Async / Await

What are callbacks?

Callbacks are just the name of a **convention** for using JavaScript functions.

There isn't a special thing called a 'callback' in the JavaScript language, it's just a convention. Instead of immediately returning some result like most functions, functions that use callbacks take some time to produce a result. The word 'asynchronous', aka 'async' just means 'takes some time' or 'happens in the future, not right now'.

Function

When you call a normal function
you can use its return value:

```
var result = multiplyTwoNumbers(5, 10)

console.log(result)

// 50 gets printed out
```

Function nope

However, functions that are async (JS is async!) and use callbacks don't return anything right away.

```
var photo = downloadPhoto('http://coolcats.com/cat.gif')

// photo is 'undefined'!
```

In this case the gif might take a very long time to download, and you don't want your program to pause (aka 'block') while waiting for the download to finish.

Callback

Instead, you store the code that should run after the download is complete in a function.

```
downloadPhoto('http://coolcats.com/cat.gif', handlePhoto)
function handlePhoto (error, photo) {
  if (error) console.error('Download error!', error)
  else console.log('Download finished', photo)
}
console.log('Download started')
```

Callback Hell

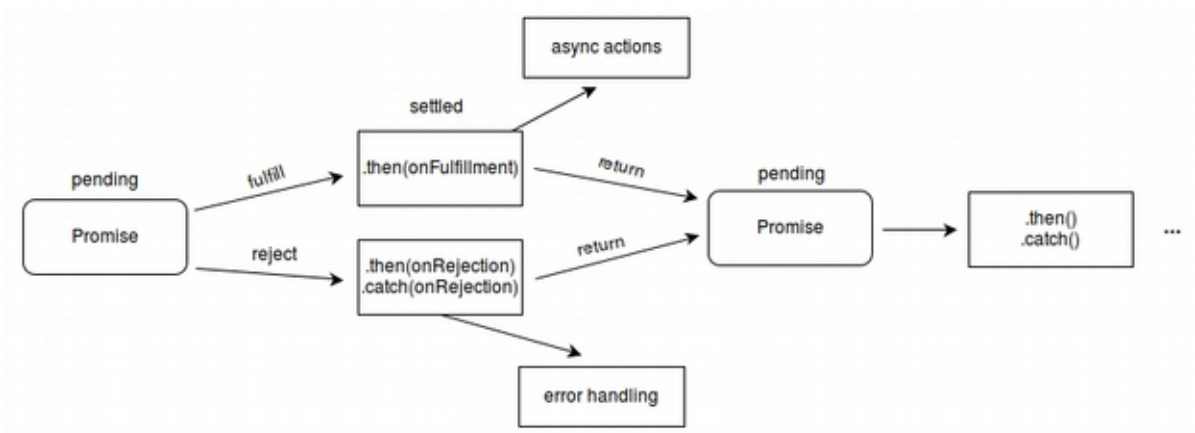
callbackhell.js

```
var floppy = require('floppy');

floppy.load('disk1', function (data1) {
    floppy.prompt('Please insert disk 2', function() {
        floppy.load('disk2', function (data2) {
            floppy.prompt('Please insert disk 3', function() {
                floppy.load('disk3', function (data3) {
                    floppy.prompt('Please insert disk 4', function() {
                        floppy.load('disk4', function (data4) {
                            floppy.prompt('Please insert disk 5', function() {
                                floppy.load('disk5', function (data5) {
                                    floppy.prompt('Please insert disk 6', function() {
                                        floppy.load('disk6', function (data6) {
                                            //if node.js would have existed in 1995
                                        });
                                    });
                                });
                            });
                        });
                    });
                });
            });
        });
    });
});

```

Asynchronous JavaScript, or JavaScript that uses callbacks, is hard to get right intuitively. A lot of code ends up looking like this:

Promises

The Promise object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.



https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

ES6: Promises

Promises were a revelation in JavaScript, providing an alternative to the JavaScript callback hell we'd quickly found ourselves in.

Promises also allowed us to introduce and better handle asynchronous tasks. While promises were an improvement over callback hell, they still require lots of thens which can become messy.

Howto

Promises are actually pretty simple: they are an object that represents a value which will **eventually** be available.

It might already be available, or it might be ready in a little while.

You interact with promises by passing a callback to its **then** function. That's why they are sometimes called "thennables."

Callback vs Promise

```
require("request")
    .get("http://www.google.com", function(err, response) {
        console.log(response.body)
    })
```

And a promise version. Of course it's very similar. The main difference is we're passing our callback to then.

```
require("popsicle")
    .get("http://www.google.com")
    .then(function(response) {
        console.log(response.body);
    })
```

Callback / Promise

More complex example/comparasion

```
require("request")
    .get("http://www.google.com", function(err, response) {
        if (err) {
            console.error(err);
        } else {
            require("fs")
            .writeFile("google.html", response.body, function(err) {
                if (err) {
                    console.error(err);
                } else {
                    console.log("wrote file");
                }
            })
        }
    })
```

```
require("popsicle").get("http://www.google.com")
    .then(function(response) {
        return require("fs-promise").writeFile("google.html", response.body);
    })
    .then(function() {
        console.log("wrote file");
    })
    .catch(function(err) {
        console.log(err);
    })
```

Fixing the complexity

This is a promisified version of our first "callback hell" example. It's much cleaner, but still requires lots of extra syntax and awkward workarounds for the scopes introduced by new functions.

```
function handler(request, response) {
    User.get(request.user, function(err, user) {
        if (err) {
            response.send(err);
        } else {
            Notebook.get(user.notebook, function(err, notebook) {
                if (err) {
                    return response.send(err);
                } else  {
                    doSomethingAsync(user, notebook, function(err, result) {
                        if (err) {
                            response.send(err)
                        } else {
                            response.send(result);
                        }
                    });
                }
            });
        }
    })
}
```

```
function(request, response) {
    var user, notebook;

    User.get(request.user)
    .then(function(aUser) {
        user = aUser;
        return Notebook.get(user.notebook);
    })
    .then(function(aNotebook) {
        notebook = aNotebook;
        return doSomethingAsync(user, notebook);
    })
    .then(function(result) {
        response.send(result)
    })
    .catch(function(err) {
        response.send(err)
    })
}
```

ES7: Await

Finally, the **await** based version of the same code. No awkward functions, clear linear style code, explicit asynchronous points, and native try/catch error handling.

```
async function(request, response) {
    try {
        var user = await User.get(request.user);
        var notebook = await Notebook.get(user.notebook);
        response.send(await doSomethingAsync(user, notebook));
    } catch(err) {
        response.send(err);
    }
}
```

The simplest explanation of how this works is that **await** takes a promise, waits for it's value to be available, and then returns that value.

All promises we made...

This code returns a promise (represented by a Request object) if you run it.

```
require("popsicle").get("http://www.google.com");
```

Whereas, this code returns the value of that promise (the Response object) once it's ready.

```
await require("popsicle").get("http://www.google.com");
```

It works...

And of course, if you await a non promise……
things work just fine.

```
await "Hello!"
```

Await

There's one more slight **complication**.

This function will produce a syntax error. That's because you can only use await inside of a function declared a special way.

```
function x(aPromise) {
    await aPromise
}
```

Here's the corrected version of this simple function. We add the **async** keyword before our function declaration.

```
async function x(aPromise) {
    await aPromise
}
```

"await" is not threads

In a language with threads (multi-thread, C++, PHP, Erlang, etc), concurrency can be out of your control.

Your thread can be interrupted at any time, so we need locks and other synchronization primitives.

But JavaScript's model hasn't changed, it's still single threaded. The code is only interrupted by your explicit command to "await". You can still mess up by sharing state, but that's nothing new.

Concurrency

await gives you explicit control over concurrency. You can combine this with powerful promise utilities like **Promise.all**, which will wait for every promise to finish and then finish itself, to write powerful and yet easy to understand asynchronous code.

```
var P = require("popsicle");
var sites = await Promise.all([
    P.get("http://www.google.com"),
    P.get("http://www.apple.com"),
    P.get("http://www.yahoo.com")
])
```

Wes Bos time

https://www.youtube.com/watch?v=9YkUCxvaLEk

Reference

- Async/await syntax
  https://javascript.info/async-await
- Fetch with Async
  https://dev.to/johnpaulada/synchronous-fetch-with-asyncawait
- Understanding Await Async
  https://ponyfoo.com/articles/understanding-javascript-async-await
- 6 reasons why async win!
  https://hackernoon.com/6-reasons-why-javascripts-async-await-blows-promises-away-tutorial-c7ec10518dd9
- Wes Bos Async/Await example
  https://gist.github.com/wesbos/1866f918824936ffb73d8fd0b02879b4
- David Walsh
  https://davidwalsh.name/async-await
- http://rossboucher.com/await/
  ES7 Async Await @boucher