Async/Await

I Promise to await for async code...

What is “asynchronous” code?

Asynchronous (aka *async*) literally means "happening at disconnected times."

Async code in JS will run at an arbitrary (unknown) future time, and *other JS code* can run in the *meantime*.

An incomplete and oversimplified statement… But that helps with comprehension (and we’ll fill in the gaps over the next slides)

What is “asynchronous” code?

```
console.log("One")
setTimeout(() => console.log("Two"), 10)
console.log("Three")
```

- In which order will the logs fire?

What do we mean by “JavaScript won't wait for it”? Well, take a look at this code? In which order will the logs fire?

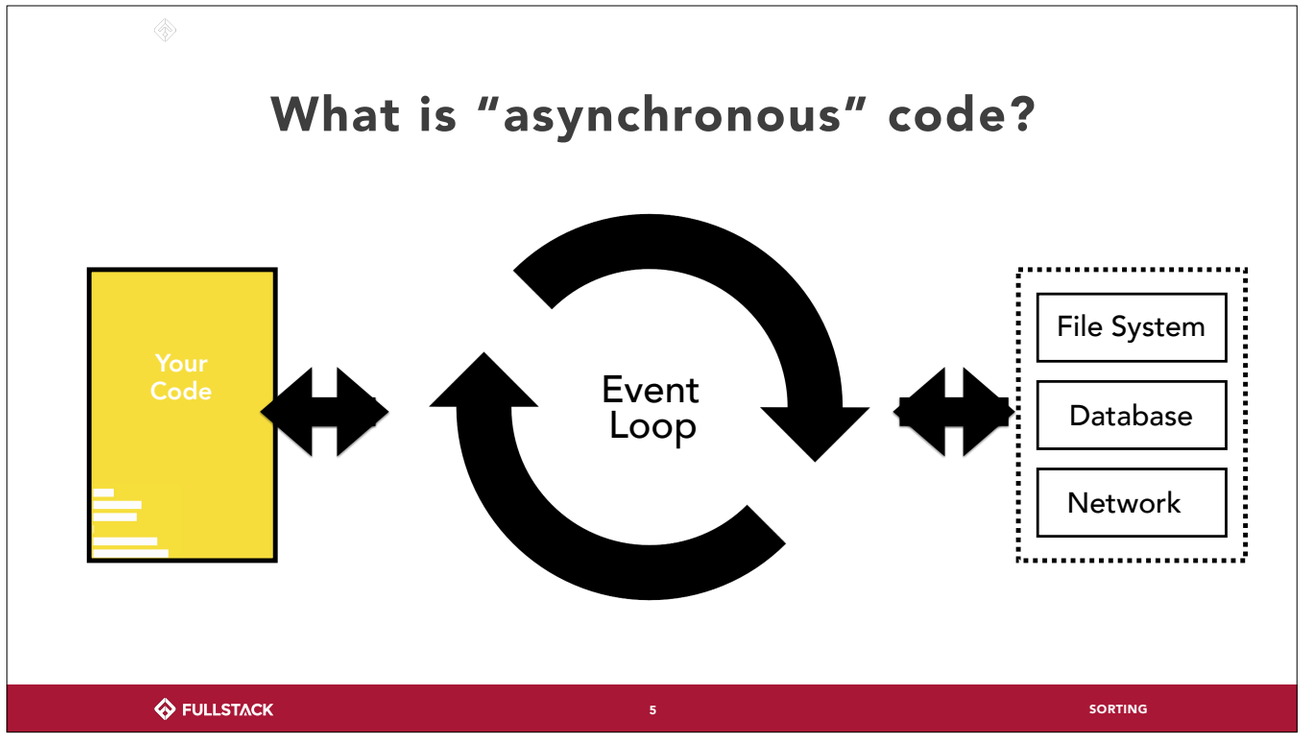
What is “asynchronous” code?

```
console.log("One")
setTimeout(() => console.log("Two"), 10)
console.log("Three")
```

- In which order will the logs fire?

  One
  Three
  Two

JavaScript won’t wait for line two - it will provide a callback and move on, executing line 3.



Asynchronous programming means that the engine runs in an event loop.
<click>
When a blocking operation is needed, the request is started, and **the code keeps running** (without blocking for the result).
<click>
When the response is ready, it causes an event handler to be run, where the control flow continues.

What’s the benefit of this model? In this way, the program can handle many concurrent operations.

How to handle asynchronous code?

1. Callbacks

When the our request to the filesystem, or to the database comes back from the event loop, where do we handle it?

Async with callbacks

```
console.log("Getting Configuration")
fs.readFile('/config.json', 'utf8', (err, data) => {
  console.log("Got configuration:", data)
});
console.log("Moving on…");
```

- BTW, In which order will the logs fire?

So, although simple, callbacks can make the order in which our program runs a little confusing…
…but that's not the only problem with callbacks.

Problems with callbacks

```
const tryGetRich = () => {
  readFile('/luckyNumbers.txt', (err, fileContent) => {
    // Do something with lucky numbers
  })
}
```


Suppose I want to get rich by betting on horses…
I'll start by reading a file with my lucky numbers…

Problems with callbacks

```
const tryGetRich = () => {
  readFile('/luckyNumbers.txt', (err, fileContent) => {
    nums = fileContent.split(",");
    nums.forEach(num => {
      bookmaker.getHorse(num, (err, horse) => {
        // Ok, this is getting a little confusing
      })
    })
  })
}
```


Then, for each one lucky number, I want to find a horse with the corresponding number…

Problems with callbacks

```
const tryGetRich = () => {
  readFile('/luckyNumbers.txt', (err, fileContent) => {
    nums = fileContent.split(",");
    nums.forEach(num => {
      bookmaker.getHorse(num, (err, horse) => {
        bookmaker.bet(horse, 1, (success) => {
          if(success)
            // Help...
        })
      })
      console.log('When will I run??')
    })
  })
}
```


And then, I'll place a bet on each horse… Ok, you got it…
Callback Hell… Pyramid of Doom… (And we're not even considering error handling here…).

How to handle asyncronous code?

1. Callbacks
2. **Promises**

For all those reasons, newer versions of JavaScript introduced a more elegant solution: Promises and Async/await

Callbacks vs Promises

CALLBACKS

```
const tryGetRich = () => {

 readFile('/luckyNumber.txt', (err, num) => {
   bookmaker.bet(num, (err, success) => {
     if(success) {
       console.log("I'm rich!")
     }
   })
 })

}
```


Take this example of two, sequential async operations…. It’s starting to become that pyramid of doom we talked about earlier…

Callbacks vs Promises

CALLBACKS

```
const tryGetRich = () => {

 readFile('/luckyNumber.txt', (err, num) => {
   bookmaker.bet(num, (err, success) => {
     if(success) {
       console.log("I'm rich!")
     }
   })
 })

}
```

ASYNC/AWAIT

(PROMISES)

```
const tryGetRich = async () => {

 let num = await readFileAsync(‘/luckyNumber.txt')
 let success = await bookmaker.bet(num)

 if(success) {
   console.log("I'm rich!")
 }

}
```


At first, it may look a subtle difference, but Async/await makes asynchronous code look and behave a little more like synchronous code.

...but we are getting ahead of ourselves.

What is a Promise?

- A promise is a JavaScript object that represents the eventual result of an asynchronous operation.
- Again, just an object with *value* and *status*.

Well, there are a few things other than value and status, but we'll talk about that later.

What is a Promise?

```
readFileAsync('/luckyNumber.txt')
```

```
{
  [[PromiseValue]]: undefined,
  [[PromiseStatus]]: "pending"
}
```


A Promise is not the value - The value will eventually be available inside the promise.

What is a Promise?

```
readFileAsync('/luckyNumber.txt')
```

```
{
  [[PromiseValue]]: "42",
  [[PromiseStatus]]: "fulfilled"
}
```


Eventually the promise gets fulfilled and we can get the value.

Promise

```
const num = readFileAsync('/luckyNumber.txt')
```


Q: So, if I say `const number = readFilePromise('/luckyNumber.txt')`, what type of value is the “num” variable pointing to?

A: It’s not the contents of the luckyNumber file - it’s pointing to the promise that readFilePromise returns;

async/await

```
const num = await readFileAsync('/luckyNumber.txt')
```


It's the `await` keyword that makes sure that the code execution pauses until the promise is resolved. In this case, the `num` variable will point to the actual contents of the file.

async/await

```
async function getNumber() {
  const num = await readFileAsync('/luckyNumber.txt')
}
getNumber()
```


Finally, remember that the `await` keyword must ALWAYS happen inside a function marked as `async`.

async/await

```
const getNumber = async () => {
  const num = await readFileAsync('/luckyNumber.txt')
}
getNumber()
```


Also, remember that you can use the `async` keyword with regular functions and arrow functions as well…

Demo

A word on error handling...

Many async operations are related to talking to the outside world: database servers, external files etc…

Although you might never expect an error from an async operation such as `setTimeout`, anything that interacts with things outside the realm of JavaScript as subject to encounter failures and errors: The file might not exist or be corrupted, the database server might be offline. For that reason, it’s very common for async operations to throw errors - and we need to be prepared to properly handle those errors.

Try/Catch

```
function getLuckyGem(birthMonth) {
  const gems = ['Emerald', 'Amethyst', 'Jade', 'Opal', 'Sapphire', 'Perl',
                'Ruby', 'Agate', 'Diamond', 'Moonstone', 'Jasper', 'Onyx'];
  if (gems[birthMonth]) {
    return gems[birthMonth];
  } else {
    throw new Error('Invalid birth Month');
  }
}

try { // statements to try
  myGem = getLuckyGem(myMonth); // function could throw exception
}
catch (error) {
  myGem = 'unknown';
  console.error(error.message);
}
```


Try/Catch is an standard JavaScript Exception handling mechanism. In this example (that has nothing to do with async) it’s being used the get the user’s “lucky gem”.

Notice that in the “try” block we put the instructions that might throw an error.
If they do, the catch block will be executed and the `error` object is provided in the catch block.

Try/Catch

```
const getNumber = async () => {

  try {
    let num = await readFileAsync('/luckyNumber.txt')
    let success = await bookmaker.bet(num)
  } catch (error) {
    console.error(error.message)
  }

}

getNumber()
```


The same, existing try/catch exception handling can be used with async operations as well.