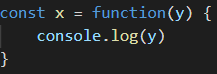
**Functional Programming**

**Functions as first class citizens** – It is a weird way of saying functions are treated as any other object. It can also be called **functions as first class objects**.

In functional programming languages, functions can be stored in variables (function expression in JavaScript), they can be passed into other functions as parameters, and they can return other functions as a result. Also we can assign a property for function!





**Hoisting**

Hoisting is JavaScript's default behavior of moving all declarations to the top of the current scope (to the top of the current script or the current function).

'use strict';

window.onload = () => {

    console.log(x);

    let x;

}

It returns:

Uncaught ReferenceError: Cannot access 'x' before initialization

So let and const keywords prevent us from using the variable before initialization.

If it is var:

'use strict';

window.onload = () => {

    console.log(x);

    var x;

}

Then it returns undefined. Cuz it is declared but not defined.

If it is a function:

'use strict';

window.onload = () => {

    console.log(x());

    var x=()=>{

        // function body

    };

}

Then it returns: Uncaught TypeError: x is not a function. Because it doesn’t know that x is a function. After Hoisting our code looks like this=>

'use strict';

window.onload = () => {

var x

    console.log(x());

    x=()=>{

        // function body

    };

}

“use strict” just makes sure that we declare our variable.

'use strict';

window.onload = () => {

    x = 3;

    console.log(x);

}

Here it gives an error that x is not defined if it weren’t for “use strict”

Then this would return 3

**Null vs Undefined**

In JavaScript, undefined means a variable has been declared but has not yet been assigned a value, such as:

 var testVar;

    alert(testVar); //shows undefined

    alert(typeof testVar); //shows undefined

null is an assignable value. It can be assigned to a variable as a representation of no value:

var testVar = null;

    alert(testVar); //shows null

    alert(typeof testVar); //shows object

From the preceding examples, it is clear that undefined and null are two distinct types: undefined is a type itself (undefined) while null is an object.

null === undefined // false

null == undefined //true

null === null // true

We have got a bookshelf with no books🡪if we take a book it is undefined

We haven’t got bookshelf 🡪if we take a book it is null

**IIFE (Immediately Invoked Function Expression)**

An **IIFE** (Immediately Invoked Function Expression) is a [JavaScript](https://developer.mozilla.org/en-US/docs/Glossary/JavaScript) [function](https://developer.mozilla.org/en-US/docs/Glossary/Function) that runs as soon as it is defined.

window.onload = () => {

    (( ) => {

        console.log("IIFE-1");

    })( );

    (function( ) {

        console.log("IIFE-2");

    })( );

We just create a function inside parenthesis and we also put parenthesis after the first parenthesis so that it is called once the page is loaded.

**Inheritance in ECMAScript 5 (ES5) call( ) method**

 const Animal = function(fname, age) {

        this.fname = fname;

        this.age = age;

        this.GetName = () => {

            console.log(this.fname);

        }

    }

    const Fox = function(fname, age) {

        Animal.call(this, fname, age);

    }

    const fox1 = new Fox("fox-1", 12)

    fox1.GetName();

we use call method to call animal constructor function in Fox. And We gotta pass “this” as well, the fox itself. Call is like a bind( ) method. The diffrence is that it calls that class immediately.

**How to equalize constructor functions(classes)**

In order for them to be equal we gotta equalize A’s prototype (base function’s) to B’s prototype!

 const Animal = function(fname, age) {

        this.fname = fname;

        this.age = age;

        this.GetName = () => {

            console.log(this.fname);

        }

    }

    const Fox = function(fname, age) {

        Animal.call(this, fname, age);

    }

    const fox1 = new Fox("fox-1", 12)

    Animal.prototype = Fox.prototype

    console.log(fox1 instanceof Animal);

the result is true.

**Function to generate a random number between a specific range**

function randomNumber(min, max) {

return Math.floor(Math.random() \* (max - min) + min);

}

For example 1-9 Math.floor((0-8)+1)

**Object.freeze(obj)**

We use this method to prevent changing the properties of the object

 const Animal = function(fname, age) {

        this.fname = fname;

        this.age = age;

    }

    const ani = new Animal("Vahid", 12)

    Object.freeze(ani)

    ani.fname = "jello";

Cannot assign to read only property 'fname' of object '#<Animal>'

A frozen object can no longer be changed, freezing an object prevents new properties from being added to it, existing properties from being removed

**Closure**

Closure means that an inner function always has access to the vars and parameters of its outer function, even after the outer function has returned.

 function OuterFunction() {

        var outerVariable = 1;

        function InnerFunction() {

            alert(outerVariable);

        }

        InnerFunction();

    }

Here, it is also called closure. Because it remembers x array even after onload event’s callback is finished. It will remember its value.

let x;

window.onload = () => {

    x = [];

    get(x)

}

function get(x) {

}

**DataTransfer**

e.dataTransfer can only be used in drag events only when there is transaction in dragZone! Once we drop something, it is sent to e.dataTransfer.files whether it is one file or multiple files.

boxes.addEventListener("drop", (e) => {

        e.preventDefault();

        for (const file of e.dataTransfer.files) {

            fileList.push(file)

        }

        drop(e)

    });

function drop(e) {

    title.textContent = "Click or drag"

    boxes.classList.remove("border")

    for (const file of fileList) {

// one by one we are gonna read the files

        const reader = new FileReader();

        reader.readAsDataURL(file)

        reader.onload = function(e) {

            boxes.insertAdjacentHTML("beforeend", `<img src="${reader.result}" alt="img">`)

        }

    }

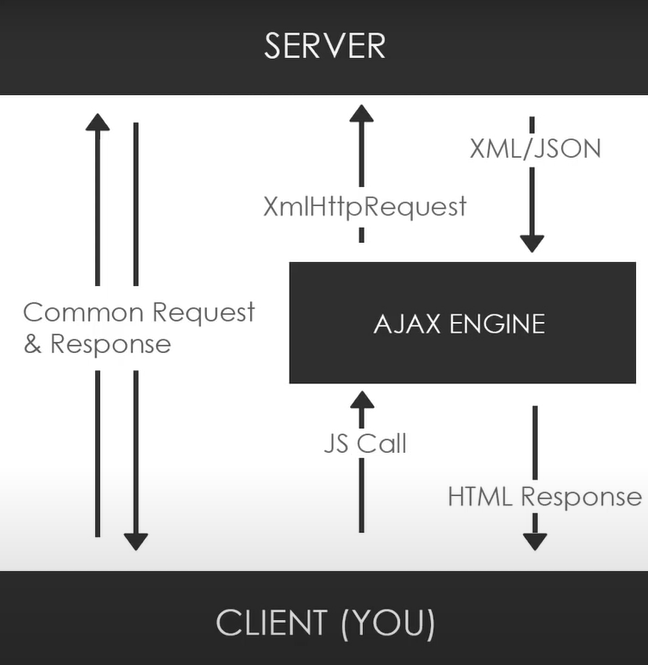
}

**Json files**

Json is a format by which we send data. For instance, in apis we use this format to send data. Another way to send data is by xml files. XML format is similar to html format. Since, it takes more space in memory json format is more preferred.

**AJAX**

AJAX stands for asynchronous javascript and xml. Now we can load in data without refreshing the entire page. We can make requests to the database without loading the page. JSON format has replaced XML part however it is still called asynchronous javascript and XML.



This is all done in Javascript file and we send XmlHttpRequest object to the server and after the request has been reached it can be responded with an Xml, Json or even a plain text.

function load() {

    let xhr = new XMLHttpRequest();//XHR object

    xhr.open("GET", "text.txt", true)

    xhr.onload = function() {

         if (this.status == 200) {

            console.log(this.responseText);

        }x

    }

    xhr.send()

}

xhr.open(method,url, asynchronous)

and we also need xhr.send( ) method to send the request only then it is going to work.

**AJAX with JQUERY**

window.onload = () => {

    $(document).on("click", "#load", () => {

        $(".test").load("text.txt", function(response, status, xhr) {

            console.log(status);

        });

    })

}

.test is a paragraph. The status will be success.

window.onload = () => {

    $(document).on("click", "#load", () => {

        $.ajax({

            method: "GET",

            url: "https://jsonplaceholder.typicode.com/posts",

            datatype: "JSON",

        }).done((response, status, xhr) => {

        })

    })

}

OR

window.onload = () => {

    $(document).on("click", "#load", () => {

        $.ajax({

            method: "GET",

            url: "https://jsonplaceholder.typicode.com/posts",

            datatype: "JSON",

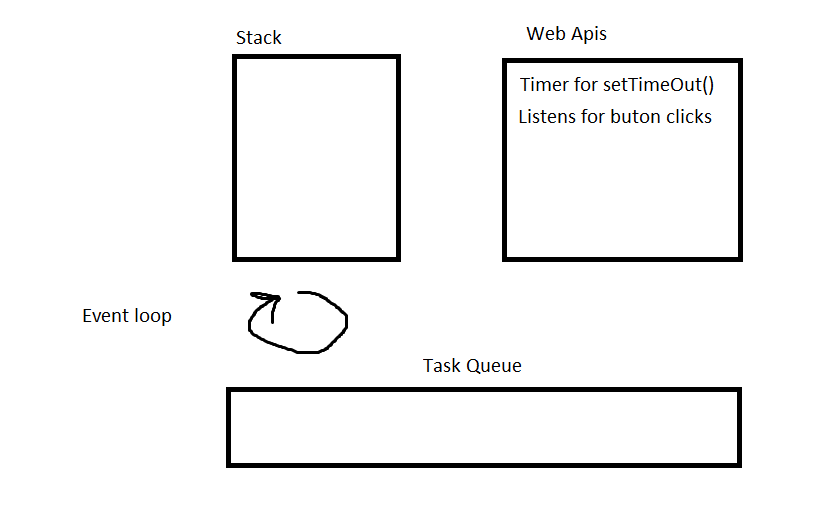
success: function(response){…..}

        });

    });

}

**Call Stack, WebApi, task Queue Event loop**

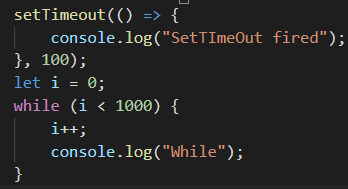


Task Queue is also called callback queue. There are 2 types tasks of Macro and Micro Tasks. Macro tasks are setTimeOut(), button clicks and etc. Micro tasks are promises, fetch and etc. Macro tasks wait for the stack to finish, and when the stack is empty they are added in order. However, if there is any micro task then their priority is higher so they are gonna be added first then micro tasks are gonna be added.

All microtasks are completed before any other event handling or rendering or any other macrotask takes place.

So by using setTimeOut() method, we ensure that minimum this much time will be waited but it doesn’t mean that it will certainly be executed after that time because the stack has to be emptied.

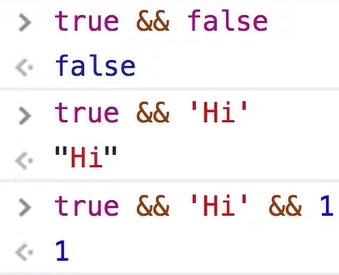
If we have setTimeOut() or any buttons that need listening then it is added to the webapi. WebApis will listen for those buttons or they will set a timer for setTimeOut() after one of them finishes it is put in the Task Queue after this once the stack is empty only then it is gonna be put in the stack. So for example 🡪



Even though the timer is set to 100 ms it will be added to the stack when the stack is empty and this will happen after while loop finishes.

**Truthy and Falsy**

0, empty strings, false values are deemed falsy values in javascript.



if the result is truthy then the last operand is always returned as a result.

**Arrow function vs Regular Functions**

In regular functions the “this” keyword represents the caller object, which could be the window, a button or whatever. With arrow functions the “this” keyword always represents the object that defined the arrow function. In an arrow function, the "this" keyword represents the object that owns the function, no matter who calls the function.

Arrow functions don't have their own this/super/arguments binding. They inherit them from their parent lexical scope

const toString = Object.prototype.toString;

const o = {

foo: () => console.log("window", toString.call(this)),

bar() {

const baz = () => console.log("o", toString.call(this));

baz();

}

}

o.foo() // logs window [object Window]

o.bar() // logs o [object Object]

When the o.foo arrow function is called there is no surrounding function from which baz could inherit its this. Consequently, it captures the this binding of the global scope which is bound to the Window object. When baz is invoked by o.bar, the arrow function is surrounded by o.bar (o.bar forms its parent lexical scope) and can inherit o.bar's this binding. o.bar was called on o and thus its this is bound to o.

**Import and Export**

In javascript there are 2 kinds of exports: default export and normal export. In 1 js file there can be 1 only 1 default export. We take default export from the name without destructuring 🡺

export default function () {

  console.log("hey");

}

export function LA() {

  console.log("La");

}

"use strict";

import Hey from "./js.js";

window.onload = () => {

  Hey();

};

We can also take LA method by destructuring 🡺

"use strict";

import { LA } from "./js.js";

window.onload = () => {

  LA();

};

Note that the script file has to be the type of module 🡺

 <script type="module" src="main.js" defer></script>

**Destructuring**

If we are destructuring an array then we must use square brackets 🡺

const [a,b]=arr;

However, if it is an object then we must use curly braces 🡺

const {a,b}=obj;