JavaScript The Hard Parts

Slides Link

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

Execution context : *Execution context* is a concept in the language spec that—in layman's terms—roughly equates to the 'environment' a function executes in; that is, variable scope (and the *scope chain*, variables in closures from outer scopes), function arguments, and the value of the this object.
Created to run the code of a function - has 2 parts (we've already seen them!)

- Thread of execution
- Memory

Call Stack: A **call stack** is a mechanism for an interpreter (like the JavaScript interpreter in a web browser) to keep track of its place in a script that calls multiple functions — what function is currently being run and what functions are called from within that function, etc.
We keep a track of the functions we are running and where our thread of execution is using a **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

DRY principle: It means refactoring code by taking something done several times and turning it into a loop or a function. DRY code is easy to change, because you only have to make any change in one place.

In JavaScript, functions are first class objects.

1. They can be assigned to variables and properties of other objects
2. Passed as arguments into functions
3. Returned as values from functions

Higher Order Functions:A function that accepts and/or returns another function is called a **higher-order function**.
It's *higher-order* because instead of strings, numbers, or booleans, it goes higher to operate on functions.

Callback Functions: In JavaScript, a callback function is a function that is passed into another function as an argument.
Callbacks and HOF simplify our code and keep it dry.

- Declarative readable code: Map, filter, reduce - the most readable way to write code to work with data
- Asynchronous JavaScript: Callbacks are a core aspect of async JavaScript, and are under-the-hood of promises, async/await

Arrow functions

- Improve immediate legibility(clear enough to read) of the code.
- Syntactic sugar.
- Understanding how they're working under-the-hood is vital to avoid confusion.

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

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

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

Memoization - giving our functions persistent memories of their previous input output combinations. When a function is called, memoization **stores the function results before it returns the result to the function caller**.

Lexical scope: A lexical scope in JavaScript means that **a variable defined outside a function can be accessible inside another function defined after the variable declaration**. But the opposite is not true; the variables defined inside a function will not be accessible outside that function.

Promises, Async & the Event Loop

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 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

JavaScript Event Loop

https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

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

Microtask Queue

https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API/Microtask_guide/In_depth

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)

Prioritise 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

Classes, Prototypes - Object Oriented JavaScript

syntatic sugar - something that changes the way it looks, but doesnt change under the hood. eg: regual function expression vs arrow functions

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

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

Problems: No problems! It's beautiful. Maybe a little long-winded Write this every single time - but it's 6 words!

Benefits: Super sophisticated but not standard

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.

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?

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

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!