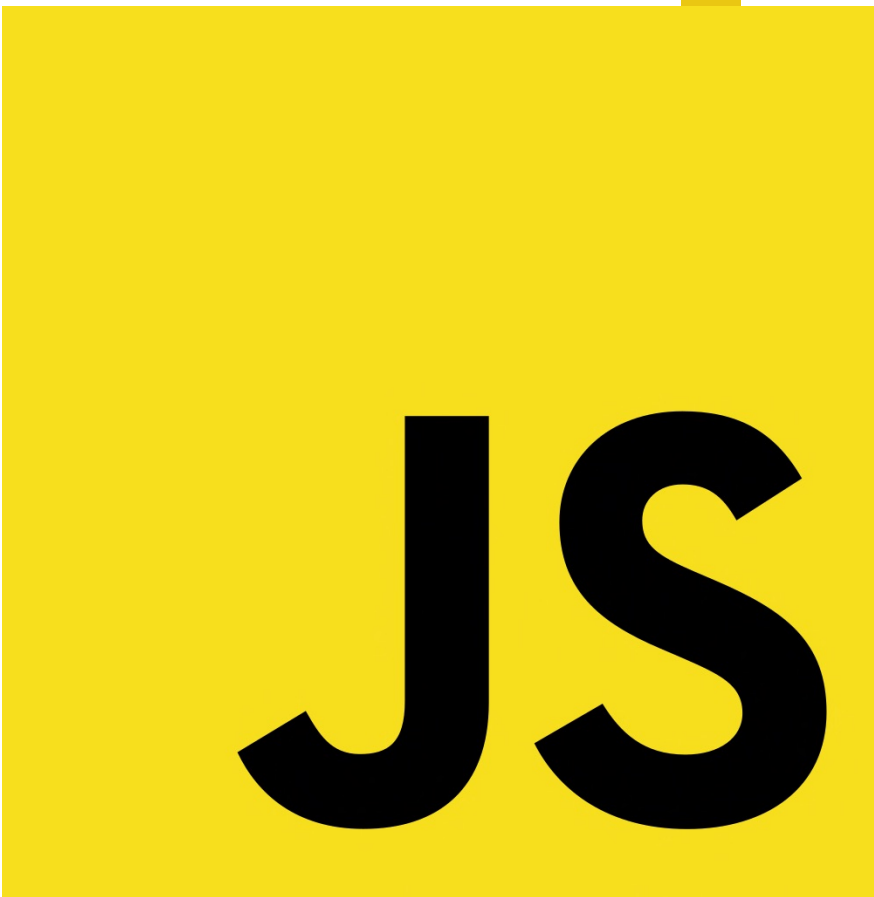
Frontend Masters

JS the Hard Parts



How does JS work

```
let num = 3
function multiplyBy2(inputNumber){
 const result = inputNumber * 2;
 return result;
}
 const output = multiplyBy2(num);
 const newOutput = multiplyBy2(10);
```

We need two things to execute code
1. the thread of execution
2. a place to store data

function v Global;
functions are like mini programs, they have an execution context

thread of execution

```
let num = 3
function multiplyBy2(inputNum)
const output = multiplyBy2(num)
```

EXECUTION CONTEX

Local Execution	Local memory
multiplyBy2(num)	inputNumber : 3
const result = inputNumber * 2	result : 6
return result	

```
Const newoutput = multiplyBy2(10)
```

EXECUTION CONTEXT

Local execution	Local memory
multiplyBy2(10)	inputNumber : 10
const result = inputNumber * 2	result : 20
return result	

Memory

num : 3
multiplyBy2 : -> F ->
output : uninitilized until execution context
output : 6
newoutput : uninitilized until execution context
newOutput : 20

JS keeps track of the thread of execution w/ the Call Stack

The call stack is a Data structure - first IN, last out: We're only engaged w/ the top

In local scope,

The label = Paramaters
The value = Arguments

Local Memory

Paramater	Argument
InputNumber	3

We define a function w/ paramaters
We call a function w/ Arguments

The 3 parts of JavaScript

1. A single thread of execution
2. Memory to store data
3. A callstack to keep track of the thread of execution

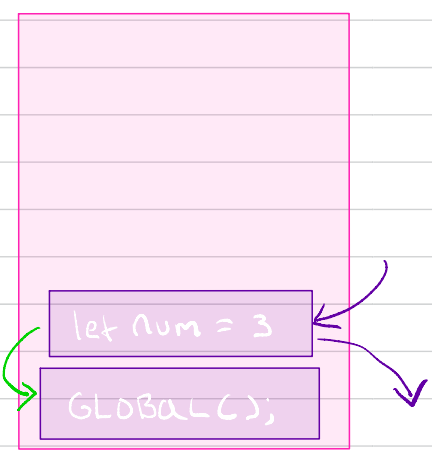
The Callstack

first in, last out STACK data structure;

the very first thing called is global()

We can think of global as a function

The call stack is how we keep track of our single thread of execution



Execution Contexts get pushed to the Callstack

Why Even Have functions?

```
function tenSquared( )
    return 10 * 10;
}

function nineSquared ( )
    return 9 * 9 ;
}
```

DRY

We're repeating ourselves

```
function SquareNum(num){
    return num * num
}
```

generalize the input,
dynamically generate data

DRY

don't
Repeat
yourself

Higher Order functions

Functions that operate on other functions, either by taking them as arguments or by returning them, are called higher-order functions

We Can NOT repeat ourselves w/ Higher order functions

```
function copyAndManipulate(array, instructions){
    const output = []
    for (let i = 0; i < array.length; i++){
      output.push(instructions(array[i]));
    }
    return output;
}

function multiplyBy2(input){
    return input * 2
}

const result = copyAndManipulate([1, 2, 3], multiplyBy2)
```

We can pass a function as an Argument to another function

This is essentially map. passing 1 function as an argument to another one

```
const mapped = array.map((element) => {
  return element * 2})
```

```
function copyAndManipulate(array, instructions)
    const output = []
  for (let i = 0; i < array.length; i++){
    output.push(instructions(array[i]));
  }
  return output;
}
function multiplyBy2(input){
  return input * 2
}

const result = copyAndManipulate([1, 2, 3], multiplyBy2)
```

GLOBAL EXEC.

Result = copyAndManipulate([1,2,3], multiplyBy2)

• EXECUTION CONTEXT •

array: [1,2,3]

instructions: multiplyBy2 → [f] →

output: [2,4,6]

for loop

Array [1,2,3]

multiplyBy2(1)

EXECUTION CONTEXT

1 * 2

input: 1

Return 2

Output: [2,]

for loop

Array [1,2,3]

multiplyBy2(2)

EXECUTION CONTEXT

2 * 2

input: 1

Return 4

Output: [2,4,...]

GLOBAL MEMORY

copyAndManipulate: → [f] →

multiplyBy2: → [f] →

result: [2,4,6]

once completed, its removed

CALLSTACK

multiplyBy2()

copyAndManipulate()

GLOBAL();

Hows This Possible?

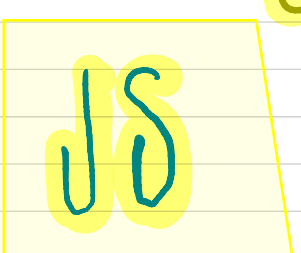
In JavaScript, functions can be passed as Arguments into other functions

> First Class functions
> Note, the functions are pointers in local to their global place in memory

functions can co-exist & Be treated As any other Javascript Object

1. Assigned to variables & properties of other objects
2. Passed As Arguments
3. Returned As values from functions

CLOSURE



How does returning a variable as a value = closure?

Functions in Javascript = first class objects

They Have all the features of objects

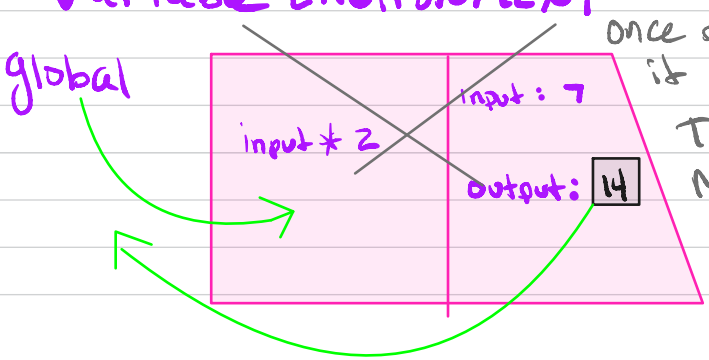
```
function copyAndManipulate(array, instructions){
    const output = []
    for (let i=0; i>array.length; i++){
      output.push(instructions(array[i]));
    }
    return output;
}

function multiplyBy2(input){
    return input*2
}

const result = copyAndManipulate([1,2,3], multiplyBy2)
```

Bad Arrow

↑ higher order function

Higher Order functions

STATE = EXECUTION CONTEXT = Variable ENVIRONMENT



once run, we delete it out.

This is State Management

arrow functions =>

function multiplyBy2(input) { return input * 2 }

Improve legability of the code, we write less code, However, we lose some readibility.

There is one under the hood change.

Arrow functions change the THIS keyword

Assignment

Returning A Function from Another function

Nested functions are the key to closure; But why even do it?

CLOSURE

- The most esoteric JS concept
- Enables powerful pro-level JS features
  - 'Once'
  - 'Memoize'
- Many JS design patterns (module) use closure
- Build iterators, handle partial application & maintain state in asyncronus world.

When we execute a function,

it creates an execution context,

Once completed, it all gets deleted.

What if, instead of deleting our data out of the execution context, we could persist state?

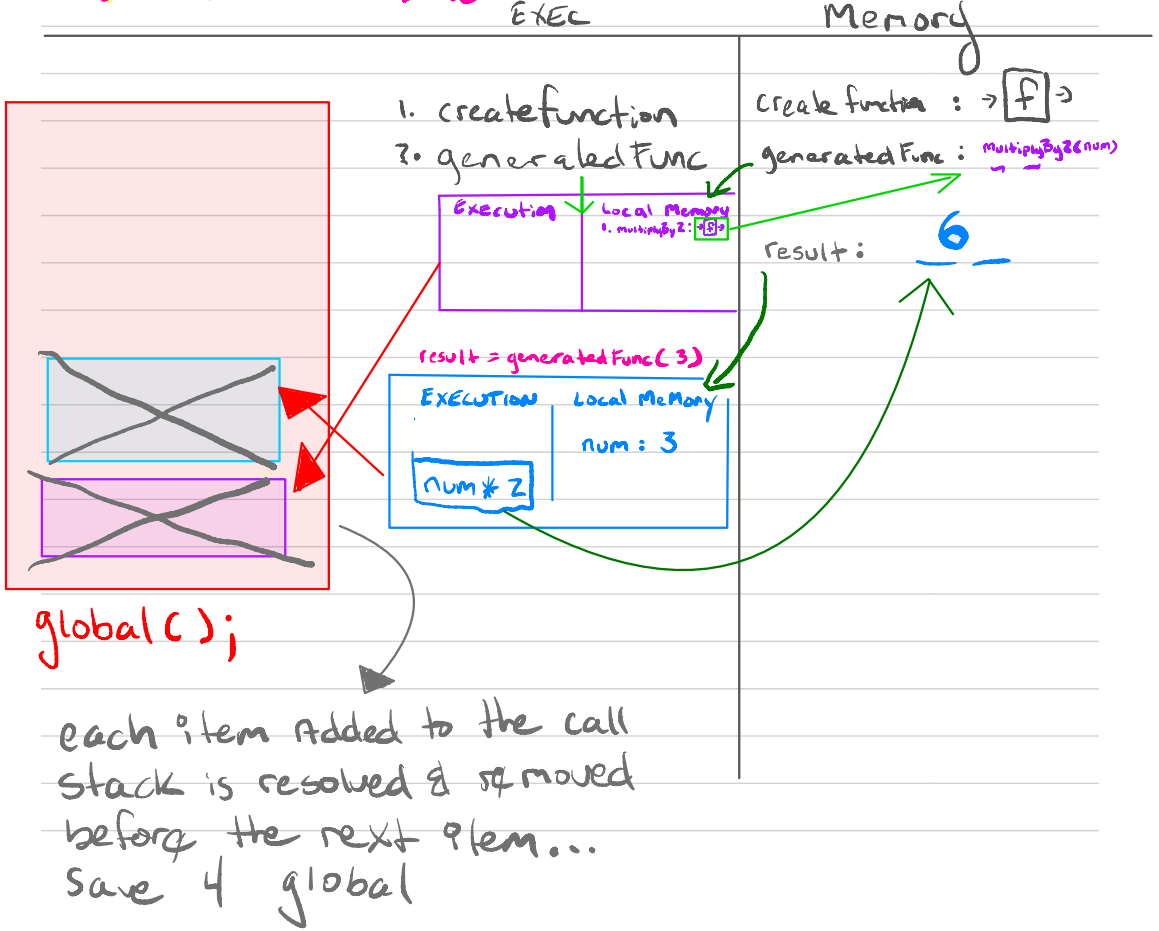
But it all starts w/ us returning a function from another function

Nested Functions: JS the hard parts

```
1. function createFunction( ) {
     function multiplyBy2( nums ){
       return nums * 2;
     }
     return multiplyBy2;
   }
   Const generatedFunc = createFunction( );
2. Const result = generateFunc(3)
```

1. storing createFunction in global memory
2. define the const generatedFunction in global memory.
   ↳ its value is uninitialized.
3. generate an execution context, call create function
4. we dont execute multiplyBy2, we return it to the const generated func
   We're replacing generateFunc( ) w/ multiplyBy2
5. execution context gets removed.
6. Define a const result, results value is the return value from running createFunc
   At this point, generateFunc has NOTHING TO DO w/ createFunc.

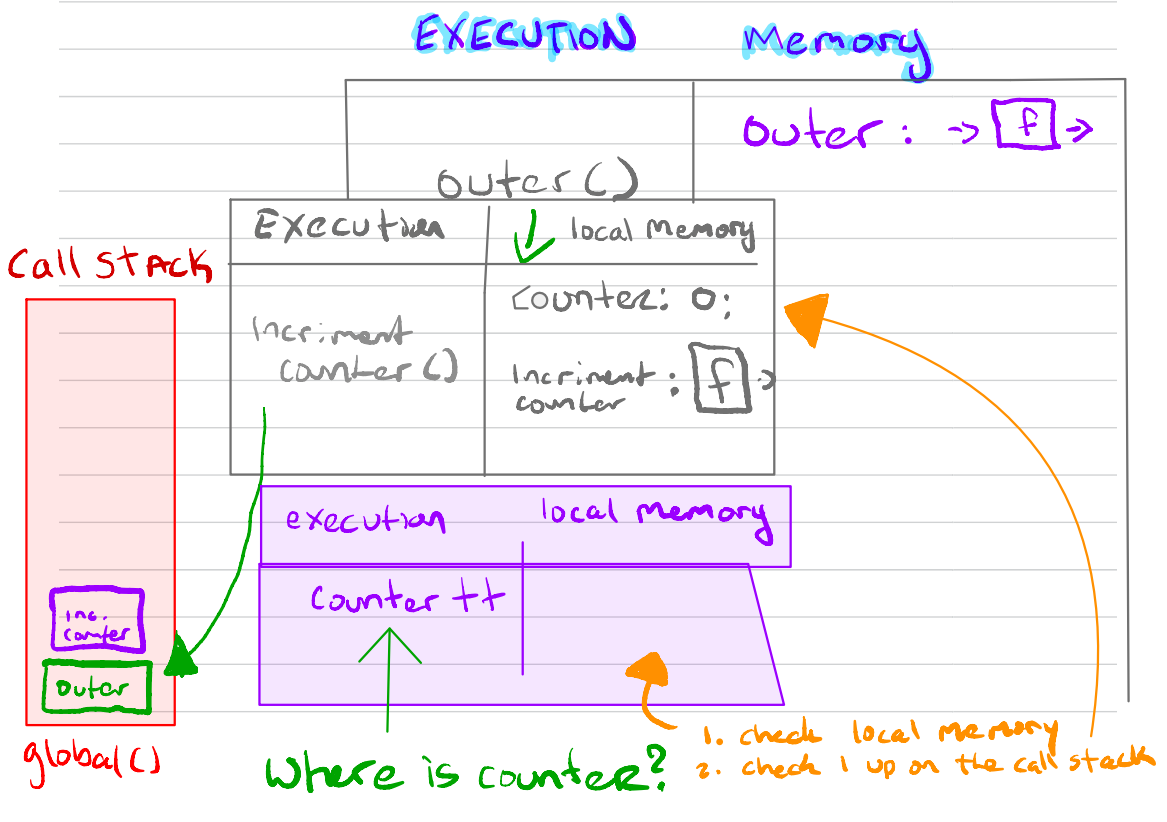
DEADBEAT DAD - creates it & is Gone



each item added to the call stack is resolved & removed before the next item...
Save 4 global

```
function outer( );
    let counter = 0;
    function incrimentCounter( ){
        counter ++;
    }
    incrementCounter();
outer();
```

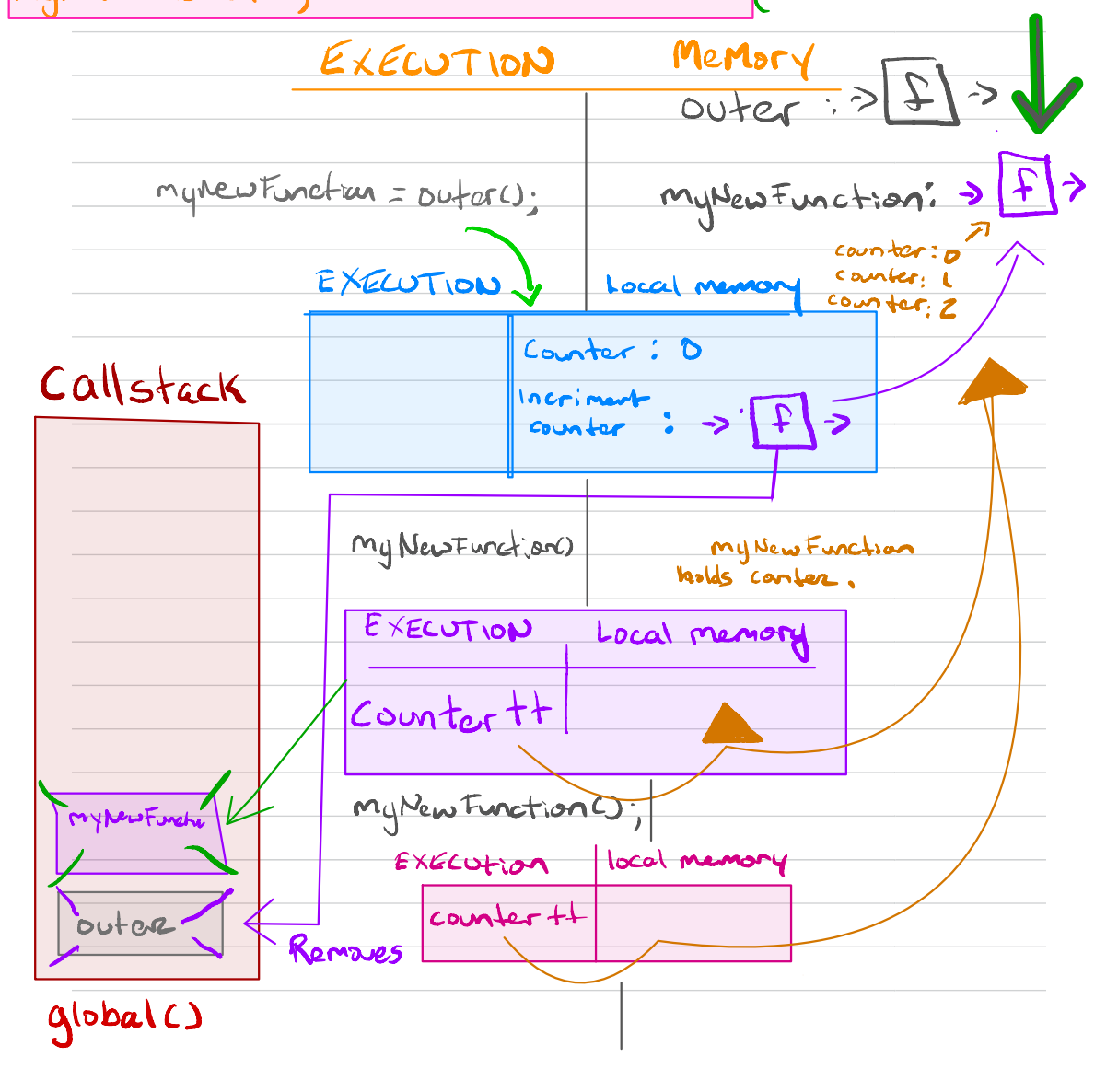
Where you define your functions Determins what data it has access to when you call it;



Retaining Function Memory

```
function outer();
    let counter =
    function incrementCounter()      counter++
    return incrementCounter
const myNewFunction = outer();
myNewFunction();
myNewFunction();
```

When we store increment counter function def, it brings local memory w/ it. (Some caveats)



Continued;

as soon as we store MyNewFunction

[[SCOPE]] ← it gets this Property

The only way we get access is if

we write the code in such a way

it looks for it in local memory.

doesnt find it, than looks to

function definition

Permanent / Private data

We Could do Something Like

```
if counter === 1;
return 'sorry, you cant run me'
```

I.e. A function you can only run once.

Asyn JavaScript

- Promises
- Asyncronicity
- The eventloop
- The microtask queue

In the Above, all the examples are syncronous in execution... what happens if we have to make a call out to some server?

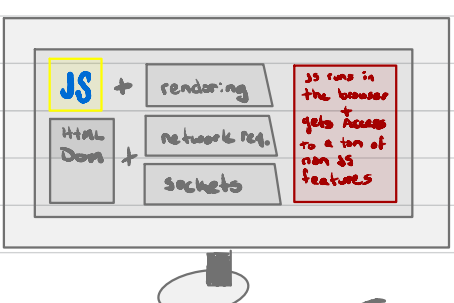
w/ sync code, we can't run any other code... were stuck until the code finishes.

Well JS isn't enough... we need new pieces (some of which are not even JS at all)

Core JS

- Web Browser API's / Node background API's
- Promises
- Event loop, callbacklbacks / TASK queue & the micro task queue.

Additional Components

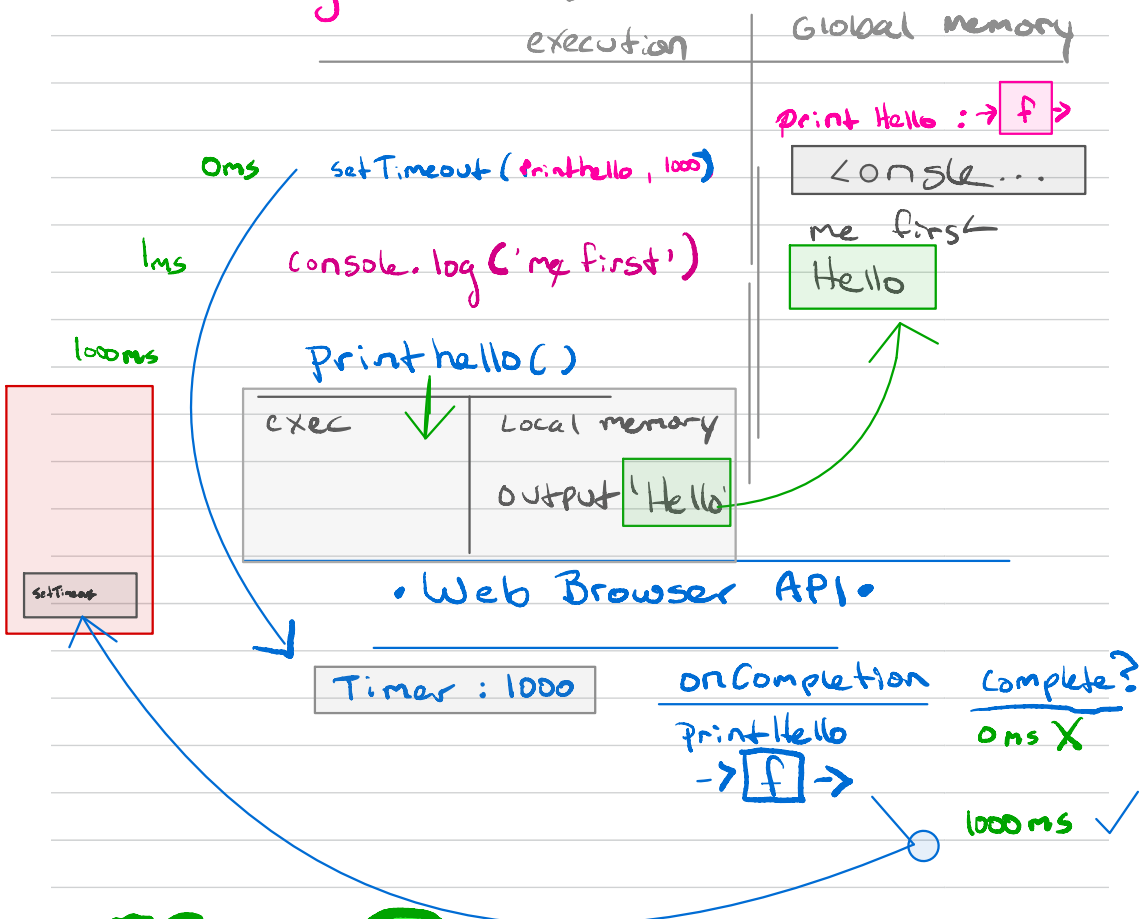


JavaScript, gets access to non JS features in the browser

console.

Xhr... fetch... setTimeout.. document.find...

1. `function printHello() { console.log("hello"); }`
2. `setTimeout(printHello, 1000);`
3. `console.log("me first")`



How SetTimeout Works Browser APIs

The key take away here is
The off loading of tasks to the browser
N the browser adding it back to the call stack

```
function printHello() { console.log("hello"); }
function blockForSec() { //blocking }

setTimeout(printHello, 0);
blockForSec()
console.log("me first")
```

1. define printHello in global memory.
2. define blockForSec in global memory.
3. Trigger web browsers Timmer & pass it printHello & 0ms as Arguments. printHello is a Callback; it's grabbed, thrown out of the global execution context @ 0ms printHello function declaration memory ref will get added to the callback Queue
4. Execution Thread flows onto blockForSec(), adding its local execution context to the call stack
5. @ 1,001 ms wq destroy the local execution context, return to the main thread & execute console.log("me first")
6. @ 1,002 ms, when global() execution finishes, printHello's function declaration memory address gets added to the call stack, opens an execution context & console.logs "Hello"

WHEN GLOBAL EXEC FINISHES CALLBACK QUEUE Gets Added to call STACK

Is the callstack empty? Is there anything in the Queue

After EVERY LINE of code is run, JS checks, is the callstack empty? is there something in the Queue? If the callstack is empty & the callback Queue isn't, Grab the callback Queue & place it on the call stack: EVENT LOOP

```
function printHello() { console.log("hello"); }
function blockForSec() { //blocking }

setTimeout(printHello, 0);
blockForSec()
console.log("me first")
```

Global Execution

Global Memory

printHello: f

blockForSec: f

1. setTimeout(printHello, 0)

→ WebBrowser API

TIMER

printHello 0ms

2. blockForSec()

local exec	local mem
// for loop	

callstack

1000 ms

3. console.log("me first")

1002

4. printHello

local exec. context	local memory
console.log "Hello"	

printHello() : -> f ->

1002

blockForSec

(EVENT LOOP)

After each execution,
1. is the callstack empty
2. is there anything in the callback queue?
If Yes, add callback queue to the callstack & invoke it!

GLOBAL()

printHello: -> f ->

CallBACK Q

console

1000 me first

1002 Hello

Web Browser Features

feature	on completion
Timer 0ms	printHello: -> f ->

ES-5 ↑ Callback Aproach

Web browser Features gets some data

Say w/ XHR or fetch, it's only available in the input & the function added Automatically to the callstack...

The problem, that data is only Available in that functions execution context. ← Callback HELL 

PROBLEMS

ERROR HANDLING

↑ Isn't the Best in ES-5 callbacks

ES-6 Sought to improve upon Asyn JS features, introducing Promises. w/ callbacks, we send Async code to the browser, there is no tracking in JS

PROMISE

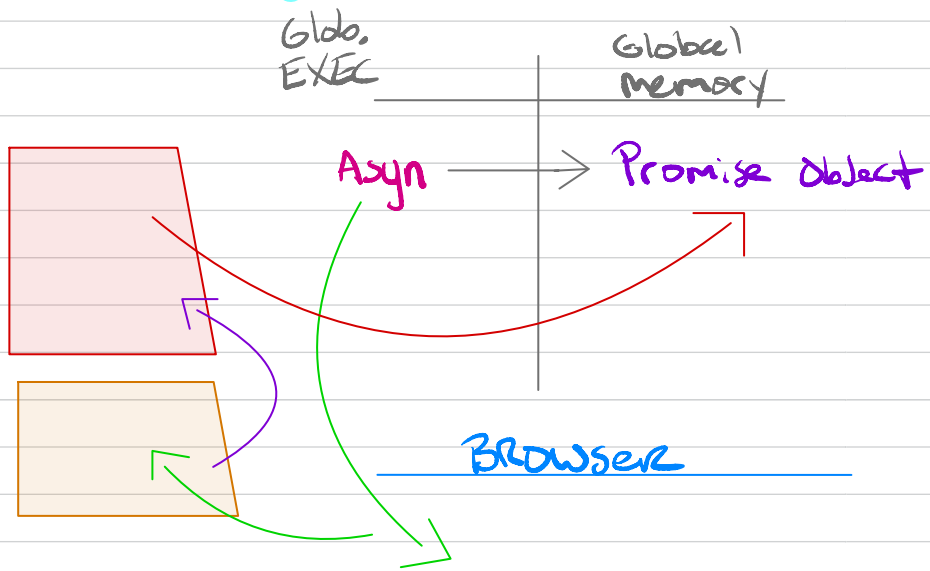
Browser Features

Xhr / fetch -> Network Request

Instead of just off loading to the browser, Asyn code will have an immediate consequence in JS as well,

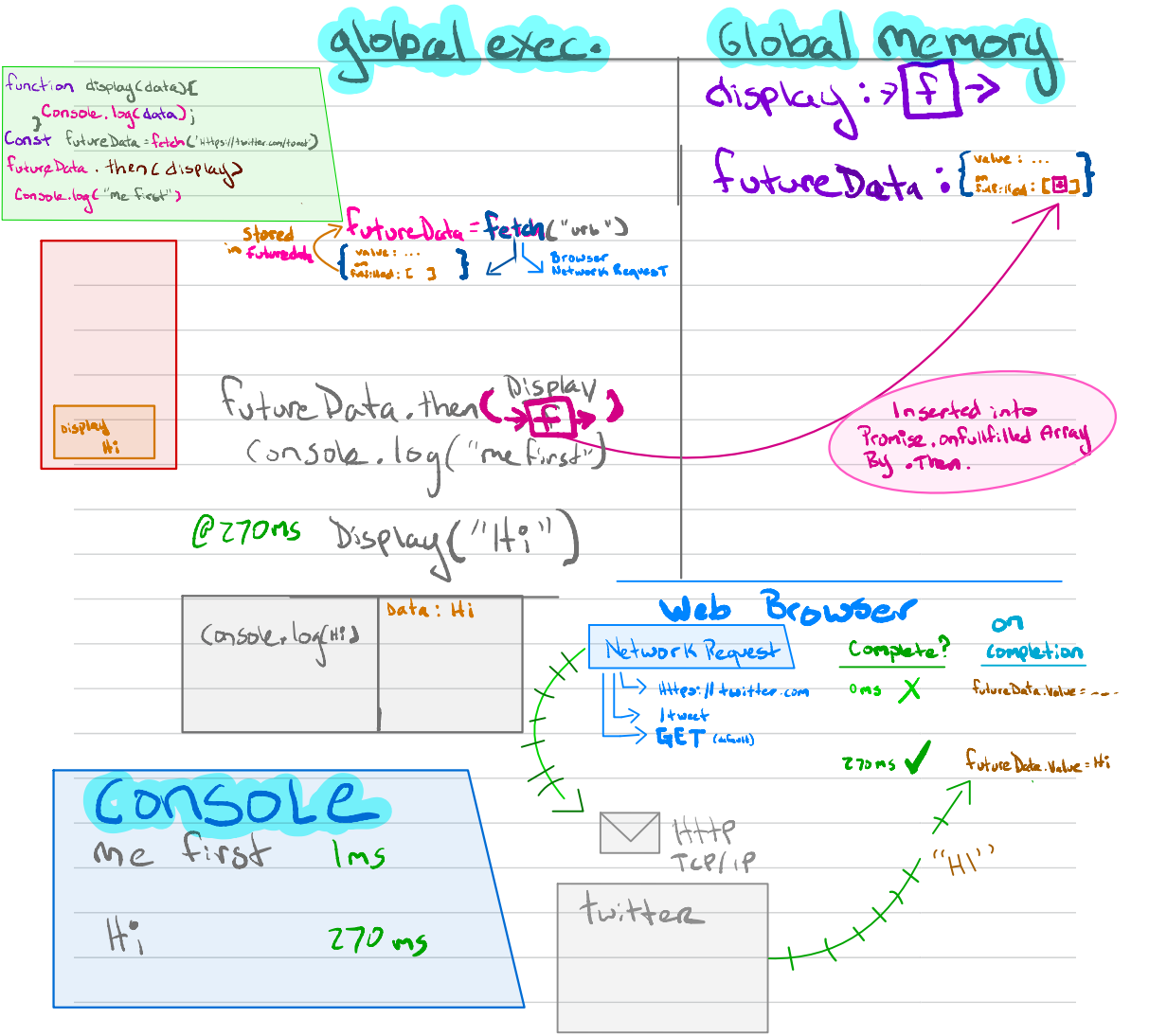
Promises use a 2. pronged Approach,

1. initiate background web browser work
2. Return a placeholder object (Promise) immediately in JS



```
function display(data){
    Console.log(data);
}
Const futureData = fetch('Https://twitter.com/tweet')
futureData . then( display)
 Console.log( "me first")
```

1. Store display's function def. in global memory.
2. Declare const, futureData & it's value is uninitalized.
3. futuredata = fetch(url) is executed. Fetch is a pseudo function, that does 2 things 1. JS create an object w/ 'value': ... & 'onFulfilled': [ ]; We add this object to the value of futureData's label in global memory.
4. The second prong, the url is added to the web browsers Network request feature. When the Http request is fullfilled, we insert it's response data into futureData.value the promise objects value key.
5. futureData.then() gets called w/ Displays function declaration; the "Display" value get inserted into Promise objects onfullfilled array.
6. We Console log "me first"
7. At some point in the future, twitter responds w/ "Hi" adding futureData.value = Hi
8. on fullfilled runs opening an execution context
9. Data is passed in w/ the value "Hi"
10. console.log Hi



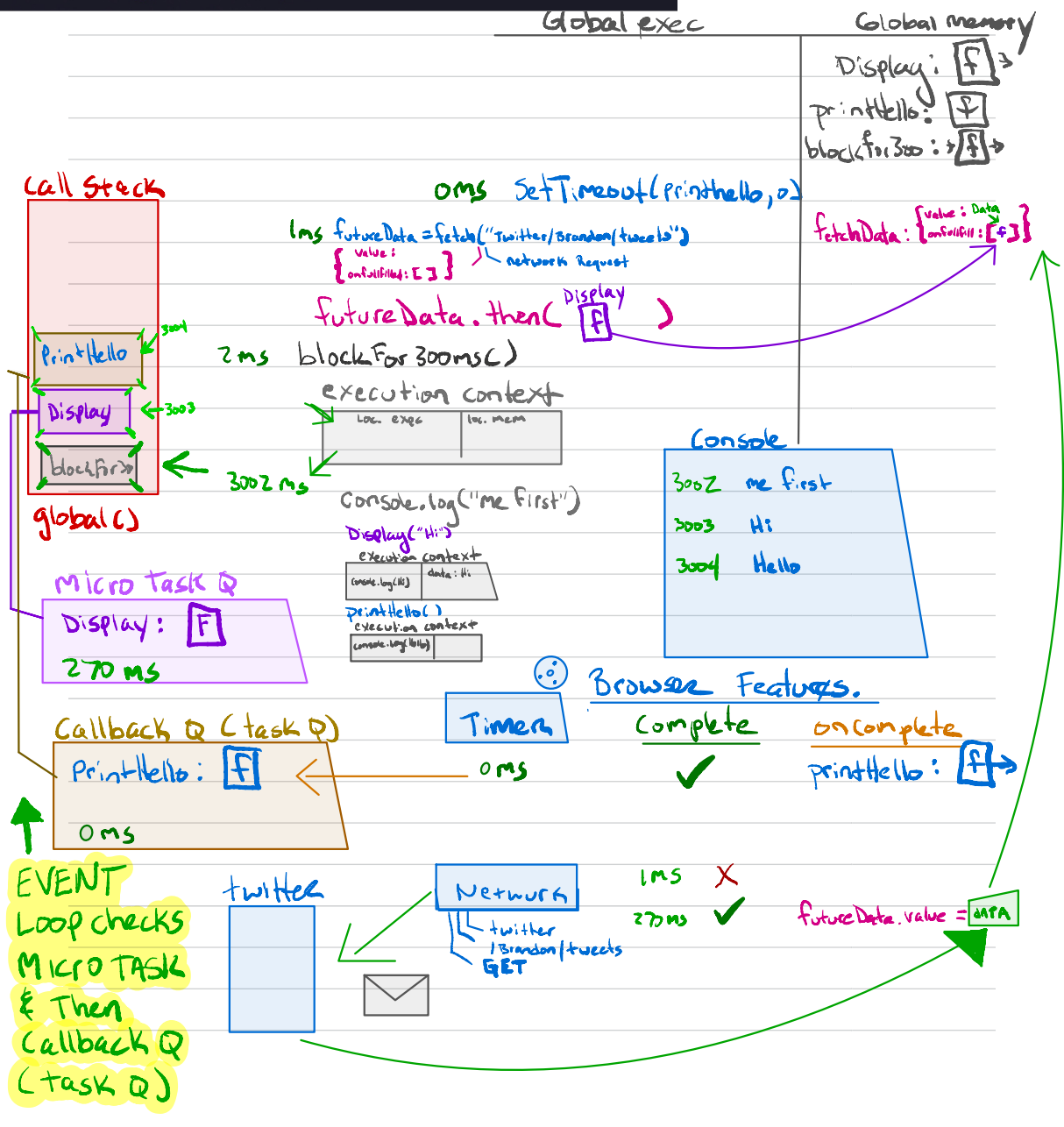
{ value: ___ onFullfilled: [ ] } ES6 Promise object,

Any code we want to run on the returned data must be saved on the promise object

Added using .then() method to the hidden property "onFullfilment"

Promise objects will Auto trigger the func. to run w/ it's input being the returned data.

```
function display(data) {console.log(data)}
function printHello() {console.log('Hello')}
function blockFor300ms() { /* block for 300 ms */ }
setTimeout(printHello, 0) // async code - browser api
const futureData = fetch('twitter.com/brandon/tweets/1')
futureData.then(display)
blockFor300ms()
console.log('Me First')
```



.then()

is a little misleading, it's really storing a func. to run later

OR STORE ME TO RUN BE later when the BACKground task completes & Promise Value property gets updated

The event loop:

1. checks "is the global call stack empty"
2. checks "is the micro-task queue empty"
   "is the callback queue empty"

Any function Attached to a Promise will go to the micro-task Q

Any callback function will go to the task queue (callback queue)

Problems:

99% of devs Don't know How this works under the hood.

which makes Debugging hard

Benefits

We can error handle w/ the promise object, there is Actually 1 more element on the object,

ON Rejection

- then()
- catch() function passed into it, gets added to the on rejection array.

Or, pass the error func. as 2nd arg. to .then()

Hold Promise-Deffered functions in a micro-task queue & callback functions in a task queue (callback Q) when the web Browser Feature (API) finishes

Add the function to the callstack when:

- Call stack is empty & all global code run (as checked by the event loop)

Prioritize functions in the micro-task Queue over the task Queue

this allows single threaded non-blocking code execution

JS

CLASS & OOP

- enormously popular paradigm for structuring our complex code
- Prototype chain - Behind the scenes feature that enables emulation of OOP - compelling tool itself
- __proto__ vs prototype
- new Class keyword as tools to automate our objects & method creation.

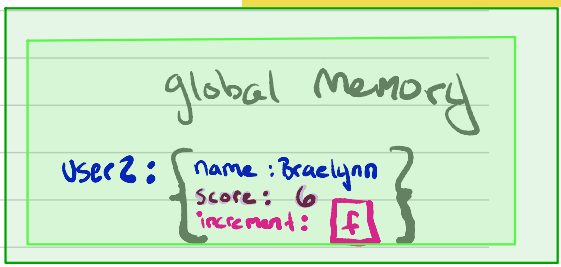
I want my code to be:

1. EASY TO REASON ABOUT
2. EASY TO Add Features to
3. Efficent & Performant

All coding Paradigms aim to check each of these 3 things

```
//*OOP

//? Encapsulation in JS
//? bundle functionality + data into a single package

const user1 = {
    name: 'Brandon',
    score: 3,
    increment: function() {user1.score ++}
}
//? function on the user1 object
user1.increment() //* user1.score 4


//* we can also create an object with dot notation
const user2 = {}
user2.name = 'Braelynn'
user2.score = 2
user2.increment = function() {user2.score ++}
```

JS



Object.create( );
on line 33
will give us more controll later on
returns an empty object

{ }

Will have hidden properties, depending upon what we pass in

```
//*OOP

//? Encapsulation in JS
//? bundle functionality + data into a single package

const user1 = {
    name: 'Brandon',
    score: 3,
    increment: function() {user1.score ++}
}
//? function on the user1 object
user1.increment() //* user1.score 4


//* we can also create an object with dot notation
const user2 = {}
user2.name = 'Braelynn'
user2.score = 2
user2.increment = function() {user2.score ++}


//! Object.create() give us more control later on.....
const user3 = Object.create(null);

user3.name = 'Kalvin'
user3.score = 1
user3.increment = function() {user3.score ++}
```

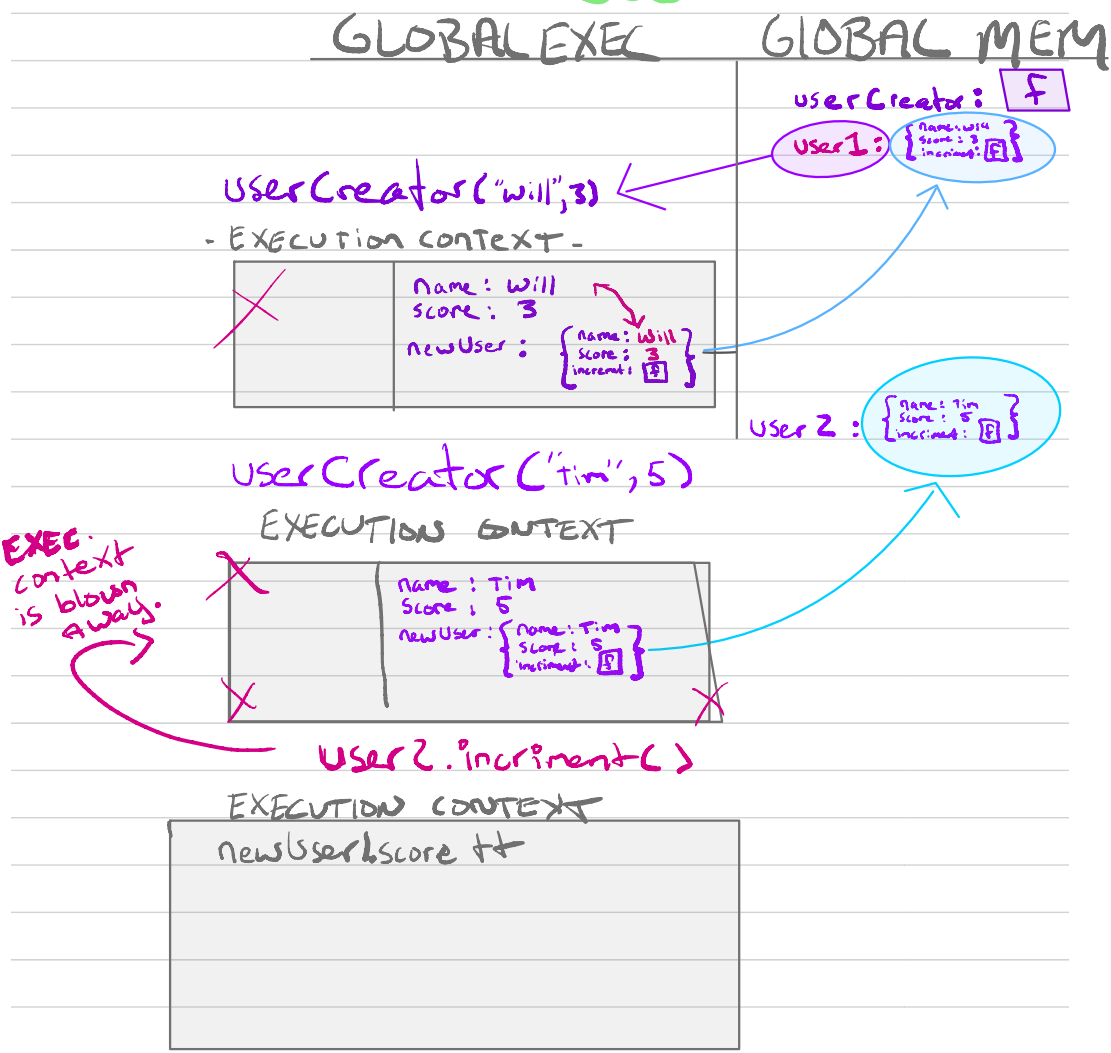
Wrap data & functionality in the same unit

```
function userCreator(name, score) {
  const newUser = {};
  newUser.name = name;
  newUser.score = score;
  newUser.increment = function() {
    newUser.score++;
  };
  return newUser;
};

const user1 = userCreator("Will", 3);
const user2 = userCreator("Tim", 5);
user1.increment()
```

newUser isn't on global memory when run, the execution context gets garbage collected.

so... where does it come from?

The BACKPACK on newUser1 creation.



HASHTAG

The Problem w/ the Above,

1. each object stores the [f] for increment, exact copies.
2. To add on attribute or method, you'd need to update all instances

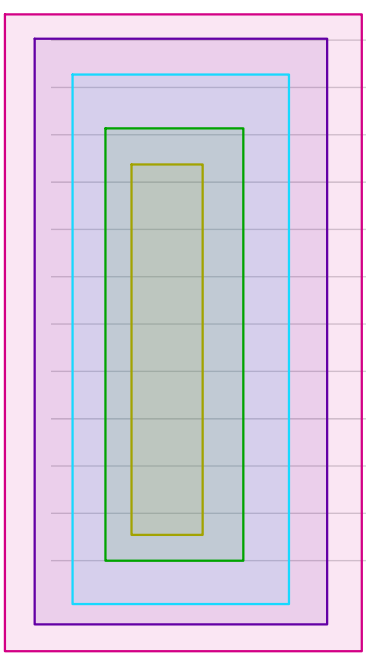
The Prototype

Store the increment function in just 1 object,

Have the interpreter, if it's not found on user1, look up to the object & check if it's there

Link user1 & functionStore so the interpreter, on not finding .increment, makes sure to check up in functionStore where it would find it.

Make the link w/ Object.create() __proto__



Solution 2: Using the prototype chain

```
function userCreator (name, score) {
  const newUser = Object.create(userFunctionStore);
  newUser.name = name;
  newUser.score = score;
  return newUser;
};

const userFunctionStore = {
  increment: function(){this.score++;},
  login: function(){console.log("Logged in");}
};

const user1 = userCreator("Will", 3);
const user2 = userCreator("Tim", 5);
user1.increment();
```

Execution thread

Global Memory

userCreator : F

userFunctionStore : { increment F, login F }

User 1 : { name: Will, score: 3 } __Proto__

userCreator("Will", 3)

Execution context

name : "Will"

score : 3

newUser : { name: Will, score: 3 }

User 2 : { name: Tim, score: 5 }

userCreator("Tim", 5)

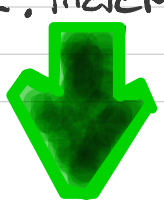
Execution Context

name : "Tim"

score : 5

newUser : { name: "Tim", score: 5 }

user1.increment()



user1.increment(),

it'll look in the global Memory for user1. we find it,

But...
there isn't a .increment() on the object so we go up to the __delable link

When we create an object w/ Object.create() & pass a function into it, we create an __delable link -- PROTO --

How Can user1.increment() know to "look" @ user functionStore? When we create an object, it'll get the name, score we pass in & also the Hidden -- proto --

Dunder Proto Has a link to userfunctionStore

user1.increment()

Execution Context

what if we use a function.

Create and invoke a new function (*add1*) inside *increment*

```
function userCreator(name, score) {
   const newUser = Object.create(userFunctionStore);
   newUser.name = name;
   newUser.score = score;
   return newUser;
};

const userFunctionStore = {
   increment: function() {
       function add1(){ this.score++; }
       add1()
   }
};

const user1 = userCreator("Will", 3);
const user2 = userCreator("Tim", 5);
user1.increment();
```

*What does **this** get auto-assigned to?* 😬

vs an Arrow function

Create and invoke a new function (*add1*) inside *increment*

```
function userCreator(name, score) {
   const newUser = Object.create(userFunctionStore);
   newUser.name = name;
   newUser.score = score;
   return newUser;
};

const userFunctionStore = {
   increment: function() {
       function add1(){ this.score++; }
       add1()
   }
};

const user1 = userCreator("Will", 3);
const user2 = userCreator("Tim", 5);
user1.increment();
```

*What does **this** get auto-assigned to?* 😬

this = undefined

Increment's Property function

global memory

userCreator : → f →

userFunctionStore : { increment : G }

user1 : { name : will, score : 3 }

user1 = userCreator ("will", 3)

execution context

name : will
score : 3
newUser { name : will, score : 3 }

user2 : { name : Tim, score : 5 }

user2 = userCreator ("Tim" : 5)

execution context

name : Tim
score : 5
newUser { name : Tim, score : 5 }

user1.increment()

execution context

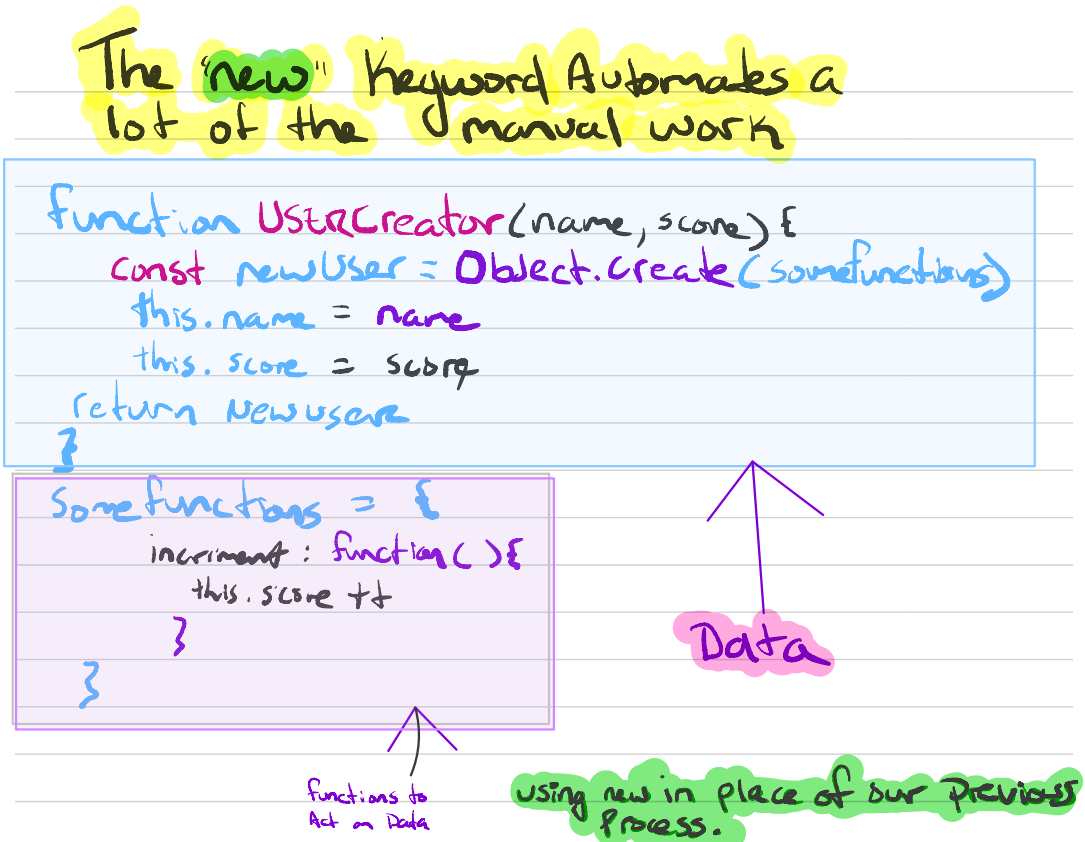
this : user1
add1 : f →

add1()

execution context

this : Global
score ++

NO score is found on this, we'll return undefined



The new keyword automates a lot of our manual work

```
function userCreator(name, score){
  this.name = name;
  this.score = score;
}

userCreator.prototype.increment = function(){ this.score++; };
userCreator.prototype.login = function(){ console.log("login"); };

const user1 = new userCreator(“Eva”, 9)

user1.increment()
```

New

The new keyword automates a lot of our manual work

```
function userCreator(name, score) {
  const newUser = Object.create(functionStore);
  newUser this.name = name;
  newUser this.score = score;
  return newUser;
};

functionStore userCreator.prototype // {};
functionStore userCreator.prototype.increment = function(){
  this.score++;
}                                    Automates the hard work
const user1 = new userCreator("Will", 3);
```

The 'new' keyword removes a lot of the work, 'this' inside of the new keyword is associated w/ the auto-generated obj.

We're dropping the whole newUser inside of Object.create() & using this

What about the link to functionStore?

Functions Are Both Functions & Objects in JS

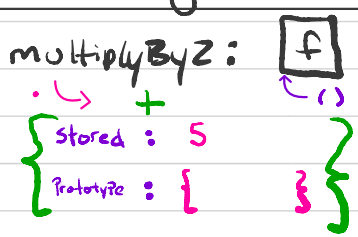
Interlude - functions are both objects and functions 🤔

```
function multiplyBy2(num){
  return num*2
}

multiplyBy2.stored = 5
multiplyBy2(3) // 6

multiplyBy2.stored // 5
multiplyBy2.prototype // {}
```

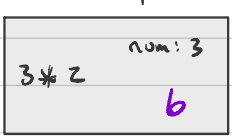
We could use the fact that all functions have a default property `prototype` on their object version, (itself an object) - to replace our `functionStore` object

execution thread

Global memory



multiplyBy2(3)



ALL FUNCTIONS in their object format have prototype properties

So, for multiplyBy2, to Access the function definition, use ()
to Access the object properties use .PropertyName

The new keyword automates a lot of our manual work

```
function userCreator(name, score){
  this.name = name;
  this.score = score;
}

userCreator.prototype.increment = function(){ this.score++; };
userCreator.prototype.login = function(){ console.log("login"); };

const user1 = new userCreator("Eva", 9)

user1.increment()
```

function + object combo, object has a property called prototype which is an empty object.

If we neglect to use new on userCreator() this will refers to global, so convention is to use Capital Letters

Execution thread

Global memory

New Automation w/ new

1. this: { }
2. __proto__ to userCreator.prototype
3. return this

userCreator: f

+

{ prototype: { increment: f, login: f } }

user1 : { name: EVA, score: 9, proto: userCreator.prototype }

user1 = new userCreator('EVA', 9)

•execution context•

	local memory
	name: EVA
	score: 9
	this: { name: 'EVA', score: 9, proto: }

user1.increment()

execution context

	this: user1
score ++	score: 9
	10

user1 in global? ✓

.increment in user1? ✗

.increment in __proto__ ✓

userCreator.prototype

Issues w/ new

1. if you omit new when calling UserCreator, "this" will refer to Global & not the instance of UserCreator

2. methods are seperate from the constructor

This is Where Class Comes In

Syntaxtic Sugar that makes Class creation similar to other languages