

* **High-level** – resursele nu trebuie administrate manual. De ex, in C, pentru a crea un obiect, gen un array, e necesar de a aloca memorie manual, de aia C e low level, dar JS e high level, caci el administreaza automat memoria
* **Interpreted or just in time compiled** – calculatorul intelege totul doar prin 0 si 1, dar noi nu scriem asa instructiunile. JS folsoeste codul nostru pentru a crea codul masina, proces numit compilare
* **First**-**class functions** - functiile sunt tratate ca variabile, si pot fi trimise ca parametrii
* **Dynamic** – variabilele nu necesita definirea la data type, JS determina tipul.
* **Single threaded** – ruleaza intr-un singur thread doar
* **non-blocking event loop** – e adevarat ca JS are un singur main thread, dar asta nu inseamna a totul se face intr-un singur thread doar. Environment in care JS e rulata, ca Node.js, poate sa mai creeze threaduri, la nevoie. Daca de ex, facem un http request sau un timer, JS nu va sta sa astepte aceste operatii ca sa mearga mai departe, ci se va crea un thread separat pentru ele, si main thread nu va sta sa astepte ca ele sa termine, nu va avea nimic cu ele. Deci, main thread nu e blocat!

**De ex**:

console.log("Start");

setTimeout(() => {

console.log("Timer callback");

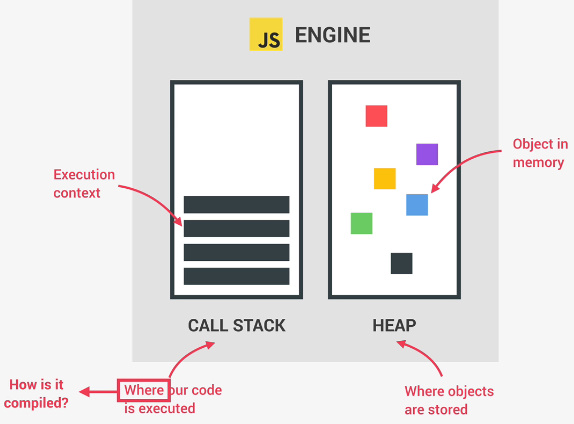
}, 1000);

console.log("End");

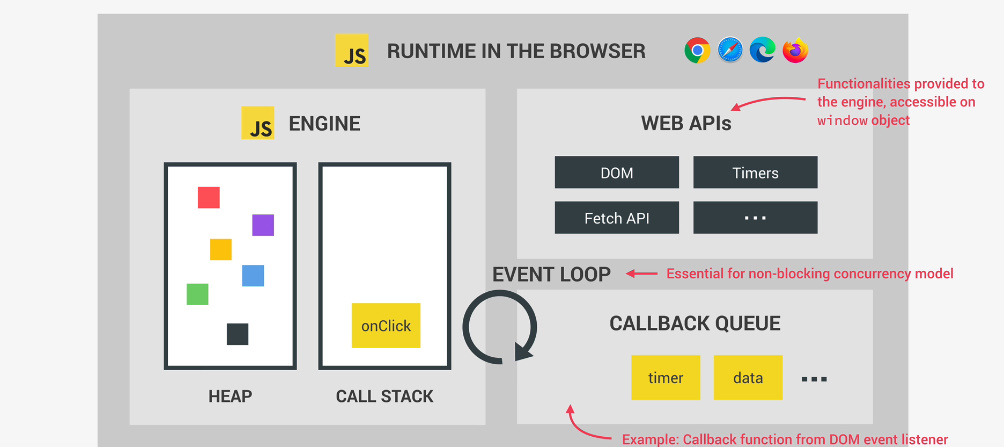
In this code, **setTimeout** schedules the callback function to be executed after 1 second, but JavaScript doesn't pause and wait for that time to pass. It immediately moves on to the "End" log statement, demonstrating its non-blocking behavior.

**JS Engine and Runtime**

* **JS Engine** – program care executa JS code
* Fiecare browser are un JS Engine, de ex Node.js
* Fiecare JS Engine are un **call** **stack** si **heap**



* **Runtime** – instructiunile executate la rularea programului



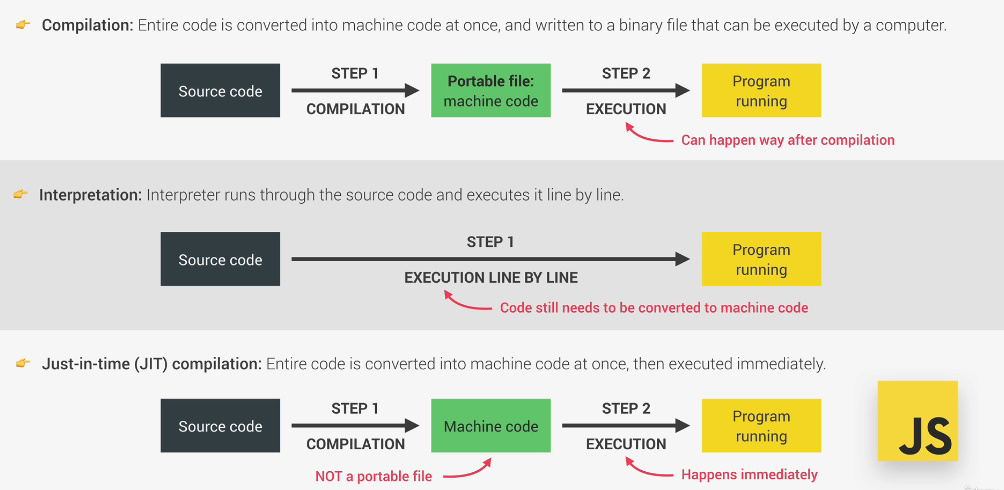
**- JS Engine** acceseaza WEB API, adica ale browser, ca de ex time, console etc.

- **Callback queue** pastreaza eventurile ce au loc. De ex, daca s-a facut un click si s-a acivat un timer si apoi un hover, callback queue va fu asa:

click, timer, hover, dar ele inca urmeaza sa fie executate in stack, cand el e liber, una cate una. In callback queue ele doar sunt pastrate si trimise la queue ca FIFO

- **Event loop** – el e cel care ia event din callback queue si il pune in stack

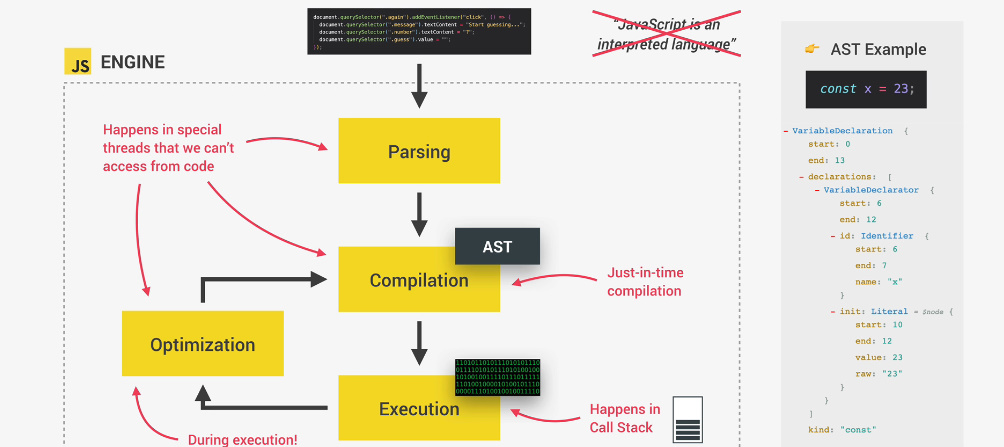
**Interpretation vs Compilation vs Just in time Compilation**



JS era initial Interpretat, dar limbajele interpretate sunt foarte lente, de aceea cum e Just-in-time compilation, adica ca un fel de Compilation + Interpretation

* **Compiled Language** – programul este in intregime compilat, inainte de a rula, adica tot codul e transformat in cod masina(in fisier .exec) si masina il executa. Daca facem vreo schimbare in el, codul din nou trebuie compilat in intregime si iar executat de masina. Deci, codul nostru intreg e transformat in alt cod, in limbaj assembly si apoi masina il executa. IDE are un compilator in el. Problema e ca codul nostru e compilat si optimizat, pe baza la os si hardware pe care il avem, ca sa poata fi executat de masina noastra doar. Deci, fisierul executabil, cu cod compilat, va rula pe masina noastra, dar nu si pe altele si asta e problema. Fisierul executabil e creat la compile time. Compilarea dureaza timp, in schimb programul apoi va rula mai rapid.
* **Interpreted language** – programul nu este compilat, deci nu este transformat direct in cod masina, si nu masina il ruleaza direct, ci codul este citit si rulat linie cu linie de un alt program,numit interpreter, ca JVM, de ex, sau alta masina virtuala. Deja interpretatorul e responsabil de a transforma codul in cod masina si masina apoi deja il executa. Avantajul e ca codul poate fi rulat asa pe diferite pc si deci os, insa problema e ca un interpreted language e mai lent ca unul compiled, asa cum codul e transformat in cod masina la runtime, si daca gen avem un for loop, codul se va transforma iar si iar pentru fiecare iteratie. Interpreter nu face optimizari, ruleaza codul asa cum il vede,c sa poaat fi rulat pe orice masina, si asta poate fi mai lent.
* **Just-in-time compilation language** – just in time compilation este o tehnica pentru a mari viteza la interpreted languages. Tot foloseste interpreter. El gaseste partile de code repetabile, ca metode, loop etc. si le transforma la runtime direct in code machine si deci nu le va recrea iar si iar, ci le va compila odata la runtime si le va refolosi cand va fi nevoie. Asa, codul ruleaza repede, nu trebuie compilat la startare de fiecare data in intregime si nu e asa lent ca rularea linie cu linie cu un interpreter.

**Just in time compilation in JS**



1. Intai, codul e citit si aranjat intr-un **AST(abstract sintax tree),** adica fiecare linei de cod e citita si se colecteaza date despre ea, necesare limbajului. La partea asta si se determina daca sunt erori de sintaxa
2. Codul e compilat
3. Codul machine obtinut e optimizat
4. Executia

Toate aceste operatii au loc in difertie threaduri, separate de main thread

**Execution contexts and call stack**

* **Execution context** – environment in care JS code e executat. El pastreaza toate informatiile despre codul de executat in el(variabile, function etc.)
* **Context** se refera la obiectul curent la care JS se refera, si deci e accesat cu this keyword. Si functiile sunt obiecte in JS!
* Intr-un execution context se gasesc:

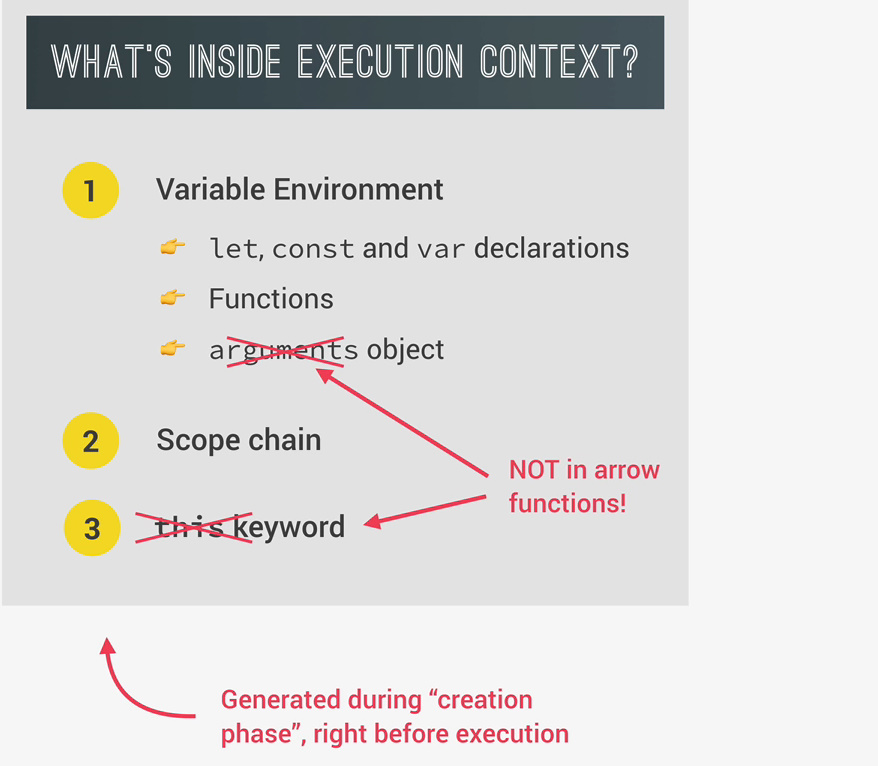
- variabile

- declaratii de functii

- arguments object

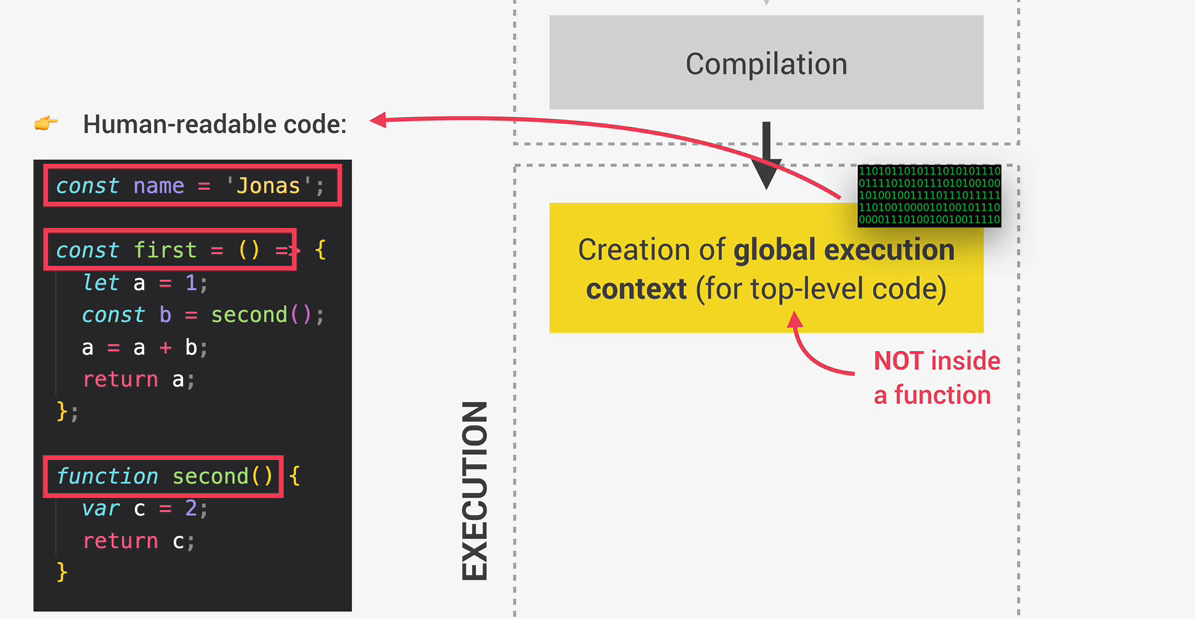
- scope chain

- this keyword

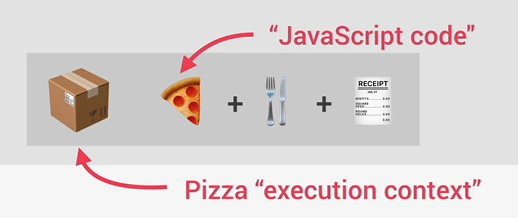
* 

Deci, arrow functions nu au this si nici arguments object

* **In JS mereu exista doar un Global execution context, unde se pastreaza tot codul ce nu face parte din functii**
* Cand codul incepe sa fie executat, se creaza **un global execution context**, unde se gaseste totul codul ce nu este in interiorul niciunei functii, deci e si logic ca se executa tot ce nu este in functie, caci functiile nu sunt executate daca nu sunt apelate.

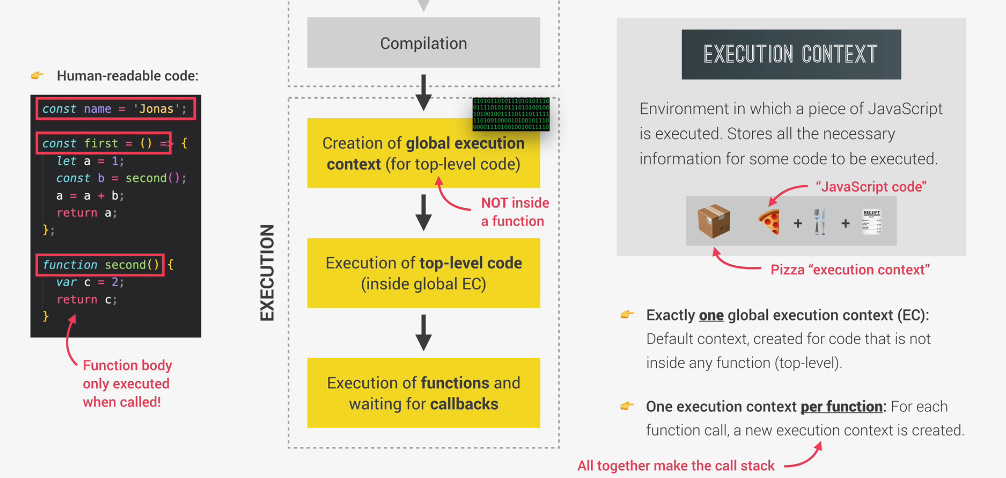


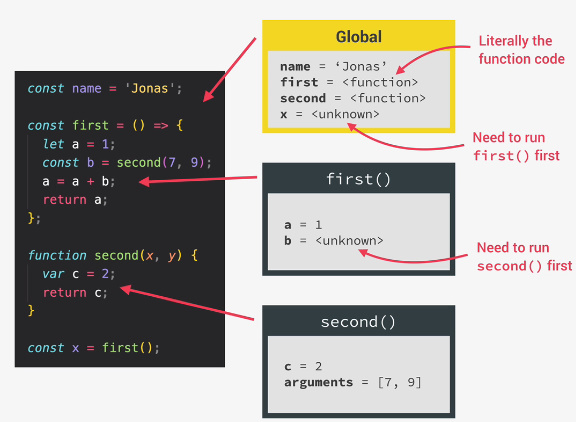
deci, name, first si second fac parte din global context, insa evident ca ceea ce e in functii nu face parte din global exectuin context

* 

De ex, vrem sa mancam o pizza, O primit intr-o cutie cu tot cu ceea ce ne trebuie ca sa o mancam. Cutia e ca “execution context” sis pizza e code

* **Odata ce o functie e apelata, se creaza un nou execution context, continand orice informatie despre codul din functiei**
* Toate aceste execution contexts creaza **call stack**

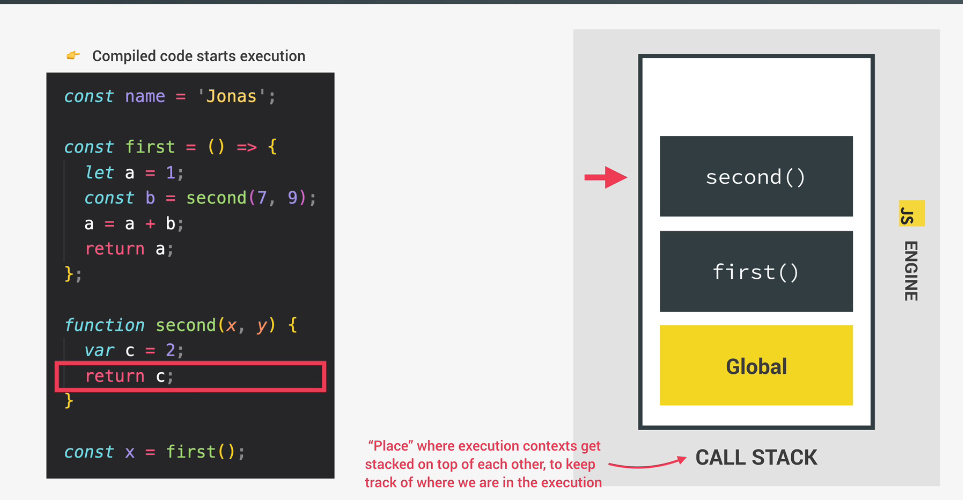
****

* ****

**Valorile pot fi stiute doar la rulare! Nu la definirea contextului**

**Call Stack**

* **Call stack –** loc in care execution contexts sunt puse in ordinea LIFO, pentru a sti care si cum sa fie executat
* **De ex:**

****

Deci, jos in stack call se gaseste global context. Se declara name, apoi ajunge la x, care apeleaza functia first, deci ea e urmatoarea in stack si in ea se apeleaza second, si ea e urmatoarea in call stack. Odata ce second termina, ea e scoasa din stack si continuta first si tot asa.

Cu call stack, JS stie unde si cum sa execute functiile

* **Global stack insa nu se inchide, doar daca inchidem pagina!!!Atunci codul si termina executia**

**Scope and Scope chain**

* Fiecare **context** are un **scope chain**
* **Scope** – zona in care variabilele si functiile sunt declarate si acesibile. Scope se refera doar la vizibilitatea variabilelor. Un scope e creat mereu cand intalneste { }
* Tipuri de scopes:

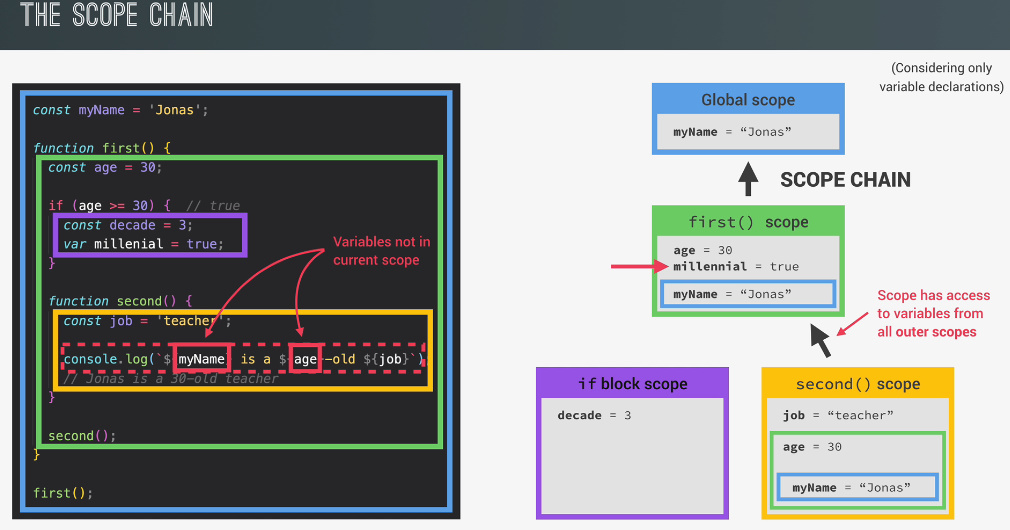
- global

- function

- block



* **Scope of variable** – regiuen din cod in care o anumita variabila poate fi accesata
* **Var** – este function scoped!!!

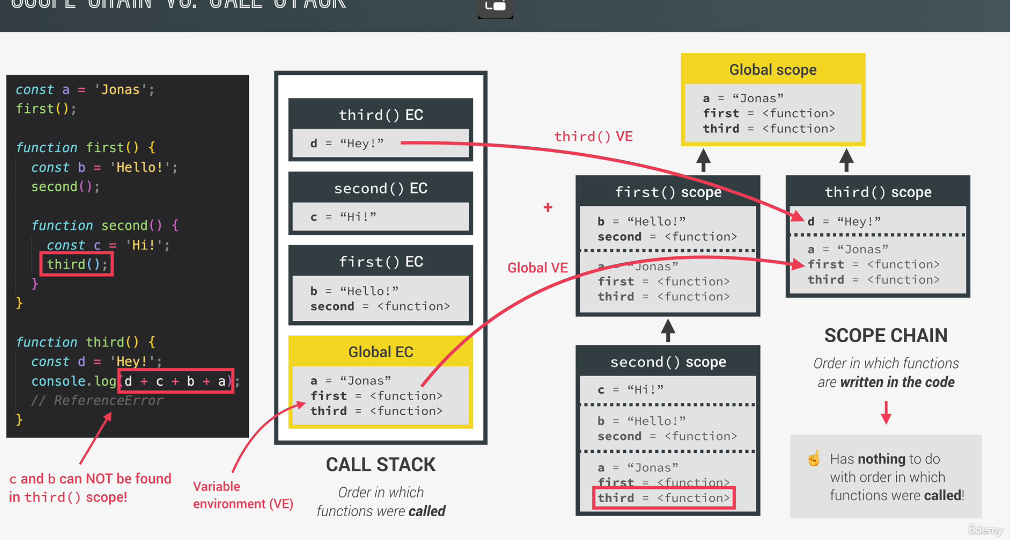


Un scope are acces la varabilele si functiile din scope parinte, deci orice variabila intai e cautata in scope curent, si daca nu e gasita se duce sa caute la scop parinte, si apoi la urmatorul parinte si tot asa pana global scope

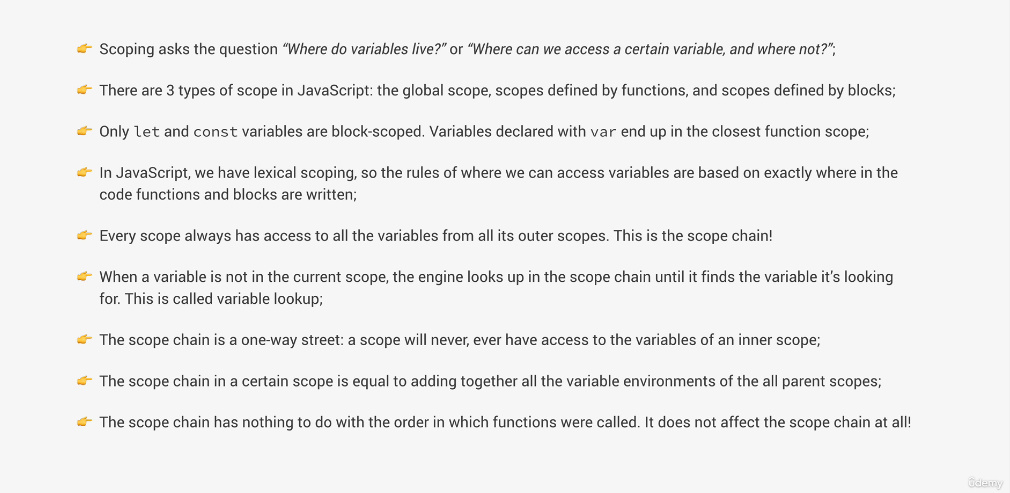
* **Scope chain** – ierarhie de scopuri, de ex un scop ce are o referinta la un scop parinte si acel parinte la alt parinte si tot asa pana la global scope

**Scope Vs context**

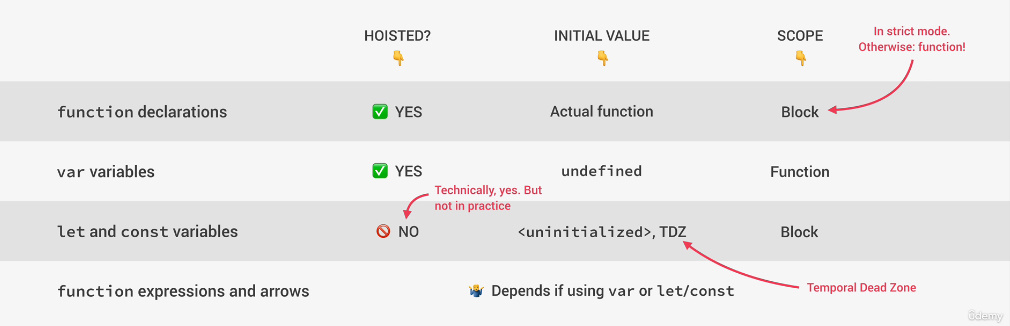
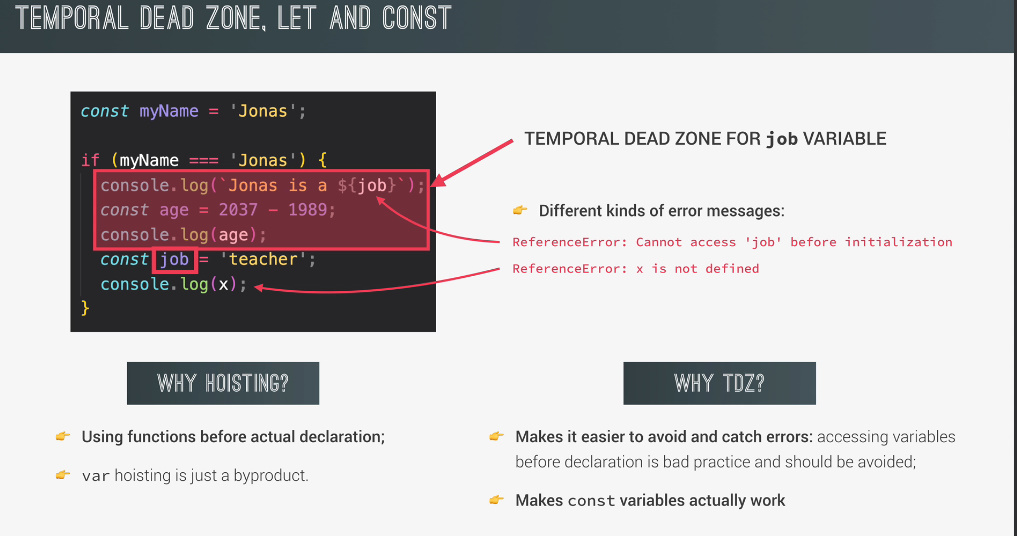
* **Deci, context e obiectul curent accesat, scope e zona in care anumite variabile si functii sunt accesibile**
* Scope este legat de context al sau. El acceseaza var si func din context al sau
* **Scope face parte din context**
* Ordinea de executie a context nu afecteaza nicidecum modul in care scope au acces unele la altele!



third function, desi a fost apelata din second(), care se gaseste in first(), nu poate accesa c si b din first si second scope, deoarece scop a functiei third nu are nimic de a face cu scope a lui first si second, si deci nu are referinta nici la scope a lui second, nici a lui first.



**Hoisting**

* **Hoisting** – proces de a face accesibile variabilele inainte de declararea lor
* JS scaneaza mai intai de toate codul, pentru a cauta declaratiile de variabile si functii.
* 
* 
* const nu ar putea fi pus ca undefined si apoi setat ca valoare.
* Daca am incerca sa accesam o functie, declarata si stocata intr-o variabila var, inainte de definirea ei, am primi ca variabila nu e o functie, deoarece s-ar apela undefined()

func();  
  
var func = function() { }



TypeError: func is not a function

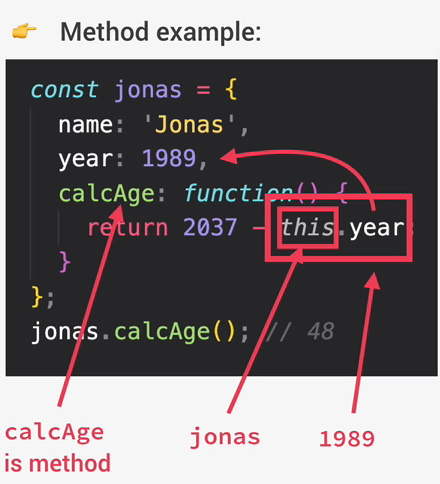
* Putem folosi doar o function declaration inainte de initializarea sa:

func();  
  
function func() { }

**this**

* **this** – variabila creata in fiecare context pentru a pastra referinta la obiectul curent
* Cand folosim this in functie, mereu va fi accesat obiectul din care e apelata functia
* **this niciodata nu pointeaza catre functia insasi!Daca functia nu e apelata de un obiect, this mereu va fi window sau undefined cu strict mode**

1. **Folosim this intr-o metoda:**



**this nu pointeaza catre obiect din cauza ca functia e definita in obiect, ci din cauza ca obiectul apeleaza functia. Deci, cand functia e apelata in orice obiect, orice functie mereu va avea this ca referinta la obiect, niciodata la functia insasi!!**

const *obj1* = {  
 name: 'Eduard',  
 func: function() { *console*.log(this.name) }  
}  
  
const *obj2* = {  
 name: 'Jora',  
}  
  
*obj2*.func = *obj1*.func;  
  
*obj2*.func();

Jora

Deci, nu s-a returnat ‘Eduard’

const *obj1* = {  
 name: 'Eduard',  
 func: function() { *console*.log(this.name) }  
}  
  
  
func = *obj1*.func;  
  
func();

undefined

1. **Folosim this intr-o functie sau global scope :**

In non strict mode, face referinta la window, caci asta e obiectul de baza

In strict mode, e undifined

*console*.log(this);

Window

function func(){  
 *console*.log(this);  
}  
  
func();

Window

Daca folosim ‘use strict’

'use strict'  
function func(){  
 *console*.log(this);  
}  
  
func();

undefined

1. **Folosim this intr-o arrow function**

Va retura this a obiectului in care s-a apelat metoda parinte, sau window daca nu e apelat intr-o metoda

const func = () => {  
 *console*.log(this);  
}  
  
func();

window

sau undefined cu strict mode

1. **This in event listener**

this este DOM element la care handler este atasat

**Regular function vs arrow function**

* In arrow function, this nu are nimic de a face cu obiectul ce a apelat metoda! Asa e la arrow function, this mereu se refera la global context, asa cum se socoate ca arrow function nu apartin decat la global scope,indiferent unde sunt definite
* const *obj1* = {  
   name: 'Eduard',  
   func: () => { *console*.log(this.name) }  
  }  
    
  *obj1*.func();

undefined

Dar, asa va merge

var *name* = 'wrong name';  
  
const *obj1* = {  
 name: 'Eduard',  
 func: () => { *console*.log(this.name) }  
}  
  
*obj1*.func();

wrong name

this se refera la window object, unde se si gaseste

var *name* = 'wrong name';

**Nu folosim niciodata arrow function ca metode!!!**

* Daca apelam o functie din interiorul alteti metode, deci metoda e in obiect, this pentru functie va fi undefined, desi in metoda el exista. Unii considera asta un bug, dar se spune ca asa trebuie sa lucrez this, indiferent de unde e apelat si cum, daca nu e apelat de un obiect direct,nu va apea this si gata.**Asta se intampla doar cu strict mode!**

'use strict'  
  
const *person* = {  
 name: 'Eduard',  
 getName: function() {  
 function test(){  
 *console*.log(this);  
 }  
 test();  
 }  
}  
  
*person*.getName();

undefined

Dar, dafa ‘use strict’, this va exista

* O solutie la problema de sus este de a crea in metoda o variabila ce sa aiba referinta la this si ea sa fie folosita in functie

'use strict'  
  
const *person* = {  
 name: 'Eduard',  
 getName: function() {  
 const self = this;  
 function test(){  
 *console*.log(self);  
 }  
 test();  
 }  
}  
  
*person*.getName();

* Asa ceva va merge, caci se primeste ca obiectul apeleaza alta metoda din metoda, cu this.test()

'use strict'  
  
const *person* = {  
 name: 'Eduard',  
 getName: function() {  
 this.test();  
 },  
 test: function (){  
 *console*.log(this);  
 }  
}  
  
*person*.getName();

* Arrow function nu are acces la this, cu exceptia cazului cand e apelata dintr-o metoda:
* const *person* = {  
   name: 'Eduard',  
   getName: function() {  
   const test = () => {  
   *console*.log(this);  
   }  
   test();  
   }  
  }  
    
  *person*.getName();

object

const *person* = {  
 name: 'Eduard',  
 getName: () => {  
 *console*.log(this);  
 }  
}  
  
*person*.getName();

undefined

Arrow function nu au acces direct la this al obiectului, dar il pot lua din scope al metodei.

**arguments keyword**

* orice argument trimis functiei e stocat si intr-un obiect arguments, ce orice functie il are, cu exceptia la arrow!
* Asa, putem trimite chiar si mai multe argumente decat parametri

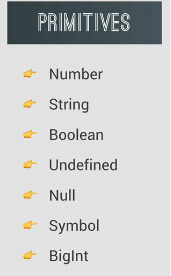
function test(a){  
 *console*.log(arguments);  
}  
  
test(10,20,30);

[Arguments] { '0': 10, '1': 20, '2': 30 }

arguments e un array, si accesam argumentele cu []

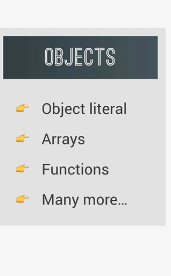
**Primite vs objects(reference)**

* Datele primitive sunt trimise prin copiere!
* Obiectele prin referinta
* **Primitive sunt:**

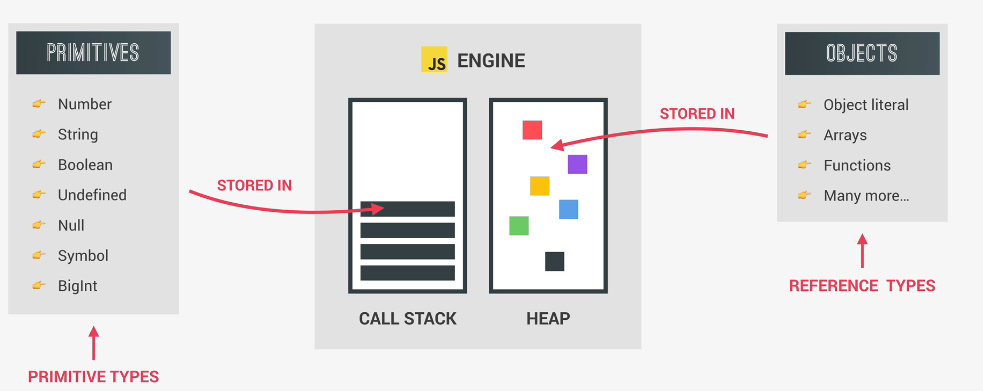


Se mai numesc primitive type

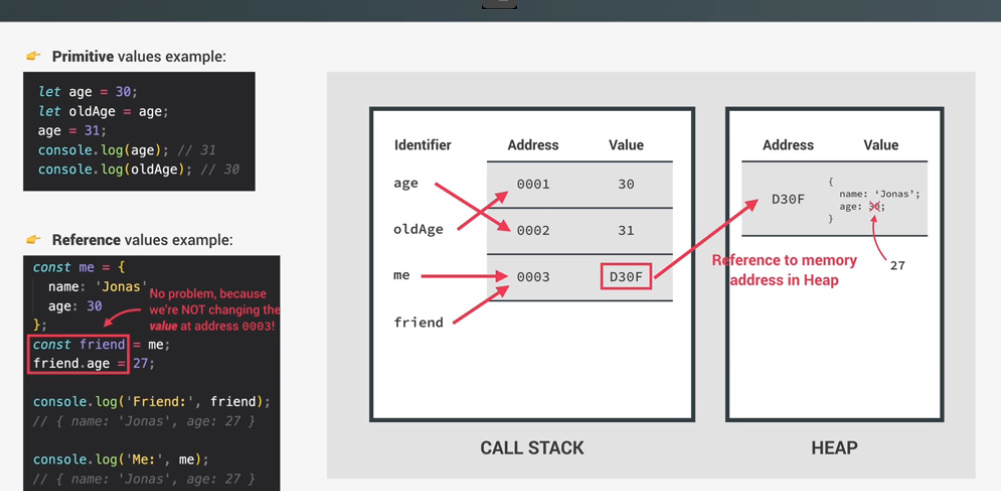
* **Iar obiecte sunt:**



Li se mai spune reference type



primitivele sunt stocate direct in context din call stack



Atentie! In mod normal, fiecare variabila are adresa ei de memorie, indiferent ca e variabila de date primitive sau referinta. Totusi, engine poate face, daca e absolut sigur

**Cum facem o copie a unui obiect**

Obj.assign(obj1, obj2) – creaza un nou obiect ce contine fieldurile la obj1 + obj2

const *obj1* = {  
 field1: 'field1'  
};  
const *obj2* = {  
 field2: 'field2'  
};  
  
*console*.log(*Object*.assign(*obj1*, *obj2*));

{ field1: 'field1', field2: 'field2' }

Dar, putem liber face o copie a lui obj1 asa:

*console*.log(*Object*.assign({ }, *obj1*));

* Problema e ca assign nu face deep copy. Daca gen ca field e un array sau alt obiect, el pur si simplu va fi copiat,ca referinta si gata.