# Scope and Closure

* We have 3 types of variable in JavaScript ***var***, ***let*** and ***const***
* var is the old one, and should not be used now in any case. As it has many issues with creating scopes
  + why it is still there ?
* Also there are 4 kinds of scope in Javascript - Block Scope, Global Scope, Function Scope, Module Scope

## Block scope & Global Scope

The **scope** is the current context of execution in which values and expressions are "visible" [MDN](https://developer.mozilla.org/en-US/docs/Glossary/Scope)

**Global Scope** : Any variable/expression which is written outside - i.e. not inside any functions, blocks etc. This is shared across files.

### let

* this creates a block scope
* re-declaration in NOT allowed (in same scope)
* re-assignment is allowed

{ *// block scope*

let x = 0;

let y = 0;

console.log(x); *// 0*

let x = 1; *// Error*

}

{

let x = 1;

let y = 1;

x = 2;

console.log(x);*// 2*

}

console.log(x); *// Error in Global Scope*

**Temporal Dead Zone**(TDZ) : the area in which a variable is not accessible. Temporal because it depends on time of excution not position

{

*// TDZ starts*

const say = () => console.log(msg); *// hi*

let msg = 'hi';

say();

}

### const

* this creates a block scope
* re-declaration in NOT allowed
* re-assignment is NOT allowed
* must be assigned at declaration time.

{

const x; *//Error*

const y=0;

}

{

const x=1;

x=2 *// Error*

}

console.log(x); *// Error*

### Variable Shadowing

let x = 0 *// shadowed variable*

{

let x = 1;

console.log(x)

}

}

### var

* it doesn't have any block scope, and can be re-declared
* it only had function scope
* var are hoisted, so they can be used before the declaration

var x = 1;

var x = 2; *// valid*

console.log(y) *// valid*

var y = 3

z=4

console.log(z) *// valid*

var z;

**NOTE** : You should NOT use **var** now ❌

### let vs var

for(let i=0;i<5;i++){

setTimeout(

()=>console.log(i),

1000)

} *// prints 0,1,2,3,4*

for(var i=0;i<5;i++){

setTimeout(

()=>console.log(i),

1000)

} *// prints 5,5,5,5,5*

## Module scope

In modern javascript, a file can be considered as module, where we use export and import syntax to use variable across files. We

<script src="index.js" type="module"></script>

export { someVar, someFunc}

import { someVar} from './app.js'

### global Object

* The global Object is the variable **window** in case of browser. This helps you to use variables across the scopes. Also, it is the **this** value for global functions
  + window.alert
  + window.Promise
* In non-browser environment, **window** doesn't exist. but other global objects exist.
* ***var*** affects this global obejct, also ***function*** declarations.

function sayHi(){

console.log(this) *// this will refer to window*

}

*// Strict mode can change this behaviour;*

`use strict`

function sayHi(){

console.log(window) *// this is a better way of code*

}

## function scope

* it is created upon execution a function

function sayHi(name){

return name;

}

sayHi() *// this call will create a function scope*

sayHi() *// this call will create another function scope*

#### Lexical Environment

* Every variable in JavaScript (within global / block / or function) has a reference to an object-like data called Lexical enviroment. This object (kind of object) serves as the basis of search for value of variable.

let name = 'john'

console.log(name)

[outer]

null

LexicalEnviroment

name: 'john'

name

Lexical Enviroment (Global variable)

let name = 'john';

function sayHi(){

let greet = "hi"

console.log(greet)

}

sayHi()

console.log(name, sayHi)

[outer]

[outer]

null

LexicalEnviroment1

name: 'john',

sayHi: function

LexicalEnviroment2

greet: 'hi'

name

sayHi

greet

Lexical Enviroment (functions)

let name = 'john';

function sayHi(){

let greet = "hi"

console.log(name)

}

sayHi()

[outer]

[outer]

null

LexicalEnviroment1

name: 'john',

sayHi: function

LexicalEnviroment2

greet: 'hi'

name

Lexical Enviroment (functions)

## Hoisting

The movement of variable declaration to top of scope - before execution

* function declarations are properly hoisted (value accessible)
* var is hoisted.

let name = 'john';

sayHi() *// valid*

function sayHi(){

let greet = "hi"

console.log(name)

}

sayHello() *// error*

let sayHello = function(){

console.log(name)

}

**Temporal Dead Zone**(TDZ) :

let x = 1;

{

console.log(x) *// Reference error*

let x = 2;

}

## Closures

* we can create nested functions in JavaScript

function createUser(name){

let greeting = 'Hi '

function greet(){

return greeting + name + ' is Created';

}

return greet()

}

createUser('john') *// Hi john is created;*

* Now more useful work is if we can return the greet function itself.

function createUser(name){

let greeting = 'Hi '

function greet(){

return greeting + name + ' is Created';

}

return greet *// returned just definition of function*

}

let welcomeJohn = createUser('john')

welcomeJohn() *// // Hi john is created;*

* This is **Closure**
  + welcomeJohn function definition has access
    - to outer **params** ( name ) which came for createUser function
    - also any other "variables" declared inside createUser will also be accessible to this welcomeJohn

### Example

function initCounter() {

let count = 0;

return function () {

count++;

};

}

let counter = initCounter();

counter() *// 0*

counter() *// 1*

let counter1 = initCounter();

counter1() *// 0*

counter1() *// 1*

**NOTE** : so whenever you have a function which wants to preserve a value over many calls - it's a time for closure.

#### Lexical Environment

function init() {

let name = 'john';

function greet() {

console.log(name)

}

return greet;

}

let sayHi = init();

sayHi();

[outer]

[outer]

[outer]

null

LexicalEnviroment1

sayHi: ----

init: function

LexicalEnviroment2

name: 'john'

greet: function

LexicalEnviroment3

--empty--

init

name

sayHi

Lexical Enviroment (functions)

### Real life example 1

function initCounter(id) {

let count = 0;

return function () {

count++;

document.getElementById(id).innerText = count;

};

}

let count = 10;

let counter1 = initCounter('btnCount1');

let counter2 = initCounter('btnCount2');

*// here `btn1` and `btn2` are id of HTML buttons.*

<button onclick="counter1()">1</button>

<p id="btnCount1"></p>

<button onclick="counter2()">2</button>

<p id="btnCount2"></p>

### Real life example 2

function initAddString(inputId, outputId) {

let str = '';

return function () {

str += ' ' + document.getElementById(inputId).value;

document.getElementById(inputId).value = '';

document.getElementById(outputId).innerText = str;

};

}

let strAdder1 = initAddString('text1', 'text-output1');

let strAdder2 = initAddString('text2', 'text-output2');

<input type="text" id="text1">

<button onclick="strAdder1()">Add String</button>

<p id="text-output1"></p>

<input type="text" id="text2">

<button onclick="strAdder2()">Add String</button>

<p id="text-output2"></p>

## IIFE - Immediately Invoked Function Expression

* this practice was popular due to var.
* Immediately invoking a function avoids - re-declaration of variables inside it

*// Immediately invoked function expressions*

(function(){

var x = 1; *// this var is now protected*

})()

(function(a){

var x = a; *// this var is now protected*

})(2)

## Currying

function sum(a){

return function(b){

return function(c){

console.log(a,b,c)

return a+b+c

}

}

}

let add = a => b => c => a+b+c

let log = time => type => msg => `At ${time.toLocaleString()}: severity ${type} => ${msg}`

log(new Date())('error')('power not sufficient')

let logNow = log(new Date())

logNow('warning')('temp high')

let logErrorNow = log(new Date())('error')

logErrorNow('unknown error')

function op(operation) {

return function (a) {

return function (b) {

return operation === 'add' ? a + b : a - b;

};

};

}

const add3 = op('add')(3);

const sub3 = op('sub')(3);

const add = op('add');

add3(6);

sub3(6);

add(1)(2);

# 2. Objects

## Basic behaviours

### Reference Copying

* Variable value is not copied in case of object/arrays

let person = {name:'john'}

let human = person;

person

Object

human

Reference are point to same value

let person = {name:'john'} *// Object1*

person = {name:'wick'}; *// Object2*

person

Object1

Object2

Reference can be changed for a variable (Garbage collection of Object1)

* it a better to use const always, and whenever you must need to re-assign change it ot let

const person = {name:'john'} *// Object1*

person = {name:'wick'}; *// ERROR*

### Nested Objects

let person = {

name: 'John',

address: { city: 'delhi', state: 'delhi' },

};

person

Object

Object\_address

addressObject

Object properties can point to other objects

let addressObject = { city: 'delhi', state: 'delhi' }

let person = {

name: 'John',

address: addressObject

};

### Copying objects

#### Shallow Copy

Many methods can be used to copy object without old reference

1. **Object.assign()**

let person = {name:'john'}

let newPerson = Object.assign({}, person)

1. **Spread Operator[...]**

let person = {name:'john'}

let newPerson = {...person}

But problem which these is they just create a copy of properties of that object , but not creating a copy of their references also.

let addressObject = { city: 'delhi', state: 'delhi' }

let person = {

name: 'John',

address: addressObject

};

let newPerson = Object.assign({}, person)

person === newPerson; *// false*

person.address === newPerson.address *// true*

#### Deep Copy

This is a hard problem to solve in past as there can be multiple level of nested objects and there can be references to functions etc also. few methods which are there:

1. **JSON.stringify and JSON.parse** : this method utilizes the fact that every JSON can be converted to a string value (exception of methods/functions)

let addressObject = { city: 'delhi', state: 'delhi' }

let person = {

name: 'John',

address: addressObject

};

let str = JSON.stringify(person)

let jsonObject = JSON.parse(str);

1. **structuredClone** : Browser API which work even for circular references (but functions not supported)

let addressObject = { city: 'delhi', state: 'delhi' }

let person = {

name: 'John',

address: addressObject,

};

person.me = person

let newPerson = structuredClone(person);

### "this" and Methods

* we can also defined function as value to properties of objecy. these will be called methods. Methods are just functions but, it means they have been called in "reference" on an Object.

let person = {

name:'john',

sayHi: function(){

return "hi";

}

}

person.sayHi() *// hi*

* methods can also access the properties and other methods of same object. To do this we use this

let person = {

name:'john',

sayHi: function(){

return "hi "+ this.name;

}

}

person.sayHi() *// hi john*

* we can also have used person instead of this but has you know references can be changed. so that could have created a problem

let person = {

name:'john',

sayHi: function(){

return "hi "+ this.name;

}

}

person.sayHi() *// hi john*

* you can even have this without an object

function sayHi(){

return "hi "+ this.name;

}

sayHi() *// Error*

*// here this will "undefined" in Strict mode*

let obj1 = {name: 'john'}

let obj2 = {name: 'wick'}

*// you can add functional property*

obj1.say = sayHi;

obj2.say = sayHi;

obj1.say() *// hi john*

obj2.say() *// hi wick*

* Arrow functions don't have a this. they use outer context

let person = {

name:'john',

sayHi: ()=> {

return "hi "+ this.name;

}

}

person.sayHi() *// Error.*