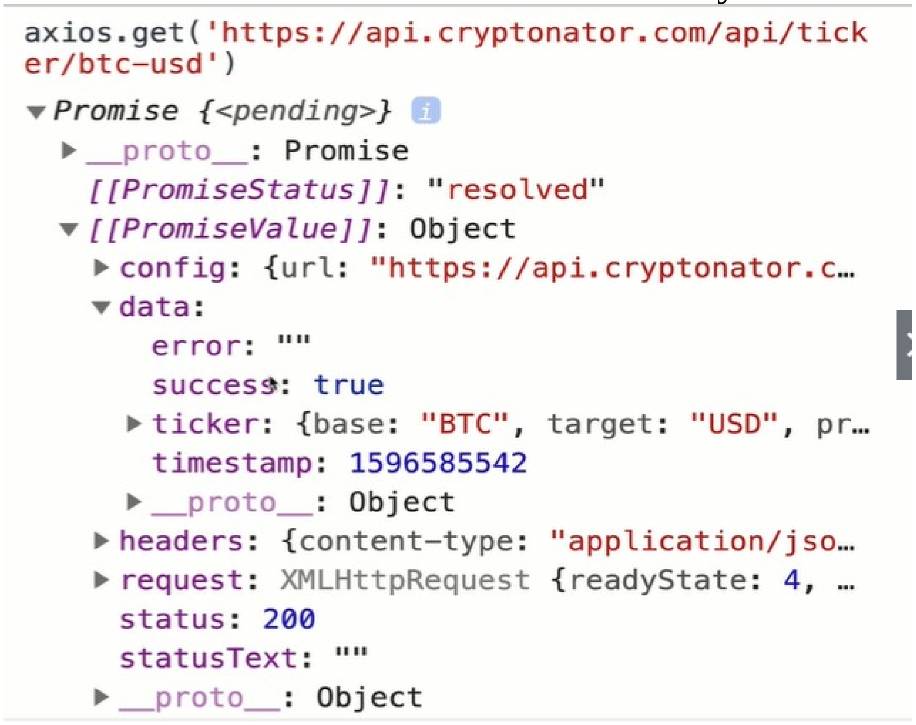
```
const fetchBitcoinPrice = async () => {
    try {
        const res = await fetch('https://api.crypstonator.com/
        const data = await res.json();
        console.log(data.ticker.price)
    } catch (e) {
        console.log("SOMETHING WENT WRONG!!!", e)
    }
}
```

[github/axios]()

Axios is a code that some-other people have written that is turn into a library that we can include, which gives us handful of useful methods to make request. It's built on top of fetch. Using Axios is easier.

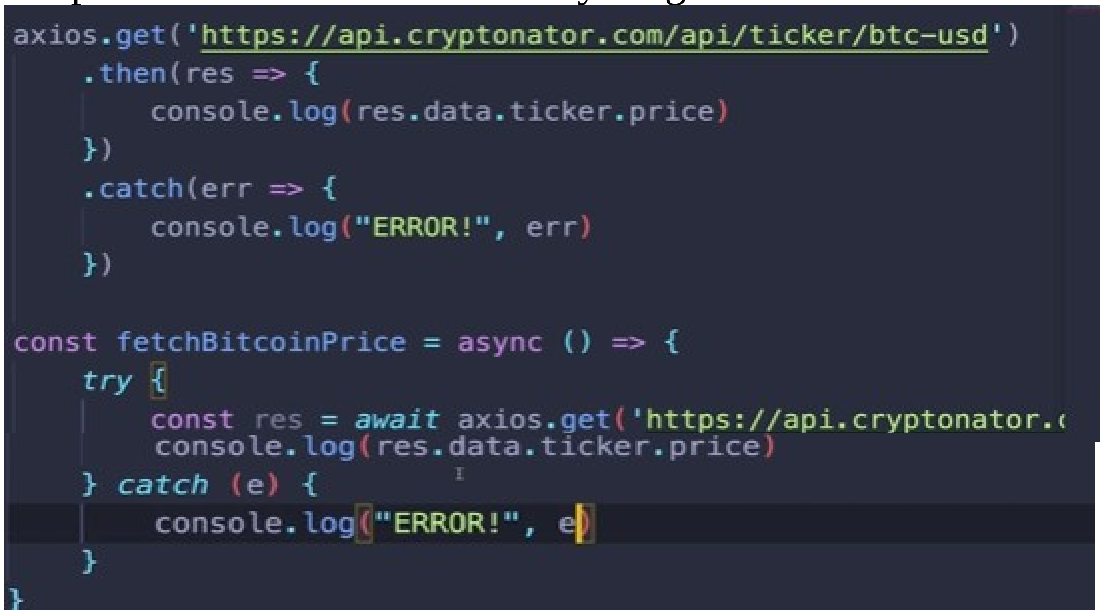
For post request, include query string, change headers
You can put the axios cdn in either script or the head. It doesn’t matter.Cuz these are external scripts they dont depend on the dom being loaded.

Axios works with both client side (js) and (node.js) server side where we install lib or import them manually.
Axios is used on both side with the same syntax.

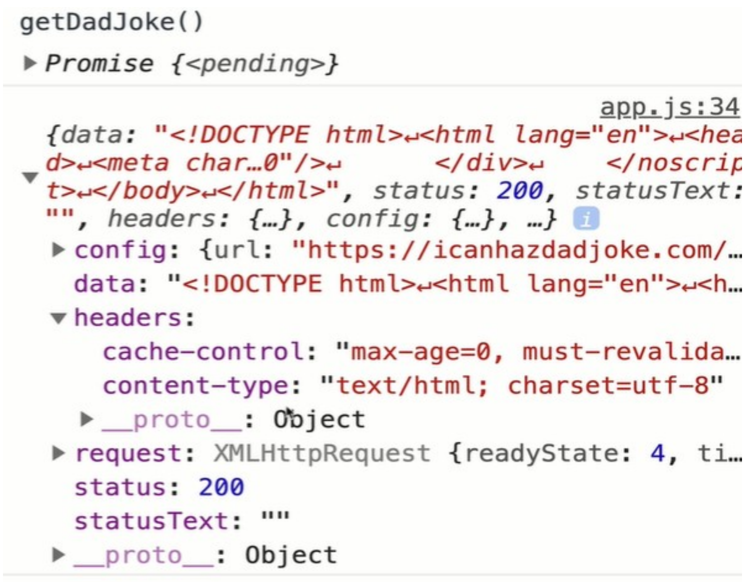


```
axios.get('https://api.cryptonator.com/api/tick
er/btc-usd')
▼Promise {<pending>}
  ▶__proto__: Promise
   [[PromiseStatus]]: "resolved"
  ▼[[PromiseValue]]: Object
    ▶config: {url: "https://api.cryptonator.c…
    ▼data:
        error: ""
        success: true
      ▶ticker: {base: "BTC", target: "USD", pr…
        timestamp: 1596585542
      ▶__proto__: Object
    ▶headers: {content-type: "application/jso…
    ▶request: XMLHttpRequest {readyState: 4, …
     status: 200
     statusText: ""
    ▶__proto__: Object
```

-No res.join() like in fetch
-The promise is resolved once everything is finished.

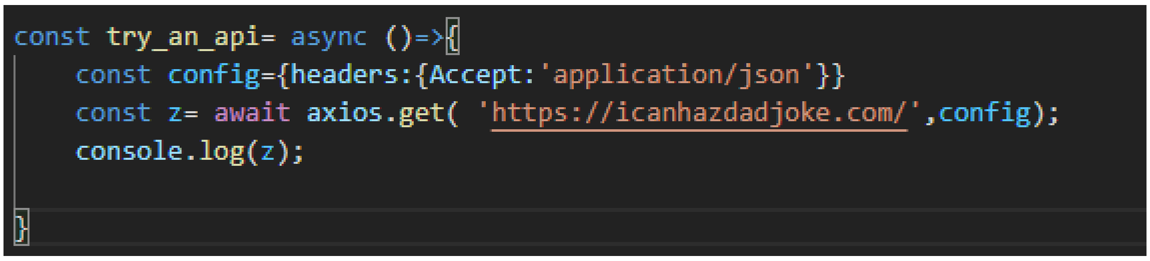


```
axios.get('https://api.cryptonator.com/api/ticker/btc-usd')
    .then(res => {
        console.log(res.data.ticker.price)
    })
    .catch(err => {
        console.log("ERROR!", err)
    })

const fetchBitcoinPrice = async () => {
    try {
        const res = await axios.get('https://api.cryptonator.
        console.log(res.data.ticker.price)
    } catch (e) {
        console.log("ERROR!", e)
    }
}
```

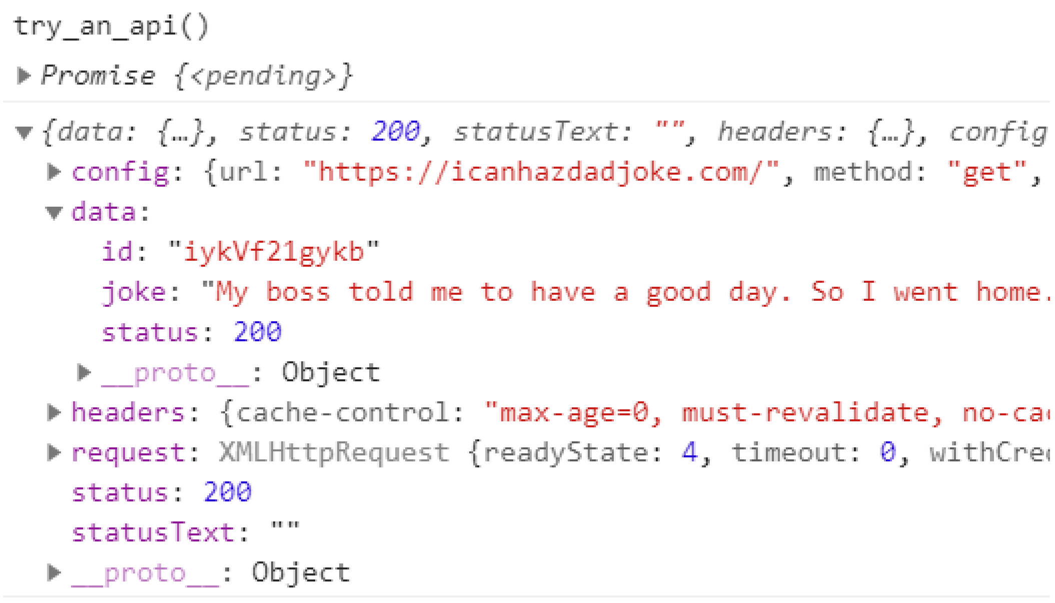


```
getDadJoke()
▶ Promise {<pending>}
                                                    app.js:34
▼ {data: "<!DOCTYPE html>↵<html lang="en">↵<hea
  d>↵<meta char…0"/>↵      </div>↵     </noscrip
  t>↵</body>↵</html>", status: 200, statusText:
  "", headers: {…}, config: {…}, …}
  ▶ config: {url: "https://icanhazdadjoke.com/…
    data: "<!DOCTYPE html>↵<html lang="en">↵<h…
  ▼ headers:
      cache-control: "max-age=0, must-revalida…
      content-type: "text/html; charset=utf-8"
    ▶ __proto__: Object
  ▶ request: XMLHttpRequest {readyState: 4, ti…
    status: 200
    statusText: ""
  ▶ __proto__: Object
```

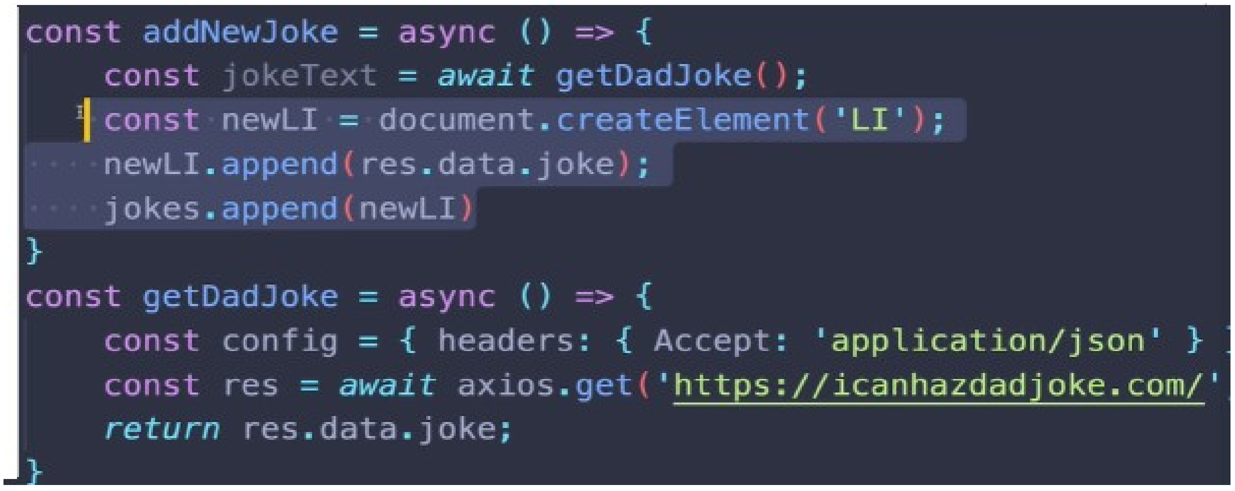
Here, you made a get request on the endpoint, the api is configured so as to use Headers where Accept:application/json to get json . You gotta read the documentation to know this.
To get json, it's different for Api to Api.
Some may have an endpoint :
 '..../json'
 '.../?type=json'



```
const try_an_api= async ()=>{
    const config={headers:{Accept:'application/json'}}
    const z= await axios.get( 'https://icanhazdadjoke.com/',config);
    console.log(z);

}
```



```
try_an_api()
▶ Promise {<pending>}
▼ {data: {…}, status: 200, statusText: "", headers: {…}, config
  ▶ config: {url: "https://icanhazdadjoke.com/", method: "get",
  ▼ data:
      id: "iykVf21gykb"
      joke: "My boss told me to have a good day. So I went home.
      status: 200
    ▶ __proto__: Object
  ▶ headers: {cache-control: "max-age=0, must-revalidate, no-ca
  ▶ request: XMLHttpRequest {readyState: 4, timeout: 0, withCre
    status: 200
    statusText: ""
  ▶ __proto__: Object
```



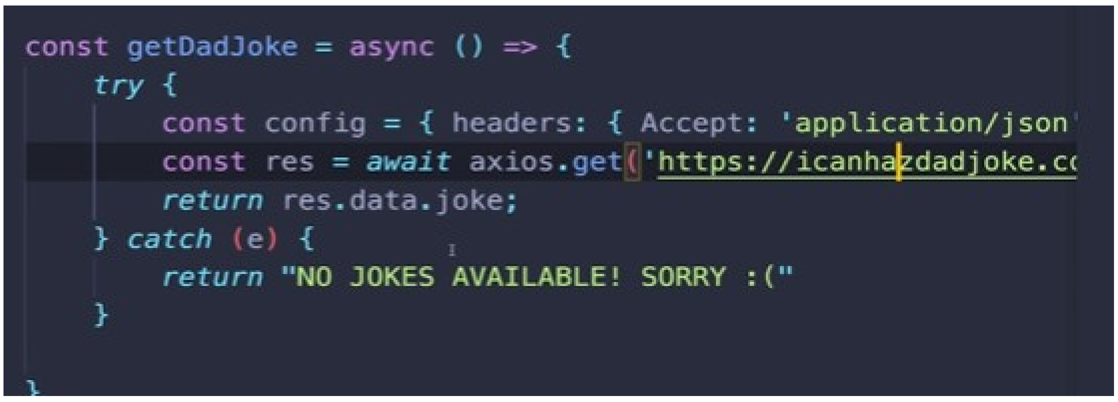
```
1  const addNewJoke = async () => {
2      const jokeText = await getDadJoke();
3      const newLI = document.createElement('LI');
4      newLI.append(res.data.joke);
5      jokes.append(newLI)
6  }
7  const getDadJoke = async () => {
8      const config = { headers: { Accept: 'application/json' }
9      const res = await axios.get('https://icanhazdadjoke.com/'
10     return res.data.joke;
   }
```

At line 1, There is async cuz we gonna return a promise with some data.
At line 2,   Await is used cuz

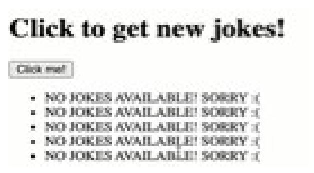
1.is used in func getDadJoke

At line 9: telling to wait till the json or text/html is resolved and unlike fetch() , Axios only returns promise with data only when it's resolved.

2. At line 2, to wait till a promise is settled.



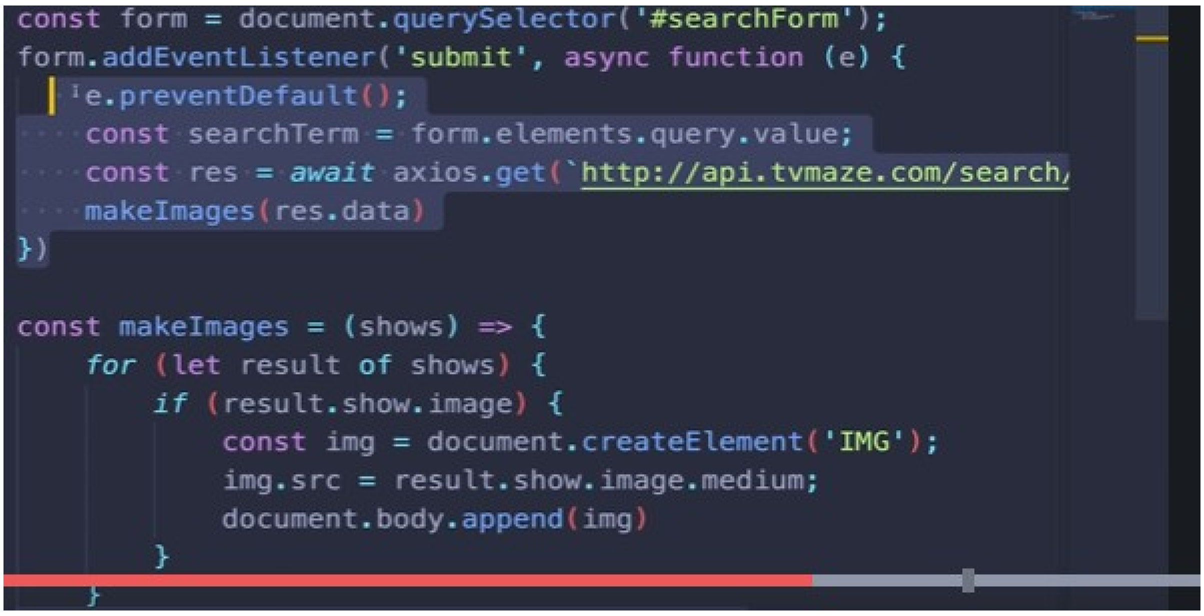
```
const getDadJoke = async () => {
    try {
        const config = { headers: { Accept: 'application/json'
        const res = await axios.get('https://icanhazdadjoke.co
        return res.data.joke;
    } catch (e) {
        return "NO JOKES AVAILABLE! SORRY :("
    }

}
```



Click to get new jokes!

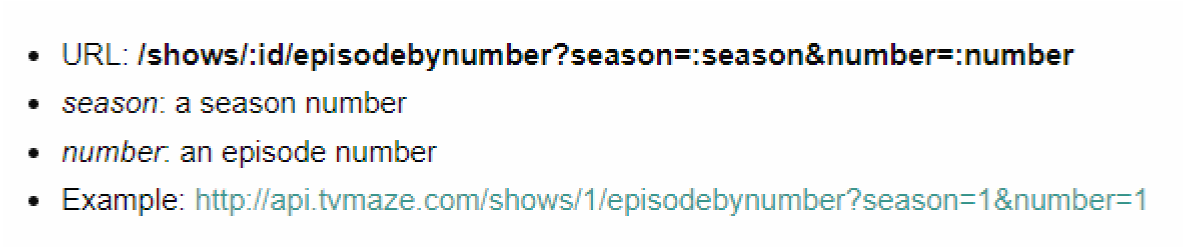
Click me!

- NO JOKES AVAILABLE! SORRY :(
- NO JOKES AVAILABLE! SORRY :(
- NO JOKES AVAILABLE! SORRY :(
- NO JOKES AVAILABLE! SORRY :(
- NO JOKES AVAILABLE! SORRY :(



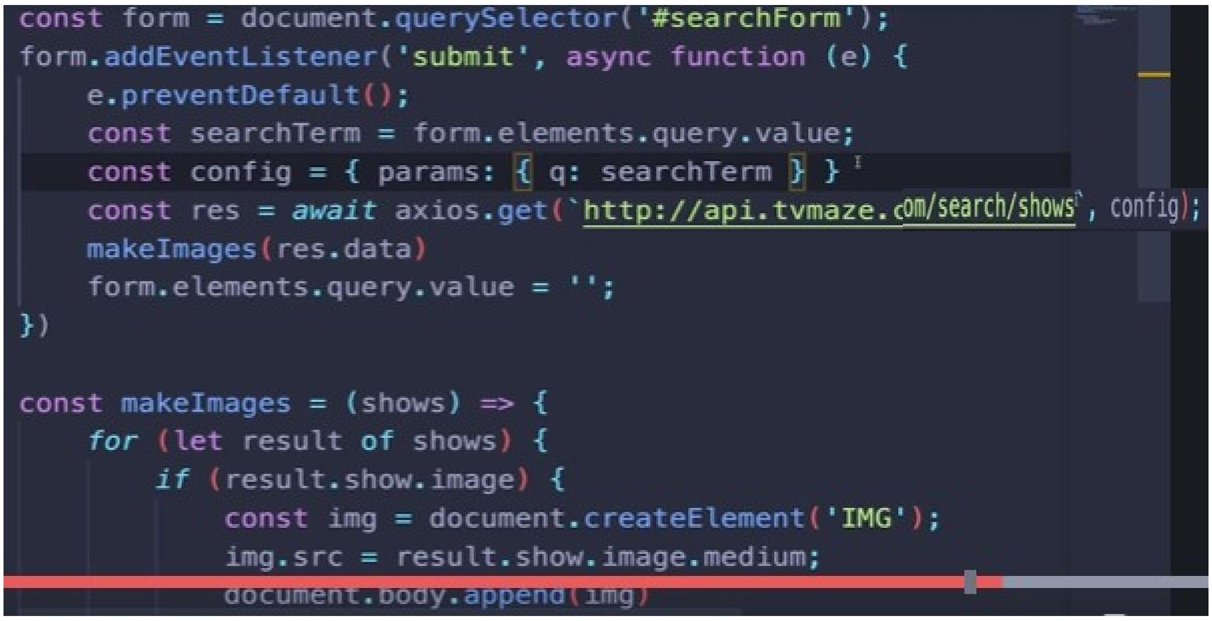
```
const form = document.querySelector('#searchForm');
form.addEventListener('submit', async function (e) {
    e.preventDefault();
    const searchTerm = form.elements.query.value;
    const res = await axios.get(`http://api.tvmaze.com/search/
    makeImages(res.data)
})

const makeImages = (shows) => {
    for (let result of shows) {
        if (result.show.image) {
            const img = document.createElement('IMG');
            img.src = result.show.image.medium;
            document.body.append(img)
        }
    }
```

tvmaze.com/api



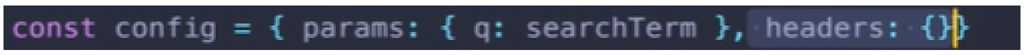
- URL: **/shows/:id/episodebynumber?season=:season&number=:number**
- *season*: a season number
- *number*: an episode number
- Example: http://api.tvmaze.com/shows/1/episodebynumber?season=1&number=1

When you make request to an api, you often have multiple params with data in the query-string Like Above so u does that like below.



```
const form = document.querySelector('#searchForm');
form.addEventListener('submit', async function (e) {
    e.preventDefault();
    const searchTerm = form.elements.query.value;
    const config = { params: { q: searchTerm } }
    const res = await axios.get(`http://api.tvmaze.com/search/shows`, config);
    makeImages(res.data)
    form.elements.query.value = '';
})

const makeImages = (shows) => {
    for (let result of shows) {
        if (result.show.image) {
            const img = document.createElement('IMG');
            img.src = result.show.image.medium;
            document.body.append(img)
```

We can also add headers in



```
const config = { params: { q: searchTerm }, headers: {}}
```

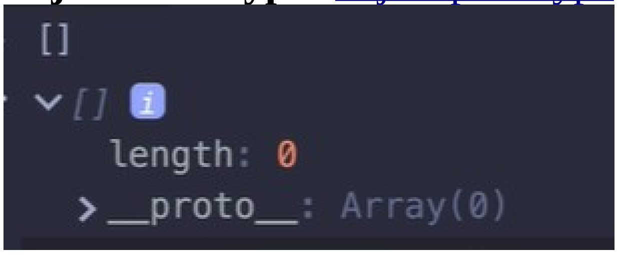
for a live search, ppl usually request their own api and or have some functionaliy so they don’t overload an api.

Prototype, Classes & OOP

How does an array and objects have dozen of methods every single time we make.

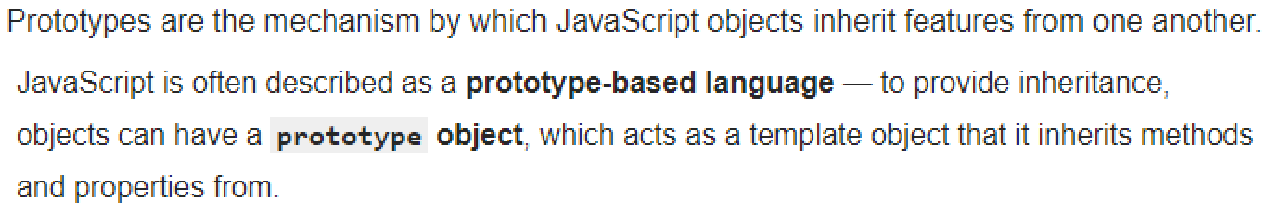
How do we replicate that functionality ourselves or a method or property in our repeatable and reusable recipe.

Object Prototypes [object prototype]()



```
[]
▾ [] ⓘ
    length: 0
  ▸ __proto__: Array(0)
```

What is this __proto__ ?

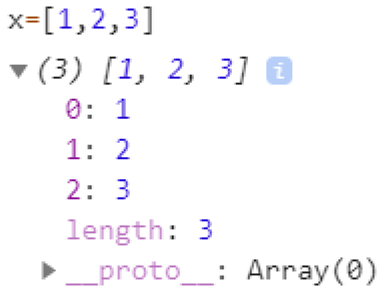


Prototypes are the mechanism by which JavaScript objects inherit features from one another.

JavaScript is often described as a **prototype-based language** — to provide inheritance, objects can have a `prototype` **object**, which acts as a template object that it inherits methods and properties from.

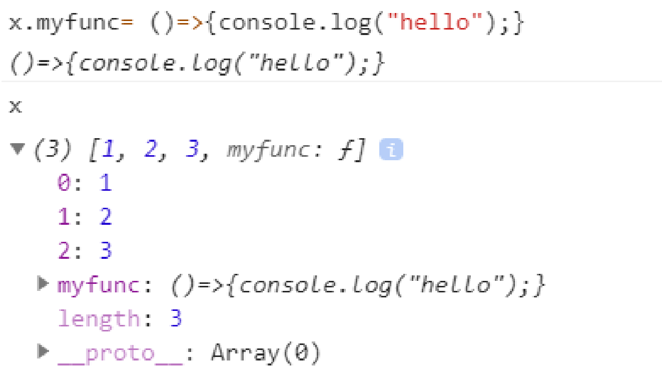
If we look about Array in MDN, we see there is so many methods like Array.prototype.push()

The push is method isn’t defined as a method on this array.



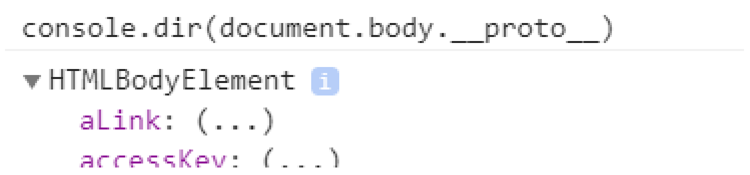
```
x=[1,2,3]
▾ (3) [1, 2, 3] ⓘ
    0: 1
    1: 2
    2: 3
    length: 3
  ▸ __proto__: Array(0)
```

I don’t see any methods right here.



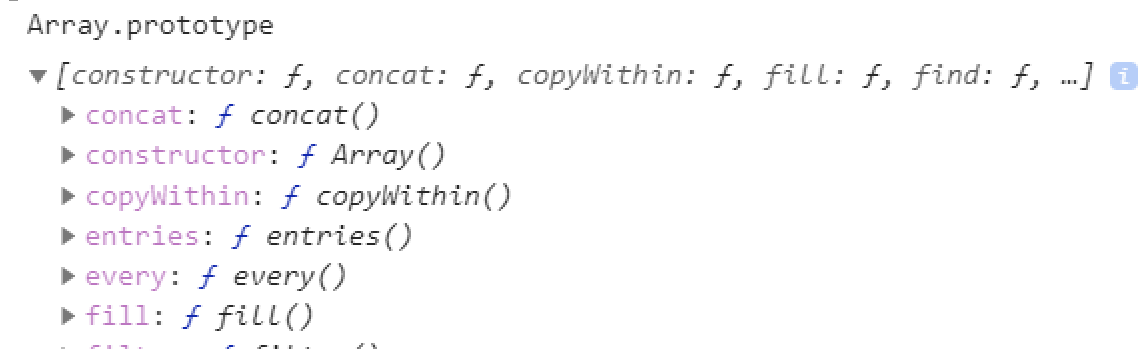
```
x.myfunc= ()=>{console.log("hello");}
()=>{console.log("hello");}
x
▾ (3) [1, 2, 3, myfunc: f] ⓘ
    0: 1
    1: 2
    2: 3
  ▸ myfunc: ()=>{console.log("hello");}
    length: 3
  ▸ __proto__: Array(0)
```

Here, if i add my own func, but i dont see myarray.push() func listed here. So a prototype is a template obj, it contains bunch of methods that every array have access to.



```
console.dir(document.body.__proto__)
▼HTMLBodyElement ℹ
    aLink: (...)
    accessKey: (...)
```

document.body prototype is HTMLBodyElement. There are bunch of properties and methods that refers to this template.

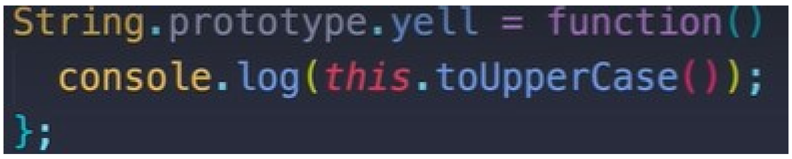


```
Array.prototype
▼[constructor: ƒ, concat: ƒ, copyWithin: ƒ, fill: ƒ, find: ƒ, …] ℹ
  ▶concat: ƒ concat()
  ▶constructor: ƒ Array()
  ▶copyWithin: ƒ copyWithin()
  ▶entries: ƒ entries()
  ▶every: ƒ every()
  ▶fill: ƒ fill()
```

Here is everything on the array prototype, i can define my own brand new object and set its prototype to Array.prototype and i would have access to those array methods.

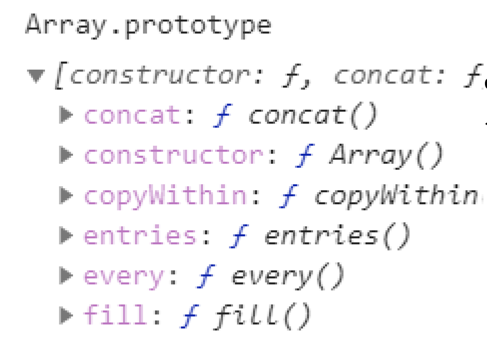
Prototype are like template objects, they contain typically a bunch of methods, we can create bunch of objects that share the same prototype, so that all have access to the same method.

We can add our own methods and properties to the array prototype
-We also have String.prototype, as we talk abt String are primitive types, in js they all get an intermediate object wrappers.



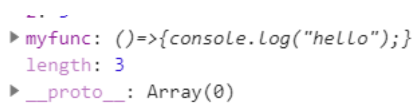
```
String.prototype.yell = function()
  console.log(this.toUpperCase());
};
```

Here, “this” refers to whatever is on left side.
You can also overide template func, pls dont but.



```
Array.prototype
▼[constructor: ƒ, concat: ƒ
  ▶concat: ƒ concat()
  ▶constructor: ƒ Array()
  ▶copyWithin: ƒ copyWithin
  ▶entries: ƒ entries()
  ▶every: ƒ every()
  ▶fill: ƒ fill()
```

Here this is actual template object where we add methods to all objects of that type.
<<<

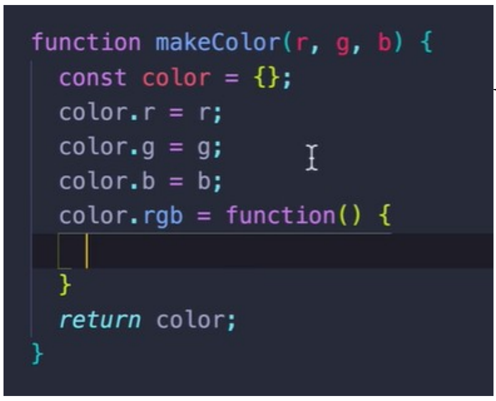


```
▶myfunc: ()=>{console.log("hello");}
  length: 3
▶__proto__: Array(0)
```

>>> this a reference to the template obj

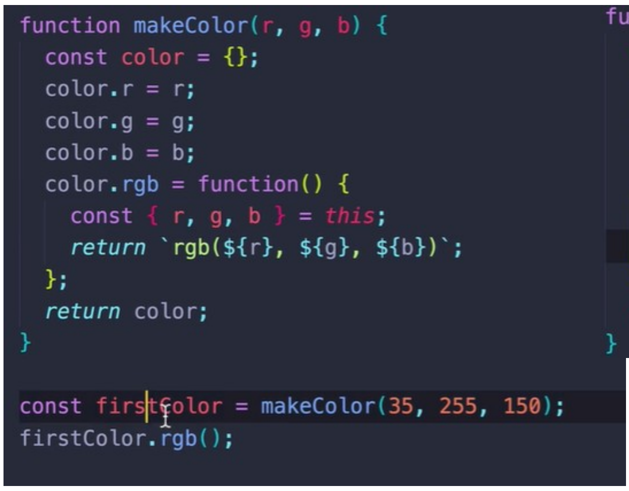
All has to do with one central idea, which is organizing our code, designing and structuring our application, by breaking things up into distinct pattern of objects. When i say pattern of objects you can also think of recipes of objects.
See this page for class and objects:
Even a dom element like h1 is some object created with same cookie-cutter i.e for h1 is called HtmlHeadingElement.

Factory functions

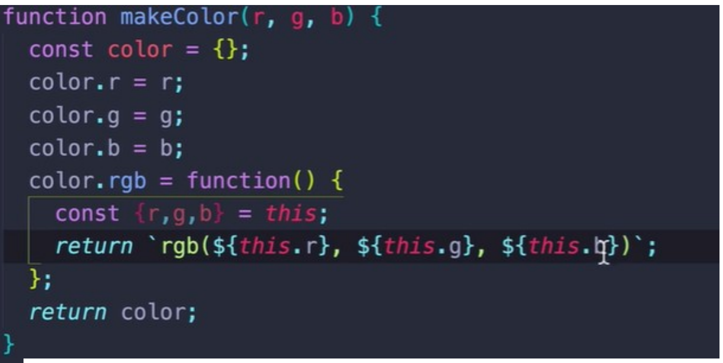


```
function makeColor(r, g, b) {
  const color = {};
  color.r = r;
  color.g = g;
  color.b = b;
  color.rgb = function() {

  }
  return color;
}
```

Here, you building up a object hence the word Factory.



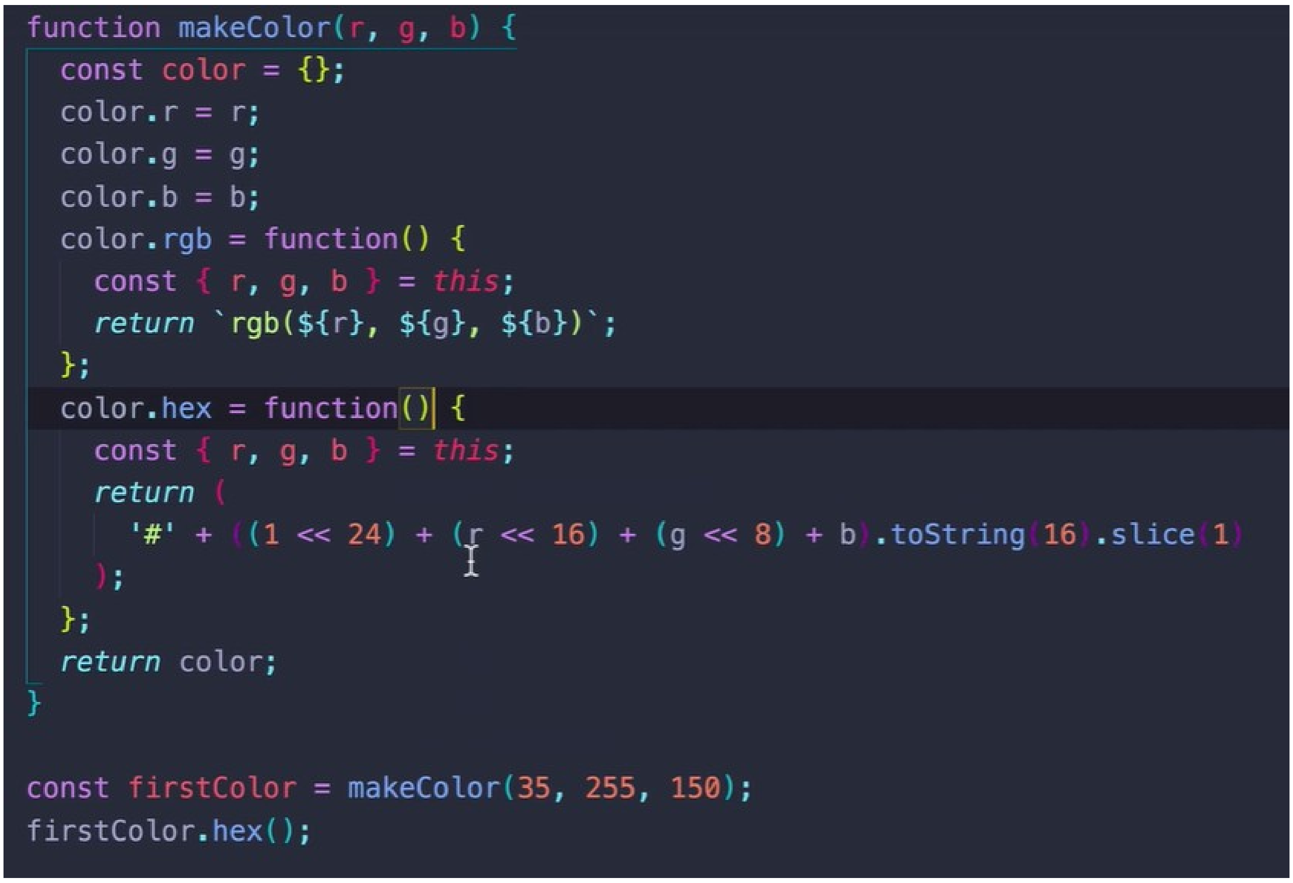
```
function makeColor(r, g, b) {
  const color = {};
  color.r = r;
  color.g = g;
  color.b = b;
  color.rgb = function() {
    const { r, g, b } = this;
    return `rgb(${r}, ${g}, ${b})`;
  };
  return color;
}

const firstColor = makeColor(35, 255, 150);
firstColor.rgb();
```



```
function makeColor(r, g, b) {
  const color = {};
  color.r = r;
  color.g = g;
  color.b = b;
  color.rgb = function() {
    const {r,g,b} = this;
    return `rgb(${this.r}, ${this.g}, ${this.b})`;
  };
  return color;
}
```

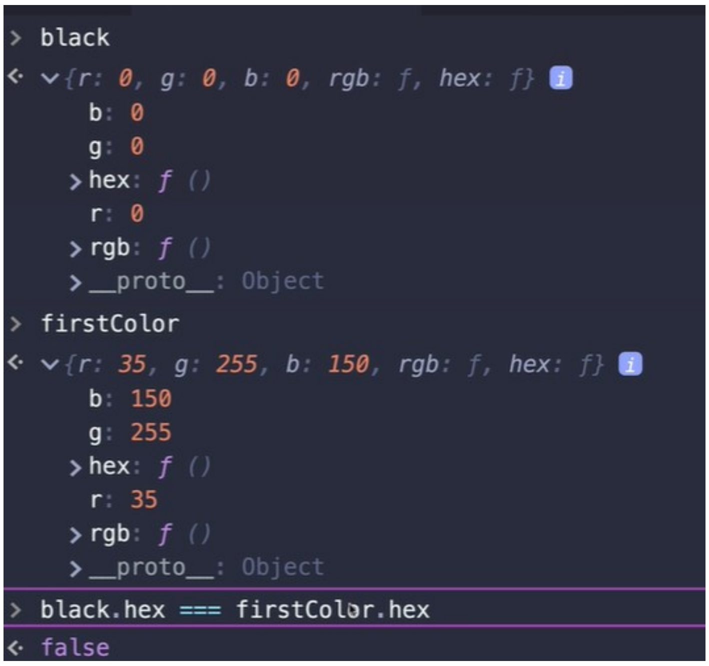
We use Object destructor and this keyword <!-- Very Smart

the ‘this’ keyword inside a func, refers to object calling that func, in above e.g it refers to firstColor.
Here,
const {r,g,b}=this
is Object Destructor where we retrive our wanted properties.

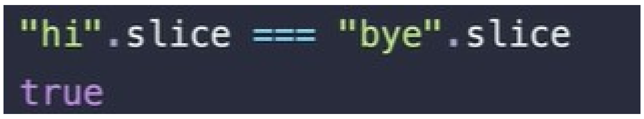


```
function makeColor(r, g, b) {
  const color = {};
  color.r = r;
  color.g = g;
  color.b = b;
  color.rgb = function() {
    const { r, g, b } = this;
    return `rgb(${r}, ${g}, ${b})`;
  };
  color.hex = function() {
    const { r, g, b } = this;
    return (
      '#' + ((1 << 24) + (r << 16) + (g << 8) + b).toString(16).slice(1)
    );
  };
  return color;
}

const firstColor = makeColor(35, 255, 150);
firstColor.hex();
```

The problem is, you start with an empty object, 3 unique property is added to that object, and these functions are re-created and a unique copy is added to each color object. So each color object has his own rgb func.
The properties is gonna be different for all colors. But there is no reason to have a unique copy of the func itself.
So for everyColor we dont want a own seperate func, which is by work similar to all Color objects.

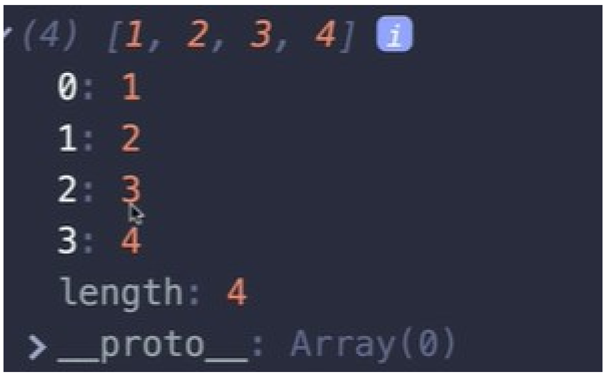


```
> black
<· ∨{r: 0, g: 0, b: 0, rgb: f, hex: f}
      b: 0
      g: 0
    > hex: f ()
      r: 0
    > rgb: f ()
    > __proto__: Object
> firstColor
<· ∨{r: 35, g: 255, b: 150, rgb: f, hex: f}
      b: 150
      g: 255
    > hex: f ()
      r: 35
    > rgb: f ()
    > __proto__: Object
> black.hex === firstColor.hex
<· false
```



```
"hi".slice === "bye".slice
true
```

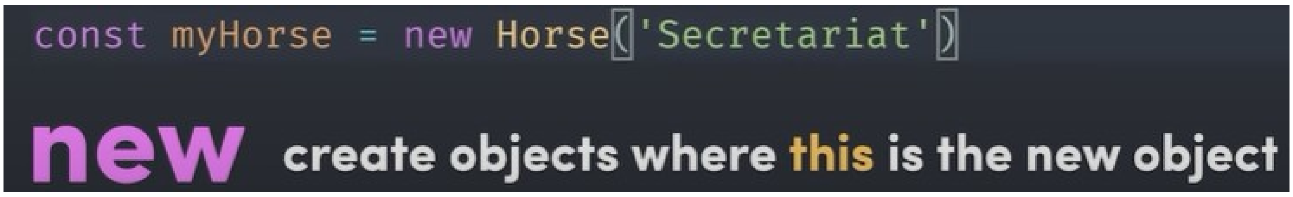
This string "hi" doesn't have his own copy of slice. Neither string "bye" too.



```
(4) [1, 2, 3, 4]
  0: 1
  1: 2
  2: 3
  3: 4
  length: 4
 > __proto__: Array(0)
```

Array built-in methods aren't defined for individual objects but __proto__ is used as a reference to access methods from Array.prototype Object.

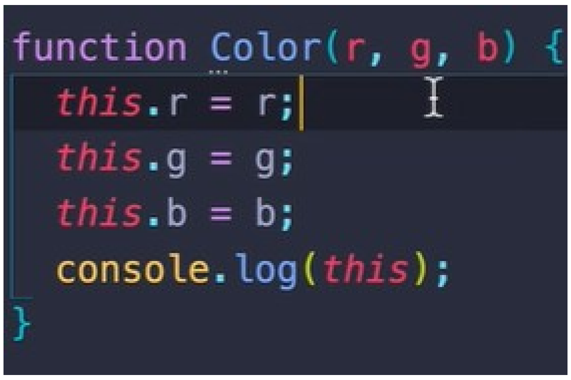
Constructor functions



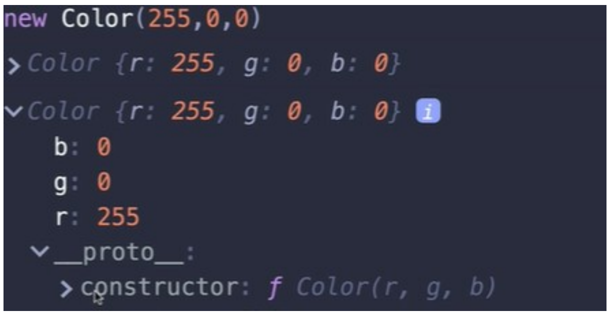
```
const myHorse = new Horse('Secretariat')
```

new create objects where this is the new object

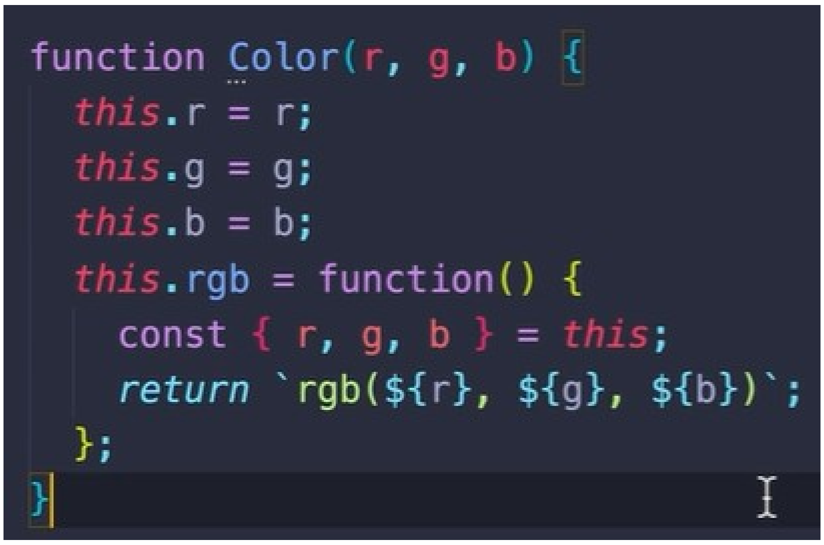
The new keyword allows you to call func that creates an object where this is automatically bound to the newly created object. That means you can create property and methods like many other OOP languages.
So we can define a property name and a func sayHello and use "this.name" to reference the name property in this object.
This is a nice way to keep the data and functionality of the object tightly coupled to each other.



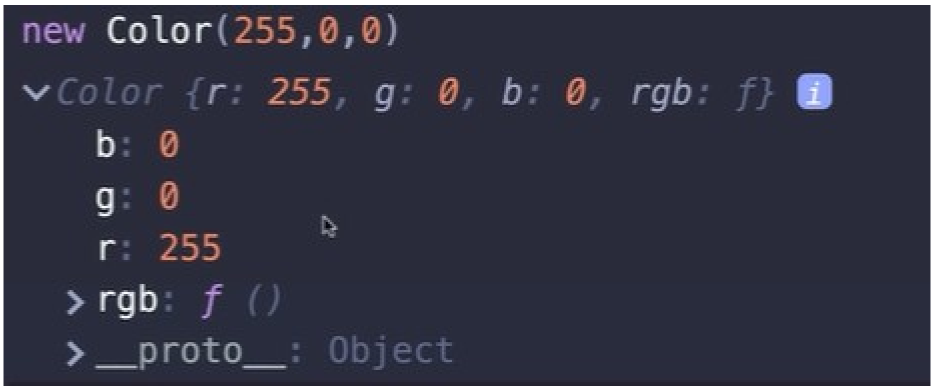
```
function Color(r, g, b) {
  this.r = r;
  this.g = g;
  this.b = b;
  console.log(this);
}
```



```
new Color(255,0,0)
>Color {r: 255, g: 0, b: 0}
vColor {r: 255, g: 0, b: 0}
   b: 0
   g: 0
   r: 255
  v__proto__:
   >constructor: f Color(r, g, b)
```

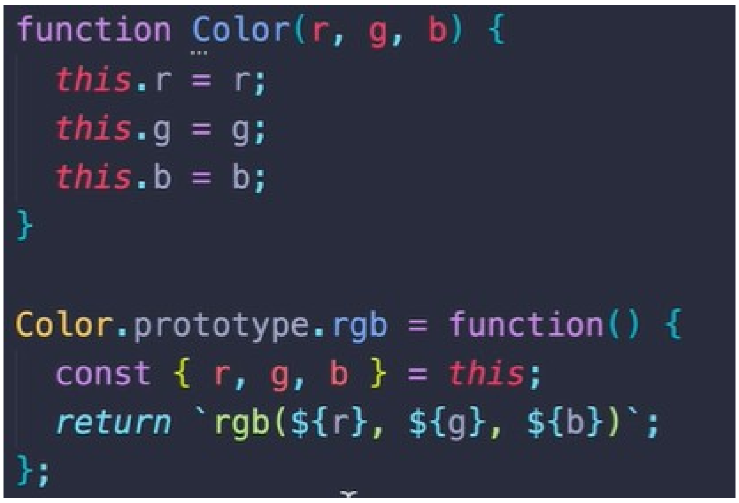


```
function Color(r, g, b) {
  this.r = r;
  this.g = g;
  this.b = b;
  this.rgb = function() {
    const { r, g, b } = this;
    return `rgb(${r}, ${g}, ${b})`;
  };
}
```



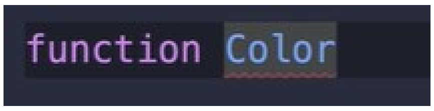
```
new Color(255,0,0)
⌄Color {r: 255, g: 0, b: 0, rgb: f} i
    b: 0
    g: 0
    r: 255
  > rgb: f ()
  > __proto__: Object
```

The rgb isn’t defined on the prototype
So,



```
function Color(r, g, b) {
  this.r = r;
  this.g = g;
  this.b = b;
}

Color.prototype.rgb = function() {
  const { r, g, b } = this;
  return `rgb(${r}, ${g}, ${b})`;
};
```

<<< dont use arrow function here.



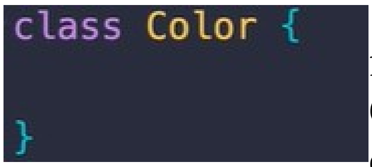
```
function Color
```

here the function name with capital letter that's just a way of indicating a regular function. It is a function that helps you create an Object. Its's a constructor function.

JavaScript Classes

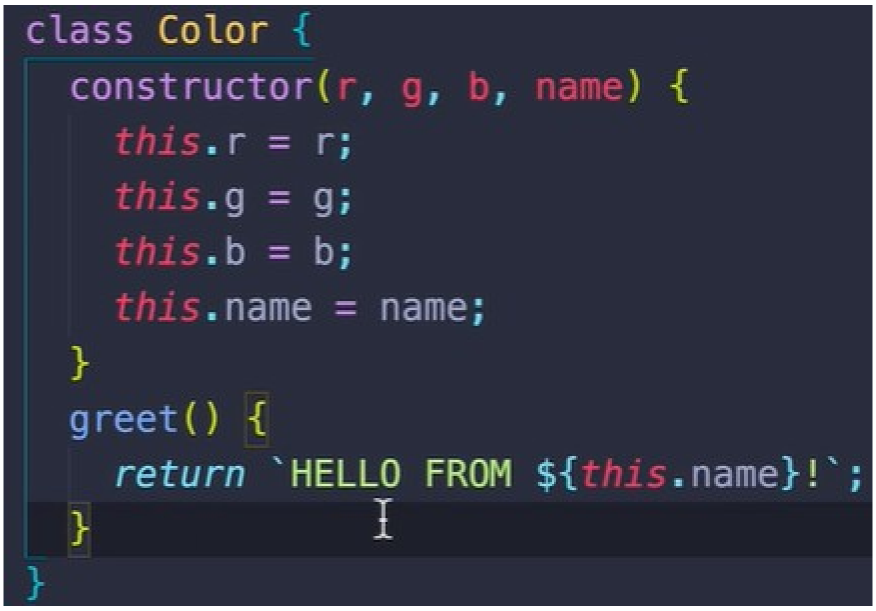
Syntactic sugar is a cleaner, cuter way of writing something that behind the scene is gonna turn into what we have here(smthg not nice)

Using Class benefits are, u dont have to add methods to prototype manually. You dont have to breakup constructor functions and seperately add functions. Cuz you end up with a nice little compact function that explain the property of that object. But then you are OoOo There is also methods, they are on the prototype. They are defined seperately and individually.



```
class Color {

}
```

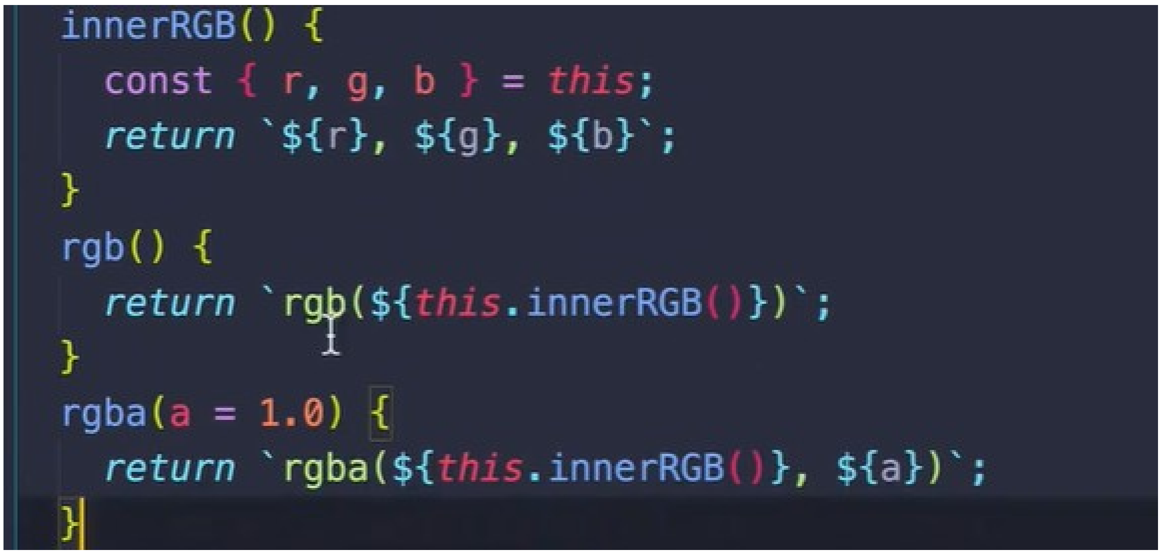
We CamelCase when there is a class or constructor function.
Constructor is executed whenever a instance is created.



```
class Color {
  constructor(r, g, b, name) {
    this.r = r;
    this.g = g;
    this.b = b;
    this.name = name;
  }
  greet() {
    return `HELLO FROM ${this.name}!`;
  }
}
```

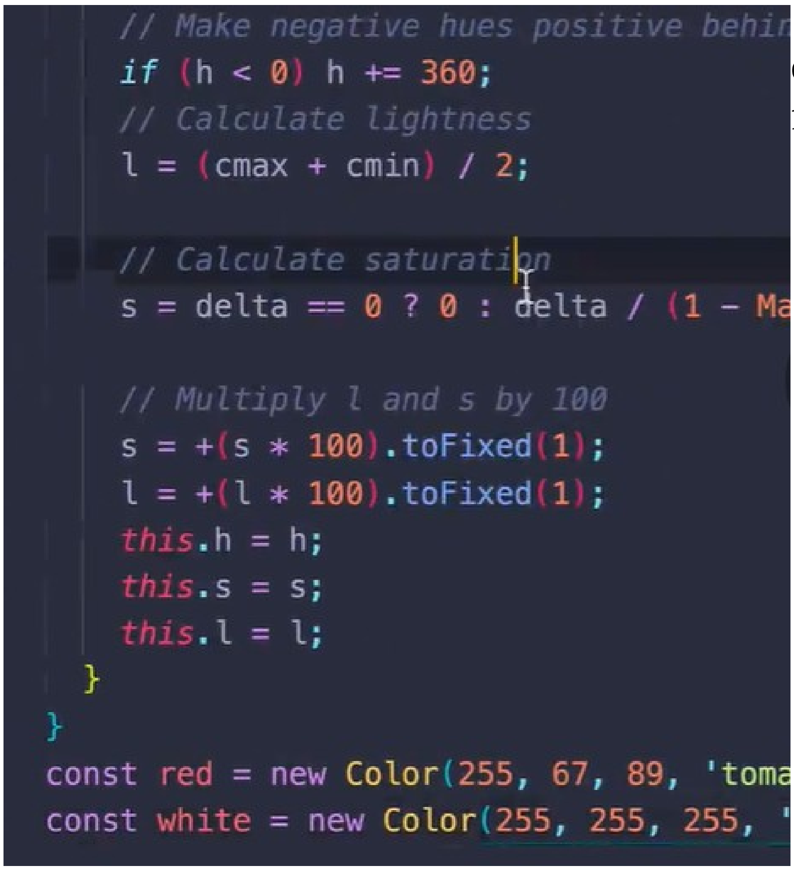
The new keyword allows you to call func that creates an object where this is automatically bound to the newly created object.
Mybook1.getName()   <!-- here the "this" keyword insided the func getName refers to the object calling that function. Anyway "this" was already bing

the object btw.



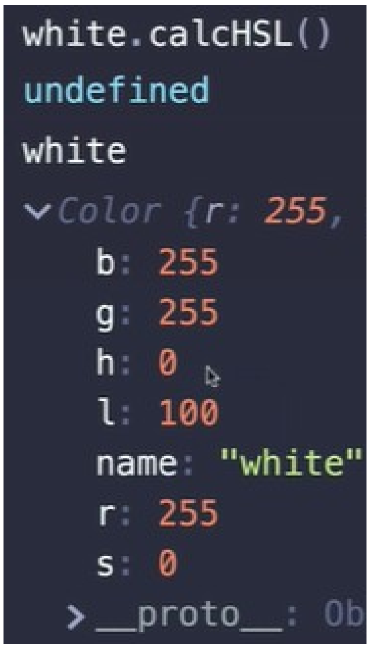
```
innerRGB() {
  const { r, g, b } = this;
  return `${r}, ${g}, ${b}`;
}
rgb() {
  return `rgb(${this.innerRGB()})`;
}
rgba(a = 1.0) {
  return `rgba(${this.innerRGB()}, ${a})`;
}
```

Above is, how to call a method is inside a method during Class creation.

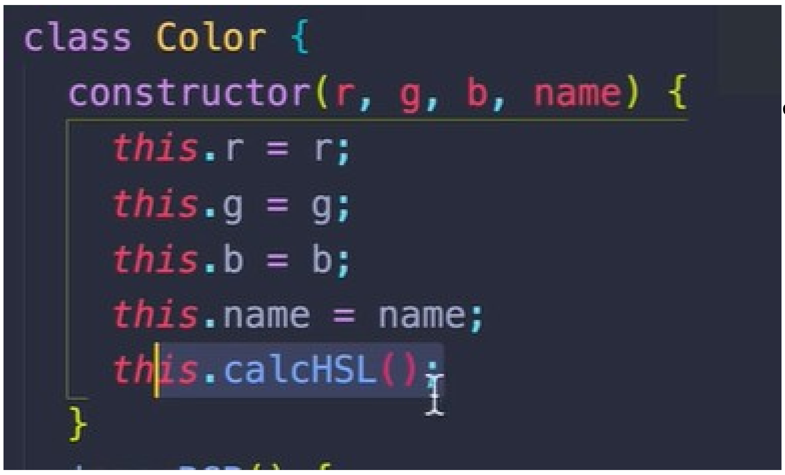


```
    // Make negative hues positive behin
    if (h < 0) h += 360;
    // Calculate lightness
    l = (cmax + cmin) / 2;

    // Calculate saturation
    s = delta == 0 ? 0 : delta / (1 - Ma

    // Multiply l and s by 100
    s = +(s * 100).toFixed(1);
    l = +(l * 100).toFixed(1);
    this.h = h;
    this.s = s;
    this.l = l;
  }
}
const red = new Color(255, 67, 89, 'toma
const white = new Color(255, 255, 255, '
```

This is a nice way of creating properties when a func is called.



```
white.calcHSL()
undefined
white
⌄Color {r: 255,
    b: 255
    g: 255
    h: 0
    l: 100
    name: "white"
    r: 255
    s: 0
  >__proto__: Ob
```

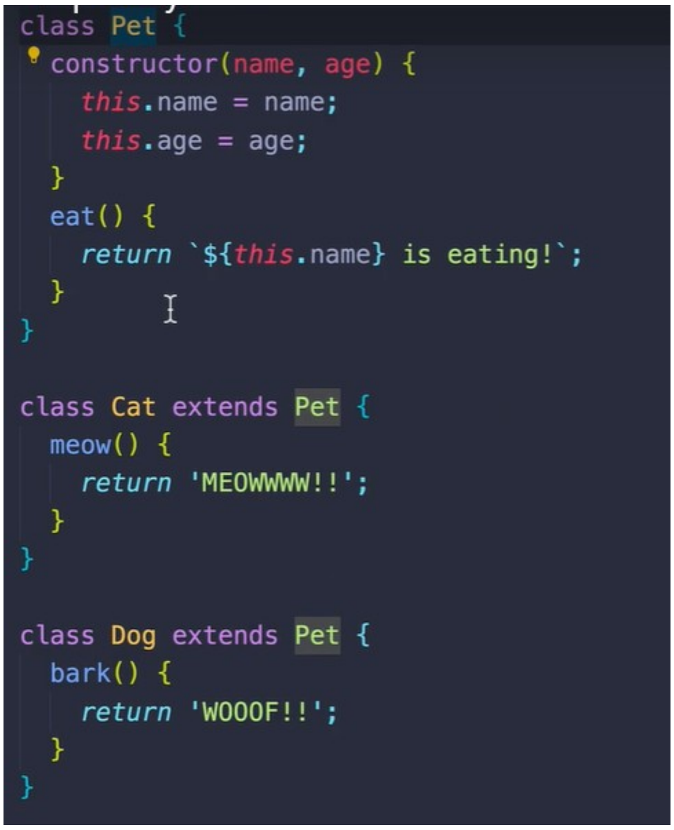


```
class Color {
  constructor(r, g, b, name) {
    this.r = r;
    this.g = g;
    this.b = b;
    this.name = name;
    this.calcHSL();
  }
```

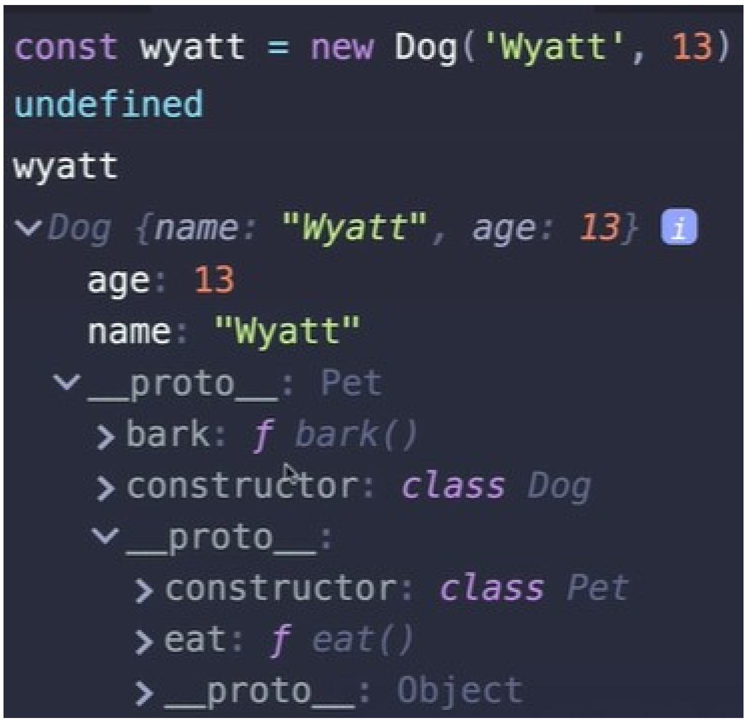
You can call a method inside a constructor.

Extends and Super keywords.

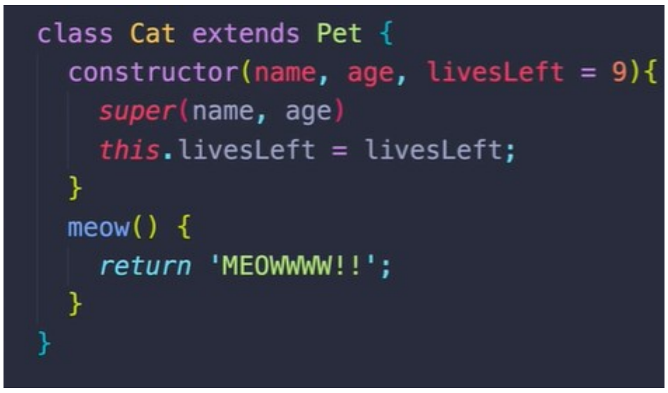
They both have to do with subclasses, especially inheritance.
This is the way of sharing functionalities between classes.



```
class Pet {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  eat() {
    return `${this.name} is eating!`;
  }
}

class Cat extends Pet {
  meow() {
    return 'MEOWWWW!!';
  }
}

class Dog extends Pet {
  bark() {
    return 'WOOOF!!';
  }
}
```



```
const wyatt = new Dog('Wyatt', 13)
undefined
wyatt
▾Dog {name: "Wyatt", age: 13} i
    age: 13
    name: "Wyatt"
  ▾__proto__: Pet
    ›bark: ƒ bark()
    ›constructor: class Dog
    ▾__proto__:
      ›constructor: class Pet
      ›eat: ƒ eat()
      ›__proto__: Object
```



```
class Cat extends Pet {
  constructor(name, age, livesLeft = 9){
    super(name, age)
    this.livesLeft = livesLeft;
  }
  meow() {
    return 'MEOWWWW!!';
  }
}
```