ASYNCHRONOUS NETWORKING

Networking with Promises & `fetch()`

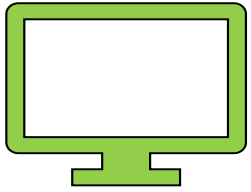
OVERVIEW

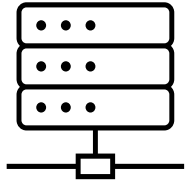
- Client-Server Model + AJAX
- Concurrency & JS
- Networking with `XMLHttpRequest()`
- Networking with Promises & `fetch()`
- Networking with `async/await & fetch()`

RECAP

Client

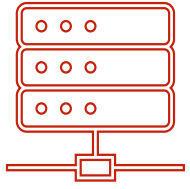
Servers





Latest News API

Super cool cats API



`XMLHttpRequest()` provides one way to do asynchronous fetching.

ES2015 introduced a new way via `fetch()` and **Promises**

Before talking about `fetch()`, what is a Promise?

PROMISE

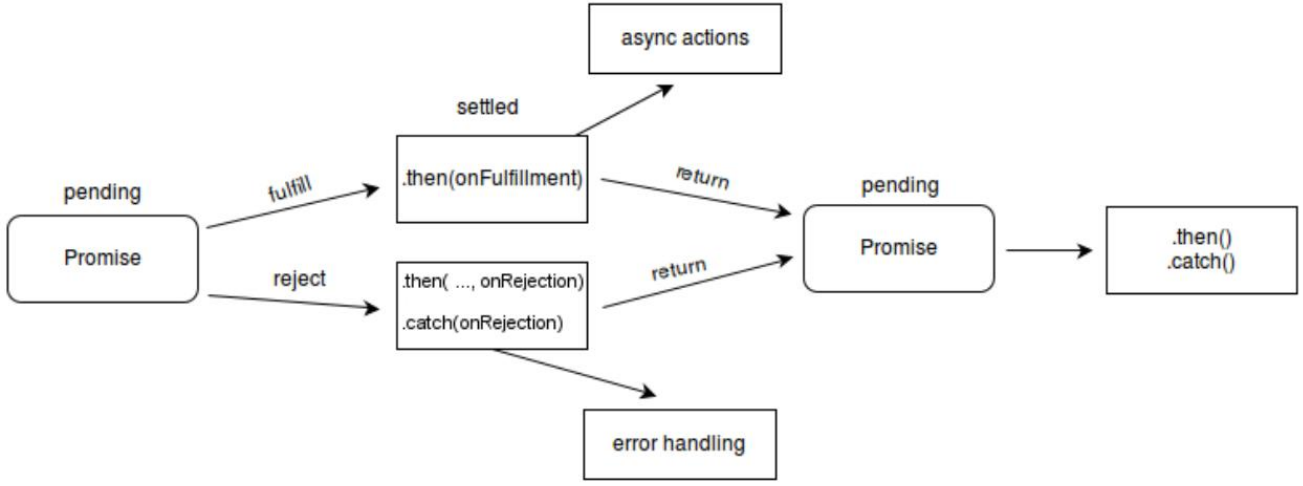


Image Credit: MDN

ES2015 Promises

- Proxy for a future value
- Evaluated asynchronously
- Support chaining, branching, error handling

Can be in one of 4 states:

- "Pending" – not evaluated yet
- "Fulfilled" – Successfully evaluated
- "Rejected" – Failed to evaluate
- "Settled" – Either rejected or fulfilled

Other useful features:

- Can be orchestrated (`Promise.all`, `Promise.race`, `etc.`)
- "Promise-like" objects can be used with Promises

BASIC API USAGE

```
1  // Creates a brand new Promise
2  const myPromise = new Promise( executor: (resolve, reject) => {
3      // if the action succeeds, call resolve() with the result
4      // or, if the action failed, call reject() with the reason
5  });
6
7  myPromise.then(
8      () => {
9          // this callback will be called if myPromise is fulfilled
10     },
11     () => {
12         // this specific callback will be called if myPromise is rejected
13     }
14 );
15
16 // In addition to giving a callback for errors in .then(), you can give a
17 // catch-all error handler as .catch()
18 myPromise.catch(
19     () => {
20         // handle the problem here
21     }
22 );
23
```

Constructor:

- Accepts a callback that takes `resolve()` and `reject()` functions
- Fulfillment = calling `resolve()`
- Rejection = calling `reject()`

.then:

- Most common way to chain promises.
- Executes the next action if the previous one fulfilled

.catch:

- Catch-all error handler for the chain above

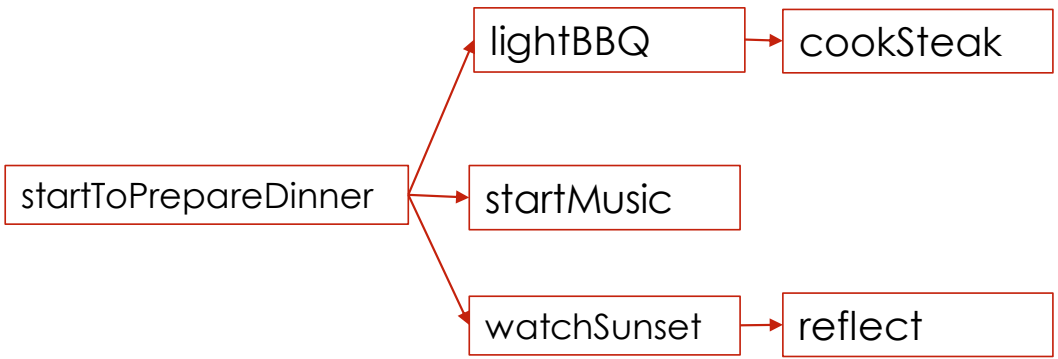
CHAINING

```
1  const dinnerPlans = startToPrepareDinner()
2      .then(lightBBQ)
3      .then(stokeFire)
4      .then(grillSteak)
5      .then(EAT)
6      .catch(eatNothing) // in case we burn ourselves
7
8  const restPlans = dinnerPlans
9      .then(watchYoutube)
10     .then(eveningStroll)
11     .catch(goToBed) // maybe the internet was out
```

- Nested execution of dependent operations
- Each `.then()` runs iff the previous promise fulfilled
- Any error/rejection that happens passed to the next-nearest `.catch()`
- After a `.catch()`, more operations can occur
- Every `.then()` returns a *new* promise which wraps the previous one.
  - Even if the given callback doesn't return a promise.
  - Stored as Russian dolls
  - But executed as a stack i.e. most-nested happens first

BRANCHING

```
1  const dinnerPlans = startToPrepareDinner();
2
3  dinnerPlans
4      .then(lightBBQ).then(cookSteak);
5
6  dinnerPlans
7      .then(startMusic);
8
9  dinnerPlans
10     .then(watchSunset).then(reflect);
```



- Multiple .then()'s on the same promise = branching
- When the parent promise is resolved, all .then()'s invoked in order
- Allows for complex control-flow on fulfillment

ERROR-HANDLING

```
const rejectedPromise = new Promise( executor: (resolve, reject) => reject( reason: "oops"))
    .then(() => "Never reached") Promise<string>
    .catch((error) => {
        // the rejection means we'll end up in here
        console.log(error);
    });

const exceptionPromise = new Promise( executor: (resolve, reject) => throw new Error("oops"))
    .then(() => "Never reached") Promise<string>
    .catch((err) => {
        // even though there wasn't an explicit rejection,
        // any exceptions cause an implicit rejection


        // rethrowing...
        throw err;
    }) Promise<string>
    .catch((err) => {
        // exceptions can also be rethrown and recaught in further down .catch() clauses
        // quite useful
        console.log(err);
    })
```

- Errors/Exceptions always cause rejections
- Explicit rejections done via calling `reject()`
- Any exceptions cause an implicit rejection
- `.catch()` clauses can handle errors or pass them to the next `.catch()` by rethrowing
- `.finally()` is also available that will run regardless of if an error occurred or not

ERROR-HANDLING GOTCHAS

```
1  function foo() {
2      try {
3          // explicit rejection
4          const baz = new Promise( executor: (res, reject) => {
5              reject( reason: "oops");
6          });
7      } catch (e) {
8          // do we enter here?
9          console.log(e);
10     }
11 }
```

- Promises always asynchronous
- Current function context *always* completes before a Promise is settled
- This means Promises don't work with `try/catch` like on the left!
- Good idea to add an event listener to the window to handle the `unhandledrejection` event

PROMISE ORCHESTRATION

```
1   Promise.all([...])
2
3   Promise.allSettled([...])
4
5   Promise.any([...])
6
7   Promise.race([...])
8
9   Promise.reject(val)
10
11  Promise.resolve(val)
12
```

- The `Promise` class has some utilities for easy orchestration
- `Promise.all()`: returns a promise that resolves iff all of the promises passed to it resolve
- `Promise.allSettled()`: returns a promise that resolves once all of the promises passed to it are resolved
- `Promise.any()`: returns a promise that resolves if at least one of the promises passed to it resolves
- `Promise.race()`: returns a promise which resolves as soon as one of the promises passed to it resolves
- `Promise.reject()`: immediately return a rejected promise with a value
- `Promise.resolve()`: immediately return a resolved promise with a value

PROMISE-LIKE OBJECTS (THENABLES)

```
1  class customThenable {
2      then(onFulfill, onReject) {
3          console.log("inside a thenable!");
4          onFulfill();
5      }
6  }
7
8  Promise.resolve(new customThenable()).then(
9      () => console.log("used a custom thenable")
10 );
11
12 // output:
13 // inside a custom thenable!
14 // used a custom thenable
```

- Any object or class with a **then()** considered "promise-like" or a "**thenable".**
- Can be used with Promise chaining
- Useful for fine-grained control over how chaining works for custom types

THE FETCH API

```
1   // only the URL is required
2   fetch("http://example.com/movies.json", {
3     method: "POST", // this object is optional
4   })
5       // return the body as JSON
6       .then(res => res.json())
7
8       // finally access the JSON
9       .then(js => console.log(js));
10
```

- Promise-based native JS API to download remote resources
- Resolves if a Response is received, even if the HTTP status code is not 200
- Rejects if there is any network error
- Access the result of the request via chaining `then()`s
- Optional 2nd argument to `fetch()` can control the Request options e.g.
  - Authentication
  - CORS
  - HTTP method

PROMISE + FETCH DEMO

See examples/promise-fetch

FETCH LIMITATIONS

XMLHttpRequest

- Works even on very old browsers
- Gives large download progress
- Easily cancelled

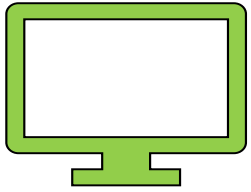
Fetch

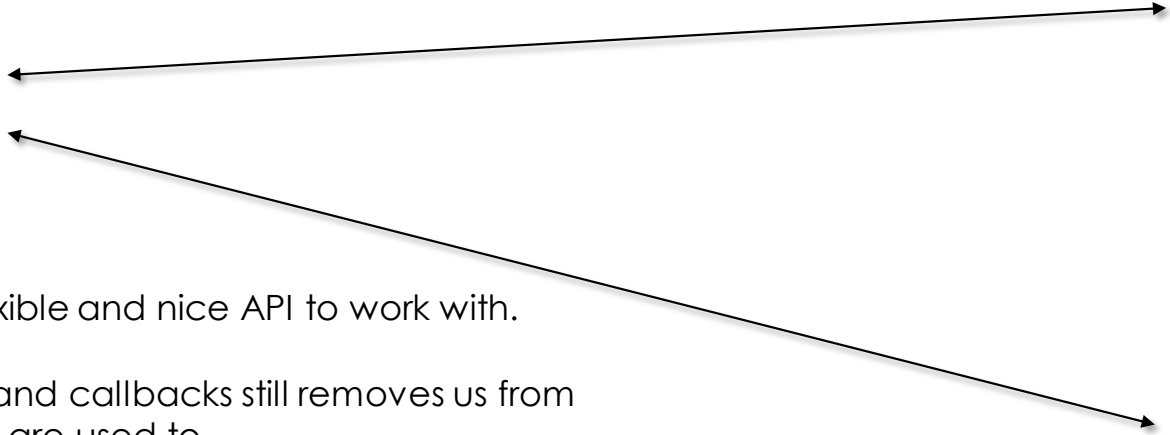
- Only works on browsers with Promise support (less of a problem nowadays)
- Promises not easily cancellable
- More complex functionality implemented via the Streams API which has a non-trivial learning curve

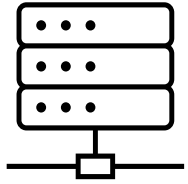
MOVING FORWARD

Client

Servers

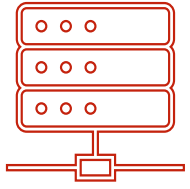






Latest News API

Super cool cats API



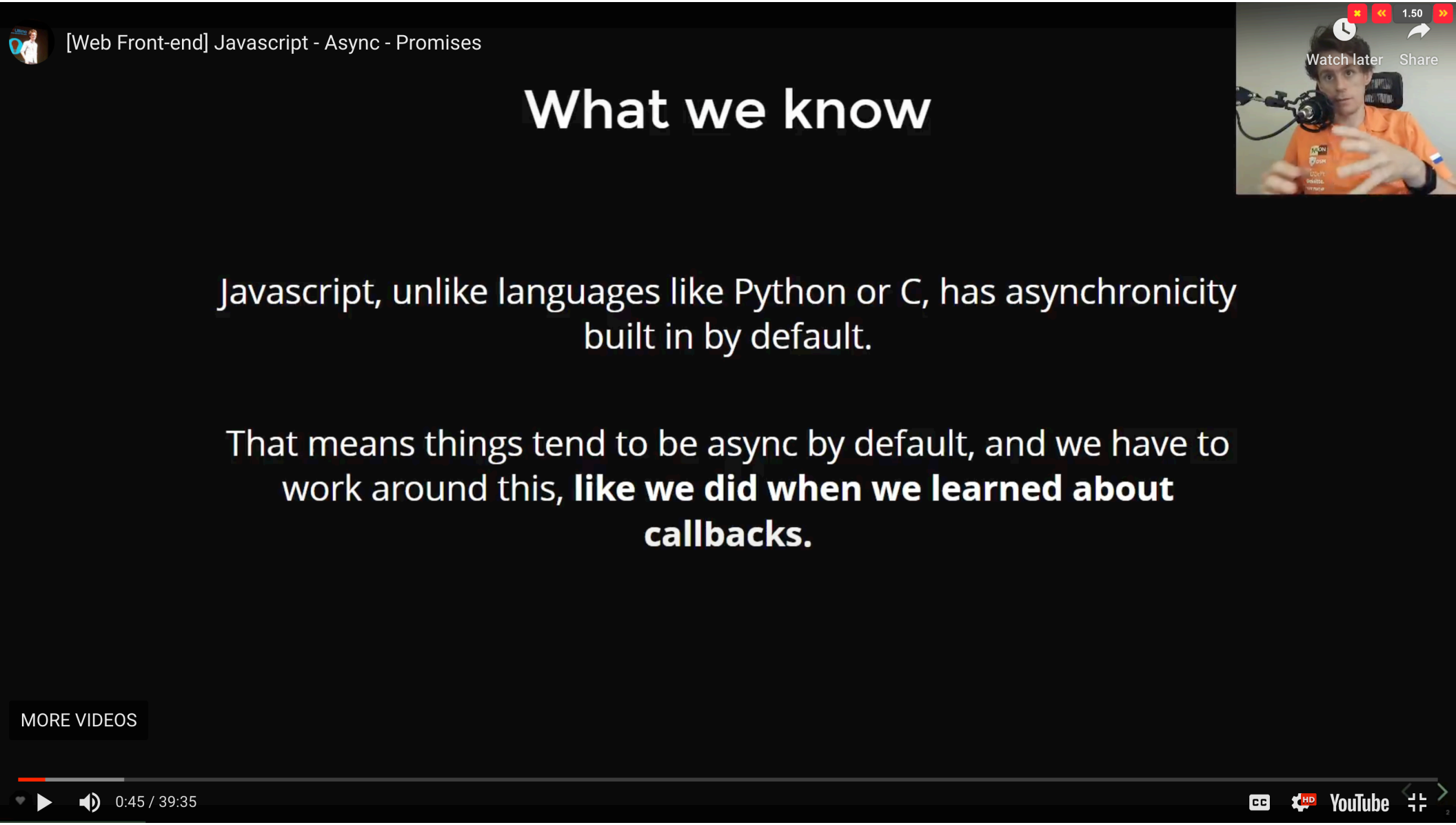
`fetch()` is a very flexible and nice API to work with.

However, chaining and callbacks still removes us from writing code like we are used to.

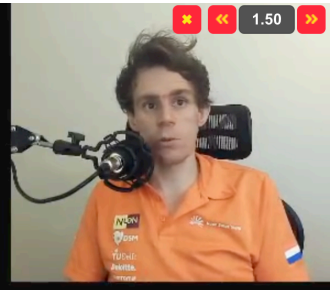
Can we do even better?

SUMMARY

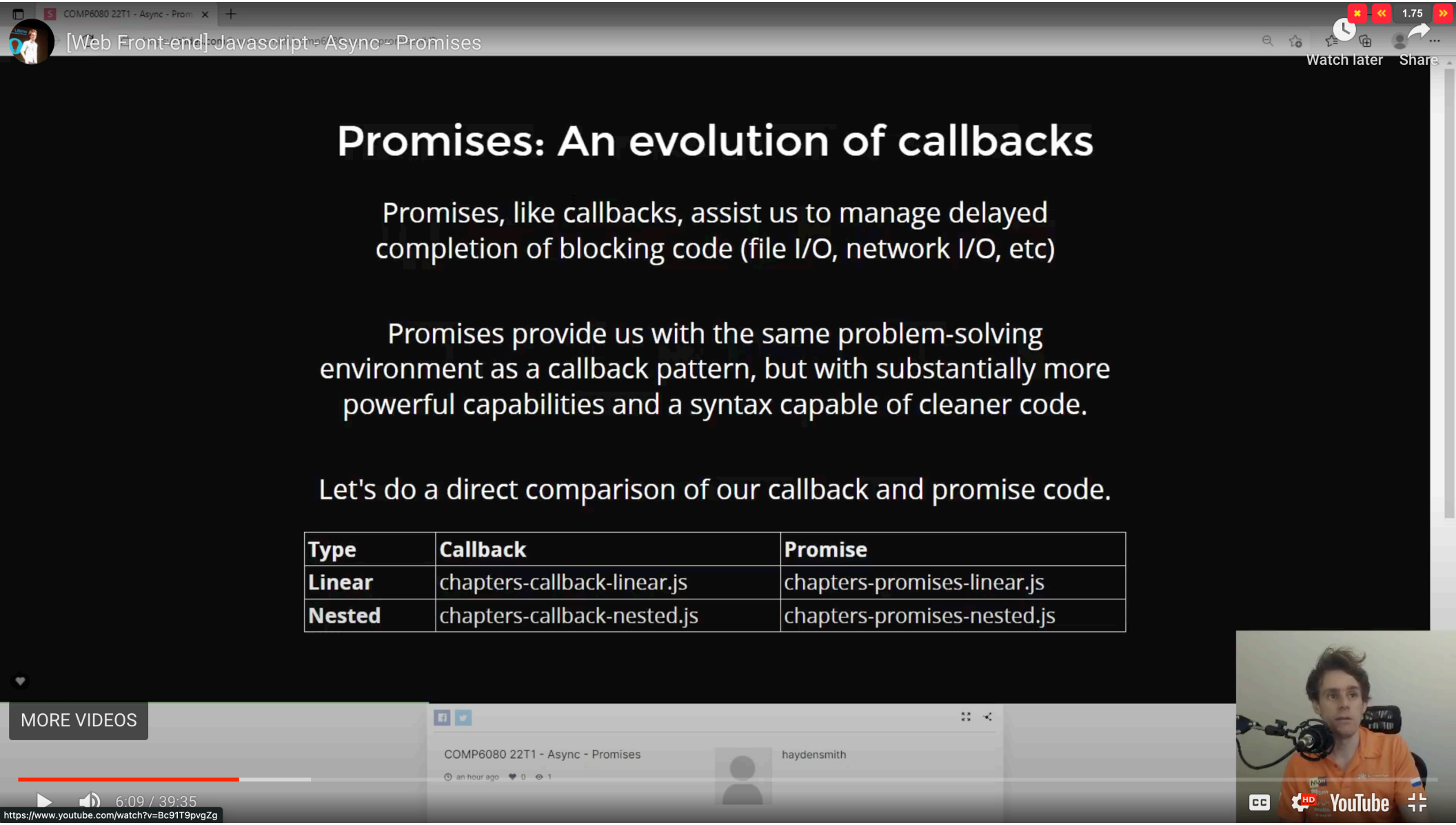
- Today:
  - Promises
  - Using Promises with `fetch()`
- Coming Up Next:
  - Networking with `async/await & fetch()`

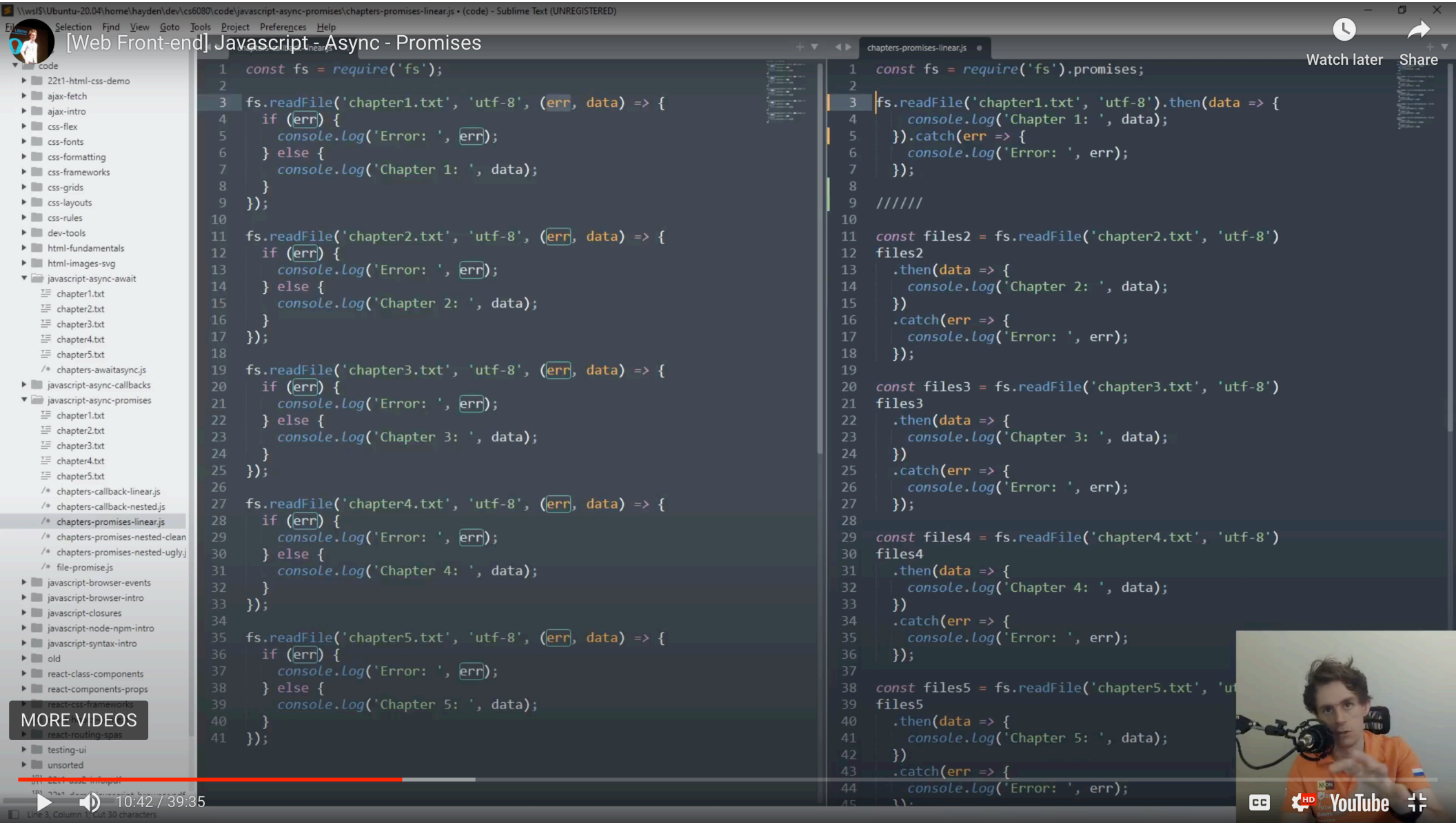

[Web Front-end] Javascript - Async - Promises
1.50
Watch later
Share
What we know
Javascript, unlike languages like Python or C, has asynchronicity built in by default.
That means things tend to be async by default, and we have to work around this, **like we did when we learned about callbacks.**
MORE VIDEOS
0:45 / 39:35
YouTube

Let's review callbacks



- Look at **chapters-callback-linear.js**:
  - How async JS works through use of callbacks
- Look at **chapters-callback-nested.js:**
  - How to control async JS through callback nesting


COMP6080 22T1 - Async - Prom
[Web Front-end] Javascript - Async - Promises
1.75
Watch later
Share
Promises: An evolution of callbacks
Promises, like callbacks, assist us to manage delayed completion of blocking code (file I/O, network I/O, etc)
Promises provide us with the same problem-solving environment as a callback pattern, but with substantially more powerful capabilities and a syntax capable of cleaner code.
Let's do a direct comparison of our callback and promise code.
Type
Callback
Promise
Linear
chapters-callback-linear.js
chapters-promises-linear.js
Nested
chapters-callback-nested.js
chapters-promises-nested.js
MORE VIDEOS
COMP6080 22T1 - Async - Promises
haydensmith
an hour ago
6:09 / 39:35
https://www.youtube.com/watch?v=Bc91T9pvgZg
YouTube


[Web Front-end] Javascript - Async - Promises
Watch later
Share
chapters-promises-linear.js
const fs = require('fs');
fs.readFile('chapter1.txt', 'utf-8', (err, data) => {
if (err) {
console.log('Error: ', err);
} else {
console.log('Chapter 1: ', data);
}
});
fs.readFile('chapter2.txt', 'utf-8', (err, data) => {
if (err) {
console.log('Error: ', err);
} else {
console.log('Chapter 2: ', data);
}
});
fs.readFile('chapter3.txt', 'utf-8', (err, data) => {
if (err) {
console.log('Error: ', err);
} else {
console.log('Chapter 3: ', data);
}
});
fs.readFile('chapter4.txt', 'utf-8', (err, data) => {
if (err) {
console.log('Error: ', err);
} else {
console.log('Chapter 4: ', data);
}
});
fs.readFile('chapter5.txt', 'utf-8', (err, data) => {
if (err) {
console.log('Error: ', err);
} else {
console.log('Chapter 5: ', data);
}
});
const fs = require('fs').promises;
fs.readFile('chapter1.txt', 'utf-8').then(data => {
console.log('Chapter 1: ', data);
}).catch(err => {
console.log('Error: ', err);
});
//////
const files2 = fs.readFile('chapter2.txt', 'utf-8')
files2
.then(data => {
console.log('Chapter 2: ', data);
})
.catch(err => {
console.log('Error: ', err);
});
const files3 = fs.readFile('chapter3.txt', 'utf-8')
files3
.then(data => {
console.log('Chapter 3: ', data);
})
.catch(err => {
console.log('Error: ', err);
});
const files4 = fs.readFile('chapter4.txt', 'utf-8')
files4
.then(data => {
console.log('Chapter 4: ', data);
})
.catch(err => {
console.log('Error: ', err);
});
const files5 = fs.readFile('chapter5.txt', 'ut
files5
.then(data => {
console.log('Chapter 5: ', data);
})
.catch(err => {
console.log('Error: ', err);
MORE VIDEOS
10:42 / 39:35
YouTube

How are these promises written?

Let's learn about how to turn a fileRead standard callback function into a promise. This involves understanding the structure of a promise.

```
// Creates a brand new Promise
const myPromise = new Promise( executor: (resolve, reject) => {
    // if the action succeeds, call resolve() with the result
    // or, if the action failed, call reject() with the reason
});

myPromise.then(
    () => {
        // this callback will be called if myPromise is fulfilled
    },
    () => {
        // this specific callback will be called if myPromise is rejected
    }
);

// In addition to giving a callback for errors in .then(), you can give a
// catch-all error handler as .catch()
myPromise.catch(
    () => {
        // handle the problem here
    }
);
```

Constructor:

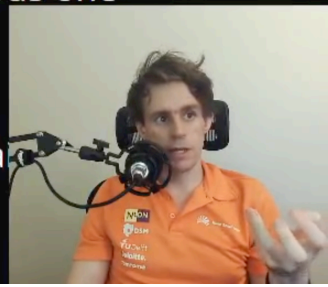
- Accepts a callback that takes resolve() and reject() functions
- Fulfillment = calling resolve()
- Rejection = calling reject()

.then:

- Most common way to chain promises.
- Executes the next action if the previous one fulfilled

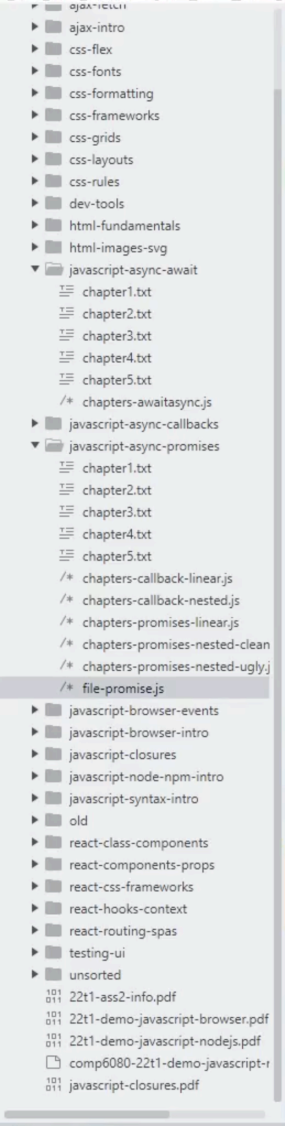
.catch:

- Catch-all error handler for the cha



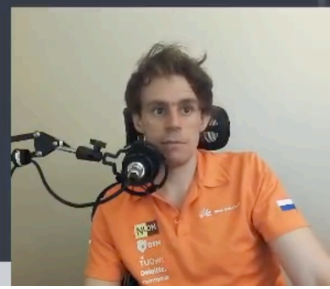
\\wsl$\Ubuntu-20.04\home\hayden\dev\cs6080\code\javascript-async-promises\file-promise.js (code) - Sublime Text (UNREGISTERED)

File Edit Selection Find View Goto Tools Project Preferences Help



chapters-callback-linear.js | chapters-callback-nested.js | file-promise.js | chapters-promises-nested-ugly.js | chapters-promises-linear.js

```
1  const fs = require('fs');
2
3  function readFilePromise1(filename, encoding) {
4    function promiseHandler(resolve, reject) {
5      console.log('Running promise!');
6      function callback(err, data) {
7        if (err) {
8          reject(err);
9        } else {
10         resolve(data);
11       }
12     }
13     fs.readFile(filename, encoding, callback);
14   }
15   return new Promise(promiseHandler);
16 };
17
18 const readFilePromise2 = (filename, encoding) => {
19   return new Promise((resolve, reject) => {
20     fs.readFile(filename, encoding, (err, data) => {
21       if (err) reject(err);
22       else resolve(data);
23     });
24   })
25 };
26
27 const myPromise = readFilePromise2('chapter1.txt', 'utf-8');
28 myPromise.then(data => console.log('Data', data));
29 myPromise.catch(err => console.log('Error', err));
30
```



Line 27, Column 35; Saved \\wsl$\Ubuntu-20.04\home\hayden\dev\cs6080\code\javascript-async-promises\file-promise.js (UTF-8)

How are these promises written?

Let's learn about how to turn a fileRead standard callback function into a promise. This involves understanding the structure of a promise.

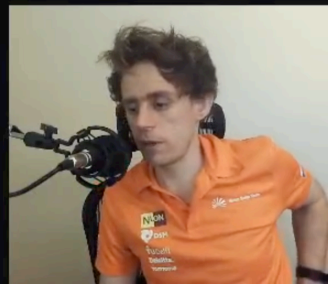
```
// Creates a brand new Promise
const myPromise = new Promise( executor: (resolve, reject) => {
    // if the action succeeds, call resolve() with the result
    // or, if the action failed, call reject() with the reason
});

myPromise.then(
    () => {
        // this callback will be called if myPromise is fulfilled
    },
    () => {
        // this specific callback will be called if myPromise is rejected
    }
);

// In addition to giving a callback for errors in .then(), you can give a
// catch-all error handler as .catch()
myPromise.catch(
    () => {
        // handle the problem here
    }
);
```

A promise has a few possible states:

- Pending: Not evaluated (still processing)
- Settled: Either rejected or fulfilled
  - Fulfilled: Done! Call resolve()
  - Rejected: Error! Call reject()



[Web Front-end] Javascript - Async - Promises

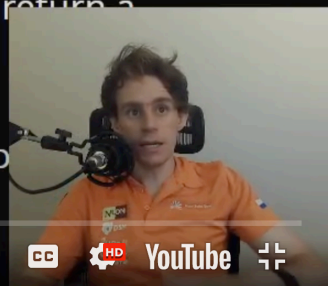
Watch later Share

Promise Chaining

Promises can be chained together to enforce order of processing and ensure clear and concise syntax. This is shown in **chapters-promises-nested-clean.js**

```
1  const dinnerPlans = startToPrepareDinner()
2      .then(lightBBQ)
3      .then(stokeFire)
4      .then(grillSteak)
5      .then(EAT)
6      .catch(eatNothing) // in case we burn ourselves
7
8  const restPlans = dinnerPlans
9      .then(watchYoutube)
10     .then(eveningStroll)
11     .catch(goToBed) // maybe the internet was out
```

- Nested execution of dependent operations
- Each .then() runs iff the previous promise fulfilled
- Any error/rejection that happens passed to the next-nearest .catch()
- After a .catch(), more operations can occur
- Every .then() returns a new promise which wraps the previous one.
  - Even if the given callback doesn't return a promise.
  - Stored as Russian dolls
    - But executed as a stack i.e. mo happens first



22:50 / 39:35

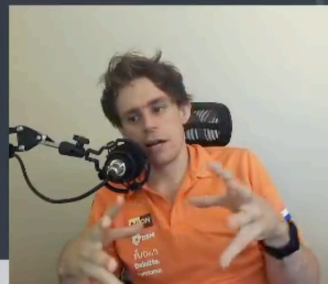
CC HD YouTube

\\wsl$\Ubuntu-20.04\home\hayden\dev\cs6080\code\javascript-async-promises\chapters-promises-nested-clean.js (code) - Sublime Text (UNREGISTERED)

File Edit Selection Find View Goto Tools Project Preferences Help

chapters-callback-linear.js | chapters-callback-nested.js | file-promise.js | chapters-promises-nested-ugly.js | chapters-promises-nested-clean.js | chapters-promises-linear.js

```
1  const fs = require('fs').promises;
2
3  const handler = (data, index) => {
4    console.log(`Chapter ${index}: `, data);
5    return fs.readFile(`chapter${index + 1}.txt`, 'utf-8');
6  };
7
8  const files1 = fs.readFile('chapter1.txt', 'utf-8')
9  files1
10   .then(data => handler(data, 1))
11   .then(data => handler(data, 2))
12   .then(data => handler(data, 3))
13   .then(data => handler(data, 4))
14   .then(data => {
15     console.log('Chapter 5: ', data);
16   })
17   .catch(err => {
18     console.log('Error: ', err);
19   });
20
```

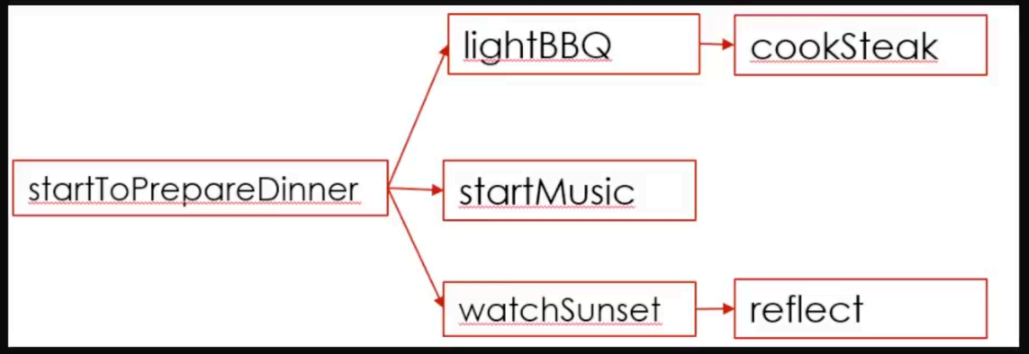


Line 12, Column 34

Promise Branching

Branching promises allow you to control what is processed concurrently and what is processed linearly

```
1  const dinnerPlans = startToPrepareDinner();
2
3  dinnerPlans
4      .then(lightBBQ).then(cookSteak);
5
6  dinnerPlans
7      .then(startMusic);
8
9  dinnerPlans
10     .then(watchSunset).then(reflect);
```



- Multiple .then()'s on the same promise = branching
- When the parent promise is resolved, all .the
  invoked in order
- Allows for complex control-flow on fulfillmer

Promise Error Handling

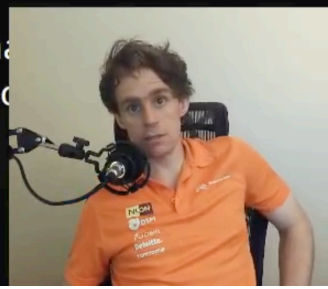
Mostly error handling is quite straightforward, but there are a few extra details worth considering.

```
const rejectedPromise = new Promise( executor: (resolve, reject) => reject( reason: "oops"))
    .then(() => "Never reached") Promise<string>
    .catch((error) => {
        // the rejection means we'll end up in here
        console.log(error);
    });

const exceptionPromise = new Promise( executor: (resolve, reject) => throw new Error("oops"))
    .then(() => "Never reached") Promise<string>
    .catch((err) => {
        // even though there wasn't an explicit rejection,
        // any exceptions cause an implicit rejection

        // rethrowing...
        throw err;
    }) Promise<string>
    .catch((err) => {
        // exceptions can also be rethrown and recaught in further down .catch() clauses
        // quite useful
        console.log(err);
    })
```

- Errors/Exceptions always cause rejections
- Explicit rejections done via calling reject()
- Any exceptions cause an implicit rejection
- .catch() clauses can handle errors or pass them to the next .catch() by rethrowing
- .finally() is also available tha
  regardless of if an error occ

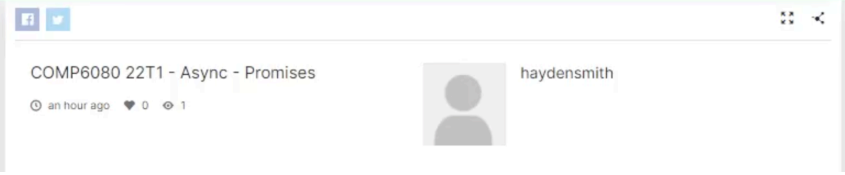


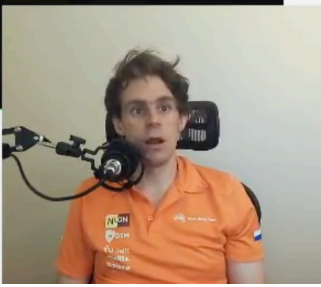
Promise Orchestration

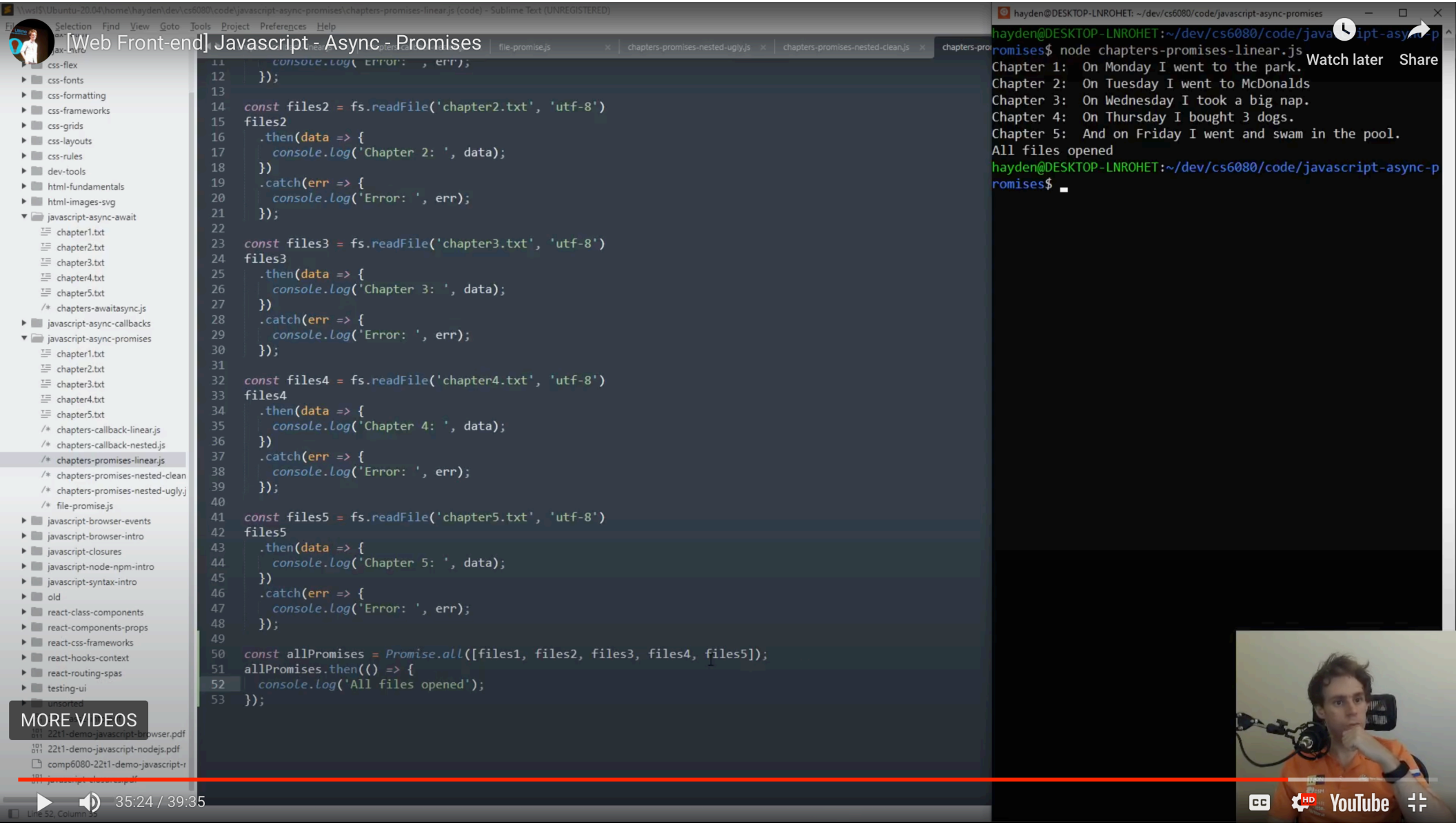
The **Promise** class has some utilities for easy orchestration

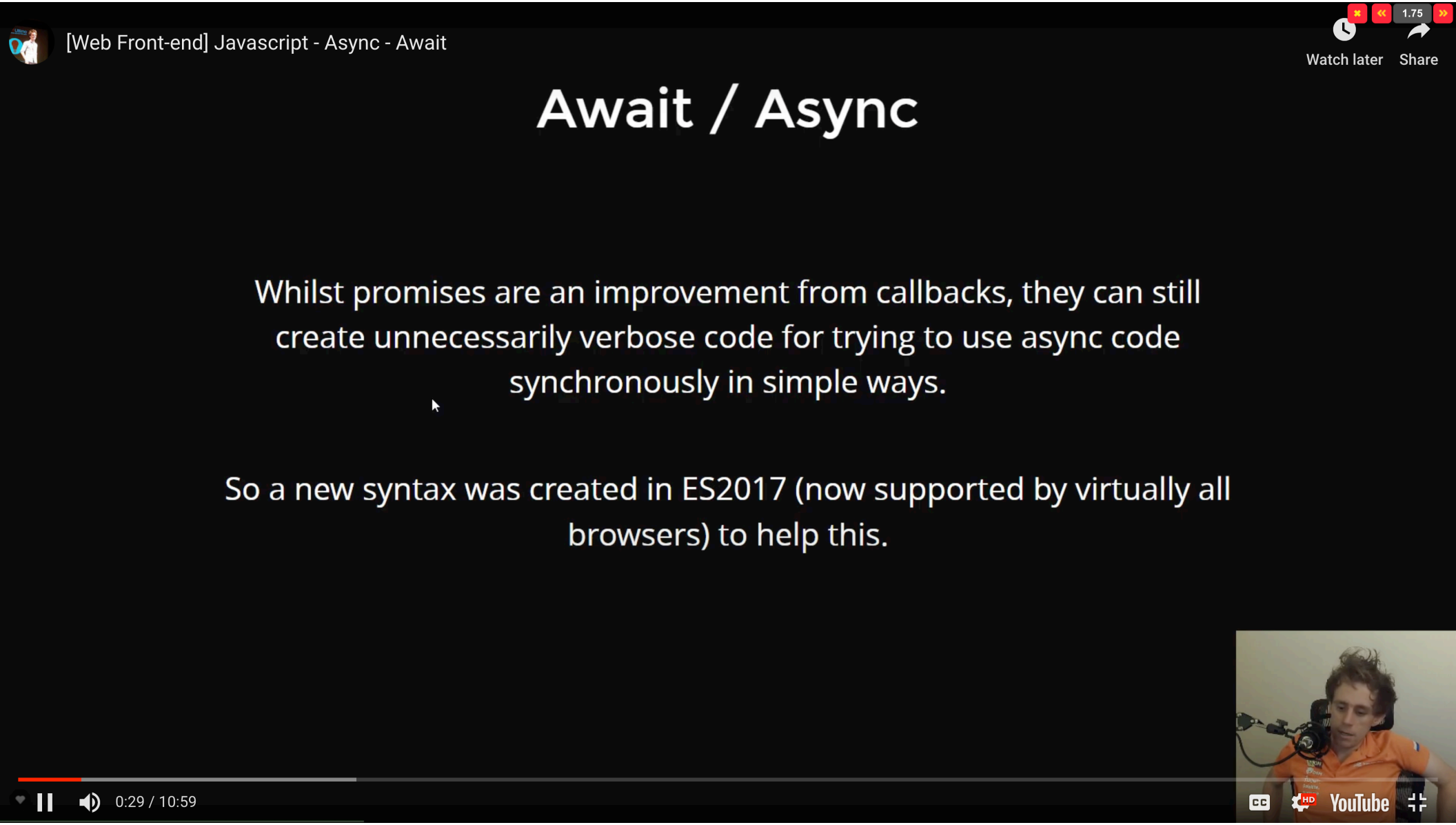
```
1   Promise.all([...])
2
3   Promise.allSettled([...])
4
5   Promise.any([...])
6
7   Promise.race([...])
8
9   Promise.reject(val)
10
11  Promise.resolve(val)
12
```

- Promise.all(): returns a promise that resolves iff all of the promises passed to it resolve
- Promise.allSettled(): returns a promise that resolves once all of the promises passed to it are resolved
- Promise.any(): returns a promise that resolves if at least one of the promises passed to it resolves
- Promise.race(): returns a promise which resolves as soon as one of the promises passed to it resolves
- Promise.reject(): immediately return a rejected promise with a value
- Promise.resolve(): immediately return a resolved promise with a value






[Web Front-end] Javascript - Async - Promises
Watch later
Share
MORE VIDEOS
35:24 / 39:35
YouTube


[Web Front-end] Javascript - Async - Await
1.75
Watch later
Share
Await / Async
Whilst promises are an improvement from callbacks, they can still create unnecessarily verbose code for trying to use async code synchronously in simple ways.
So a new syntax was created in ES2017 (now supported by virtually all browsers) to help this.
0:29 / 10:59
YouTube

Await / Async

This is a very basic comparison of the two in action.

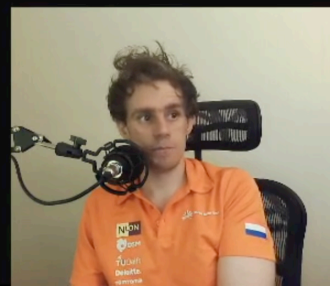
```
const fs = require('fs').promises;

const files1 = fs.readFile('chapter1.txt', 'utf-8')
files1
  .then(data => {
    console.log('Chapter 1: ', data);
  })
  .catch(err => {
    console.log('Error: ', err);
  });
```

```
const fs = require('fs').promises;

const readFiles = async () => {
  let data = '';
  try {
    data = await fs.readFile('chapter1.txt', 'utf-8');
    console.log('Chapter 1: ', data);
  } catch (err) {
    console.log('Error: ', err);
  }
};

readFiles();
```



Await / Async

The benefit of async/await becomes more noticeable when doing substantially more async calls that you want to process linearly.

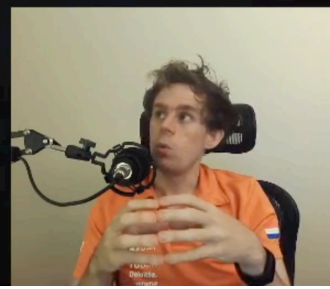
```
const fs = require('fs').promises;

const files1 = fs.readFile('chapter1.txt', 'utf-8')
files1
  .then(data => {
    console.log('Chapter 1: ', data);
    return fs.readFile('chapter2.txt', 'utf-8')
  })
  .then(data => {
    console.log('Chapter 2: ', data);
    return fs.readFile('chapter3.txt', 'utf-8')
  })
  .then(data => {
    console.log('Chapter 3: ', data);
    return fs.readFile('chapter4.txt', 'utf-8')
  })
  .then(data => {
    console.log('Chapter 4: ', data);
    return fs.readFile('chapter5.txt', 'utf-8')
  })
  .then(data => {
    console.log('Chapter 5: ', data);
  })
  .catch(err => {
    console.log('Error: ', err);
  });
```

```
const fs = require('fs').promises;

const readFiles = async () => {
  let data = '';
  try {
    data = await fs.readFile('chapter1.txt', 'utf-8');
    console.log(data);
    data = await fs.readFile('chapter2.txt', 'utf-8');
    console.log(data);
    data = await fs.readFile('chapter3.txt', 'utf-8');
    console.log(data);
    data = await fs.readFile('chapter4.txt', 'utf-8');
    console.log(data);
    data = await fs.readFile('chapter5.txt', 'utf-8');
    console.log(data);
  } catch (err) {
    console.log(err);
  }
};

readFiles();
```



\\wsl$\Ubuntu-20.04\home\hayden\dev\cs6080\code\javascript-async-await\chapters-awaitasync.js (code) - Sublime Text (UNREGISTERED)

File Edit Selection Find View Goto Tools Project Preferences Help

- ajax-intro
- css-flex
- css-fonts
- css-formatting
- css-frameworks
- css-grids
- css-layouts
- css-rules
- dev-tools
- html-fundamentals
- html-images-svg
- javascript-async-await
  - chapter1.txt
  - chapter2.txt
  - chapter3.txt
  - chapter4.txt
  - chapter5.txt
  - chapters-awaitasync.js
- javascript-async-callbacks
- javascript-async-promises
  - chapter1.txt
  - chapter2.txt
  - chapter3.txt
  - chapter4.txt
  - chapter5.txt
  - chapters-callback-linear.js
  - chapters-callback-nested.js
  - chapters-promises-linear.js
  - chapters-promises-nested-clean
  - chapters-promises-nested-ugly.j
  - file-promise.js
- javascript-browser-events
- javascript-browser-intro
- javascript-closures
- javascript-node-npm-intro
- javascript-syntax-intro
- old
- react-class-components
- react-components-props
- react-css-frameworks
- react-hooks-context
- react-routing-spas
- testing-ui
- unsorted
- 22t1-ass2-info.pdf
- 22t1-demo-javascript-browser.pdf
- 22t1-demo-javascript-nodejs.pdf
- comp6080-22t1-demo-javascript-r
- javascript-closures.pdf

chapters-promises-linear.js | chapters-promises-nested-clean.js | chapters-awaitasync.js | untitled

```
1  const fs = require('fs').promises;
2
3  const readFiles = async () => {
4    let data = '';
5    try {
6      console.log('Chapter 1:', await fs.readFile('chapter1.txt', 'utf-8'
7      console.log('Chapter 2:', await fs.readFile('chapter2.txt', 'utf-8'
8      console.log('Chapter 3:', await fs.readFile('chapter3.txt', 'utf-8'
9      console.log('Chapter 4:', await fs.readFile('chapter4.txt', 'utf-8'
10     console.log('Chapter 5:', await fs.readFile('chapter5.txt', 'utf-8'
11   } catch (err) {
12     console.log(err);
13   }
14 };
15
16 readFiles();
17
```

5 selection regions

hayden@DESKTOP-LNROHET: ~/dev/cs6080/code/javascript-async-await 1.75

```
hayden@DESKTOP-LNROHET:~/dev/cs6080/code/javascript-async-a
wait$ node chapters-awaitasync.js
Chapter 1:  Promise { <pending> }
hayden@DESKTOP-LNROHET:~/dev/cs6080/code/javascript-async-a
wait$ node chapters-awaitasync.js
/home/hayden/dev/cs6080/code/javascript-async-await/chapter
s-awaitasync.js:6
    data = await fs.readFile('chapter1.txt', 'utf-8');
           ^^^^^

SyntaxError: await is only valid in async function
    at wrapSafe (internal/modules/cjs/loader.js:1001:16)
    at Module._compile (internal/modules/cjs/loader.js:1049
:27)
    at Object.Module._extensions..js (internal/modules/cjs/
loader.js:1114:10)
    at Module.load (internal/modules/cjs/loader.js:950:32)
    at Function.Module._load (internal/modules/cjs/loader.j
s:790:12)
    at Function.executeUserEntryPoint [as runMain] (interna
l/modules/run_main.js:76:12)
    at internal/main/run_main_module.js:17:47
hayden@DESKTOP-LNROHET:~/dev/cs6080/code/javascript-async-a
wait$ node chapters-awaitasync.js
Chapter 1:  On Monday I went to the park.
hayden@DESKTOP-LNROHET:~/dev/cs6080/code/javascript-async-a
wait$ node chapters-awaitasync.js
Chapter 1:  On Monday I went to the park.
hayden@DESKTOP-LNROHET:~/dev/cs6080/code/javascript-async-a
wait$ node chapters-awaitasync.js
Chapter 1:  On Monday I went to the park.
hayden@DESKTOP-LNROHET:~/dev/cs6080/code/javascript-async-a
wait$ node chapters-awaitasync.js
Chapter 1: On Monday I went to the park.
Chapter 2: On Tuesday I went to McDonalds
Chapter 3: On Wednesday I took a big nap.
Chapter 4: On Thursday I bought
Chapter 5: And on Friday I went
hayden@DESKTOP-LNROHET:~/dev/cs6
wait$ node chapters-awaitasync.j
Chapter 1: On Monday I went to t
Chapter 2: On Tuesday I went to
Chapter 3: On Wednesday I took a
Chapter 4: On Thursday I bought
Chapter 5: And on Friday I went
hayden@DESKTOP-LNROHET:~/dev/cs6
wait$
```