
JavaScript the Hard Parts
FRONTEND MASTERS
Will Sentance
CEO Codesmith

# Will Sentance

**Academic work:**
Oxford, Harvard

**Currently:**
CEO & Cofounder
Codesmith

**Previously:**
Cocreator @Icecomm
Software Engineer @Gem


# I teach softare engineering at Codesmith

### 1. Center of Software Engineering Excellence

- Recent Codesmith student projects have been featured at Google I/O and as Facebook's top developer tool
- Guest mentors include Brian Holt at LinkedIn, Gavin Doughtie at Google, Tom Occhino at Facebook
- Centered in LA, NYC and at Oxford University

### 2. Selective and tight-knit community

- Each selected student has shown enormous potential in five capacities that make an excellent engineer
- Students come from an exceptional and eclectic range of backgrounds from PhDs to software engineers, from Stanford graduates to self-teachers

### 3. Community of alumni for life

- 25% of graduates receive offers for Senior Engineer position and above, 70% receive offers for Mid-level Engineer (Codesmith graduate salaries range from $95k to $190k)
- Postgraduate education in advanced software architecture and Machine Learning

*Recent Codesmith graduates are building at* LinkedIn NIKE amazon Microsoft WEVR

# The team that makes it all possible

Will Sentance

Hira Qarni

Victoria Leon

Brandon Mizuno

Eric Kirsten

Haley Godtfredsen

Chris Cano

George Norberg

Sam Claney

Thai-Duong Nguyen

Weylin Wagnon

Jenny Mith

Jon McAlpine

Dhani Mayfield

Jimmy Huynh

Schno Mozingo

Mark Marcelo

# The 5 capacities we look for in candidates

1. Analytical problem solving with code
2. Technical communication (can I implement your approach just from your explanation)
3. Engineering best practices and approach (Debugging, code structure, patience and reference to documentation)
4. Non-technical communication (empathetic and thoughtful communication)
5. Language and computer science experience

# Our expectations

- Support each other - engineering empathy is the critical value at Codesmith
- Work hard, Work smart
- Thoughtful communication

# Frontend Masters - JavaScript the Hard Parts - Day 1

Part 1 – Principles of JavaScript – Thread, Execution context and Call stack

Part 2 – Callbacks and Higher order functions

Part 3 – Closure

# Frontend Masters - JavaScript the Hard Parts - Day 2

Part 4 – Asynchronous JavaScript

Part 5 - Object-oriented JavaScript – Approaches to OOP

## Principles of JavaScript

In JSHP we start with a set of fundamental principles

These tools will enable us to problem solve and communicate almost any scenario in JavaScript

— We'll start with an essential approach to get ourselves up to a shared level of understanding
— This approach will help us with the hard parts to come

# What happens when javascript executes (runs) my code?

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

# What happens when javascript executes (runs) my code?

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

As soon as we start running our code, we create a *global execution context*

— Thread of execution (parsing and executing the code line after line)
— Live memory of variables with data (known as a Global Variable Environment)

# The thread in JavaScript

— Single threaded (one thing at a time)

— Synchronous execution (for now)

# Running/calling/invoking a function

This is not the same as defining a function

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

const output = multiplyBy2(4);
const newOutput = multiplyBy2(10);
```

When you execute a function you create a new execution context comprising:

1. The thread of execution (we go through the code **in the function** line by line)
2. A local memory ('Variable environment') where anything defined in the function is stored

## We keep track of the functions being called in JavaScript with a Call stack

Tracks which execution context we are in - that is, what function is currently being run and where to return to after an execution context is popped off the stack

One global execution context, multiple function contexts

## Functional Programming

# Functional programming core features

1. Pure functions (no side effects)
2. 'Higher order functions' - highly valuable tool & often part of the Codesmith interview

# Create a function 10 squared

Takes no input

Returns 10*10

**How do we do it?**

# tensquared

```
function tensquared(){
    return 10*10;
}
tensquared(); // 100
```

# Now let's create a function that returns 9 squared

```
function ninesquared(){
    return 9*9;
}
ninesquared(); // 81
```

**Now 8 squared...and so on**

...

**We have a problem - it's getting repetitive, we're breaking our DRY principle**

**What could we do?**

## We can generalize the function

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

## We've generalized our function

Now we're only deciding what data to apply our multiplication functionality to when we *run* our function, not when we define it

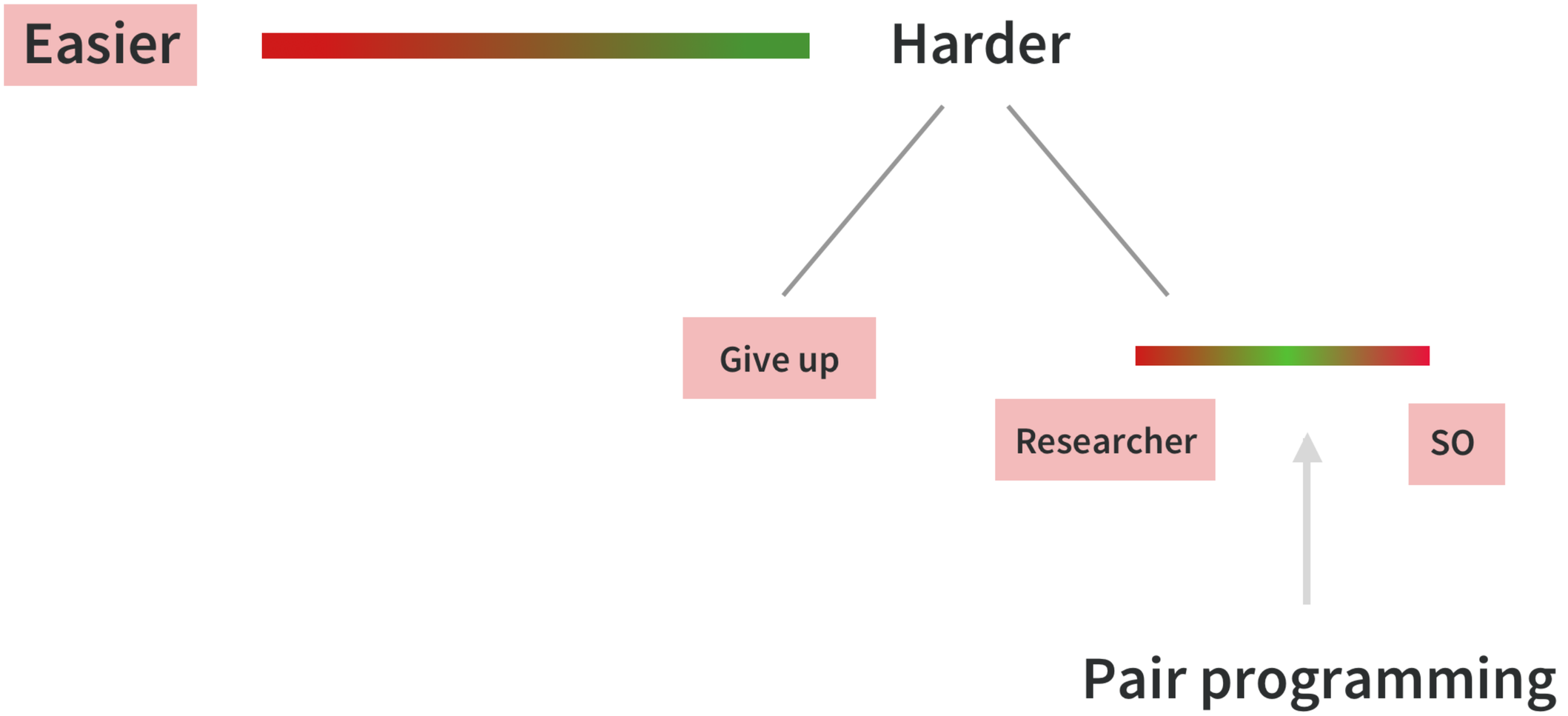
We're going to see later on that our higher order functions follow this same principle - that we may not want to decide exactly what our *functionality* is until we run our function

# Pair Programming

Answer these:

— I know what a variable is
— I've created a function before
— I've added a CSS style before
— I have implemented a sort algorithm (bubble, merge etc)
— I can add a method to an object’s prototype
— I understand the event loop in JavaScript
— I understand 'callback functions'
— I've built a project in React or Angular
— I can handle collisions in hash tables

# Pair programming



# Pair Programming

http://csbin.io/callbacks

**Now suppose we have a function copyArrayAndMultiplyBy2. Let's diagram it out**

```
function copyArrayAndMultiplyBy2(array) {
  let output = [];
  for (let i = 0; i < array.length; i++) {
    output.push(array[i] * 2);
  }
  return output;
}
const myArray = [1,2,3]
let result = copyArrayAndMultiplyBy2(myArray)
```

## What if want to copy array and divide by 2?

```
function copyArrayAndDivideBy2(array) {
  let output = [];
  for (let i = 0; i < array.length; i++) {
    output.push(array[i] /2);
  }
  return output;
}
const myArray = [1,2,3]
let result = copyArrayAndDivideBy2(myArray);
```

## Or add 3?

```
function copyArrayAndAdd3(array) {
  let output = [];
  for (let i = 0; i < array.length; i++) {
    output.push(array[i] +3);
  }
  return output;
}
const myArray = [1,2,3]
let result = copyArrayAndAdd3(myArray);
```

What principle are we breaking?

# We're breaking DRY

# What could we do?

# We could generalize our function so that we pass in our specific instruction only when we run the copyArrayAndManipulate function!

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


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


let result = copyArrayAndManipulate([1, 2, 3], multiplyBy2);
```

# Back to pairing

# 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

# Callback vs. Higher-order function

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

```
function multiplyBy2(input) {
  return input * 2;
}
let result = copyArrayAndManipulate([1, 2, 3], multiplyBy2);
```

**Which is our callback function?**

**Which is our higher order function?**

# Callback vs. Higher-order function

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

```
function multiplyBy2(input) {
  return input * 2;
}
let result = copyArrayAndManipulate([1, 2, 3], multiplyBy2);
```

**The function we *pass in* is a callback function**

**The outer function that *takes in* the function (our callback) is a higher-order 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

# So callbacks and higher order functions simplify our code and keep it DRY

And they do something even more powerful

# They allow us to run asynchronous code

# Frontend Masters - JavaScript the Hard Parts - Day 1

Part 1 – Principles of JavaScript – Thread, Execution context and Call stack ✓

Part 2 – Callbacks and Higher order functions ✓

**Part 3 – Closure**

# Frontend Masters - JavaScript the Hard Parts - Day 2

Part 4 – Asynchronous JavaScript

Part 5 - Object-oriented JavaScript – Approaches to OOP

# Closure

When our functions get called, we create a live store of data (local memory/ variable environment/state) for that function's execution context.

When the function finishes executing, its local memory is deleted (except the returned value)

But what if our functions could hold on to live data/state between executions? 😲

This would let our function definitions have an associated cache/persistent memory

**But it starts with returning us returning a function from another function**

# We just saw that functions can be returned from other functions in JavaScript

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

let generatedFunc = instructionGenerator()
```

How can we run/call multiplyBy2 now?

# Let's call (run) our generated function with the input 3

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

let generatedFunc = instructionGenerator()

let result = generatedFunc(3) //6
```

# Pair up

Answer these:

— I know what a variable is
— I've created a function before
— I've added a CSS style before
— I have implemented a sort algorithm (bubble, merge etc)
— I can add a method to an object’s prototype
— I understand the event loop in JavaScript
— I understand 'callback functions'
— I've built a project in React or Angular
— I can handle collisions in hash tables

# Pair Programming

http://csbin.io/closures

# Calling a function in the same scope as it was defined

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

outer();
```

Where you define your functions determines what variables your function have access to when you call the function

**But what if we call our function outside of where it was defined?**

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

outer()

incrementCounter();
```

**What happens here?**

# There is a way to run a function outside where it was defined - return the function and assign it to a new variable

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

var myNewFunction = outer(); // myNewFunction = incrementCounter
```

# Now we can run incrementCounter in the global context through its new label myNewFunction

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

let myNewFunction = outer(); // myNewFunction = incrementCounter
myNewFunction();
```

# What happens if we execute myNewFunction again?

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

let myNewFunction = outer(); // myNewFunction = incrementCounter
myNewFunction();
myNewFunction();
```

# Lexical Scope

When a function is defined, it gets a [[scope]] property that references the Local Memory/Variable Environment in which it has been defined

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

```
let myNewFunction = outer(); // myNewFunction = incrementCounter
myNewFunction();
myNewFunction();
```

Wherever we call that incrementCounter function - it will always look first in its immediate local memory (variable environment), and then in the [[scope]] property next before it looks any further up

## JavaScript static/lexical scoping

This is what it means when we say JavaScript is lexically or statically scoped

Our lexical scope (the available live data when our function was *defined*) is what determines our available variables and prioritization at function execution, **not** where our function is *called*

# What if we run 'outer' again and store the returned 'incrementCounter' in 'anotherFunction'

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

```
let myNewFunction = outer();
myNewFunction();
myNewFunction();

var anotherFunction = outer(); // myNewFunction = incrementCounter
anotherFunction();
anotherFunction();
```

# Back to Pair-programming

# The power of Closure

Now: Our functions get 'memories' - once, memoize

Advanced: We can implement the module pattern in JavaScript

# Frontend Masters - JavaScript the Hard Parts - Day 1

Part 1 – Principles of JavaScript – Thread, Execution context and Call stack ✓

Part 2 – Callbacks and Higher order functions ✓

Part 3 – Closure ✓

# Frontend Masters - JavaScript the Hard Parts - Day 2

## Part 4 – Asynchronous JavaScript

Part 5 - Object-oriented JavaScript – Approaches to OOP

# Asynchronous JavaScript

# Asynchronicity is the backbone of modern web development in JavaScript

JavaScript is single threaded (one command executing at a time) and has a synchronous execution model (each line is executed in order the code appears)

So what if we need to wait some time before we can execute certain bits of code? We need to wait on fresh data from an API/server request or for a timer to complete and then execute our code

We have a conundrum - a tension between wanting to delay some code execution but not wanting to block the thread from any further code running while we wait

What do we do? Let’s see two examples

# In what order will our console logs occur?

```
function printHello(){
    console.log(“Hello”);
}

setTimeout(printHello,1000);

console.log(“Me first!”);
```

No blocking!?

## In what order will our console logs occur?

```
function printHello(){
    console.log(“Hello”);
}

setTimeout(printHello,0);

console.log(“Me first!”);
```

Our previous model of JavaScript execution is insufficient

# We need to introduce 3 new components of our platform

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

**Adding**

— Web Browser APIs/Node background threads
— Callback/Message queue
— Event loop

# Let's see the first of these (the Web Browser API) in action

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

setTimeout(printHello,0);

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

# Pair programming

# But now we are interacting with a world outside of JavaScript

We need a way of predictably understanding how this outside world will interact with our JavaScript execution model. What would happen here?

```
function printHello(){
    console.log(“Hello”);
}

function blockFor1Sec(){
    //blocks in the JavaScript thread for 1 second
}

setTimeout(printHello,0);

blockFor1Sec()

console.log(“Me first!”);
```

## We have two rules for the execution of our asynchronously delayed code

1. Hold each deferred function in a queue (the Callback Queue) when the API ‘completes’
2. Add the function to the Call stack (i.e. execute the function) ONLY when the call stack is totally empty (Have the Event Loop check this condition)

## There are many things where waiting would block our thread and we use Browser APIs for instead

— A timer to finish running

— New information from a server (Ajax)

— Indication that a portion of the page has loaded

— User interaction (clicks, mouseovers, drags)

— Writing/Reading to File system (Node)

— Writing/reading database (Node)

## Some come back with data. The design of the Browser API we are using determines how we access the returned data

That we were waiting on to run our deferred functionality

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

$.get("http://twitter.com/willsen/tweet/1", display);

console.log(“Me first!”);
```

Asynchronous callbacks, Web APIs, the Callback Queue and Event loop allow us to defer our actions until the ‘work’ (an API request, timer etc) is completed and continue running our code line by line in the meantime

---

# Asynchronous JavaScript is the backbone of the modern web - letting us build fast ‘non-blocking’ applications

# Frontend Masters - JavaScript the Hard Parts - Day 1

Part 1 – Principles of JavaScript – Thread, Execution context and Call stack ✓

Part 2 – Callbacks and Higher order functions ✓

Part 3 – Closure ✓

# Frontend Masters - JavaScript the Hard Parts - Day 2

Part 4 – Asynchronous JavaScript ✓

**Part 5 - Object-oriented JavaScript – Approaches to OOP**

# OOP - an enormously popular paradigm for structuring our complex code

[EXPAND ON CORE FEATURES]

# We're building a quiz game with users

# Some of our users

```
Name: Will
Score: 3

Name: Tim
Score: 6

Functionality
+ Ability to increase score
```

What would be the best way to store this data and functionality?

# Objects - store functions with their associated data!

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

user1.increment(); //user1.score => 4
```

# What alternative techniques do we have for creating objects?

# Creating user2 user 'dot notation'

```
let user2 = {}; //create an empty object

user2.name = "Tim"; //assign properties to that object
user2.score = 6;
user2.increment = function() {
  user2.score++;
};
```

# Creating user3 using Object.create

```
let user3 = Object.create(null);

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

Our code is getting repetitive, we're breaking our DRY principle

And suppose we have millions of users!

What could we do?

# Solution 1. Generate objects using a function

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


//later

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

**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!

# Pair up

Answer these:

— I know what a variable is
— I know what an object is (in programming)
— I've added a method to an object’s prototype before
— I've added a CSS style before
— I understand 'callback functions'
— I can explain the event loop in JavaScript
— I've built a project in React or Angular
— I can explain closure in JavaScript
— I can handle collisions in a hash table

# Pair Programming

http://csbin.io/oop

## Solution 2:

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

How to make this link?

# Using the prototypal nature of JavaScript - Solution 2 in full

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


let userFunctionStore = {
  increment: function(){this.score++;},
  login: function(){console.log("You're loggedin");}
};

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

## Problem

No problems! It's beautiful

Maybe a little long-winded

```
let newUser = Object.create(functionStore);
...
return newUser
```

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

Super sophisticated but not standard

## Solution 3

Introduce magic keyword `new`

```
let user1 = new userCreator("Will", 3)
```

What happens when we invoke `userCreator("Will", 3)` **without** the `new` keyword?

When we call the constructor function with `new` in front we automate 2 things

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

# The new keyword automates a lot of our manual work

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

functionStore userCreator.prototype // {};
functionStore userCreator.prototype.increment = function(){
 this.score++;
}

let user1 = new userCreator("Will", 3);
```

# Complete Solution 3

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

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

let user1 = new User(“Eva”, 9)

user1.increment();
```

# Benefits

— Faster to write

— Still typical practice in professional code

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

## Solution 4

We're writing our shared methods separately from our object 'constructor' itself (off in the `User.prototype` object)

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

# The class ‘syntactic sugar’

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

let user1 = new User("Eva", 9);

user1.increment();
```

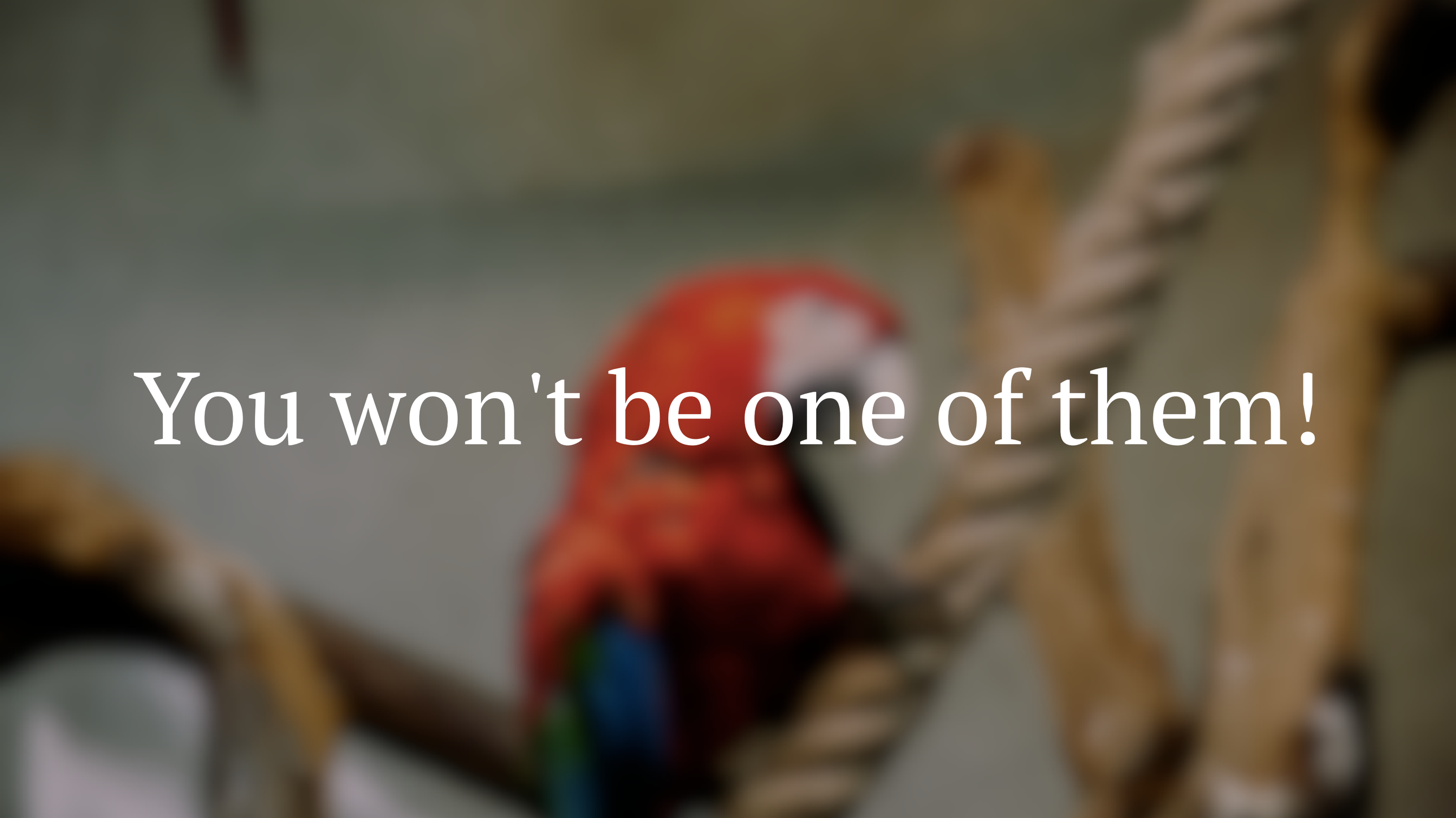
# [Side by side comparison slide - DONE]

**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


You won't be one of them!

# The Hard Parts Challenge Code

— 90% of accepted students attend JSHP

— We created the Hard Parts Challenge code to guarantee an interview for the Hard Parts community members

— It builds upon the OOP content you worked on today

— Drinks