**JavaScript**

JavaScript is a programming language used to make webpages interactive.

Everything in JavaScript happens inside an Execution Context.

This is how Execution Context looks like

There are two components

|  |  |
| --- | --- |
| **Memory(Variable Environment)**  Variables and functions are stored as key Value Pair  Key : Value  A:10 | **Code(Thread of Execution)**  In which code is executed one line at a time. |

**Java Script is a synchronous single threaded (meaning it can execute only one command at a time unlike Java)**

For Example:

Lets say this is a code

1. var n = 2;
2. function square (num) {
3. var ans = num \* num;
4. return ans;
5. }
6. var square2 = square(n);
7. var square4 = square(4);

Global Execution context is created;

First the Memory creation phase:

|  |  |
| --- | --- |
| **Memory**  n : undefined  square : {function code…}  square1 : undefined  square2 : undefined | **Code** |

Second phase(code execution)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Memory**  n : 2  square2 : 4  Square4 : 16 | **Code**  Line 1  When line 6 is executed a new execution context is created   |  |  | | --- | --- | | Memory | Code | | ans : undefined  num: undefined | When executing function  num becomes n which is equal to 2 and ans will become 4 | | After the execution this context is deleted |  |   When line 7 is executed again a new execution context is created.   |  |  | | --- | --- | | **Memory**  num : undefined  ans : undefined | **Code**  When executing function  num becomes 4 and ans will become 16 |   After the execution of function this context will also be deleted |

In order to implement Execution contexts JavaScript creates a Call stack

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| First Global Exec. Context   |  | | --- | |  | |  | | GEC | | Execution context one   |  | | --- | |  | | EC1 | | GEC | | EC1 is popped out and EC2 is inserted   |  | | --- | |  | | EC2 | | GEC | |

After completion of EC2 it is popped out and then GEC is popped out.

**Call Stack maintains the order of execution of execution contexts.**

Call stack is also called Execution context stack, Program stack, control stack, Runtime stack and Machine stack.

**Hoisting.**

Hoisting is the JavaScript’s default behavior of allocating memory to the Variables and Functions even before the code is executed (also known as Variable Environment)

For Example:

1. var x = 7;
2. function getName() {
3. console.log(“Hello World”);
4. }
5. getName();
6. console.log(x);

The output for the above code will be

Hello World

7

Now if we rearrange the code:

1. getName();
2. cosole.log(x);
3. var x = 7;
4. function getName() {
5. console.log(“Hello World”);
6. }
7. var getName2 = ()=>{
8. console.log(“Hello”);
9. }
10. Var getName3 = function(){
11. Console.log(“Hello”);
12. }

Output of the code

Hello World

Undefined

That is because when creating Memory phase the variable x is assigned value undefined while the function will be assigned its whole value.

Things to be noted is that getName2 and getName3 will also be treated as variables and not functions therefore they will be assigned value undefined during the Memory Phase.

**Functions**

A block of code to perform specific task.

1. var x = 1;
2. a();
3. b();
4. cosole.log(x);
5. function a(){
6. var x = 10;
7. console.log(x);
8. }
9. function b(){
10. var x = 100;
11. console.log(x);
12. }

Output will be

10

100

1

|  |  |  |  |
| --- | --- | --- | --- |
| **Memory**  x : undefined  a = {}  b = {} | **Code**  When line executes x becomes 1  When line2 is executed a new execution context is created   |  |  | | --- | --- | | Memory  x : undefined | Code  When function a() is executed x becomes 10 which is not related to Global x even though they have same name |   Now this execution context is deleted same happens for b() and the context is deleted  When line 4 is executed value of x is taken from local memory which is 1 thus it will print 1 |

**What is “this” and “Window”?**

Whenever a Execution context is created a new “this” object is created with it. In case of bowsers this Object is usually equal to window when creating a global object.

1. var x = 1;
2. console.log(x);
3. console.log(this.x);
4. console.log(window.x);

output will be

1

1

1

In above case the global “this” is equal to window

**JavaScript creates a global execution context and assign memory to the variables and function even before the code is even run.**

What is Undefined?

Undefined is a special keyword in javascript that acts as a place holder for any variable until some value is put into it.

**Lexical Environment**

Lexical means in order or hierarchy.

Lexical Environment is the Local Memory with the Lexical Environment of its Parent.

1. function a(){
2. var b = 10;
3. c();
4. function c(){
5. console.log(b);
6. }
7. }
8. a();
9. console.log(b);

when the program is executed callstack will look like

|  |  |  |  |
| --- | --- | --- | --- |
| c()   |  |  |  | | --- | --- | --- | | Memory   |  | | --- | | Parent’s lexical environment (b()) | | Code | |
| b()   |  |  | | --- | --- | | Memory  b:10  Parents’s lexical environment(Global) | Code | |
| Global   |  |  | | --- | --- | | Memory | Code | |

When we log b in c() it looks for b in its local first and then parents local lexical environment if it does not find it will look for b in the local lexical environment of the parent of its parent which is also known as Scope chaining.

**Let and Const**

Are let and const hoisted?

Yes, but we can’t access value of let before its initialization (a.k.a Temporal Dead Zone) unlike var. Also they are allocated a different memory space unlike var. therefore can not be accessed through window keyword.

**Temporal Dead Zone**

The phase between a variable is allocated memory and its initialization. That’s the reason we cant access value of let before its initialization. If we try to access it, it throws a reference error.

const is more strict than let which in turn is more strict than var

1. const z = 100; //correct
2. const y;
3. y = 100; //throws error const should be initialized when declaring
4. let x = 100; //correct
5. let y;
6. y = 100; //correct
7. let y = 10; // throws error y is used again y already been declared

**Type error**:- assignment to a constant variable.

**Syntax error**:- const b; missing initializer, redeclare variable

**Reference error**:- accessing something that not has been initialized(let) or not declared(var)

**Block(Compound Statement)**

Grouping multiple statements in a block where javascript excepts single statement.

**Block Scope**

The variables declared inside block can only be accessed inside the block. Let and Const are Block scoped.

**Shadowing**

Using local value of the block instead of the global value is called shadowing.

1. let b = 1;
2. var a = 100;
3. {
4. var a = 10;
5. let b = 10;
6. const c = 10;
7. console.log(a); //will print 10
8. console.log(c); // 10
9. console.log(b); // 10 as it will shadow global b
10. }
11. console.log(a); // 10 because a has global scope and it gets changed even inside
12. console.log(c); // reference error because const c has block scope.
13. console.log(b); // 1 from the global variable

**Illegal Shadowing**

A let variable can be re declared inside a block as ‘let’ only but not as var while var can be re declared inside the block as ‘let’.

**Closure**

Binding a function with its lexical scope is called closure. Function with its lexical scope bundled together creates a closure.

1. function x(){
2. var a = 7;
3. function y(){
4. console.log(a)
5. }
6. return y;
7. }
8. var z = x();
9. console.log(z); //z prints a function
10. z(); // it prints 7 as y returns a function with its lexical scope which contains has a

Uses of Closures:

1. Module Design Pattern
2. Currying
3. Functions like once
4. Memoize
5. Maintain state in async world
6. setTimeouts
7. Iterators

More on Closures

**Write a program to print 1,2,3,4,5 each after 1s, 2s, 3s, 4s and 5s respectively.**

1. function x(){
2. for(var i = 1; i < = 5; i++){
3. setTimeout(function() {
4. console.log(i);
5. }, i\*1000);
6. }
7. console.log(“Hello”);
8. }

Will print

Hello

6

6

6

6

6

Because the setTimeout stores the function and attaches time to it with the reference to the variable’s memory rather than its value so when the callback function is called the value of i has been changed to 6 as var has a global scope.

One solution to the above problem is using let instead of var because var has block scope.

1. function x(){
2. for(let i = 1; i < = 5; i++){
3. setTimeout(function() {
4. console.log(i);
5. }, i\*1000);
6. }
7. console.log(“Hello”);
8. }

If we want to use var using closure

1. function x(){
2. for(let i = 1; i < = 5; i++){
3. function close(x){
4. setTimeout(function() {
5. console.log(x);
6. }, x\*1000);
7. }
8. close(i);
9. }
10. console.log(“Hello”);
11. }

**Anonymous Functions**

A function without a name is called an anonymous function.

**Function statement (Function Declaration)**

1. function a(){
2. console.log(“Hello”);
3. }

**Function expression**

1. var b = function (){
2. console.log(“Hello”);
3. }

Difference between function expression and function statement.

During variable environment phase function statement will have value as function wile expression will have undefined.

Anonymous function does not have an identity so it’s used in Function expression (or when we return a function).

**Named function expression**

1. var b = function xyz(){
2. console.log(“Hello”);
3. }

**Parameters and Arguments**

1. var b = function (param){ //parameter
2. console.log(param);
3. }
4. b(“Hello World”); // argument

**First Class Function**

1. var b = function (param){
2. console.log(param);
3. }
4. b(function (){
5. var x = 13;
6. });

The ability of functions to use as values either as argument or as return type.

Functions are first class citizens (same thing)

**CallBack Function**

A function passed to another function, that is called sometime later in the code.

Example

1. setTimeout(function () {
2. console.log(“timer”);
3. }, 3000);

Hence it mimics asynchronous behavior.

Also setTimeout does not wait in callstack but stored somewhere else.

CallStack is also called as Main Thread. If any context in callstack is not able to pop its called blocking of the main thread, hence async operations are used where response takes time.

**Event Listener**

1. document.getElementById(“ID”)
2. .addEventListener(“click”, function(){
3. Console.log(“callback function”);
4. });

Event Listener takes a lot of memory.

**Event** **Loop**

|  |
| --- |
| **Browser**  Timer Url Local Storage Location Bluetooth DOM Apis Console |

1. setTimeout(function(){
2. console.log(‘callback’);
3. }, 5000);

|  |
| --- |
| Web APIs Window  setTimeout()  DOM APIs  Fetch() (External Scope)  LocalStorage  Console  Location  cb() |

Console, setTimeout all are not the part of javascript but APIs that js makes to browser in order to access its different parts. Fetch acts different than other these APIs as function that return promises are stored in micro task queue which has higher priority than callback stack.

**Callback** **Queue**

|  |  |  |
| --- | --- | --- |
| Call Stack | EventLoop | Callback Queue cb() |

Event Loop acts as a gatekeeper between these two to execute programs on callback queue and Micro task Queue.

Only job of Event loop is to govern call stack in order to execute program that are in callback queue.

We need call back Queue in case there are multiple events waiting to be executing

JavaScript Runtime Environment(Browser,Node.js)

|  |
| --- |
| API  CallBack Queue  Microtask Queue  JS Engine(callStack, Memory heap Event Loop  Garbage Collector Mark & Sweep algorithm) |

JsEngine – V8, SpiderMonkey(First JSEngine created by Brandon Eyke)

What happens inside JSEngine?

Code -> Parsing -> Compilation -> Execution

Code is broken down into tokens (like in let a = 10 let, a , = and 10 form three different tokens), then comes syntax parser which converts tokens into abstract syntax tree (that looks like a json file).

Than comes compilation, so Java Script has JIT compilation which takes the best of both interpreter and compilation

Inlining, Copy Elision and inline caching are used to optimize.