

JavaScript the Hard Parts

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

Will Sentance



Currently

CEO, Cofounder: Codesmith

Speaker: Frontend Masters, BBC

Previously

Cocreator & Engineer: Icecomm

Software Engineer: Gem

Academic Work:

Oxford University,

Harvard University

What is Codesmith?

12 week full-time software engineering immersive program in Los Angeles & New York

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



amazon

Software Engineer



Software Application Developer



Google

Software Engineer

200+

Graduates

Per year from Codesmith LA and Codesmith NY

92%

Hired within 180 days

Excludes ineligible to work in the US, health issues or no response to 6 contacts (full report at codesmith.io for details)

\$110k

Median starting salary

\$112k in NY, \$106k in LA
(Third-party audited for CIRR; Jan-June 2018 reporting period)

50,000+

Github stars

Projects by Codesmith students have achieved global acclaim

What to focus on in the workshop



Analytical problem
solving



Technical
communication



Engineering
approach



Non-technical
communication



JavaScript and programming
experience

Contents

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

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

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

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



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

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

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



Contents

1. Principles of JavaScript
2. **Callbacks & Higher order functions**
3. Closure (scope and execution context)
4. Asynchronous JavaScript & the event loop
5. 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
- Forms the backbone of the Codesmith technical interview (and professional mid/senior level engineering interviews)

Why do we even have functions?

Let's see why...

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() // 100
```



And an a 8 squared function? 125 squared?

What principle are we breaking?

nineSquared

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

```
nineSquared() // 100
```



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

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

Stackoverflow

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 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 can implement filter
- I can handle collisions in a hash table

For each topic you know give yourself a point to get a total out of

Now suppose we have a function `copyArrayAndMultiplyBy2`

```
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);
```

What if want to copy array and divide by 2?

```
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);
```

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?*

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

We could generalize our function - So we pass in our specific instruction only when we run `copyArrayAndManipulate` !

```
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);
```

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

Introducing arrow functions - a shorthand way to save functions

```
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
```

Updating our callback function as an arrow function

```
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);
```


We can even pass in multiplyBy2 directly without a name

But it's still just the code of a function being passed into copyArrayAndManipulate

```
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);
```



Anonymous and arrow functions

- Improve immediate legibility of the code
- But at least for our purposes here they are ‘syntactic sugar’ - we’ll see their full effects later
- Understanding how they’re working under-the-hood is vital to avoid confusion

Contents

1. Principles of JavaScript
2. Callbacks & Higher order functions
3. **Closure (scope and execution context)**
4. Asynchronous JavaScript & the event loop
5. 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

Functions get a new memory every run/invocation

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

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

Functions with memories

- 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 between executions?
- This would let our function definitions have an associated cache/persistent memory
- But it all starts with us **returning a function from another function**



Functions can be returned from other functions in JavaScript

```
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 incrementCounter (){  
    counter ++;  
  }  
  incrementCounter();  
}  
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 incrementCounter (){ counter ++; }  
  return incrementCounter;  
}
```

```
const myNewFunction = outer();  
myNewFunction();  
myNewFunction();
```

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

```
const myNewFunction = outer();  
myNewFunction();  
myNewFunction();
```

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

Individual backpacks

If we run 'outer' again and store the returned 'incrementCounter' function definition in 'anotherFunction', this new incrementCounter 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. **Asynchronous JavaScript & the event loop**
5. Classes & Prototypes (OOP)

Promises, Async & the Event Loop

- Promises - the most significant 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)

So what if we have a task:

- Accessing Twitter's server to get new tweets that takes a long time
- Code we want to run using those tweets

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

Slow function blocks further code running

```
const tweets = getTweets("http://twitter.com/will/1")  
  
// 🛑 350ms wait while a request is sent to Twitter HQ  
  
displayTweets(tweets)  
  
// more code to run  
console.log("I want to runnnn!")
```

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

setTimeout is a built in function - its first argument is the function to delay followed by ms to delay by

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

```
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!");
```

JavaScript is not enough - 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!");
```

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

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://twitter.com/will/tweets/1')  
futureData.then(display);  
  
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 set up to return promises (not all are)

Promises act as a placeholder for the data we expect to get back from the web browser feature's background work

***then* method and functionality to call on completion**

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

Added using `.then` method to the hidden property `'onFulfilment'`

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://twitter.com/will/tweets/1')
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

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

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. Asynchronous JavaScript & the event loop
5. **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 `__proto__` 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: 'Tim'
- score: 3

user2:

- name: 'Stephanie'
- 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: "Will",  
  score: 3,  
  increment: function() { user1.score++; }  
};
```

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

Let's keep creating our objects. What alternative techniques do we have for creating objects?

Creating user2 user 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 = "Tim";
user2.score = 6;
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 = "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) {  
  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()
```

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

*We can use the hasOwnProperty method -
but where is it?*

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

Let's start by simplifying (just increment method - written over 3 lines now)

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?



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

Now our inner function gets its this set by where it was saved - it's a 'lexically scoped this'

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("Eva", 9)
const user2 = new userCreator("Tim", 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("Will", 3);
```

Automates the hard work

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("Eva", 9)
```

```
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 do so 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("Eva", 9);
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"); };
```

```
const user1 = new UserCreator("Eva", 9);  
user1.increment();
```

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!

Fin

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