__/__/____

What is Javascript Engine (or) Ecmascript Engine

A JS Enigine is a software component that executes Javascript code.

All modern browsers has their own version of JS engines.

V8 → Chrome
Chakra → Edge (IE)
Spider monkey → Mozilla Geckodriver
JavascriptCore → Safari, webkit

* First JS engines was just interpreters. modern JS engines use just-in-time (JIT) for improved performance

Interpreter: It is a computer program that directly executes instructions of a programming (or) scripting language without being compiled into a machine language.

Just-In-Time (JIT): (also called as dynamic translation or run-time compilations) is a way of executing code that involves compilation of code during execution of a program (at run time

* European computer Manufacture association script (ECMAScript) is a Javascript standard

* Babel is a compiler that is used to convert ECMA 2015+ code into backward compatible (or) browser compatible

__/__/____

Careful with J.S.

* eval()
* arguments
* for in
* with
* delete
* hidden classes
* inline caching

Inline caching

```
function findUser(user) {
    return `found ${user.firstName} ${user.lastName}`
}

const userData = {
    firstName: 'sri',
    lastName: 'harsha'
}
```

call ⇒ `findUser(userData);`

The inline cache replaces above function call as 'found sri harsha' for optimization..

* It is an code optimization technique. The goal of inline caching is to speed up runtime method binding by remembering results of previous method lookup at the call site.

__/__/____

Hidden classes

* properties of an object should be in its constructor

* Creating properties for an object dynamically may not be able to use inline cache as it treats the object as two different class.

```
function Animal(x,y) {
    this.x = x;
    this.y = y;
}

const obj1 = new Animal(1,2);
const obj2 = new Animal(3,4);
// upto here the obj1, obj2 shares the same hidden class

obj1.a = 5;
obj2.b = 10;

obj2.b = 10;
obj2.a = 5;

// Now as a and b were added in opposite orders
obj1 & obj2 end up with different hidden classes
```

* Source : richardartoul.github.io/jekyll/update/2015/04/26/hidden-classes.ht

* deleting an object also changes the hidden classes and obj1 and obj2 points to different hidden classes

__/__/____

Web Assembly

WASM is a new type of code - with a small-sized fast binary format - for modern browsers.

* It is used for compiling any programming lang

* It is NOT an assembly language, it is not built for specific machine

* you don't write code in assembly, you use it to compile any given language

⇒ What it does is, "compile higher level lang and then run those web apps in the browser a lot faster than Javascript.

Memory heap and call stacks

* Memory heap is where memory allocation happens

* call stack keeps track of where our code is in its execution.

__/__/____

```
function subtractTwo(num) {
  return num - 2;
}

function calculate() {
  const sumTotal = 5 + 4;
  return subtractTwo(sumTotal);
}
debugger;
calculate();
```

Chrome consle > sources > snippets > new snippet
⇒ copy paste above code and observe call stack

* callStack stores functions & variable as code executes

* Each frame is pushed into stack during execution and works as FILO (first in, last out) (or) LIFO (last in, first out)

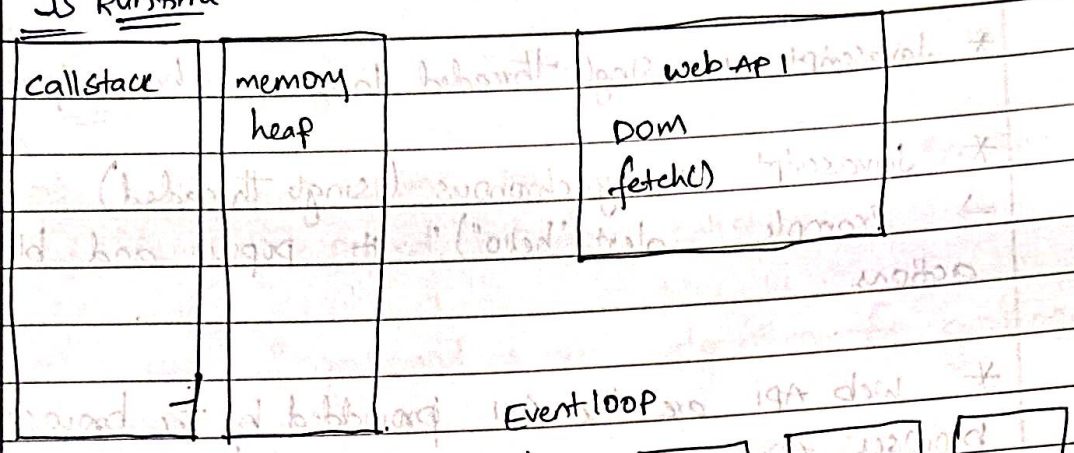
Causing a stack overflow

```
function inception() {
  inception()
}
inception()
```

⇒ throws exception
"uncaught RangeError: maximum call stack size exceeded."

__/__/____

JS Runtime



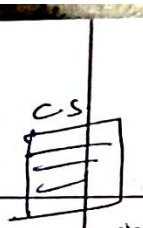
* JS Engine & JS Runtime (nodeJS) are different.

* node
  > window
    ⇒ error
  ⇒ global.window

* In nodes webAPI can be accessed by "global"

```
Const list = new Array(60000).join('1:1').split('.');

function removeFromList() {
    var item = list.pop();
    if (item) {
        setTimeout(removeFromList, 0);
    }
};
removeFromList();
```



Task queue → web API

microtask queue → then.

→ event loop.

__/__/____

* setTimeout (() ⇒ { console.log ("1"); } → Task Queue
}, 0);

promise.resolve ('hi').then(() ⇒ { console.log ('2'); } → microtask queue
});

console.log ("3"); → execution.

3
2
1.

MICROTASKS

* Even the promise is immediatly resolved, the code in lines .then/.catch/.finally still execute before these handlers.

Ex: let promise = Promise.resolve();
promise.then(() ⇒ alert("promise is done"); → 2nd

alert ("code finished"); ⇒ executed 1st

* because statements in then are pushed to queue

⇒ promise.resolve()
.then(() ⇒ alert ("promise done!");
.then (() ⇒ alert ("code finished");

__/__/____

Foundation 2

* The Execution context is a place where the Javascript code actually gets Executed

* In simple, callstack is collection of Execution contexts

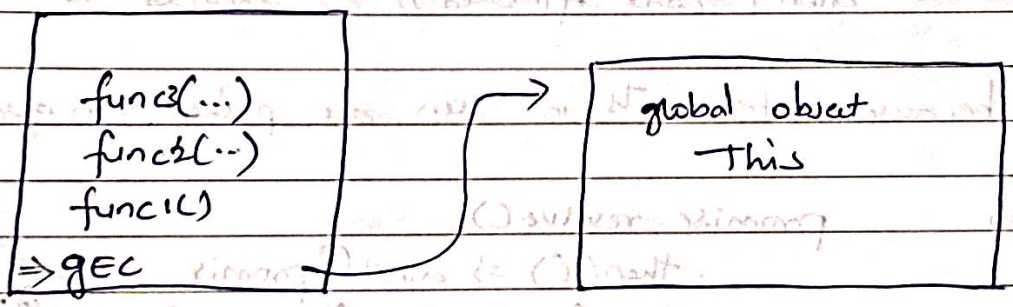
Two types of Execution context

→ GEC (Global Execution Context)
→ FEC (function Execution context)

GEC

* When a JS engine recieves a script file, it first creates a global Execution context by default

* The code that is not inside a function gets Executed in GEC



* GEC contains "global object" and "this"

FEC

* when a function is called, JS engine Creates a Execution context called FEC

__/__/____

A tricky code

```
getAge(99);
var getAge = function(year) {
  console.log(new Date().getFullYear() - year);
}
```

⇒ here we get error ⇒ getAge is not a function

⇒ because as "getAge" is declared as variable & not as a function as per hoisting rules.

⇒ Execution context has two phases.

1) Creation phase
2) Execution phase

Lexical Environment

* in JS lexical scope determines our available variables. Not where the function is called.

```
function print1() {
  console.log("hello");
}

function print2()
{
  print1() → this is accessible as print1() is in lexical scope / global scope
}
```

* first lexical env is the global lexical env

__/__/____

* A variable declared outside a function can be accessible inside another function defined after a variable declaration. called lexical scope.

* But the opp is not possible

* Closures & lexical scope are different topics

Hoisting

* In Creation phase in GEC ^ hoisting moves variables & functions.

* It allocates memory for 'var' & 'functions' keyword.

```
console.log(teddy)
console.log(sing())
var teddy = "bear";
function sing() {
    console.log('test');
}
```

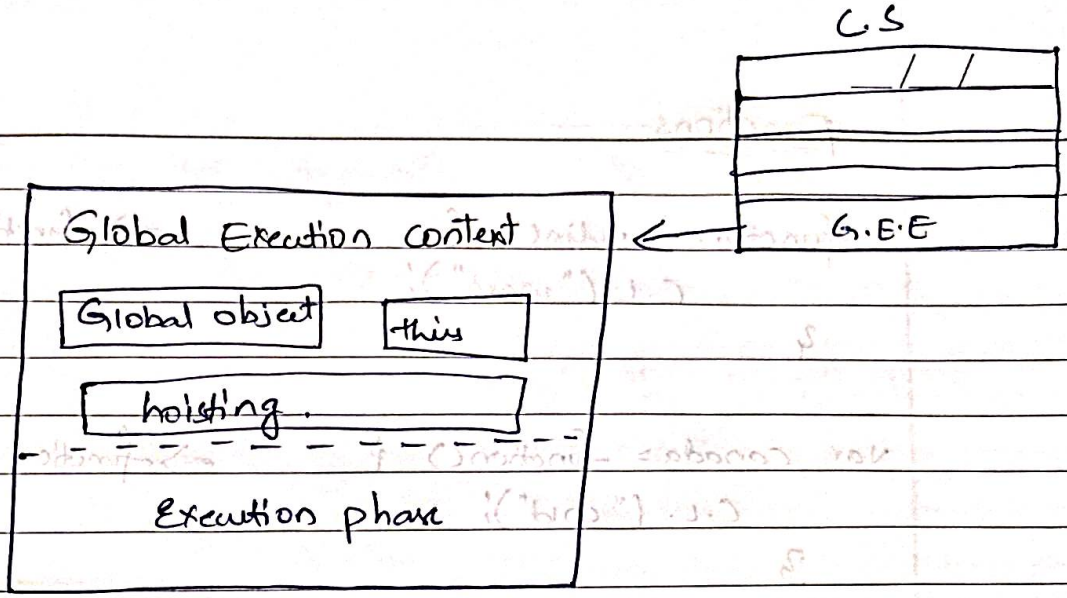
```
var teddy = undefined
function sing() {
    c.s.l('test');
}
c.l(teddy). c.l(sing());
teddy = "bear";
```

It works because the code becomes

Ex2

```
console.log(teddy);
console.log(sing());
var teddy = 'bear';
(function sing(){
    console.log('test');
})
```

It throws error as JS sees '(' instead of function and it wont hoist.



Exercise

*
```
a()

function a() {
    c.l("hi");
}
function a() {
    c.l("bye");
}
```

*
```
function a()
{ c.l("hi"); }

a()

function a() {
    c.l("bye");
}
```

*
```
var favFood = "grapes";
var foodThoughts = function() {
    c.l("Original fav food" + favFood);
    var favFood = "sushi";
    c.l("New food :" + favFood);
}

foodThoughts();
```

function lexical scope hoisting ha

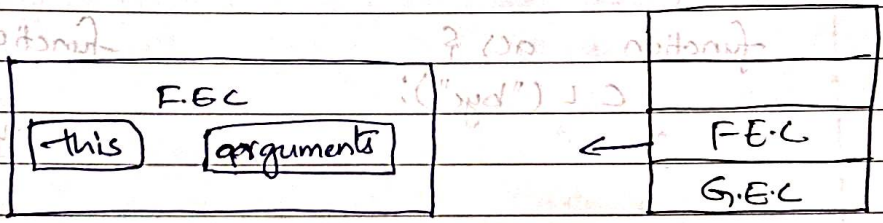
__/__/____

Functions

```
function india() {                    → function declaration
    C.L.("warn");
}

var canada = function() {             → function expression
    C.L ("cold");
}
```

* function declaration only gets hoisted not function expression.
* Calling a function india(), canada().
* For Each function Execution context we get this & arguments.



Ex

```
function marry (p1, p2) {
    console.log(arguments);
    console.log("p1" "marrying" +p2);
}
```

* arguments are only avaible in F.E.C (Not available in global execution context)

__/__/____

* Best way to use arguments

=> Array.from(arguments)

=> function m2(...args) { => rest operator
Console.log("arguments", args);
}

m2("tina", "tim");

* Spread operator vs rest (...)

Ex let arr = [...iteratable]; → spread

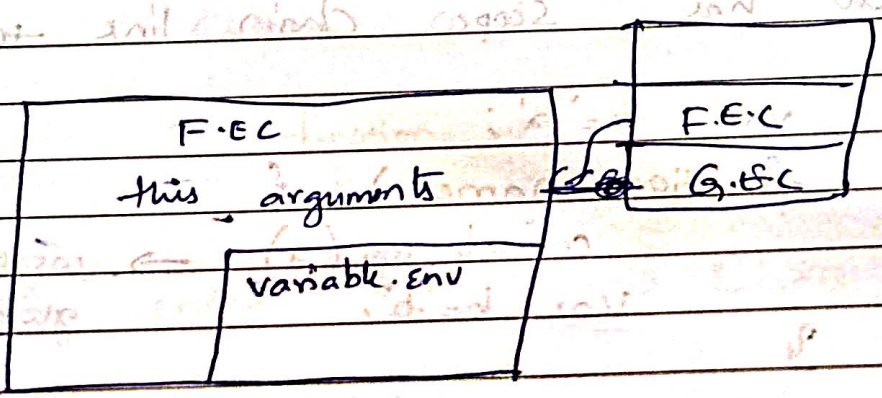
function m2(...args) → rest.

* Spread is used to Expand (one to many)
* rest is used to compress (many to one).

* arguments will be {} in F.E.C Even no parmas passed.

Variable Env

* F.E.C also will have place or storing variables



* Each G.E.C. has its own variable Env

```
function two(){
    var isvalid;
}

function one(){
    var isvalid=true;
    two();
}

var isvalid=false;
one();
```

// two() → undefined
// one() → true
// G.E.C() → false

Scope chain

* All functions will have access to global scope

* Every F.E.C has its own variable Env but it also has scope chain link to global env

```
var x='x';
function name(){
    console.log(x);   → works as x is in global scope
    var b='b';
}
```

__/__/____

* in J.S our lexical scope is place where function is defined and data available. Not where the function is called

*
```
function sayMyname() {
    var a='a';
    return function findName() {
        var b='b';
        console.log(c);
        return function printName() {
            var c='c';
            return "harsha"
        }
    }
}
sayMyname()()();
```

⇒ in above findName lexical Env is sayMyname. and printname lexical env is findName.

* undefind : undefined in JS states that the variable is available but doesn't have any value

Ex Chrome console

```
function a() {
}
```

→ window → a() → [[Scope]] → Type = Global

Exercise

1)

```
function weird()
    height = 50;
    return height;
}

weird()  =>  50
```

strict ex

```
b = 10
console.log(b)  valid if no strict
```

⇒ if strict; b is undefined.

* If not declared it autocreats in global scope
* It is a Global leackge
* If 'use strict' is used it dont create global var.

2)

```
var hey = function doodle() {
    console.log("hey");
}

hey()  =>  hey

doodle()  =>  reference error.
```

* Function Scope vs block Scope

```
if (4>3) {
    var secret = 12345;
}

secret;  →  12345
```

```
if (4>3) {
    let (or) const secret = 12345
}

secret;  →  error
```

let & const only works in block Scope

__/__/____

* Why dont we declare all variables in global.?

⇒ it will cause variable collision.

IIFE (immediatly invoked function Expressions)

* (function () {
  ↳ function Expression
  })();

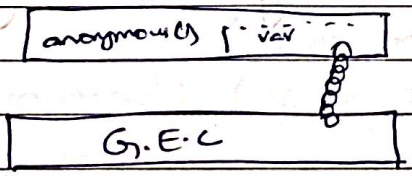
⇒

```
(function () {
  var a="10";
})();
```

a → is undefined

⇒ IIFE is used to place code in local scope to avoid names collision.

IIFE call stack



__/__/__

This

⇒ this is an object that the function is property of

* in G.E.C, this refers to window object

Try

```
console.log(this);

function a() {
    console.log(this);
}
```

Example

```
const obj = {
    name: "billy",
    sing: function() {
        console.log("hello sing" + this.name");
    }
}
```

* The thumb rule is if we use "this" in a function, that function should be invoked as a method.

uses

1) gives access to ~~higher~~ their objects
2) Execute same code for multiple objects.

Example

```
function importantPerson() {
    console.log("hello" + this.name);
}

const name = "harsha";

const obj1 = {
    name: "sri",
    importantPerson: importantPerson
}

importantPerson();
obj1.importantPerson();
```

* what is .npmrc

⇒ It is a configuration file for npm, it defines settings on how npm command should behave when using commands. (rc → runtime configuration).

⇒ cache = <path to cache>
⇒ registry = <path to registry.npmjs.org"
⇒ loglevel = silent / notice
⇒ logs-dir

* npm-shrinkwrap.json similar to package-lock.json, not recommend, the only use case is it is bundled while publishing a package

__/__/____

* npm pack creates a tgz file exactly the way it would if we are going to publish package.

* Package-lock.json.

* Generates a single version of dependency tree.
* It also optimizes the installation by allowing npm to skip repeated metadata for packages.

* lock file version 1 if npm is 5 (or) 6
* lock file version 2 if npm >= 7.

* npm ci

* Similar to npm install, used to clean install of dependencies

* npm ci is faster when

  * lock file or npm-shrinkwrap.json available
  * node_modules are missing (or) empty.

* If node_modules are already present npm ci removes it before install.

* The project must have lock file (or) shrinkwrap.json

__/__/____

* Avoid sending/calling methods in callback function that may throw error due to invalid this.

⇒
```
let person = {
    name: "harsha",
    print: function() {
        console.log(`hello ${this.name}`);
    }
}
```

✗ setTimeout(person.print, 2000); → Hello undefined
↳ in here this refers to timout function.

correct way →
```
setTimout(function() {
    person.print()
}, 2000);
```

* Call & Apply

:) Allows the ability to modify this object

```
function sqnum(a,b) {
    return Math.pow(a,2) + Math.pow(b,2);
}
```

```
> sqnum(2,3);
> sqnum.call(null, 2,3);
> squm.apply(null, [2,3]);
```

Function Constructor

```
function Cart() {
    this.store = " Indian Grocery";
    this.items = [];
}

let cart = new Cart();
    ↓
new object

cart.store;
cart.items.push('milk');
cart.items.
```

⇒ Calling function without new may lead to leak of variables in global.

```
let cart1 = Cart();
```

cart1 → undefined

store → Indian Grocery.

items → [].

__/__/____

Error handling JS

1) try catch

2) using callbacks

ex: fs.readfile ("path", (err, result) => {

if(err) c.l.(err)

})

3) using promises, as promises reject the with error

Ex promise().then( ).catch(err)

4) using async/await

try{ async code with await } catch (err) {

5) Event emitters

ws.on ("error", (er) => {

c.l("error")

});