Asynchronous Programming

OR
Simulating Asynchrony With Synchrony

LIGHTHOUSE LABS

AGENDA

Callback Refresher

Sync vs Async

Interleaving

Demos

LIGHTHOUSE LABS

Callbacks a Refresher

Demo

LIGHTHOUSE LABS

Synchrony

LIGHTHOUSE LABS

JavaScript is Single Threaded

- JavaScript is single threaded!
- JavaScript thus has a synchronous runtime environment
  - Each line of code is executed one after another
  - JavaScript only has a single thread
  - No two pieces of code can execute simultaneously in a single process

Single Threaded randomDelay

Demo

LIGHTHOUSE LABS

Blocking Code

- Blocking code is code that **blocks other code from executing** while it executes
- In JavaScript code is *always blocking* unless explicitly designed for async

```
console.log("before");
someLongRunningFunction()
console.log("after");
```

LIGHTHOUSE LABS

Blocking Code

- Blocking code is code that **blocks other code from executing** while it executes
- In JavaScript code is *always blocking* unless explicitly designed for async

```
console.log("before");
someLongRunningFunction()
console.log("after");
```

LIGHTHOUSE LABS

Blocking Code

- Blocking code is code that **blocks other code from executing** while it executes
- In JavaScript code is *always blocking* unless explicitly designed for async

```
console.log("before");
someLongRunningFunction()
console.log("after");
```

LIGHTHOUSE LABS

Pseudo Asynchronous

Just Because We're not Asynchronous, Doesn't Mean We Can't Look Asynchronous

LIGHTHOUSE LABS

Pseudo Async

Primary Execution

```
function printShoppingCart() {
  console.log("Shopping Cart");
  for (const item of shoppingCart) {
     const product = products[item.productId];
  }
  console.log("End of Shopping Cart");
}
addProductToShoppingCart('pencil', 1);
console.log(shoppingCart);
printShoppingCart();
addProductToShoppingCart('charcoal', 1);
printShoppingCart();
```

Callback Queue

```
fx { … }
```

```
fx { … }
```

```
fx { … }
```

LIGHTHOUSE LABS

Pseudo Async

Primary Execution

Javascript will attempt to clear the Callback Queue whenever it can, by _interleaving_ callback functions in between the primary execution code.

Callback Queue

```
fx { ... }
```

```
fx { ... }
```

```
fx { ... }
```

LIGHTHOUSE LABS

Pseudo Async

Primary Execution

Before it invokes `addProductToShoppingCart`, it will look to see if there is any function on the fallback queue, and if so, invoke that first.

```
}
addProductToShoppingCart('pencil', 1);
console.log(shoppingCart);
printShoppingCart();
addProductToShoppingCart('charcoal', 1);
printShoppingCart();
```

Callback Queue

```
fx { … }
```

```
fx { … }
```

```
fx { … }
```

LIGHTHOUSE LABS

Pseudo Async

Primary Execution

So, before it invokes `addProductToShoppingCart` it will resolve the first function on the callback queue.

```
addProductToShoppingCart('pencil', 1);
console.log(shoppingCart);
printShoppingCart();
addProductToShoppingCart('charcoal', 1);
printShoppingCart();
```

Callback Queue

```
fx { … }
fx { … }
fx { … }
```

LIGHTHOUSE LABS

Pseudo Async

Primary Execution

And after it resolves that first callback, it will jump back to the primary execution, and invoke `addProductToShoppingCart`.

```
}
addProductToShoppingCart('pencil', 1);
console.log(shoppingCart);
printShoppingCart();
addProductToShoppingCart('charcoal', 1);
printShoppingCart();
```

Callback Queue

```
fx { … }
```

```
fx { … }
```

LIGHTHOUSE LABS

setTimeout && setInterval

Demo

LIGHTHOUSE LABS

setTimeout(callback, time)

- `setTimeout` will call your callback after `time` milliseconds
- JavaScript makes a best-effort to call the function on time, but…
- It may take longer!

Pseudo Async

Primary Execution

```
console.log("First");
setTimeout(() => {
  console.log("inCallback");
}, 0);
console.log("Second");
```

Callback Queue

LIGHTHOUSE LABS

Pseudo Async

Primary Execution

```
console.log("First");
setTimeout(() => {
  console.log("inCallback");
}, 0);
console.log("Second");
```

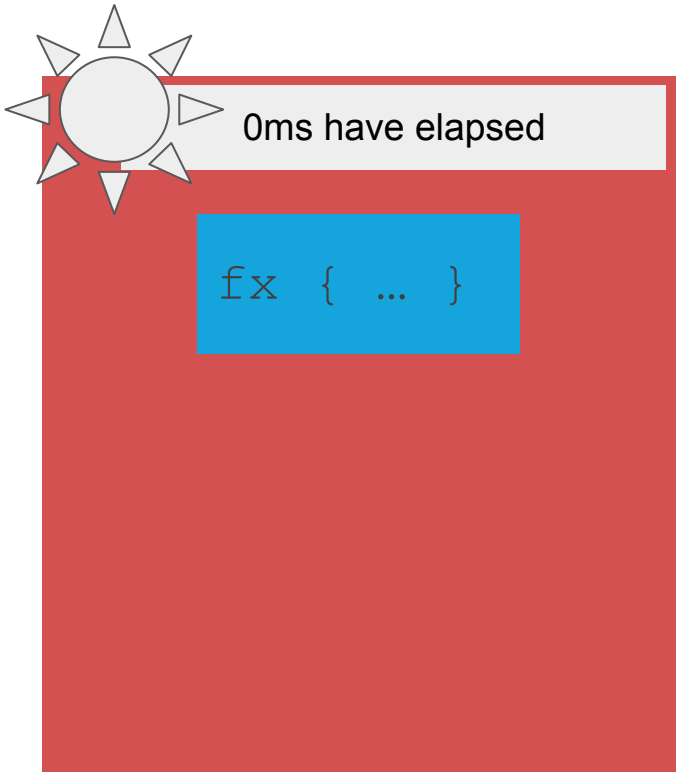
Callback Queue

LIGHTHOUSE LABS

Pseudo Async

Primary Execution

```
console.log("First");
setTimeout(() => {
  console.log("inCallback");
}, 0);
console.log("Second");
```



LIGHTHOUSE LABS

Pseudo Async

Primary Execution

```
console.log("First");
setTimeout(() => {
  console.log("inCallback");
}, 0);
console.log("Second");
```

Callback Queue

```
fx { ... }
```

LIGHTHOUSE LABS

Pseudo Async

Primary Execution

```
console.log("First");
setTimeout(() => {
  console.log("inCallback");
}, 0);
console.log("Second");
```

Callback Queue

No code remains in the Primary Execution Execution environment, and the Callback Queue is empty - so this program is DONE!

LIGHTHOUSE LABS

How Does This Work

*this is a simplification

```
setTimoeut(fx, 10);
```

This line of code was just invoked - the timeout has been setup, and we're about to begin execution of a bunch of code, while we wait for that timeout to occur.

LIGHTHOUSE LABS

How Does This Work

*this is a simplification

t0

Primary Execution

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFileSync('./file.txt', aLotOfData);
console.log('bar');
sum = sum - 100;
```

Callback Queue

LIGHTHOUSE LABS

How Does This Work

*this is a simplification

Primary Execution

t1

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFileSync('./file.txt', aLotOfData);
console.log('bar');
sum = sum - 100;
```

Callback Queue

LIGHTHOUSE LABS

How Does This Work

*this is a simplification

Primary Execution

t2

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFileSync('./file.txt', aLotOfData);
console.log('bar');
sum = sum - 100;
```

Callback Queue

LIGHTHOUSE LABS

How Does This Work

*this is a simplification

Primary Execution

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFileSync('./file.txt', aLotOfData);
console.log('bar');
sum = sum - 100;
```

t3

Callback Queue

LIGHTHOUSE LABS

How Does This Work

*this is a simplification

Primary Execution

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFileSync('./file.txt', aLotOfData);
console.log('bar');
sum = sum - 100;
```

t4

Callback Queue

LIGHTHOUSE LABS

How Does This Work

*this is a simplification

Primary Execution

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFileSync('./file.txt', aLotOfData);
console.log('bar');
sum = sum - 100;
```

t5

Callback Queue

LIGHTHOUSE LABS

How Does This Work

*this is a simplification

Primary Execution

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFileSync('./file.txt', aLotOfData);
console.log('bar');
sum = sum - 100;
```

t6

Callback Queue

LIGHTHOUSE LABS

How Does This Work

*this is a simplification

Primary Execution

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFileSync('./file.txt', aLotOfData);
console.log('bar');
sum = sum - 100;
```

t7

Callback Queue

LIGHTHOUSE LABS

How Does This Work

*this is a simplification

Primary Execution

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFileSync('./file.txt', aLotOfData);
console.log('bar');
sum = sum - 100;
```

t8

Callback Queue

LIGHTHOUSE LABS

How Does This Work

*this is a simplification

Primary Execution

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFileSync('./file.txt', aLotOfData);
console.log('bar');
sum = sum - 100;
```

t9

Callback Queue

LIGHTHOUSE LABS

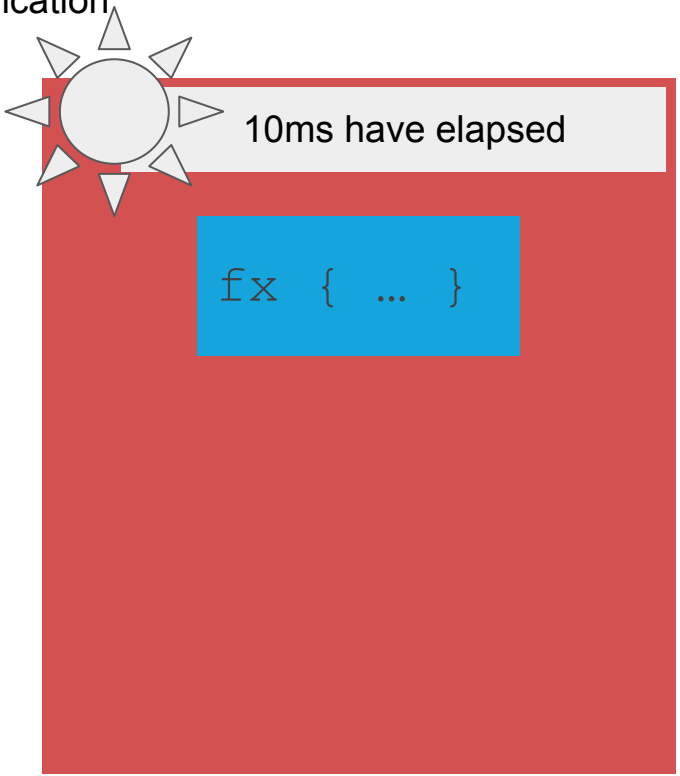
How Does This Work

*this is a simplification

Primary Execution

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFileSync('./file.txt', aLotOfData);
console.log('bar');
sum = sum - 100;
```

t10



LIGHTHOUSE LABS

How Does This Work

*this is a simplification

Primary Execution

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFileSync('./file.txt', aLotOfData);
console.log('bar');
sum = sum - 100;
```

t11

Callback Queue

```
fx { … }
```

LIGHTHOUSE LABS

How Does This Work

*this is a simplification

Primary Execution

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFileSync('./file.txt', aLotOfData);
console.log('bar');
sum = sum - 100;
```

t12

Callback Queue

```
fx { … }
```

LIGHTHOUSE LABS

How Does This Work

*this is a simplification

Primary Execution

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFileSync('./file.txt', aLotOfData);
console.log('bar');
sum = sum - 100;
```

t13

Callback Queue

```
fx { … }
```

LIGHTHOUSE LABS

How Does This Work

*this is a simplification

Primary Execution

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFileSync('./file.txt', aLotOfData);
console.log('bar');
sum = sum - 100;
```

t14

Callback Queue

```
fx { … }
```

LIGHTHOUSE LABS

How Does This Work

*this is a simplification

Primary Execution

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFileSync('./file.txt', aLotOfData);
console.log('bar');
sum = sum - 100;
```

t15

Callback Queue

```
fx { … }
```

LIGHTHOUSE LABS

How Does This Work

*this is a simplification

Primary Execution

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFileSync('./file.txt', aLotOfData);
console.log('bar');
sum = sum - 100;
```

Callback Queue

t16 → `fx { ... }`

LIGHTHOUSE LABS

How Does This Work

*this is a simplification

Primary Execution

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFileSync('./file.txt', aLotOfData);
console.log('bar');
sum = sum - 100;
```

t17

Callback Queue

LIGHTHOUSE LABS

JavaScript simulates asynchronous with interleaving

- JavaScript opportunistically interleaves the execution of callbacks with normal code
- This simulates multi-threaded behaviour
- Long running synchronous code will block the execution of asynchronous code

Filesystem calls

`require('fs')`

Demo

LIGHTHOUSE LABS

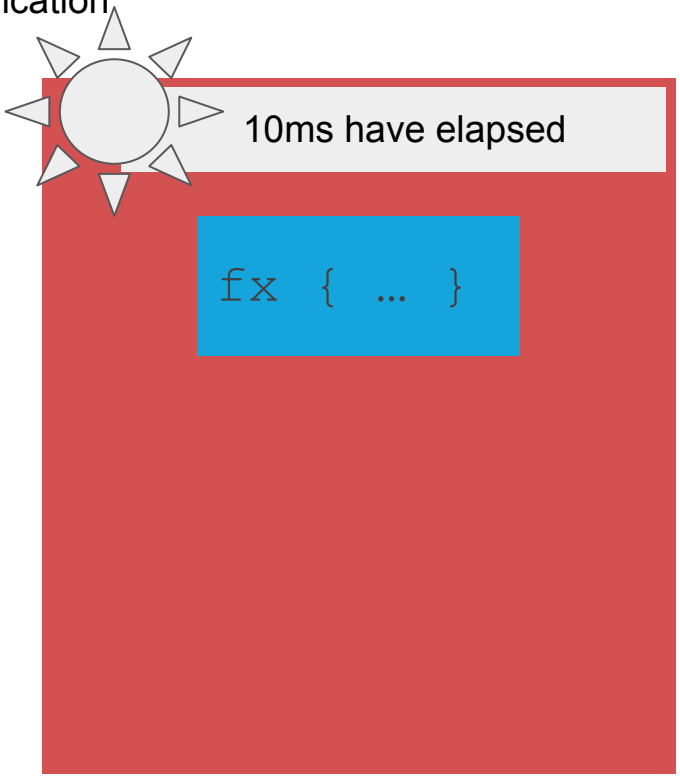
How Does This Work

*this is a simplification

Primary Execution

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFile('./file.txt', data, () => {
  console.log('bar');
  sum = sum - 100;
});
```

t10



LIGHTHOUSE LABS

How Does This Work

*this is a simplification

Primary Execution

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFile('./file.txt', data, () => {
  console.log('bar');
  sum = sum - 100;
});
```

Callback Queue

t11 → `fx { ... }`

LIGHTHOUSE LABS

How Does This Work

*this is a simplification

Primary Execution

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFile('./file.txt', data, () => {
  console.log('bar');
  sum = sum - 100;
});
```

t12

Callback Queue

LIGHTHOUSE LABS

How Does This Work

*this is a simplification

Primary Execution

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFile('./file.txt', data, () => {
  console.log('bar');
  sum = sum - 100;
});
```

t13

Callback Queue

LIGHTHOUSE LABS

How Does This Work

*this is a simplification

Primary Execution

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFile('./file.txt', data, () => {
  console.log('bar');
  sum = sum - 100;
});
```

t14

Callback Queue

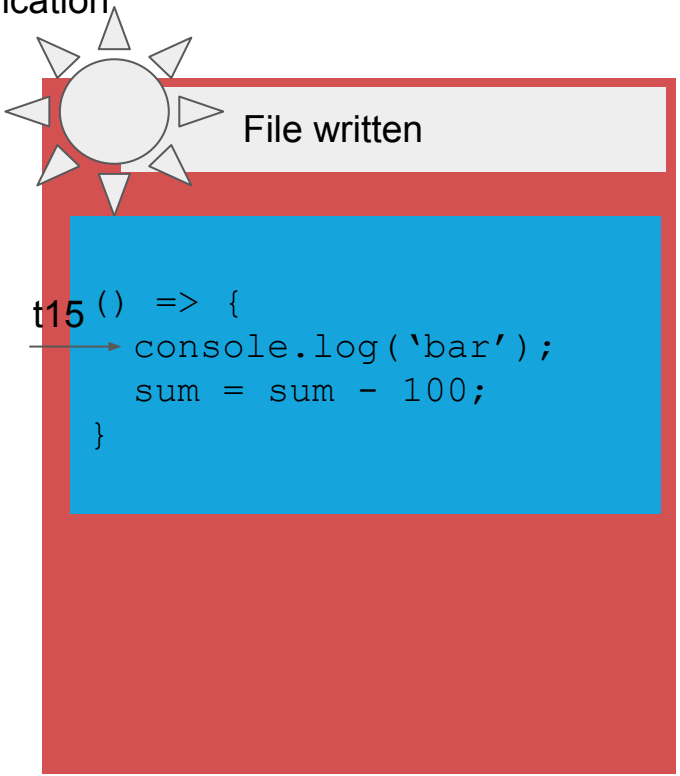
LIGHTHOUSE LABS

How Does This Work

*this is a simplification

Primary Execution

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFile('./file.txt', data, () => {
  console.log('bar');
  sum = sum - 100;
});
```



LIGHTHOUSE LABS

How Does This Work

*this is a simplification

Primary Execution

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
    sum += b;
}
fs.writeFile('./file.txt', data, () => {
  console.log('bar');
  sum = sum - 100;
});
```

Callback Queue

t16

```
() => {
  console.log('bar');
  sum = sum - 100;
}
```

LIGHTHOUSE LABS

Sum an Array

Demo

LIGHTHOUSE LABS

Questions?

LIGHTHOUSE LABS