Synchronous single threaded language.

All the code execution happens in the Execution Context.

Even before 1st line of JS code executes, JS engine creates the Global Execution Context.

Execution Context has 2 components

1. Variable Environment Or Memory Component

All the variables and functions get stored in key value pair

1. Code Component Or Thread of Execution

Code executes line by line.

# Hoisting

In Javascript variables and functions can be invoked even before they are declared in code. This is possible for hoisting

console.log(x); //Undefined

Hello(); // Hello World

var x=5;

function Hello(){

    console.log("Hello World!")

}

console.log(x); //5

console.log(x); //Uncaught ReferenceError: Cannot access 'x' before initialization

Hello();

let x=5;

function Hello(){

    console.log("Hello World!")

}

console.log(x);

## Hoisting var

Before execution of code, All the vars (in Global scope) initialized with undefined.

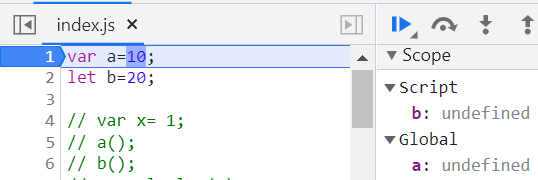
## Hoisting function

Before execution of code, JS engine initializes the function with it’s body as the value and function name as the key

|  |  |
| --- | --- |
| Function Hello(){  Console.log(“Hello World!”)} | Global / Window  Hello: f(){  Console.log(“Hello World!”)} |

## Hoisting let & const

Let & const declarations also get hoisted before code execution, they also get the special value undefined. But they don’t get attached to Global scope, in fact they get attached to Script and these can’t be invoked till some value is assigned. This period between hoisting to actual value assignment is known as Temporal Dead zone.



# Var vs let vs const

console.log(a); //10

console.log(b); //Uncaught ReferenceError: Cannot access 'b' before initialization

var a=10;

let b=20;

console.log(b);//20

console.log(c); //Uncaught ReferenceError: c is not defined

let is stricter than var. You can’t access let before assigning any value.

You can’t re-declare let. It will give syntax error and the app won’t run at all.

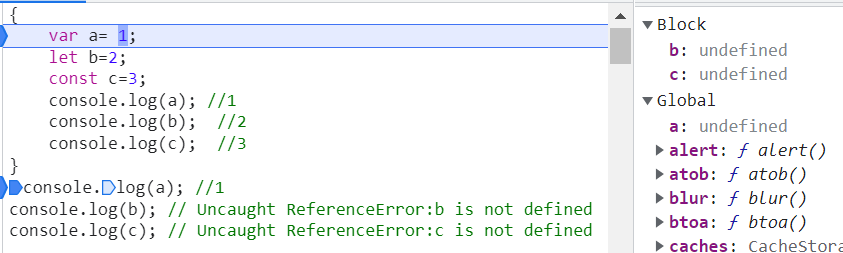
var a =5;

var a=50;

let b=1;

let b=10; //Uncaught SyntaxError: Identifier 'b' has already been declared

var is global scoped. Let & const are block scoped. Var can be accessed outside the defined block. Let and const are available on in the block.



|  |  |  |  |
| --- | --- | --- | --- |
|  | **Defined in function** | **Defined in block** | **Globally defined** |
| **var** | Local | Global | Global |
| **let** | Local | Block | Script |
| **const** | Local | Block | Script |

# Shadowing

Var Identifiers defined in a block (if, while switch etc. not function) shadow the declarations outside the block. Let & const also shadow(within the block) the declarations outside.

var a= 10;

let b=20;

const c= 30;

{

    var a= 1;

    let b=2;

    const c=3;

    console.log(a); //1

    console.log(b);  //2

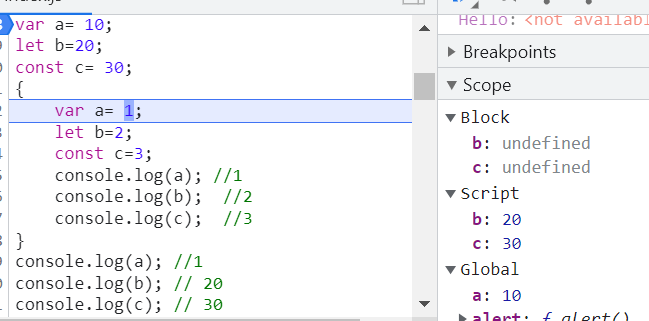
    console.log(c);  //3

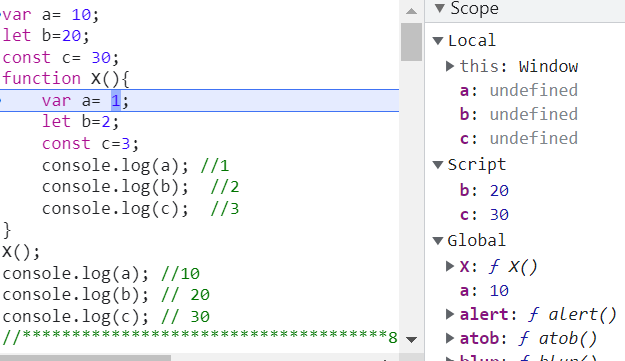
}

console.log(a); //1

console.log(b); // 20

console.log(c); // 30





The Shadowing identifier can’t have more scope than the original declaration. Var & function behave the same in terms of scope & shadowing.

Let defined outside block can’t be shadowed by a var declared in block since var in block has Global scope

|  |  |
| --- | --- |
| var a= 10;  {      var a= 1;      console.log(a); //1  }  console.log(a); //1 | var a= 10;  function X(){      var a= 1;      console.log(a); //1  }  X()  console.log(a); //10 |
| var a= 10;  {      let a= 1;      console.log(a); //1  }  console.log(a); //10 | var a= 10;  function Y(){      let a= 1;      console.log(a); //1  }  Y()  console.log(a); //10 |
| let a= 10;  {      var a= 1; // Uncaught SyntaxError: Identifier 'a' has already been declared      console.log(a);  }  console.log(a); | let a= 10;  function Z(){      var a= 1;      console.log(a); //1  }  Z()  console.log(a); //10 |
| const a= 10;  {      const a= 1;      console.log(a); //1  }  console.log(a); //10 | |
| var a= 10;  {      const a= 1;      console.log(a); //1  }  console.log(a); //10 | const a= 10;  {      var a= 1; // Uncaught SyntaxError: Identifier 'a' has already been declared      console.log(a);  }  console.log(a); |
| let a= 10;  {      const a= 1;      console.log(a); //1  }  console.log(a); //10 | const a= 10;  {      let a= 1;      console.log(a); //1  }  console.log(a); //10 |

# Closure

Function along with it’s Lexical environment.

function A(){

    var x= 5;

    function B(){

        console.log(x);

    }

    return B;  // Clolsure: Function along with it's Lexical scope

}

var c= A();

var x=50;

c(); //5

{

    var x= 5;

    function B(){

        console.log(x);

    }

    x=20;

}

B(); // 20

var x=50;

B(); //50

|  |  |
| --- | --- |
| setTimeout(function(){      console.log("Within setTimeout");  },2000);  console.log("Hello World!");  //Hello World!  //Within setTimeout | for(let i=0; i<5; i++){      setTimeout(function(){          console.log(i);      },2000);  }  console.log("Hello World!");  //Hello World!  // 0 1 2 3 4 |
| for(var i=0; i<5; i++){      setTimeout(function(){          console.log(i);      },2000);  }  console.log("Hello World!");  //Hello World!  // 5 5 5 5 5 | for(var i=0; i<5; i++){      function Close(x){          setTimeout(function(){              console.log(x);          },2000);      }      Close(i);  }  console.log("Hello World!");  //Hello World!  // 0 1 2 3 4 |

Closure helps in Data encapsulation. In the following example the “counter” variable can only be modified by the containing method through the closure incrementCounter()

function UpdateCounter(){

    var counter= 0;

    return function incrementCounter(){

        counter++;

        console.log(counter);

    }

}

var counter1 =UpdateCounter();

counter1(); counter1(); counter1(); // 1  2   3

var counter2 =UpdateCounter(); // WIll have a different instance

counter2(); counter2(); // 1   2

counter1(); // 4

counter2();//3

## Constructor function

function UpdateCounter(){

    var counter= 0; var test="";

    this.incrementCounter = function (){

        counter++;

        console.log(counter);

    }

    this.decrementCounter = function (){

        counter--;

        console.log(counter);

    }

}

var counter1 =new UpdateCounter(); // Constructor function

counter1.incrementCounter(); counter1.incrementCounter(); counter1.decrementCounter(); // 1  2   1

var counter2 =new UpdateCounter(); // Will have a different instance

counter2.decrementCounter(); counter2.incrementCounter(); // -1   0

counter1.incrementCounter(); // 2

counter2.incrementCounter();//1

closures consume more memory since the variables & functions defined in the lexical scope don’t get garbage collected till JS engine makes sure the closure is no longer needed. In the above example, by the modern browsers the variable “test” would be garbage collected once control moves out of the function scope since it’s not used in closure. But “counter” won’t get garbage collected since it’s used in closure. This may effectively lead to data leakage if not handled properly.

# Function

## Function Statement/ Declaration vs Expression

A(); //Function Statement or Function Declaration

B(); //Uncaught TypeError: B is not a function

function A(){

    console.log("Function Statement or Function Declaration")

}

var B = function(){

    console.log("Function Expression")

}

B(); //Function Expression

Function statement & expression differ in the way they hoisted.

Function expressions are treated normally like any other variable and hoisted same as var. Function expression will have undefined till the line of assignment is invoked. So a function expression can’t be invoked before it’s assigned.

## Anonymous Function

function(){ //Uncaught SyntaxError: Function statements require a function name

    console.log("Anonymous Function")

}

## Named Function Expression

A named function assigned to a variable.

var a = function xyz(){

    console.log("Named function expression");

    xyz(); // The named function can only be accessible within the function like recursion

}

a(); //Named function expression

xyz(); //Uncaught ReferenceError: xyz is not defined

## First Class Function

The ability of functions to be used as values is known as First class function. Functions can be passed as arguments, returned from a function

## Arrow Function

Let, const arrow function etc were introduced inES6/ECMA Script 2015. The scope rules for a arrow function is same as a regular function statement/declaration.

## Higher Order Function

A function that takes another function as argument is Higher Order Function.

function y(arg1){

    console.log("y");

    arg1();

}

function x(){

    console.log("x")

}

y(x);  //y is higher order function. x is first order function/ callback

Date.prototype.timeNow = function () {

    return ((this.getHours() < 10)?"0":"") + this.getHours() +":"+ ((this.getMinutes() < 10)?"0":"") + this.getMinutes() +":"+ ((this.getSeconds() < 10)?"0":"") + this.getSeconds();

}

console.log("Start: ", new Date().timeNow());

setTimeout(function(){

    console.log("callback: ", new Date().timeNow())

}, 0);

// Long running code block

const endTime = new Date(new Date().getTime() + 1000 \* 10);

while(new Date()< endTime){

}

console.log("End: ", new Date().timeNow());

//Start

//end after 10secs since the while block would run for 10secs.

//callback

# Call Stack, Callback Queue, Microtask Queue, Event loop

Event Loop keeps checking the call stack, when empty pushes the callbacks from queue for execution

Call Stack

JS code executes synchronously

Microtask Queue

Has higher priority.

Holds promises & Mutations observables

Callback Queue / Task Queue

* JS execution happens inside the call stack only.
* The callbacks from API calls & Mutation observers go in the Microtask Queue.
* The callbacks from timer(setTimeout, setInterval etc.), DOM event listener(click, hover etc) go in callback Queue or Task Queue.
* These callbacks wait in their respective Queues till the call stack is free. Callbacks in Microtask Queue have higher priority than the callbacks in the callback queue.
* Event Loop keeps checking if the call stack is empty. Once the call stack is free, Event loop pushes the callbacks in the Microtask queue one by one for execution.
* When call stack is free and all the callbacks in the Microtask Queue have executed, then Event loop pushes the callbacks in the Callback queue to the call stack for execution.

# Map() filter() reduce()

var list=[

    {    firstname:"Satya", lastname:"Dash", age:37},

    {    firstname:"Sasmita", lastname:"Dash", age:36},

    {    firstname:"Rasmita", lastname:"Dash", age:32},

    {    firstname:"Surya", lastname:"Dash", age:28},

    {    firstname:"Sandeep", lastname:"Patro", age:44},

    {    firstname:"Subhalaxmi", lastname:"Palo", age:28},

    {    firstname:"Debasish", lastname:"Misra", age:33},

    {    firstname:"Subhadarshini", lastname:"Mohapatro", age:25}

];

list.map(user=> user.fullName= user.firstname + " "+ user.lastname);

console.log(list);

var list\_35=list.filter(user=> user.age > 35);

console.log(list\_35);

// var totalAge = list.reduce(function(cum, curr){

//     cum= cum+curr.age;

//     return cum;

// }, 0);

var totalAge = list.reduce((cum, user, index) => {

    cum= cum+user.age;

    return cum;

}, 0);

console.log(totalAge);

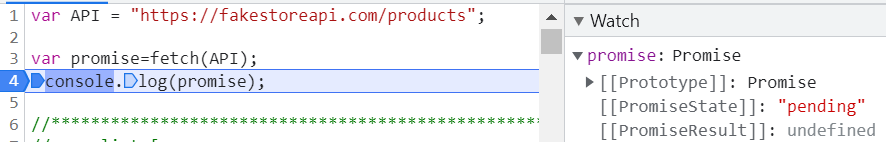
# Promise

An object representing the eventual completion or failure of asynchronous operation

Promise has 3 states. pending, fulfilled, rejected.

Javascript assures execution of the promise exactly once.

Promise oibject is immutable.



## Create Promise

// let cart=["pen", "cap","toy","glass"];

let cart=[];

createOrder(cart).then(data =>console.log(data))

.catch(err=> console.log(err.message));

function createOrder(cart){

    return new Promise(function(resolve, reject){

        if(cart.length ==0){

            const err=new Error("your cart is empty");

            reject(err);

        }else{

            let orderId= 123

            resolve(orderId);

        }

    });

}

# Call, Apply, Bind

function userDetails(city, state){

    console.log(this.firstName + " " + this.lastName + " from " + city + ", " + state);

}

let user1={

    firstName:"Rasmita",

    lastName:"Dash"

}

let user2={

    firstName:"Nipun",

    lastName:"Bajaj"

}

let user3={

    firstName:"Meenakshi",

    lastName:"Joshi"

}

userDetails.call(user1, "Bangalore", "Karnataka"); //user1 becomes this

userDetails.apply(user2, ["New Delhi", "Uttar Pradesh"]); // same as call, just extra parameters passed as list

let user3Details = userDetails.bind(user3, "Delhi", "Delhi");

user3Details();

# Function Currying

Can be achieved in 2 ways. 1. Bind 2.Higher order function

## With bind()

function multiply(x,y){

    console.log(x\*y);

}

let multiplyByTwo = multiply.bind(this,2); // x is 2

let multiplyByThree = multiply.bind(this,3); // x is 3

multiplyByTwo(10);

multiplyByThree(10);

let custom1=multiply.bind(this,4,7);

custom1();

let custom2=multiply.bind(this);

custom2(6,7)

## With Higher order functions

let sum = function(x){

    return function (y){

        console.log(x+y)

    }

}

let sum4 = sum(4);

sum4(8);

let sum3=sum(3);

sum3(8);

# async vs defer with <script>

<script src=""></script>

HTML Parsing

HTML Parsing

Script execution Parsing

Src fetching Parsing

During HTML parsing when browser encounters a script tag, HTML parsing stops for the period script downloads and executes.

<script async src=""></script>

HTML Parsing

HTML Parsing

Script execution Parsing

Src fetching Parsing

During HTML parsing when browser encounters a script tag, Script gets fetched asynchronously and parallelly with HTML parsing. But HTML parsing stops for the script execution period.

<script defer src=""></script>

HTML Parsing

Script execution Parsing

Src fetching Parsing

Script downloads parallelly with HTML parsing but script doesn’t execute till HTML parsing is completed.

# CORS

Browser follow the same-origin policy and hence restricts HTTP requests initiated from script to different domain, sub-domain, port, protocols for security reasons . With CORS resource sharing is possible.

<https://rasmita.in> can’t access followings

<https://google.com/api/getUser>

<https://api.rasmita.in>

<https://rasmita.in:5050>

<http://rasmita.in>

## Simple Request

The client browser makes a direct request to the remote server and if remote server has enabled CORS access, client gets response.

GET

HEAD

POST with specific headers like

content-type application/x-www-form-urlencoded

Multipart/form-data

Text/plain

## Preflight Request

Client browser first makes a preflight call to the server to check if the call is safe to make. Remote server returns response with a few additional OPTIONS in the header and then the client browser makes the actual API call

Access-Control-Allow-Origin

Access-Control-Allow-Methods

Access-Control-Allow-Headers

Access-Control-Max-Age: For how long the response to the preflight request can be cached before making another preflight request. Default 5 seconds.

# Web Storage

Storage data follows the same origin policy. Local data created by a domain can only be accessed by the same domain. It’s not available to other domain, sub-domain, protocol or port.

## Cookies

Maximum capacity is 4KB. With every request cookie data is sent to the server and back.

## Session Storage

Maximum capacity 5MB.

Data is stored for the browser session. Once user closes the browser or the browser tab, stored data is lost.

## Local Storage

Size depends on the browser and the user machine. However, it’s more than 5MB even for a mobile.

Data stored is available through out till it’s expiration time or till user clears it.

Provided by browser and available in the window object.  
var user={

    "Name":"Rasmita",

    "DOB":"10th Oct 1991"

}

localStorage.setItem("name", "Rasmita");

localStorage.setItem("user", JSON.stringify(user));

localStorage.getItem("name");

JSON.parse(localStorage.getItem("user"));

// localStorage.removeItem("name");

// localStorage.clear();

