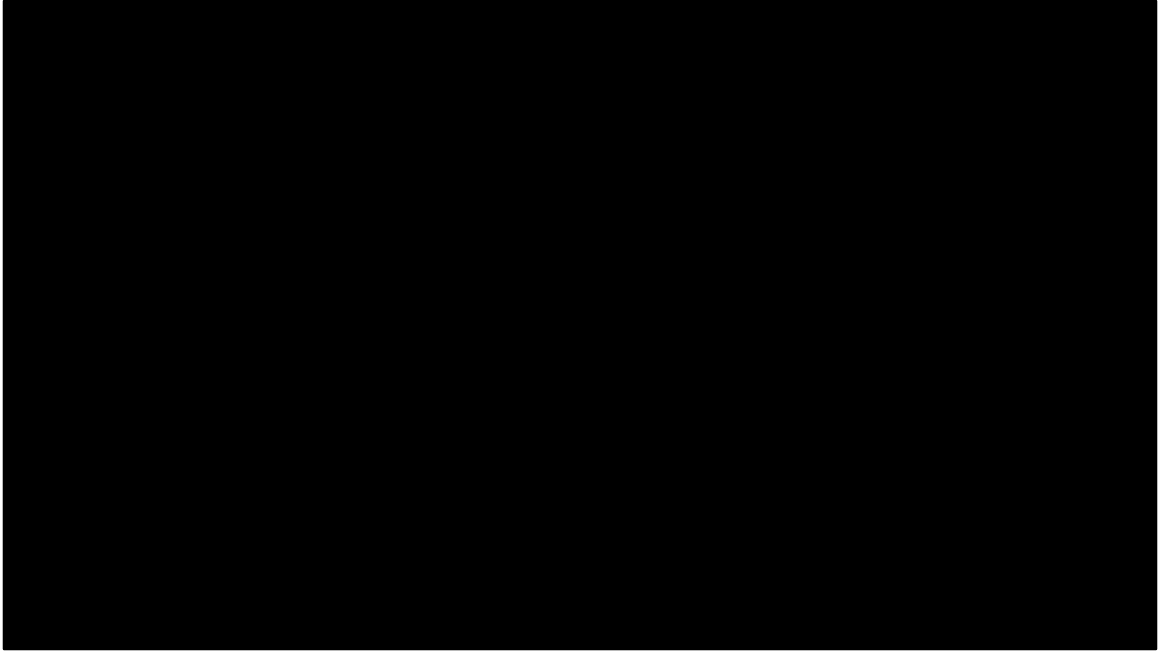


**Why are you
“Awaiting”
to use Async!**



All right, let's get going here.

How we doing tonight?

Okay
So we begin,
With...
Hello...

It is very possible that
You may recognize me



No, I'm not Anthony Edwards...



Nor am I Brian Poeschen.



And I'm not...
No, wait, that is me.

Long story.

Anyways,



Glen R. Goodwin



@areinet



arei.net



github.com/arei

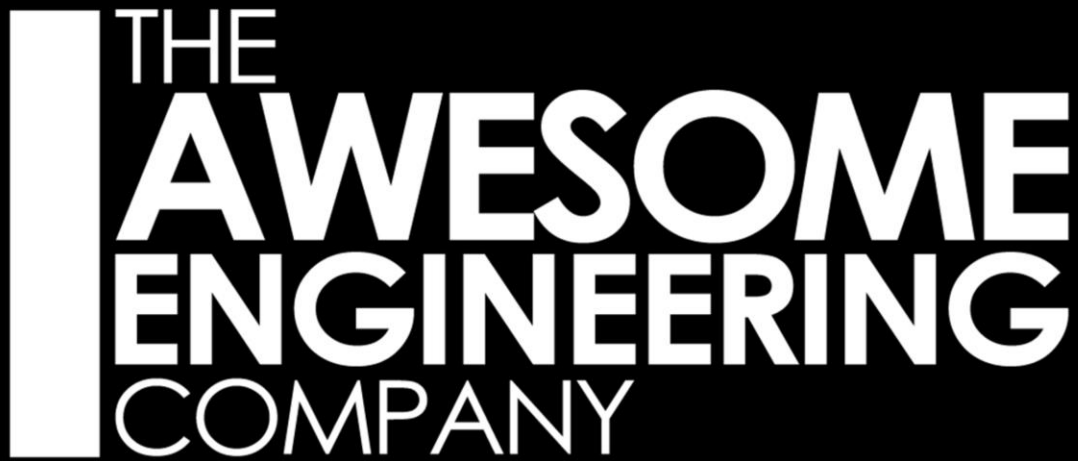
Here I am.

You might know me from here at CharmCityJS
Or NodeSchool
Or from around the community.

But if you don't...
these are my places...

<pause>

and I work for this company

The logo for The Awesome Engineering Company is displayed on a black rectangular background. It features a thick white vertical bar on the left side. To the right of this bar, the word "THE" is written in a small, white, sans-serif font. Below "THE", the word "AWESOME" is written in a large, bold, white, sans-serif font. Below "AWESOME", the word "ENGINEERING" is written in a large, bold, white, sans-serif font. Finally, the word "COMPANY" is written in a medium-sized, white, sans-serif font at the bottom. The overall design is clean and modern, using high contrast between the white text and the black background.

THE AWESOME ENGINEERING COMPANY

The Awesome Engineering Company.
Which makes libraries and enterprise applications
In nodejs for handling massive scale.

I'll hit you with a sales pitch in a few minutes.

But for now I'm here to talk to you about

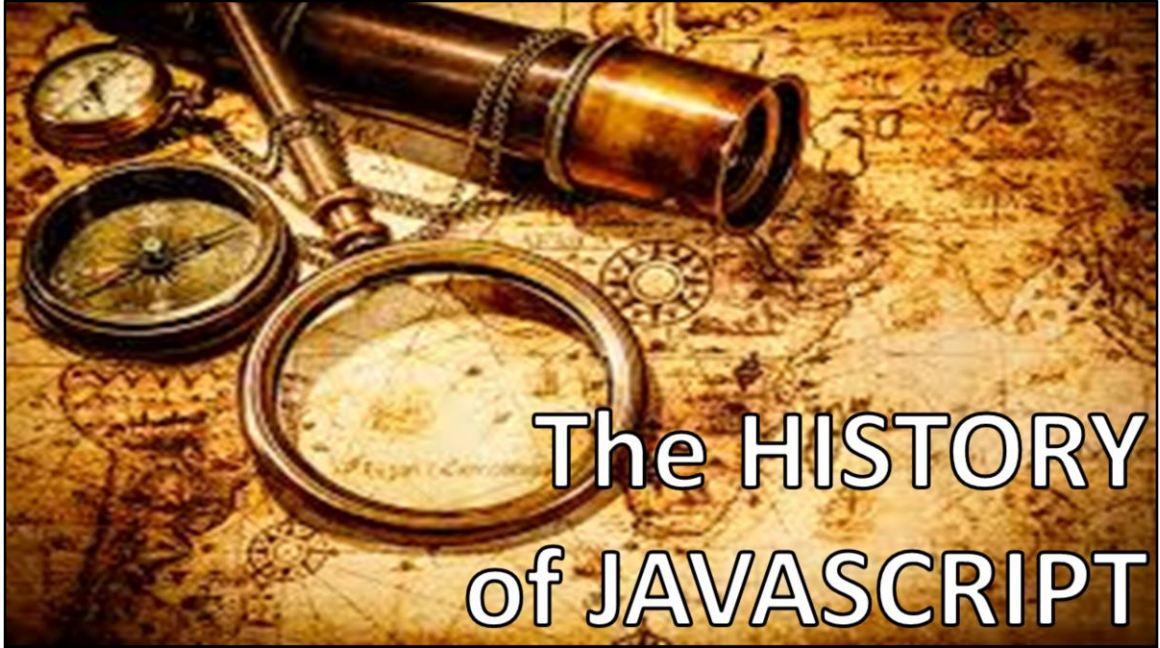
Why are you “Awaiting” to use Async!

Why Are You "Awaiting" to use Async!

(Something about this font makes me want to modulate my tone
Make it sound more baritone, more grandious!

So yea, today we are talking about async/await...

BUT!



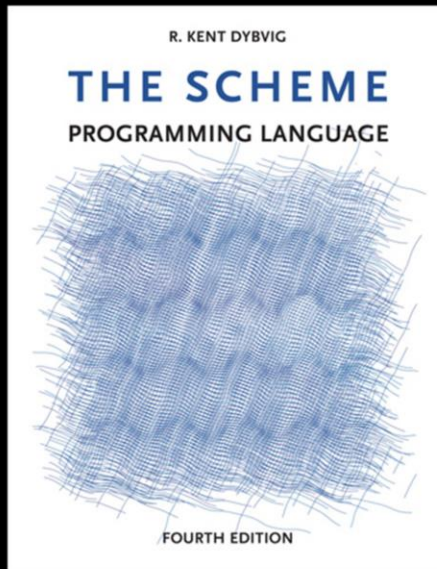
In order to talk to you about Async/Await
we first need to revisit the history of JavaScript.
And I'm sure many of you have heard this before,
so I will try to be quick.



1993 we get the first graphical browser NCSA Mosaic



1994 we get the first commercial browser, Mosaic Netscape 0.9



1995 Netscape realizes that HTML needs a "glue language".
They hire Brendan Eich to implement the Scheme Programming Language for their Browser.

Yes, the initial plan was to have Scheme which derives from LISP as the scripting language of the browser.

If you want to learn how far computer languages have come, go read about LISP some time.

But just as Eich was coming on board,

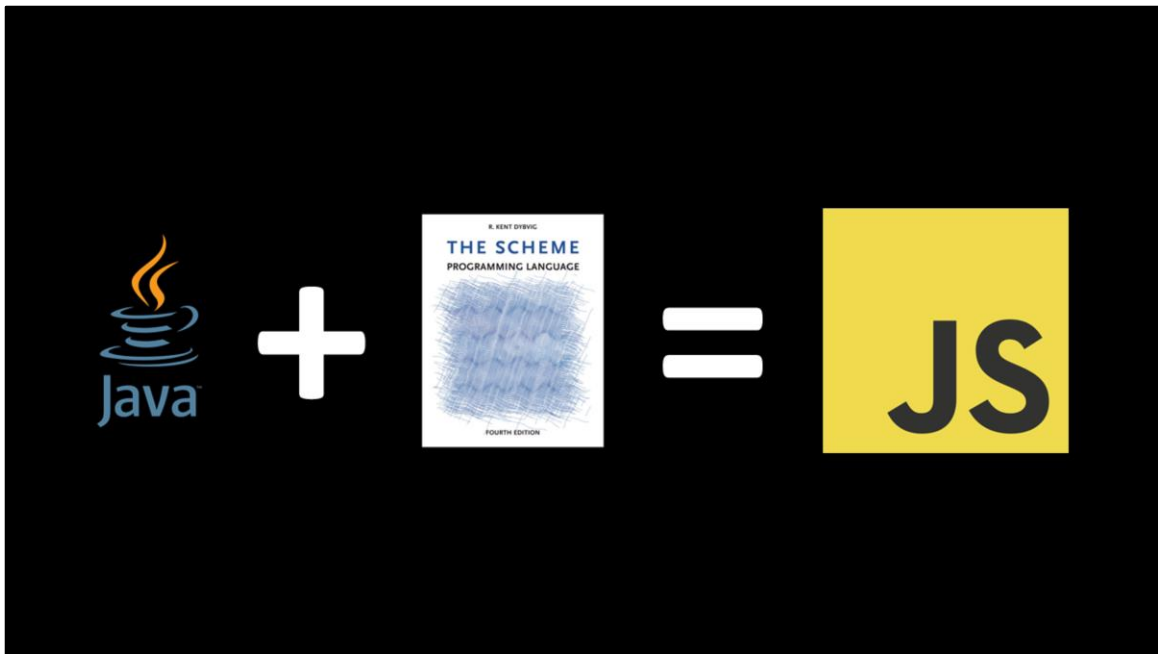


Netscape signed an agreement with Sun Microsystems,
Who invented and owned the Java Programming language
to provide Java in browsers.

And thus the dark age of java applets was born.

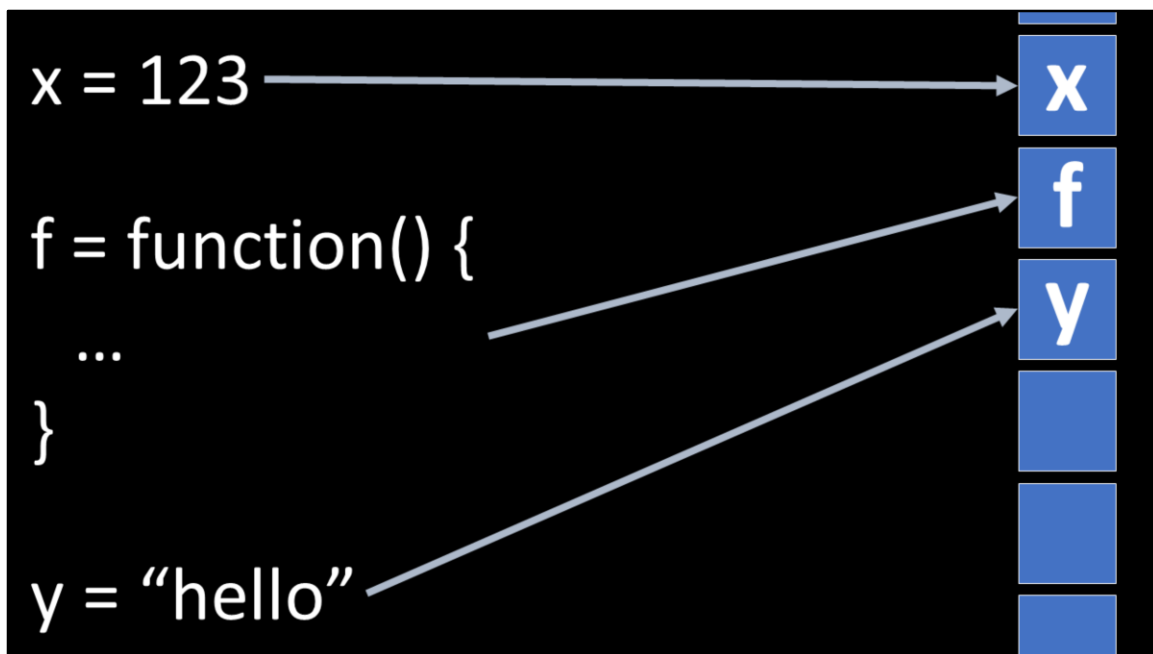
But Netscape realizes that despite Java Applets,
We still need a “glue language” for HTML.

So...



In May of 1995 Eich writes JavaScript in 10 days.
It borrows Java's syntax.

But it also borrows some very interesting features from Scheme.



In particular, Scheme stores functions and variables in the same reference lookup table.

Meaning a variable can be a reference to a function. Until JavaScript this had largely not been done.

What this means is that functions are variables. And they can be passed as arguments to other functions and called within those functions.

Thus the JavaScript Callback was born.

```
on("click",function(){  
    ... do something ...  
});
```

And its gets used pretty much right away in the browser for event handling and the like.

In 1998, we get our first AJAX/XMLHttpRequest efforts.

And thus begins the slightly less dark age known as "Callback Hell"


```
doSomething(function(){  
});
```

CALLBACK HELL

Wherein one function
calls a function with a callback

which in turn calls a function with a callback



which in turn calls a function with a callback
which in turn calls a function with a callback
which in turn calls a function with a callback
which in turn calls a function with a callback
which in turn calls a function with a callback

A lot of people find this frustrating
And a bunch of these people come up with the idea
to add the notion of a Promise to JavaScript.



then

A Promise represents an operation to be completed in the future.

A Promise represents an operation to be completed in the future.

this is sometimes called a Future or Deferred in other languages
And is not unique to javascript.

Promises are callbacks taken up a notch.

But its important to know that promises
Are built on callbacks and
underneath a promise is just callback magic.



Promises/A+

2012ish We see the first Promise specifications
Promises/A and B and C and D and eventually A+
Which is what our current spec is based on.



June 2015 Promise Formally Added to JavaScript.
And everyone can start using them!
Hooray!

And thus begins the medium gray age called Promise Purgatory(tm).



Wherein one promise
chains another promise



then another promise
then another promise
then another promise
then another promise
then another promise
then another promise

But the people who came up with Promises
Knew this was going to happen...
But they also knew that it wasn't really about promises.
Promises were a stepping stone.
To something better.

async/await

Async/await

But first...



A KITTEN!

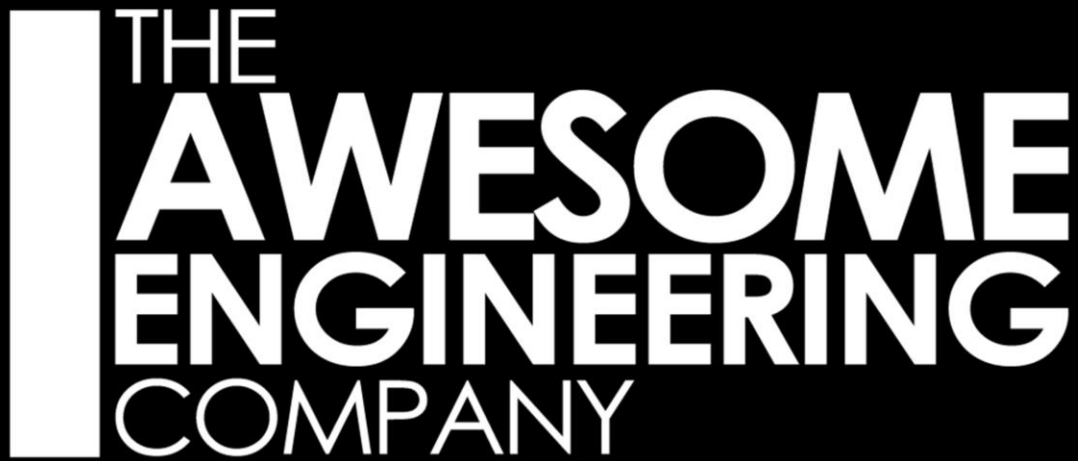
This is our brand new kitten Zephyr.
He is so cute!

<pause>

And now that I've pulled you in with a kitten, a commercial!

commercial

In July of this year,
I quit my job
And started
The Awesome Engineering Company.



THE AWESOME ENGINEERING COMPANY

My first goal at Awesome Engineering
Is to come up with stable, secure, robust tooling
for building enterprise applications with nodejs.
Tooling that is free from external dependencies,
Tooling that is designed with scale in mind,
Tooling that is architected to be cohesive and work together.

In the coming months
I will be sharing some of what I have been building.
And I hope that all of you will help me out.
By taking my stuff for a test drive.
Kicking the tires, Filling a bug or two,
Or just generally talking with me about the enterprise problems you see with nodejs.

And If you are interested you can checkout my initial work right now at

github.com/awesomeeng
npmjs.com/org/awesomeeng

Github and/or npm.

<TAKE PAUSE>

Okay.

Commercial over.

Now back to...

async/await

Async/Await

```
async function HelloWorld() {  
  console.log("hello");  
  await sleep(1000);  
  console.log("world");  
};
```

The purpose of Async/Await is to make javascript...

wait...

for a promise to resolve or reject.

That's it.

Nothing complicated.

But its so much more than that!

ONE

```
async function HelloWorld() {  
  console.log("hello");  
  await sleep(1000);  
  console.log("world");  
};
```

using await means no more then() methods.

TWO

```
async function HelloWorld() {  
  console.log("hello");  
  await sleep(1000);  
  console.log("world");  
};
```

TWO

values returned from the promise, are just returned from the await.

THREE


```
async function HelloWorld() {  
  console.log("hello");  
  await sleep(1000);  
  console.log("world");  
};
```

when a promise rejects, it just becomes a thrown error!
That you can catch!
With a try/catch block.
No more catch() methods!

FOUR

```
async function HelloWorld()  
  console.log("hello");  
  await sleep(1000);  
  console.log("world");  
};
```

4

async/await is fundamentally cleaner to read.
The flow is inherent in the code.

FIVE



A KITTEN!

How can you say no to him?

So, now that you've had a second kitten,
Let take a look at how we use `async/await`.

BASIC EXAMPLE

Here is a basic example:

BASIC EXAMPLE

Sleep =>
wait for n milliseconds
return

Say we have a function, sleep,
which returns a promise
that resolves after n milliseconds.

BASIC EXAMPLE

Sleep =>

```
wait for n milliseconds  
return
```

HelloWorld =>

```
print "hello"  
sleep 1000ms  
print "world"
```

Next, we have a HelloWorld function
Which we want to print out "hello",
wait 1000ms,
and then print out "world".

BASIC EXAMPLE

CALLBACK

```
function sleep(ms,done){
  setTimeout(()=>{
    done();
  }, ms);
};

function HelloWorld() {
  console.log("hello");
  sleep(1000,()=>{
    console.log("world");
  });
};

HelloWorld();
```

With callbacks, our code would look like this:

Sleep takes two arguments,
the number of milliseconds to sleep
and the callback function to execute when done.

Our HelloWorld function,
prints out "hello",
calls sleep with 1000 and the function to run when done
which after 1000ms executes.
and prints out "world".

BASIC EXAMPLE

CALLBACK

```
function sleep(ms,done){
  setTimeout(()=>{
    done();
  }, ms);
};

function HelloWorld() {
  console.log("hello");
  sleep(1000,()=>{
    console.log("world");
  });
};

HelloWorld();
```

PROMISE

```
function sleep(ms) {
  return new Promise(
    (resolve,reject)=>{
      setTimeout(()=>{
        resolve();
      }, ms);
    });
};

function HelloWorld() {
  console.log("hello");
  sleep(1000).then(()=>{
    console.log("world");
  });
};

HelloWorld();
```

With promises, our code would look like this:

Sleep takes one argument,
the number of milliseconds to sleep,
and returns a promise.

The promise will resolve, after the given number of milliseconds has passed.

Our HelloWorld function,
prints out "hello",
calls sleep with 1000
and tells the returned promise what to execute next.
which after 1000ms executes.
then prints out "world".

BASIC EXAMPLE

CALLBACK

```
function sleep(ms,done){
  setTimeout(()=>{
    done();
  }, ms);
};

function HelloWorld() {
  console.log("hello");
  sleep(1000,()=>{
    console.log("world");
  });
};

HelloWorld();
```

PROMISE

```
function sleep(ms) {
  return new Promise(
    (resolve,reject)=>{
      setTimeout(()=>{
        resolve();
      }, ms);
    });
};

function HelloWorld() {
  console.log("hello");
  sleep(1000).then(()=>{
    console.log("world");
  });
};

HelloWorld();
```

async/await

```
function sleep(ms) {
  return new Promise(
    (resolve,reject)=>{
      setTimeout(()=>{
        resolve();
      }, ms);
    });
};

async function HelloWorld() {
  console.log("hello");
  await sleep(1000);
  console.log("world");
};

HelloWorld();
```

With async/await, our code would look like this:

Sleep is the same as the promise example before.

It takes one argument,

the number of milliseconds to sleep,

and returns a promise.

The promise will resolve, after the given number of milliseconds has passed.

Our HelloWorld function,

prints out "hello",

Calls sleep with 1000,

Sleep returns a promise.

Await pauses the execution until the promise resolves.

sleep resolves,

the await resumes,

and it prints out "world".

Comparing the three...

BASIC EXAMPLE

CALLBACK

```
function sleep(ms,done){
  setTimeout(()=>{
    done();
  }, ms);
};

function HelloWorld() {
  console.log("hello");
  sleep(1000,()=>{
    console.log("world");
  });
};

HelloWorld();
```

PROMISE

```
function sleep(ms) {
  return new Promise(
    (resolve,reject)=>{
      setTimeout(()=>{
        resolve();
      }, ms);
    });
};

function HelloWorld() {
  console.log("hello");
  sleep(1000).then(()=>{
    console.log("world");
  });
};

HelloWorld();
```

async/await

```
function sleep(ms) {
  return new Promise(
    (resolve,reject)=>{
      setTimeout(()=>{
        resolve();
      }, ms);
    });
};

async function HelloWorld() {
  console.log("hello");
  await sleep(1000);
  console.log("world");
};

HelloWorld();
```

We see that the sleep method,
is slightly more complicated in the promise and async/await versions
But the code for HelloWorld is much cleaner in the async/await version.
Also, in the callback and promise versions
our HelloWorld runs internal functions.
None of that is necessary in the async/await version.

Imagine if we wanted to call our sleep function twice in a row.
Or three times. Gets complicated with callbacks
A little better with promises
But with async/await it readily simple and apparent...

<pause>

Next, lets break async/await down a bit more to understand it better.

The async Keyword

```
async function HelloWorld() {  
  console.log("hello");  
  await sleep(1000);  
  console.log("world");  
};
```

First, in order to use `await`, we must first tell JavaScript to expect it.
this is done at the function level
by specifying that the function is an `async` function.

This creates a slightly different type of function than a function definition without the `async` keyword.

You can see this with a quick test in `nodejs`.

The async Keyword

```
async function HelloWorld() {  
  console.log("hello");  
  await sleep(1000);  
  console.log("world");  
};
```

```
[-----] C:\DEV\Awesome\Code> node  
>  
>  
> function f() {};  
undefined  
> f  
[Function: f]  
>  
>  
> async function g() {};  
undefined  
> g  
[AsyncFunction: g]  
>
```

an async function will execute like a standard function until it hits an await statement.

At that point the async function execution will pause until the awaited promise resolves

This pause doesn't block the main javascript thread, So tasks waiting in the execution stack will continue to run while your async function is paused.

Once the promise resolves,
The await completes
And execution of the function continues.

<pause>

Now, once we have an async function, we can use await.

The await Keyword

```
async function HelloWorld() {  
  console.log("hello");  
  await sleep(1000);  
  console.log("world");  
};
```

The await keyword is used to tell an async function that it should wait for the expression that follows to resolve.

The await Keyword


```
await square(4);  
await 123;  
await "concat" + "this" + "string";
```

await must be followed by an expression
as in "await square(4)" or "await some value".

The await Keyword

If this returns a promise,
wait for the promise to
resolve or reject.

```
await square(4);  
await 123;  
await "concat" + "this" + "string";
```



If the following expression results in a promise,
await will pause execution
until the promise resolves or rejects


The await Keyword

await square(4);

await 123;

await "concat" + "this" + "string";

If this or these return anything else, wrap it in a `Promise.resolve()` and wait for it to resolve.



If the evaluated expression does not result in a promise,
a value for example,
Like 123 here,
await will wrap the value in an immediately resolving promise
With `Promise.resolve()`
And wait for that to resolve.

The await Keyword

```
let x = await 123;  
return await "concat" + "this" + "string";  
assert.equal(await square(4),16);
```

Await is an expression in and of itself

And thus can be used anywhere an expression is used

Including assignments

Or returns

Or as arguments to other functions,

As shown here.

So that's the basics of async/await...

But it wouldn't be JavaScript if there weren't a few side effects to know...

await on a promise

```
function multiply(x,y) {  
  return new Promise((resolve)=>{  
    setTimeout(()=>{  
      resolve(x*y);  
    },1000);  
  });  
}  
  
async function HelloWorld() {  
  console.log("hello");  
  console.log("82 times 47 equals "+(await multiply(82,47))+".");  
  console.log("world");  
}  
HelloWorld();
```

Generally if you await on a called function,
that function should return a promise.
Otherwise, the await may not pause as expected.
So get in the habit of only awaiting on things that return a promise.

Also, await doesn't really save you from writing Promises,
They are still very, very much a part of async/await.

But it does save you from writing then() methods.

Catching a Rejection

```
function throwSomething(x,y) {  
  return new Promise((resolve,reject)=>{  
    reject(new Error("something error."));  
  });  
}  
  
async function HelloWorld() {  
  try {  
    await throwSomething();  
  }  
  catch (ex) {  
    console.error(ex);  
  }  
}
```

Also, one of the awesome features of async/await is that promise rejections are just thrown as exceptions.

This means you can use a standard try/catch block to handle the rejection you get back from your promise!

It is worth pointing out you should always catch your await exceptions or you end up with the dreaded UnhandledRejection exception

If you have used promises a lot, you are all too familiar with this exception;

But, the good news is these are far easier to read and to trace, with async/await then just straight promises.

Iterating and Awaiting

```
function square(x) {  
  return x*x;  
};  
function map(a) {  
  return a.map((x)=>{  
    return square(x);  
  });  
}  
map([0,1,2,3,4,5,6,7,8,9]);  
// returns [ 0, 1, 4, 9, 16, 25, 36, 49, 64, 81 ]
```

Also, a special case comes up with async/await and iteration.

By and large we no longer really use for loops
for iterating over arrays anymore.

Instead we use Array methods

like `forEach()` and `map()` and `filter()` which take an iteration method

Like I've shown here.

However, if the iteration method needs to await ,
things don't quite work as expected.

Iterating and Awaiting

```
function square(x) {  
  return new Promise((resolve,reject)=>{  
    setTimeout(()=>{resolve(x*x);},250);  
  });  
}  
function map(a) {  
  return a.map(async(x,i)=>{  
    return await square(x);  
  });  
}  
map([0,1,2,3,4,5,6,7,8,9]);  
// returns [ Promise { <pending> },Promise { <pending> },Promise {  
<pending> },Promise { <pending> },Promise { <pending> },Promise {
```

Here we see our square function returns a promise
And takes 250ms to square a number
For whatever reason.

Calling `a.map()` and awaiting inside of it doesn't work like we expect.
The map function will actually return
BEFORE the await functions have resolved.
Thus giving us an array of unresolved promises.

Instead, you have to do it like this...

Iterating and Awaiting

```
function square(x) {  
  return new Promise((resolve,reject)=>{  
    setTimeout(()=>{resolve(x*x);},250);  
  });  
}  
async function map(a) {  
  return await Promise.all(a.map((x,i)=>{  
    return square(x);  
  }));  
}  
map([0,1,2,3,4,5,6,7,8,9]);  
// returns [ 0, 1, 4, 9, 16, 25, 36, 49, 64, 81 ]
```

Square() is the same, but we have changed our map function.

Instead of awaiting inside of the iterator,
For each iteration of our array
we return the unresolved promise.
The resulting array of promises,
is then passed to Promise.all(),
Which returns a promise that resolves
only when all the promises in the passed array
Have resolved.

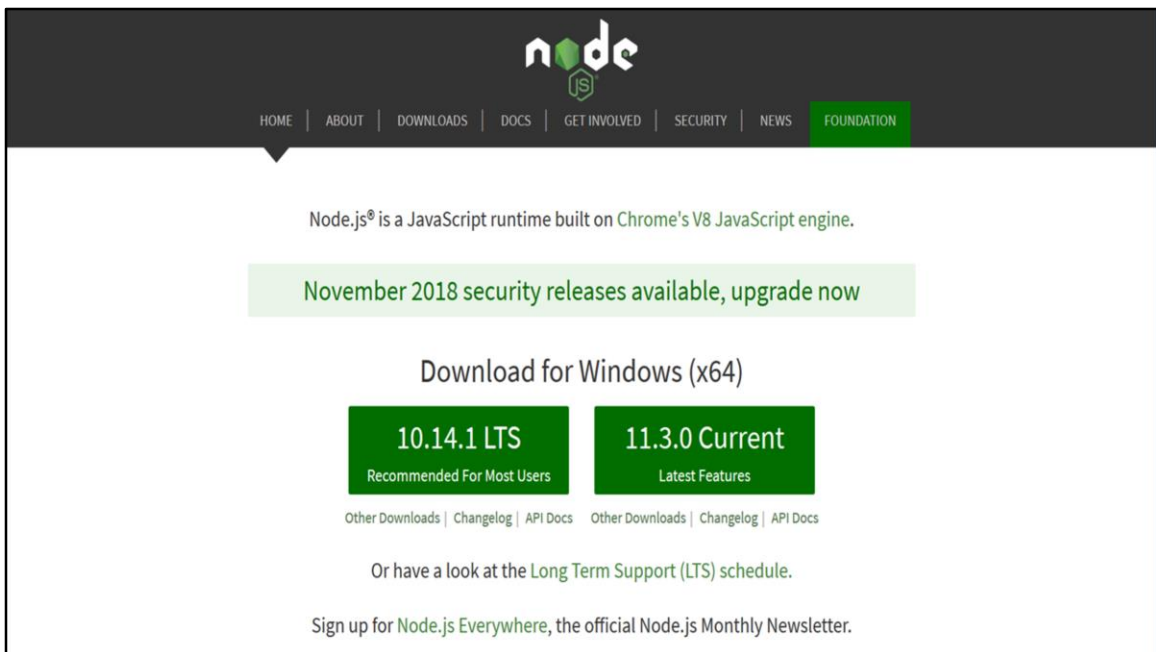
If we await on the Promise.all() promise,
Then we achieve the desired result.
<PAUSE>
So that's how you use async/await.

async/await

But before we all go rushing out for beer
Which you absolutely should do
because half of the reason to come to these meetups
is to meet other people...

A few last things.

First, async/await is not without its drawbacks.



Almost nothing in nodejs core libraries uses promises at this point.

However, they are working on it.

Node.js
About these Docs
Usage & Example
Assertion Testing
Async Hooks
Buffer
C++ Addons
C/C++ Addons - N-API
Child Processes
Cluster
Command Line Options
Console
Crypto
Debugger
Deprecated APIs
DTrace

fs Promises API

Stability: 1 - Experimental

The `fs.promises` API provides an alternative set of asynchronous file system methods that return `Promise` objects rather than using callbacks. The API is accessible via `require('fs').promises`.

class: FileHandle

Added in: v10.0.0

A `FileHandle` object is a wrapper for a numeric file descriptor. Instances of `FileHandle` are distinct from numeric file descriptors in that, if the `FileHandle` is not explicitly closed using the `filehandle.close()` method, they will automatically close the file descriptor and will emit a process warning, thereby helping to prevent memory leaks.

Instances of the `FileHandle` object are created internally by the `fsPromises.open()` method.

Unlike the callback-based API (`fs.fstat()`, `fs.fchown()`, `fs.fchmod()`, and so on), a numeric file descriptor is not used by the promise-based API. Instead, the promise-based API uses the `FileHandle` class in order to help avoid accidental leaking of unclosed file descriptors after a `Promise` is resolved or rejected.

filehandle.appendFile(data, options)

Added in: v10.0.0

Recently nodejs v10 added the `fs.promises` api.
 which give us promise versions of all the `FileSystem` calls.
 And hopefully, more will follow.

Additionally nodejs offers us `util.promisify()`
 to convert nodejs style callbacks

Net

OS

Path

Performance Hooks

Process

Punycode

Query Strings

Readline

REPL

Stream

String Decoder

Timers

TLS/SSL

Trace Events

TTY

UDP/Datagram

URL

Utilities

V8

VM

Worker Threads

ZLIB

util.promisify(original)

Added in: v8.0.0

- **original** <Function>
- **Returns:** <Function>

Takes a function following the common error-first callback style, i.e. taking an `(err, value) => ...` callback as the last argument, and returns a version that returns promises.

```
const util = require('util');
const fs = require('fs');

const stat = util.promisify(fs.stat);

stat('.').then((stats) => {
  // Do something with `stats`
}).catch((error) => {
  // Handle the error.
});
```

Or, equivalently using **async functions**:

```
const util = require('util');
const fs = require('fs');

const stat = util.promisify(fs.stat);

async function callStat() {
  const stats = await stat('.');
  console.log(`This directory is owned by ${stats.uid}`);
}
```

Into promise version of the same call.

but you need to be careful with using `util.promisify`,
Especially around the number of arguments.

In many cases nodejs allows for the same call
to be used with different arguments,
which can make `util.promisify()` act a little wierd.

Secondly, when we were previously talking about the `async` keyword
I mentioned that in order to use `await`, it must be inside an `async` function.
This means that you cannot use the `await` keyword in your top-level code.

ECMAScript proposal: Top-level `await`

Status

This proposal is currently in stage 2 of [the TC39 process](#).

Background

The `async / await` proposal was originally brought to committee in [January of 2014](#). In [April of 2014](#) it was discussed that the keyword `await` should be reserved in the module goal for the purpose of top-level `await`. In [July of 2015](#) the `async / await` proposal advanced to Stage 2. During this meeting it was decided to punt on top-level `await` to not block the current proposal as top-level `await` would need to be "designed in concert with the loader".

Since the decision to delay standardizing top-level `await` it has come up in a handful of committee discussions, primarily to ensure that it would remain possible in the language.

Motivation

top-level main and IIFEs

The current implementation of `async / await` only support the `await` keyword inside of `async` functions. As such many programs that are utilizing `await` now make use of top-level main function:

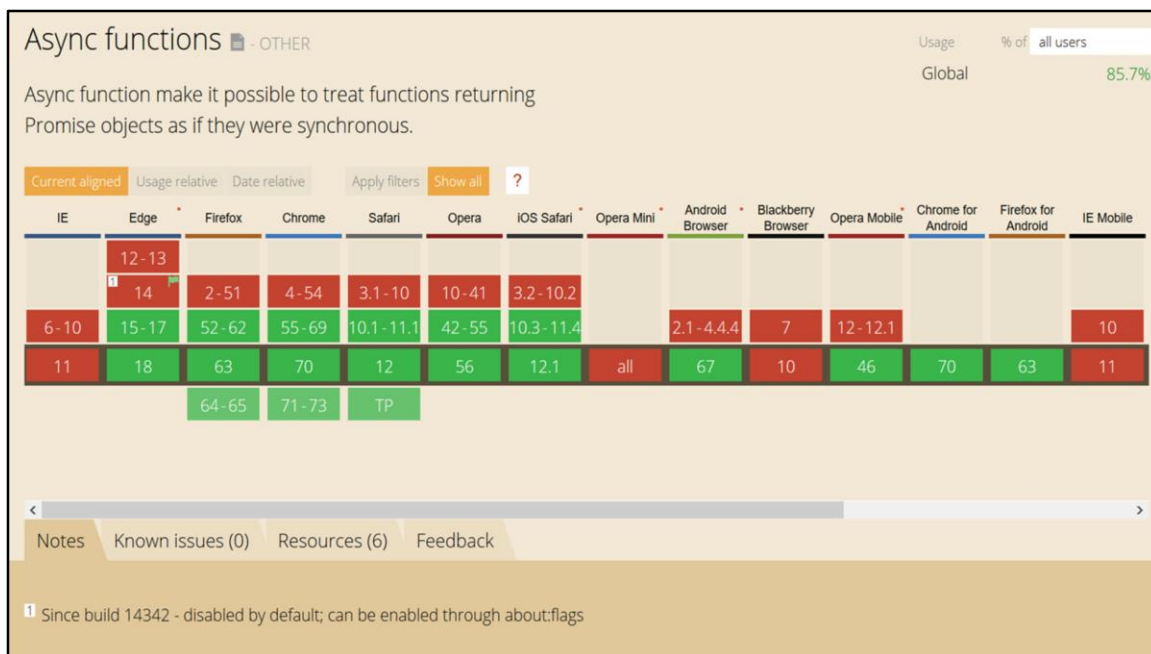
```
import ...  
async function main() {
```

However, there is currently a proposal for supporting top-level `await` for ES2019. and there are work arounds such as using an IIFE or the like.

<PAUSE>

All right that's it.
That's why you should be using `async/await` in your code.
It's super simple,
It makes your code more readable,
And it frees us from callback hell and promise purgatory.

So what are you waiting for?
`Async/await` is supported by all the modern browsers



as well as nodejs v8 or later.

There are some edge cases but hopefully internet explorer will die a horrible death soon and we can finally move on.

And seriously, is blackberry really still a thing?

Why are you “Awaiting” to use Async!

So

stop “Awaiting”
and resolve your feelings
for using async/await...
before the rest of the world
rejects you.
I promise you won't regret it.
If you do you can call me back.

Thanks you for listening.
<NEXT SLIDE>



Glen R. Goodwin



@areinet



arei.net



github.com/arei

Awesome Engineering

github.com/awesomeeng

npmjs.com/org/awesomeeng