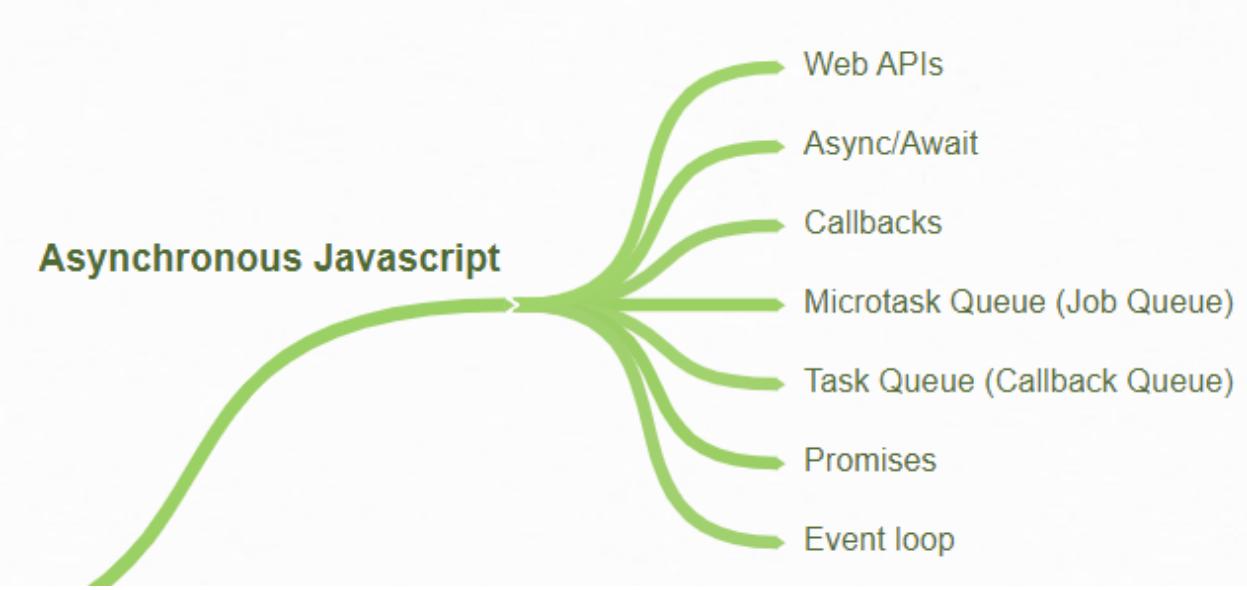
Section 9: Asynchronous JavaScript

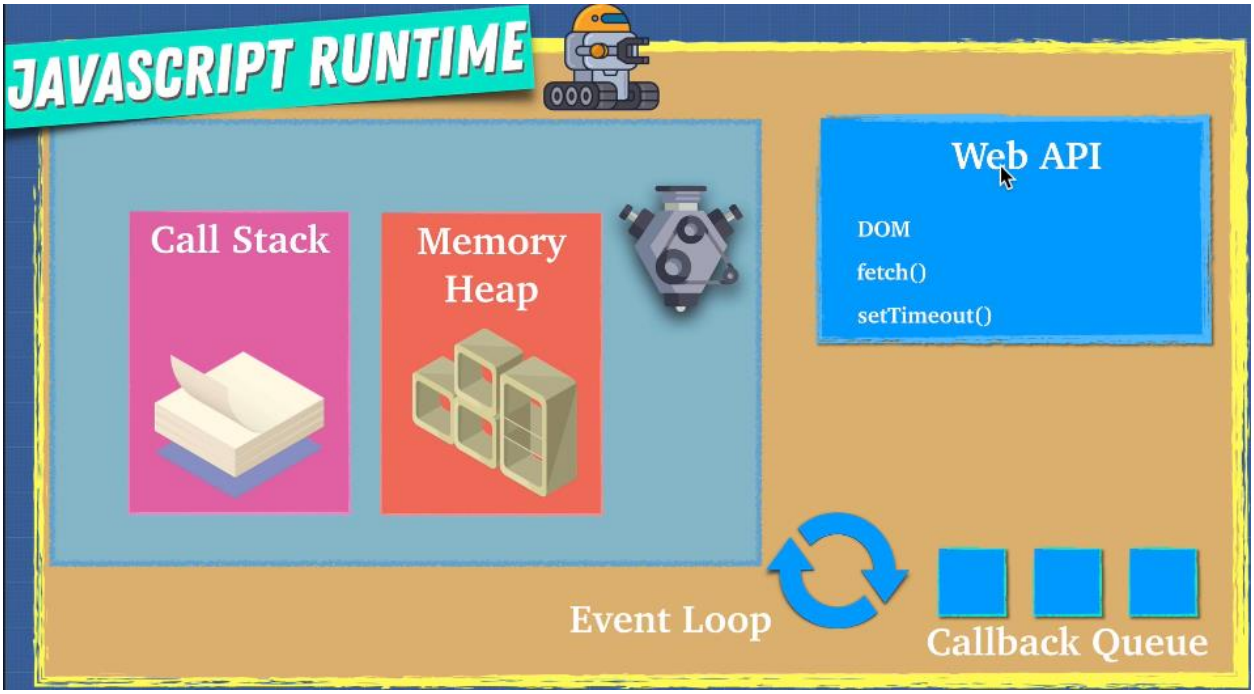
130. Section Overview:



- Asynchronous functions are **functions that we can execute later**.
- In this section we're going to finalize our understanding of asynchronous JavaScript and revisit the problem that we saw in the first part of the course.
- 

- When we run this, we have this weird output.
- **JavaScript engine reading our code** setTimeout () says hey I'm calling the web API again for setTimeout () I'm calling a web API.
- **For Promise I'm not sure** what I'm doing here but this is asynchronous. So, we're going to come back to it and then we run third console.log () first.

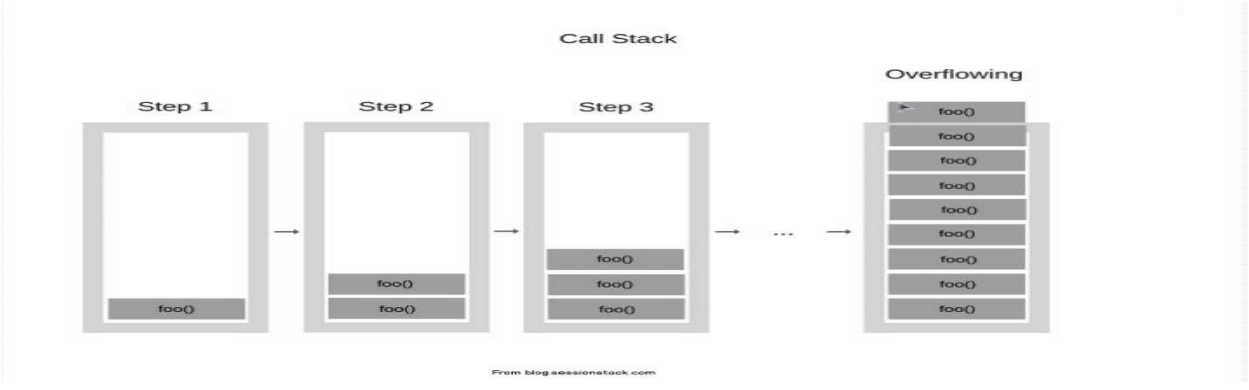
- **Remember this diagram once our JavaScript engine** sees something that is asynchronous or something like setTimeout that is part of a web API we send it over to the web API and then the web API is going to do something for us in the background when it's done with whatever it is such as a timeout or five seconds it will add the callback or the function that we need to invoke into the callback queue.



- **And then the event loop** is going to check and see if our call stack is empty and our entire JavaScript file has been read once and then if it's empty it pushes it onto the call stack. So, the event loop simply monitors the call stack and the callback
- **But then we had this problem** right where we expect three is a crowd to get printed first, but next part gets confusing.
- **For some reason the Promise console.log gets run first**. Then program returns undefined. So, this entire JavaScript file has been read and only then do we get this set timeout and then another set timeout block.

132. How JavaScript Works:

- **V8 engine reads the JavaScript** that we write and changes into machine executable instructions for the browser.
- Now the engine consists of two parts a memory heap and a call stack.
- **Memory heap**: This is where the memory allocation happens.

- Well by having unused memory just laying around it fills up this memory heap and that's why you might hear why global variables are bad.
- **Global variables are bad** because if we forget to clean up after ourselves, we fill up this memory heap and eventually the browser will not be able to work.
- **Call stack**: This is where your code is read and executed. It tells you where you are in the program.
- **Single threaded means** that it has only one call stack and one call stack can only do one thing at a time. And as you saw call stack is first in last out.
- So, whatever is at the top the call stack gets run first then below then below till the call stack becomes empty.
- **Other languages can have multiple call stacks**, and these are called multi-threaded. You can also see how that might be beneficial to have multiple call stack so that we don't keep waiting around for stuff.
- **Why was JavaScript designed to be single threaded**: Running code on a single thread can be quite easy since you don't have to deal with complicated scenarios that arise in multi-threaded environment. You just have one thing to worry about.
- **And when I say issues with multi-threaded environment can have such thing as deadlocks**.
- **Synchronous programming simply means** line one gets executed then line 2 gets executed and then line 3 gets executed. The latter can start before the first finishes.
- **Stack Overflow:**
- 

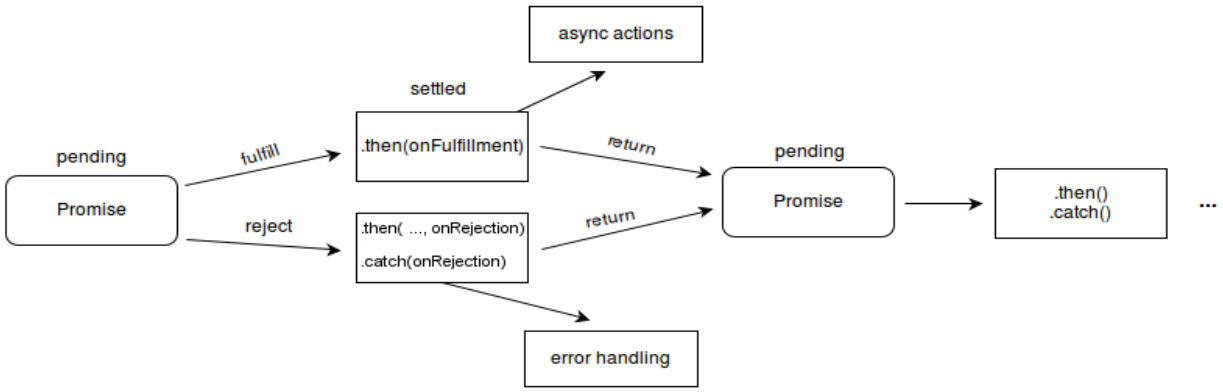
- 

- This happens when the call stacks just get bigger and bigger and bigger until it just doesn't have enough space anymore.
- **Q. But there is a problem now isn't it**. What if Line 2 was a big task before line 3 and line 4 We needed to do loop through an array that had millions of items what would happen there.
- **Line 3 will take a long time to get execute**. And the user wouldn't be able to do anything. The Web site would pretty much freeze until that task is done and the user just waits there. That's not very good.
- **Well with synchronous task if we have** one function that takes up a lot of time it's going to hold up the line.
- **Imagine a buffet restaurant**. If all the people want to eat but Bobby says Hold on guys have to keep eating and putting bacon on my plate. Well everybody must wait in line. So, sounds like we need something non-blocking.
- Remember our first statement that we made in this **video JavaScript as a single threaded language that can be non-blocking.**
- Ideally, we don't wait around for things that take time. **So how do we do this well asynchronous is here to rescue.**
- **synchronous execution is great because is predictable**. We know what happens first then what happens next what happens third, but it can get slow.
- **So, when we must do things like image processing** or making requests over the network like API calls, we need something more than just synchronous tasks.
- **We can do asynchronous programming by doing something like this:** Set time out which is a function that comes within our browsers and it allows us to create a timeout and we can just give it the first parameter is the function that we want to run and then the second parameter is how many seconds we want to wait.
- **We need a JavaScript runtime environment** and JavaScript runtime environment is again part of the browser. It's included in the browsers. They have extra things on top of the engine.

- They have something called a **web API as callback queue and an event loop** and as you can see here **set timeout is part of the web API**. It's not technically part of JavaScript.
- It's what the browsers give us to use so we can do asynchronous programming.

133. Promises:

- **We have Promise instead of callbacks**. The Promise object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.
- A Promise is in one of these states:
- **Pending**: initial state, neither fulfilled nor rejected.
- **fulfilled**: meaning that the operation completed successfully.
- **rejected**: meaning that the operation failed.
- **A pending promise can either be fulfilled with a value or rejected** with a reason (error). When either of these options happens, the associated handlers queued up by a promise's then method are called.
- As the **Promise.prototype.then() and Promise.prototype.catch()** methods return promises, they can be chained.



- **Now in order to fully grasp the concept must** first talk about what we had before promises and that is callbacks.
- **Callbacks are something like this when something is done, execute this piece of code.**

```
el.addEventListener("click", submitForm);

// callback pyramid of doom
movePlayer(100, 'Left', function() {
  movePlayer(400, 'Left', function() {
    movePlayer(10, 'Right', function() {
      movePlayer(330, 'Left', function() {
      });
    });
  });
});
```

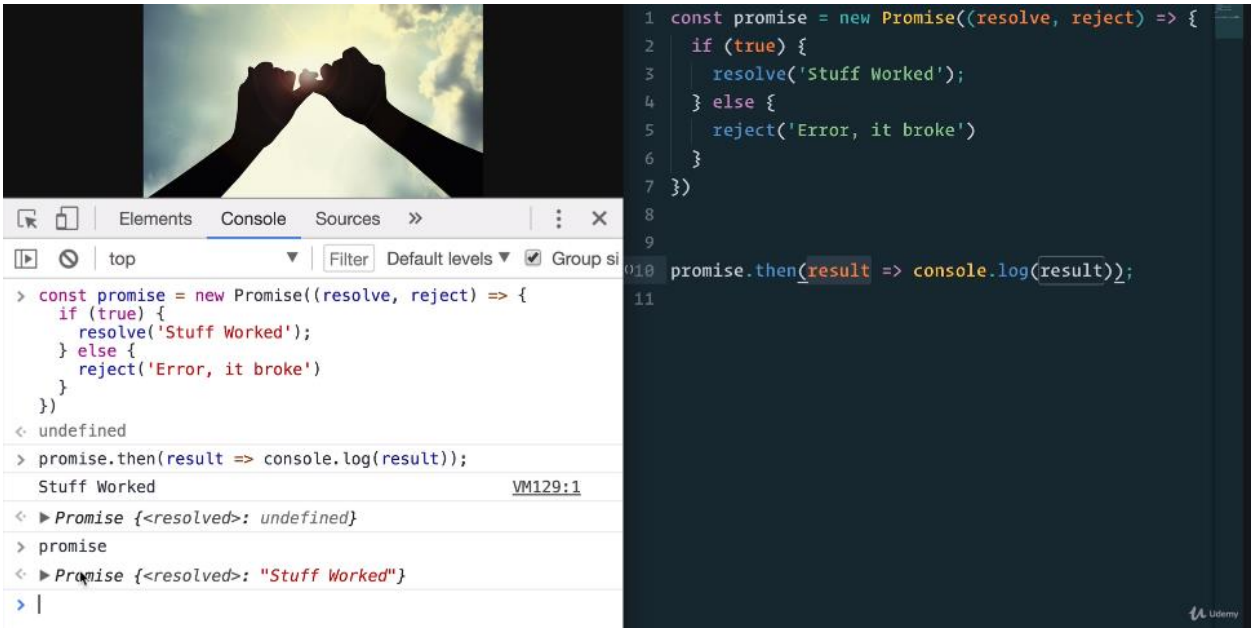
- **Another Example:**

```
grabTweets('twitter/andreineagoie', (error, andreiTweets) => {
  if(error) {
    throw Error;
  }
  displayTweets(andreiTweets)
  grabTweets('twitter/elonmusk', (error, elonTweets) => {
    if(error) {
      throw Error;
    }
    displayTweets(elonTweets)
    grabTweets('twitter/vitalikbuterin', (error, vitalikTweets) => {
      if(error) {
        throw Error;
      }
      displayTweets(vitalikTweets)
    }
  }
}
```

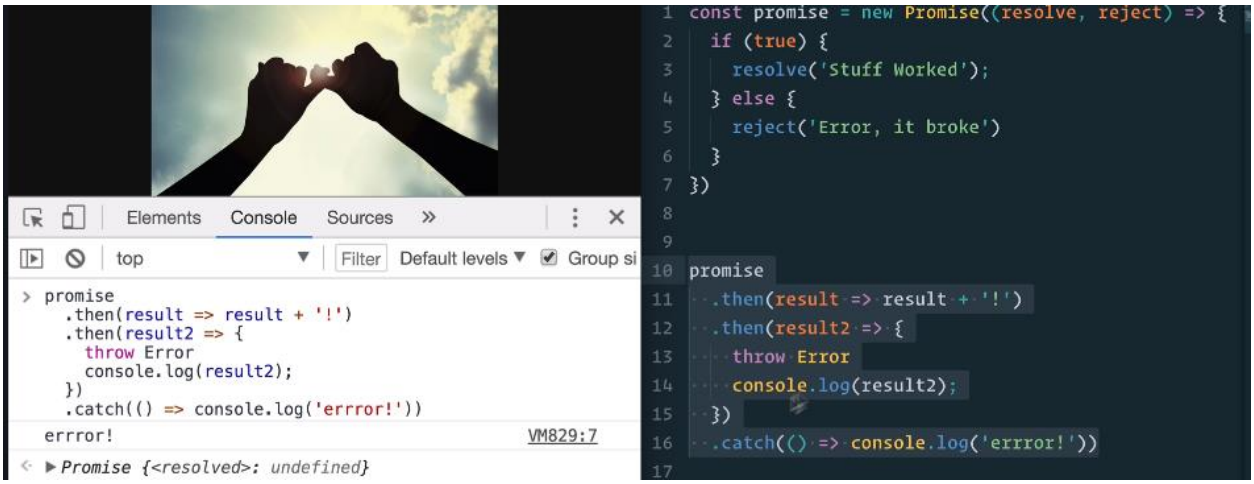
- And this doesn't look very pretty. I mean we must do the same thing we have to check for error and at each time we have it nested and just overall, **we have a lot of repetition of code** now promises to serve the same purpose as callbacks.
- **So why do we have two things**: What promises are new in the S6 and they're a little bit more powerful.
- **Earlier movePlayer example with Promise**: look much better and easy to understand and maintain.

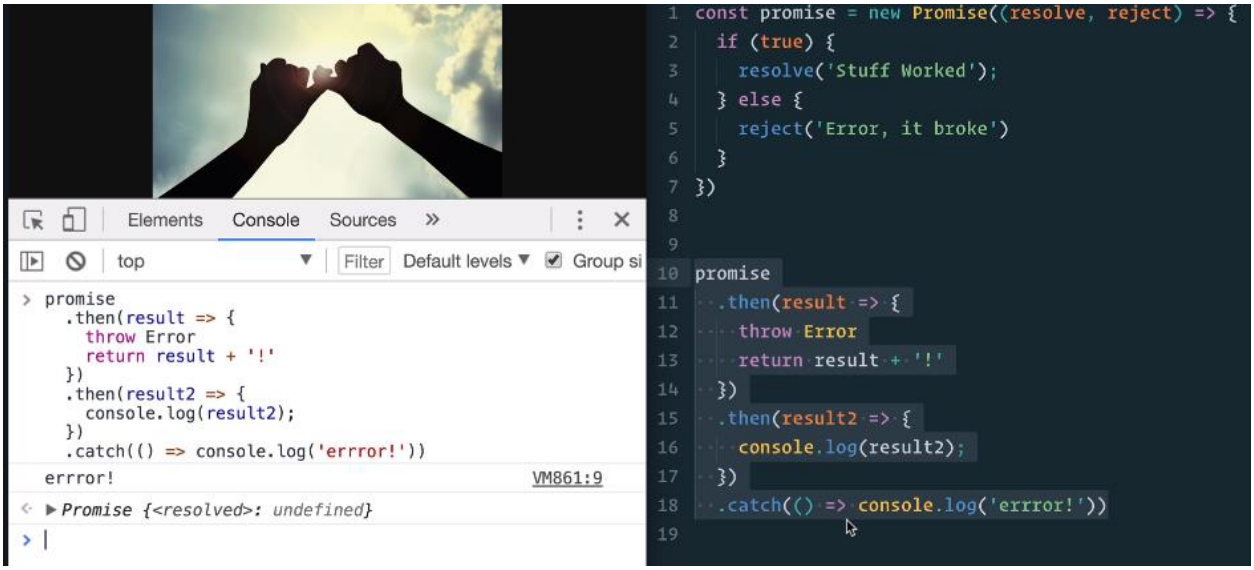
```
movePlayer(100, 'Left')
  .then(() => movePlayer(400, 'Left'))
  .then(() => movePlayer(10, 'Right'))
  .then(() => movePlayer(330, 'Left'))
```

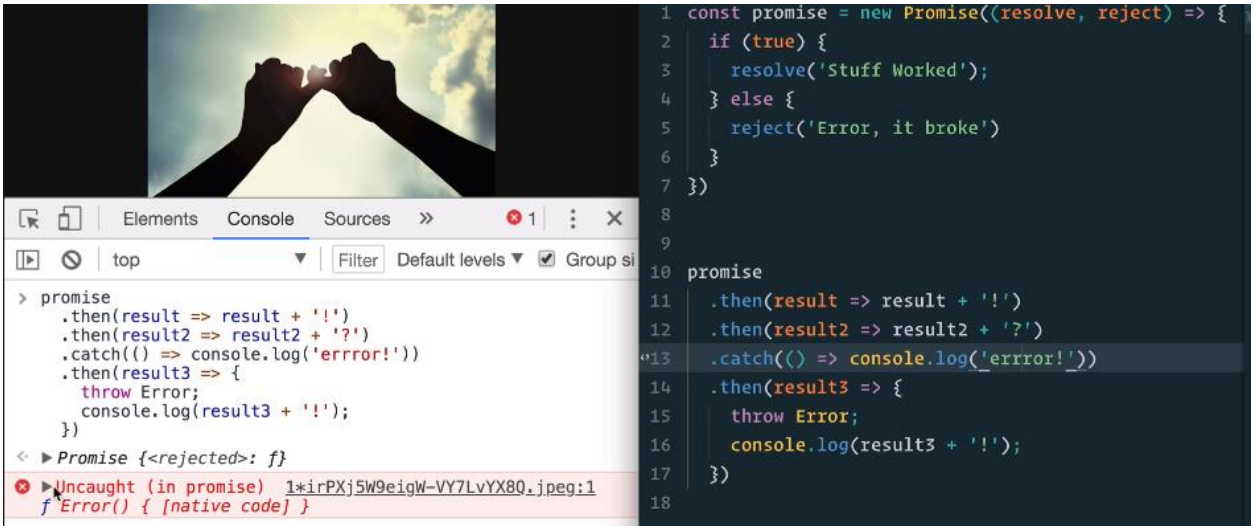
- **Create a promise:**



- **Chaining with then and catch:**

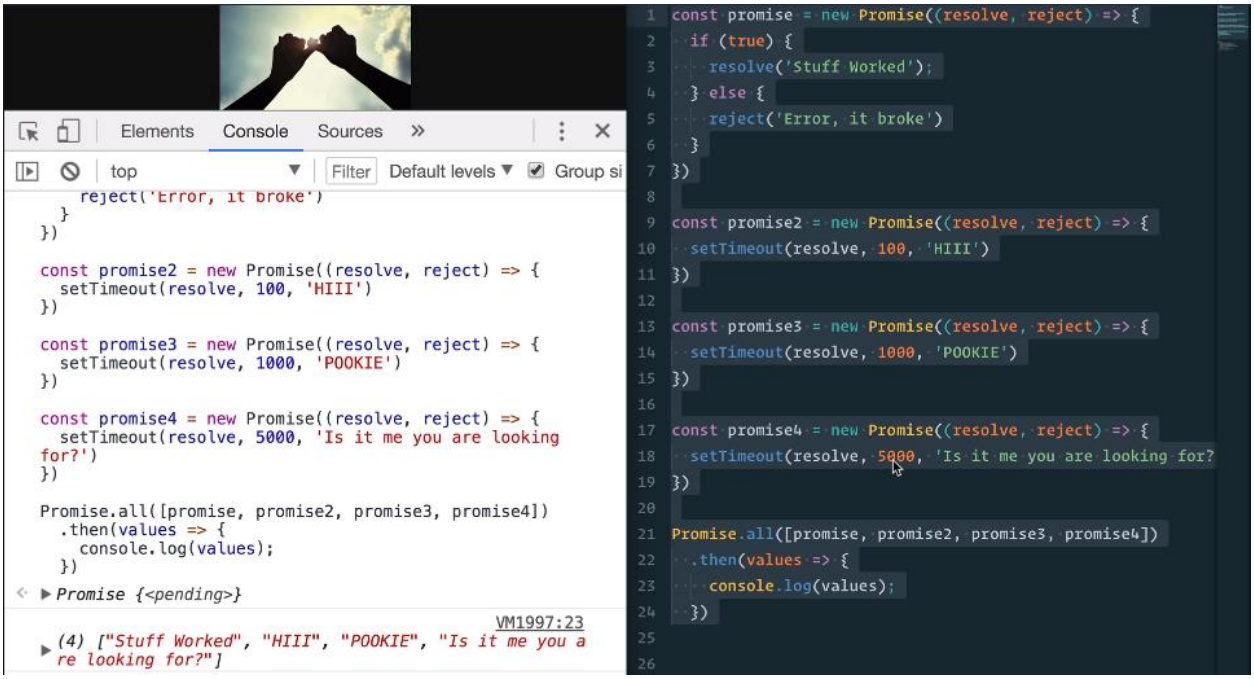


- 

- **‘. catch’ catches** any errors that may happen between the chains of ‘. then’.
- **‘. catch’ before ‘. then’ example:**
- 

- **Well we don't get the error console**. We get an error within our console because we did throw an error. The catch statement never runs.
- So, where you put the catch statement, **it's going to check and run if anything before it fails.**
- And so far, it's only been resolved. We've never run the reject.
- **When you don't want JavaScript to block the execution of your code like making API calls, grabbing data from a database, optimizing an image**, you use a promise so that the task happens in the background when the promise gets resolved or reject then you'll get that response.

- **Promise.all:** It takes an array of promises. In our we will pass promise, promise2, promise3 and promise4. And here we can do a '. then'.
- **And the values will be returned** as an array in the order that they were just written down. And here we can just console.log these values.
-



- And we have our result five seconds later. You see here that even though these ones took a lot less well because we did promise that all it waited until all the promises were resolved and then logged out the values
- **Real Example:**
-

```
const urls = [
  'https://jsonplaceholder.typicode.com/users',
  'https://jsonplaceholder.typicode.com/posts',
  'https://jsonplaceholder.typicode.com/albums'
]

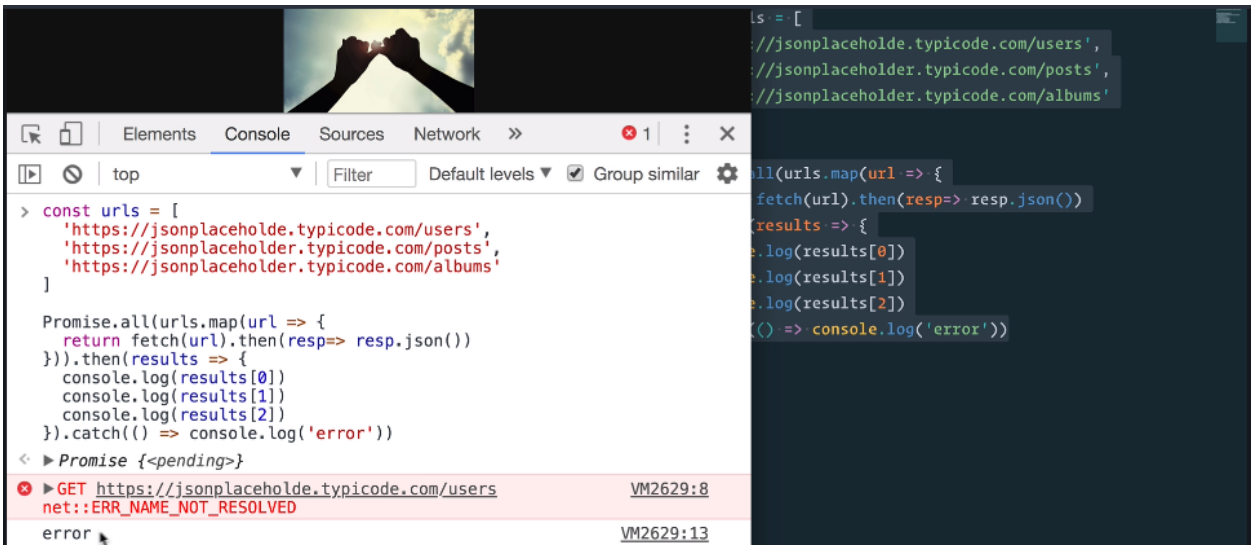
Promise.all(urls.map(url => {
  return fetch(url).then(resp=> resp.json())
})).then(results => {
  console.log(results[0])
  console.log(results[1])
  console.log(results[2])
})
```

- **Result:**

```
> const urls = [
    'https://jsonplaceholder.typicode.com/users',
    'https://jsonplaceholder.typicode.com/posts',
    'https://jsonplaceholder.typicode.com/albums'
  ]

  Promise.all(urls.map(url => {
    return fetch(url).then(resp=> resp.json())
  })).then(results => {
    console.log(results[0])
    console.log(results[1])
    console.log(results[2])
  })
  ▶ Promise {<pending>}
  ▶ (10) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}]   VM2510:10
                                                               VM2510:11
  ▶ (100) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
                                                               VM2510:12
  ▶ (100) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
```

- **Example with wrong url:** first URL is not correct



- Look at that we get an **error** with Promise.all.
- **We get a reject from the promise** and we can catch it and we can do whatever we want with this error. That's why promises are so beneficial. We're able to do something like this in a very clean way.
- **Remember the fetch simply returns a promise** if I just run fetch here:

- ```
  > fetch('https://jsonplaceholder.typicode.com/users')
  ◂ ▶ Promise {<pending>}
  ```
- So, at their most basic, promises are a bit like event listeners. **Promise can only succeed or fail once. It cannot succeed or fail twice.**
- **So remember a promise is an object that may produce a single value sometime in the future either resolved or rejected with a reason** why it was rejected and a promise maybe in one of three possible states, It can be fulfilled rejected or pending.
- **Note**: A promise is said to be *settled* if it is either fulfilled or rejected, but not pending. You will also hear the term *resolved* used with promises — this means that the promise is settled or "locked in" to match the state of another promise.
- `Promise.all(iterable)`

  Wait for all promises to be resolved, or for any to be rejected.

  If the returned promise resolves, it is resolved with an aggregating array of the values from the resolved promises ,in the same order as defined in the iterable of multiple promises.

  If it rejects, it is rejected with the reason from the first promise in the iterable that was rejected.
- `Promise.allSettled(iterable)`

  Wait until all promises have settled (each may resolve or reject).

  Returns a promise that resolves after all of the given promises have either resolved or rejected, with an array of objects that each describe the outcome of each promise.
- `Promise.reject(reason)`

  Returns a new `Promise` object that is rejected with the given reason.
- `Promise.resolve(value)`

  Returns a new `Promise` object that is resolved with the given value. If the value is a thenable (i.e. has a `then` method), the returned promise will "follow" that thenable, adopting its eventual state; otherwise the returned promise will be fulfilled with the value.

  Generally, if you don't know if a value is a promise or not, `Promise.resolve(value)` it instead and work with the return value as a promise.
  ```

`Promise.prototype.catch()`

Appends a rejection handler callback to the promise, and returns a new promise resolving to the return value of the callback if it is called, or to its original fulfillment value if the promise is instead fulfilled.

`Promise.prototype.then()`

Appends fulfillment and rejection handlers to the promise, and returns a new promise resolving to the return value of the called handler, or to its original settled value if the promise was not handled (i.e. if the relevant handler `onFulfilled` or `onRejected` is not a function).

`Promise.prototype.finally()`

Appends a handler to the promise, and returns a new promise which is resolved when the original promise is resolved. The handler is called when the promise is settled, whether fulfilled or rejected.

- **Promise. Prototype. catch ():**

Internally calls `Promise.prototype.then` on the object upon which it was called, passing the parameters `undefined` and the received `onRejected` handler. Returns the value of that call, which is a `Promise`.

- **Promise. Prototype. then ():**

The **`then()`** method returns a `Promise`. It takes up to two arguments: callback functions for the success and failure cases of the `Promise`.

The `then` method returns a `Promise` which allows for method chaining.

- **Promise. Prototype. finally ():**

The **`finally()`** method returns a `Promise`. When the promise is settled, i.e either fulfilled or rejected, the specified callback function is executed. This provides a way for code to be run whether the promise was fulfilled successfully or rejected once the `Promise` has been dealt with.

This helps to avoid duplicating code in both the promise's `then()` and `catch()` handlers.

134. ES8 - Async Await:

- **async await is part of ES8 and is built on top of promises.**
- Underneath the hood an **async function is a function that returns a promise.** But the benefit of async await is that **it makes code easier to read**. Same asynchronous call with promise and async await:

```
// ASYNC AWAIT

movePlayer(100, 'Left')
  .then(() => movePlayer(400, 'Left'))
  .then(() => movePlayer(10, 'Right'))
  .then(() => movePlayer(330, 'Left'))

async function playerStart() {
  const firstMove = await movePlayer(100, 'Left'); //pause
  await movePlayer(400, 'Left'); //pause
  await movePlayer(10, 'Right'); //pause
  await movePlayer(330, 'Left'); //pause
}
```

- The goal with async await is to make code look synchronous a code that's **asynchronous look synchronous**.
- **JavaScript being single threaded, for programs to be efficient, we can use asynchronous programming to do these things so promises help us solve this.**
- Now remember what I said at the beginning async await code are just promises underneath the hood. **We call the syntactic sugar something that still does the same thing but is just different syntax to make it look prettier**. Promises have '. then' and async on the other hand has 'async' and 'await' keywords.
- **We can use 'await' keyword in front of any function that returns a promise.**
- From above program which we know that movePlayer return promise and once the promises resolved then the it moves to the next line and it awaits the next move.
- Instead of chaining it we can assign just like a synchronous programming variable:

```
async function playerStart() {
  const first = await movePlayer(100, 'Left'); //pause
  const second = await movePlayer(400, 'Left'); //pause
  await movePlayer(10, 'Right'); //pause
  await movePlayer(330, 'Left'); //pause
}
```

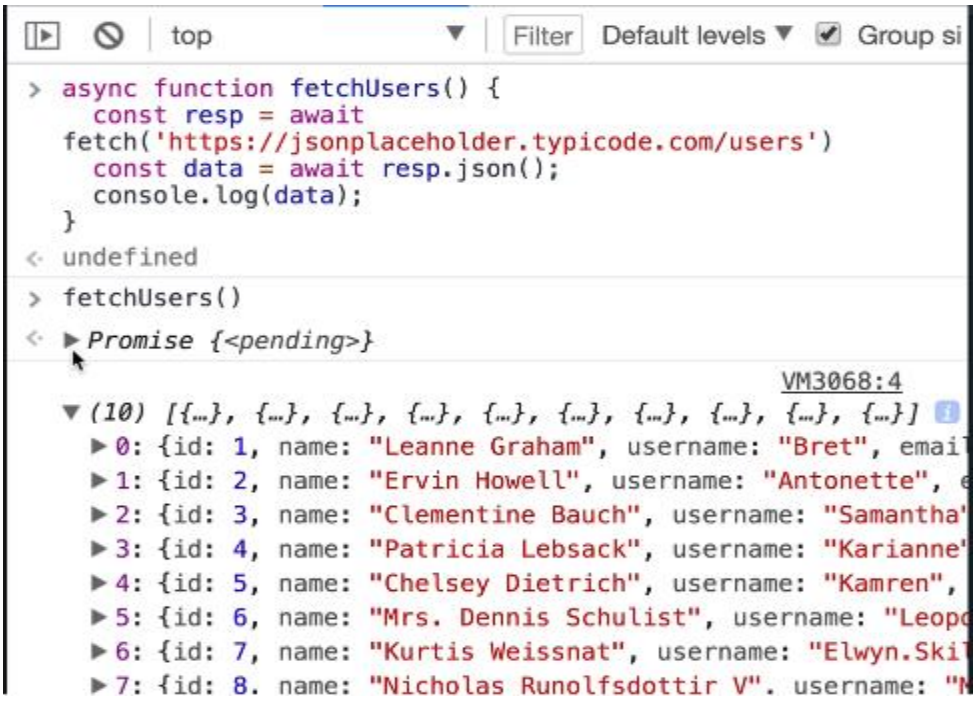
- **Let's use more realistic example:** fetch () function returns a promise

```
fetch('https://jsonplaceholder.typicode.com/users')
  .then(resp => resp.json())
  .then(console.log)
```

- **Q. How can we turn this into async and await function?** The same feature will work the same, but it will look a little bit cleaner. It will be syntactic sugar.

```
1  // ASYNC AWAIT
2
3  async function fetchUsers() {
4    const resp = await fetch('https://jsonplaceholder.typicode.com/users')
5    const data = await resp.json();
6    console.log(data);
7  }
```

- **Result:**



- We have our promise that got resolved and we have our users **nothing different from the promise but now we have a nice step by step synchronous looking code.** That says fetch this api call, get a response then run it through the Json method and then console log.
- **Promise vs async await:**
-

```
// ASYNC AWAIT

fetch('https://jsonplaceholder.typicode.com/users')
  .then(resp => resp.json())
  .then(console.log)

async function fetchUsers() {
  const resp = await fetch('https://jsonplaceholder.typicode.com/users')
  const data = await resp.json();
  console.log(data);
}
```

- **One more example:**
-

```
const urls = [
  'https://jsonplaceholder.typicode.com/users',
  'https://jsonplaceholder.typicode.com/posts',
  'https://jsonplaceholder.typicode.com/albums'
]

Promise.all(urls.map(url =>
    fetch(url).then(resp => resp.json())
)).then(array => {
  console.log('users', array[0])
  console.log('posta', array[1])
  console.log('albums', array[2])
}).catch('oops');

const getData = async function() {
  const [ users, posts, albums ] = await Promise.all(urls.map(url =>
      fetch(url).then(resp => resp.json())
  ))
  console.log('users', users)
  console.log('posts', posts)
  console.log('albums', albums)
}
```

- **Result:**

```
> const getData = async function() {
    const [ users, posts, albums ] = await
  Promise.all(urls.map(url =>
        fetch(url).then(resp => resp.json())
    ))
    console.log('users', users)
    console.log('posts', posts)
    console.log('albums', albums)
  }
<- undefined
> getData()
<- ▶ Promise {<pending>}
   users                                                    VM3455:5
   ▶ (10) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}]
   posts                                                    VM3455:6
   ▶ (100) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
   albums                                                   VM3455:7
   ▶ (100) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
```

- **We got the exact same results**. But there might be one thing that you noticed here that is in the promised way we have the '. Catch' if any of these fails, we want to catch these errors.
- **Q. How can we do that with async await?**
- **JavaScript has something called try catch blocks**: first URL is **erroneous**:

```
  'https://jsonplaceholde.typicode.com/users',
  'https://jsonplaceholder.typicode.com/posts',
  'https://jsonplaceholder.typicode.com/albums'
]

Promise.all(urls.map(url =>
    fetch(url).then(resp => resp.json())
)).then(array => {
  console.log('users', array[0])
  console.log('posta', array[1])
  console.log('albums', array[2])
}).catch('oops');

const getData = async function() {
  try {
    const [ users, posts, albums ] = await Promise.all(urls.map(url =>
        fetch(url).then(resp => resp.json())
    ))
    console.log('users', users)
    console.log('posts', posts)
    console.log('albums', albums)
  } catch (err) {
    console.log('oops', err)
  }
}
```

- **Result:**

```
> const getData1 = async function() {
    try {
      const [ users, posts, albums ] = await
  Promise.all(urls.map(url =>
          fetch(url).then(resp => resp.json())
      ))
      console.log('users', users)
      console.log('posts', posts)
      console.log('albums', albums)
    } catch (err) {
      console.log('oops', err)
    }
  }
< undefined
```

```
> getData1()
< ▶ Promise {<pending>}
✖ ▶GET https://jsonplaceholde.typicode.com/users VM3720:4
  net::ERR_NAME_NOT_RESOLVED
  oops TypeError: Failed to fetch                VM3720:10
```

- So, we got the cache block with the error just like we got with Promise.all just using promises.

135. ES9 (ES2018)

- **Object spread operator:**

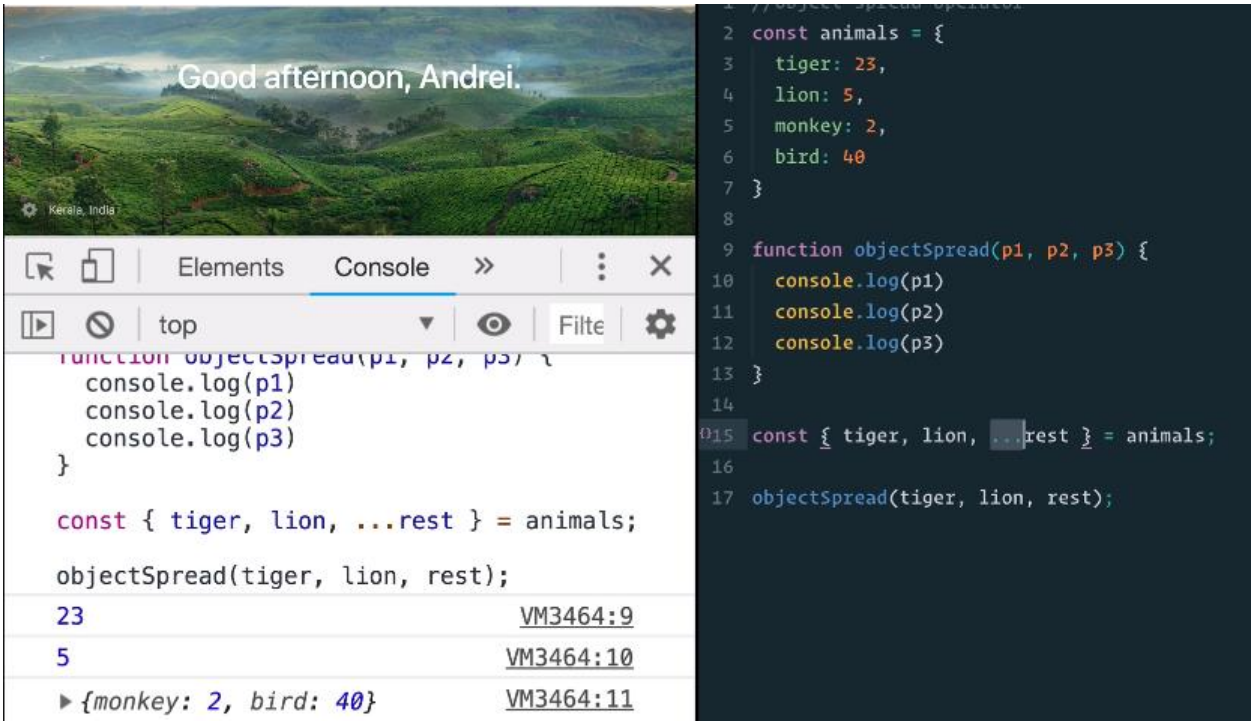
```
> const animals = {
    tiger: 23,
    lion: 5,
    monkey: 2
  }

  const { tiger, ...rest } = animals;
< undefined
> tiger
< 23
> rest
< ▶ {lion: 5, monkey: 2}
>
```

- **This is something that we were able to do with arrays in ES6:**

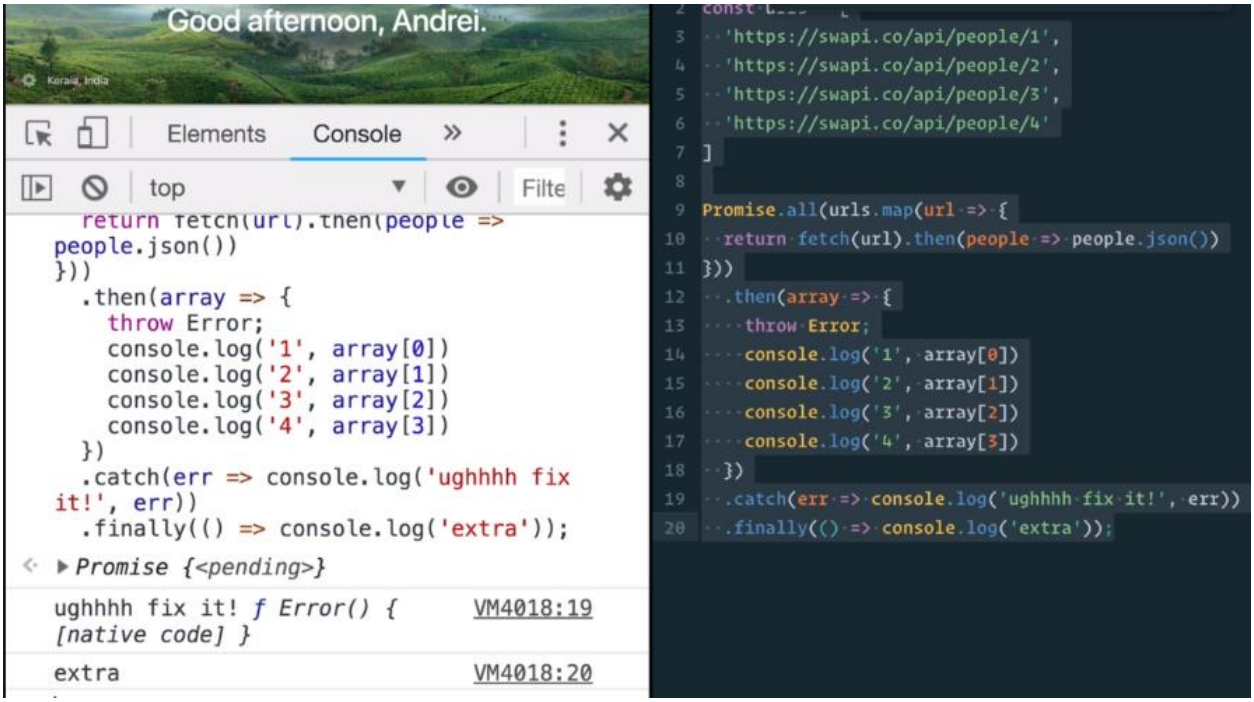
```
> const array = [1,2,3,4,5];
  function sum (a, b, c, d, e) {
    return a + b + c + d + e;
  }
<· undefined
> sum(...array);
<· 15
> sum(1,2,3,4,5)
<· 15
```

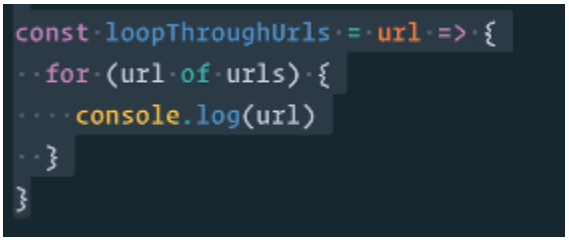
- **Object Spread Operator example:**



136. ES9 (ES2018) – Async:

- **Finally block with promise:**
- This finally block will be called regardless of whether '. then' works or the promise error catches. So, no matter what after everything is done inside of a promise finally will be called, whether it resolves or rejects.

- 

- **'For await of' feature**:
- This new feature is that it allows us to loop through our async await calls if we have multiple of them just like we are able to use the 'for of' so using the 'for of' loop that allowed us to iterate over iterables. We're now able to iterate over the await promises that we're going to have.
- **'For of' example for array:**
- 

- **‘for of’ example for ‘awaits’:**

```javascript
// for await of
const urls = [
  'https://jsonplaceholder.typicode.com/users',
  'https://jsonplaceholder.typicode.com/postss',
  'https://jsonplaceholder.typicode.com/albums'
]
const getData = async function() {
  try {
    const [ users, posts, albums ] = await Promise.all(urls.map(async function(url) {
        const response = await fetch(url);
        return response.json();
    }));
    console.log('users', users);
    console.log('posts', posts);
    console.log('albums', albums);
  } catch (err) {
    console.log('ooooooops', err);
  }
}

const getData2 = async function() {
  const arrayOfPromises = urls.map(url => fetch(url));
  for await (let request of arrayOfPromises) {
    const data = await request.json();
    console.log(data;)
  }
}
```

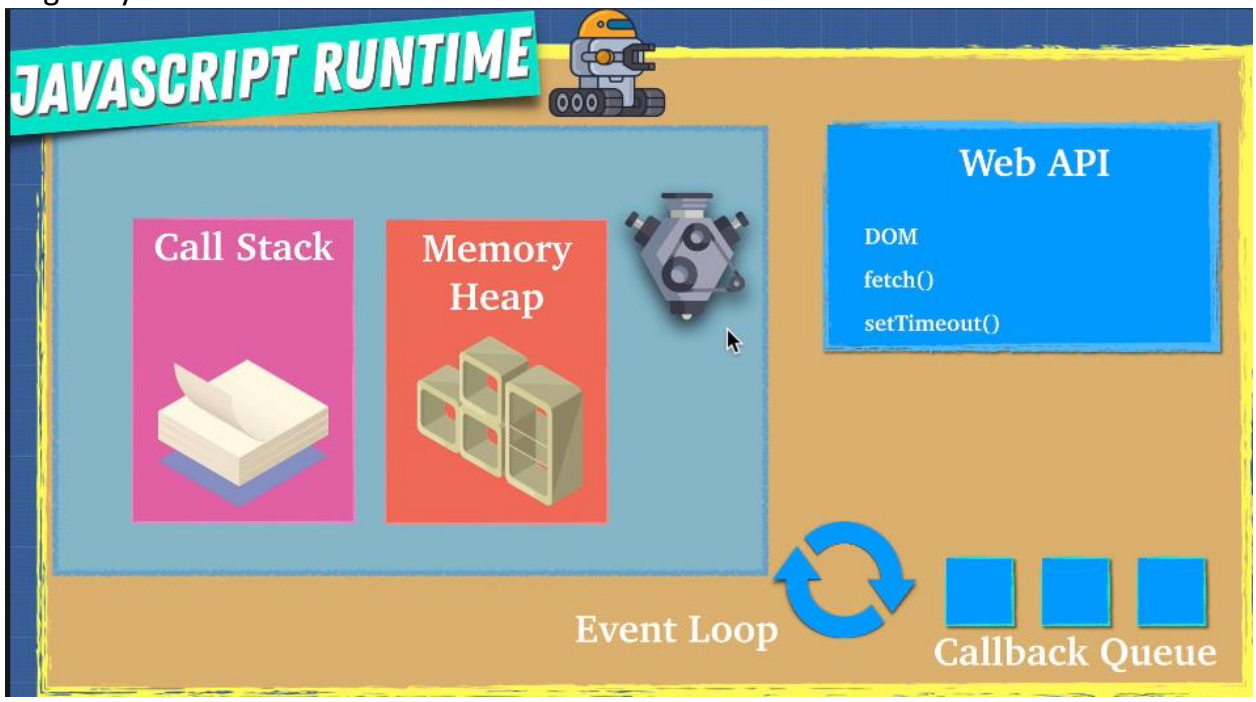
- **Result:**

```
> getData2()
<· ▶ Promise {<pending>}
                                    VM4925:5
  ▶ (10) [{…}, {…}, {…}, {…}, {…},
    {…}, {…}, {…}, {…}, {…}]
                                    VM4925:5
    (100) [{…}, {…}, {…}, {…}, {…},
    {…}, {…}, {…}, {…}, {…}, {…}, {…}
    , {…}, {…}, {…}, {…}, {…}, {…},
    {…}, {…}, {…}, {…}, {…}, {…}, {…}
    , {…}, {…}, {…}, {…}, {…}, {…},
    {…}, {…}, {…}, {…}, {…}, {…}, {…}
    , {…}, {…}, {…}, {…}, {…}, {…},
  ▶ {…}, {…}, {…}, {…}, {…}, {…}, {…}
    , {…}, {…}, {…}, {…}, {…}, {…},
    {…}, {…}, {…}, {…}, {…}, {…}, {…}
    , {…}, {…}, {…}, {…}, {…}, {…},
    {…}, {…}, {…}, {…}, {…}, {…}, {…}
```

```javascript
        const response = await fetch(url);
        return response.json();
    }));
    console.log('users', users);
    console.log('posts', posts);
    console.log('albums', albums);
  } catch (err) {
    console.log('ooooooops', err);
  }
}

const getData2 = async function() {
  const arrayOfPromises = urls.map(url => fetch(url));
  for await (let request of arrayOfPromises) {
    const data = await request.json();
    console.log(data);
  }
}
```

- The only thing that the **'for await of'** feature does it allows us to loop through these multiple promises almost **as if we're writing synchronous code**.

137. Job Queue:

- **This diagram here of our JavaScript runtime** that you'll see on the web all over the place has a missing piece and it's missing because well this is how it was originally like.
- 

- **But as of ES6 with the new updates** in JavaScript came another piece of the JavaScript runtime that often doesn't get mentioned in older resources.

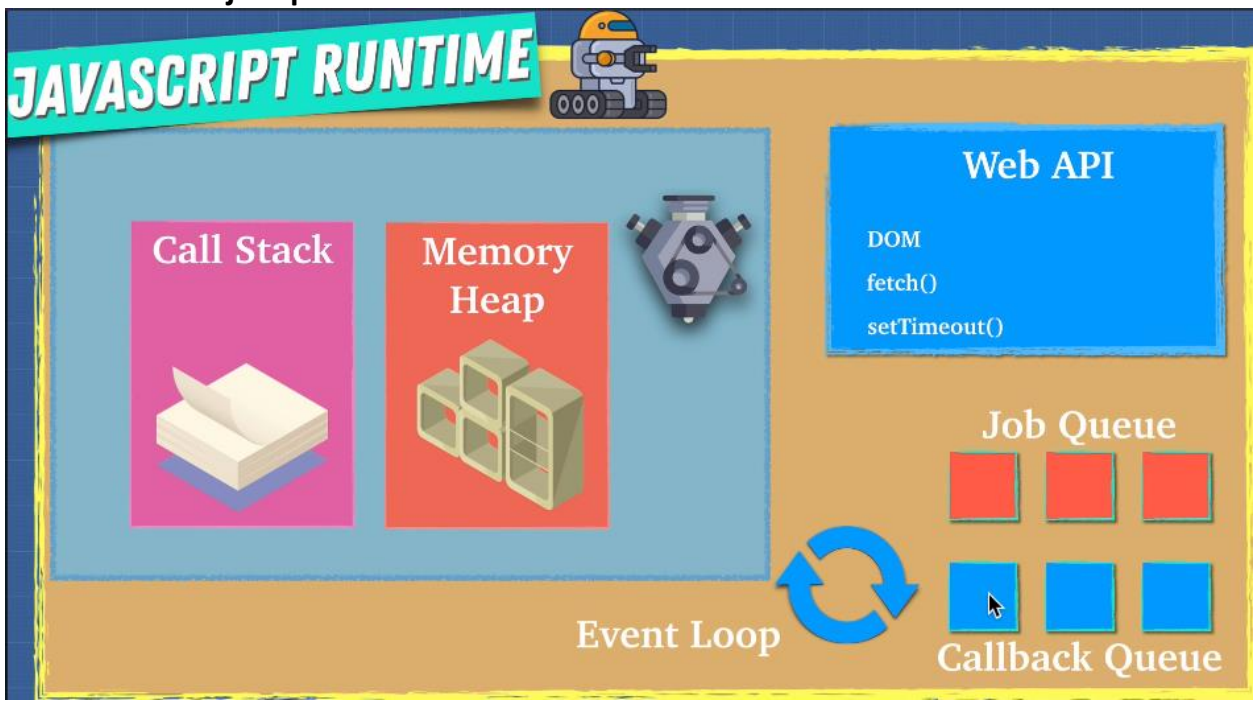
- **You see promises are new** to JavaScript or they were added quite recently and to accommodate this new addition we had to change the event loop.

- **You see the event loop had callback queue**. It was also called the task queue but with promises we had this thing natively in JavaScript. Now instead of just using callbacks we now had a native way to handle asynchronous code using promises. So, it wasn't really part of the web API.

- It's part of JavaScript, **ECMAScript the JavaScript committee said well we need another queue for our promises and that is the Job queue or Microtask queue.**
- This queue is like the callback queue just a little smaller but has a higher priority than callback queue.

- **It means that the event loop is going to check the job queue first and make sure that there's nothing in that queue before it starts looking at the callback queue because job queue has higher priority than callback queue.**

- So that our **diagram of the JavaScript runtime** looks something like **this with new addition of the job queue.**



- **This job queue is now just like the callback queue** in our JavaScript runtime implemented by the browser. But the event loop is going to check the job queue first to make sure that it is empty before we start putting some of the callback queue functions onto the call stack.

- **Now our code should make sense:**

```
main.js   saved
1  //Callback Queue - Task Queue
2  setTimeout(()=>{console.log('1', 'is the loneliest
   number')}, 0)
3  setTimeout(()=>{console.log('2', 'can be as bad as
   one')}, 10)
4
5  //2 Job Queue - Microtask Queue
6  Promise.resolve('hi').then((data)=> console.log('2',
   data))
7
8  //3
9  console.log('3','is a crowd')
```

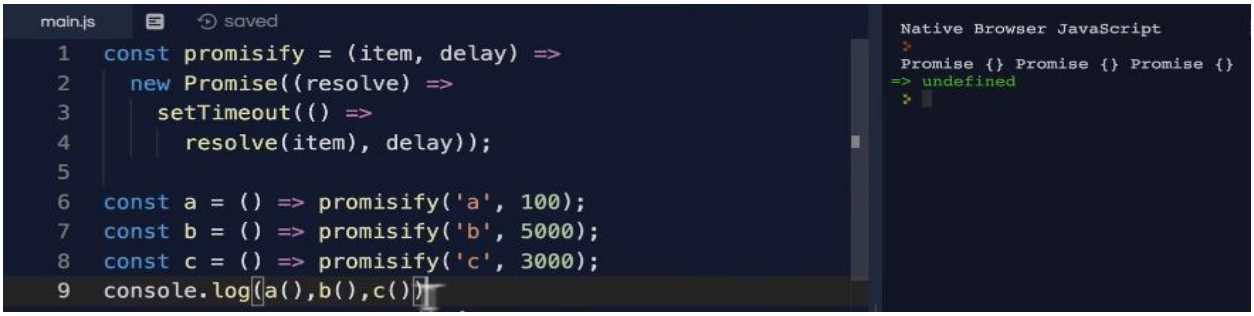
```
Native Browser JavaScript
3 is a crowd
2 hi
=> undefined
1 is the loneliest number
2 can be as bad as one
```

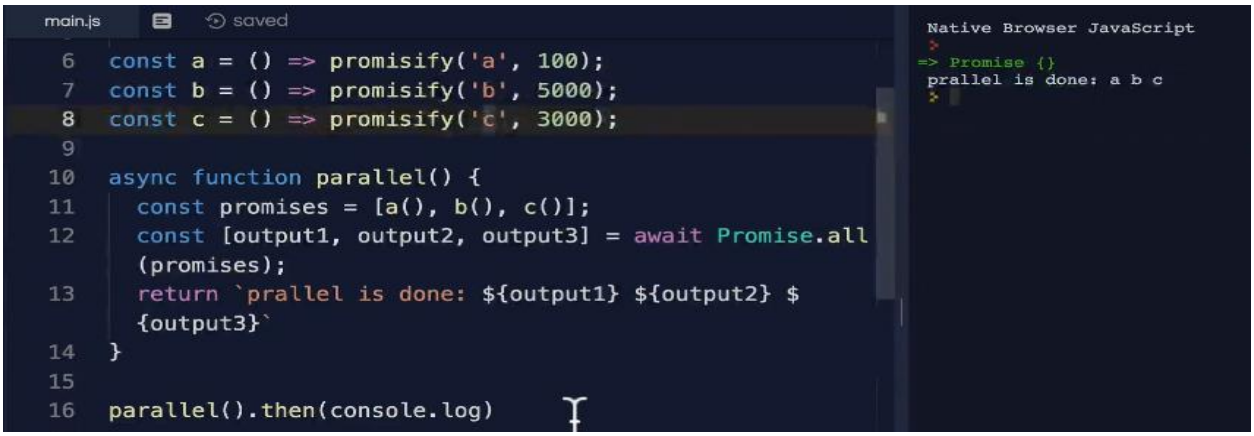
- **Because when we run our program first we call 'setTimeout'** which is something called a facade function because it isn't actually JavaScript, it just looks like JavaScript but underneath the hood it's a web API that calls the timer API to run for 0 seconds.
- **And then we get second** setTimeout which again calls the web API. And it's a facade function that triggers the timer in the browser to run for 10 milliseconds.
- **And as those things are being** processed, we go to the next line Promise.resolve that says Well that it is for the job queue. It's a promise. So, we're going to send it to the job queue. And once it's resolved we want you to console log.
- **So now these three above statements are outside** of our world and we get to console log and is says '3 is a crowd'. So, this gets logs first because it's not asynchronous.
- **And then the event loop says well the job queue has the priority**. So, I'm not even going to bother checking the callback you first. I'm going to check the job queue. Hey job queue does you have anything for me.
- **And the job queue is going to** say I've been waiting for you. Here's some console log for you. So, we log this out.
- **Once the job queue is cleared event loop will start looking into callback queue for our setTimeout statements.**
- **Hey, the first setTimeout are you done** with your work, Yep I'm done here then we console log our first setTimeout. Then it checks the next setTimeout Hey are you done with this. Yep I'm done here then we console log our second setTimeout.
- **So, you see because of this new job queue, even though setTimeout was before the promise but promises are part of job queue and this job queue gets checked first because it has higher priority.**
- Now the interesting thing with this implementation is that **it's implemented by the browser.** And we have different browsers and although the implementation is quite standard across all browsers, now you'll still get some few weird kinks because browsers implement this differently where this might not work sometimes as you expect.
- **For example, some legacy browsers might not even have the job queue and might just have the callback queue.**

- So that's something that we don't want to rely too heavily on, but we do want to be aware that **as time progresses, we will have these two queues to look out for.**

138. Parallel, Sequence and Race:

- **Let's say you had three promises that you need to handle. There are a few ways that we can manage this.**
- **One is parallel** that is I want you to execute all three of these promises or let's say Fetch calls or anything they are I want you to run them in parallel all at the same time.
- **Another way that we might want to run them is sequential**. Maybe we want to run the first one and if the first one succeeds then the second one then if the second one succeeds then the third one.
- **Another way is called a race**. That is, I want you to call three things but whichever one comes back first just do that one and ignore the rest.

- **Code Base for all three scenarios**:
- 

- **Parallel Execution:**
- 

- **Race condition Example 1:**

```
main.js                                    saved
11    const promises = [a(), b(), c()];
12    const [output1, output2, output3] = await Promise.all
      (promises);
13    return `prallel is done: ${output1} ${output2} $
      {output3}`
14  }
15
16  async function race() {
17    const promises = [a(), b(), c()];
18    const output1 = await Promise.race(promises);
19    return `race is done: ${output1}`;
20  }
21
22  race().then(console.log)
23
```

```
Native Browser JavaScript
> 
=> Promise {}
race is done: a
> 
```

- **Race condition Example 2(with "promisify('b',0)"):**

```
main.js                                    saved
1   const promisify = (item, delay) =>
2     new Promise((resolve) =>
3       setTimeout(() =>
4         resolve(item), delay));
5
6   const a = () => promisify('a', 100);
7   const b = () => promisify('b', 0);
8   const c = () => promisify('c', 3000);
```

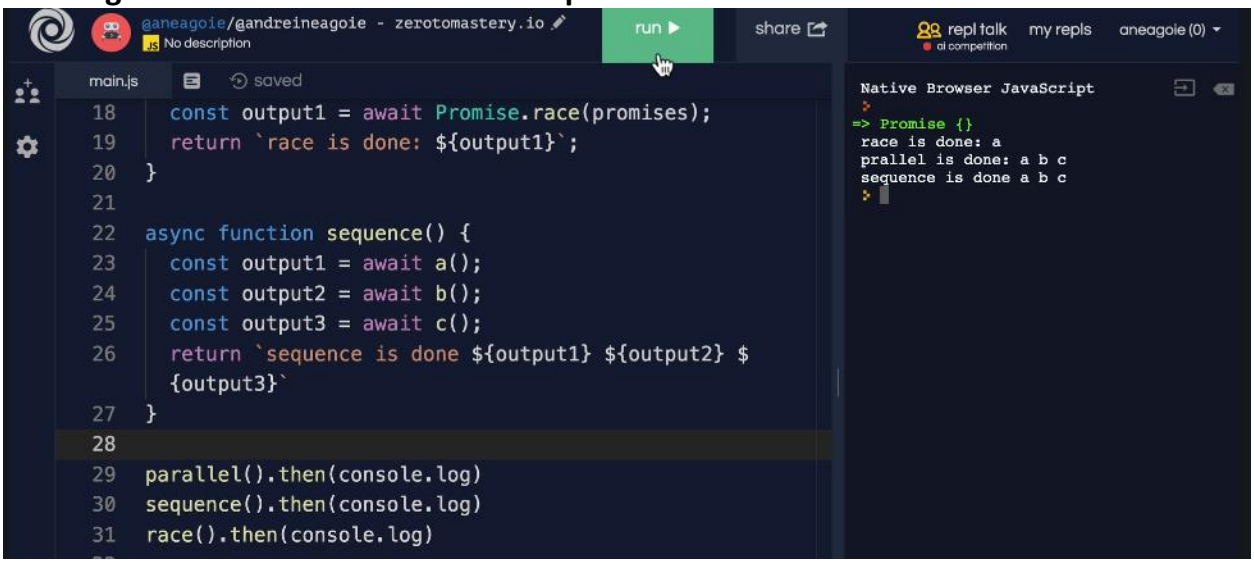
```
Native Browser JavaScript
> 
=> Promise {}
race is done: b
> 
```

- **Sequential Example:**

```
main.js                                    saved
18    const output1 = await Promise.race(promises);
19    return `race is done: ${output1}`;
20  }
21
22  async function sequence() {
23    const output1 = await a();
24    const output2 = await b();
25    const output3 = await c();
26    return `sequence is done ${output1} ${output2} $
      {output3}`
27  }
28
29  sequence().then(console.log)
```

```
Native Browser JavaScript
> 
=> Promise {}
sequence is done a b c
> 
```

- **Running all the scenarios with an example:**



- **Complete Code of all three scenarios:**

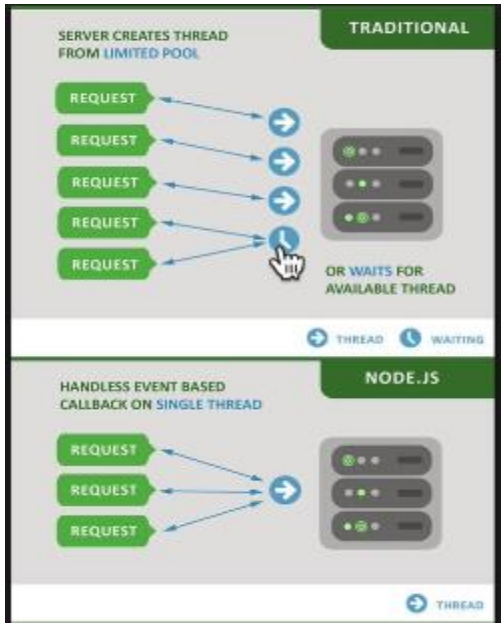
main.js

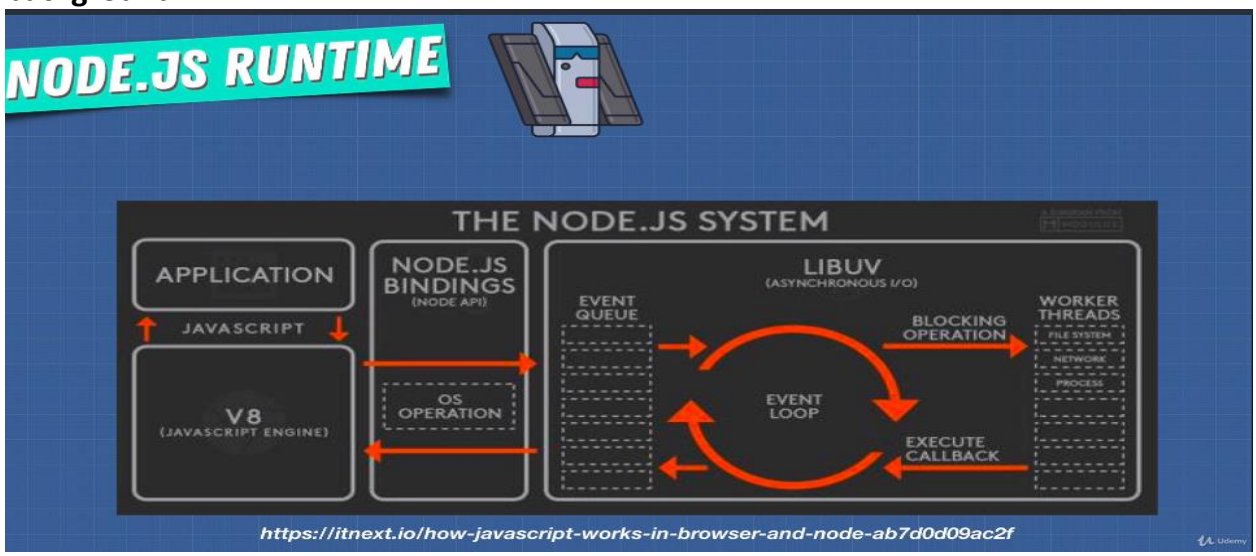
```
1  const promisify = (item, delay) =>
2    new Promise((resolve) =>
3      setTimeout(() =>
4        resolve(item), delay));
5
6  const a = () => promisify('a', 100);
7  const b = () => promisify('b', 5000);
8  const c = () => promisify('c', 3000);
9
10 async function parallel() {
11   const promises = [a(), b(), c()];
12   const [output1, output2, output3] = await Promise.all(promises)
     ;
13   return `prallel is done: ${output1} ${output2} ${output3}`
14 }
15
16 async function race() {
17   const promises = [a(), b(), c()];
18   const output1 = await Promise.race(promises);
19   return `race is done: ${output1}`;
20 }
21
22 async function sequence() {
23   const output1 = await a();
24   const output2 = await b();
25   const output3 = await c();
26   return `sequence is done ${output1} ${output2} ${output3}`
27 }
28
29 sequence().then(console.log)
30 parallel().then(console.log)
31 race().then(console.log)
```

```
Native Browser JavaScript
=> Promise {}
race is done: a
prallel is done: a b c
sequence is done a b c
```

139. Threads, Concurrency and Parallelism:

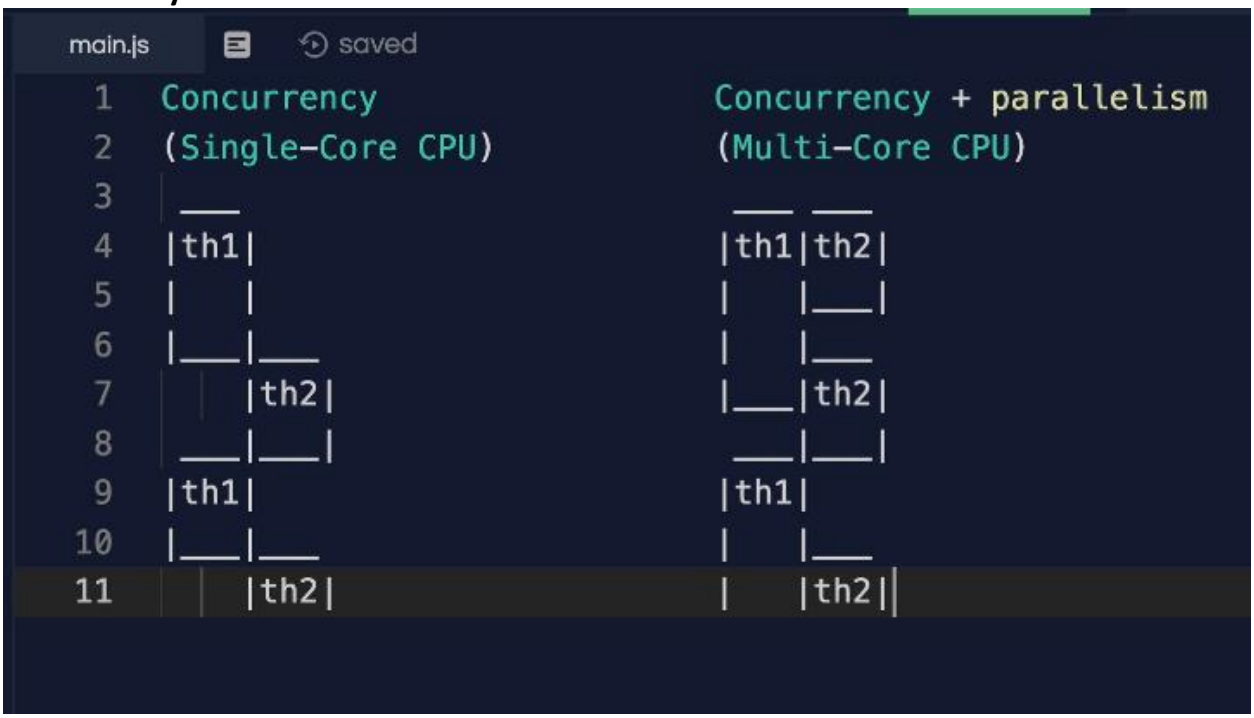
- https://www.internalpointers.com/post/gentle-introduction-multithreading
- https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers
- https://www.freecodecamp.org/news/scaling-node-js-applications-8492bd8afadc/
- **JavaScript is a single threaded language** but with the asynchronous ability we're able to do things in the background so that even though we have **JavaScript that is just one thread we're able to do these complex** things and still have Web sites and programs that perform well because with the asynchronous model these requests that take a long time don't block the main thread.
- But the thing is where do they go, **where does the Web API have this compute power to run these things.**
- **And the thing is tasks in a web browser** for example or even a node where we use something like fetch or setTimeout or investigate the database or read a file system in node these are still executed in threads.
- **The thing is this is hidden from us because** often they are running on their own **separate background threads outside of JavaScript** you see when I create a new tab here the browser creates one thread per tab so that I have an entire JavaScript call stack and memory heap in here every time on new tabs.
- **And as soon as I close the tab that thread dies and is gone.** Now this Website or the JavaScript program that I'm running on the main thread but sometimes there's things that we need to do in the background and the **browser has something called Web workers that work in the background for me**. That I don't really need to know about.
- **But if there's something complicated** that needs to happen that's outside of my control, they take care of it for me.

- **Remember this diagram:**
- 

- Where I talked about the idea of a single threaded model and Node.js became **popular because even though it had this one thread it was able to pass off things into the callback queue**. So, that it never blocked this one single thread and it just passed things on two different things.
- **Unlike the traditional model where any request** that we would have towards a **server we would spawn a new thread to handle** that.
- But because that will be blocked, we would have to **spawn another thread for the next request and if we ran out of threads,** we would have to **wait for our thread pool to be available.**
- And Node.js can do this because of something **called WORKER THREADS in the background.**
- 

- Although the limitation **of the V8 JavaScript engine is single threaded model with one call stack we can pass things off to something like library that is written in C++ to handle worker threads or multiple worker threads in the background** for us.
- And just like that on the browser we also have this idea of a web worker and a **web worker is a JavaScript program running on a different thread in parallel to our main thread**.
- Now how can we web worker it's quite easy:
- 

- **And this worker is a window object** will have access to it and it can read from a JavaScript file. So, we can spawn a new Web worker.
- **The main takeaway is that web worker is a JavaScript program running on a different thread alongside our main thread** just like a browser creates a new thread for us when I create new tabs on our program.
- Now keep in mind that these **Web workers communicate through these messages** that I just showed you, but **they don't really have access to all the browser web API.**
- This is like **they don't have access to Windows or document object, but they do have some abilities like using setTimeout, location or navigator**.
- Using things like fetch on the browser we don't need to worry about different threads working on something else, we just call it using this facade function and it's a facade because underneath the hood it calls on to the web API and says Hey web API I have something for you, Can you take care of it for us.
- **What about in node what happens in node?**
- Well as you saw **we have worker threads in node**. When we do something like reading from the file system in the background node does these things for you, reading from a database node does it for you.
- So**, we don't really need to worry about having to create our own threads** which is nice. It makes our programs simple.

- **Concurrency vs Parallelism:**
- 

- **First Program with Single Core CPU:**
- If we run this first program where we only have a single core CPU on our computer what the computer would be doing is it'll be working on one thread.
- As soon as it's done with that thread or has a bit of time a bit of pause it can switch to the next thread and work on something there and then go back to thread one because we have more work to do.
- And then back to thread 2 it can switch back and forth between these two threads but only allowing one thread to run out of time because we have a single core CPI that is one CPU you can do work.
- **So, the best way to think about it is just like this man over here eating in a single threaded model.**
- 

- **You just have one mouth to eat these things. One mouth means a CPU**, but you have two hands right. So, you can grab something from the plate.
- Let's say an apple and bring that to your mouth and once you're done with that you can grab with another hand let's say a banana and grab that and bring it towards your mouth and that is what concurrency is.
- **Concurrency is something that we can achieve in JavaScript when we work on our single threaded JavaScript but in the background**, we use node or Web browsers to allow us to do things in the background on different threads.
- **Parallelism can only happen when we have things like multicore CPU** that is many CPU that is like using many more mouth than just one.
- If we run this program the multicore CPU would allow us to execute threads at the same time **because each one would be on a different CPU so we'd be able to run these threads in parallel side by side at the exact same time.**

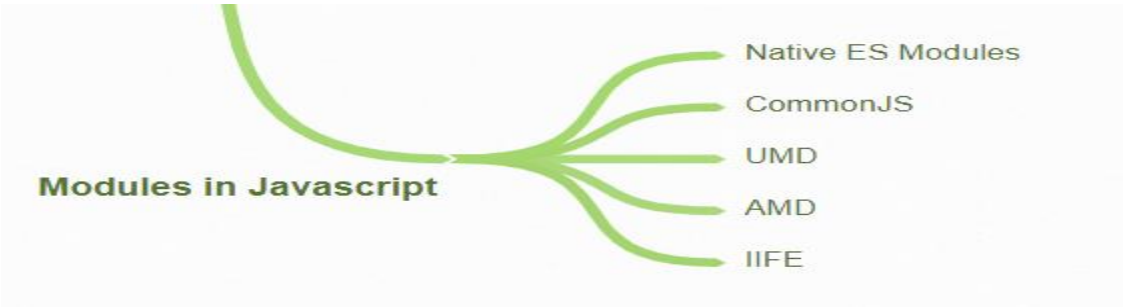
- **How cool is that now when it comes to parallelism and JavaScript?**
- **You can't really do that can you**. It's not really built into the language **but there is a way to achieve something like this.**
- Let's say something like node you see node was built for simplicity but if you have let's say four CPU cores on your computer or server then **you can run that node instance on all those four CPU cores.**
- **So how do we create more of these processes that can run on multiple cores?**
- Well in node we can do something like spawn **and there's three other ways** of doing this.
- 

- Different ways of doing this but we'll just use spawn as a good example and we'll just require the child process module that comes with node which will spawn a new process and then we can just run this process by saying let's just say get stuff.
- And if I run **this it will generate a new process** that I can run.

- Now this is very advanced. You can read up on your own. The point of this video is just to show you the idea of concurrency and parallelism.
- **Although JavaScript might be single thread, it might have just one call stack. The beauty is in its restriction and simplicity because this is hard to manage.**
- **Parallelism is extremely difficult especially when you have these two threads touching the same data.**
- The fact that we have a single call stack in our program allows us to write really nice clean code and have asynchronous events, which I think makes things really fun to code and avoid some really strange bugs that might occur in a model like this.

Section 10: Modules in JavaScript

140. Section Overview:



- **Modules are just like in that different pieces** of code are grouped together so that things are organized and as our application gets larger and larger**, we can combine different pieces together to make these large application**s.
- It's very similar to the **idea of separation of concern**.
- We don't necessarily have just one massive JavaScript file Ideally our code is broken up into different files with different functionality.
- So, **we need a way in JavaScript to import dependencies** that is code that we want to use, and export functionality often called an interface to allow others to use our code.

142. Module Pattern ES5 :

- In an ideal world we have **another scope which is the module scope** that is a little bit higher up than the function scope so that we can combine multiple functions together under this module scope and to share these variables between different functions as well but **still not pollute our global namespace**.

- **Earlier we were using closure and encapsulation** to create our own module scope.
- **Ex: avoiding interaction to global scope:**

```
IIFE
(function() {

})()
```

- Well **it's not a function declaration because** the **first thing that the compiler sees won't be function and** then we're immediately invoking it calling the function like this.
- So, **we're using the idea that we have function scope to simply just write all our code in here that will be private to this function**, but we also can access global variables.
- **Earlier way of separations of concerns:**

```
//IIFE
(function() {
  var harry = 'potter'
  var voldemort = 'He who must not be named'

  function fight(char1, char2) {
    var attack1 = Math.floor(Math.random() * char1.length);
    var attack2 = Math.floor(Math.random() * char2.length);
    return attack1 > attack2 ? `${char1} wins` : `${char2} wir
  }
  console.log(fight(harry, voldemort))
})()
```

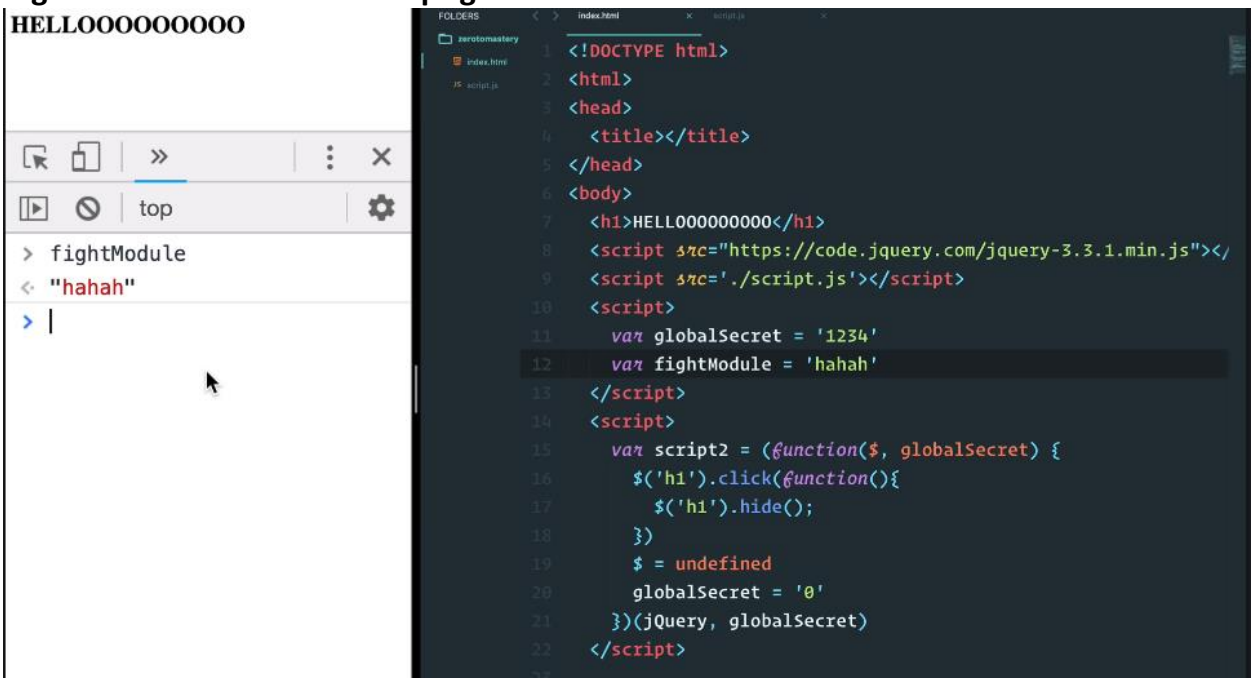
- **But what if we wanted other scripts to use this fight function:**

```
//IIFE
//Module Pattern
var fightModule = (function() {
  var harry = 'potter'
  var voldemort = 'He who must not be named'

  function fight(char1, char2) {
    var attack1 = Math.floor(Math.random() * char1.length);
    var attack2 = Math.floor(Math.random() * char2.length);
    return attack1 > attack2 ? `${char1} wins` : `${char2} wir
  }
  return {
    fight: fight
  }
})()
```

- This **pattern of returning what we need is called the revealing module pattern**. **We just reveal whatever we need** so we can have private functions if we want.

143. Module Pattern Pros and Cons:

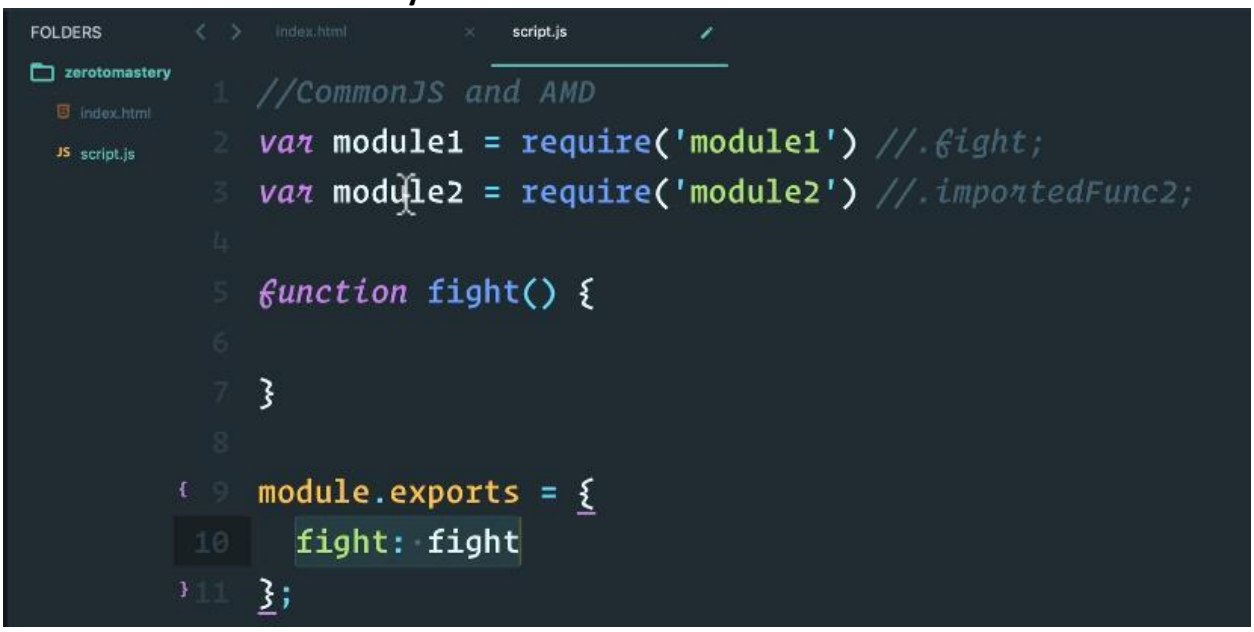
- **Benefit of using IIFE in above program is instead of writing in global scope is that we're only revealing one variable polluting the global namespace just once.**
- And this is **great for maintainability**. **A module is self-contained, a well-designed, could just contain specific functionality and lower the dependencies on other parts of the code base.**
- **Module is decoupled from the other pieces of code.**
- **There are two main problems** with this approach that we see:
- **One is that we technically are still polluting the global namespace** 'fightModule' is still technically a variable on the global namespace.

- **So, if we had a script that has a 'fightModule' variable then our old 'fightModule' from other script gets overridden**.
- 

- **So, we've minimized the number of global variables, but we can still have name clashes**.
- And then the other issue and a lot of people who have worked with JavaScript in the past know this is that we don't necessarily know all the dependencies, so we have to make sure that the order of the script tags is correct.

144. Commons.Js, AMD, UMD:

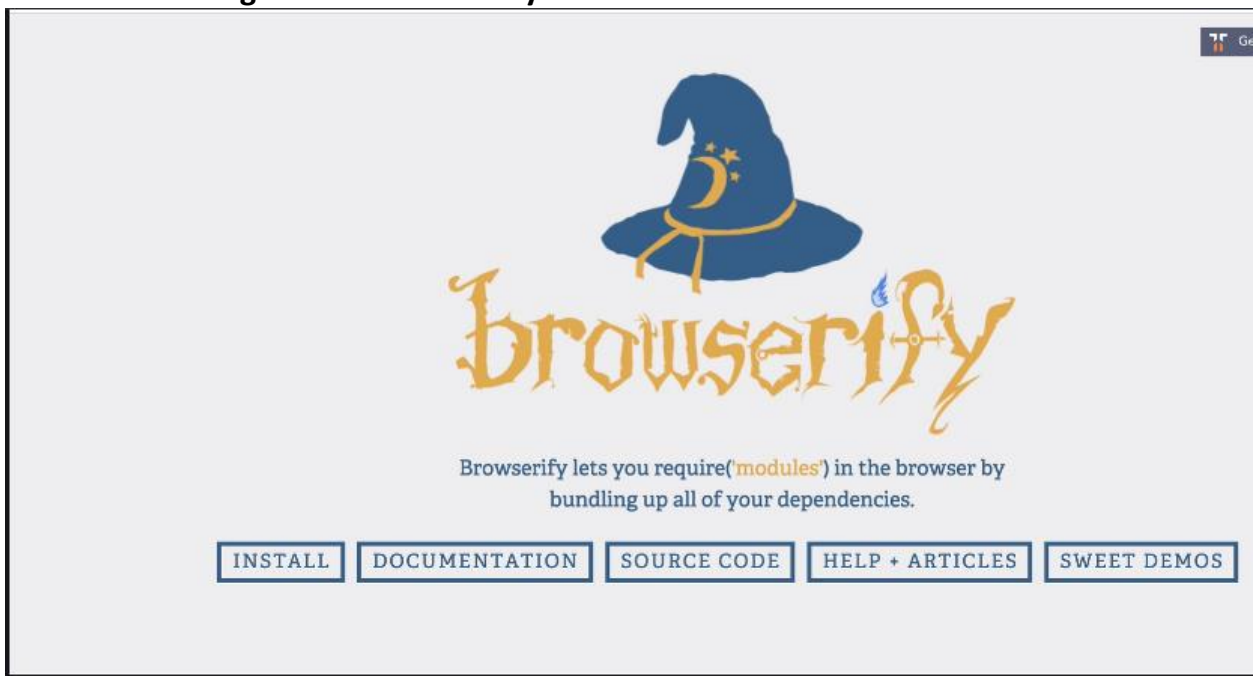
- Because of the problems mentioned in the previous video. Well luckily for us **after the module pattern came to great solutions instead of using an IIFE and the module pattern something called common.js and AMD(AsynchronousModuleDefinition) came out.**
- And they solved the problem of designing a module in a way that **we won't have the interference of global scope where we can overwrite certain variables.**

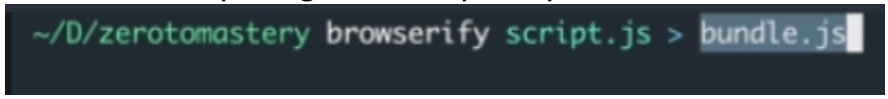
- **Let's have a look at how they work:**
- 

- **Common J.S. looked something like this we can import different modules** or different files using require. We can even export specific functions.

- **A matter of fact common.js was created mainly** for the server with Node.js in mind to use for servers and desktop applications and it's one of the main reasons that Node.js became so popular.
- **This common.js import export module system** that came before we even had it in the browser made code very easy to share for Node.js programmers.
- **You might have heard of NPM**. It is just a way for people to share their modules and Node.js popularity, one of the main reasons is because of NPM, because people using common.js were able to share their code to other programmers so that in Node.js sharing code and using third party code became really easy.

- **Now with common.js modules are meant to be loaded synchronously** that means remember JavaScript has one call stack so if a module takes a long time to load, we're just waiting there until that gets loaded.
- **And then the next one gets loaded** and then we get to run our script. And that's not ideal for browsers where we have users clicking on buttons, entering data in two forms.
- **A lot of interaction synchronous code on the browser can get dangerous** and that is why common.js was mainly used on the server.
- **But people want to use all these packages** that these people are writing in NPM. So how can we use some of these useful packages on the browser. Even though this imports syntax is synchronous and can slow down our web pages.

- **we had two things. One is browserify:**
- 

- **It lets you require modules in the browser by bundling up all your dependencies**.
- What does that mean.
- It means that by using browserify in my command line we can run something like:
- 

- **What it will do is it will read our script.js and understand this require syntax. Understand the module that exports syntax and output it all into a bundle.**
- **Now this bundle.js would be a bundle of all the scripts in my project.**

- So, this way we're able to use common.js even though with synchronous because when we deploy this to the browser it's one giant JavaScript file with everything already in it.
- **And this is what is called a module bundler and things like webpack can also do this.**
- **So, common.js extremely popular because things like Webpack and browserify simply traverse the dependency tree of our code and bundle them up into a single file.**
- **But finally, we have the benefit of no global namespace pollution and order doesn't matter anymore**.

- **Second Way: AMD(AsynchronousModuleDefinition):**
- AMD was designed specifically for the browsers that means **it loads scripts or modules asynchronously.**
- **Pending……**

145. ES6 Modules:

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export
-

```
const harry = 'potter'
const voldemort = 'He who must not be named'

export function jump() {
  console.log('jump')
}

export default function fight(char1, char2) {
  const attack1 = Math.floor(Math.random() * char1.length);
  const attack2 = Math.floor(Math.random() * char2.length);
  return attack1 > attack2 ? `${char1} wins` : `${char2} wins`
}
```

- ```
  index.html
  script.js
  <!DOCTYPE html>
  <html>
  <head>
    <title></title>
  </head>
  <body>
    <h1>HELLOOOOOOOOO</h1>
    <script src="https://code.jquery.com/jquery-3.3.1.min.js"></
    <script type="module" src='./script.js'></script>
    <script>
    </script>
    <script>
    </script>
    <script type="module">
      import fight, { jump } from './script.js';
      console.log(fight('ron', 'hedwig'))
      console.log(jump())
    </script>


  </body>
  </html>
  ```
- **Result:**
- ```
  hedwig wins
  jump
  undefined
  >
  ```

Section 11: Error Handling

148. Errors in JavaScript:

- In JavaScript we have a **native error constructor function**.
- **And I can create new instances** of these errors by doing new error by passing a message. If I run this, I get an error with message passed now.

```
> Error
ƒ Error() { [native code] }
> new Error('oopsie')
Error: oopsie
    at <anonymous>:1:1
>
```

- **In JavaScript we have the throw keyword**. Throw statement is used to generate are exceptions are errors. So, when we do something like throw new error, I now get a proper error.

```
Uncaught Error
    at <anonymous>:1:7
```

- **What has happened underneath the hood** is that the error gets thrown and says all right stop the program handled the error somehow.
- So, during runtime when a throw statement is encountered by the program the **execution of the current function will stop, and control will be passed to the next part of the call stack.**
- **We can throw anything in JavaScript like string, Boolean, error constructor** function and real errors also:

```
> throw 'string'
Uncaught string
> throw true
Uncaught true
> throw Error
Uncaught ƒ Error() { [native code] }
> throw new Error()
Uncaught Error
    at <anonymous>:1:7
```

- Now when I do '**myError'** you see that I have the 'Error' here, but I have three properties with this:

```
> const myError = new Error('oopsie')
<· undefined
> myError.name
<· "Error"
```

-

```
> myError.message
<· "oopsie"
```

-

```
> const myError = new Error('oopsie')
<· undefined
> myError.stack
<· "Error: oopsie↵ at <anonymous>:1:17"
```

-

- **Stack trace is just a string**, but it shows me where the error happened.
- In this case it happened inside of an anonymous function which is the main global execution context.

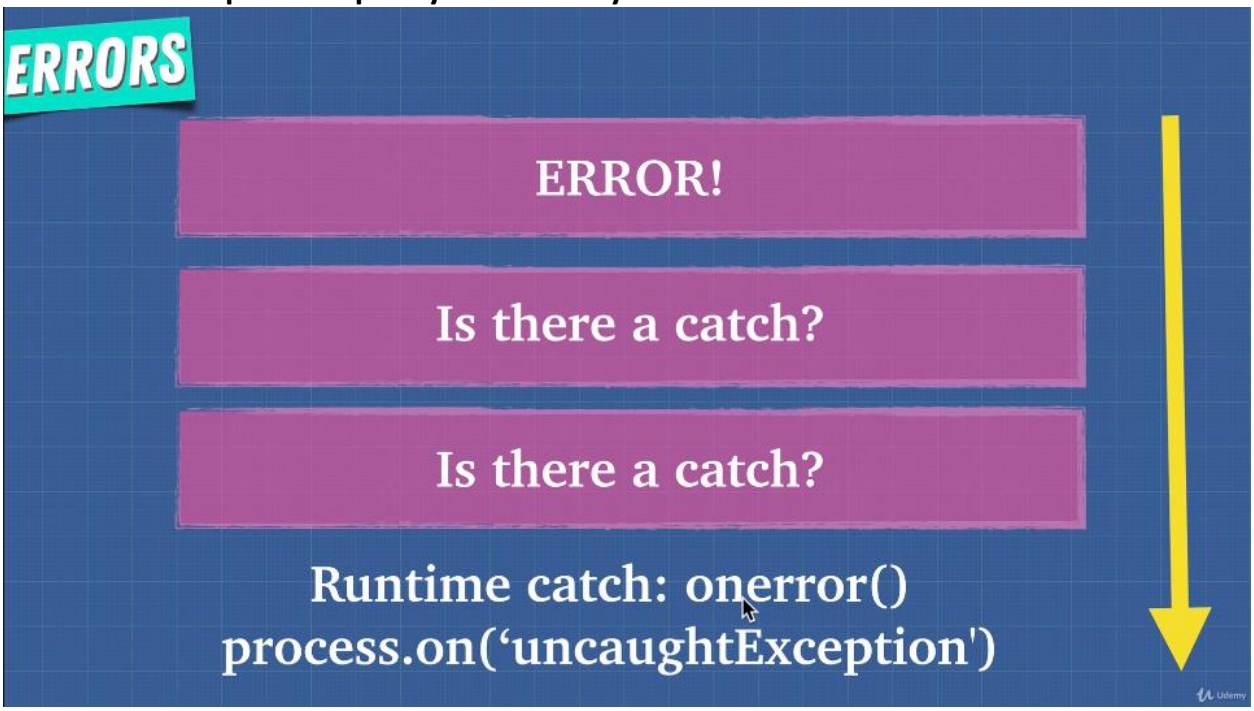
```
> function a() {
   const b = new Error('what??')
   return b
  }
<· undefined
> a()
<· Error: what??
       at a (<anonymous>:2:12)
       at <anonymous>:1:1
> a().stack
<· "Error: what??
       at a (<anonymous>:2:12)
       at <anonymous>:1:1"
```

-

- **The JavaScript has many built in constructors for errors:**

```
> new SyntaxError
< SyntaxError
      at <anonymous>:1:1
> (,
✖ Uncaught SyntaxError: Unexpected token ,
> new ReferenceError
< ReferenceError
      at <anonymous>:1:1
> something
✖ ▶ Uncaught ReferenceError: something is not defined
      at <anonymous>:1:1
```

- **Well in JavaScript this is pretty much the system for errors:**



- This is our call stack and as soon as an error happens on the call stack, we go to the execution context and say hey is there catch for us. Is there something handling this error.
- No. Okay then I'm going to keep going. Is there a catch. Is this handled anywhere in this part of the execution context. No.

- **And if all the way through the call stack there's nothing handling it, we're going to get this 'onerror ()' function that runs inside of the browser that gives us that 'red text' that we see.**
- **In node.js instead of the 'onerror ()' we have the 'process.on()**'. so, the runtime handles the errors if nothing in our program catches this.

- **Summary:**

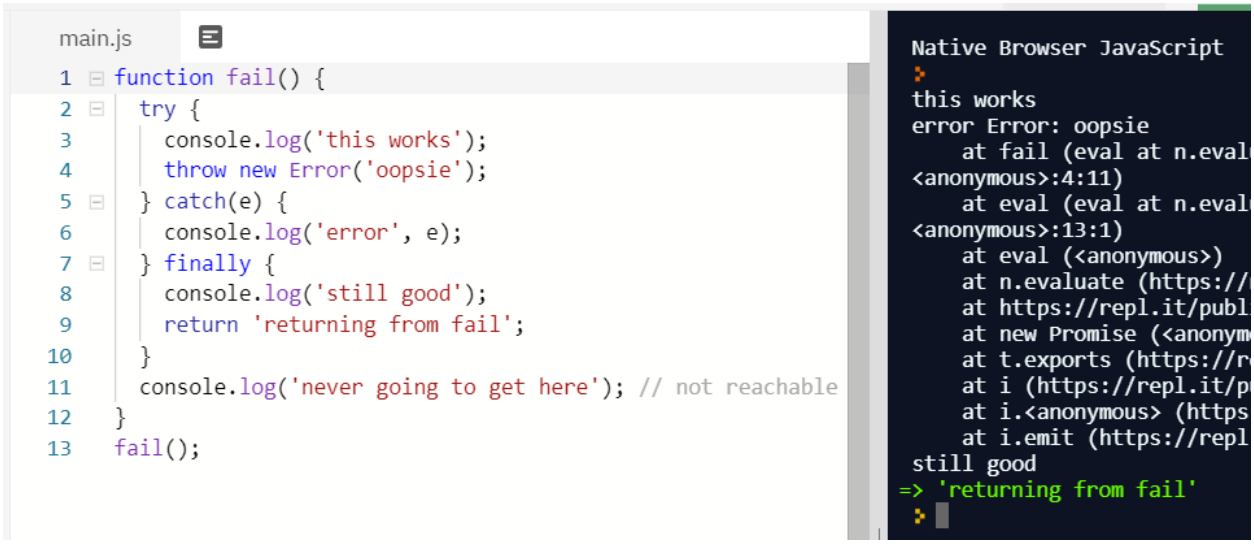
main.js saved

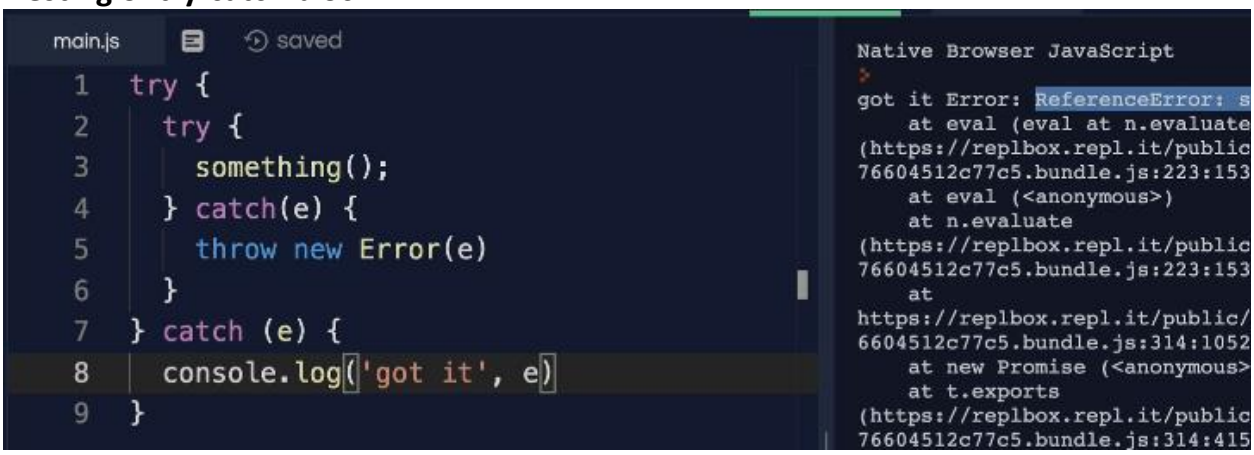
```
1   throw 'Error2';    // String type
2   throw 42;          // Number type
3   throw true;        // Boolean type
4   throw Error
5   throw new Error // will create an instance of an Error in JavaScript and stop
    the execution of your script.
6
7   function a() {
8     const b =  new Error('what?')
9     return b
10  }
11
12  a().stack
13
14  let error = new Error(message);
15  let error2 = new SyntaxError(message);
16  let error3 = new ReferenceError(message);
```

149. Try Catch:

```
> function fail() {
    try {
      consol.log('this works')
    } catch (error) {
      console.log('we have made an oopsie', error)
    }
  }

  fail()

  we have made an oopsie ReferenceError: consol is not defined
      at fail (<anonymous>:3:5)
      at <anonymous>:9:1

<· undefined
```

- 

- 

- **Nesting of try-catch block:**
- 

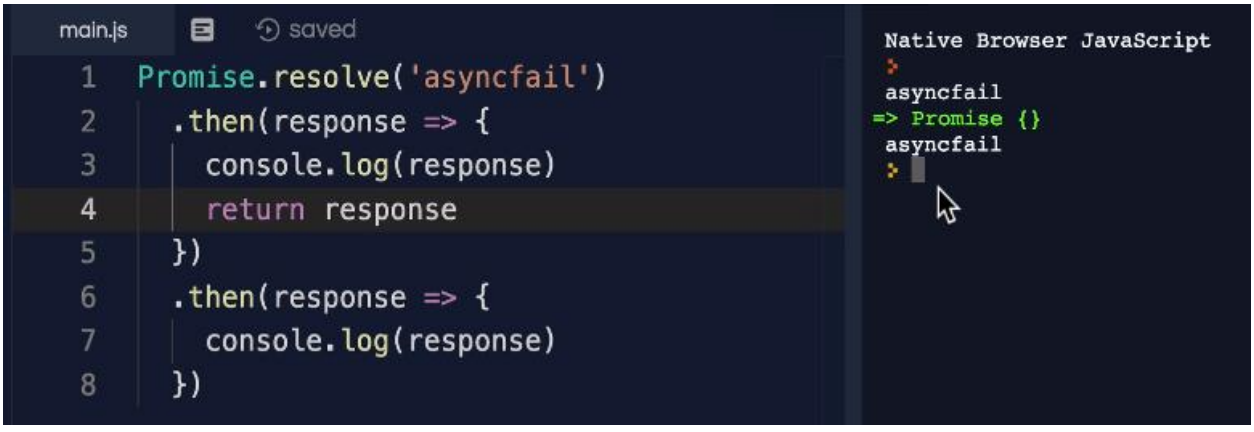
- This type of try catch block can be used **to catch any type of synchronous errors**.

- **For example, if I have a set timeout which is asynchronous** and had a function that doesn't do anything let's say a function in here that uses a fake variable that doesn't exist. We haven't declared it anywhere.
- 

- And when we'll run this, in one second, do you see that everything is working. I get no errors. **This catch block doesn't catch it because this code is asynchronous**.
- And as we know this exits our current call stack goes all the way to the web API, the callback queues the event loop and pushed onto the call stack after whole piece of synchronous code is already done executing so that's a problem.

150. Async Error Handling:

- Will see how to handle errors in promises and with async await.

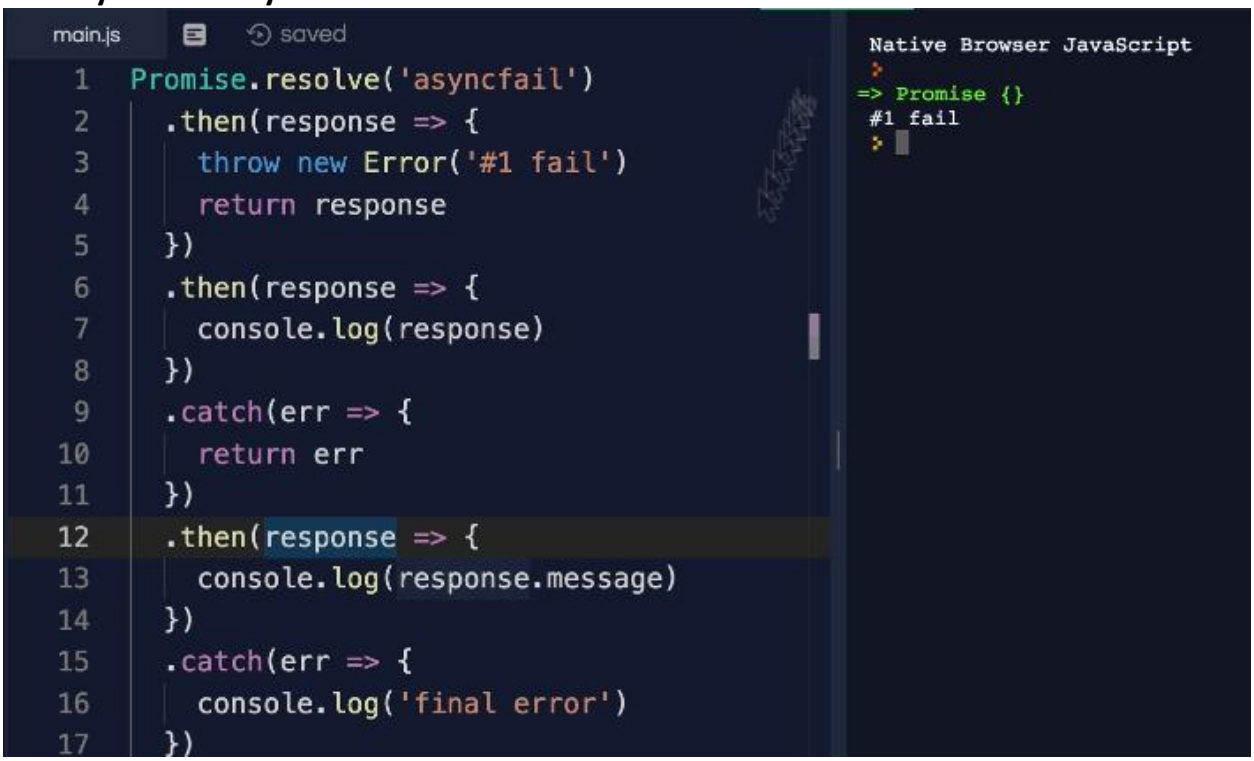
- **Code base:**
- 

- **Throwing error in promise**: Here it failed to catch error because we did not use catch
- 

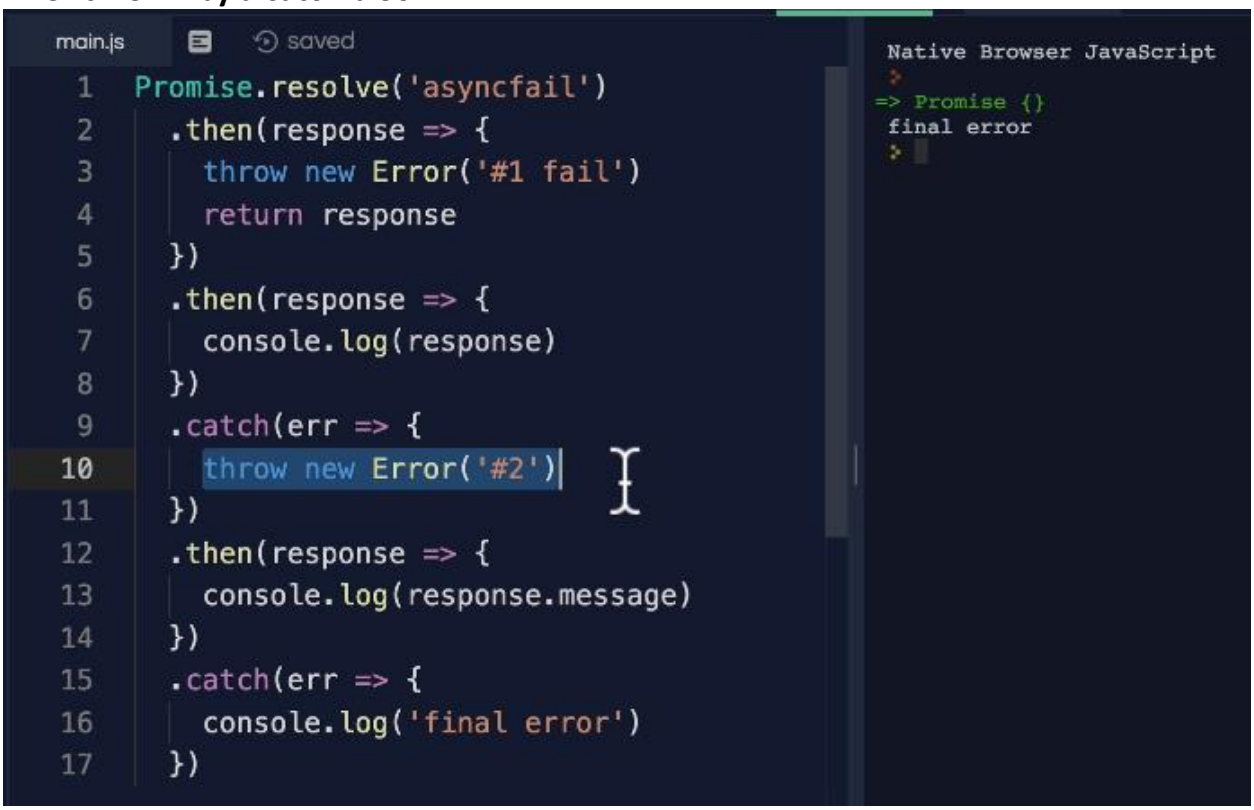
- **Throwing error and catching:**
- 

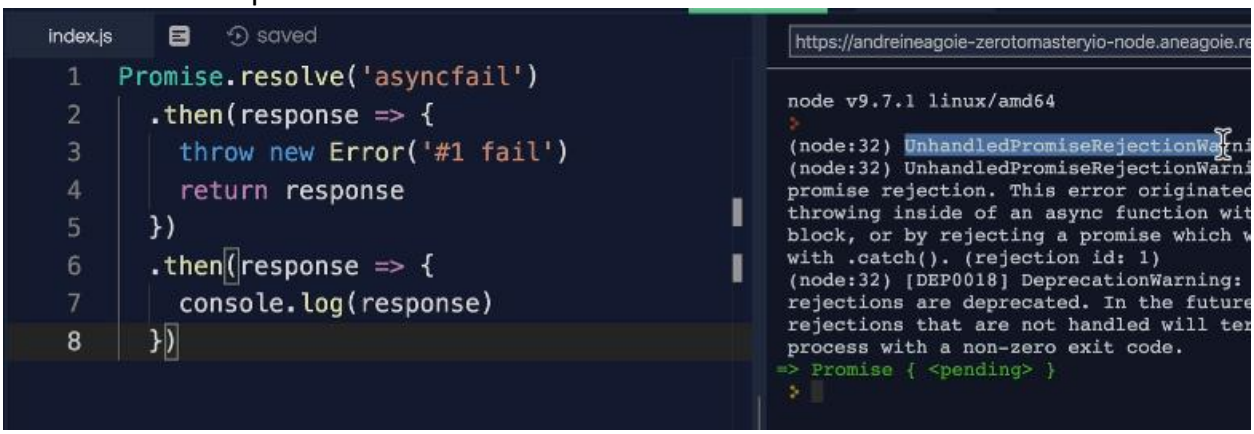
- **‘. then’ after ‘ .catch’:**
- 

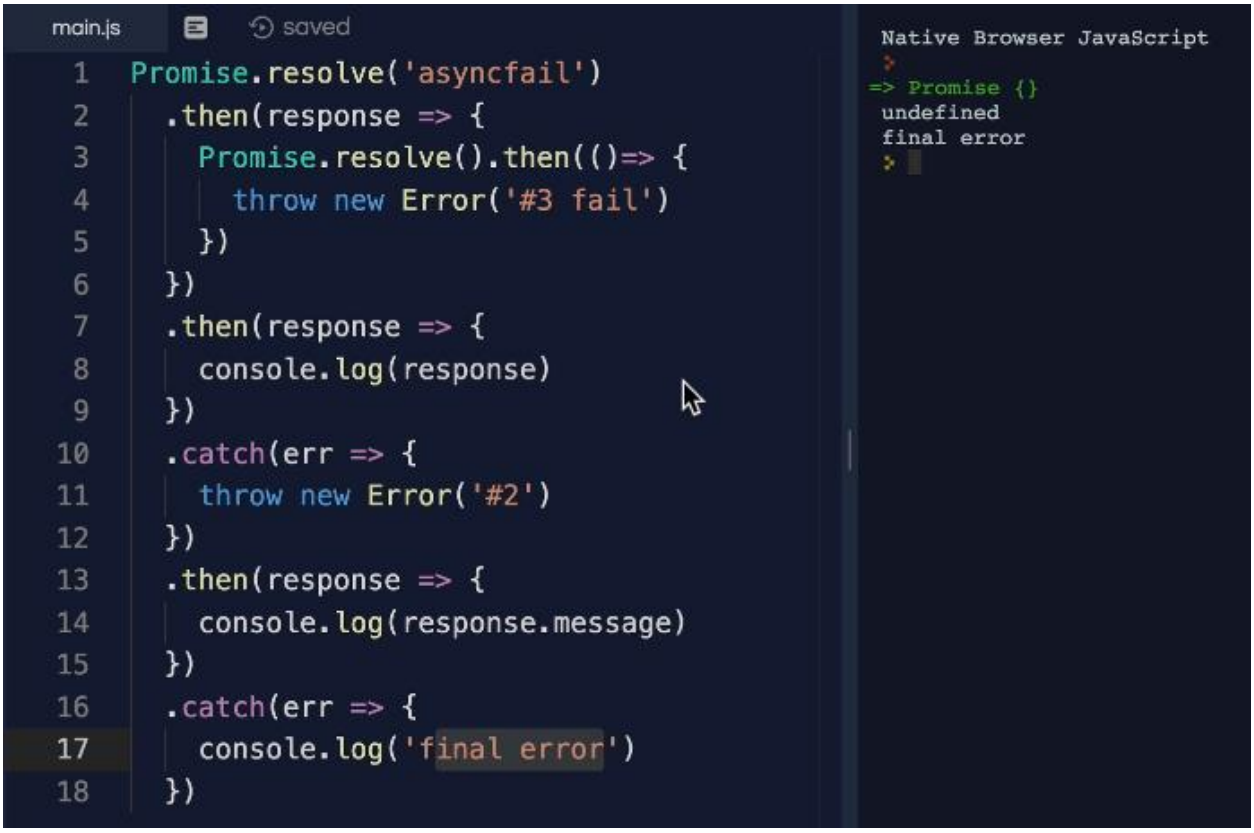
- **‘. catch’ at the end is never going to run because error thrown by first ‘. then’ is already handled by earlier catch.**



- **Error thrown by a catch block:**



- **In node.js we caught error thrown by our code without having catch because it is a different runtime.** But in JavaScript it is a silent failure as we already have seen in first example.
- 

- **Nested Promise:**
- 

- **What is undefined in above example:** If we return 5 will get 5. But why this final error is coming. this shows you how tricky it can be to handle asynchronous promises.

- main.js — saved

```
1  Promise.resolve('asyncfail')
2    .then(response => {
3      Promise.resolve().then(()=> {
4        throw new Error('#3 fail')
5      })
6      return 5
7    })
8    .then(response => {
9      console.log(response)
10   })
11   .catch(err => {
12     throw new Error('#2')
13   })
14   .then(response => {
15     console.log(response.message)
16   })
17   .catch(err => {
18     console.log('final error')
19   })
```

```
Native Browser JavaScript
=> Promise {}
5
final error
```

- So, we should make make sure to really understand your flow and that on each level you're handling these promises with a dot catch.
- **So, let's fix this let's remove this from here:**
- main.js — saved

```
1  Promise.resolve('asyncfail')
2    .then(response => {
3      Promise.resolve().then(()=> {
4        throw new Error('#3 fail')
5      }).catch(console.log)
6      return 5
7    })
8    .then(response => {
9      console.log(response)
10   })
11   .catch(err => {
12     console.log('final error', err)
13   })
```

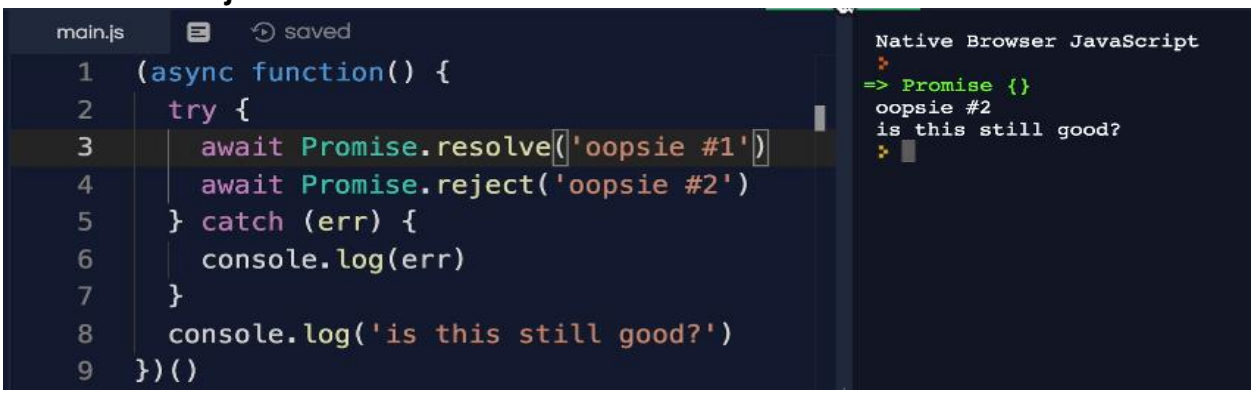
```
Native Browser JavaScript
=> Promise {}
5
Error: #3 fail
    at Promise.resolve.then (eval at n.evaluate
(https://replbox.repl.it/public/replbox_javascript.838ca97
76604512c77c5.bundle.js:223:153193), <anonymous>:4:11)
```

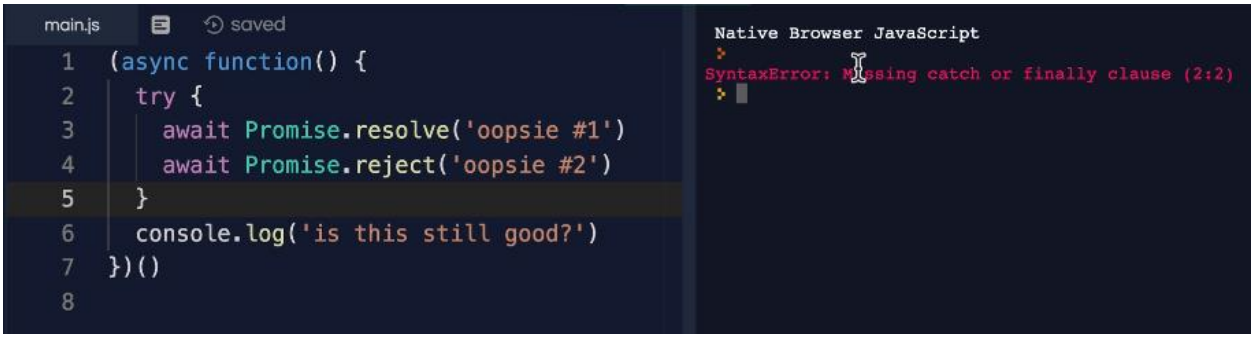
- Everything is now good. We're handling our promises each individually

151. Async Error Handling 2:

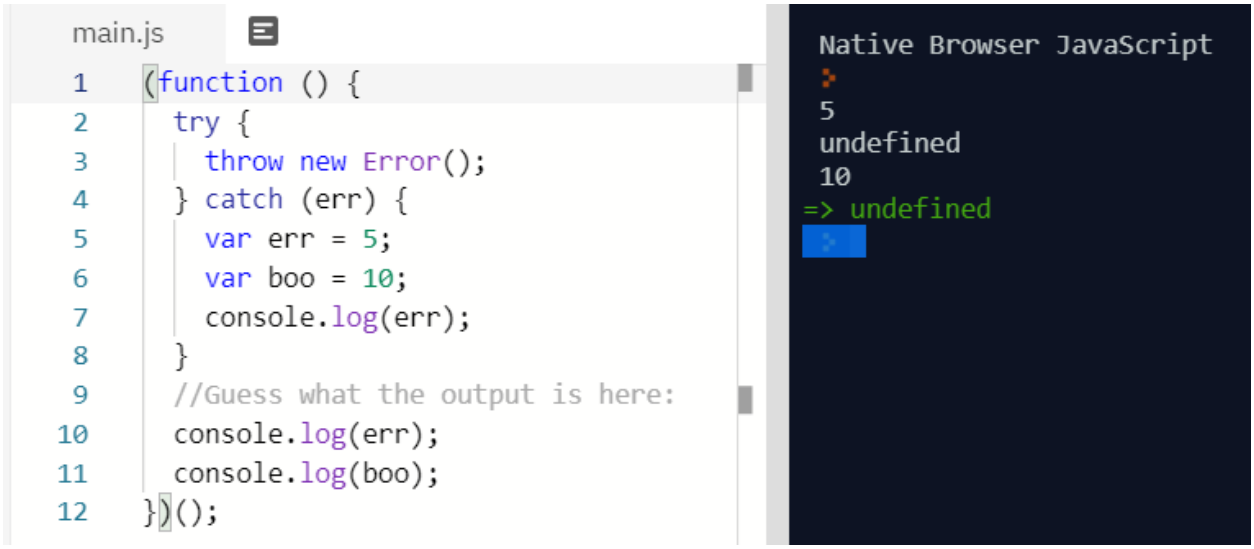
- As we discussed try catch block is only for synchronous code and we used the dot catch for our asynchronous code using promises.
- **Well because async await although asynchronous makes our code look synchronous, we can actually use try catch blocks with them.**
- 

- **Multiple rejects:**
- 

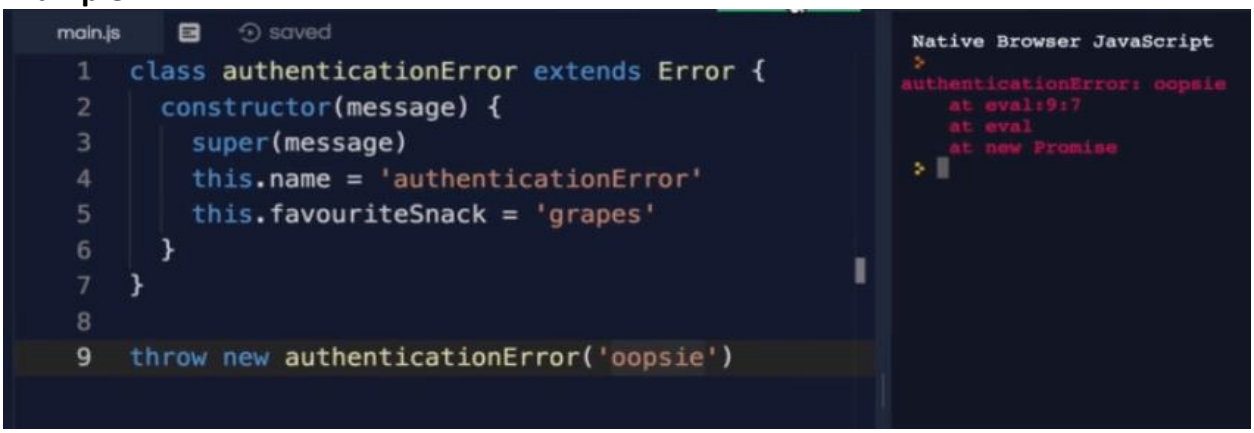
- **Resolve with reject:**
- 

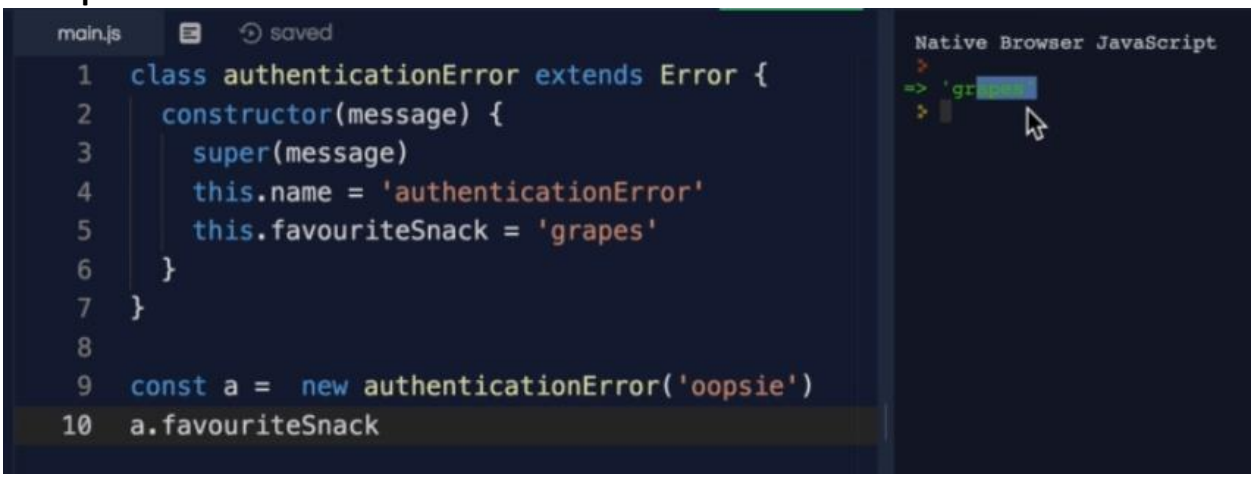
- **The only gotcha with async await is that it's very easy to just forget to wrap it in a try catch** block so that these errors just don't get caught if we use this in our code
- 

152. Exercise: Error Handling

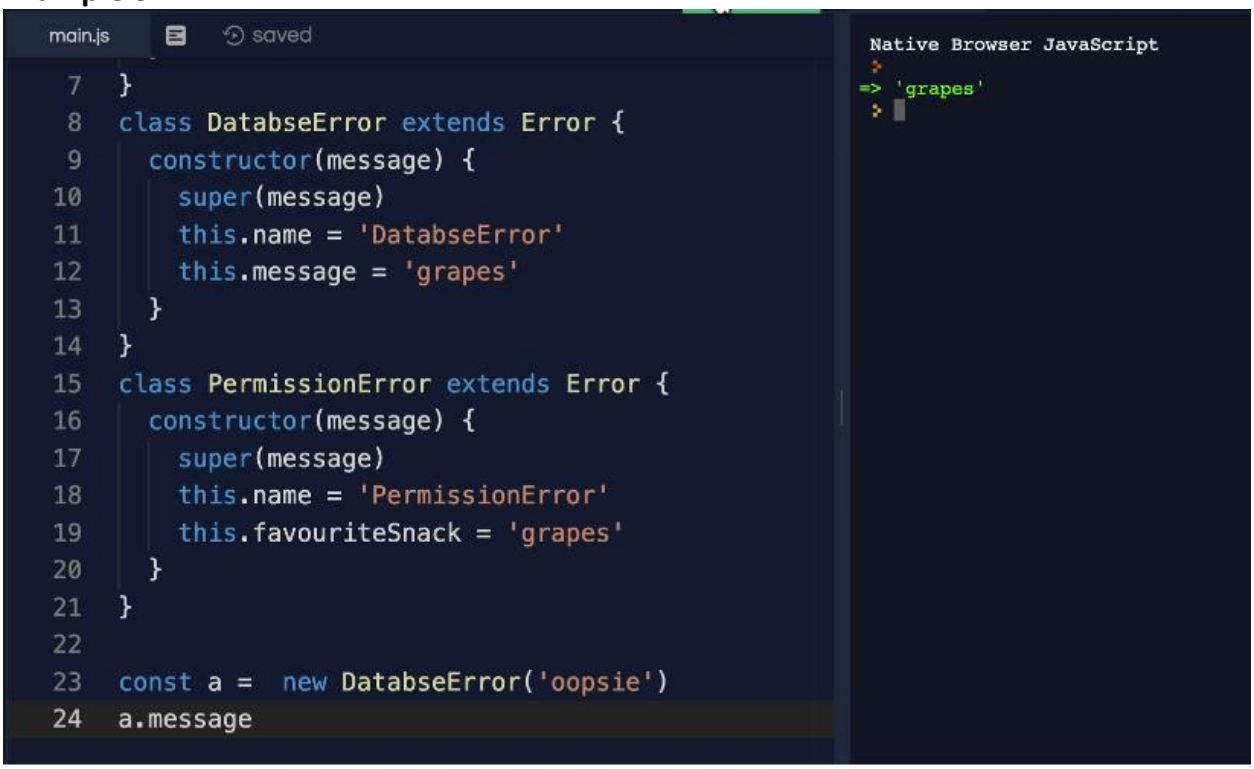
- 

153. Extending Errors:

- **Example 1:**
- 

- **Example 2:**
- 

- **Now the tricky part here is that when you throw an error to a user you don't want to reveal too much information about your system because hackers and bad actors might use that information to compromise your system.**
- So, you don't necessarily want to just throw any error to the user so that they can see for example in a node server you never want to return a response with a full error with a stack trace of the error where it happened and all that information so you can actually use this to customize the errors that you may have.

- **Example 3:**
- 

- **Example 4:**

main.js saved

```
9   constructor(message) {
10    super(message)
11    this.name = 'DatabseError'
12    this.message = 'grapes'
13  }
14  }
15  class PermissionError extends Error {
16    constructor(message) {
17      super(message)
18      this.name = 'PermissionError'
19      this.favouriteSnack = 'grapes'
20    }
21  }
22
23  const a =  new DatabseError('oopsie')
24  a instanceof DatabseError
```

```
Native Browser JavaScript
=> true
```

- **Example 5:**

main.js

```
1   class authenticationError extends Error {
2     constructor(message) {
3       super(message)
4       this.name = 'ValidationError'
5       this.message = message
6     }
7   }
8   class PermissionError extends Error {
9     constructor(message) {
10      super(message)
11      this.name = 'PermissionError'
12      this.message = message
13      this.favouriteSnack = 'grapes'
14    }
15  }
16  class DatabaseError extends Error {
17    constructor(message) {
18      super(message)
19      this.name = 'DatabaseError'
20      this.message = message
21    }
22  }
23
24  throw new PermissionError('A permission error')
```

```
Native Browser JavaScript
PermissionError: A permission error
    at eval:24:7
    at eval
    at new Promise
```

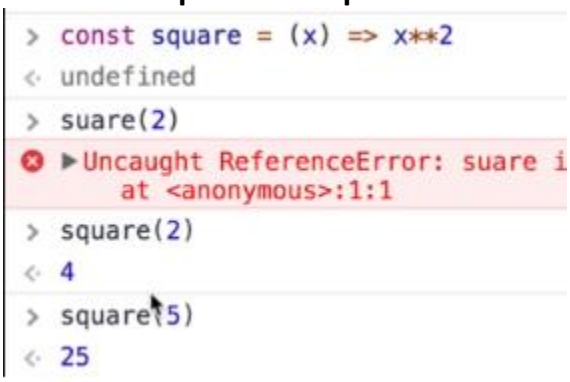
-

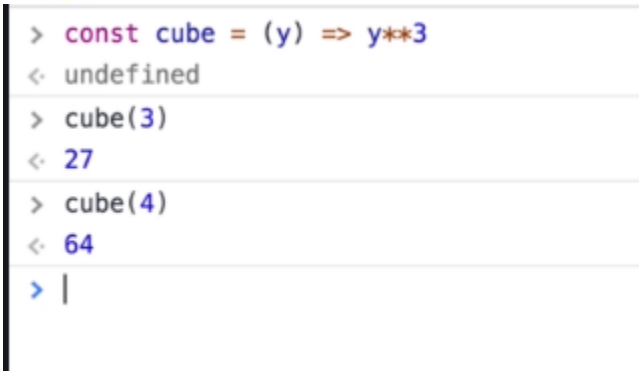
Section 15: Appendix II: Intermediate JavaScript:

208. ES7:

- ES7 only has two additions to the language.
- **First one is 'includes' method** that was added on strings and arrays.

- **Second is exponential operator:**





209. ES8:

- 1**. String Padding**: this is useful just for aligning characters of strings.
- 

- **2. Trailing commas in functions:**
- ```
  const fun = (a,b,c,d,) => {
    console.log(a);
  }

  fun(1,2,3,4,)
  1
  undefined
  ```

- **3. Object.values and Object.entries:** we already have Object.keys.
- Object.keys and Object.values
- ```
  let obj = {
    username0: 'Santa',
    username1: 'Rudolf',
    username2: 'Mr.Grinch'
  }

  Object.keys(obj).forEach((key, index) => {
    console.log(key, obj[key]);
  })
  username0 Santa                                   VM1137:8
  username1 Rudolf                                  VM1137:8
  username2 Mr.Grinch                               VM1137:8
  undefined
  Object.values(obj).forEach(value => {
    console.log(value);
  })
  Santa                                             VM1191:2
  Rudolf                                            VM1191:2
  Mr.Grinch                                         VM1191:2
  undefined
  ```

- **Object.entries:**

```
> Object.entries(obj).forEach(value => {
    console.log(value);
})
  ▶ (2) ["username0", "Santa"]                    VM1218:2
  ▶ (2) ["username1", "Rudolf"]                   VM1218:2
  ▶ (2) ["username2", "Mr.Grinch"]                VM1218:2
< undefined
> Object.entries(obj).map(value=> {
    return value[1] + value[0].replace('username', '');
})
< ▶ (3) ["Santa0", "Rudolf1", "Mr.Grinch2"]
```

211. ES10 (ES2019):

- 1. **Flat method**: we can use on arrays. It will return a new array.

```
> const array = [1,2,3,4,5]
  array.flat()
< ▶ (5) [1, 2, 3, 4, 5]
> const array2 = [1,[2,3],[4,5]]
  array2.flat()
< ▶ (5) [1, 2, 3, 4, 5]
> const array3 = [1,2,[3,4,[5]]]
  array3.flat()
< ▶ (5) [1, 2, 3, 4, Array(1)]
> const array3 = [1,2,[3,4,[5]]]
  array3.flat(2)
< ▶ (5) [1, 2, 3, 4, 5]
```

- 

- **clean up our data with flat:**

```
> const entries = ['bob',
  'sally',,,,,,'cindy']
  entries.flat()
<· ▶ (3) ["bob", "sally", "cindy"]
```

- **3. flatMap method:** The **flatMap()** method first maps each element using a mapping function, then flattens the result into a new array. It is identical to a map() followed by a flat() of depth 1

```
> const jurassicParkChaos =
  jurassicPark.flatMap(creatur
  e => creature + '🦖')
<· undefined
> jurassicParkChaos
<·   (6) ["🦖,🦖🦖", "🐉🦖", "🦕
  ▶ 🦖", "🐊,🦖🦖", "🦖,🦕🦖",
    "🦀,🦑🦖"]
```

- **4. trimStart() and trimEnd():**

```
> const userEmail = '          eddytheeagle@gmail.com'
  const userEmail2 = 'jonnydangerous@gmail          '
  console.log(userEmail.trimStart())
  console.log(userEmail2.trimEnd())
  eddytheeagle@gmail.com
  jonnydangerous@gmail
<· undefined
> userEmail
<· "          eddytheeagle@gmail.com"
```

- **5. fromEntries:** It transforms a list of key value pairs into an object.

```
> userProfiles = [['commanderTom', 23],
  ['derekZlander', 40], ['hansel', 18]]

  Object.fromEntries(userProfiles)
<· ▼ {commanderTom: 23, derekZlander: 40, ha
    nsel: 18} i
       commanderTom: 23
       derekZlander: 40
       hansel: 18
     ▶ __proto__: Object
```

- **fromEntries vs Object.entries:**

```
> userProfiles = [['commanderTom', 23],
  ['derekZlander', 40], ['hansel', 18]]

  const obj =
  Object.fromEntries(userProfiles)
  Object.entries(obj)
<· ▼(3) [Array(2), Array(2), Array(2)]
    ▶0: (2) ["commanderTom", 23]
    ▶1: (2) ["derekZlander", 40]
    ▶2: (2) ["hansel", 18]
     length: 3
    ▶__proto__: Array(0)
```

- **6. Catch will work without parenthesis** but if you want error info, we need to use parenthesis.

```
> try {
    bob + 'hi'
  } catch {
    console.log('you messed up')
  }
  you messed up                    VM2403:4
```