Async JavaScript

JavaScript is a single-threaded language which means it has a single call stack. This means that JavaScript in the browser **can only do one thing at at time.**

As a result, if we have a stack of operations to perform and one of them is really slow, everything else has to wait until the slow operation completes. This is called **blocking**.

So when we get a network request, we have two options:

1. Do the network request right away, and block everything else.

2. Queue the network request, and run it when we have some spare time.

This used to be done with callbacks

```
fetchFromSomeApi(function(data) {
  // now we have the data we need
})
```

but say you had to make 3 requests:

```
fetchFromSomeApi(function(data1) {
  fetchFromSomeOtherApi(function(data2) {
    fetchFromSomeOtherOtherApi(function(data3)
      // well this is confusing.
      // and what if one of these goes wrong?
    })
  })
})
```

Callback Hell

Not Secure callbackhell.com

Callback Hell

A guide to writing asynchronous JavaScript programs

What is "*callback hell*"?

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

```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }.bind(this))
        }
      })
    })
  }
})
```

See the pyramid shape and all the `})` at the end? Eek! This is affectionately known as **callback hell**.

The cause of callback hell is when people try to write JavaScript in a way where execution happens visually from top to bottom. Lots of people make this mistake! In other languages like C, Ruby or Python there is the expectation that whatever happens on line 1 will finish before the code on line 2 starts running and so on down the file. As you will learn, JavaScript is different.

What are callbacks?

Callbacks are just the name of a convention for using JavaScript functions. There isn't a special thing

So we moved on, and in ES2015, we gained **Promises.**

A promise is a box that may or may not have a value inside.

Hey, what’s in the box?

???

Hey, can you let me know when something is in the box?

Promises can *resolve* or *reject.*

A resolved promise can be said to have *fulfilled.*

A rejected promise is said to have been *rejected.*

A promise can also be *pending*, where we don't know the state. For example, we made a network request but haven't had a response back.

Promise.resolve can create a promise that resolves with a value.

5

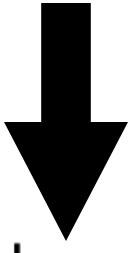
```
Promise.resolve(5)
```

Promise chains.

These can be thought of like a pipeline.

```
Promise.resolve(5).then(value => value + 1)
```

this is how we give a function that will be called with the value inside the box.



```
Promise.resolve(5).then(value => value + 1)
```

```
Promise.resolve(5)
    .then(value => {
        return value + 1;
    }).then(value => {
        console.log(value)
    })
```

promise functions can return the next value

Let's play with promises.

open the console!

```
npm run exercise async 1
```

Wrapping callbacks with promises

```
setTimeout(() => {
  console.log('I will run after 5 seconds!')
}, 5000)
```

if a function uses callbacks, we can wrap it in a promise

new Promise()

when we create a new promise, we get a function that we can call when we want the promise to resolve

```
const timeoutPromise = new Promise(function(resolve) {
  setTimeout(() => resolve(), 5000)
})

timeoutPromise.then(() => {
  console.log('I will run after 5 seconds')
})
```

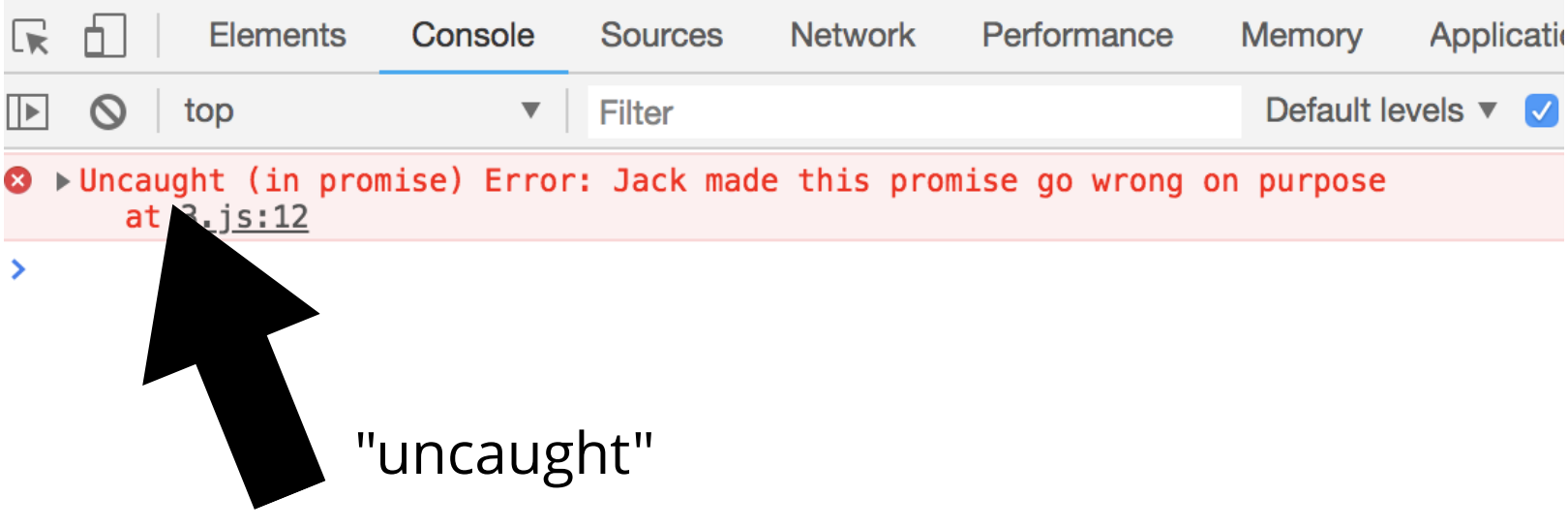
```
const timeoutPromise = new Promise(function(resolve) {
  setTimeout(() => resolve(), 5000)
})

timeoutPromise.then(() => {
  console.log('I will run after 5 seconds')
})
```

```
npm run exercise async 2
```

when promises go wrong

```
Promise.resolve(5).then(value => {
  throw new Error('Jack made this promise go wrong on purpose')
})
```



```
Promise.resolve(5).then(value => {
  throw new Error('Jack made this promise go wrong on purpose')
}).catch(e => {
  console.log('We caught the error!')
})
```

error handling

```
npm run exercise async 3
```

.then/.catch in promise chains

```
Promise.resolve(5)

  .then(value => return value + 1)

   .then(value => {
     throw new Error('whoops!')
   })

   .catch(error => {
     console.log('error')
     return 10
   })

   .then(value => {
     console.log('got value', value)
   })
```

```
Promise.resolve(5)

  .then(value => return value + 1)

   .then(value => {
     throw new Error('whoops!')
   })

   .catch(error => {
     console.log('error')
     return 10
   })

   .then(value => {
     console.log('got value', value)
   })
```

we catch the error and deal with it and return a new value

```
Promise.resolve(5)

  .then(value => return value + 1)

  .then(value => {
    throw new Error('whoops!')
  })

  .then(value => {
    console.log('hello world!')
  })

  .catch(error => {
    console.log('error')
    return 10
  })

  .then(value => {
    console.log('got value', value)
  })
```

does this get run?

```
Promise.resolve(5)

  .then(value => return value + 1)

  .then(value => {
    throw new Error('whoops!')
  })


  .then(value => {
    console.log('hello world!')
  })

  .catch(error => {
    console.log('error')
    return 10
  })

  .then(value => {
    console.log('got value', value)
  })
```

```
Promise.resolve(5)

  .then(value => return value + 1)

  .then(value => {
    throw new Error('whoops!')
  })


  .then(value => {
    console.log('hello world!')
  })

  .catch(error => {
    console.log('error')
    return 10
  })

  .then(value => {
    console.log('got value', value)
  })
```

```
Promise.resolve(5)

  .then(value => return value + 1)

   .then(value => {
     throw new Error('whoops!')
   })


   .then(value => {
     console.log('hello world!')
   })

   .catch(error => {
     console.log('error')
     return 10
   })

   .then(value => {
     console.log('got value', value)
   })
```

```
Promise.resolve(5)

  .then(value => return value + 1)

   .then(value => {
     throw new Error('whoops!')
   })


   .then(value => {
     console.log('hello world!')
   })

   .catch(error => {
     console.log('error')
     return 10
   })

   .then(value => {
     console.log('got value', value)
   })
```

```
Promise.resolve(5)

  .then(value => return value + 1)

   .then(value => {
     throw new Error('whoops!')
   })


   .then(value => {
     console.log('hello world!')
   })

   .catch(error => {
     console.log('error')
     return 10
   })

   .then(value => {
     console.log('got value', value)
   })
```

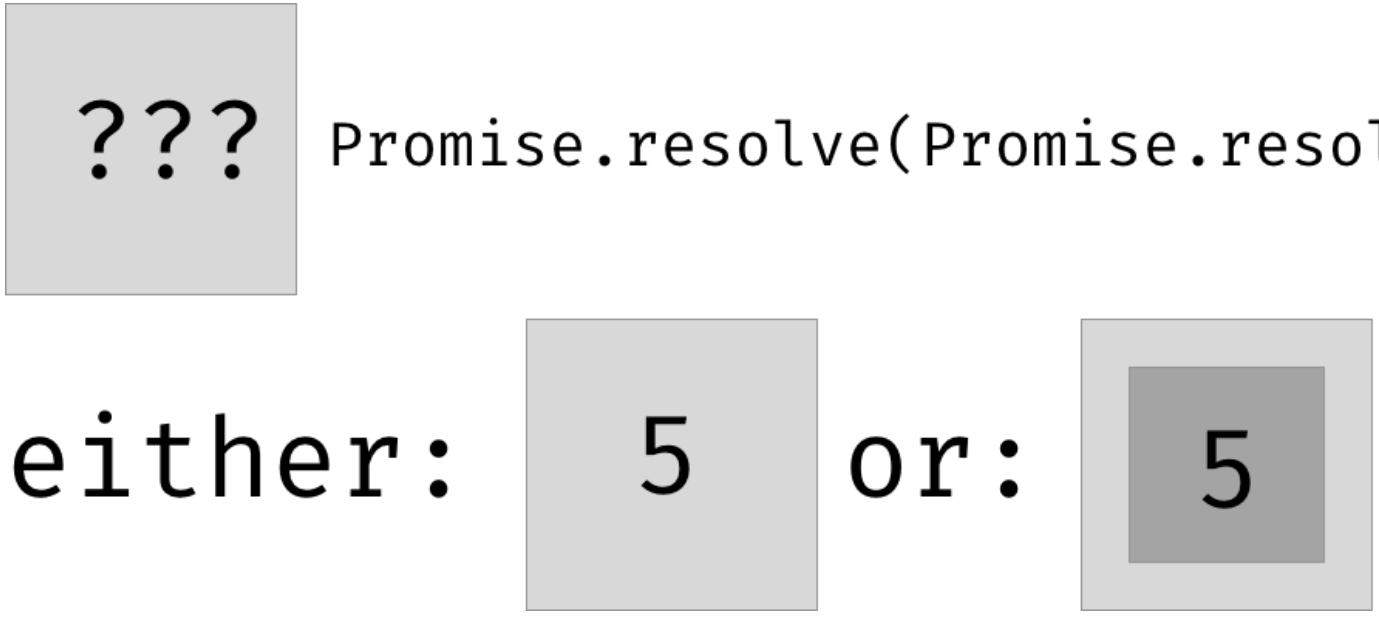
promise chains with error handling

```
npm run exercise async 4
```

promises returning promises

5

```
Promise.resolve(5)
```


???
Promise.resolve(Promise.resolve(5))
either:
5
or:
5

5

```
myPromise.then(promise => {
  promise.then(value => {
  })
})
```

you can never have nested promises.

```
npm run exercise async 5
```

the fetch API

```
fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then(response => {
    return response.json()
  })
  .then(todo => {
    console.log('got todo')
  })
```

```
fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then(response => {
    return response.json()
  })
  .then(todo => {
    console.log('got todo')
  })
```

returns a promise that resolves with the response, parsed to JSON.

wrapping the fetch API promise in a function

```
npm run exercise async 6
```

sequential promises

Secure https://jsonplaceholder.typicode.com/photos/1


```
// 20180814183256
// https://jsonplaceholder.typicode.com/photos/1

{
  "albumId": 1,
  "id": 1,
  "title": "accusamus beatae ad facilis cum similique qui sunt",
  "url": "http://placehold.it/600/92c952",
  "thumbnailUrl": "http://placehold.it/150/92c952"
}
```

Secure https://jsonplaceholder.typicode.com/photos/1

```
// 20180814183256
// https://jsonplaceholder.typicode.com/photos/1

{
  "albumId": 1,
  "id": 1,
  "title": "accusamus beatae ad facilis cum similique qui sunt",
  "url": "http://placehold.it/600/92c952",
  "thumbnailUrl": "http://placehold.it/150/92c952"
}
```

Secure https://jsonplaceholder.typicode.com/albums/1

```
// 20180814212654
// https://jsonplaceholder.typicode.com/albums/1

{
  "userId": 1,
  "id": 1,
  "title": "quidem molestiae enim"
}
```

fetch a photo, and then fetch its album

```
npm run exercise async 7
```

parallel requests

fetch recent photos
and recent albums

Promise.all

```
Promise.all([
  Promise.resolve(5),
  Promise.resolve(6)
]).then(values => {
  console.log(values) // [5, 6]
})
```

```
npm run exercise async 8
```

async / await

Part of ES2017

async / await is syntactic sugar over promises

when you write
async/await, you are
using promises

```
fetch('/posts').then(response => response.json()).then(posts => {...})

const posts = await fetch('/posts').then(response => response.json())
```

you can only use
await in a function
that's marked with
async

the async keyword denotes a function that returns a promise

```
const fetchPhoto = async id => {
  const response = await fetch(
    `https://jsonplaceholder.typicode.com/photos/${id}`
  )
  return response.json()
}
```

we don't need await here because the return value is automatically wrapped in a promise, because our function is async

```
const fetchPhoto = async id => {
  const response = await fetch(
    `https://jsonplaceholder.typicode.com/photos/${id}`
  )
  return response.json()
}
```

```
npm run exercise async 9
```

async/await can simplify promise chains

```
Promise.resolve(5)
  .then(value => value + 1)
  .then(value => value + 2)
  .then(value => value + 3)
  .then(value => {
    logPromiseValue(1, value)
  })

const asyncVersion = async () => {
  const firstValue = await Promise.resolve(5)
  logPromiseValue(2, firstValue + 1 + 2 + 3)
}
```

simplify the promise chain using async/await

```
npm run exercise async 10
```

straight into another one! fetch a photo and its album using async await

```
npm run exercise async 11
```

error handling with async await

we simply use
try/catch

```
try {
  const response = await fetch('/photos')
} catch (e) {
  // got an error!
}
```

```
try {
  const response = await fetch('/photos')
  const nextThing1 = await fetch(...)
  const nextThing2 = await fetch(...)
  const nextThing3 = await fetch(...)
  const nextThing4 = await fetch(...)

} catch (e) {
  // this will catch all errors from above.
}
```

catch the error!

```
npm run exercise async 12
```

async/await is sequential!

```
const photos = await fetchAllPhotos()
```

```
const albums = await fetchAllAlbums()
```

these are not run in parallel!

we will wait for photos before fetching albums

because async/await uses promises, we can use async with promises

```
const photosAndAlbums = await Promise.all([
  fetchPhotos(),
  fetchAlbums()
])
```

this will run in parallel as we expect

fix up the slowness

```
npm run exercise async 13
```

a...sync we are done here!

setting up with Babel

https://stackoverflow.com/a/28709165

TLDR:
babel-plugin-transform-runtime

Async functions - OTHER

Usage Global | % of all users 84.01%

Async function make it possible to treat functions returning Promise objects as if they were synchronous.

Current aligned | Usage relative | Date relative | Show all

IE	Edge *	Firefox	Chrome	Safari	iOS Safari *	Opera Mini *	Chrome for Android	UC Browser for Android	Samsung Internet
			49						
			63						
			66		10.3				
			67		11.2				4
11	17	61	68	11.1	11.4	all	67	11.8	7.2
	18	62	69	12	12				
		63	70	TP					
			71						

Notes | Known issues (0) | Resources (6) | Feedback