the Master Course

{CODENATION}

Intermediate JavaScript

Asynchronous JavaScript

{CODENATION}

{CODENATION}

Learning Objectives

To explore what synchronous and asynchronous mean

To be able to work with higher order functions and be familiar with callback functions

To recognise what promises are in JavaScript

To identify the async and await keywords and use them to handle data

What does synchronous mean ?

One thing at a time.

In order.

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.

Someone decides they want to load their plate up with as many sausage rolls as they possibly can.

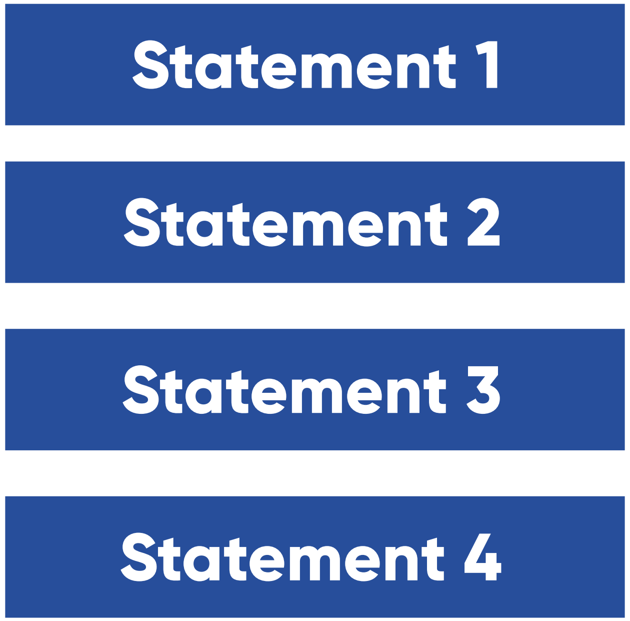
While they do this we have to wait until they are finished.

Back to the call stack.

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

Call Stack

Keeps track of our code as it runs.


Statement 1

Statement 2

Statement 3

Statement 4

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

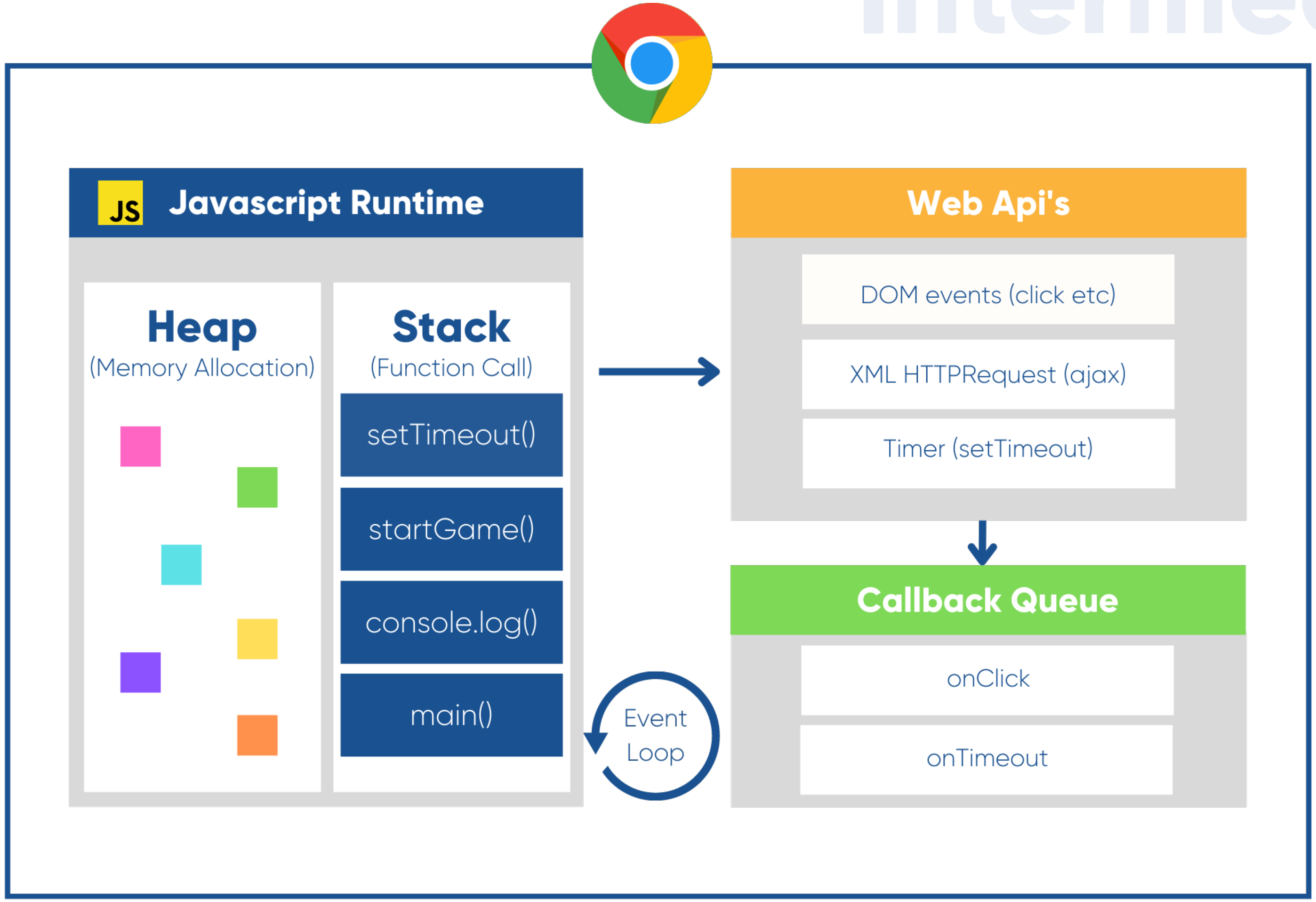
Asynchronous JavaScript allows us to carry on down the call stack, without getting blocked by a slow function.

```
console.log(1);
console.log(2);
console.log(3);
console.log(4);
```

In what order are things printed to the console?

Let’s use a function called **setTimeout()** to stimulate a function taking ages.

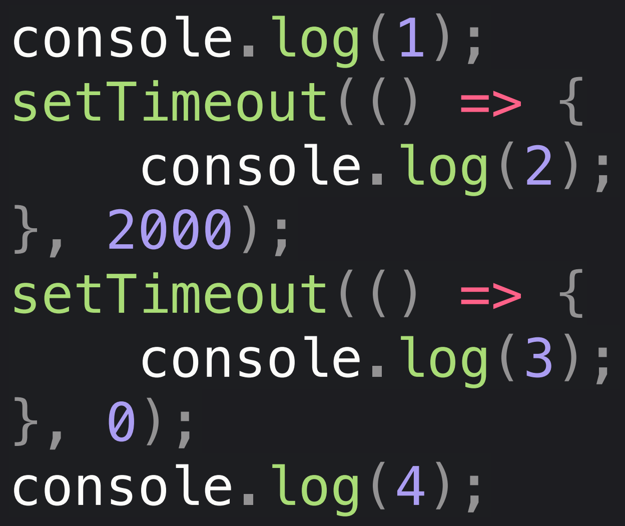
```javascript
console.log(1);
setTimeout(() => {
    console.log(2);
}, 2000);
setTimeout(() => {
    console.log(3);
}, 0);
console.log(4);
```



JavaScript Engine Operation

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.



```
console.log(1);
setTimeout(() => {
    console.log(2);
}, 2000);
setTimeout(() => {
    console.log(3);
}, 0);
console.log(4);
```

We used **setTimeout** in this example to stimulate the idea that some functions take some 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 in a function as a parameter?

Those functions which we pass in have their own name:

We call them **callback functions.**

What's the point of a callback function?

```javascript
let myPosts = ['post1', 'post2', 'post3'];

const allPosts = () => {
    setTimeout(() => {
        myPosts.map((post) => console.log(post));
    }, 1000);
};

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

createPost('post4');
allPosts();
```

Console

```
post 1
post 2
post 3
```

Even though we call the function createPost first and then log out all of our posts to the console, post 4 is not logged, why?

```
let myPosts = ['post1', 'post2', 'post3'];

const allPosts = () => {
    setTimeout(() => {
        myPosts.map((post) => console.log(post));
    }, 1000);
};

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

createPost('post4');
allPosts();
```

Console

```
post 1
post 2
post 3
```

We need to call allPosts AFTER we know createPost is completed. This is where a callback can be used.

```
let myPosts = ['post1', 'post2', 'post3'];

const allPosts = () => {
    setTimeout(() => {
        myPosts.map((post) => console.log(post));
    }, 1000);
};

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

createPost('post4', allPosts);
```

Console

```
post 1
post 2
post 3
post 4
```

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.

```
let users = ['Dave', 'Gary', 'Steve'];

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

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

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

Console

```
['Dave', 'Gary', 'Steve']
```

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

```
let users = ['Dave', 'Gary', 'Steve'];

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

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

addUser('Charlie', getUsers);
```

Console

```
['Dave', 'Gary', 'Steve', 'Charlie']
```

A solution with a callback function.

Promises

JavaScript has a built in function type called a **promise.** It essentially, promises to do something once a function has **completed**.

A promise has three states:

Pending.
Resolved.
Rejected.

```
let users = ['Dave', 'Gary', 'Steve'];

const addUser = (username) => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            users.push(username);
             const error = false;
            if (!error) {
                resolve();
            } else {
                reject('oops there has been an error');
            }
        }, 2000);
    });
};

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

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

Intermediate JS

Console

['Dave', 'Gary', 'Steve', 'Charlie']

If there is an error, it will run the reject() method , and continue with .catch to pass the parameter to run. If there is no error, it will resolve and not hit the catch block.

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

Async and Await

Intermediate JS

{CODENATION}

Keywords

Async: defines a function/method as asynchronous

Await: waits for code to finish processing async and await is a more elegant way to handle promises.

Intermediate JS

```
let users = ['Dave', 'Gary', 'Steve'];

const addUser = (username) => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            users.push(username);
             const error = false;
            if (!error) {
                resolve();
            } else {
                reject('oops there has been an error');
            }
        }, 2000);
    });
};
const getUsers = () => {
    setTimeout(() => {
        console.log(users);
    }, 1000);
};

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

Console

['Dave', 'Gary', 'Steve', 'Charlie']

With native promises.

```
let users = ['Dave', 'Gary', 'Steve'];

const addUser = (username) => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            users.push(username);
            const error = false;
            if (!error) {
                resolve();
            } else {
                reject('oops there has been an error');
            }
        }, 2000);
    });
};

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

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

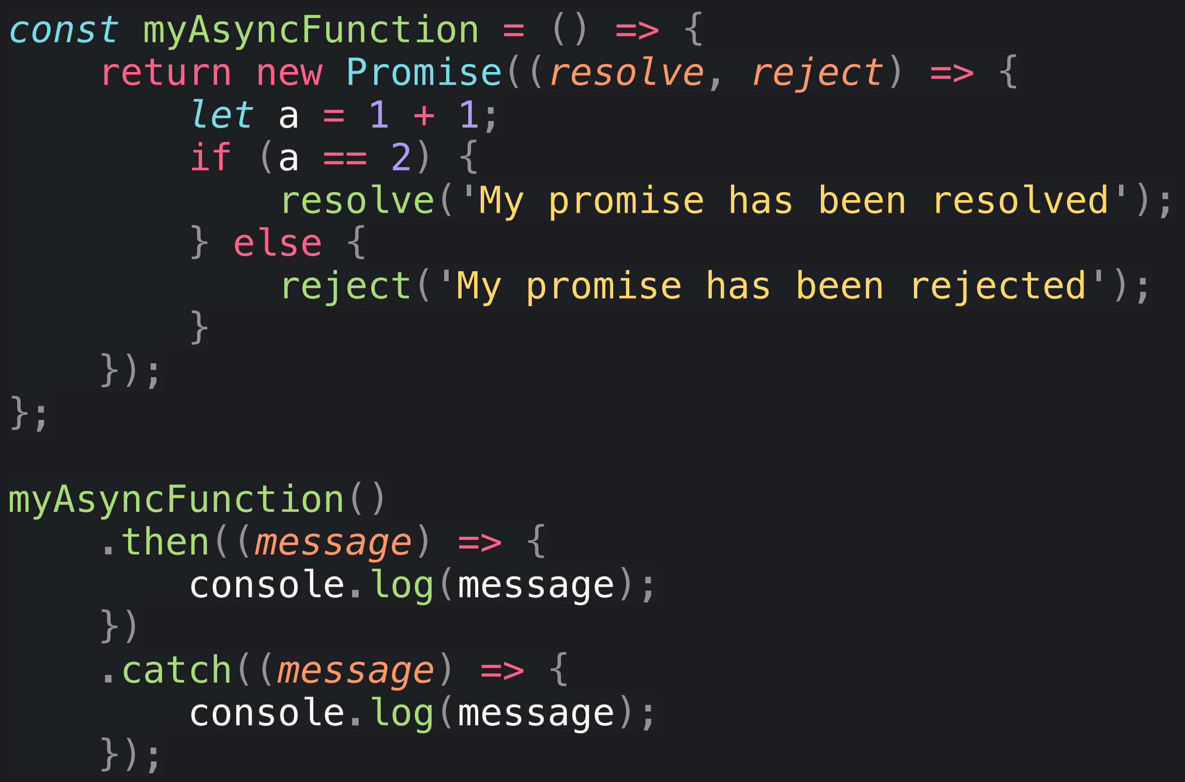
Intermediate JS

Console

['Dave', 'Gary', 'Steve', 'Charlie']

With async and await.

One more example



```
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);
    });
```

Console

```
My promise has been resolved
```

Without async and await.

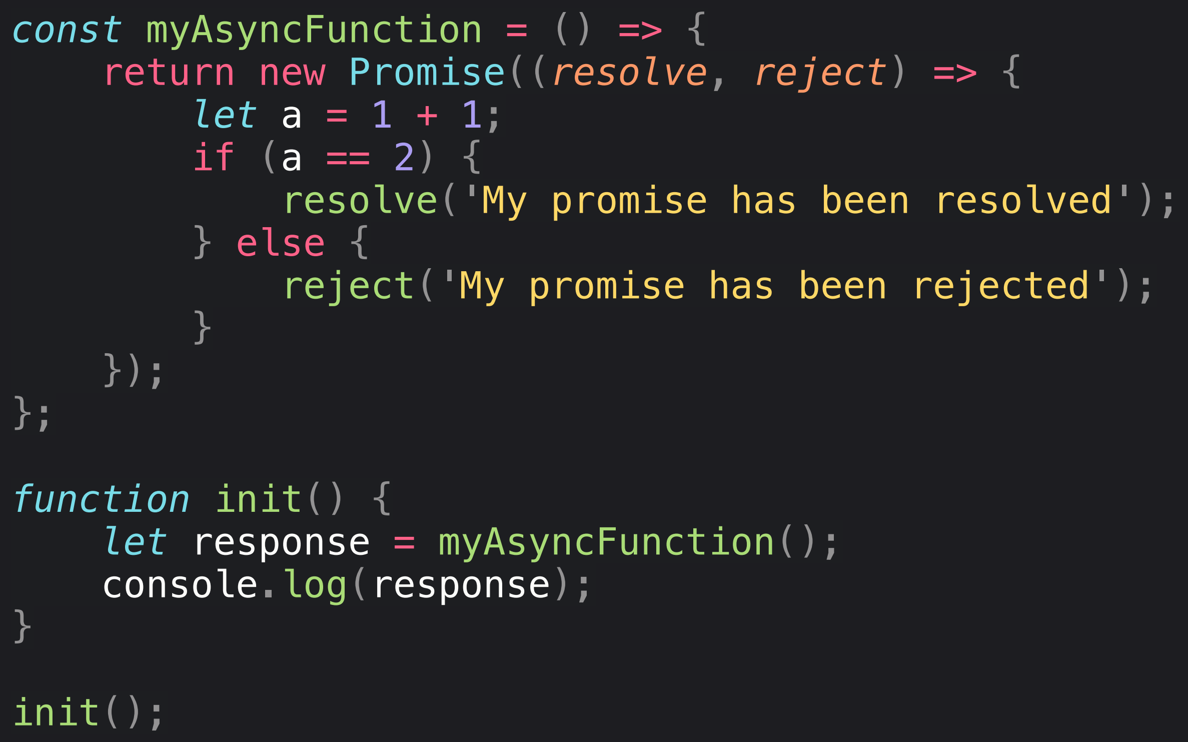
Intermediate JS

```javascript
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();
```

Console

```
My promise has been resolved
```

With async and await.



```
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');
        }
    });
};

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

init();
```

Console

```
Promise { 'My promise has been resolved' }
```

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

Try, catch

The **try** statement allows you to define a block of code that will be **checked** for **errors** while it runs. If an error occurs in the **try** block, the **catch** statement allows you to define a block of code that will be executed.

```
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() {
    try {
        let response = await myAsyncFunction();
        console.log(response);
    } catch (error) {
        console.log(error);
    }
}

init();
```

With try/catch error handling.

```
const myPosts = [
    { title: 'Post One', body: 'This is post one body' },
    { title: 'Post Two', body: 'This is post two body' },
];
function getPosts() {
    setTimeout(() => {
        myPosts.forEach((post) => {
            console.log(post.title);
        });
        console.log(myPosts);
    }, 1000);
}
function createPost(post) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            myPosts.push(post);
            const error = true;
            if (!error) {
                resolve();
            } else {
                reject('something went wrong');
            }
        }, 5000);
    });
}

async function init() {
    try {
        await createPost({ title: 'Post Three', body: 'This is post three body' });
        getPosts();
    } catch (error) {
        console.log(error);
    }
}
init();
```

```
//Output:
//Post One
//Post Two
//Post Three
//[{title:’Post One’,body:'This is post one body'},
//{title:’Post Two’,body:'This is post two body'},
//{title:’Post Three’,body:'this is post three body'}]
```

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.

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

{CODENATION}

Learning Objectives

To explore what synchronous and asynchronous mean

To be able to work with higher order functions and be familiar with callback functions

To recognise what promises are in JavaScript

To identify the async and await keywords and use them to handle data