Code Fellows

Class 03
Async

seattle-javascript-401n14

```
3  class Validator {
4
5    constructor(schema) {
6      this.schema = schema;
7    }
8
9    isString(input) { return typeof input === 'string'; }
10
11   isObject(input) { return typeof input === 'object' && !(input
12
13   isArray(input, valueType) {
14     return Array.isArray(input) && (valueType ? input.every( v
       true);
15   }
16
17   isBoolean(input) { return typeof input === 'boolean'; }
18
19   isNumber(input) { return typeof input === 'number'; }
20
21   isFunction(input) { return typeof input === 'function'; }
22
23   isTruthy(input) { return !!input; }
24
25   isCorrectType(input, field) {
26     switch(field.type) {
```

Lab 02 Review

Vocab Review!

What is an object?

What is an object?

Also called an **instance**, an object is a basic **unit of data/code** that we use in our program. It can be a variable, a **data structure**, a function, etc. Objects usually have a **type** that defines what kind of data and operations that object can do.

What is a class?

What is a class?

A class is a template for the structure/**type** of an object. Classes define what kind of data an object should hold and what kind of operations you can do on an object.

What is inheritance?

What is inheritance?

Inheritance is the idea that we can make new classes based off of an existing class, building on top of it. We can define a class that inherits from another class by using the `extends` keyword. Using the function `super()`, we can access the parent class **constructor**.

```
class Dog extends Animal { }
```

What does `super()` do?

What does `super()` do?

If you are using a class that `extends` a parent class, calling `super()` will call the parent class constructor. We typically want to do this before adding our own constructor code.

What is `this`?

What is `this`?

`this` is a keyword in JavaScript used to refer to the object or entity that the current running code is within. What `this` ends up being equal to depends on the **context** that the code is executed in.

What is `Object.prototype`?

What is `Object.prototype`?

All JavaScript objects can be thought of as originating from a class. All JavaScript objects **inherit** from `Object.prototype`. By using `.prototype` on variables or functions, we can hook into the constructor for that object and change it.

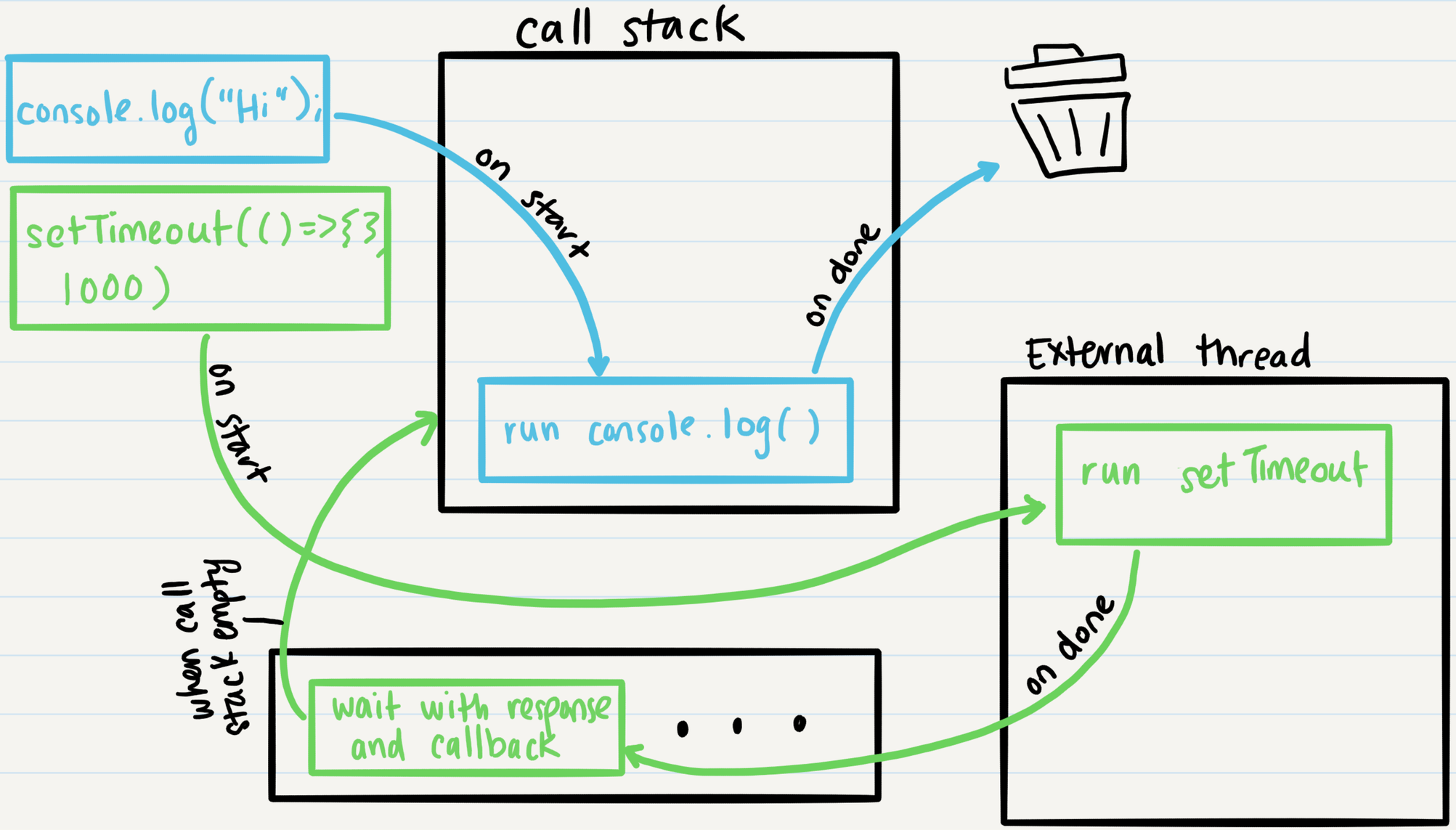
What is a factory function?

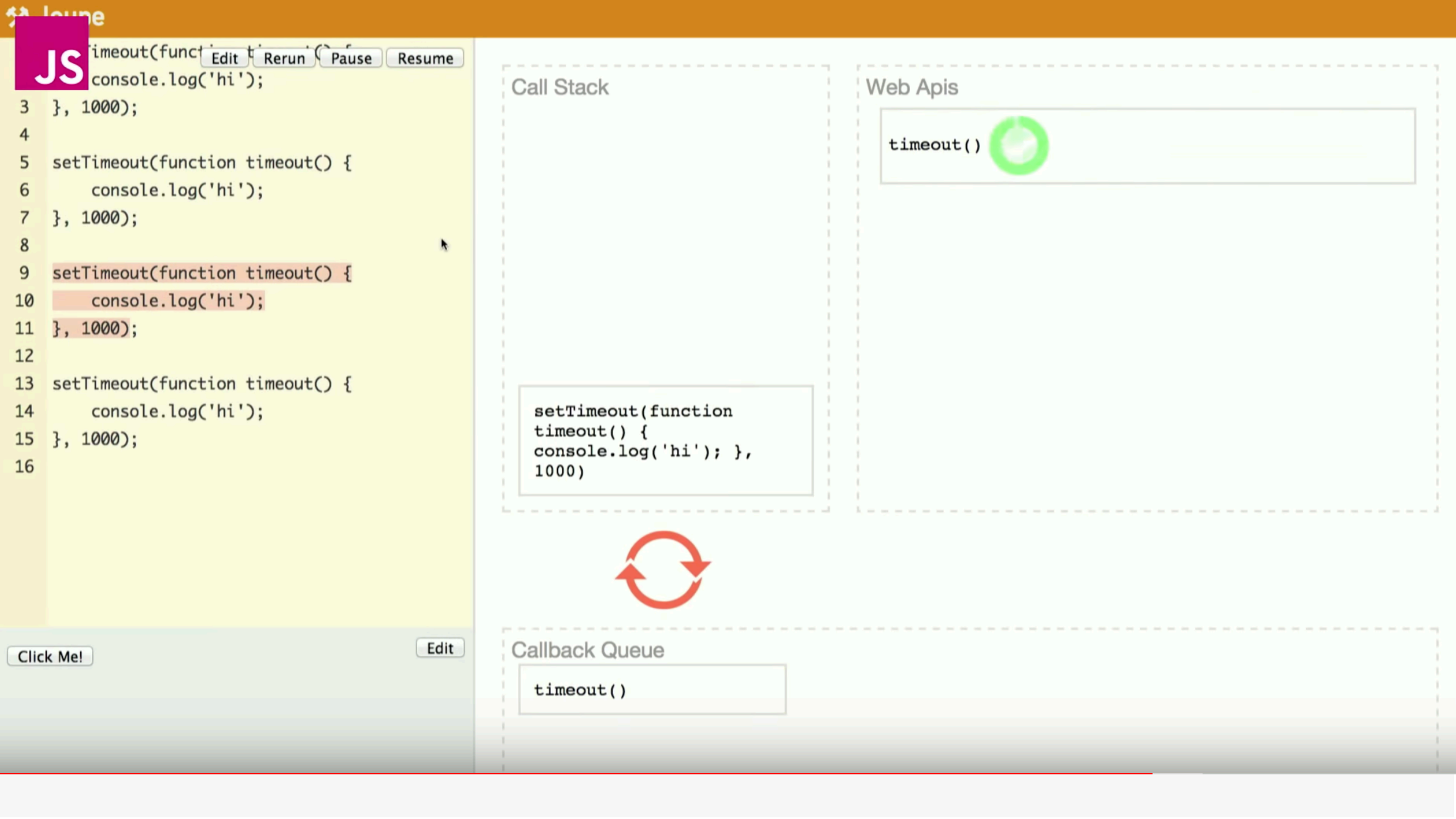
What is a factory function?

A factory function is a function that returns a new object, but is not a class or a constructor. We've made a lot of factory functions before! Now we want to move to using classes.

How Does a Program Run?

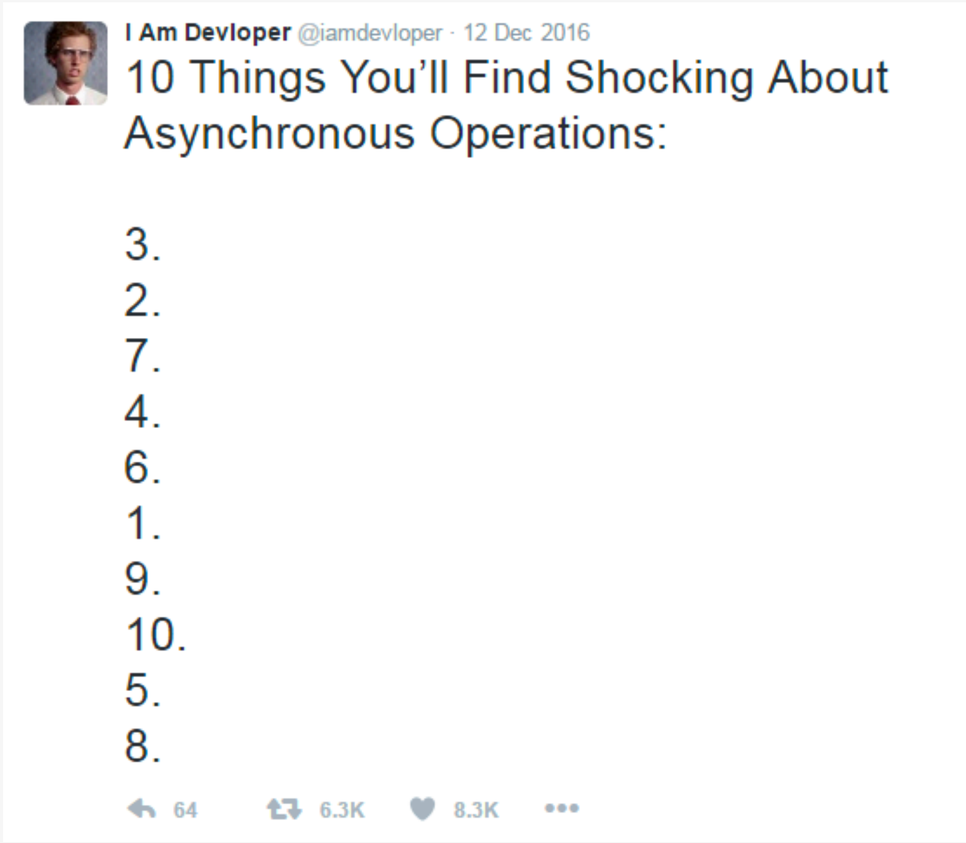
- All applications use a **call stack** to manage running the program
- Calls to a function are added to the stack, and then removed or popped off the stack when they complete
- Items in the stack are interpreted in order, so if something takes a long time, it holds up the execution of other lines
- Infinite loops, never-ending recursive functions, etc, add too much to the stack too quickly, causing a **stack overflow**, or the **maximum call stack limit** to be hit


call stack
console.log("Hi");
setTimeout(()=>{},
1000)
on start
on start
on done
run console.log( )
External thread
run setTimeout
on done
when call stack empty
wait with response and callback


Edit
Rerun
Pause
Resume
timeout(func
console.log('hi');
}, 1000);
setTimeout(function timeout() {
console.log('hi');
}, 1000);
setTimeout(function timeout() {
console.log('hi');
}, 1000);
setTimeout(function timeout() {
console.log('hi');
}, 1000);
Call Stack
Web Apis
timeout()
setTimeout(function timeout() { console.log('hi'); }, 1000)
Callback Queue
timeout()
Click Me!
Edit

What is Async?

- **Asynchronous code** is code that doesn't interrupt your program from continuing
- It's useful when we want to do something that might take an indeterminate amount of time (API calls, reading a file, showing an animation)
- The important part of async is figuring out when the async is done so you can do something else (save data, edit file, queue next animation)



So How Do We Do Async?

- There are a few methods to kick off an async process, the important thing is doing something when that async process is done
- **Callbacks** - After you do an async thing, call this function
- **Promises** - Save the async process as an object, add responses to it dynamically
- **Async/Await** - Make async look more sync!

```
function apiCall(str) {
  let randomNumber = Math.floor(Math.random() * 1000) + 1;

  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log(str);
      resolve({ data: str });
    }, randomNumber);
  });
}
```

```
soniakandah … class-03 demo async-concepts master + 3 node callbacks.js
Async A callback
Async X
Async Z
Async Y
Async B callback
Async C callback
Done!
soniakandah … class-03 demo async-concepts master + 3 node promises.js
Async A Promise
{ data: 'Async A Promise' }
Async B Promise
{ data: 'Async B Promise' }
Async C Promise
```

Demo

class-01/ demo/async-concepts

Callbacks is the old way to "do something after async". Promises improve upon callbacks, getting rid of callback hell. `async`/`await` is an even nicer way to make async understandable.


CALLBACKS
PROMISES
ASYNC/AWAIT
imgflip.com

Error-first Callbacks

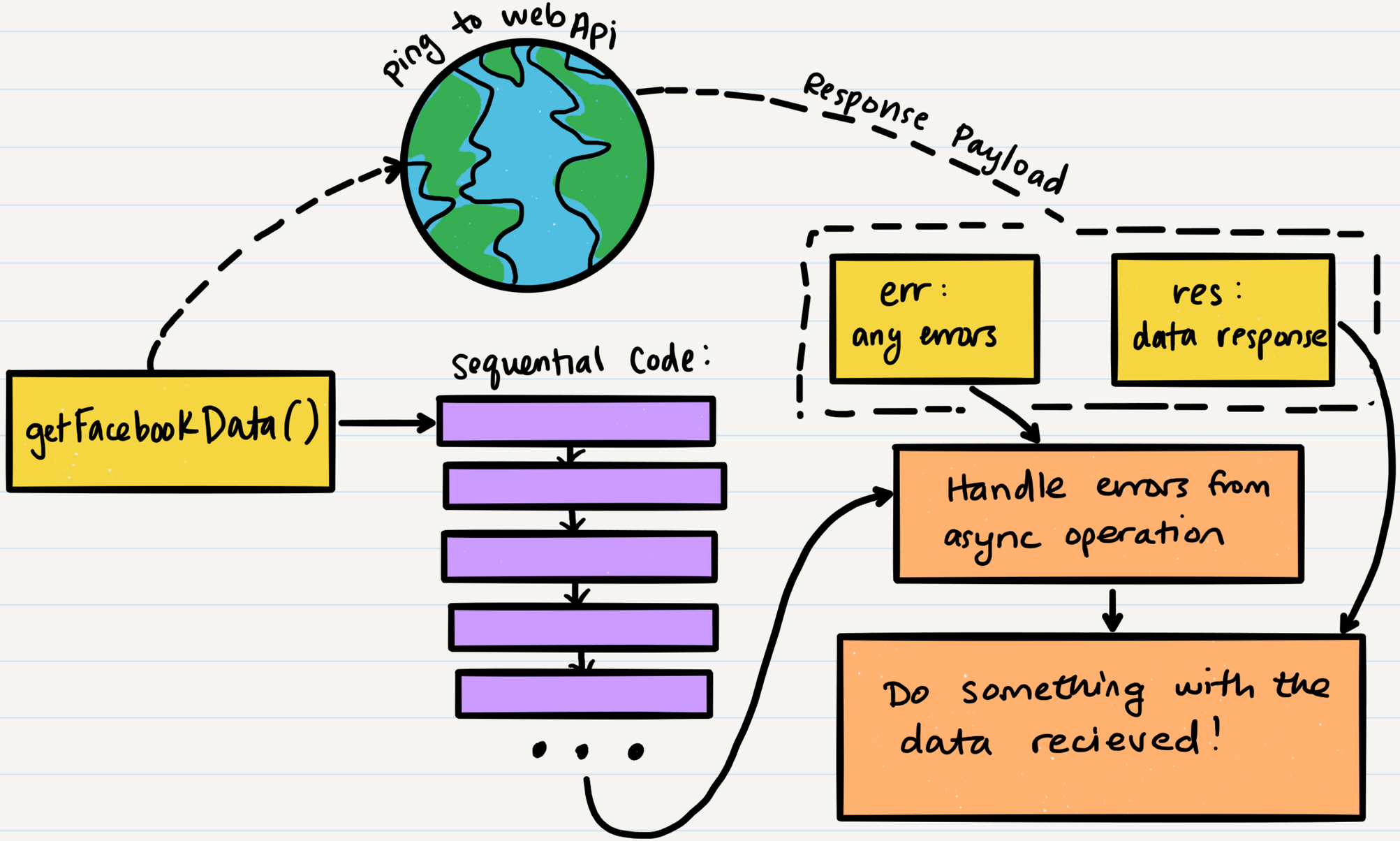
- We usually use async operations for api calls
- When we do, we should always handle errors first, response later (the response might not exist!)
- Because of this, we usually structure our callbacks like:

```
(err, res) => {

  if(err) { // handle error }

  else if (res) { // save response data }

}
```


Ping to web Api
Response Payload
err :
any errors
res :
data response
Sequential Code:
getFacebookData()
Handle errors from
async operation
Do something with the
data recieved!

Buffers and Files

- A **buffer** usually refers to a **stream** of binary data
- Node has a global class `Buffer` which lets us do cool operations on "**raw data**"
- A common operation of programs is to read and write to files on the computer
- Node has a module called `fs` (stands for **file system**) which lets us do a lot of cool operations on files

```
3   let data = Buffer.from("Bald!");
4   console.log(data);
5
6   // This is the buffer turned back into a UTF-8 string
7   console.log(data.toString());
8
9   // This is the buffer turned into a hex string (notice the numbers match the buffe
10  console.log(data.toString("hex"));
11
12  // This is the first byte of the buffer, in DECIMAL
13  console.log(data[0]);
14
15  // Change second byte of the buffer to the letter o
16  data[1] = 111;
17
18  // Dig ... the new word
19  console.log(data.toString());
```

```
soniakandah … class-03 demo buffers master + 5 node buffer.js
<Buffer 42 61 6c 64 21>
Bald!
42616c6421
66
Bold!
```

Demo

class-01/ demo/buffer

Buffers are streams of data that can be interpreted in many ways (buffers save their data as bytes and don't enforce a type). Typically, a buffer is a **temporary** place to store data.

Command Line Arguments

- We can often save buffers and files by getting that data as a command line argument when the program is run:

```
node index.js myfile.txt
```

- We can save a command line argument by accessing the global variable `process.argv`
- You can pass multiple kinds of command line arguments!

More about Files

- `fs` is included as a part of Node, but you do have to `require('fs')` in order to load the functionality into your app
- Every function has an asynchronous (default) and synchronous (adds "`Sync`" to end of function) form
- You can `.readFile`, `.writeFile`, `.copyFile`, and much more!

```
5   const fs = require("fs");
6   const util = require("util");
7
8   function read(file, onDone) {
9       // We want to read the file, and then
10      // do some error-first handling of the response
11
12      fs.readFile(file, (err, data) => {
13          if (err) onDone(err);
14          else onDone(undefined, data.toString().trim());
15      });
16  }
17
18  // What if we wanted to export this as a promise instead?
19  // fs was designed for a callback structure,
20  // but we can force it to return a promise using
```

```
soniakandah … class-03 demo file-reader-module master + 5 npm start

> file-reader@1.0.0 start /Users/soniakandah/cf/js-401n14/curriculum/class-03/demo/
> node index.js

FOO
```

Demo

class-01/ demo/file-reader-module

The file system class helps make it easier for us to read and write to files.

While the file system functions are designed for callbacks, you can use `promisify` to make them more like Promises.

Mocks

- When you have a complicated application with a lot of packages and api calls, you might not want to test all of that every time you run `npm test`
- Packages you install via `npm install` will ideally already be tested by their developers, so testing them again is redundant
    - Instead, we mock out the input and output of these modules
- Jest will do some work for you for some node modules, provided you have a root directory `__mocks__` folder. Otherwise, you run `jest.mock(<module name>)` in your test file

Example Mock

- Input parameters are of the same number and format: `(file, cb)` for readFile
- Instead of thinking of `file` as an actual full file that needs to be read, we **hard-code** it as a `Buffer`
- We have the same return / end results as the original function, just with some fudged content

```
exports.readFile = (file, cb) => {
    if (file.match(/bad/i)) {
        cb("Invalid File");
    } else {
        cb(undefined, new Buffer("File Conter
    }
};
```

```
exports.writeFile = (file, buffer, cb) =>
  if (file.match(/bad/i)) {
    cb("Invalid File");
  } else {
    fileContents = buffer;
    cb(undefined, true);
  }
};
```

What's Next:

- Due by Midnight tonight: **Learning Journal 03**
- Due by Midnight Sunday:
    - **Career: Professional Etiquette** and **Career: Accountability Partners**
    - **Feedback Survey Week #2**
- Due by Midnight Monday: **Code Challenge 03**
    - **We're going to do this in pairs!**
- Due by 6:30pm Tuesday:
    - **Lab 03**
    - **Read: Class 04**
- Next Class: **Class 04 - Data Modeling**

Questions?