JS Fundamentals

Credits : These are my notes from various JS Course : Javascript Mastery, Sheriyan coding school, Namaste JS : Akshay Saini

Socials :

- Hardikk kamboj Twitter
- Hardikk Kamboj LinkedIn

Variables

- Let - can be changed -> braces scoped -> cannot be aaded to window object
- const - constants in js - cannot be changed
- Var - can be changed (with some scope constraints) -> scope is inside function -> adds itself to window object
- Valid First characters for naming -> $ or _
- Multiple same name vars cannot be there

Data Types

- String - sigle, double and backticks(``) qoutes. - Can use JS in it - ${ js inside here}
- Number - Number normal
- Null - null value (can be assigned)
- Undefined - no defined value (cannot be assigned)
- Typeof- to know the type of data type
- Object - group variables and store group of data types. `const object1={ name :'John', age:25,} with dot notation to access the property.`
- Array -> object

statically typed vs dynamically typed languages

js - dynamic - var can hold any data
c++ - statically typed

Operations

-> `+,-,*,/`
-> modulo - %
-> exponent - `** -> comparison` < > (is equal) (true or false),!=(not equal), = (strict equals)`

Strict vs Loose equality

```
== vs ===
5===5 -> true
5 == "5" -> true
5 === "5" -> false

=== -> compares both values and Data types
- true only when both are equal (Use it more )

== -> Doesnt compare data types
```

Logical op

- AND && -> checks all and returns the last value (truthy) if false is encountered (return false value)
- OR || -> returns the first value (truthy) (returns the first true) and if false then the last false value
- NOT -> ! -> reverse bool value

TRUTHY VS FALSY

- 1, string, any other number, { }, [ ]- truthy
- 0, null, false, ' ', Nan, Undefined -> falsy

Switch Statements

```
switch(value){
case 'choice':
        console.log("stiuff")
        break;
```

```
default:
        cosole.log("default value")
}
```

Ternary Operator :

```
if(){

}
else if{

}
else{

}

or

? : notation

condition ? True : false
```

Loops

```
1. For loop - for(let i=0; i<something; i++)
2. While loop - While(condition){
        what to do
```

```
}
DRY PRINCIPAL -> DO Repeat Yourself
```

Functions

```
// Function declaration - defining
// Function Call -> calling/executing the function

function testing(parameter){
        return parameter*parameter;
}

// call
const result = testing(argument to be passed)

result stores the value

// Annonomous function

const name = function(params){
// statements
}

// Arrow Functions

const name = (params) => {
        // statements
}

// Arrow with only one line
```

```
const name = (params) => params * params


// Function with defualt value

const add = (a=0,b=0) => {
return a + b;
}

const result = add(2);
```

Scope

- Where are the variables we make are available
- block, function and global scope

```
// Global scope

const name = 'john'

const logName = () => {
        console.log(name);
}
logName();

// Local Scope / function scope

const hey = () => {
        // Local to this function
        const name = 'phil'
        console.log(name);
```

```
}
```

Hoisting

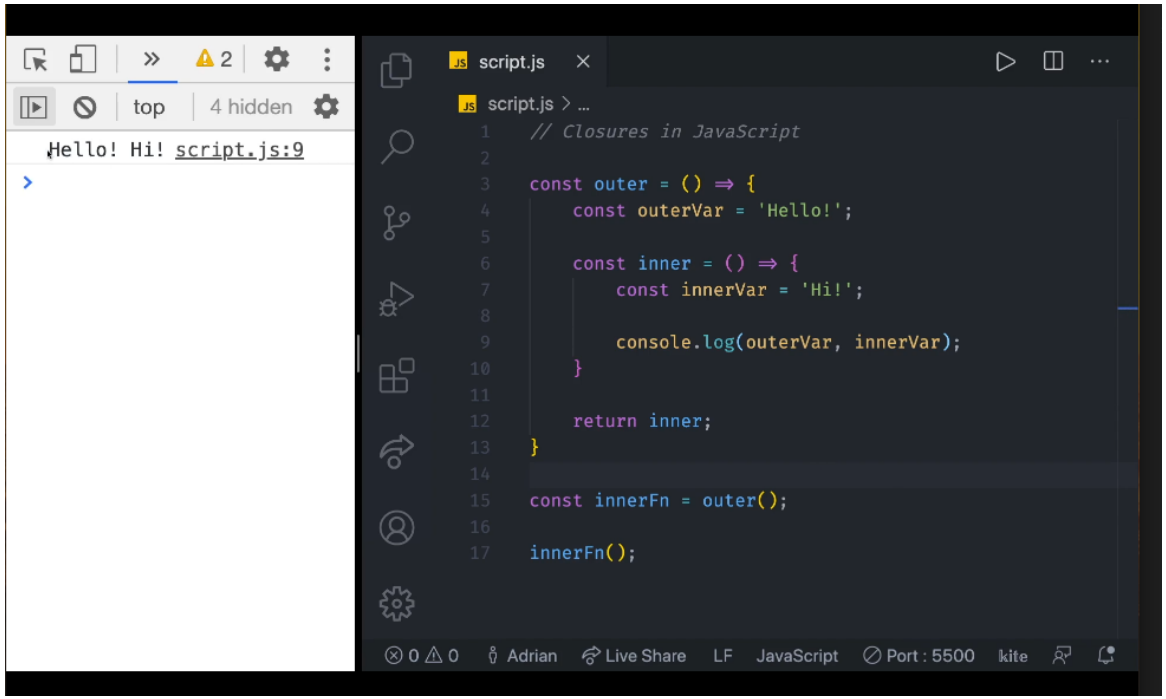
- JS mechanism where variable and function declarations are moved to the top of their scope before code execution.

```
console.log(age);

var age = 20; // only the declarations goes to the top not the value;

--> undefined -> hoisting

case 2.

var hoist
console.log(hoist);
hoist = "something"

--> undefined ( hoisting )

case 3 :
function hoist(){
console.log(message);
var message = 'test';
}
hoist();

--> undefined

if function call is done first
```

```
hoist();
function hoist(){
var message = 'test';
console.log(message);

}
--> Test
Hoisting was done with function is time

// For consts and let variable
console.log(age);
const age = 25

--> error (no hoisting in modern js for consts and let)



hoist();
const hoist = () => {
console.log(message)
}

--> error
```

Closures

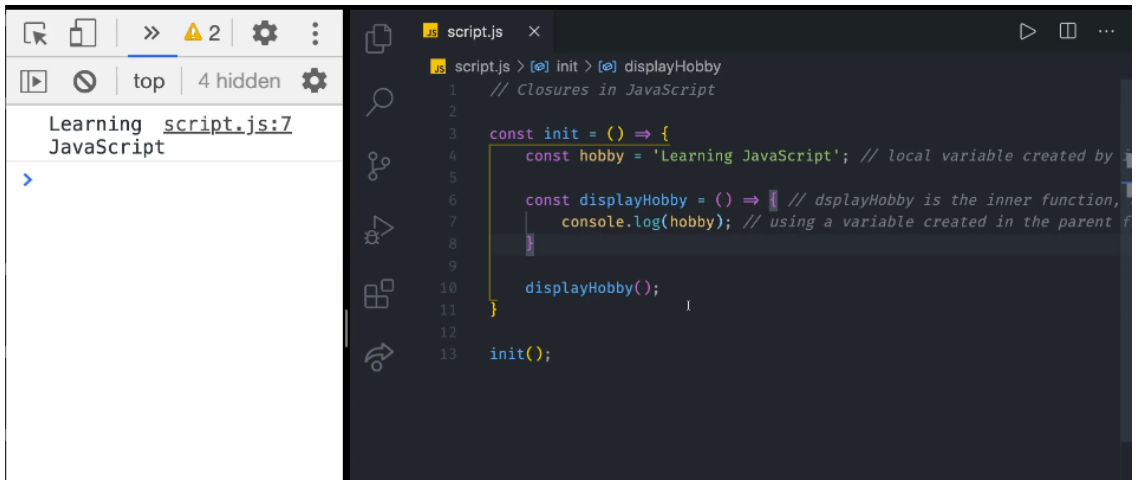
- As soon as the function is done with the execution all of it's content is gone, and that's why we cannot access the local scope var outside the function.
- Function inside a function



--> Access to the var of the parent scope : due to the closures.

-> that's why we get both Hi and Hello above

-> The inner data is not getting deleted here.



Strings

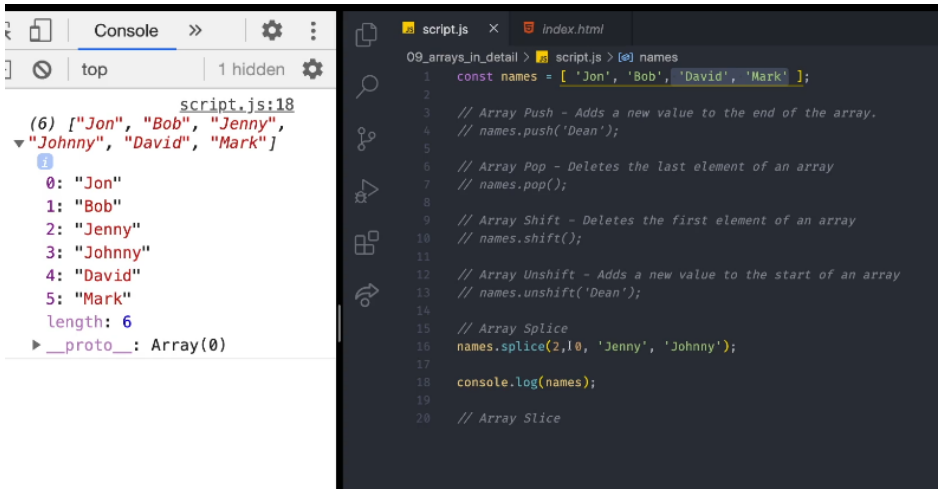
- types `' ', " ", ``` `
- property `string.length, string[position], Cases : string.toLowerCase(), string.toUpperCase() -> store them in a new variable as well, String.indexOf('word') -> searchs the first substring, save it to a var as well, string.includes('word')-> true or false`
- Substing of a string - slice(start,end)
- Split a string into single chars -split method
  string.split(' '); splits it on a each character, word. -> turns into an array
- Reverse - reverse method works on arrays ->

  First we turn the string into an array using the split method and then use reverse method to reverse the characters.
  const reversed = string1.split("").reverse().join
- String.trim() -> removes the spaces : used for storing emails.

Arrays

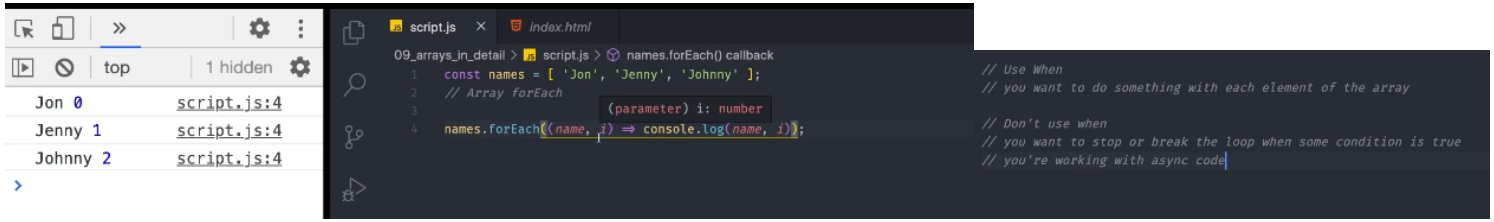
- Ordered Collection of Data
- const array = ['ele1', 'el2'];

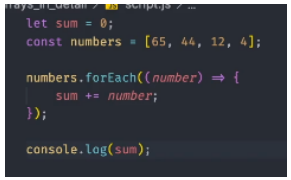
- loop over array -> for loop
- Array methods - push, pop, shift(remove first element of array), unshift(adds new value to the start of the array)
- Array Splice and Array Slice -
  Splice -> array.splice(where to add, how many to remove, what all to add) : adds or remove and



  Slice -> array.slice(start,end) ->copy certains parts of an array into a newly created array.
  from (start to end) excluding end.
- Array For Each :
  - easier way to write for loop
  - Does not have a return value

```
        array.forEach(value, index) => {
        console.log(value,index)
    }
```



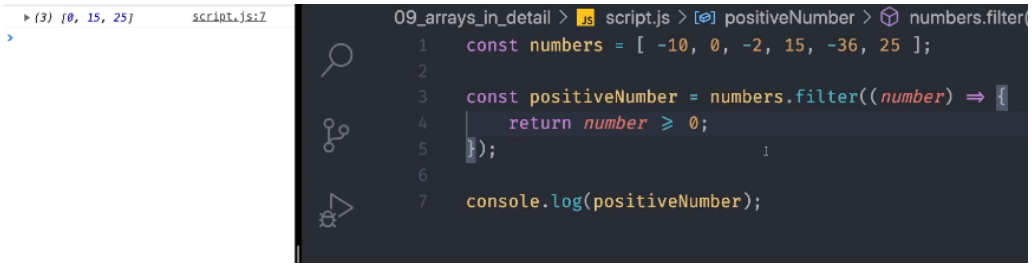


- Array map method :

  - It allocates memory to store and return value (opposite to the for each method)

  - Gives a new array all together

- Array Filter method :
  - It filters certain elements from an array.
  - for ex- if you want only positive values
  - Doesnt change the old array
  - Returns a new array



- Real life ex -

```
// A start up wants to reward the employees
// with 7 or more hours of overtime

const employeesData = [
    { name: 'Sebastian', overtime: 5 },
    { name: 'Cardi Vee', overtime: 10 },
    { name: 'George Lopez', overtime: 12 }
];

const employeesToReward = employeesData.filter((employee) => employee.overtime >= 7);

console.log(employeesToReward);
```

- Array Find - returns the first value that satisfy the condition.

```
const numbers = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ];

const value = numbers.find((number) => number > 5);

console.log(value);
```

```
const states = [ 'Alaska', 'California', 'Colorado', 'Hawaii' ];

const state = states.find((state) => state.startsWith('C'));

console.log(state);
```

- Array includes - checks if array includes something or not -> returns a Boolean value.
  - It is case sensitive

```
const bookshelf = ["Moby Dick", "The Great Gatsby", "Pride and Prej

if(bookshelf.includes("Moby Dick")) {
    console.log('We have that book, here you go.');
} else {
    console.log('Cannot find the book, sorry!');
}
```

- Array Sort - sorts the array
  - Modifies the same array
  - array.sort()
  - Ascending/ decending value

```
script.js:9
▶ (10) [1, 2, 3, 6, 7, 9, 15, 25, 44, 99]
```

```
// Array Sort
// => alphabetically
// => sorts numbers in ascending order

const numbers = [ 6, 3, 1, 7, 9, 2, 15, 25, 99, 44 ];

numbers.sort((a, b) => a - b);

console.log(numbers);
```

```
// Ascending Order
// numbers.sort((a, b) => a - b);

// Descending Order
numbers.sort((a, b) => b - a);
```

- Array Some and Array Every
  - Array Some - are some elements (checks a condition) and returns true or false.
  - ```
    const array = [1, 2, 3, 4, 5]

    // Array Some
    console.log(array.some((el) => el > 3)); // true
    ```
  - Array Every - Check if every element is passing some condition
  - ```
    // Array Some => return true if at least one element passes the test
    console.log(array.some((el) => el > 3)); // true

    // Array Every => returns true only if all elements pass the test
    console.log(array.every((el) => el > 0)); // false
    ```
  - Array Reduce - iterates over all the values and computes them to a single value
    - No need of an external var is there
    - Same example with forEach and reduce method
    - For each :
      ```
      const groceryList = [ 29, 12, 45, 35, 87, 110 ];

      let total = 0;

      groceryList.forEach((price) => {
          total += price;
      });

      console.log(total);
      ```
    - Reduce :
    - Takes in 2 arguments and a callback-function value called accumulator
    - 0 is the initial value
      ```
      const groceryList = [ 29, 12, 45, 35, 87, 110 ];

      const total = groceryList.reduce((accumulator, currentValue) => {
          return accumulator + currentValue;
      }, 0);

      console.log(total);
      ```

```
const numbers = [ 1, 2, 3, 4, 5 ];

const sum = numbers.reduce((acc, val) => {
    return acc + val;
}, 0);

console.log(sum);

// acc = 0, val = 1 => 0 + 1 == 1; <= acc
// acc = 1, val = 2 => 1 + 2 == 3; <= acc
// acc = 3, val = 3 => 3 + 3 == 6; <= acc
// acc = 6, val = 4 => 6 + 4 == 10; <= acc
// acc = 10, val = 5 => 10 + 5 == 15
```

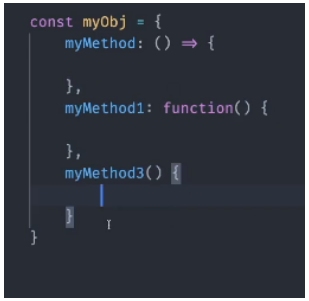
Objects

- Objects are unordered collection of related data
- In form of key-value pairs

```
const person = {
firstName:'john'
lastName: 'Doe'
age: '40'
car = {
        brand: 'maruti'
        year: 2016
        color: 'red'
}
}
```

- Object Methods

- another property of an obj that is a function



- Accessing - Adding and Updating object properties
  - Access - Use "DOT" notation
  - Add - use dot notation and { brackets } with key value pairs

- Access using [Sqaure Brackets] - can access properties dynamically.

```
// DOT NOTATION
const person = {
    firstName: 'Brad',
    age: 25,
}

const property= 'age';

console.log(person.property);

// SQUARE BRACKET NOTATION

console.log(person[property]);
```

  - Can access properties with dashes/ spaces using a string.

```
// DOT NOTATION
const person = {
    firstName: 'Brad',
    age: 25,
    "this is a key with spaces": true,
}
console.log(person.age);

// SQUARE BRACKET NOTATION
console.log(person['this is a key with spaces']);
```

- Built In Methods
  - object.keys() -> creates an array containing the keys of an object.

```
script.js:13
(4) ["boss", "secretary", "sales", "accountant"]
  0: "boss"
  1: "secretary"
  2: "sales"
  3: "accountant"
  length: 4
  __proto__: Array(0)
>
```

```
// Initialize an object
const employees = {
    boss: 'Michael',
    secretary: 'Pam',
    sales: 'Jim',
    accountant: 'Oscar'
};

const positions = Object.keys(employees);

console.log(positions);
```

- Object.values() : values of the keys

```
script.js:13
(4) [1, "26-July-2018", "mobile", "Chrome"]
  0: 1
  1: "26-July-20…
  2: "mobile"
  3: "Chrome"
  length: 4
  __proto__: Arr…
```

```
// Initialize an object
const session = {
    id: 1,
    time: `26-July-2018`,
    device: 'mobile',
    browser: 'Chrome'
};

const sessionInfo = Object.values(session);

console.log(sessionInfo);
```

- Object.entries() : creates a nested array of key/value pairs of an object.

```
top | 1 hidden
script.js:12
(3) [Array(2), Array(2), Array(2)]
```

```
10_objects_in_details > script.js > ...
// Object.entries() creates a nested array of the key/value pairs of

// Initialize an object
const operatingSystem = {
    name: 'Ubuntu',
    version: 18.04,
    license: 'Open Source'
};

const entries = Object.entries(operatingSystem);

console.log(entries);
```

- To display them all -

```
// Loop through the results
entries.forEach(entry => {
    let key = entry[0];
    let value = entry[1];

    console.log(`${key}: ${value}`);
});
```

- Object.freeze() : prevents modification to properties and values of an object.

```
// Initialize an object
const user = {
    username: 'John',
    password: '123123'
};

const admin = Object.freeze(user);

admin.username = 'test'
admin.password = '123123123'

console.log(admin);
```

Values Vs Reference - Deep and Shallow Clone

- The concept comes in play when we are copying values
- For numbers

```
2                script.js:11
1                script.js:12
>
```

```
// JavaScript differentiates Data Types on:

// - Primitive values (Number, String, Boolean ...)
// - Complex values (Objects, Arrays)

let x = 1; // 2
let y = x; // 1

x = 2;

console.log(x);
console.log(y);
```

```
                                 script.js:11
▶ (3) ["dogs", "cats", "llamas"]
                                 script.js:12
▶ (3) ["dogs", "cats", "llamas"]
```

```
// - Primitive values (Number, String, Boolean ... )
// - Complex values (Objects, Arrays)

const animals = [ 'dogs', 'cats' ]; // [ 'dogs', 'cats' ]
const otherAnimals = animals; // [ 'dogs', 'cats' ]

animals.push('llamas');

console.log(animals); // [ 'dogs', 'cats', 'llamas' ]
console.log(otherAnimals); // [ 'dogs', 'cats' ]
```

- Complex values -
  - Objects -

```
{firstName: "Johnny", lastName: "Snow"}
                                 script.js:20
{firstName: "Johnny", lastName: "Snow"}
```

```
// - Primitive values (Number, String, Boolean ... )
// - Complex values (Objects, Arrays)

// const animals = [ 'dogs', 'cats' ]; // [ 'dogs',
// const otherAnimals = animals; // [ 'dogs', 'cats'

// animals.push('llamas');

// console.log(animals); // [ 'dogs', 'cats', 'llama
// console.log(otherAnimals); // [ 'dogs', 'cats' ]

const person = { firstName: 'Jon', lastName: 'Snow'
const otherPerson = person;
(property) firstName: string
person.firstName = 'Johnny';

console.log(person);
console.log(otherPerson);
```

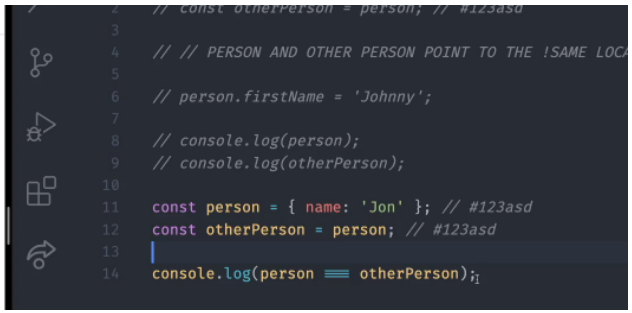
  - why is it behaving like this in complex data types?

- when a variable is assigned a primitive value it just copies that value. (number and string) (copy by value)
- When a variable is assigned a non-primitive value (array, objects, functions) -> it is given an address to the location in the memory of that variable. (copied by reference -> location to memory)

```
false     script.js:14
```

- Even if the values are equal they point to the different location in memory.

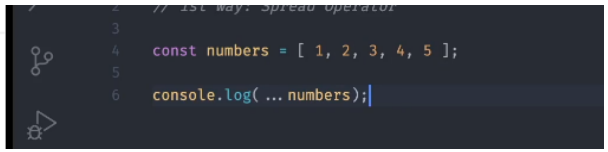
```
true      script.js:14
```



- Now they hold the same location in memory.

- Shallow Cloning :
  - Cloning arrays
    - Spread operator (...) -> get values individually

```
1 2 3 4 5 script.js:6
```



```
true        script.js:8
false       script.js:9
```

```
// 1st way: Spread operator

const numbers = [ 1, 2, 3, 4, 5 ]; // #123asd
const copiedNumbers = numbers; // #123asd
const newNumbers = [ ...numbers ];

console.log(numbers === copiedNumbers);
console.log(numbers === newNumbers);
```

- In the second image 2 copies are created, first is equal as they point to same location in memory.
- Newnumber array is not pointing to the same location, as they represent 2 different arrays.

```
script.js:10
▶ (6) [1, 2, 3, 4, 5, 6]
script.js:11
▶ (6) [1, 2, 3, 4, 5, 6]
script.js:12
▶ (5) [1, 2, 3, 4, 5]
```

```
const numbers = [ 1, 2, 3, 4, 5 ]; // #123asd
const copiedNumbers = numbers; // #123asd
const newNumbers = [ ...numbers ];

numbers.push(6);

console.log(numbers);
console.log(copiedNumbers);
console.log(newNumbers);
```

- The Newly created array using the spread operator remains unchanged, meaning it is a shallow clone.

- Array.slice ( ) -> also creates a shallow clone -> creates a completely diffrent array pointing to a different location in memory.

```
script.js:9
▶ (6) [1, 2, 3, 4, 5, 6]
script.js:10
▶ (6) [1, 2, 3, 4, 5, 6]
script.js:11
▶ (5) [1, 2, 3, 4, 5]
```

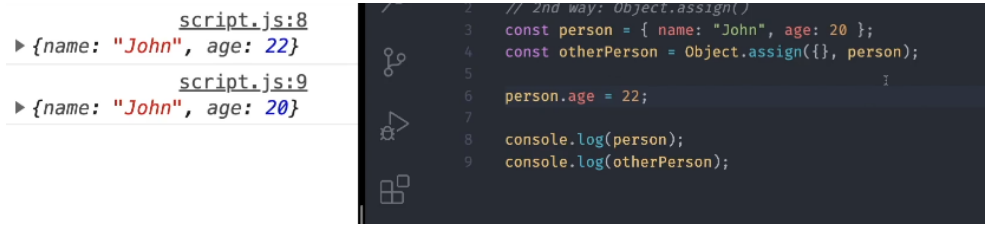
```
// 2nd way: Array.slice()
const numbers = [ 1, 2, 3, 4, 5 ]; // #123asd
const copiedNumbers = numbers; // #123asd
const newNumbers = numbers.slice(); // #321dsa

numbers.push(6);

console.log(numbers);
console.log(copiedNumbers);
console.log(newNumbers);
```

- Cloning Objects :
  - Spread operator (...)

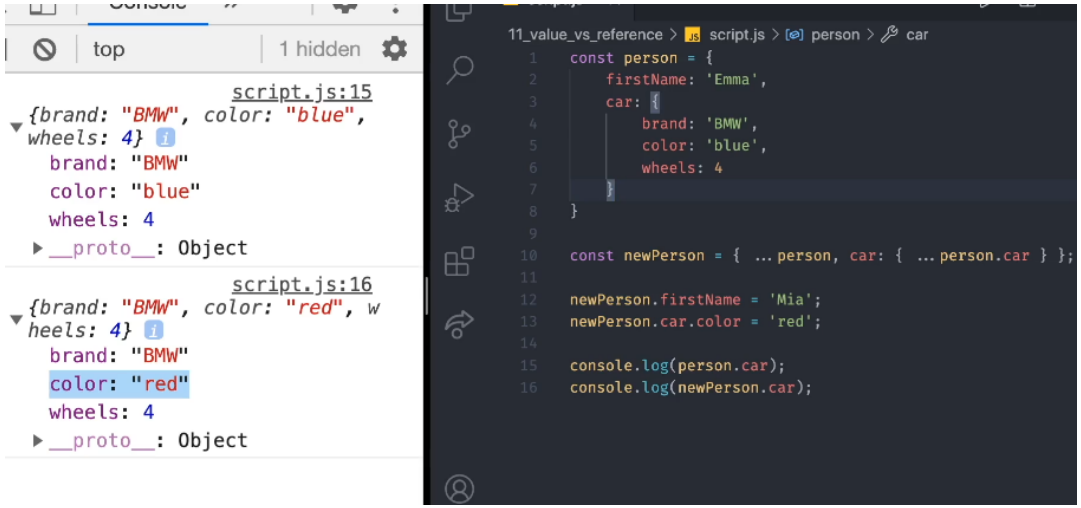
```
▶ {name: "John", age: 22}
script.js:9
▶ {name: "John", age: 20}
```

```
const person = { name: "John", age: 20 };
const otherPerson = { ...person };

person.age = 22;

console.log(person);
console.log(otherPerson);
```

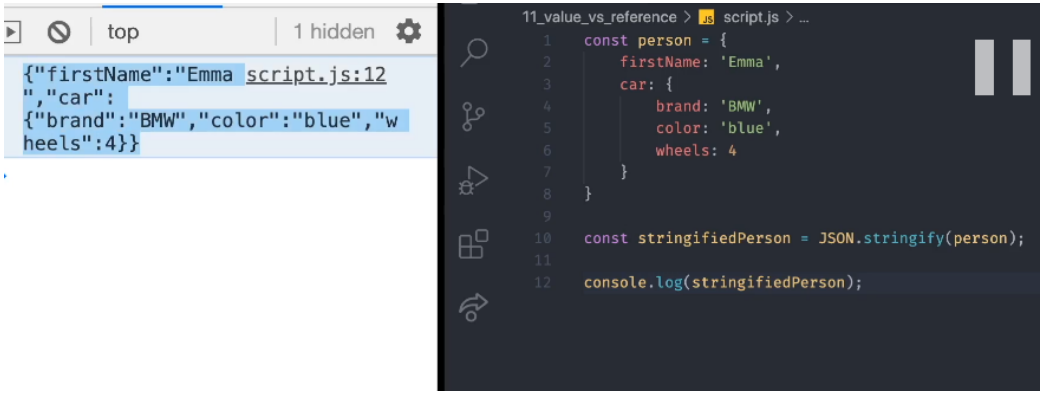
  - Object.assign({new object}, og object)



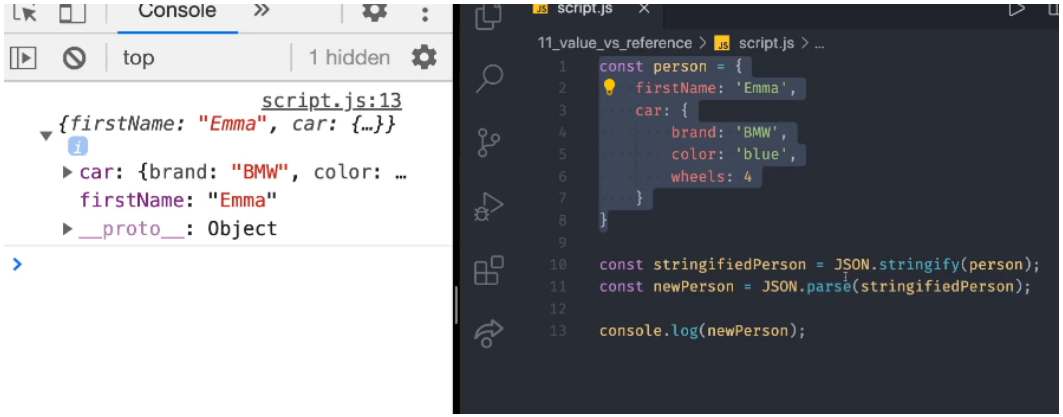
- Deep Cloning :
  - Issue with Shallow :



    - If the car color, which is an object inside an object has to be changed, the spread operator will just keep on increasing, as we have to mention the car obj as well in the spread operator for it to work, otherwise the car color will not change as the location we passed is only for the outer parent obj.
  - Solution : This is for deeply nested obj we need a deep clone, for an obj to be a deep clone it needs to destroy all the references.
    - JSON.stringify( ) : Converts all the JS obj into string, thereby all the references are destroyed.



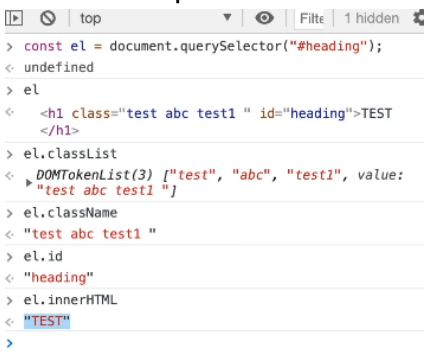
- JSON.parse( ) : to turn it back into a an obj we use parse()



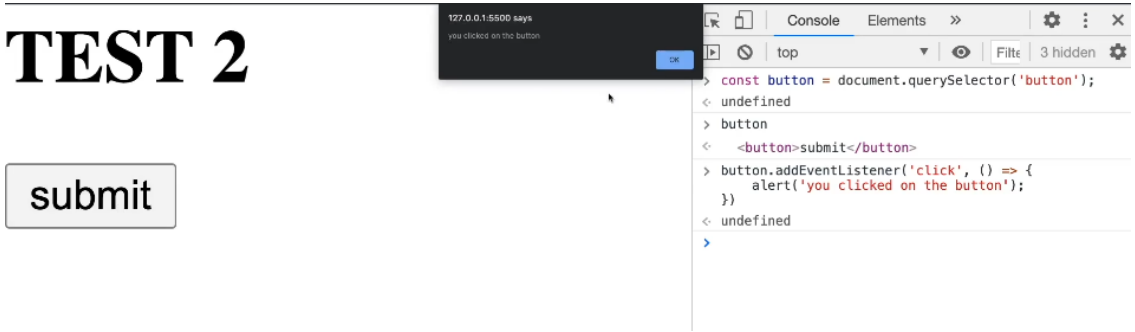
  - It is an obj but an obj that is the deep clone of the persons obj.
- Shortcut
  - `const newPerson = JSON.parse(JSON.stringify(person));`
- Result : deep cloning

DOM : Document object model

- Accessing the elements in our html, via Javascript using certain methods :
  - document.getElementById() - get the elements by the id name
  - document.getElementsByTagName('h1') - returns all the elements by that tag
  - document.getElementsByClassName('h1') - same but with classname
  - document.querySelectorAll('h2.classname') : returns all the h2 with classname "what ever the classname is"
    - can use # to target ids, . to target classnames or just Tagname to target any tagname.
- Element Properties -



- Methods :
  - addEventListener



- getBoundingClientRect( )

- hasAttribute( )

Classes in DOM :

```
![[Pasted image 20231202101009.png]]
```

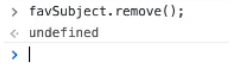
- Creating Nodes
  - document.createElement("h1")
  - appendChild( ) : to add it to the dom

- InnerText : to add the text to that created element
- InnerHtml : To add html to that element

- Traversing :

  - Maths
  - Science
  - English

- Removing :



New Keyword

- Functionality : Creates a new empty object
- diffrences with normal way of creating it

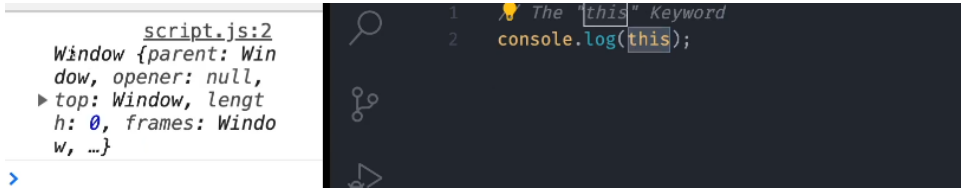
- For just normal creation it is not used that much.

- Uses?
  - Can use a lot of properties of present objs as everything in JS is an Obj.
    - ```
      // It creates a new object.
      const myNumber = new Number(100.234);

      console.log(myNumber.toFixed(0));
      ```
  - Dates
    - ```
      2025                    script.js:6
      ```
      
      ```
      // It creates a new object.
      const myDate = new Date('August 11, 2025');

      console.log(myDate.getFullYear());
      ```
    - This date methods are extended of a different object.

This Keyword

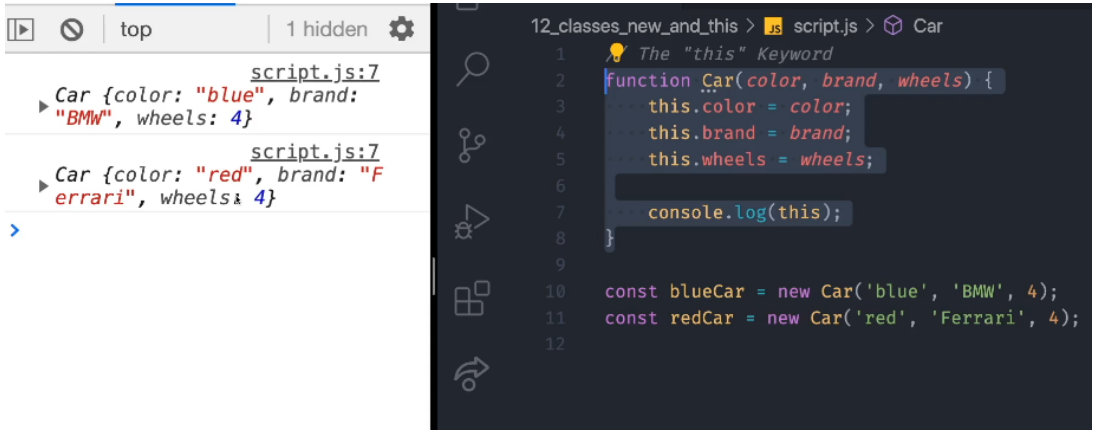
- Used to reference the obj that is executing the current function.
- Every function has a reference to it, in it's current execution context.
- Cannot use in arrow function
  - 

- This keywords points to a specific object
  - Without any function : references to the window object



- Inside a function : Refers to the Object 'this ' keyword in

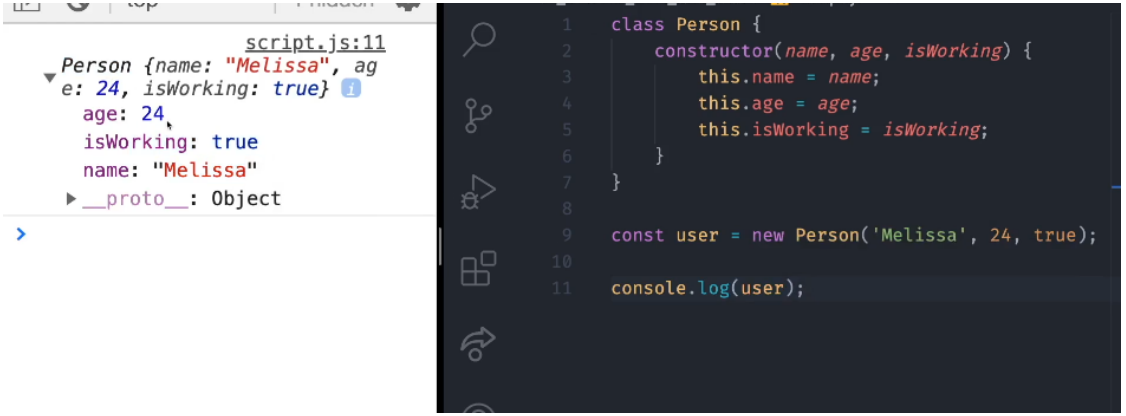
- Example :



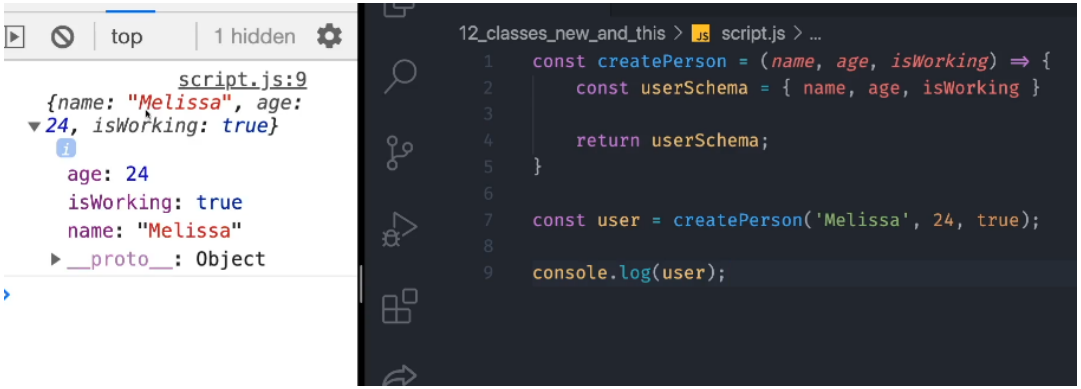
Classes

- It is SCHEMA for an obj that can save many values
- Constructor is just like parameters in a function (keys are passed inside)

- Initiating a user for that class

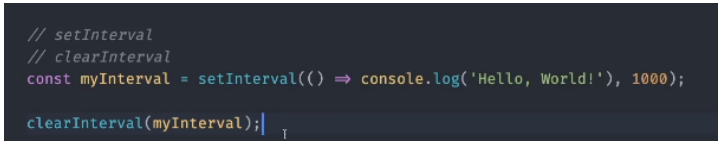


- Using Normal Functions



Intervals and Trimers

- setInterval() -> 1000 is 1 second (counted in miliseconds)
- clearInterval() -> clears the interval once it is done

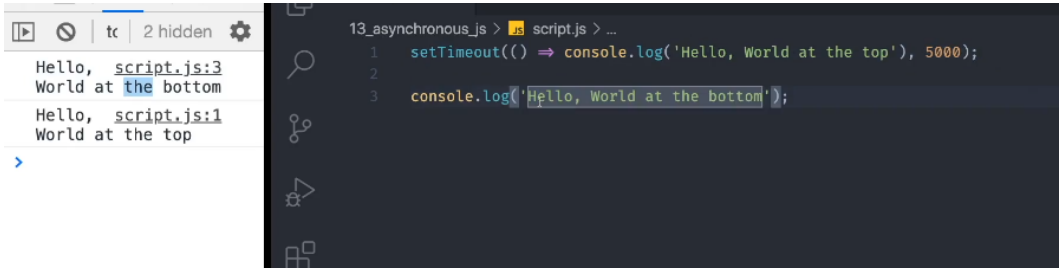


- setTimeout : -> wait certain amount of time before executing a chunk of code

- clearTimeout : ->

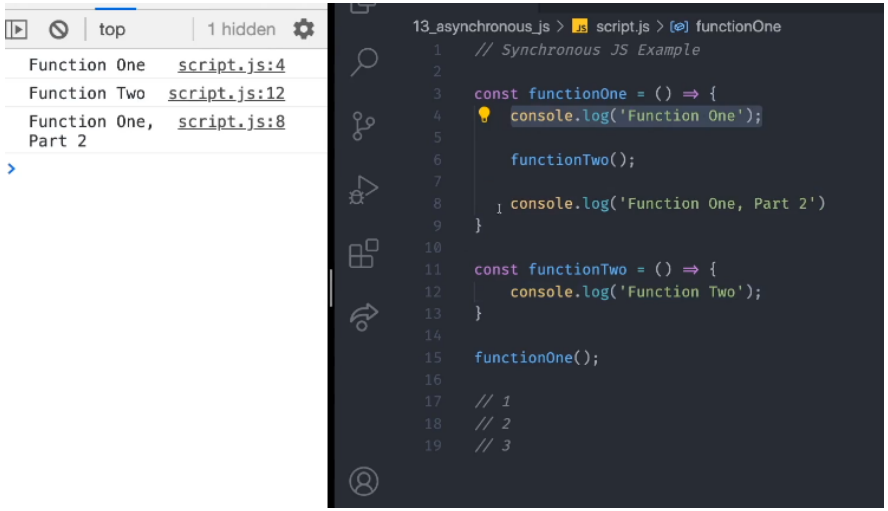
```
// setTimeout
// clearTimeout
const myInterval = setTimeout(() => console.log('Hello, World!'), 5000);

clearTimeout(myInterval);
```

- Asynchronous Nature :

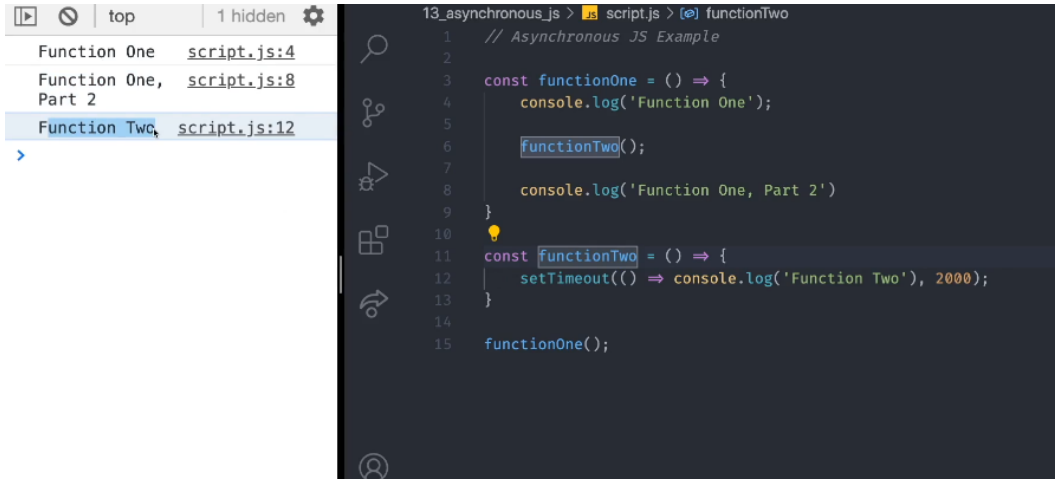


Asynchronous JavaScript

- Synchronous : code is executed Line by Line and tasks are completed instantly.
  - There is no Time delay in the completion of the tasks for those lines of code.
  - Example :



- Asynchronous :
  - Using the same example showing Async code:



- Here the code takes some time to run. These task are run in the background while tge js engine keeps executing other lines of code. When the result of the waiting gets available, it is then used in the program.
- This Happens because of something known as Event Loop.

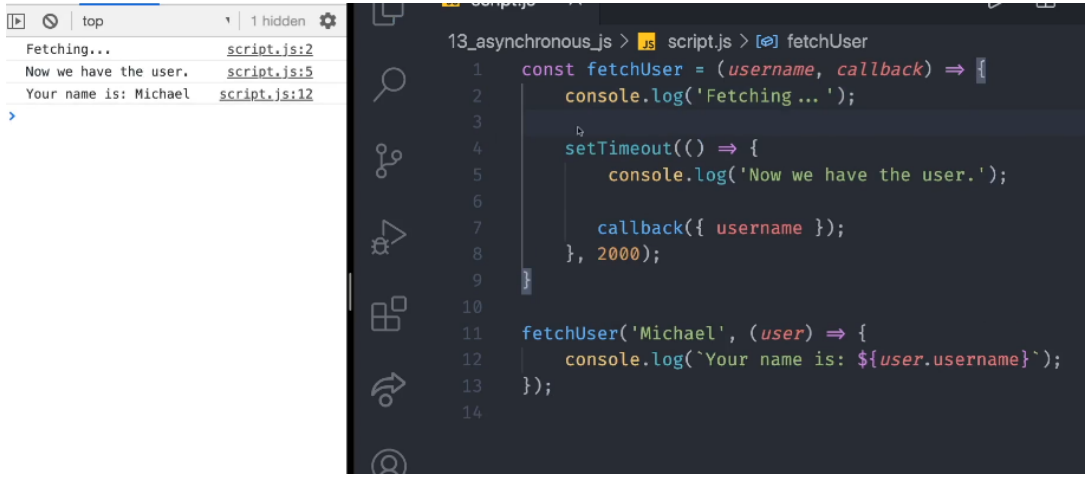
Callbacks

- Data Fetching from API : Async : cannot be sure how long it will take
  - Sync :

```
const fetchUser = (username) => {
    return { user: username };
}
```

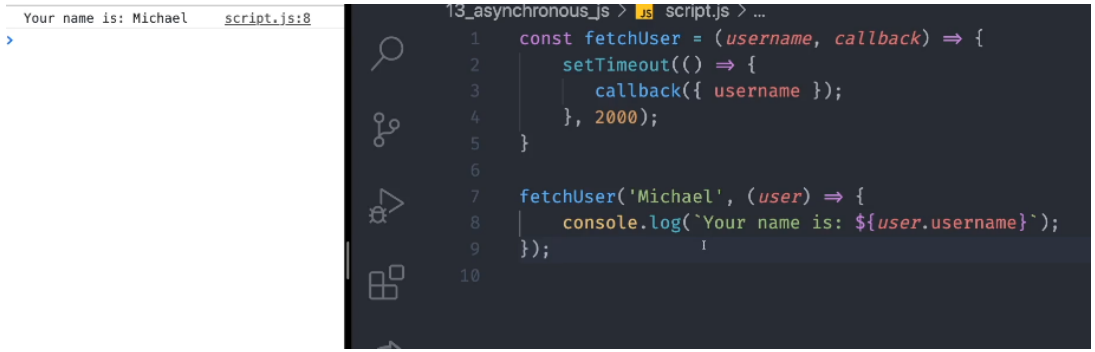
  - Async :

- The error is there because the data was not returned from the function immediately, but after 2 secs.
- To make it work we use Callback ( ) functions.
- pass in a callback function which will run when the data is fetched
- After 2 secs we get

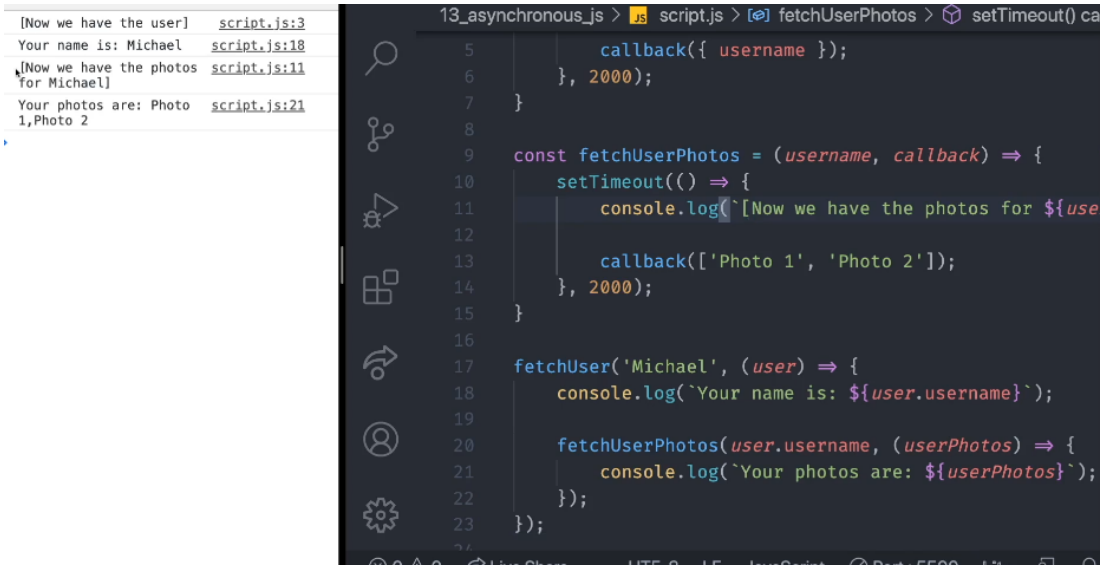


Callback HELL

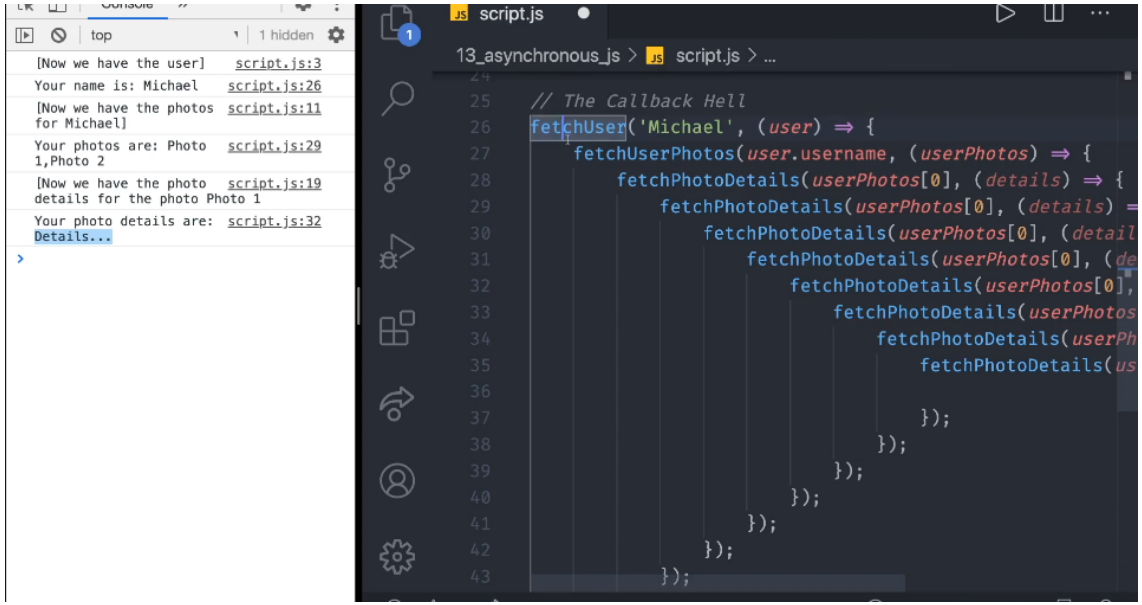
- A normal callback functionality looks like :



  - We will replace callbacks with promises and async await due to the callback hell.
- A complicated functionality of callbacks - ex of social media app



- Now as the functionality grows it becomes a lot of code to do the same functionality that is called a CALLBACK HELL!!!



  - This is not maintainable and also not DRY (do not repeat yourself)
  - To solve this issue Promises were introduced.

Promises

- They are objects that either return the successfully fetched data, or the error.
- It has two parameters resolve and reject
- Then keyword is used to invoke it, catch ( ) to get the error
  - Using the callback hell example

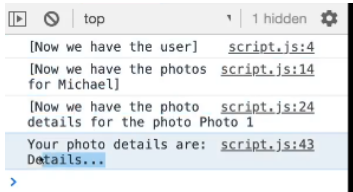
```
// Promises
const fetchUser = new Promise((resolve, reject) => {
    setTimeout(() => {
        console.log('[Now we have the user]');

        resolve({ username: 'Michael' });
    }, 2000);
});

fetchUser.then((user) => console.log(user.username));
```

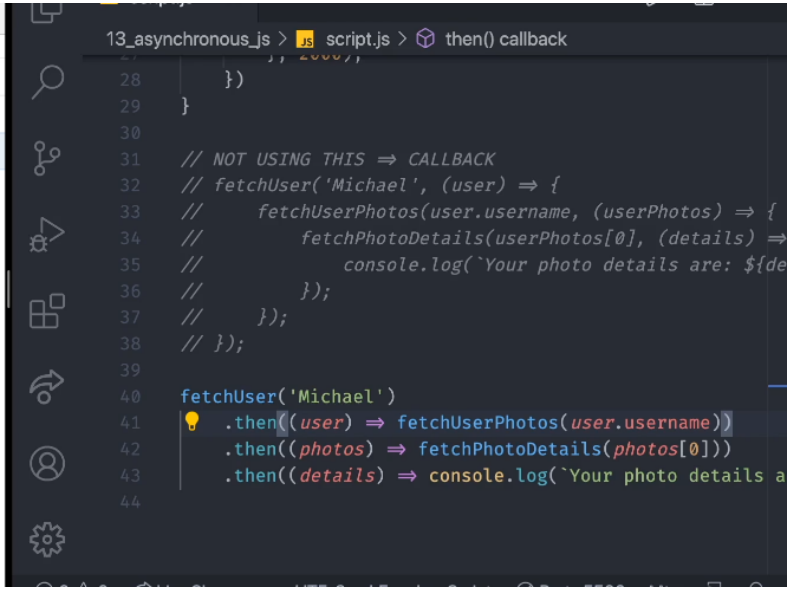
```
        console.log('[Now we have the user]');

        reject('User not found');
        // resolve({ username: 'Michael' });
    }, 2000);
});

fetchUser
    .then((user) => console.log(user.username))
    .catch((error) => console.log(error));
```

- Multiple Promises

```
        }, 2000);
    });
}

const fetchUserPhotos = (username) => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log(`[Now we have the photos for ${

            resolve(['Photo 1', 'Photo 2']);
        }, 2000);
    });
}

const fetchPhotoDetails = (photo) => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log(`[Now we have the photo details
```

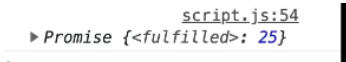
- The calling will be changed as :

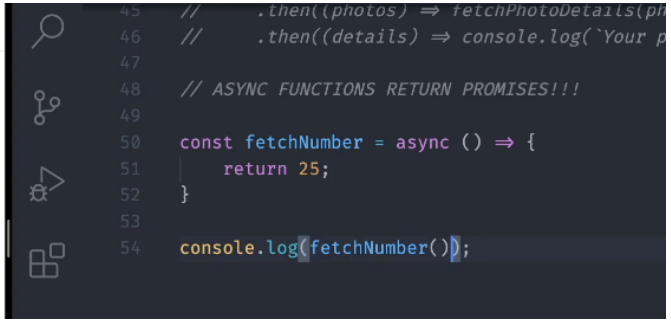




Async () Await ()

- It is an easier and a cleaner way to work with promises
- They look like sync functions so they are easier to write
- Async functions returns promises





```
25          script.js:55
>
```

```
46  //      .then((details) => console.log(`Your photo
47
48  // ASYNC FUNCTIONS RETURN PROMISES!!!
49
50  const fetchNumber = async () => {
51      return 25;
52  }
53
54  fetchNumber()
55      .then((number) => console.log(number));
```

- Await Keyword : Waits for the promises to return a result.
  - If you want you use await it needs to be inside an async function
  - This is good as we do not have to use then() or fetch()

```
25          script.js:56
```

```
46  //      .then((details) => console.log(`Your photo de
47
48  // ASYNC FUNCTIONS RETURN PROMISES!!!
49
50  // ASYNC => AWAIT
51  const fetchNumber = async () => {
52      return 25;
53  }
54
55  const consoleFetchedNumber = async () => {
56      console.log(await fetchNumber());
57  }
58
59  consoleFetchedNumber();
60
```

    - Using the callback hell example

Import and Export

- To export one thing from a file
  - `export default name`
- To Import that thing from that file
  - `import name from 'path'`
- Add type = module in html src
  - 

- Example
  - 

- To export and import multiple things out of one file
- Add export keyword before all the things

- In Import use { } and add names of things you want to import
  - ```
    export const dogs = [ 'Bear', 'Fluffy', 'Doggo' ];
    export const numberOfDogs = dogs.length;
    ```
  - ```
    // IMPORTS AND EXPORTS
    import { dogs, numberOfDogs } from './dogs.js';
    import onlyOneThing from './test.js';

    console.log(onlyOneThing);

    console.log(`I have ${numberOfDogs} dogs`);
    console.log(`My dogs are: ${dogs}`);
    ```
- In import you can change the name of the import as well but it is not good practice

  -
  ```
  import { dogs, numberOfDogs } from './dogs.js';
  import abc from './test.js';

  console.log(abc);
  ```

Extras

- Object and array Destructuring
  - Before : An object with Multiple values has been created, this is DRY
    - ```
      }
      // DRY
      console.log(person.firstName);
      console.log(person.lastName);
      console.log(person.car.color);
      console.log(person.car.wheels);
      console.log(person.car.wheels);
      console.log(person.animals.dog.name);
      console.log(person.animals.dog.age);
      console.log(person.animals.cat.name);
      console.log(person.animals.cat.age);
      ```
  - After : Using Destructuring
    - ```
      const { firstName, lastName, car, animals } = person;

      console.log(firstName);
      console.log(lastName);
      console.log(car.color);
      console.log(car.wheels);
      console.log(car.wheels);
      console.log(animals.dog.name);
      console.log(animals.dog.age);
      console.log(animals.cat.name);
      console.log(animals.cat.age);
      ```

- For internal properties
  - ```
    const { firstName, lastName, car: { color, wheels }, animals } = perso

    console.log(
    console.log(lastName);
    console.log(color);
    console.log(wheels);
    console.log(animals.dog.name);
    console.log(animals.dog.age);
    console.log(animals.cat.name);
    console.log(animals.cat.age);
    ```
    const lastName: string
  - ```
    const { animals: { dog, cat } } = person;

    console.log(dog.name);
    console.log(dog.age);
    console.log(cat.name);
    console.log(cat.age);
    ```
- Real Usecases :
  - ```
    // Node
    const test = (req, res) => {
        const { body, params } = req;
    }

    // React
    const Component = ({ keys, }) => {

    }
    ```
- Array
- Before
  - ```
    const introduction = ["Hello", "I", "am", "Sarah"];
    const greeting = introduction[0];
    const name = introduction[3];

    console.log(greeting, name);
    ```
- After

```
Hello Sarah   script.js:10
>
```

```
// Object Destructuring
// Array Destructuring

const introduction = ["Hello", "I", "am", "Sarah"];
// const greeting = introduction[0];
// const name = introduction[3];

const [greeting,,,name] = introduction;

console.log(greeting, name);
```

- First Class Functions
  - A concept which tells that you can use functions as values

```
function abcd(a){
    a();
}

abcd(function(){console.log("hey");})
```

    - This prints hey in the console
- Arrays Under the hood
  - Arrays are also objects, stored in a key value pair format

```
var arr = [1,2,3,4];

arr = {
    0: 1,
    1: 2,
    2: 3,
    3: 4
}
```

    - You can add -ve index in array as well, it will acted as a key
- Higher Order Functions
  - Accepting a function in parameters or returns a functions, forEach is a higher order function.

```
function abcd(val){

}

abcd(function(){})
```

```
function abcd(){
    return function(){}
}

abcd()
```

- Constructor Functions
  - Normal function where this is used and new keyword is used when it is called

```
function saanchaOfBiscuit(){
    this.width = 12;
    this.height = 22;
    this.color = "brown";
    this.taste = "sugary";
}

var bis1 = new saanchaOfBiscuit();
var bis2 = new saanchaOfBiscuit();
var bis3 = new saanchaOfBiscuit();
```

  - Used when elements to be used have similar properties.

```
function circularButtonBanao(color){
    this.radius = 2;
    this.color = color;
    this.icon = false;
    this.pressable = true;
}

var redbtn = new circularButtonBanao("red");
var greenbtn = new circularButtonBanao("green");
```

- IIFE : Immediately Invoked Function Expression
  - Used to make Private Variables

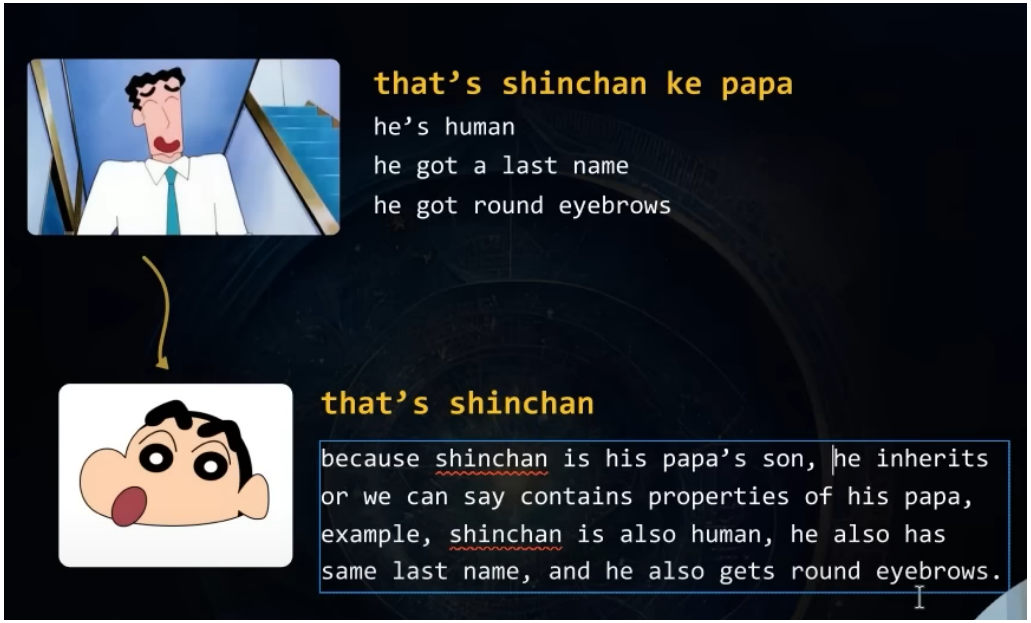
```
(function(){
    var a = 12;
})()
```

    - a here is a private variable, not available in DOM
    - How to access

```
var ans = (function(){
    var privateval = 12;

    return {
        getter: function(){
            console.log(privateval);
        },
        setter: function(val){
            privateval = val;
        }
    }
})()
```

```
> ans.setter(24)
< undefined
> ans.getter()
  24
< undefined
>
```

- Prototype :
  - Create an obj, followed by a dot operator
  - Every obj created get prototype property
  - It contains many helper properties to complete our tasks
  - for ex - .length property
- Prototype Inheritance :

- 

- This is called Inheritance : properties of parents getting passed to children, this is also the case in JS.
- This is what is called prototypal inheritance in JS
- Base properties can be inherited and extra properties can be added.
  - `.__proto__` is the way to inherit it

```
var human = {
    canFly: false,
    canTalk: true,
    canWalk: true,
    haveEmotions: true,
    hasFourLegs: false
}

var sheryiansStudent = {
    canMakeAmazingWebsite: true,
    canMakeAwesomeAnimations: true,
    canMakeWorlClassAwwwardedWebsites: true
}

sheryiansStudent.__proto__ = human;
```

- call apply bind
  - If you want to change where this points inside a function containing an object

- CALL

```
function abcd(){
    console.log(this.age);
}

var obj = {age: 24}

abcd.call(obj)
```

```
function abcd(val, val2, val3){
    console.log(this);
}

var obj = {age: 24}

abcd.call(obj,1,2,3)
```

- Apply : Array is passed instead of the direct parameter values
  - Usecase : To change the value to this

```
function abcd(val, val2, val3){
    console.log(this,val, val2, val3);
}

var obj = {age: 24}

abcd.apply(obj, [1,2,3])
```

- Bind : Bind will not run, but will give a function, can be stored in a var

```
function abcd(){
    console.log(this);
}

var obj = {age: 24}

var bindedfnc = abcd.bind(obj);
bindedfnc();
```

- Pure and Impure Functions :
  - Pure :
    - It always returns same output for same input.
    - It will never change/update the value of a global variable.

```
function abcd(a,b){
    return a*b;
}

var ans1 = abcd(1,2);
var ans2 = abcd(1,2);
```

- Impure is just the opposite of pure

```
var abcdef = 12;

function abcd(a,b){
    abcdef = 24;
    return a*b;
}

var ans1 = abcd(1,2);
var ans2 = abcd(1,2);
```

```
function abcd(val){
    return Math.random()*val;
}

var ans1 = abcd(2);
var ans2 = abcd(2);
```

- Concurrency and Parallelism :
  - Concurrency : When sync code is running on Main stack and the async code on the side stack together.
  - Parallelism : This Focuses on making the task run on different processors and on their cores.
- Throttling : Controlling the number of executions of a code : ex - Scrolling