Threading in JavaScript

The event loop, deferred objects, the promise interface, and other confusing things

tl;dr

- JavaScript is single-threaded but it does allow us to do some async activities if we know how
- It uses the event loop to do this
- We can create promises which run on a separate thread and call a callback function when finished
- Async and await are simpler (and less capable) ways of creating promises and registering a callback

JavaScript is single-threaded

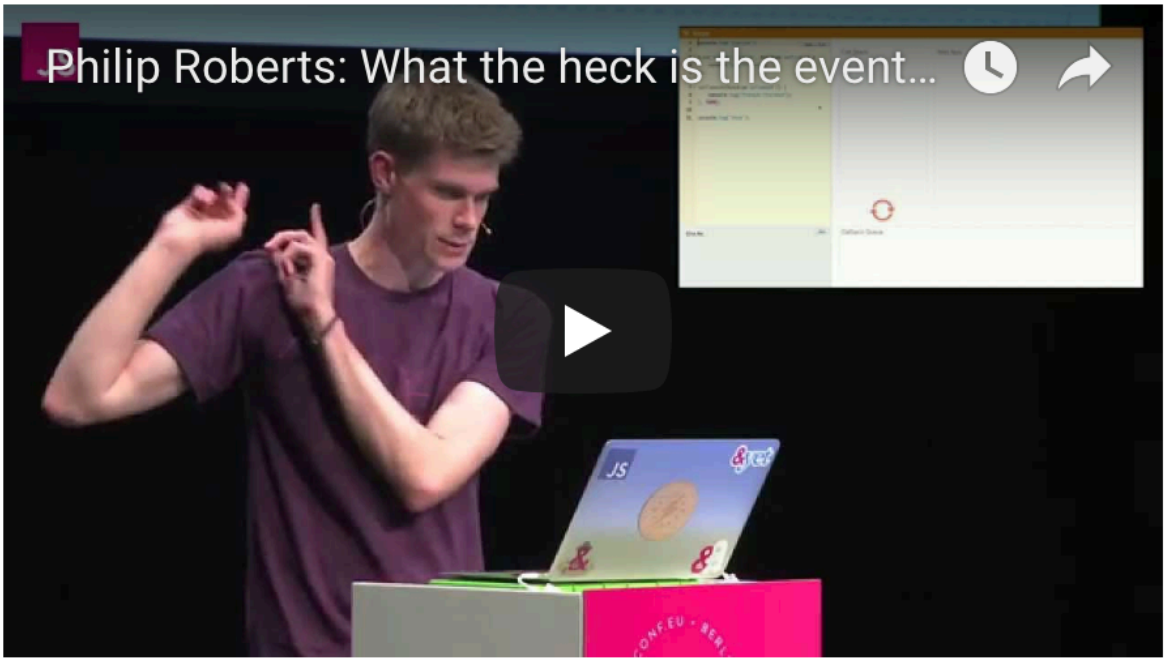
- JavaScript is a synchronous language
- Meaning: one line must complete before it moves on to the next line.
- So how does it do multithreading?

The browser/node is multithreaded

- JavaScript *borrows* the capability of the browser

The Event Loop

- Q: So how does JavaScript make use of that multithreading capability?
- A: It uses the event loop!



The promise interface

A promise represents the future return of data

I promise I'll give you something when I'm finished.

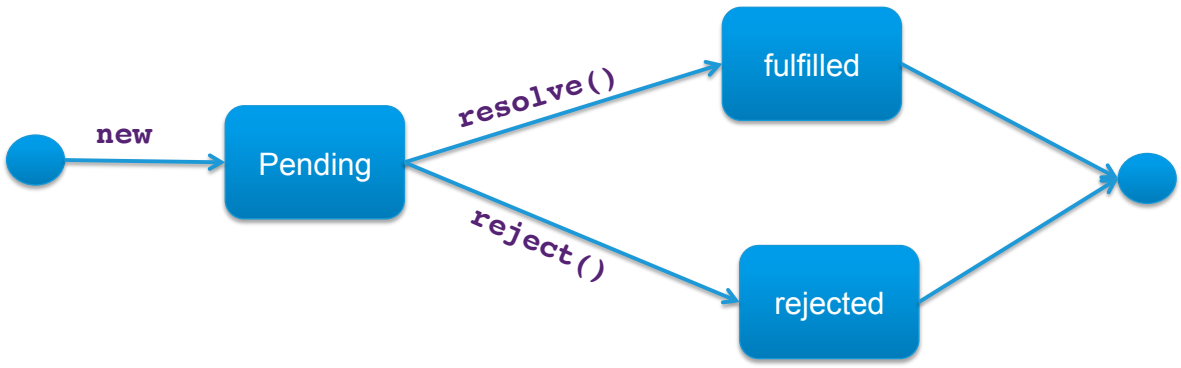
Promises can *not* make a sync activity async

- It only helps manage async activities via callbacks.

It's like being on hold when you phone tech support

- You're actively waiting/listening until tech support is freed up
- Wouldn't it be nice if you could register your phone number and they call YOU back when they're ready?
- Wouldn't you call that a "callback"?

A promise exists in one of three states



Creating a promise

A promise may look like this:

```
const p = new Promise((resolve, reject) => {
  const exitStatus = LongRunningProcess();
  if (exitStatus === "good")
    resolve();
  else
    reject();
});
```

Functions to be provided later. Arbitrary for now

Any process we want to run.

Any condition check. If it is successful, we call the resolve function passed in. If not, we call the reject function.

resolve and reject allow you to pass some data out of the async function

- Instead of this:

```
resolve()
```

- do this:

```
resolve(someData)
```

- Instead of this:

```
reject()
```

- do this:

```
reject(anErrorObject)
```

When the promise is fulfilled, you can read the data

```
function asyncFunc() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (Math.random() > 0.5)
        resolve('DONE');
      else
        reject('FAIL');
    }, 3000);
  });
}

asyncFunc()
```

For example

resolve()/reject() calls do NOT end the function

Do this ...

```
doStuff();
if (error)
  reject(error);
else {
  doMoreStuff();
  resolve(data)
}
```

,,, NOT this ...

```
doStuff();
if (error)
 reject(error);
doMoreStuff();
resolve();
```

Registering callbacks

- The resolve and reject are simply input parameters to the promise function -- they are holders of values
- The promise expects them to be of type *function*
- They are "callbacks"
- Those values will be supplied when it is time to run the async function
- We do that with .then()

Promises have a .then() method to registers callbacks

```
let p = asyncFunction();
p.then(success, error);
```

What to do when the asyncFunction finishes successfully

What to do if it fails

```
function asyncFunc() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (Math.random() > 0.5)
        resolve('DONE');
      else
        reject('FAIL');
    }, 3000);
  });
}
```

For example

```
asyncFunc()
  .then(
    x => console.log(`Result: ${x}`),
    e => console.error(`Error: ${e}`)
  );
```

There's also the .catch and .finally methods

```
asyncFunc()
  .then(successHandler)
  .catch(failureHandler)
  .finally(finallyHandler);
```

Called when the promise is resolved

Called when it is rejected

Called at the end whether it is resolved or rejected

Async and await

async and await do not add any new capabilities

- Merely a more abstract way to handle promises coming back from functions
- Less typing and more declarative

Some ground rules ...

- What is marked as async? Only function definitions
- What is marked as await? Only function calls

Here's a function without async

```
function foo() {
  const x = sub1();
  const y = sub2();
  return x + y;
}

console.log(foo());  // Runs foo. Logs x + y

foo().then( val => console.log(val));
// Fails - "then is not a function"
```

An async function always returns a promise

```
async function foo() {
  const x = sub1();
  const y = sub2();
  return x + y;
}

console.log(foo());
// Runs foo. Logs [object Promise]

foo().then( val => console.log(val));
// Logs x + y
```

- When you mark a call with await, the JavaScript engine looks to see if the call returns a promise. If so, it awaits until that promise is resolved, then returns the VALUE coming back as part of the resolution.

- It turns this ...

```
let x;
someAsyncFunction().then(val => x=val);
```

- Into this ...

```
const x = await someAsyncFunction();
```

- You can technically await any function ... even one that doesn't return a promise. But it is meaningless.

```
const status = await doItNow();
console.log(status);  //Immediately logs "done!"

function doItNow() {
  return "done!"
}
```

What about errors?

- await throws hard if the promise rejects.

```
x = await someAsyncFunction();
y = `We'll never get here if the
       promise above rejects.`;
```

- So a try/catch works:

```
try {
  x = await someAsyncFunction();
} catch(err) {
  console.error(err);
}
y = "Now we get here either way";
```

They *can* be used together

```
async function theFunction() {
  doSomethingSync();
  return somePromiseFunc();
}

const x = await theFunction();
```

(although they don't have to be)

tl;dr

- JavaScript is single-threaded but it does allow us to do some async activities if we know how
- It uses the event loop to do this
- We can create promises which run on a separate thread and call a callback function when finished
- Async and await are simpler (and less capable) ways of creating promises and registering a callback