
JAVASCRIPT

JS

&



Contents

1. What is it ?
2. How it works ?
3. Callbacks
4. Promises
5. Async Await

What is Javascript ?

Single-threaded

Non-blocking

Asynchronous

Concurrent

language

I have a Call stack

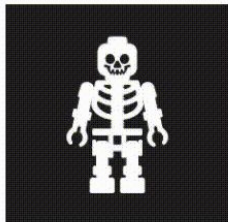
Event loop

Callbacks

Some other **apis**

And stuff

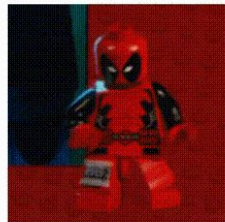
HTML
structure



CSS
presentation/appearance



JavaScript
dynamism/action



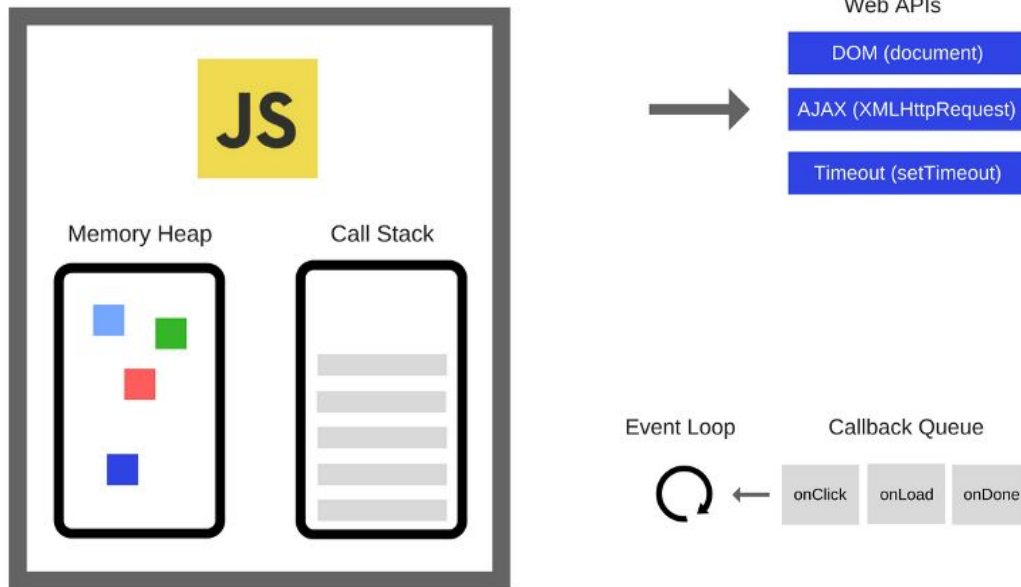
How it works ?

Javascript Runtime Environment

In the context of a browser this is comprised of the following elements:

Callstack:

- One-thread
- One-call-stack
- One-thing-at a time



Call stack

```
console.log("Start ...");  
  
setTimeout(function () {  
  console.log("Hello I am Javascript...")  
}, 3000);  
  
console.log("End...");
```

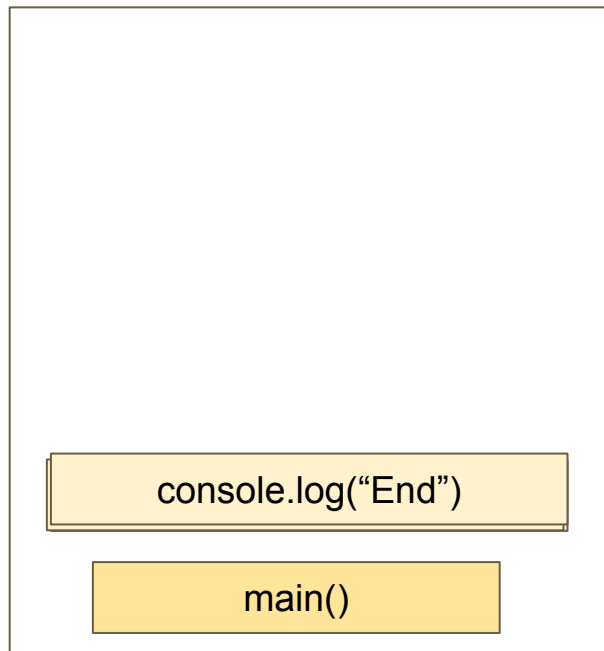
Output:

Start...
Hello I am Javascript...
End...

My Output:

Start...
End...
Hello I am Javascript...

stack



Concurrency & Event Loop

```
console.log("Start ...");  
  
setTimeout(function () {  
  console.log("Hello I am Javascript...")  
}, 3000);  
  
console.log("End...");
```

Console

Start...

End...

Hello I am J...

Stack

console.log("End...")

webAPIs

setTimeout(3000)



Event Loop



Task Queue

cb

Callback

Callback is a function that is passed as an argument to another function and its execution is delayed until that function in which it is passed is executed.

```
console.log("Start ...");

function timeOut(cb) {
    setTimeout(function(){
        console.log("Hello I am Javascript...")
        cb()
    }, 2000);
}

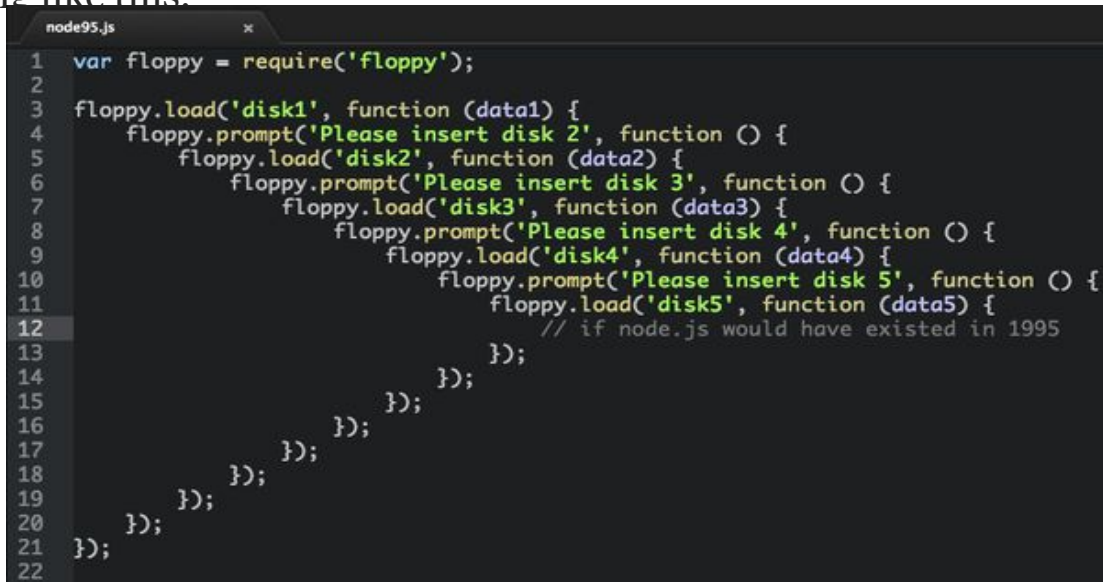
function printEnd() {
    console.log("End ...")
}

timeOut(printEnd)
```

function `printEnd()` is passed as an argument inside `.timeOut()`, so we are not actually calling it here, so there are no parentheses `()`.

Callback Hell

Imagine there are many nested callbacks inside a callback; it would look something like this.



```
node95.js x
1  var floppy = require('floppy');
2
3  floppy.load('disk1', function (data1) {
4    floppy.prompt('Please insert disk 2', function () {
5      floppy.load('disk2', function (data2) {
6        floppy.prompt('Please insert disk 3', function () {
7          floppy.load('disk3', function (data3) {
8            floppy.prompt('Please insert disk 4', function () {
9              floppy.load('disk4', function (data4) {
10               floppy.prompt('Please insert disk 5', function () {
11                 floppy.load('disk5', function (data5) {
12                   // if node.js would have existed in 1995
13                 });
14               });
15             });
16           });
17         });
18       });
19     });
20   });
21 });
22
```

- they do not scale well for even moderately complex asynchronous code.

Promises

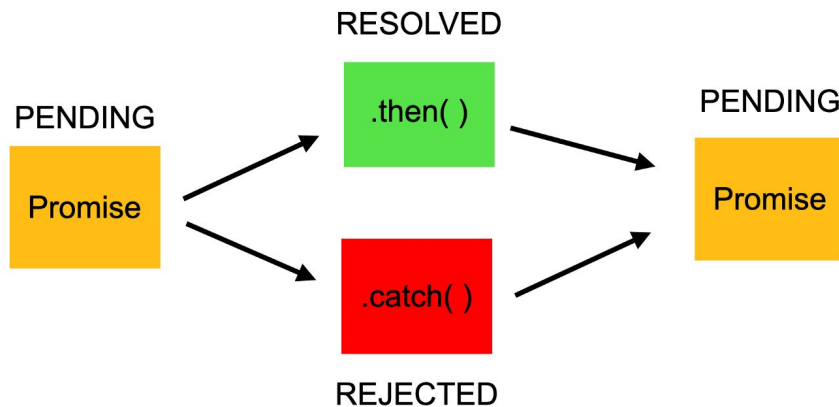
**Async
/
Await**

E.g: The Script Bulb

Promises

A promise is an **object** that may produce a **single value** sometime in the future. Either a **resolved** value or a **rejected** value.

Possible states



Promises

Use a constructor to create a Promise object

```
const myPromise = new Promise();
```

It takes two parameters, one for success (resolve) and one for fail (reject):

```
const myPromise = new Promise((resolve, reject) => {  
  // condition  
});
```

.then() for resolved Promises

.catch() for errors or failures

Promises

```
const myPromise = new Promise(function(resolve, reject) {  
  setTimeout(function() {  
    resolve();  
  }, milliseconds);  
});
```

```
console.log("Turning off . . .")  
myPromise.then(function(){  
  statusText.innerHTML = "Off";  
  image.src = "assets/pic_bulboff.gif";  
  console.log("Turn Off! Complete. . .")  
});
```

```
console.log("End of block...")
```

Benefits of Promises

- Improves Code Readability
- Better handling of asynchronous operations
- Better flow of control definition in asynchronous logic
- Better Error Handling

Async / Await

Caution: Before getting in details of Async and Await, you should have a good understanding of **Promises** in JavaScript

- Introduced in JavaScript Version ES6 - ECMAScript 2015
- When we append the keyword “async” to the function, this function returns the Promise by default on execution
 - The function contains some Asynchronous Execution
 - The returned value will be the Resolved Value for the Promise.
- Capturing Promise ?

```
async function asyncPromise() {  
    return "I am JS";  
}  
  
asyncPromise().then(function(data){  
    console.log(data)  
});
```

Await

- Promises are asynchronous and waiting on another thread for completion
 - JavaScript does not wait for the promise to resolve, it executes further
 - Once the promise is resolved the callback function is invoked.
-
- Adding “**await**” before a promise makes the execution thread to wait for asynchronous task/promise to resolve before proceeding further.
 - When we are adding the “**await**” keyword, we are introducing synchronous behavior to the application.
 - Even the promises will be executed synchronously.

```
async function returnPromises() {  
  return new Promise((resolve) => {  
    setTimeout(() => {  
      console.log("Promise Executed...");  
      resolve("Sample Data");  
    }, 3000);  
  });  
}  
  
async function ExecuteFunction() {  
  var newData = "UniCourt";  
  var getPromise = await returnPromises();  
  console.log(newData);  
  console.log(getPromise);  
}
```




References

- [Loupe tool](#)
- [Async and Await](#)

Typescript

- TypeScript adds additional syntax to JavaScript to support a tighter integration with your editor. Catch errors early in your editor
- TypeScript code converts to JavaScript, which runs anywhere JavaScript runs: In a browser, on Node.js or Deno and in your apps.

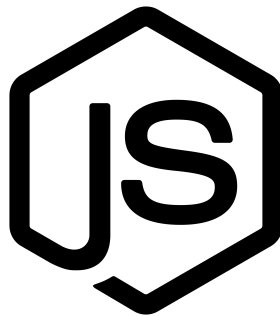
Reference

[Typescript Basics](#)

[Types](#)

Node JS

- Node.js is an open-source and cross-platform JavaScript runtime environment.
- **server-side JavaScript**
- **Express.js** or simply Express, is a back end web application framework for Node.js,



References

[Express JS](#)

[Express Js with Typescript](#)