Javascript Notes

Part 4

Topics Covered

- The Call stack
- Scope and scope chain
- Types of scope
- Scope chain Vs Call stack
- Hoisting
- Destructuring Assignment
- Destructuring Arrays
- Destructuring Objects
- Calling a method with an object

⊛ The call stack

A place where execution context get stacked on top of each other, to keep track of where we are in the execution.

⊛ Scope and the Scope chain

Scoping controls how our program's variables are organized and accessed.

Scoping asks this question "where do variables live"?

or

"where we can access a certain variable, and where not?"

Lexical scoping → Scoping is controlled by placement of functions and blocks in the code.

Scope → It is a space or environment in which a certain variable is declared (variable environment in case of functions). ~~[illegible]~~ There is global scope, function scope, and block scope.

Scope of a variable → Region of our code where a certain variable can be accessed.

(*) The three types of scope ->

(*) Javascript variable have 3 types of scope.

(i) Block scope

(ii) function scope

(iii) Global scope.

Global scope -> This is for variables that are declared outside of any function or block.

These variables will be accessible everywhere.

ex ->

```
const name = 'Yash';
const job = 'coder';
const year = 1999;
```

function scope -> Each and every function creates a scope and the variable declared inside that function scope are only accessible inside that function. This is also called a local scope. Outside of the function, the variable are not accessible at all.

ex ->

```
function calcAge (birthyear)
{
    const now = 2024;
    const age = now - birthyear;
    return age;
}
```

Since local variable are only recognized inside their junction, variable with the same name can be used in different junctions.

Local variable are created when a junction starts and deleted when the junction is completed.

Block Scope (ES6) → Variables are accessible only inside block (block scoped)

However, this only applies to let and const variables.
→ function are also block scoped (only in strict mode).

Note:-) Before ES6(2015), Javascript variables had only Global scope and function scope.
ES6 introduced two important new javascript keywords: let and const.
These two keywords provide Block scope in Javascript.

variable lookup → when a variable is not in the current scope, the engine looks up in the scope chain until it finds the variable it's looking for.

(*) Difference between scope chain and call stack.

→ The scope chain is a one-way street : a scope will never, ever have access to the variables of an inner scope;

→ The scope chain in a certain scope is equal to adding together all the variable environments of all the parent scopes.

⊛ Hoisting in Javascript.

Hoisting makes some types of variables accessible/usable in the code before they are actually declared. "Variables lifted to the top of their scope".

↓ Behind the scenes.

Before execution, code is scanned for variable declaration, and for each variable, a new property is created in the variable environment object.

	Hoisted ?	Initial value	scope
function declaration	Yes	Actual function	Block
Var variables	Yes	undefined	function
let & const variables	No	<uninitialized>, TDZ Temporal dead zone	Block
function expressions & arrows	Depends if using var or let/const.		

⊛ Destructuring Assigment.

The two most used data structures in Javascript are Object and Array

- object allow us to create a single entity that stores data items by key.
- Arrays allows us to gather data items into an ordered list.

Destructuring assignment is a special syntax that allow us to "unpack" arrays or objects into a bunch of variables, as sometimes that's more convenient.

Destructuring also works well with complex functions that have a lot of parameters, default values, and so on.

#. Destructuring Array →

```
const arr = [2, 3, 4];
 const a = arr[0];
 const b = arr[1];
  const c = arr[2];
```

but we can destructure it like this.

```
const [x, y, z] = arr;
      console.log(x, y, z);
```

another example.

```
// we have an array with a name and surname
    let arr = ["John", "smith"]
    let [firstName, surName] = arr;
        console.log(firstName, surNane);
```

Note:→ while destructuring, the original array remains the same.

another example.

```
const restaurant = {
            name: 'classico Italiano',
        location: 'Via Angelo Tavanti 23',
    categories: ['Italian', 'Pizzeria', 'vegetarian',
                                   organic'],

  StarterMenu: ['focaccia', 'Bruschetta', 'Garlic bread',
                            'caprese salad'],

  mainMenu: ['Pizza', 'Pasta', Risotto'];

  };
```

now let say I want to take 1st and 3rd element from categories then It will be like this

```
const [first, , second] = restaurant.categories;
    console.log(first, second];
```

Just skip that element

another example

—— x —— x ——

suppose if we want to swap the two values.

```
let [main, ,secondary] = restaurant.categories;
    console.log(main, secondary);
```

we can swap it like this →

```
[main, secondary] = [secondary, main]
        console.log(main, secondary);
```

another example

—— x —— x —— x →

An array inside another array.

```
const nested = [2, 4, [5,6]];
    const [i, , j] = nested;
```

But if we want the individual values then,

```
const [i, , [j, k]] = nested;
```

If we want to set default values.

```
const [p=1, q=1, r=1] = [8, 9]
console.log(p, q, r)
```

Here the output is → 8, 9, 1.

#. Destructuring Objects

```
const restaurant = {
        name: 'classico Italiano',
    location: 'via Angelo Tavanti 23, firenze, Italy',
    categories: ['Italian', 'Pizzeria', 'vegetarian', 'organic'],
    starterMenu: ['Focaccia', 'Bruschetta', 'Garlic bread', '
                                              caprese salad'],

mainMenu: ['Pizza', 'Pasta', 'Risotto'],
openingHours: {
            thu: { open: 12,
                   close: 22,
            },
            fri: { open: 11,
                   close: 23,
            },
            sat: {
                   open: 0,
                   close: 24,
            },
        },
```

```
const {name, openingHours, categories} = restaurant.
```

on objects curly braces will come

Need to use the exact property name

If we wanted the variables names to be different, then we can do it like this,

```
const {name: restaurantName, openingHours: hours, categories: tags}
                                        = restaurant;
```

$\underbrace{\text{name}}_{\text{Exact property name}}$: $\underbrace{\text{restaurantName}}_{\text{variable name of our choice}}$

```
console.log(restaurantName, hours, tags);
```

we can also set default values like this

```
const { menu = [], starterMenu: starters = [] } = restaurant.
```

$\boxed{\text{menu = []}}$ → This is how we can set default value while destructuring objects.

→ mutating variables.

```
let a = 111;
let b = 999
const obj = {a: 23, b: 7, c: 14};

({a, b} = obj);
```

We have to enclose it in paranthesess otherwise it will give error.

* nested objects.

```
const {fri : {open, close}} = openingHours;
```
exact property name.

```
console.log (open, close)
```

(*) calling a method with an object.

In the restaurant data we will create a method.

```
orderDelivery : function({starterIndex, mainIndex, time,
                                        address})

        {console.log (obj);
                      ↳ starterIndex, mainIndex, time, address
        }
```

```
restaurant.orderdelivery({
                         time: '22:30',
                         address : 'Via del Sole, 21',
                         mainIndex: 2,
                         StarterIndex: 2,
                         });
```

Follow for more

Thank You !!!