the Master Course

{CODENATION}

Introduction to asynchronous Javascript

{CODENATION}

{CODENATION}

Learning Objectives

To understand what **synchronous and asynchronous** mean.

To be able to work with higher-order functions, and to understand what **callback functions** are.

To understand what **Promises** are in Javascript.

To understand the keywords **async** and **await**.

First. What does synchronous mean?

{CODENATION}

One thing at a time.
In order.

{CODENATION}

Synchronous

In javascript, synchronous refers to our code executing one thing at a time. So our program waits until the current function has finished before moving on to the next.

Asynchronous

Asynchronous refers to our code not having to wait until a function has finished, before moving on to the next.

Imagine we are all stood in line at a buffet.

Ben decides he wants to load his plate up with as many steaks as he possibly can.

While he does this we have to wait until he's finished.

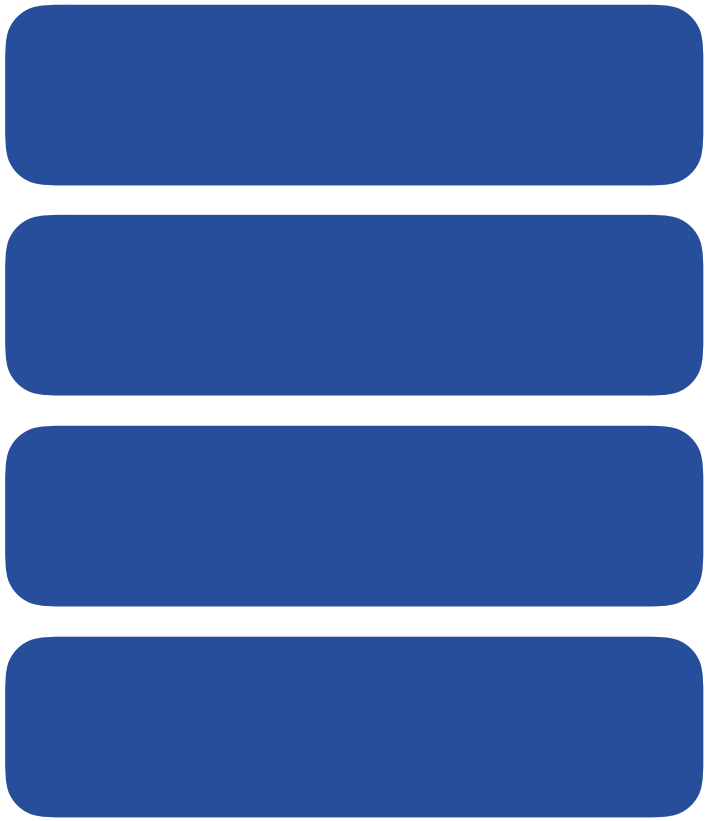
Back to the call stack.

Javascript is a single threaded language. Which means it only has one call stack.

{CODENATION}

Call stack

Keeps track of our code as it runs.



Imagine you have a particular function which is taking ages. With only synchronous javascript, our program would have to wait.

Asynchronous javascript allows us to carry on down the call stack, without getting blocked by a slow function.

Examples galore.

```
let a = 1
let b = 2
let c = 3
let d = 4


console.log(a)
console.log(b)
console.log(c)
console.log(d)
```

In what order are things printed to the console?

Let's use a function called setTimeout( ) to simulate a function taking ages.

{CODENATION}

```
let a = 1
let b = 2
let c = 3
let d = 4

console.log(a)

setTimeout(() => {
  console.log(b)
}, 2000)
console.log(c)
console.log(d)
```

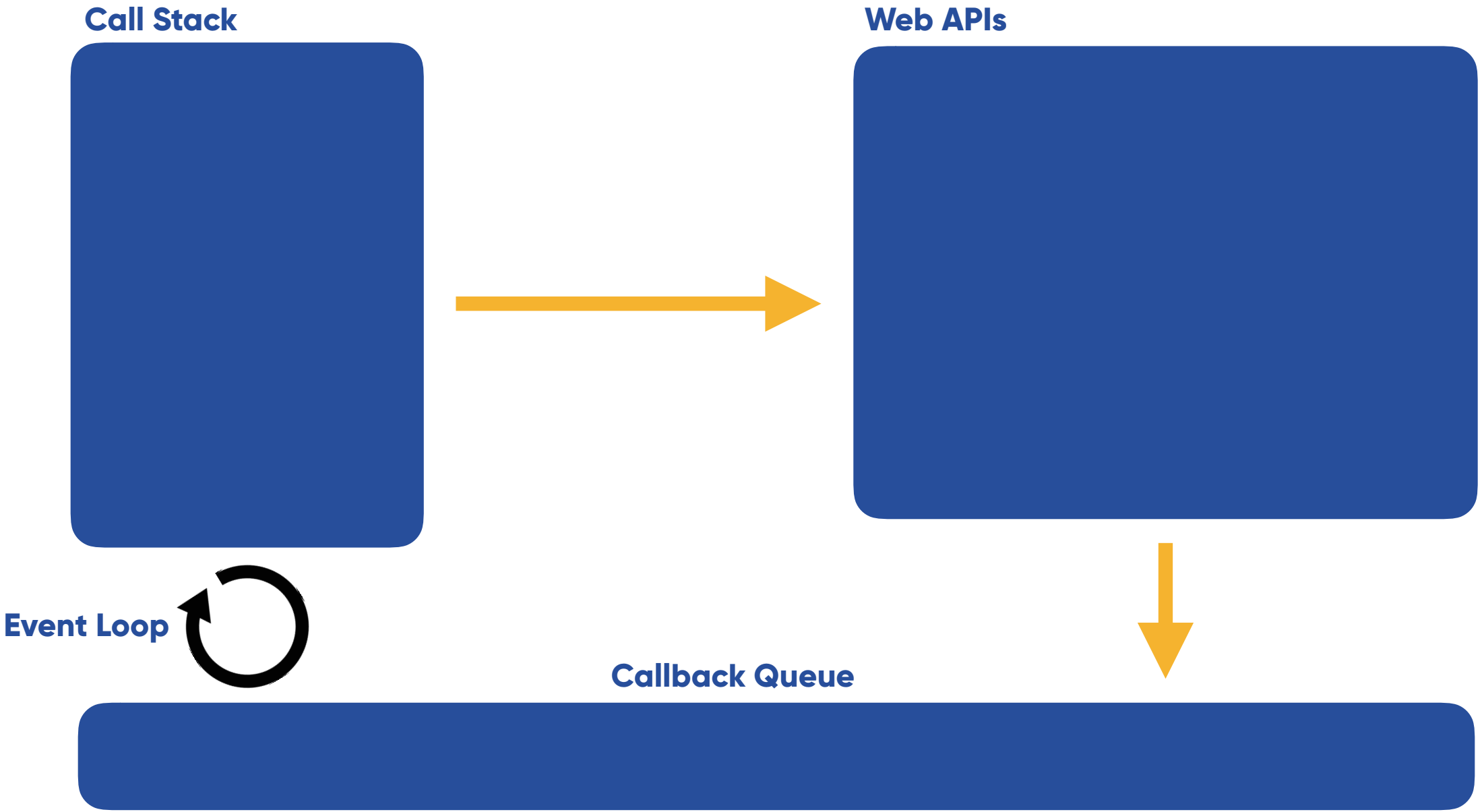
```
let a = 1
let b = 2
let c = 3
let d = 4

console.log(a)

setTimeout(() => {
  console.log(b)
}, 2000)

setTimeout(() => {
  console.log(c)
}, 0)

console.log(d)
```


Call Stack
Web APIs
Event Loop
Callback Queue

Web APIs allow us to do additional stuff that isn't part of the Javascript language. Our code calls an API, which can do something and provide a response.

setTimeout is actually part of a Web API provided by the browser. It is not part of the Javascript language.

```
let a = 1
let b = 2
let c = 3
let d = 4

console.log(a)

setTimeout(() => {
  console.log(b)
}, 2000)

setTimeout(() => {
  console.log(c)
}, 0)

console.log(d)
```

So what's happening here? Let's dive deeper.

We used setTimeout in this example to simulate the idea that some functions take time to complete.

When functions take time, we need ways to handle them so our code doesn't have to wait.

Remember higher-order functions which take a function as a parameter?

Those functions which we pass in have their own name:

We call them **callback functions.**

What's the point in a callback function? Check out the following code

```
let myPosts = ["post1", "post2", "post3"]


const allPosts = () => {
 setTimeout(() => {
   for(let i = 0; i < myPosts.length; i++){
    console.log(myPosts[i])
  }
 }, 1000)
}


const createPost = (post) => {
 setTimeout(() => {
   myPosts.push(`${post}`)
 }, 2000)
}


createPost("post4")
allPosts()
```

Console

"post1"

"post2"

"post3"

Even though we called createPost first, then logged all our posts to the console, post 4 is not logged. Why?

```
let myPosts = ["post1", "post2", "post3"]


const allPosts = () => {
 setTimeout(() => {
   for(let i = 0; i < myPosts.length; i++){
    console.log(myPosts[i])
  }
 }, 1000)
}


const createPost = (post) => {
 setTimeout(() => {
   myPosts.push(`${post}`)
 }, 2000)
}


createPost("post4")
allPosts()
```

Console

```
"post1"
"post2"
"post3"
```

We need to call allPosts AFTER we know createPost is completed. That's where callbacks come in.

```
let myPosts = ["post1", "post2", "post3"]


const allPosts = () => {
 setTimeout(() => {
   for(let i = 0; i < myPosts.length; i++){
    console.log(myPosts[i])
  }
 }, 1000)
}


const createPost = (post, callback) => {
 setTimeout(() => {
   myPosts.push(`${post}`)
   callback()
 }, 2000)

}


createPost("post4", allPosts)
```

By passing allPosts in as a parameter, we can ensure we only call it after createPost is completed (however long it takes)

The benefit of using the callback design pattern is that you can pass in whatever function you like. Rather than hard coding functions in the order you want.

Is there an alternative to callback functions?

Yep

```
let users = ['dan', 'ben', 'stuart']

const addUser = (username) => {
  setTimeout(()=> {
    users.push(username)
  },2000)
}

const getUsers = () => {
  setTimeout(() => {
    console.log(users)
  }, 1000)
}


addUser('Charlie')
getUsers()
```

Another problem similar to the one before. Even though we added a user first, when we log all the users it isn't there!

{CN}®

```
let users = ['dan', 'ben', 'stuart']

const addUser = (username, callback) => {
  setTimeout(()=> {
    users.push(username)
    callback()
  },2000)
}

const getUsers = () => {
  setTimeout(() => {
    console.log(users)
  }, 1000)
}


addUser('Charlie', getUsers)
```

Solution with a callback function

{CN}®

I promise.

Javascript has a built in function type called a Promise.

It essentially, promises to do something once a function has completed. Useful.

A promise has three states:

Pending.
Resolved.
Rejected.

```
const addUser = (username) => {
  return new Promise((resolve, reject) => {

    setTimeout(()=> {
    users.push(username)

      let error = false

      if (!error){
        resolve()
      } else {
        reject('Oops there has been an error')
      }
  },2000)

  })
}
```

Solution with promises

{CN}®

Solution with promises

We can then use .then( ) and .catch( ) to handle the resolved or rejected routes.

Although, we usually put them on separate lines for readability.

```
addUser('Charlie').then(getUsers).catch((err) => {console.log(err)})
```

Solution with promises

We can then use .then( ) and .catch( ) to handle the resolved or rejected routes.

```
addUser('Charlie')
    .then(getUsers)
    .catch((err) => {console.log(err)})
```

{CN}®

We'll use promises a lot more when we work with servers and databases.

Async and Await

Standard joke incoming...

{CODENATION}

Keywords

Async: defines a function/method as asynchronous

Await: waits for code to finish processing

async/await is a more elegant way to handle promises.

Before with native promises

```
addUser('Charlie')
  .then(getUsers)
  .catch((err) => {console.log(err)})
```

With async/await

```
async function init(){
  await addUser('Charlie')
  getUsers()
}

init()
```

const init = async () => { }

{CN}®

```
const myAsyncFunction = () => {

  return new Promise((resolve, reject) => {
      let a = 1 + 1
      if (a == 2) {
      resolve('My promise has been resolved')
    } else {
      reject('My promise has been rejected')
    }
  })
}

myAsyncFunction()
.then((message) => {console.log(message)})
.catch((message) => {console.log(message)})
```

One more example:

Without async/ await

{CN}®

```
const myAsyncFunction = () => {

  return new Promise((resolve, reject) => {
      let a = 1 + 1
      if (a == 2) {
      resolve('My promise has been resolved')
    } else {
      reject('My promise has been rejected')
    }
  })
}

async function init() {
  let response = await myAsyncFunction()
  console.log(response)
}

init()
```

One more example:

With async/ await

{CN}®

```
function init() {
  let response = myAsyncFunction()
  console.log(response)    // just logs a pending promise object
}

init()
```

If I remove the async and await keywords, the console log will not wait for myAsyncFunction to be resolved. So it will just log a pending promise request.

{CN}®

```
async function init() {
  let response = await myAsyncFunction()
  console.log(response) // waits for promise to be resolved then logs
}

init()
```

When javascript hits the await keyword, it will pause the execution of that function (and go off to do other things) then once the promise is resolved, it will carry on.

{CN}®

In our examples, we created and returned a new promise.

In most real-world cases, you will not be creating promises. You will be handling them when they are returned from things like data calls.

Understanding asynchronous javascript is key to working with data and servers.

{CN}®

We will come back to it when we start fetching data for our applications.

Revisiting Learning Objectives

To understand what **synchronous and asynchronous** mean.

To be able to work with higher-order functions, and to understand what **callback functions** are.

To understand what **Promises** are in Javascript.

To understand the keywords **async** and **await**.