Promises

CSC336 – Web Technologies

Using callback functions

then

```
// In the browser
setTimeout(function () {
  // Will be called in the future
}, 2000);


// In the server
fs.readFile('file.txt', function () {
  // Will be called when file.txt is read
});
```

Node.js callback standard

```
fs.readFile('file.txt', function (err, data) {
  // If an error occurred, err will have a value
  // Always check for errors using if clauses
})
```

Node.js callback scenario

then

- Let's say we have a `fetch` function
- It does plain HTTP GET
- Accepts a URL and a callback
- Callback receives error and response

```
fetch ('url', function (err, res) { })
```

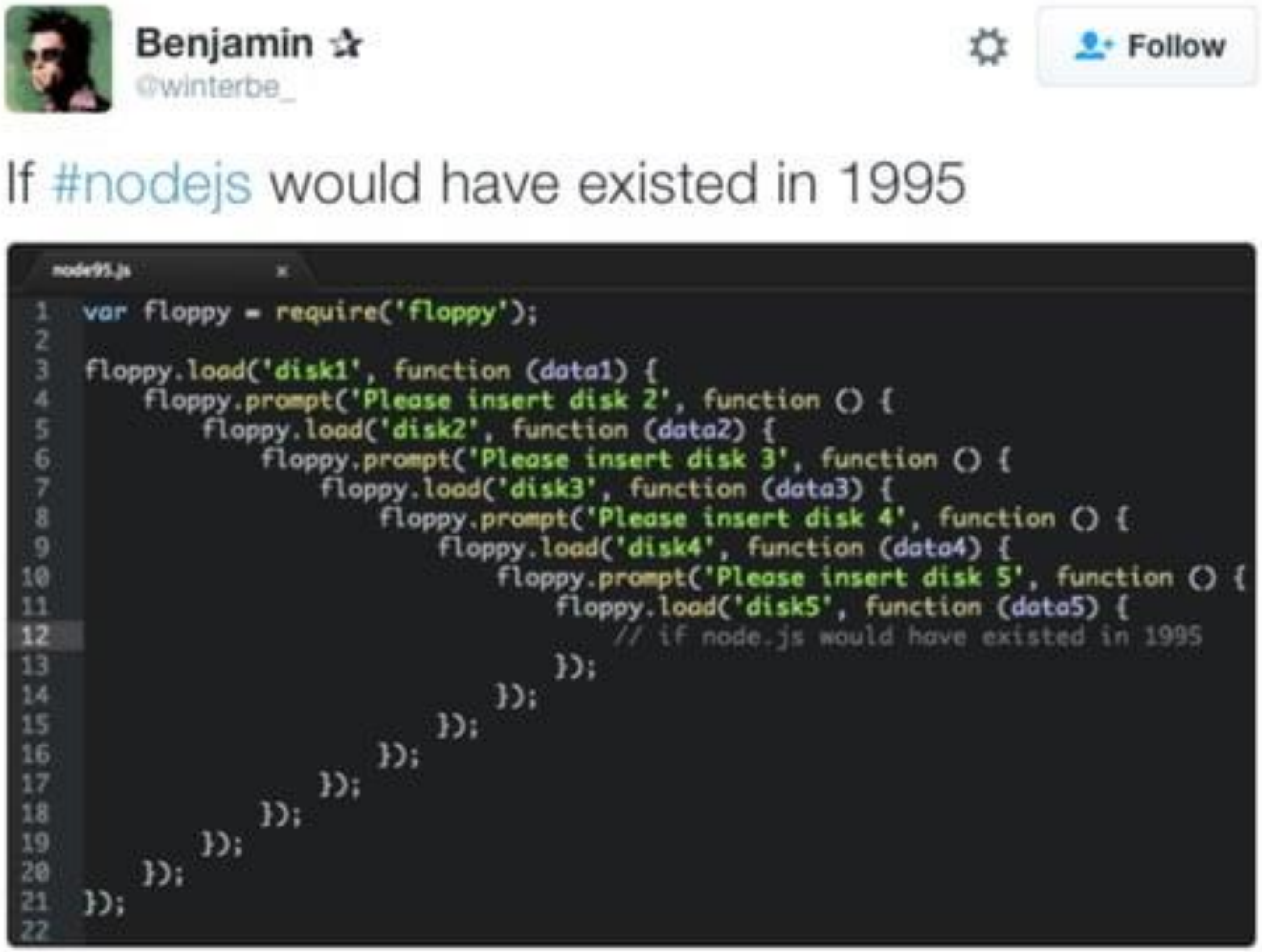
Node.js callback scenario

then

```
fetch('/users/session', function (sessionError, user) {
  if (sessionError) {
    alert('Error fetching session')
    return
  }
  fetch('/users/' + user.id + '/posts', function (userErr, posts) {
    if (userErr) {
      alert('Error fetching user posts')
      return
    }
    renderUserPosts(posts)
  })
})
```

Node.js callback hell

then



If #nodejs would have existed in 1995

```
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) {
                                    // if node.js would have existed in 1995
                                });
                            });
                        });
                    });
                });
            });
        });
    });
});
```

then

How could we flatten that tree?

```
new Promise()
```

Formal definition

then

"A promise represents the eventual result of an asynchronous operation."

Formal definition

then

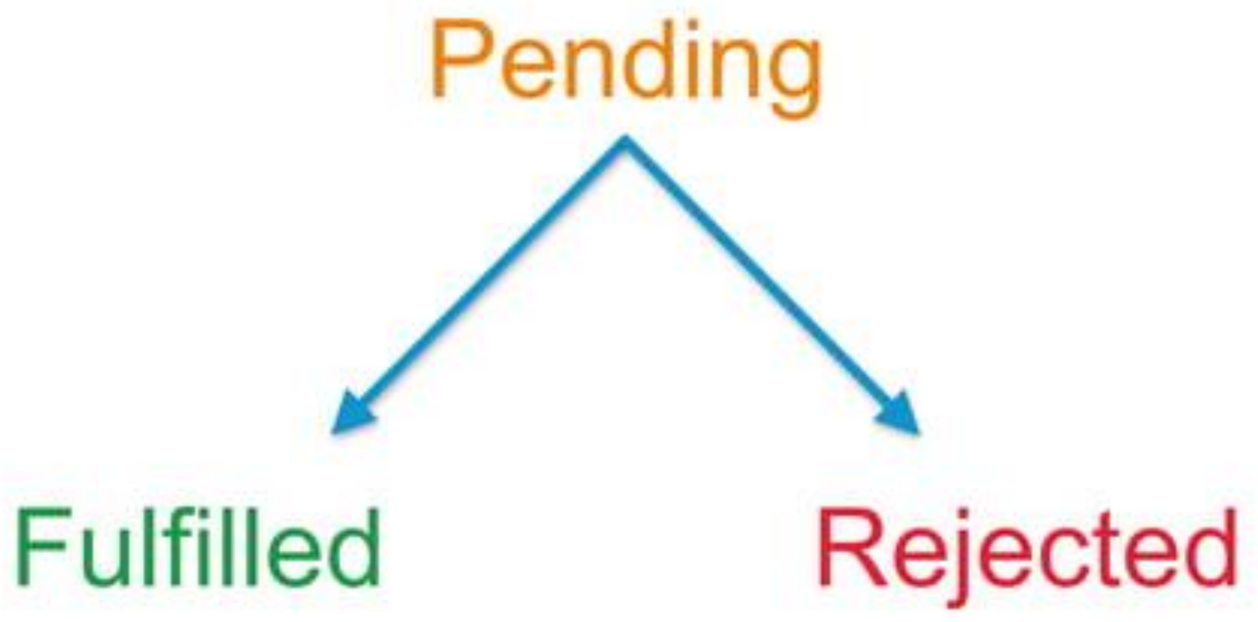
"A promise represents the **eventual result** of an asynchronous operation."

Enter Promises

then

An object which represents and manage the lifecycle of a future result

Promise states



Promise states

$$\begin{tikzcd}
& \text{Pending} & \\
\text{Fulfilled} \arrow[ur] & & \text{Rejected} \arrow[ul]
\end{tikzcd}$$

Promise API

then

```
// New Promises start in "Pending" state
new Promise(function (resolve, reject) {

  // Transition to "Rejected" state
  reject(new Error('A meaningful error'))

  // Transition to "Fulfilled" state
  resolve({ my: 'data' })

})
```

Promise API

then

```
var promise = new Promise(...)

promise.then(function (result) {
  console.log(result)
})

=> { my: "data" }
```

Promise API

then

```
var promise = new Promise(...)

promise.catch(function (error) {
  console.log(error.message)
})

=> A meaningful error
```

Promise API

Node.js callbacks can be easily wrapped in promises

Promise API

then

```
function fetchAsync (url) {
  return new Promise(function (resolve, reject) {
    fetch(url, function (err, data) {
      if (err) {
        reject(err)
      } else {
        resolve(data)
      }
  })
}
```

Promise API

Every `.then` and `.catch` returns a new promise, so promises are chainable

Flattening the tree

then

```
function fetchPosts (user) {
  return fetch('/users/' + user.id + '/posts')
}

function fetchSession () {
  return fetch('/users/session')
}

fetchSession()
  .catch(handleSessionError)
  .then(fetchPosts)
  .catch(handlePostsError)
  .then(renderUserPosts)
```

Flattening the tree

then

Chaining allows flattening the callback hell and make continuation passing look sequential

Chaining (a.k.a. sequence monad)

then

```
const makeObject   = e => ({ l: e[0], r: e[1] })
const attachPlus   = e => merge(e, { plus: e.l + e.r })
const attachMinus  = e => merge(e, { minus: e.l - e.r })
const attachTimes  = e => merge(e, { times: e.l * e.r })
const attachDivide = e => merge(e, { divide: e.l / e.r })

fetchTuples()
  .then(makeObject)
  .then(attachPlus)
  .then(attachMinus)
  .then(attachTimes)
  .then(attachDivide)
  .then(console.log.bind(console))
```

Promise Libraries

There are a handful of Promise implementations

Solving different issues, focusing on different areas

Promise Libraries

So I have to be tied to a single implementation?

Promise Libraries

Promises/A+ Contract

then

https://promisesaplus.com

Promises/A+ Contract

then

Promises/A+ provides interface and behaviour specification for interoperable promises

Promises/A+ Contract

So you are free to use the implementation which better fit your needs while keeping your code compatible

Promises/A+ Contract

then

This contract was created because there was no native Promise specification in ECMAScript

ECMAScript 2015 Promise

Since ECMAScript 2015 the Promise object was included in the spec

https://tc39.github.io/ecma262/#sec-promise-constructor

ECMAScript 2015 Promise

then

It allows more fun stuff do be done

ECMAScript 2015 Promise

then

Waiting for multiple Promises

Waiting for multiple Promises

then

```
var promises = [
  new Promise(function (resolve, reject) {
    setTimeout(resolve, 1000);
  }),
  new Promise(function (resolve, reject) {
    setTimeout(resolve, 2000);
  })
]

Promise.all(promises).then(function () {
  console.log('Ran after 2 seconds')
})
```

ECMAScript 2015 Promise

then

Racing multiple Promises

Racing multiple Promises

```
var promises = [
  new Promise(function (resolve, reject) {
    setTimeout(resolve, 1000);
  }),
  new Promise(function (resolve, reject) {
    setTimeout(resolve, 2000);
  })
]

Promise.race(promises).then(function () {
  console.log('Ran after 2 seconds')
})
```

then

You should definitely look into Promises

Bluebird

A complete Promise library

http://bluebirdjs.com

HTML Fetch

A Promise approach to HTTP requests

https://fetch.spec.whatwg.org

Demo

Fetching stuff from Github

https://github.com/derekstavis/promises-on-the-browser

Thanks for watching

Questions?

github.com/derekstavis

twitter.com/derekstavis

facebook.com/derekstavis

then

Bluebird

A complete Promise library

http://bluebirdjs.com

Bluebird Promise

then

Catch rejections like exceptions

Fine-grained exceptions

then

```
function SessionError(message) {
    this.message = message
    this.name = "SessionError"
    Error.captureStackTrace(this, SessionError)
}

SessionError.prototype =
Object.create(Error.prototype)
SessionError.prototype.constructor = SessionError
```

Fine-grained exceptions

then

```
function fetchPosts (user) {
  throw new PostsError('could not fetch posts')
}

function fetchSession () {
  return new SessionError('could not fetch session')
}

fetchSession()
  .then(fetchPosts)
  .then(renderPosts)
  .catch(SessionError, handleSessionError)
  .catch(PostsError, handlePostsError)
```

Bluebird Promise

Spread Promise.all result as parameters

Parameter spread

```
Promise.all([
  download('http://foo.bar/file.tar.gz'),
  download('http://foo.bar/file.tar.gz.sig')
]).spread((file, signature) =>
  sign(file) == signature
      ? Promise.resolve(file)
      : Promise.reject('invalid signature')
)
```

Bluebird Promise

Use .all & .spread for dynamic amount of promises

When doing fixed number of promises use .join

Join promises results

then

```
Promise.join(
  download('http://foo.bar/file.tar.gz'),
  download('http://foo.bar/file.tar.gz.sig'),
  (file, signature) =>
    sign(file) == signature
      ? Promise.resolve(file)
      : Promise.reject('invalid signature')
)
```

Bluebird Promise

Resolve objects with promises

Join promises results

```
Promise.props({
  file: download('http://foo.bar/file.tar.gz'),
  sig: download('http://foo.bar/file.tar.gz.sig')
}).then(data =>
  sign(data.file) == data.sig
    ? Promise.resolve(file)
    : Promise.reject('invalid signature')
)
```

Bluebird Promise

Do some .reduce with promises

Reduce promises results

then

```
const urls = fetchProjects()

Promise.reduce(urls, (total, url) =>
  fetch(url).then(data =>
    total + data.stars), 0)
```

HTML Fetch

A Promise approach to HTTP requests

https://fetch.spec.whatwg.org

#DeprecateXHR

```
fetch('/users.json')
  .then(function(response) {
    return response.json()
  }).then(function(json) {
    console.log('parsed json', json)
  }).catch(function(ex) {
    console.log('parsing failed', ex)
  })
```

#DeprecateXHR

then

```
fetch('/users.json')
  .then(function(response) {
    return response.json()
  }).then(function(json) {
    console.log('parsed json', json)
  }).catch(function(ex) {
    console.log('parsing failed', ex)
  })
```

#DeprecateXHR

fetch is growing **so powerful**

#DeprecateXHR

then

```
$ telnet mockbin.org 80
GET /bin/2294df68-ae10-4336-a732-3170597543a9 HTTP/1.1
Accept: */*
Host: mockbin.org

------------------------------------------------------------

HTTP/1.1 200 OK
Content-Type: text/html
Custom-Header: CustomValue

------------------------------------------------------------

{"fetch": "is so cool"}
```

#DeprecateXHR

fetch promise resolve as soon as headers are ready

then

Demo

Fetching stuff from Github

https://github.com/derekstavis/promises-on-the-browser