JavaScript the Hard Parts

JavaScript principles, Callbacks & Higher Order functions, Closure, Type coercion, Classes/Prototypes & Asynchronicity

Will Sentance

Founder & Teacher: Codesmith

Visiting Research Fellow: Oxford University

Member: South Park Commons SF

Speaker: BBC, NYTimes, MIT, Frontend Masters

Cocreator: Icecomm, Hard Parts

Academic work: Oxford Harvard

What is Codesmith?

3 month full-time (or 9 month part time) software engineering & AI immersive technology program

Our mission is to create the next generation of leaders in technology who care about impact and substance

5000+ **Graduates**

Starting own companies and working at the UN, OpenAI and Google - across US and globe

$110k **Median starting salary**

(Third-party audited for CIRR 2023-24 reporting period)

100,000+ **Github stars**

Projects by Codesmith students have achieved global acclaim

Jenna Davis

Covers the route into Apple as a Senior Software Engineer.

Brandi Richardson

From employee retention to TPM at Google, while driving diversity in tech

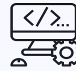
Carlos Botero-Vargas

From Orchestra Conductor to Senior Software Engineer at Capital One

What to focus on in the course

Analytical problem solving

Technical communication



Engineering approach

Non-technical communication

JavaScript and programming experience

Hard Parts of JavaScript

- We'll build the mental models that sit beneath all of JavaScript, Node, React, Next
    - But also many other contemporary languages: Golang, Rust and even Haskell
- We'll start from foundations then dive deep into core 5 pillars of the hard parts of JavaScript
    - Higher order functions, closure, async, oop and type coercion
- We'll introduce recent features from JavaScript that extend each of these pillars including *immutable array methods*, *promise aborts*, *big integers* and *class private & static fields*
- Through all this we'll build a full model of modern JavaScript to let us:
    - Write clean, readable and resilient code for us and our teams
    - Autonomously tackle any new feature, tool or codebase

Contents

1. **Principles of JavaScript**
2. Callbacks & Higher order functions
3. Closure (scope and execution context)
4. Type Coercion & Metaprogramming
5. Asynchronous JavaScript & the event loop
6. Classes & Prototypes (OOP)

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


const output = multiplyBy2(num);
const newOutput = multiplyBy2(10);
```

JavaScript principles

When JavaScript code runs, it:

↘ Goes through the code line-by-line and runs/ 'executes' each line - known as the **thread of execution**

Saves 'data' like strings and arrays so we can use that data later - in its **memory**

We can even save code ('functions')

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

const output = multiplyBy2(num);
const newOutput = multiplyBy2(10);
```

Functions

Code we save (‘define’) functions & can use (call/invoke/execute/run) later with the function’s name & ( )

Execution Context

Created to run the code of a function - has 2 parts (we’ve already seen them!)

- Thread of execution
- Memory

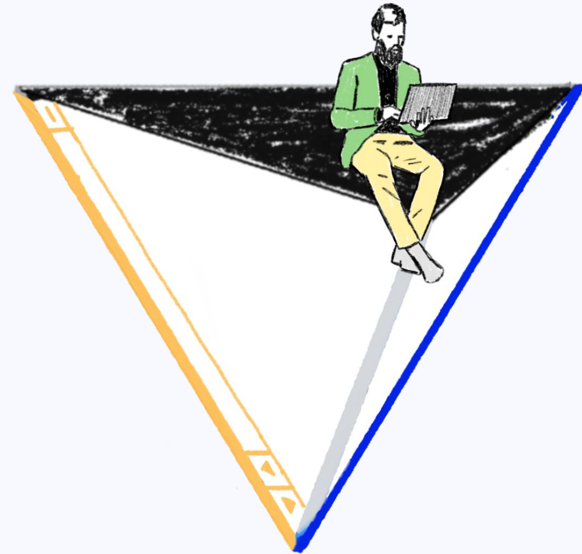
(We’ll see another way of defining functions later)

Call stack

- JavaScript keeps track of what function is currently running (where’s the thread of execution)
- Run a function - add to call stack
- Finish running the function - JS removes it from call stack
- Whatever is top of the call stack - that’s the function we’re currently running

Contents

1. Principles of JavaScript
2. **Callbacks & Higher order functions**
3. Closure (scope and execution context)
4. Type Coercion & Metaprogramming
5. Asynchronous JavaScript & the event loop
6. Classes & Prototypes (OOP)



Callbacks & Higher Order Functions

- One of the most misunderstood concepts in JavaScript
- Enables powerful pro-level functions like map, filter, reduce (a core aspect of functional programming)
- Makes our code more declarative and readable

Why do we even have functions?

Create a function 10 squared

- Takes no input
- Returns 10*10

What is the syntax (the exact code we type)?

tenSquared

```
function tenSquared() {
    return 10*10
}
tenSquared() // 100
```

What about a 9 squared function?

nineSquared

```
function nineSquared() {
    return 9*9
}
nineSquared() // 81 🤨
```

And an a 8 squared function? 125 squared?

What principle are we breaking?

eightSquared

```
function eightSquared() {
    return 8*8
}
eightSquared() // 64
```

😐

And an a 8 squared function? 125 squared?

What principle are we breaking? ***DRY (Don't Repeat Yourself)***

We can generalize the function to make it reusable

```
function squareNum(num){
    return num*num
}
squareNum(10) // 100
squareNum(9) // 81
squareNum(8) // 64
```

Generalizing functions

‘Parameters’ (placeholders) mean we don’t need to decide what **data** to run our functionality on until we run the function

- Then provide an actual value (‘argument’) when we run the function

Higher order functions follow this same principle.

- We may not want to decide exactly what some of our **functionality** is until we run our function

```javascript
function copyArrayAndMultiplyBy2(array) {
   const output = []
   for (let i = 0; i < array.length; i++) {
     output.push(array[i] * 2)
   }
   return output
 }

const myArray = [1,2,3]
const result = copyArrayAndMultiplyBy2(myArray)
```

```
function copyArrayAndDivideBy2(array) {
   const output = []
   for (let i = 0; i < array.length; i++) {
     output.push(array[i] / 2)
   }
   return output
 }

const myArray = [1,2,3]
const result = copyArrayAndDivideBy2(myArray)
```

```javascript
function copyArrayAndAdd3(array) {
  const output = []
  for (let i = 0; i < array.length; i++) {
    output.push(array[i] + 3)
  }
  return output
}

const myArray = [1,2,3]
const result = copyArrayAndAdd3(myArray)
```

What principle are we breaking?

Or add 3?

```
function copyArrayAndAdd3(array) {
   const output = []
   for (let i = 0; i < array.length; i++) {
     output.push(array[i] + 3)
   }
   return output
 }

const myArray = [1,2,3]
const result = copyArrayAndAdd3(myArray)
```

What principle are we breaking?

DRY - Don’t Repeat Yourself

```
function copyArrayAndManipulate(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 = copyArrayAndManipulate([1, 2, 3], multiplyBy2)
```

How was this possible?

Functions in javascript = first class objects

They can co-exist with and can be treated like any other javascript object

1. Assigned to variables and properties of other objects
2. Passed as arguments into functions
3. Returned as values from functions

```
function copyArrayAndManipulate(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 = copyArrayAndManipulate([1, 2, 3], multiplyBy2)
```

```
function copyArrayAndManipulate(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 = copyArrayAndManipulate([1, 2, 3], multiplyBy2)
```

Which is our Higher Order Function?

The outer function that *takes in* a function is our higher-order function

Which is our Callback Function

The function we insert is our callback function

Higher-order functions

Takes in a function or passes out a function

Just a term to describe these functions - any function that does it we call that - but there's nothing different about them inherently

Callbacks and Higher Order Functions simplify our code and keep it DRY

Declarative readable code: Map, filter, reduce - the most readable way to write code to work with data

Codesmith & pro interview prep: One of the most popular topics to test in interview both for Codesmith and mid/senior level job interviews

Asynchronous JavaScript: Callbacks are a core aspect of async JavaScript, and are under-the-hood of promises, async/await

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

```
const multiplyBy2 = (input) => { return input * 2 }
```

```
const multiplyBy2 = (input) => input * 2
```

```
const multiplyBy2 = input => input * 2

const output = multiplyBy2(3) //6
```

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

const multiplyBy2 = input => input*2
const result = copyArrayAndManipulate([1, 2, 3], multiplyBy2)
```

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

const multiplyBy2 = input => input*2
const result = copyArrayAndManipulate([1, 2, 3], input => input*2)
```

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

const multiplyBy2 = input => input*2
const result = map([1, 2, 3], input => input*2) // [2,4,6]
const sameResult = [1, 2, 3].map(input => input*2) // [2,4,6]
```

Anonymous, arrow functions & built-in 'array methods'

- Improve immediate legibility of the code
- Here arrow functions are ‘syntactic sugar’ - we’ll see their full effects later
- Built-in array methods are available after a '.' on arrays - see later where they come from
- Understanding how they’re working under-the-hood is vital to avoid confusion

Some of the *classic* built-in methods ‘mutated’ arrays!

- Our copyArrayAndManipulate created a new array with the changes made
- The built-in `map` returns a brand new array like ours does - no mutation!
  - But some don't 😂 `.reverse()` reverses ‘in place’ - a crime
  - In fact multiple historically changed the array directly including:
    - `.sort()`
    - `.splice()`

```
const arr1 = [1, 2, 3];  const arr2 = [1, 2, 3]

arr1.reverse()   // [3, 2, 1]
arr1.splice(1, 1, 6) // replace 1 element at index 1 with 6 - [3, 6, 1]
arr1.sort() // [1, 3, 6]

const reversed = arr2.toReversed()          ES2023
const spliced = arr2.toSpliced(1,1,6)       ES2023
const sorted = arr2.toSorted()              ES2023

console.log(arr1)
// [1, 3, 6]


console.log(arr2, reversed, spliced, sorted)
// [1, 2, 3] [3, 2, 1] [1, 6, 3] [1, 2, 3]
```

And JavaScript keeps adding more useful array methods

```
const deepArr = [1, 2, [1, 2], 2]   ES2019 (ES10)
const flattened = deepArr.flat() // [1, 2, 1, 2, 2]
// Can set depth (even 'Infinity'), defaults to 1

const last2 = flattened.findLastIndex(x => x === 2)
// last 2 is at idx 4                ES2023 (ES14)

function oddOrEven(num) {
  return num % 2 === 0 ? "even" : "odd"
}
const grouped = Object.groupBy(flattened, oddOrEven)
// → { even: [2, 2, 2], odd: [1, 1] }   ES2024 (ES15)
// Popular SQL statement
```

Pair programming

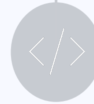
The most effective way to grow as a software engineer

Researcher

Avoids blocks by reading everything they can find on their block/bug

Vibe coder

Uses code snippets to fix bug without knowing how they work



Pair programming

- Tackle blocks with a partner
- Stay focused on the problem
- Refine technical communication
- Collaborate to solve problem

Pairing up

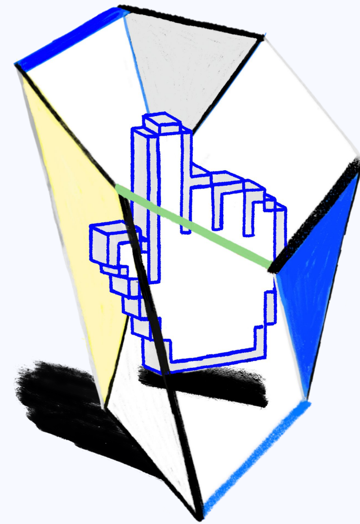
csbin.io/callbacks

- I know what a string is
- I've created a function before
- I have added an image to a webpage
- I understand what an arrow function is
- I can add a method to an object's prototype
- I understand the event loop in JavaScript
- I can implement reduce from scratch
- I can implement filter
- I can handle collisions in a hash table

Give yourself a point for each topic you know for a total out of 9

Contents

1. Principles of JavaScript
2. Callbacks & Higher order functions
3. **Closure (scope and execution context)**
4. Type Coercion & Metaprogramming
5. Asynchronous JavaScript & the event loop
6. Classes & Prototypes (OOP)



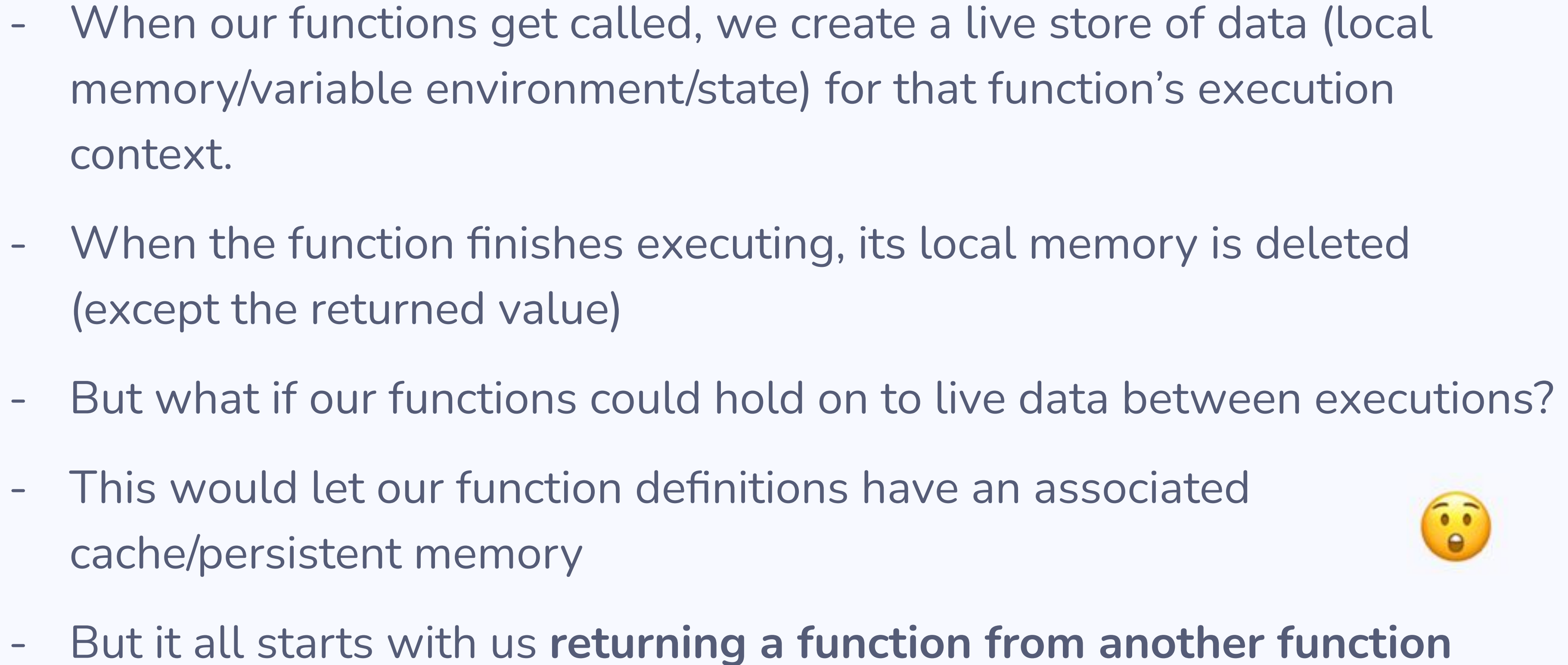
Closure

- Closure is the most esoteric of JavaScript concepts
- Enables powerful pro-level functions like ‘once’ and ‘memoize’
- Many JavaScript design patterns including the module pattern use closure
- Build iterators, handle partial application and maintain state in an asynchronous world

```
function multiplyBy2 (inputNumber){
  const result = inputNumber*2
  return result
}

const output = multiplyBy2(7)
const newOutput = multiplyBy2(10)
```

Functions with memories



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

const generatedFunc = createFunction()
const result = generatedFunc(3) // 6
```

Pair programming challenges

csbin.io/closures

Calling a function in the same function call as it was defined

```
function outer (){
    let counter = 0
    function add1 (){
        counter ++
    }
    add1()
}
outer()
```

Where you *define your functions* determines what data it has access to when you call it

Calling a function outside of the function call in which it was defined

```
function outer (){
    let counter = 0
    function add1 (){ counter ++ }
    return add1
}

const newFunc = outer()
newFunc()
newFunc()
```

The bond

When a function is defined, it gets a bond to the surrounding Local Memory (“Variable Environment”) in which it has been defined

The ‘backpack’

- We return *add1* code (function definition) out of *outer* into global and give it a new name - *newFunc*
- We **maintain the bond to outer’s live local memory** - it gets ‘returned out’ attached on the back of *add1's* function definition.
- So *outer*’s local memory is now stored attached to *newFunc* - even though *outer*’s execution context is long gone
- When we run *newFunc* in global, it will first look in its own local memory first (as we’d expect), but then in *newFunc*’s ‘backpack’

What can we call this ‘backpack’?

- Closed over ‘Variable Environment’ (C.O.V.E.)
- Persistent Lexical Scope Referenced Data (P.L.S.R.D.)
- ‘Backpack’
- ‘Closure’

The ‘backpack’ (or ‘closure’) of live data is attached add1 (then to newFunc) through a hidden property known as [[scope]] which persists when the inner function is returned out

Let’s run outer again

```
function outer (){
   let counter = 0
   function add1 (){
       counter ++
   }
   return add1
}
```

```
const newFunc = outer()
newFunc()
newFunc()

const anotherFunction = outer()
anotherFunction()
anotherFunction()
```

Individual backpacks

If we run '**outer**' again and store the returned '**add1**' function definition in '**anotherFunction**', this new add1 function was created in a new execution context and therefore has a brand new independent backpack

Closure gives our functions persistent memories and entirely new toolkit for writing professional code

Helper functions: Everyday professional helper functions like ‘once’ and ‘memoize’

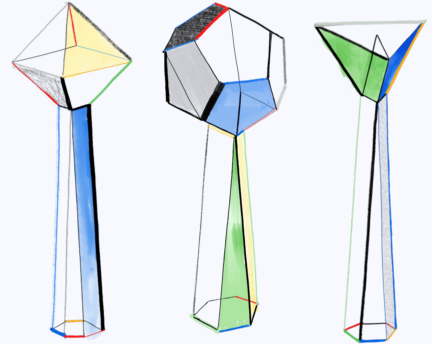
Iterators and generators: Which use lexical scoping and closure to achieve the most contemporary patterns for handling data in JavaScript

Module pattern: Preserve state for the life of an application without polluting the global namespace

Asynchronous JavaScript: Callbacks and Promises rely on closure to persist state in an asynchronous environment

Contents

1. Principles of JavaScript
2. Callbacks & Higher order functions
3. Closure (scope and execution context)
4. **Type Coercion & Metaprogramming**
5. Asynchronous JavaScript & the event loop
6. Classes & Prototypes (OOP)



Type Coercion, Operators & metaprogramming

- Hidden and infamously powerful system built to be flexible to browser I/O
- Depends on a deep understanding of primitives, operators and memory under the hood
- Introduces Symbols to give us manual control over coercion & even metaprogramming
- If understood, we can write code that’s more bug-resistant and answer interviewer's favorite JS quirk questions from first principles!

```
const price = 7
let quantity = 3
let total = price * quantity
            // 7 * 3
            // = 21
```

```
const price = 7
let quantity // document.getElementById("q").value
let total // initially undefined

function onSubmit(){
  total = price * quantity // 7 * "3" 🥴
}

onSubmit() // runs when user clicks submit
```

Nope! Introducing Type coercion

As soon as JavaScript sees the math operator * our "3" gets automatically turned into 3

```
total = 7 * "3"  // 7 * →3 = 21
```

A ‘ToNumber’ **type coercion** is kicked off

- Then just a regular * operation

What about other math-y operators? Do they 'coerce' ToNumber?

Nope! Introducing Type coercion

As soon as JavaScript sees the math operator * our "3" gets automatically turned into 3

```
total = 7 * "3" // 7 * →3 = 21
```

A ‘ToNumber’ **type coercion** is kicked off

- Then just a regular * operation

What about other math-y operators? Do they 'coerce' ToNumber?

```
const price = 7
let quantity // DOM gives us "3"
let total // initially undefined

function onSubmit(){
  total = price * quantity // 7 * →3 = 21
}

onSubmit() // runs when user clicks submit
```

```
const price = 7
let quantity // DOM gives us "3"
const max = 10
let total

function onSubmit(){
  total = price * quantity
  if(quantity < max) // →3 < 10 (true)
    {console.log("All good!")}
}
onSubmit()
```

```
const price = 7
let quantity // DOM gives us "3"
let donation // DOM gives us "10"
let total

function onSubmit(){
  total = price * quantity + donation
          // 7 * →3 + "10" // 21 + "10"
          // →"21" + "10" // "2110"
}
onSubmit()
```

Add (ie '+') it to our total

- But if either side of '+' is a string JS automatically kicks off a different coercion: **'ToString'**
- Giving us some v odd results

Most math operators (-,*,/,%,**) always → ToNumber, '+' only does legit math if both sides are numbers

What can we do at this DOM boundary to be predictable?

We need to take manual control of type coercion

If we want to guarantee we're working w numbers we use **'unary operator' +/- - or Number()** to manually kick off ToNumber coercion

If we want to guarantee strings we can use **String() & `${}`** to kick off 'ToString' coercion

There's probably no other coercions right?

```
const price = 7
let quantity // DOM gives us "3"
let donation // DOM gives us "10"
let total

function onSubmit(){
  total = +price * +quantity + +donation
          // →7 * →3 + →10 = 31
}
onSubmit()
```

```
const price = 7
let quantity // DOM gives us " " or 0
let donation // DOM gives us "10"
let total

function onSubmit(){
  if (quantity){ // 0 → false and " " → false
    total = +price * +quantity + +donation
  }
  else { console.log("Add items!") }
}
onSubmit()
```

We could explicitly check donation is 0 with equality (==)

```
const price = 7
let quantity // DOM gives us "3"
let donation // DOM gives us " " or 0
let total


function onSubmit(){
  if (donation == 0){ // 0 → 0 and " " → 0
    console.log("0 donation, no problem")
  }
  else {console.log("Want to donate?")}
}
```

“threeequals” (===) or “strict equality”

```
const price = 7
let quantity // DOM gives us "3"
let donation // DOM gives us " " or 0
let total


function onSubmit(){
  if (donation === 0){ // no coercion
    console.log("0 donation, no problem")
  }
  else {console.log("Want to donate?")}
}
```

So our 3x coercions are:

ToNumber

- "3" → 3, "-3" → -3
- true → 1, false → 0, null → 0
- undefined → NaN (itself a number 🫠)

ToString

- 5 → "5", -5 → "-5"
- true/false, null → "true"/"false", "null"
- undefined, NaN → "undefined", "NaN"

ToBoolean

- 0, "", null, undefined, NaN → false
- Everything else → true

And each coercion is kicked off by operators or actions

- **Math** (*,-,/,%,**) → ToNumber
- **unary** (+,- right in front) → ToNumber
- **+** → ToString (unless both are numbers)
- **Relational** (<,>,<=,>=) → first try ToNumber then ToString (alphabetical comparison)
- **Loose equality** (==) → all over the place (avoid)
- **Conditionals** (if, ||, &&, !) → ToBoolean
- `${}` and Browser APIs → ToString

And each coercion is kicked off by operators or actions

- **Math** (*,-,/,%,**) → ToNumber
- **unary** (+,- right in front) → ToNumber
- **+** → ToString (unless both are numbers)
- **Relational** (<,>,<=,>=) → first try ToNumber then ToString
- **Loose equality** (==) → all over the place (avoid)
- **Conditionals** (if, ||, &&, !) → ToBoolean
- `${}` and Browser APIs → ToString

```
const userStored = {name: "Will", id: 105}
const userSubmitted = {name: null, id: null}
    // user enters {name : "Will", id : 105}


function onSubmit(){
  // Compare submitted user with our valid user
  if(userSubmitted == userStored){
    // Submit
  }
}
onSubmit()
const anotherLink = userStored
```

Coercions done? Let's just check our user is valid first

We can't have any person buying!

Our user submits their name + id and we compare to an object with our (only!) valid user (in practice this is done securely)

When user enters 'Will' and 105 userSubmitted object is updated we compare `userSubmitted == userStored`

But it's false - Maybe we're doing unintentional coercion - try `===` Still false!

```
const userStored = {name: "Will", id: 105}
const userSubmitted = {name: null, id: null}
      // user enters {name : "Will", id : 105}

function onSubmit(){
  if(userSubmitted == userStored){
    // Submit
  }
}
onSubmit()
const anotherLink = userStored
```

The stack vs the heap

All our values we've seen so far **7, "3", and true** have been '**primitives**' stored directly where we save them - the '**stack**' (as they're a predictable size & single)

Our coercion rules so far applied to primitives

The stack vs the heap

But objects combine primitives & can be big as we want

- Instead of storing directly they're stored in a flexible store - the '**heap**'. All that's saved to the stack is a link or '**reference**' - *anotherLink* just has a copy of *userStored*'s link to the heap!
- To compare objects' contents we have to manually '**traverse**' them 🫠
- Shame as there's 2 objects we could compare to seriously improve our UX

'Time objects' could prevent accidental submissions

Built-in objects from `Date()` that gives exact time (to improve UX)

Stored on hidden property *[[DateValue]]*

- The time is stored as 'total ms starting at 0' since `Jan 1 1970 00:00:00 (midnight)` 🫠

Suppose we're running `Date()` at:

- `Jan 15 2027 00:00:00 (midnight)`
- How many ms since 0 is that?

```
const time1 = new Date() // Jan 15 2027 00:00:00

// 3 seconds pass
const time2 = new Date() // Jan 15 2027 00:00:03
```

```
const time1 = new Date() // Jan 15 2027 00:00:00

// 3 seconds pass

const time2 = new Date() // Jan 15 2027 00:00:03


const month = "jan"
time1[month] = true
time2[month] = true
```

Useful numbers but we can't get access to them!

We can't access the hidden *[[DateValue]]*

Even though we can add properties to them like any other

- Here we're referencing month ("jan") and setting it as a key (regular property names are just strings!)

So how can we possibly compare the times of `time1` & `time2`?

- And fix our UX?

```
const time1 = new Date() // Jan 15 2027 00:00:00

// 3 seconds pass

const time2 = new Date() // Jan 15 2027 00:00:03


const month = "jan"
time1[month] = true
time2[month] = true
```

Surely there's no way to compare the objects directly?

Could we subtract the objects from each other to see if difference is <2000ms?

- We'd just be comparing links no?

And yet, we can ...

- Introducing **ToPrimitive coercion**

Automatically coerces both objects to a primitive - here, to a number *[[DataValue]]* (the ms since jan 1 1970)

- But how?!

```
const time1 = new Date() // Jan 15 2027 00:00:00

// ⏰ Only 1 sec passes

const time2 = new Date() // Jan 15 2027 00:00:01


const month = "jan"
time1[month] = true
time2[month] = true


if (time2 - time1 < 2000) {
  console.log("Accident?") // Too soon
}
```

Via the hidden *@@toPrimitive* property

The math (`time2 - time1`) kicks off object → primitive coercion

JS automatically looks for hidden property *@@toPrimitive* on the objects (we can't refer to it directly)

JS has stored instructions there to coerce both `time1` and `time2` to their (also hidden) [[DateValue]] numbers!

1,800,000,001,000ms - 1,800,000,000,000ms <2000ms

= 1,000 ms < 2000 ms (true - 'Accident?')

Via the hidden *@@toPrimitive* property

The math (`time2 - time1`) kicks off object → primitive coercion

JS automatically looks for hidden property *@@toPrimitive* on the objects (we can't refer to it directly)

JS has stored instructions there to coerce both `time1` and `time2` to their (also hidden) [[DateValue]] numbers!

1,800,000,001,000ms - 1,800,000,000,000ms <2000ms

= 1,000 ms < 2000 ms (true - 'Accident?')

```
const time1 = new Date() // Jan 15 2027 00:00:00

// ⏰ 1 sec passes

const time2 = new Date() // Jan 15 2027 00:00:01


const month = "jan"
time1[month] = true
time2[month] = true


if (time2 - time1 < 2000) {
  console.log("Accident?") // Too soon
}
```

```
const userStored = {name: "Will", id: 105}
const userSubmitted = {name: "Will", id: 105}

function onSubmit(){
  if(+userStored === +userSubmitted){
    // NaN === NaN ← false
  }
}



onSubmit()
```

```
const userStored = {name: "Will", id: 105}
const userSubmitted = {name: "Will", id: 105}

function onSubmit(){
  if(+userStored === +userSubmitted){
    // NaN === NaN ← false
  }
}

function coerce (){ return 105 }


onSubmit()


// userStored.@@toPrimitive - Nope!!
```

Let's create a function that just returns 105 to attach

That's what we want our objects to 'coerce' to

- But how to add it to `userStored` & `userSubmitted`?

We can't add a *@@toPrimitive* property directly - it's hidden!

So how can we ever store a function (method) on it? We're stuck!

```
const userStored = {name: "Will", id: 105}
const userSubmitted = {name: "Will", id: 105}

function onSubmit(){
  if(+userStored === +userSubmitted){
    // NaN === NaN ← false
  }
}
function coerce (){ return 105 }


onSubmit()


// userStored.@@toPrimitive - Nope!!
```

JS lets us *use* (but not write) a *@@toprimitive* label

Stored on a built-in object called '**Symbol**'

While we can't 'write'/type these sorts of hidden **@@** labels into our code directly

- We can refer to them (using square bracket notation) and JS will evaluate to the actual hidden label ***@@toPrimitive***
- Then we attach our coerce function!

When JS tries to coerce our '**objects** → **primitive**' the coerce functions will run & each will return **105**!

```
// Symbol: { toPrimitive: @@toprimitive, + more}
const userStored = {name: "Will", id: 105}
const userSubmitted = {name: "Will", id: 105}

function onSubmit(){
  if(+userStored === +userSubmitted){
    // 105 === 105 ← true!
  }
}

function coerce (){ return 105 }


userStored[Symbol.toPrimitive] = coerce
userSubmitted[Symbol.toPrimitive] = coerce

onSubmit()
```

```
// Symbol: { toPrimitive: @@toprimitive, + more}
const userStored = {name: "Will", id: 105}
const userSubmitted = {name: "Will", id: 105}

function onSubmit(){
  if(+userStored === +userSubmitted){
    // 105 === 105 ← true!
  }
}

function coerce (hint){
  if (hint === "string"){return "user"}
  if (hint === "number"){return 105}
}

userStored[Symbol.toPrimitive] = coerce
userSubmitted[Symbol.toPrimitive] = coerce

onSubmit()
```

We now have control of our object → primitive coercion!

We can even add different rules depending on whether we're corecing the object to a number or a string!

When JS runs the function stored on hidden property *@@toPrimitive* it auto inserts:

- "number" if ToNumber: e.g **+userStored or Number()**
- "string" if ToString: eg: **`$(userStored)`**

We just have to write the conditionals!

What are these 'hidden properties' with so much power? Even logging `Symbol.toPrimitive` won't show *@@toPrimitive*

Introducing Symbols - ES6 feature for adding semi-hidden properties to objects

Labels (unique 'identifiers') that cannot be written out directly and so won't override developer's existing code

- *@@toPrimitive* is not a string & won't override developers' existing 'toPrimitive' property!
- Looping through an object's properties won't find *@@toPrimitive*

Introducing Symbols - ES6 feature for adding semi-hidden properties to objects

But JavaScript let's us use them to give us access to under-the-hood features of the language

- E.g. the ability to manually control ToPrimitive coercion flow (both ToNumber and ToString)
- These built-in symbols that JS recognizes are known as 'well-known symbols' 😐

And symbols open up 'metaprogramming'

Beyond coercion - we can access many behaviors

- From iterators, to async features to the behavior of classes

By ensuring backwards-compatibility and semi-hidden status, JavaScript can safely let us control and override default language rules and make explicit implicit behaviors

And symbols open up 'metaprogramming'

We know we can manually control our ToNumber, ToString and ToBoolean coercion steps.

With symbols we can now fully control our coercion pipeline - taking full control of our object ToPrimitive coercion

Making our type coercion explicit gives us both flexibility and predictability

Coercion everywhere: Every operator or API call may trigger hidden coercions - but with a clear map you can predict these instead of being surprised

Explicit control over flexibility: JS has type flexibility to create readable code at the browser edge but it's mostly too unpredictable & requires explicit handling

Making our type coercion explicit gives us both flexibility and predictability

Symbols & metaprogramming: Our 'well-known symbols' give fine-grained backwards-compatible control over type coercion and other built-in JS features

TypeScript: Automates type control but understanding type coercion under-the-hood remains crucial to wielding JS effectively (& passing interviews 🥲)

Weakref/deref

```
let obj = { name: "Will" }
linkToObj = new WeakRef(obj)
console.log(linkToObj.deref()) // { name: 'Will' }

obj = null // drop strong reference

// At some later point when unattached objects after garbage collected
console.log(linkToObj.deref()) // undefined
```

ES2021

ES2021

ES2021

deref just returns and displays the weak referenced object

We can also trigger a function to run on garbage collection with `FinalizationRegistry()`

Extension: Latest JavaScript features

BigInt - handles big numbers that regular numbers can't handle

```
let oneEth = 1_000_000_000_000_000_000 // 1 quintillion wei (like cents in a dollar)

oneEth = oneEth + 10 // Add 1 wei
// Regular 'numbers' this big (in most languages) can't handle small changes
      // They're designed to be efficiently stored
// Limit for integers is: 2^53 - 1 = 9,007,199,254,740,991

console.log(oneEthTenWei === oneEth) // true 🥴

// BigInt let's any size number in! only limit is memory & processing power - Just add 'n'
oneBigIntEth = 1_000_000_000_000_000_000n
oneBigIntEthOneWei = oneBigIntEth + 1n // e tiny addition

console.log(oneBigIntEthOneWei === oneBigIntEth) // false
```

ES2020

ES2020

Extension: Latest JavaScript features

Temporal object (Coming soon)

ECMA Stage 3 (stable not final)
Firefox v 139

```
// Since JS day 1

const now = new Date()
console.log(`${now}`)

now.setHours(19) // set to 7 PM
console.log(`${now}`)

// Mutates original 'now'
object
```

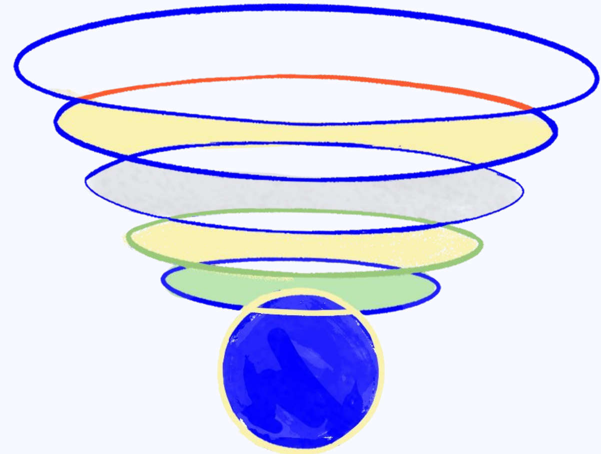
```
const now = Temporal.Now.zonedDateTimeISO()
console.log(`${now}`)

const changed = now.with({ hour: 19 })
console.log(`${now}` , `${changed}`)

// Can't mutate original
'now' object
```

Contents

1. Principles of JavaScript
2. Callbacks & Higher order functions
3. Closure (scope and execution context)
4. Type Coercion & Metaprogramming
5. **Asynchronous JavaScript & the event loop**
6. **Classes & Prototypes (OOP)**



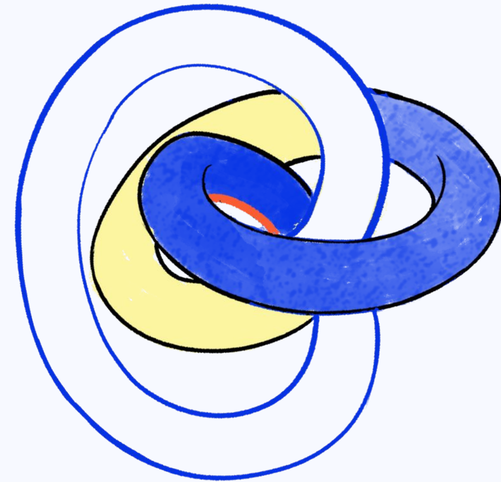
JavaScript the Hard Parts Day 2

JavaScript principles, Callbacks & Higher Order functions, Closure,

Type coercion, Classes/Prototypes & Asynchronicity

Contents

1. Principles of JavaScript
2. Callbacks & Higher order functions
3. Closure (scope and execution context)
4. Type Coercion & Metaprogramming
5. **Asynchronous JavaScript & the event loop**
6. Classes & Prototypes (OOP)



Promises, Async & the Event Loop

- Promises - the most signficant ES6 feature
- Asynchronicity - the feature that makes dynamic web applications possible
- The event loop - JavaScript's triage
- Microtask queue, Callback queue and Web Browser features (APIs)

A reminder of how JavaScript executes code

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


const output = multiplyBy2(num)
const newOutput = multiplyBy2(10)
```

Asynchronicity is the backbone of modern web development in JavaScript yet...

JavaScript is:

- Single threaded (one command runs at a time)
- Synchronously executed (each line is run in order the code appears)

Asynchronicity is the backbone of modern web development in JavaScript yet...

So what if we have a task:

- Accessing TikTok's server to get new video urls that takes a long time
- Code we want to run using those videos

Challenge: We want to wait for the videos links to be stored in `videos` so that they're there to run `displayVideos` on - but no code can run in the meantime

Slow function blocks further code from running (JS single thread)

```
const videos = getVideos("http://tiktok.com/will/1")
// ⛔350ms wait while a request is sent to Tiktok's HQ

displayVideos(videos)
// But we do need to wait for the videos to display them!

// more code to run
console.log("I want to runnnn!")
```

What if we try to delay a function directly using setTimeout?

Single thread - counting time passing is work so presumably also blocks?

```
function printHello(){
    console.log("Hello")
}

// Manually make our code wait (block?) 1000 ms
setTimeout(printHello, 1000)

console.log("Me first!")
```

In what order will our console logs appear?

So what about a delay of 0ms?

Now, in what order will our console logs occur?

```
function printHello(){
    console.log("Hello")
}

setTimeout(printHello, 0)

console.log("Me first!")
```

Our regular JavaScript model is failing us...

JavaScript is not enough to understand this - We need new pieces (some of which aren't JavaScript at all)

Our core JavaScript engine has 3 main parts:

- Thread of execution
- Memory/variable environment
- Call stack

We need to add some new components:

- Web Browser APIs/Node background APIs
- Promises
- Event loop, Callback/Task queue and micro task queue

ES5 solution: Introducing ‘callback functions’, and Web Browser APIs

```
function printHello(){ console.log("Hello") }

setTimeout(printHello, 1000)

console.log("Me first!")
```

What web browser features (APIs) do we have?

We’re interacting with a world outside of JavaScript now - so we need rules

```
function printHello(){ console.log("Hello") }
function blockFor1Sec(){
  //blocks in the JavaScript thread for 1 sec
}

setTimeout(printHello, 0)
blockFor1Sec()
console.log("Me first!")
```

ES5 Web Browser APIs with callback functions

Problems

- Our response data is only available in the callback function - Callback hell
- Maybe it feels a little odd to think of passing a function *into* another function only for it to run much later

Benefits

- Super explicit once you understand how it works under-the-hood
- Confidence in order of execution (if not exact timing)

Pair programming challenges

csbin.io/async

ES6+ Solution: Promises

Using two-pronged ‘facade’ functions that both:

- Initiate background web browser work and
- Return a placeholder object (promise) immediately in JavaScript

ES6+ Promises

```
function display(data){
    console.log(data)
}

const futureData = fetch('https://tiktok.com/will')
// [[Result]]

futureData.then(display) // [[FulfillReactions]]

console.log("Me first!")
```

ES6+ Solution (Promises)

Special objects built into JavaScript that get returned immediately when we make a call to a web browser API/feature (e.g. fetch) that's setup to return promises

Promises act as a placeholder for the data we expect to get back from the web browser features background work (stored in hidden property **[[Result]]**)

then method and functionality to call on completion

Any code (functions) we want to run on the retrieved (from internet) data must also be saved on the promise object

Added using .then method to the hidden property **[[FulfillReactions]]**

Promise objects will automatically trigger the attached function to run (with its input being the returned data

But we need to know how our promise-deferred functionality gets back into JavaScript to be run

```
function display(data){console.log(data)}
function printHello(){console.log("Hello")}
function blockFor300ms(){/* blocks js thread for 300ms }

setTimeout(printHello, 0)

const futureData = fetch('https://tiktok.com/will')
futureData.then(display)

blockFor300ms()
console.log("Me first!")
```

```
function display(data){console.log(data)}
function printHello(){console.log("Hello")}
function blockFor300ms(){/* blocks js thread for 300ms }

setTimeout(printHello, 0)

const futureData = fetch('https://tiktok.com/will')
futureData.then(display)

blockFor300ms()
console.log("Me first!")
```

Promises

Problems

- 99% of developers have no idea how they’re working under the hood
- Debugging becomes super-hard as a result
- Developers fail technical interviews

Benefits

- Cleaner readable style with pseudo-synchronous style code
- Nice error handling process **[[RejectReactions]]**

We have rules for the execution of our asynchronously delayed code

Hold promise-deferred functions in a microtask queue and callback function in a task queue (Callback queue) when the Web Browser Feature (API) finishes

Add the function to the Call stack (i.e. run the function) when:

- Call stack is empty & all global code run (Have the Event Loop check this condition)

Prioritize functions in the microtask queue over the Callback queue

```
function display(data){console.log(data)}
function printHello(){console.log("Hello")}
function blockFor300ms(){/* blocks js thread for 300ms }

setTimeout(printHello, 0)

const signal = AbortSignal.timeout(200)
const futureData = fetch('tiktok.com/will', {signal: signal})
futureData.then(display)
futureData.catch(display)

blockFor300ms()
console.log("Me first!")
```

WHATWG (2022)

Promises, Web APIs, the Callback & Microtask Queues and Event loop enable:

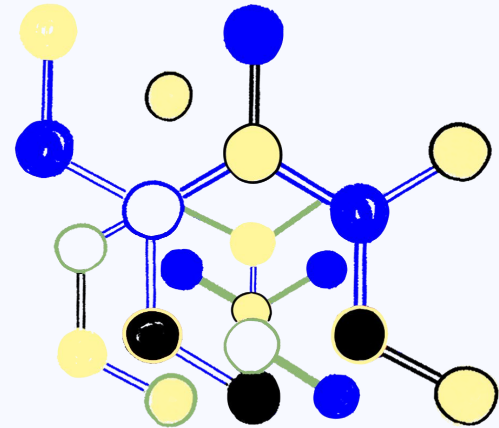
Non-blocking applications: This means we don't have to wait in the single thread and don't block further code from running

However long it takes: We cannot predict when our Browser feature's work will finish so we let JS handle *automatically* running the function on its completion

Web applications: Asynchronous JavaScript is the backbone of the modern web - letting us build fast 'non-blocking' applications

Contents

1. Principles of JavaScript
2. Callbacks & Higher order functions
3. Closure (scope and execution context)
4. Type Coercion & Metaprogramming
5. Asynchronous JavaScript & the event loop
6. **Classes & Prototypes (OOP)**



Classes, Prototypes - Object Oriented JavaScript

- An enormously popular paradigm for structuring our complex code
- Prototype chain - the feature behind-the-scenes that enables emulation of OOP but is a compelling tool in itself
- Understanding the difference between **[[Prototype]]** and prototype
- The **new** and **class** keywords as tools to automate our object & method creation

Core of development (and running code)

1. Save data (e.g. in a quiz game the scores of user1 and user2)
2. Run code (functions) using that data (e.g. increase user 2's score)

Easy! So why is development hard?

In a quiz game I need to save lots of users, but also *admins*, *quiz questions*, *quiz outcomes*, *league tables* - all have data and associated functionality

In 100,000 lines of code

- Where is the functionality when I need it?
- How do I make sure the functionality is only used on the right data!

That is, I want my code to be:

1. Easy to *reason about*

But also

2. Easy to add features to (new functionality)

3. Nevertheless efficient and performant

The Object-oriented paradigm aims is to let us achieve these three goals

So if I'm storing each user in my app with their respective data (let's simplify)

user1:

- name: 'Ari'
- score: 3

user2:

- name: 'Jae'
- score: 5

And the functionality I need to have for each user (again simplifying!)

- increment functionality (there'd be a ton of functions in practice)

How could I store my data and functionality together in one place?

Objects - store functions with their associated data!

This is the principle of encapsulation - and it's going to transform how we can 'reason about' our code

```
const user1 = {
  name: "Ari",
  score: 3,
  increment: function() { user1.score++; }
}

user1.increment() //user1.score -> 4
```

Creating user2 using dot notation

Declare an empty object and add properties with dot notation

```
const user2 = {} //create an empty object


//assign properties to that object
user2.name = "Jae"
user2.score = 5
user2.increment = function() {
  user2.score++
}
```

Creating user3 using `Object.create`

Object.create is going to give us fine-grained control over our object later on

```
const user3 = Object.create(null)


user3.name = "Tam"
user3.score = 9
user3.increment = function() {
  user3.score++
}
```

Solution 1. Generate objects using a function

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

const user1 = userCreator("Ari", 3)
const user2 = userCreator("Jae", 5)
user1.increment()
```

Solution 1. Generate objects using a function

Problems: Each time we create a new user we make space in our computer's memory for all our data and functions. But our functions are just copies.

Is there a better way?

Benefits: It's simple and easy to reason about!

Solution 2: Using the prototype chain

Store the increment function in just one object and have the interpreter, if it doesn't find the function on **user1**, look up to that object to check if it's there

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

Make the link with **`Object.create()`** technique

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("Ari", 3)
const user2 = userCreator("Jae", 5)
user1.increment()
```

What if we want to confirm our user1 has the property score

```
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("Ari", 3)
const user2 = userCreator("Jae", 5)
user1.hasOwnProperty('score')
```

What if we want to confirm our user1 has the property score

We can use the hasOwnProperty method - but where is it? Is it on user1? 🤔

All objects have a hidden [[Prototype]] property by default which defaults to linking to a big object - Object.prototype full of (somewhat) useful functions

We get access to it via userFunctionStore's [[Prototype]] property - the chain

Declaring & calling a new function *inside* our 'method' increment

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

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

const user1 = userCreator("Ari", 3)
const user2 = userCreator("Jae", 5)
user1.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("Ari", 3)
const user2 = userCreator("Jae", 5)
user1.increment()
```

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

Arrow functions override the normal *this* rules

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

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

const user1 = userCreator("Ari", 3)
const user2 = userCreator("Jae", 5)
user1.increment()
```

Solution 2: Using the prototype chain

Problems: No problems! It's beautiful. Maybe a little long-winded

Write this every single time - but it's 6 words!

```
const newUser = Object.create(userFunctionStore);

...

return newUser;
```

Benefits: Super sophisticated but not standard

Pair programming challenges

csbin.io/oop

Solution 3 - Introducing the keyword that automates the hard work: new

When we call the function that returns an object with ***new*** in front we automate 2 things

1. Create a new user object
2. Return the new user object

```
const user1 = new userCreator("Ari", 3)
const user2 = new userCreator("Jae", 5)
```

But now we need to adjust how we write the body of **userCreator** - how can we:

- Refer to the auto-created object?
- Know where to put our single copies of functions?

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++
}
const user1 = new userCreator("Ari", 3)
```

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

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("Ari", 3)
user1.increment()
```

Solution 3 - Introducing the keyword that automates the hard work: new

Benefits:

Faster to write. Often used in practice in professional code

Problems:

95% of developers have no idea how it works and therefore fail interviews

We have to upper case first letter of the function so we know it requires 'new' to work!

Solution 4: The class ‘syntactic sugar’

We’re writing our shared methods separately from our object ‘constructor’ itself (off in the **`userCreator.prototype`** object)

Other languages let us do this all in one place. ES2015 lets us too

Solution 4: The class ‘syntactic sugar’

```
class UserCreator {
  constructor (name, score){
    this.name = name
    this.score = score
  }
  increment (){ this.score++ }
  login (){ console.log("login") }
}

const user1 = new UserCreator("Ari", 3)
user1.increment()
```

Solution 4: The class ‘syntactic sugar’

```
class UserCreator {
  constructor (name, score){
    this.name = name
    this.score = score
  }
  increment (){ this.score++ }
  login (){ console.log("login") }
}
```

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

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

Solution 4: The class ‘syntactic sugar’

Benefits:

Emerging as a new standard

Feels more like style of other languages (e.g. Python)

Problems:

99% of developers have no idea how it works and therefore fail interviews

But you will not be one of them!

Public *static* fields - just a function on the class

ES2022

```
class UserCreator {
  static describe(){ console.log("Creates users") }
  constructor (name, score){
    this.name = name
    this.score = score
  }
  increment (){ this.score++ }
  login (){ console.log("login") }
}
const user1 = new UserCreator("Ari", 3)
UserCreator.describe() //logs "Creates users"
```

Public *instance* fields - every new object gets the property

ES2022

```
class UserCreator {
  loggedIn = false // great for default values
  constructor (name, score){
    this.name = name
    this.score = score
  }
  increment (){ this.score++ }
  login (){ this.loggedIn = true }
}
const user1 = new UserCreator("Ari", 3)
user1.login() //user1.loggedIn now true
```

Private *instance* fields - true encapsulation

ES2022

```
class UserCreator {
  #score // sets up a private property on the returned object
  constructor (name, score){
    this.name = name
    this.#score = score
  }
  increment (){ this.#score++ }
  login (){ this.loggedIn = true }
}
const user1 = new UserCreator("Ari", 3)
user1.increment()
// user1.#score now 4 (only accessible from increment 🕵️)
```

Private *static* fields - no tampering

Extension: Latest JavaScript features

ES2022

```
class UserCreator {
  static #count = 0 // only all user objs have access to 🕵️
  constructor (name, score){
    if (UserCreator.#count >= 2){throw Error("Max users reached")}
    this.name = name
    this.score = score
    UserCreator.#count++
  }
  increment (){ this.score++ }
}
const user1 = new UserCreator("Ari", 3)
const user2 = new UserCreator("Jae", 5)
const user3 = new UserCreator("Tam", 9) // Error: Max users reached
```

Prototype chain, ‘class’ sugar, and new field features give JavaScript serious OOP capabilities

Prototypal under-the-hood: Which means we're always implementing classic OOP features on top of a powerful (but distinct from object-oriented) system

Growing set of OOP features: the new and class keywords that automate prototypal work and our new private & public static & instance fields

Encapsulated, well-structured, resilient code: These tools let us emulate many strengths of classical OOP - but only if we understand the prototypal foundations