

1) **How does JavaScript works?**

Everything in JavaScript happens inside an "execution context".

2) Execution context has two component

a) **memory component [variable environment]** = This is the place where all variables and functions are stored as key value pairs. eg-{key: value || n:2;}

b) **code component [Thread of execution]** = This is the place where code is executed one line at a time

3) JavaScript is asynchronous single-threaded language

Single threaded means JavaScript can execute once command at a time

4) Synchronous single-threaded that means JavaScript can execute one command at a time in a specific order.

5) In JS, before the code is executed, the variables get initialised to undefined.

6) Arrow functions enact as variables and get "undefined" during the memory creation phase while functions actually get run.

---

7) **Hoisting:** Mechanism in JS where the variable declarations are moved to the top of the scope before execution. Therefore it is possible to call a function before initialising it.

8) Whenever a JS program is run, a global execution block is created, which comprises of 2: Memory creation and Code execution.

9) Variable declarations are scanned and are made undefined.

10) Function declarations are scanned and are made available

---

11) Memory allocation phase- all the variables and functions get their memory allocated in the memory with undefined and the entire code respectively.

12) Code execution phase - in this phase thread execution happens and all the variables get their actual values which were assigned to them and as function is invoked, a new execution environment gets created in the code part, and again there are two phases, memory allocation phase and code execution phase. And the cycle repeats.

---

13) We learnt how functions work in JS.

14) At first a global execution context is created, which consists of Memory and code and has 2 phases: Memory allocation phase and code execution phase.

15) In the first phase, the variables are assigned "undefined" while functions have their own code.

16) Whenever there is a function declaration in the code, a separate local execution context gets created having its own phases and is pushed into the call stack.

17) Once the function ends, the EC is removed from the call stack.

18) When the program ends, even the global EC is pulled out of the call stack.

---

19) Reserves the memory space specifically for GEC to be created in stack

20) GEC is created

21) Creates a 'Window': a javascript 'global object' which 'runs with GEC' with an object whose values are in global scope(can be accessed by using any of the key in [1:37](#) )

22) Js object 'this' is created (really the name is 'this') and this level this=== window

23) then our script starts execution

24) The variable in javascript always assigns its value from Global level (unless specified earlier 'in the script' itself or in function)

25) and if you defined a variable (eg. a=10) : this.a===10; global.a===10;

---

- 26) Undefined is like a placeholder till a variable is not assigned a value.
- 27) undefined !== not defined
- 28) Not defined: memory is not allocated
- 29) undefined: memory is allocated
- 30) Java is a **loosely typed language**. It is very flexible. It can hold multiple data of different data type.
- 

- 31) Scope of a variable is directly dependent on the lexical environment.
- 32) Whenever an execution context is created, a lexical environment is created. Lexical environment is the local memory along with the lexical environment of its parent. Lexical as a term means in hierarchy or in sequence.
- 33) Having the reference of the parent's lexical environment means the child or the local function can access all the variables and functions defined in the memory space of its lexical parent.
- 34) The JS engine first searches for a variable in the current local memory space, if it's not found here it searches for the variable in the lexical environment of its parent, and if it's still not found, then it searches that variable in the subsequent lexical environments, and the sequence goes on until the variable is found in some lexical environment or the lexical environment becomes NULL.
- 35) The mechanism of searching variables in the subsequent lexical environments is known as Scope Chain. If a variable is not found anywhere, then we say that the variable is not present in the scope chain.
- 

- 36) **Let** and **const** are in the temporal dead zone for time being.
- 37) let and const are hoisted that is memory is allocated but they are stored in a separate memory space which cannot be accessed before initialisation.
- 38) **The Temporal Dead Zone** exists until a variable is declared and assigned a value.
- 39) We cannot redeclare the same variable with let/const(even with using var the second time).
- 40) level of strictness ... var<<let<<const.
- 41) const variable declaration and initialisation must be done on the same line.
- 42) **Types of declaration:=**
- a) var : no temporal dead zone, can redeclare and re-initialize, stored in GES
  - b) let : use TDZ, can't re-declare, can re-initialize, stored in separate memory
  - c) const : use TDZ, can't re-declare, can't re-initialize, stored in separate memory

- 43) **Types of error:=**
- a) syntax error ... violation of JS syntax
  - b) type error ... while trying to re-initialize const variable
  - c) reference error ... while trying to access variable which is not there in global memory.
- 

- 44) Multiple statements are grouped inside a block so it can be written where JS expects single statements like in if, else, loop, function etc.
- 45) Block values are stored inside separate memory than global. They are stored in block. (the reason let and const are called block scope).
- 46) block follows the lexical scope chain pattern while accessing the variable.
- 47) **Shadowing** :- Providing same name to the variable as of those variable which are present in outer scope.
- 48) The shadow should not cross the scope of original otherwise it will give error.

49) Shadowing with var is illegal shadowing and gives errors.

Illegal shadowing:

```
let a = 200;
{
  var a = 20;
}
```

50) as 'var' declaration goes to 'Global environment' and sets it in the Memory context, it cannot be set using the 'Block environment' value Hence: Uncaught SyntaxError: Identifier 'a' has already been declared.

51) Var value is stored in the nearest outer function or global scope and hence can be accessed outside the block as well whereas the same is not the case with let and const.

---

52) **Closure** :Function bundled with its lexical environment is known as a closure. Whenever a function is returned, even if it vanishes in execution context, it still remembers the reference it was pointing to. It's not just that function alone it returns but the entire closure and that's where it becomes interesting

```
Ex
Function x(){
  Var a=7;
  Function y(){
    console.log(a);
  }
  Return y;
}
Var z=x();
console.log(z);] //y()
z(); //7
```

53) In closures, values in variable are 'pass by reference' hence their values can be changed, and whenever comes the closure inside closure(multilevel one) we can see the function refers the 'parent' variable

54) **Advantages of Closure**

- a) Module Design Pattern
- b) Currying
- c) Functions like once
- d) Memoize
- e) Maintaining state in Async world
- f) setTimeouts
- g) Iterators
- h) Data Hiding and encapsulation

---

55) setTimeout stores the function in a different place and attached a timer to it, when the timer is finished it rejoins the call stack and executes.

56) Without closure the var reference gives the latest value as it does not retain the original value but rather has the reference so any update in value after timeout will be shown.

57) If we use let/const because they have block scope, every time a new copy of variable is attached, thus this can be done without closure.

Ex

```
function x(){
  for(var i=1;i<=5;i++){
    setTimeout(function (){
      console.log(i);
    },i*1000)
  }
  x();          // 6 6 6 6 6 6
```

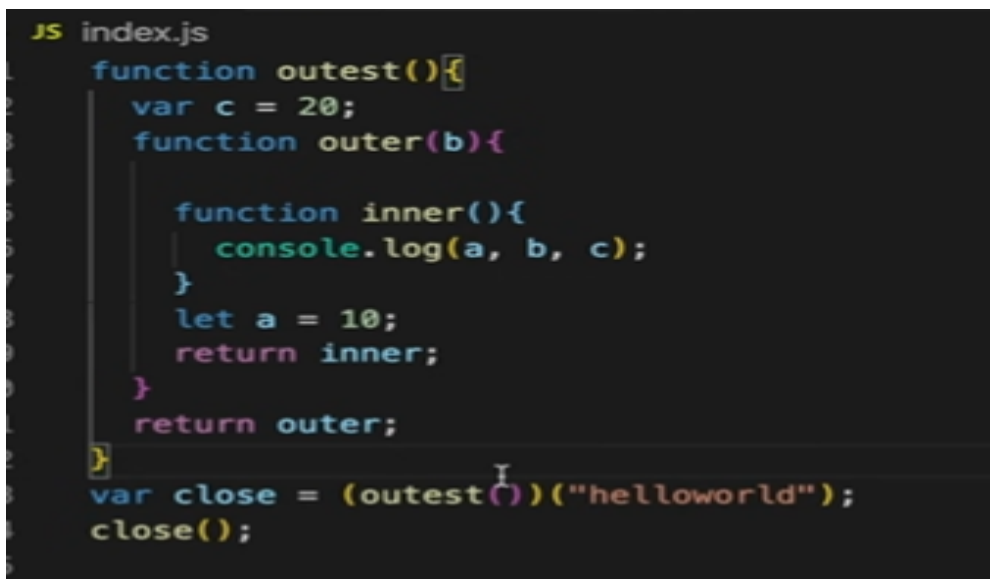
```
function x(){
  for(let i=1;i<=5;i++){
    setTimeout(function (){
      console.log(i);
    },i*1000)
  }
  x();          // 1 2 3 4 5
```

```
function x(){
  for(var i=1;i<=5;i++){
    Function close(x){
      setTimeout(function (){
        console.log(x);
      },x*1000)
    }
    close(i);
  }
  x();
```

---

58) An inner function can be directly called using two parenthesis ()().

59) Even parameters can be passed this way (Remember that the function needs to be returned to do this)



```
JS index.js
function outest(){
  var c = 20;
  function outer(b){
    function inner(){
      console.log(a, b, c);
    }
    let a = 10;
    return inner;
  }
  return outer;
}
var close = (outest())("helloworld");
close();
```

60) Closures can also be used for data hiding and encapsulation. So other code cannot access this value.

Ex:

```
js > JS index.js
1  function counter(){
2      var count = 0;
3      return function incrementCounter(){
4          count++;
5          console.log(count);
6      }
7  }
8
9  var counter1 = counter()
10 counter1();
11 counter1();
12
13 var counter2 = counter();
14 counter2();
15 counter2();counter2();counter2();counter2();
```

#### 61) Function Constructor

```
function Counter(){
    var count = 0;
    this.incrementCounter = function (){
        count++;
        console.log(count);
    }
    this.decrementCounter = function (){
        count--;
        console.log(count);
    }
}
var counter1 = new Counter();

counter1.incrementCounter();
counter1.incrementCounter();
```

#### 62) Disadvantages of Closure

Over consumption of memory because every time a closure is formed it consumes memory and those variables are not garbage collected, if it is not handled properly it can lead to memory leak.

63) Unused variables are automatically deleted in High Level Programming language by garbage collector.

64) Some browsers now have smart garbage collectors that automatically deletes variables that are not used outside closures.

---

65) What is a **Function Statement** or **Declaration**?

A normal function that we create using Naming convention. & By this we can do the Hoisting.

For Ex -

```
function a(){  
  console.log("Function Statement");  
}
```

66) What is **Function Expression**?

When we assign a function into a variable that is Function Expression. & We can not do Hoisting by this because it acts like a variable.

For Ex -

```
var a = function(){  
  console.log("Function Expression");  
}
```

67) What is **Anonymous Function**?

A Function without the name is known as Anonymous Function. & It is used in a place where function are treated as value.

For Ex -

```
function(){  
}
```

68) What is **Named Function Expression**?

A function with a name is known as Named Function Expression.

For Ex -

```
var a = function xyx(){  
  console.log("Names Function Expression");  
}
```

69) Difference b/w **Parameters** and **Arguments** ?

When we create a function & put some variables in this ( ) that is our Parameters.

For Ex -

```
function ab( param1, param2 ){  
  console.log(" ");  
}
```

And When we call this function & pass a variable in this ( ) that is our Arguments

For Ex - ab( 4, 5 );

70) What is **First Class Function** Or **First class citizens**?

- The Ability of use function as value,
- Can be passed as an Argument,
- Can be executed inside a closure function &
- Can be taken as a return form.

For Ex -

```
var b = function(param){  
  return function xyz(){  
    console.log(" F C F ");  
  }  
}
```

71) **Functions** are the heart of JS. They are called first class citizens or first class functions because they have the ability to be stored in the variables, passed as parameters and arguments. They can also be returned in the function.

---

72) **Callback Function:** Function that is passed on as argument to another function is called callback function.

Ex:  
 Function x(){  
 }  
 x (function y(){  
  
 })

73) JS is single threaded and synchronous. SetTimeout turns JS into asynchronous.

```

setTimeout(function () {
  console.log("timer");
}, 5000);

function x(y) {
  console.log("x");
  y();
}
x(function y() {
  console.log("y");
});

```

74) Event listeners can also invoke closures with scope.

75) Event listeners consume a lot of memory which can potentially slow down the website therefore it is good practice to remove it if it is not used.

---

76) The **Event Loop** pushes the "queue" into the Call Stack only when the Call Stack is empty (i.e. the global execution context has been pushed off the call stack).

77) The order in which the Event Loop works is:

- a) Call Stack
- b) Microtask Queue
- c) Callback Queue

78) Promises and MutationObserver goes inside Microtask Queue

79) Callback function from setTimeout and DOM APIs are passed into Callback Queue also known as Task Queue

80) Event Loop completes the task in Microtask Queue then performs the task of Callback Queue.

81) If any task in Microtask Queue creates another Microtask Queue which again creates another Microtask Queue then the task in Callback Queue will never be completed and this leads to **Starvation**.

---

82) The JS runtime environment contains all elements required to run JS.

83) It contains a JS engine, set of API's, callback queue, microtask queue, event loop.

84) The JS engine is a piece of code.

85) Process includes Parsing ---> Compilation -----> Execution.

86) Parsing breaks code into tokens and converts it into AST(Abstract Syntax Tree).

87) Modern JS engine follows JIT compilation, it interprets while it optimises code as much as it can.

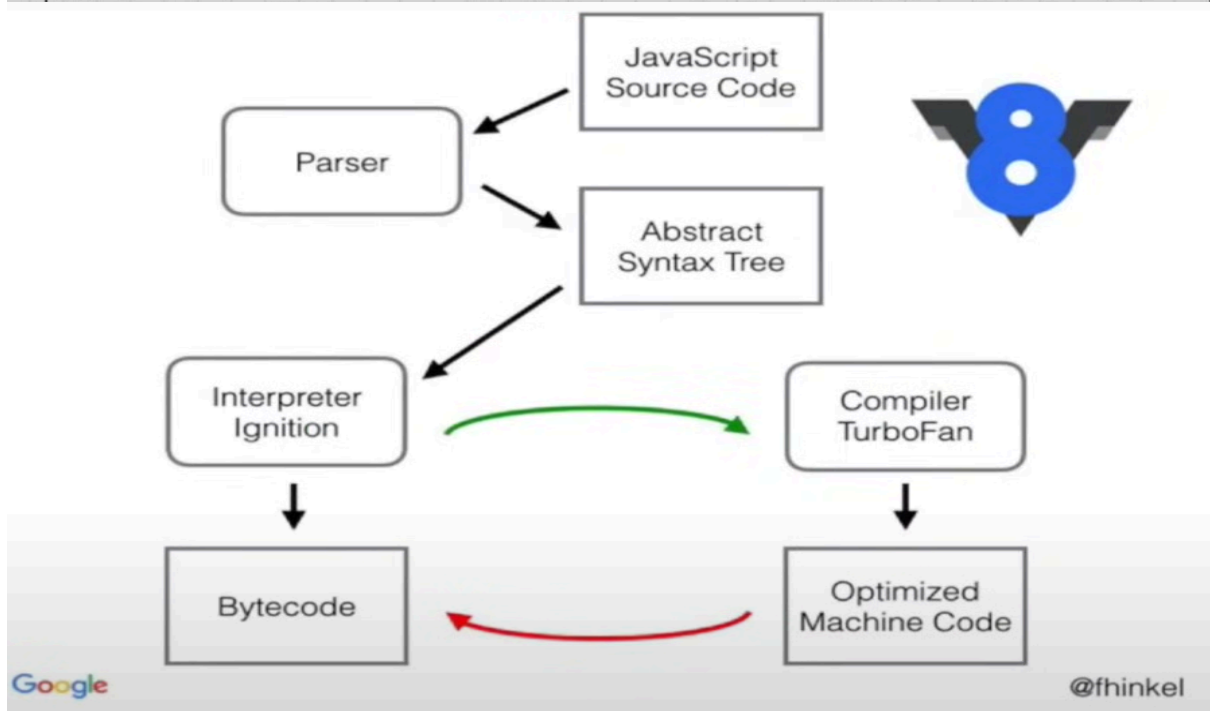
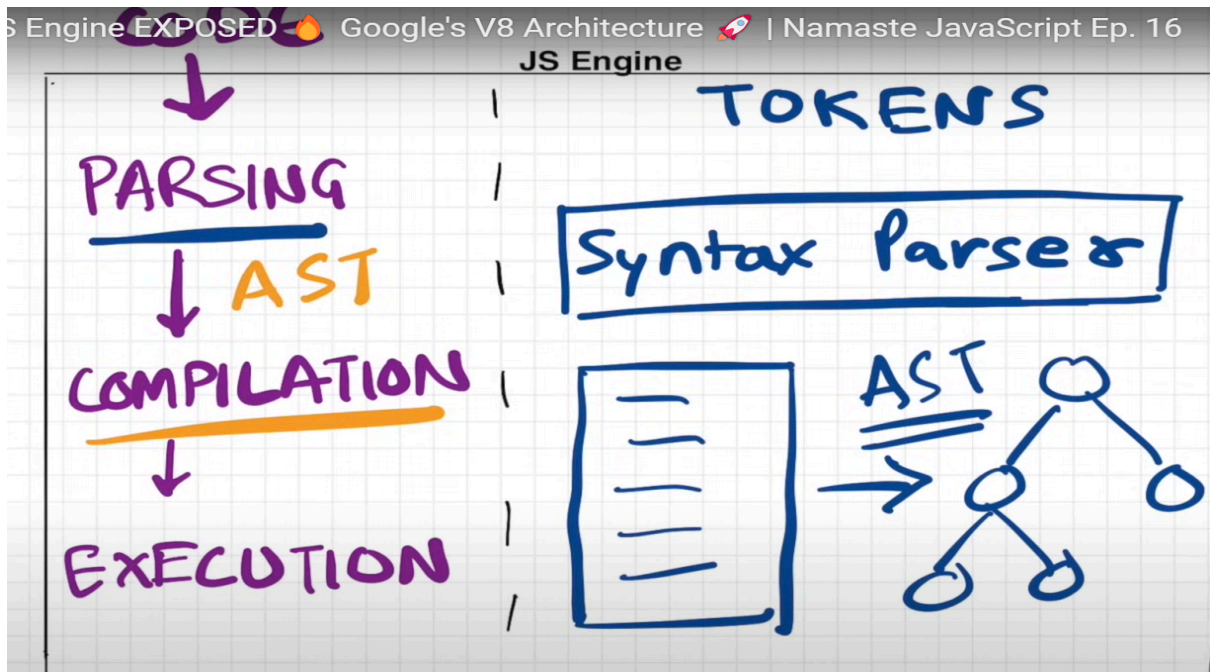
88) Execution and Compilation are done together.

89) Execution has Garbage collector and other optimisation such as inlining, copy elusion, inline caching etc.

## 90) Basic idea about Mark & Sweep Algo:

- a) Mark phase = All objects are marked as 0 initially (at creation) and in mark phase the objects that will be accessible are marked as 1 (reachable) by a DFS graph traversal.
- b) Sweep phase = During sweep phase, the objects marked with 0 are removed from heap memory. and also all reachable objects are again initialised with 0 (made unreachable) because the algorithm will run again.

So, it's basically tracing garbage collector concept. :)





- 91) The `setTimeout` function stores it in the callback queue which is executed only after the call stack is empty, even if `setTimeout` is set to 0ms.
- 92) `SetTimeout` ensures that at minimum it will take the time mentioned because it may be paused due to the call stack not empty.

```
console.log("Start");

setTimeout(function cb() {
  console.log("Callback");
}, 5000);

console.log("End");

// million

let startDate = new Date().getTime();
let endDate = startDate;
while (endDate < startDate + 10000) {
  endDate = new Date().getTime();
}

console.log("While expires");
```

- 
- 93) Follow DRY(Don't Repeat Yourself) principle while coding.
- 94) Use function to stop writing repeating lines of codes.
- 95) Function that takes another function as argument(callback function) is known as Higher order functions.
- 96) It is this ability that functions can be stored, passed and returned, they are called first class citizens.
- 97) If we use `Array.prototype.function-name`. This function is accessible to any array in your code.

- 
- 98) `map` method is used when we want transformation of the whole array.
- 99) `filter` is used when we want to filter the array to obtain required value.
- 100) `reduce` is used when we want to reduce the array to a single value eg (max, min, avg, sum, difference etc).
- 101) `reduce` passes two arguments: one function(which includes accumulator and initial value as argument itself) and another initial value of accumulator.
- 102) inheritance in JS => When an object is trying to access variables and properties of another object.
- 103) prototype is an Object that get attach to function/method/object and this object has some hidden properties
- 104) Whenever we create object/ function/ methods/ array/ variable , these all are attached with some hidden properties, which we call prototype
- 105) `__proto__` is reference to prototype ( or it points towards prototype ), if we want to access prototype, we do `__proto__`
- 106) a prototype object has a prototype of its own, and so on until an object is reached with null as its prototype, this is called prototype chaining.