**JAVASCRIPT**

**JavaScript Foundation I**

**JavaScript engine:**

* Translator
* Understands js codes and Executes js codes (tells computer what to do )
* Many JavaScript engines (ECMASCRIPT engines )
* Example : V8 ( c++ , google )
* JS Engine can be built by anybody
* [Babel](https://babeljs.io/) is a JavaScript compiler that takes your modern JS code and returns browser compatible JS (older JS code).
* [Typescript](https://www.typescriptlang.org/) is a superset of JavaScript that compiles down to JavaScript.
* **V8 engine** uses Interpreter and compiler - **JIT compiler** (just in time compiler)
* JIT uses- interpreter to produce byte code which starts execution faster and compiler produce Optimized code which is added later for faster execution
* JavaScript is an interpreted or compiled language (or both) depending on the implementation
* The More predictable the code is the better as optimization is better (for user/machine)
* **WebAssembly** (abbreviated *Wasm*) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable target for compilation of high-level languages like C/C++/Rust, enabling deployment on the web for client and server applications.
* JavaScript engine -> **call stack and memory heap**
  + - **Memory heap - >** storage space where information is stored
    - **Call stack ->** Maintains the current execution information as stack.
* Call stack -> **stack overflow** occurs when memory limit exceeds (ex: recursion)
* Memory heap - > **memory leak** occurs when memory is used more and more .Memory leak may occur when  using

i) array ii) global variable iii) Event Listeners iv)setInterval()

* JavaScript is a garbage collected language - cleans the memory that is no longer used - uses mark and sweep algorithms
* JavaScript is a **single threaded language(has only one call stack and one heap)** - only one set of instructions can be executed at a time - synchronous - simple to implement but it has disadvantage as other actions cannot be performed at a given time
* **JavaScript Runtime** - //JavaScript engine alone does not execute the program.
* Browser has JSRuntime

Example program :

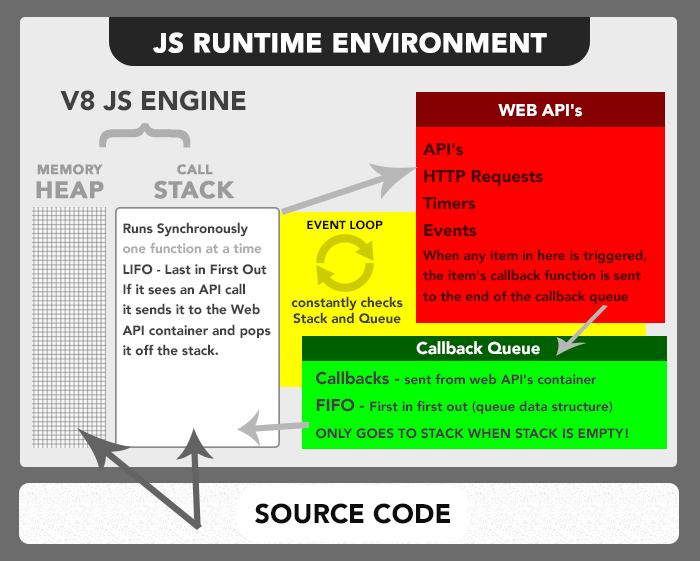
console.log('1');

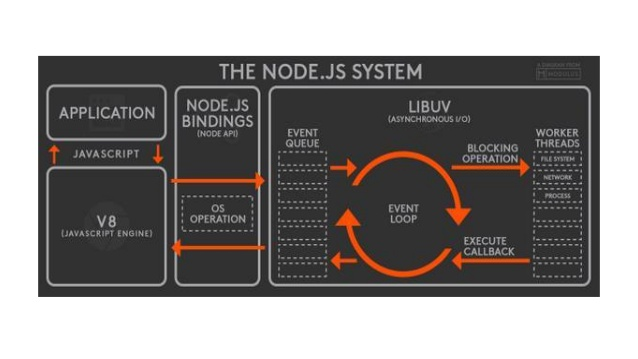
setInterval(()=>{console.log('2'),1000});

console.log('3');

After execution:132

**Node.js** (node.exe - executable) **Runtime** - similar to browser based runtime but it can access system resources. It runs outside the browser.





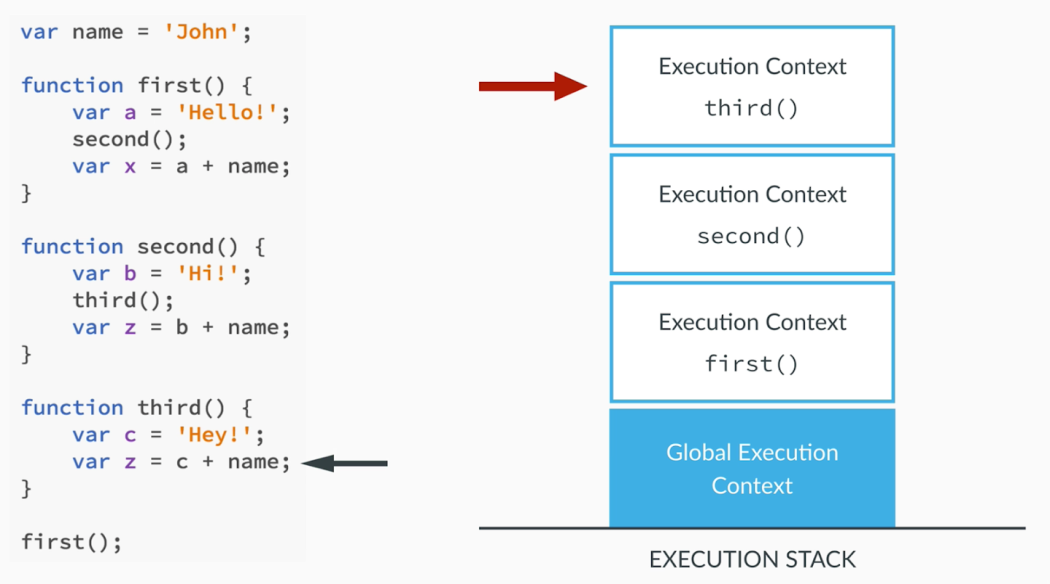
**JavaScript Foundation II**

**Execution Context:**

JS engine when sees the function bracket it is going to create execution context.

Creates an execution context and adds the execution context on the call stack

Another function -> creates  new execution context and so on.

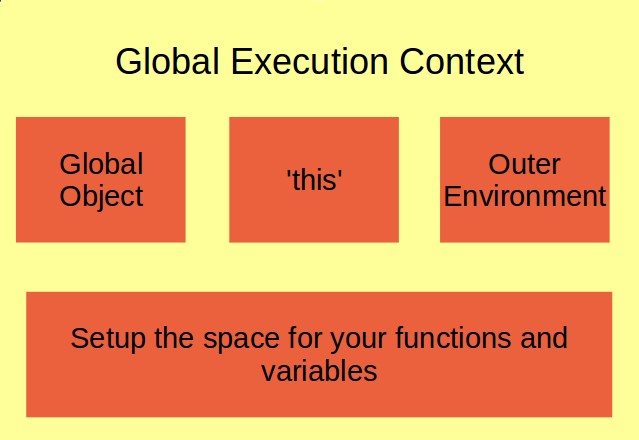


**Global Execution Context:**

Initially JS engine creates a global execution context then adds the user context

When ever code is run , it is always going to be in execution context - > global or function context

(empty file itself creates **global** **object** and **this** - by JS ENGINE)



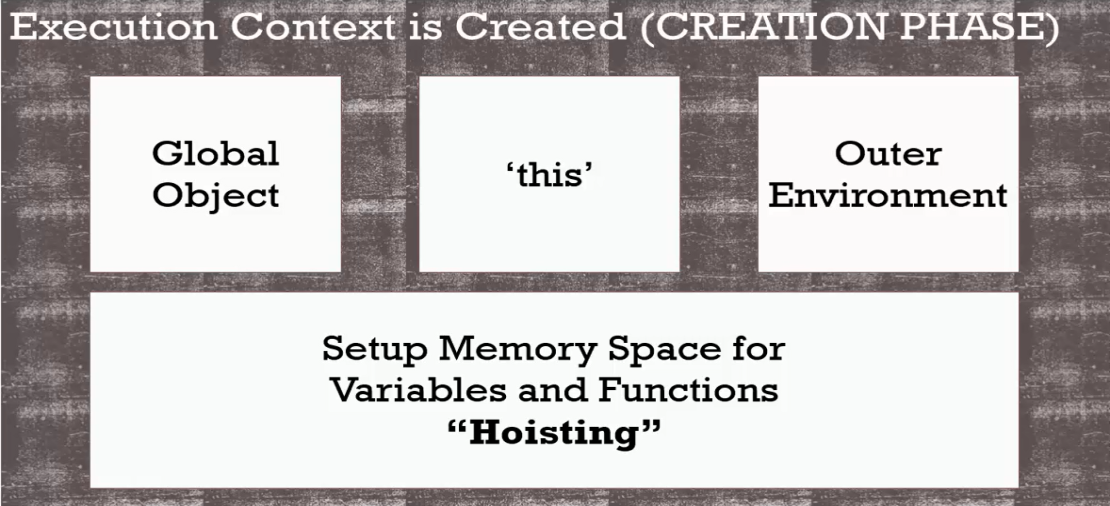
Global object is **window** in the browser because browser has created global execution context

Creation phase - global execution context and execution phase - your function execution context in node.js the global object is **global**

**Lexical Environment:**

* Where you write something or where the code is available/written.
* An environment / separate universe is created for (say each function in the execution context)
* Compiler does lexical analysis - it is checking to see the words where it is written and the location ( or what part of the universe it is in)
* Every time a function is created a new planet is created
* An execution context shows which lexical environment is currently running
* **Lexical scope** = available data + variables where the function was defined (not where the function is called - **dynamic scope**)

**Hoisting:**



* variables are partially hoisted and functions are fully hoisted (before execution phase where is code is run)
* creation phase :  allocates memory for variables but does not  store values so undefined(partially hoisted)
* adds function in memory (fully hoisted)
  + One pass through the code during hoisting
  + Function expression -> its like variable so undefined.
  + Only var and function keywords gets memory during hoisting , no other keywords are hoisted.
* Note: **For each function hoisting is done** . Even during execution , when function is called it enters into creation phase where hoisting is done and then it goes into execution phase.
* You can avoid hoisting issues by not using **‘var’** ,instead use **‘const’** or **‘let’**.

**Function Invocation:**

Without functions programs would not do anything.

**Function Expression**

var canada = function (){ or var canada = () => {

console.log(“cold”); console.log(“cold”);

} }

**function declaration :**

function india(){

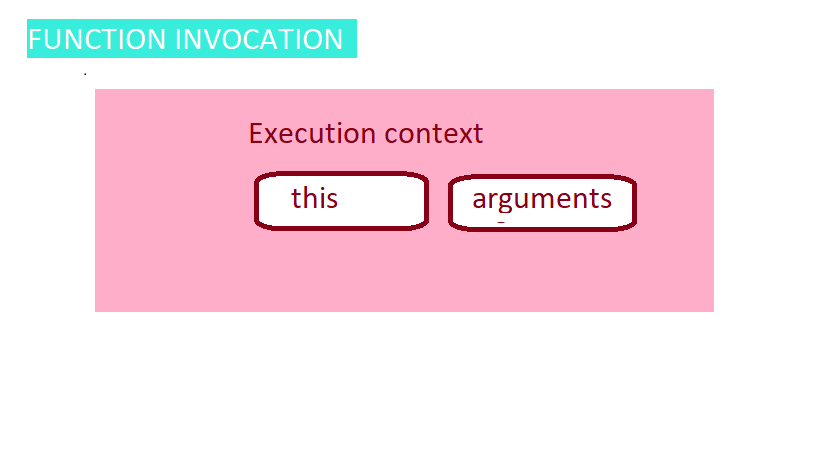
console.log(“Warm”);

}

**Function Invocation / Calling / Execution**

canada(); // canada function is defined at run time

india(); // india function gets defined at parse time or compiler looks to initialize the code or hoisting

****

function marry(person1,person2){

**console.log(arguments);**

    return `${person1} married ${person2} in a lovely ceremony`;

}

marry("Tina","Tim");

**output:**

Arguments { 0: "Tina", 1: "Tim", … }

"Tina married Tim in a lovely ceremony"

//arguments keyword gives object

// arguments keyword is only available after invocation

**Arguments Keyword:**

arguments keyword or special object can be used or accessed like an array . So to **loop through** arguments two other concepts can be used

1) **Array.from**method 2) **spread operator …**

1) function marry(person1,person2){

**console.log(Array.from(arguments));**

    return `${person1} married ${person2} in a lovely ceremony`;

}

2)function marry(...args){

**console.log(args);**

    return `${args[0]} married ${args[1]} in a lovely ceremony`;

}

marry("Tina","Tim");

// using these methods arguments keyword/object can be changed into array like objects

**Variable environment:**

A place where variables live in or information is stored. **Each execution context has its own variable Environment**

**Scope Chain:**

Links and gives access to variables in parent environment.

Function checks for its variable environment for any variable and if it’s not there it searches for the variable in the parent environment through scope chain.

Lexical or static scope - js compiler attaches scope chain before it runs

Scope - where can i access the variables / where is the variable

function sayMyName(){

  a='a';

  return function findMyName(){

      b='b';

      return function printMyName(){

          c='c';

          return "Autobiography of MSK";

      }

  }

}

**sayMyName()()();**o/p : Autobiography of MSK

Global scope : Variables declared outside the function is called global scope

Local scope : any scope local to a function .

// undefined is an actual type in js (variable is present) but reference error (undeclared variables) is not

Lexical Environment === [ [scope] ]

**JS weird:**

**1) ‘use strict’:**

'use strict'

function weird(){

  height =10;

   return height;

}

// height is not declared using var / let / const, so it will be looked for in global scope & it will be created

//’use strict’ does not allow this  and returns a reference error

**2)**var heyhey = function doodle(){

   return 'heyhey';

}

heyhey(); // returns heyhey

doodle(); // returns reference error since doodle() is not in this scope

//doodle can be accessed like this

var heyhey = function doodle(){

**doodle();**

   return 'heyhey';

}

**Function scope vs Block scope**

Scope -> what variables we have access to

Block scope -> **let** and **const**

**Global Variables :**

possibility of memory leaks

variable collision

**IIFE :**

global variables should be used minimally .

to avoid global variable issue , IIFE is used : **Immediately invoked Function Expression .**

**( function () { or ( function () {**

**var a =1;**

**}) (); } ());**

we can place all library code inside of local scope to avoid any namespace collision.

jquery and backbonejs - used this pattern

**var script1=(function(){**

**function a(){**

**return 5;**

**}**

**return {**

**a:a,**

**};**

**}());**

**function can be called using : script1.a();**

Script1 returns an obj

Still global namespace is used. But here only one var that is object that contains many properties that pollutes with only one global variable.

**this:**

**this is the object that the function is a property of**

initially this gets set to window obj

this in execution context acts as a placeholder - (who called me / which ever obj that called that method)

to the left of the dot (this?)

uses of this ,

**function a(){ function a(){**

**console.log(this); ‘use strict’**

**} console.log(this); }**

**//window // undefined**

**// gives methods access to their objects**

const obj={

  name:"Billy",

  sing(){

      return "Kannaana kannay kannaana kannay yen meedhu sayavaa : by "+this.name;

  },

  singAgain(){

      return this.sing()+" \n Movie : Visvasam \n";

  }

}

obj.sing();

obj.singAgain();

**// execute same code for multiple objects**

const name="Martin";

function importantPerson(){

  console.log(this.name);

}

const obj1={

  name:"Cassy",

  important: importantPerson,

}

const obj2={

  name:"Veronica",

  important:importantPerson,

}

importantPerson();

obj1.important();

obj2.important()

**Scope VS Dynamic Scope :**

In JS everything is lexically scoped except for this . **this - > dynamically scoped** ; It doesn’t matter where it is written , this always works based on **‘how it is called ‘**

const a= function(){

  console.log('a :',this);

  const b= function(){

      console.log('b :',this);

      const obj={

          c: function(){

              console.log('c :',this)

          }

      }

      obj.c();

  }

  b();

}

a();

o/p : // window

window

obj

const obj1={

  name:"Billy",

  sing(){

      console.log('a',this);

      const singAgain= function(){

          console.log('b',this);

      }

  singAgain();

  }

}

obj1.sing();

// o/p : obj

window

//dynamic scope

To solve this issue, two ways can be used

1. **Arrow function : 2) using ‘bind’**

**1) Arrow function:**

Arrow function (in ES6) is lexically bound , hence.

const obj1={

  name:"Billy",

  sing(){

      console.log('a',this);

      const singAgain= ()=>{

          console.log('b',this);

      }

  singAgain();

  }

}

obj1.sing();

**2) using ‘bind’**

const obj1={

  name:"Billy",

  sing(){

      console.log('a',this);

      const singAgain= function(){

          console.log('b',this);

      }

  return singAgain.bind(this);

  }

}

obj1.sing()();

**call() apply() and bind ():**

**call ():**

all the function has this call **functionnanme.call();**in short we call as **functionname();**

**function sample(){**

**console.log("hi");**

**}**

**sample.call(); // sample();**

**sample.apply() // both call and apply works in the same context here**

**call() and apply() : useful for borrowing methods from an object**

**bind () : is useful for us to call functions later on with a certain context or certain this keyword**

const wizard={

  name:'Billy',

  health:100,

  heal(){

      this.health=100;

  }

}

const archer={

  name:"Sicilly",

  health:30,

}

console.log('before',archer);

**wizard.heal.call(archer);**

**//wizard.heal.apply(archer);**

console.log('After',archer)

const wizard={

  name:'Billy',

  health:100,

  heal(num1,num2){

     this.health=100 +num1+num2;

  }

}

const archer={

  name:"Sicilly",

  health:30,

}

console.log('before',archer);

**wizard.heal.call(archer,50,30);**

**//wizard.heal.apply(archer,[50,30]);**

console.log('After',archer)

**bind() :** like call and apply it does not run a function it **returns a function**

const wizard = {

  name:"Billy",

  health:100,

  heal(){

      this.health = 100;

  },

}

const archer = {

   name: "Veronica",

   health:50,

}

const archerHealth = wizard.heal.bind(archer);

archerHealth();

console.log(archer);

Exercise:

How would you implement this:

*const array = [1,2,3];*

*function getMaxNumber(arr){*

*//code here*

*}*

*getMaxNumber(array) // should return 3*

(my solution)

const array = [1,2,3];

function getMaxNumber(arr){

arr.sort();

console.log(arr.pop());

}

getMaxNumber.call(window,array);

getMaxNumber.apply(window,[array])

exercise : Andrei solution

const array = [1,2,3];

// in this case, the 'this' keyword doesn't matter!

function getMaxNumber(arr){

return Math.max.apply(null, arr);

}

getMaxNumber(array)

**bind() and Currying :**

// function currying:

function multiply(a,b){

  return a\*b;

}

const multiplyByTwo = multiply.bind(window,2);

console.log(multiplyByTwo(4));

const multiplyByFour = multiply.bind(window,4);

console.log(multiplyByFour(5));

**//create partial parameter**

**context and scope:**

scope is visibility of the variable

context is most often determined by how a function is invoked with the value of this keyword