

# Asynchronous Programming

OR  
Simulating Asynchrony  
With Synchrony

# AGENDA

Callback Refresher

Asynchronous

Interleaving

Demos

# Callbacks a Refresher

Demo



# Synchrony

# 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

# 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");`

# 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");
```



# 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");
```

# Pseudo Asynchronous

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

# 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 { ... }

# 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 { ... }
```

# 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 { ... }
```

# 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 { ... }

# 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 { ... }
```

# setTimeout && setInterval

Demo



# 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!

# How Does This Work

\*this is a simplification

```
cc  
cc  
le  
fo  
}  
fs  
cc  
su  
  
setTimeout(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.

# How Does This Work

\*this is a simplification

t0

Primary Execution

Callback Queue

```
→ 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;
```

# How Does This Work

\*this is a simplification

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;
```

Primary Execution

Callback Queue

# 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

# How Does This Work

\*this is a simplification

## Primary Execution

```
const arr = [1, 2, 3];  
console.log('foo');  
let sum = 0;  
t3 → for(const b in arr) {  
    sum += b;  
}  
fs.writeFileSync('./file.txt', aLotOfData);  
console.log('bar');  
sum = sum - 100;
```

## Callback Queue

# How Does This Work

\*this is a simplification

## Primary Execution

```
const arr = [1, 2, 3];  
console.log('foo');  
let sum = 0;  
t4 → for(const b in arr) {  
      sum += b;  
}  
fs.writeFileSync('./file.txt', aLotOfData);  
console.log('bar');  
sum = sum - 100;
```

## Callback Queue

# How Does This Work

\*this is a simplification

## Primary Execution

```
const arr = [1, 2, 3];  
console.log('foo');  
let sum = 0;  
t5 → for(const b in arr) {  
    sum += b;  
}  
fs.writeFileSync('./file.txt', aLotOfData);  
console.log('bar');  
sum = sum - 100;
```

## Callback Queue



# How Does This Work

\*this is a simplification

## Primary Execution

```
const arr = [1, 2, 3];  
console.log('foo');  
let sum = 0;  
t6 → for(const b in arr) {  
      sum += b;  
}  
fs.writeFileSync('./file.txt', aLotOfData);  
console.log('bar');  
sum = sum - 100;
```

## Callback Queue

# How Does This Work

\*this is a simplification

## Primary Execution

```
const arr = [1, 2, 3];  
console.log('foo');  
let sum = 0;  
t7 → for(const b in arr) {  
    sum += b;  
}  
fs.writeFileSync('./file.txt', aLotOfData);  
console.log('bar');  
sum = sum - 100;
```

## Callback Queue

# How Does This Work

\*this is a simplification

## Primary Execution

```
const arr = [1, 2, 3];  
console.log('foo');  
let sum = 0;  
t8 → for(const b in arr) {  
      sum += b;  
}  
fs.writeFileSync('./file.txt', aLotOfData);  
console.log('bar');  
sum = sum - 100;
```

## Callback Queue

# How Does This Work

\*this is a simplification

## Primary Execution

```
const arr = [1, 2, 3];  
console.log('foo');  
let sum = 0;  
t9 → for(const b in arr) {  
    sum += b;  
}  
fs.writeFileSync('./file.txt', aLotOfData);  
console.log('bar');  
sum = sum - 100;
```

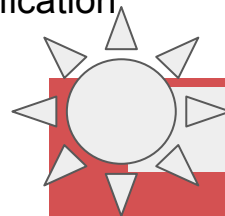
## Callback Queue

# How Does This Work

\*this is a simplification

```
const arr = [1, 2, 3];
console.log('foo');
let sum = 0;
for(const b in arr) {
  sum += b;
}
t10 → fs.writeFileSync('./file.txt', aLotOfData);
console.log('bar');
sum = sum - 100;
```

Primary Execution



10ms have elapsed

fx { ... }

# 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;  
}  
t11 → fs.writeFileSync('./file.txt', aLotOfData);  
console.log('bar');  
sum = sum - 100;
```

## Callback Queue

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

# 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;
}
t12 → fs.writeFileSync('./file.txt', aLotOfData);
console.log('bar');
sum = sum - 100;
```

**Callback Queue**

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

# 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;  
}  
t13 → fs.writeFileSync('./file.txt', aLotOfData);  
console.log('bar');  
sum = sum - 100;
```

**Callback Queue**

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



# 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;
}
t14 → fs.writeFileSync('./file.txt', aLotOfData);
console.log('bar');
sum = sum - 100;
```

**Callback Queue**

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

# 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;
}
t15 → fs.writeFileSync('./file.txt', aLotOfData);
console.log('bar');
sum = sum - 100;
```

**Callback Queue**

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

# How Does This Work

\*this is a simplification

```
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;
```

Primary Execution

Callback Queue

t16



fx { ... }

# 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;  
}  
t17 fs.writeFileSync('./file.txt', aLotOfData);  
→ console.log('bar');  
sum = sum - 100;
```

## Callback Queue

# 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')`

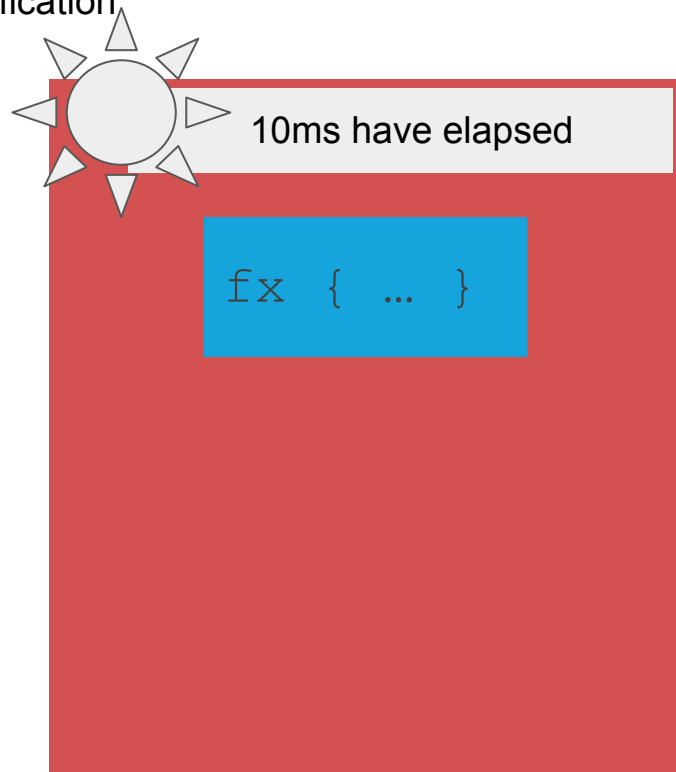
## Demo

# How Does This Work

\*this is a simplification

```
const arr = [1, 2, 3];  
console.log('foo');  
let sum = 0;  
for(const b in arr) {  
  sum += b;  
}  
t10 → fs.writeFile('./file.txt', data, () => {  
  console.log('bar');  
  sum = sum - 100;  
});
```

Primary Execution



# How Does This Work

\*this is a simplification

```
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;  
});
```

Primary Execution

Callback Queue

t11



fx { ... }



# How Does This Work

\*this is a simplification

```
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;
});
```

Primary Execution

Callback Queue

t12



# How Does This Work

\*this is a simplification

```
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;
});
```

Primary Execution

Callback Queue

t13



# How Does This Work

\*this is a simplification

```
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;
});
```

Primary Execution

Callback Queue

t14

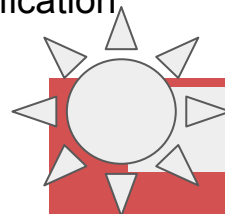


# How Does This Work

\*this is a simplification

```
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;
});
```

Primary Execution



File written

```
t15 () => {
  console.log('bar');
  sum = sum - 100;
}
```

# How Does This Work

\*this is a simplification

```
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;
});
```

Primary Execution

t16 →

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

Callback Queue

# Sum an Array

Demo

# Questions?

