**default parameters**

* JS suporta si default parameters values pentru functions
* Putem chiar folosi valoarea altui parametru dinaintea acestuia:

function test(a,b,c = a+b){  
 *console*.log(a,b,c);  
}  
  
test(10,20);

**About functions**

* Functions sunt la fel un tip de obiect
* Functiile sunt tratate ca valori si ele
* De asta, functiile pot fi stocate in variabile, caci odata ce sutn obiecte, putem avea referinte la ele
* Putem trimite functii ca parametri, sa returnam functii
* Orice functie are un field name, care e numele ei:
* function test(){  
     
  }  
    
  *console*.log(test.name)
* O functie returnata poate liber accesa parametrii functiei de unde a fost ea returnata:
* function greet(expression){  
   return function(name) {  
   *console*.log(`${expression} ${name}`)  
   }  
  }  
    
  greet('Hello')('Eduard')

Asta e multumita la closures

Putem si cu arrow function:

const greet = expression => name => *console*.log(`${expression} ${name}`)

**call, apply, bind**

* this functioneaza doar daca o functie este metoda, adica e apelata direct de un obiect, de altfel, this va fi window sau undefined, cu strict mode

Avem un obiect airline1, si acum vrem sa cream inca un obiect, identic, insa iar ar trebui sa dam copy paste la metoda getInfo.

* O solutie ar fi sa facem asta:
* const *airline1* = {  
   name: 'Fly',  
   clients: 500,  
   getInfo() {  
   *console*.log(`Company ${this.name} has ${this.clients} clients`)  
   }  
  }  
    
  const *airline2* = {  
   name: 'MDA',  
   clients: 150,  
   getInfo: *airline1*.getInfo  
  }  
  *console*.log(*airline2.getInfo()*)
* Insa, daca apelam functia nu din obiect, this va fi window
* const func = *airline1*.getInfo;  
    
  func()

Company undefined has undefined clients

* **call(obj, param1, param2,...)**

functia este un obiect, deci are si ea functii. Metoda call() a functiei ia ca prim argument obiectul ce va fi this, si restul argumentelor ce trebuie trimsie functiei

const func = *airline1*.getInfo;  
  
func.call(*airline2*)

this va fi airline2, si daca ar mai fi argumente la func, le-am pune dupa ariline2

De ex func.call(airline2, param1, param2, ...)

* **apply(obj, [param1, param2, ...])** – exact ca call, doar ca argumentele trebuie puse in array
* **bind(obj)** – nu apeleaza functia! Creaza o noua functie, cu, si o returneaza, si va trebui apelata cu argumentele trimise de noi deja.

const *airline1* = {  
 name: 'Fly',  
 clients: 500,  
 getInfo(year) {  
 *console*.log(`Company ${this.name} has ${this.clients} clients in ${year}`)  
 }  
}  
  
const *airline2* = {  
 name: 'MDA',  
 clients: 150,  
}  
const *func* = *airline1*.getInfo.bind(*airline2*);  
  
*func*(2023)

Putem sa si folosim bind(obj, param1, param2..)

Ideea e ca daca de ex metoda are 2 parametri, si noi folosim bind(obj, param1), cand vom apela metoda returnata, vom oferi doar un parametru, asa cum unul deja a fost setat in noua functie, chiar setat, nu doar pusa valoare default, el nici nu va mai exista ca parametru in noua functie.

const countTax = function(tax, value) {  
 *console*.log(value + value\*tax);  
}  
  
countTax.bind(null,0.23)(2500);

punem null pentru this, caci oricum nu folosim this nicaieri

**Problema cu this la eventListener**

* eventListener mereu modifica this de la metoda trimisa de noi sa pointeze catre event
* const *planes* = {  
   numbers: 200,  
   buyPlane(){  
   this.numbers++;  
   *console*.log(this.numbers);  
   }  
  }  
    
  *document*.querySelector(".buy").addEventListener('click',*planes*.buyPlane);

Apasand be butonul cu clasa .buy, ar trebui sa primim

201. dar primim

undefined

Asta e din cauza ca eventListener a avut grija sa modifice this a metodei trimise de noi, cu bind(), si iata de ce, this nu mai e planes object

* Solutia este de a seta this cu bind(),asa se va returna o noua functie, si asa this nu va mai putea fi setat din nou de listener
* *document*.querySelector(".buy").addEventListener('click',*planes*.buyPlane.bind(*planes*));

**Imediately Invoked function expressions(IIFE)**

* IIFE este o functie executata la momentul declararii si care nu mai poate fi executata mai mult
* Asta o facem asa (function() { })()

(function() { *console*.log("Hello")})()

Si row function

(() => { *console*.log("Hello")})()

* Asta poate fi util pentru a defini un scope izolat de main

**Closures**

* **Closure** ofera acces unei functii la scope a functiei parinte, care a definit-o, chiar si dupa ce context al functiei parinte a fost sters din stack. Deci, context al functiei parinte va continua totusi sa existe in memorie, nu si in stack
* **Closure** este variable environment din execution context inchis/finisat in care o functie a fost creata, chiar si dupa ce execution context e terminat
* const func = function() {  
   let counter = 0;  
    
   return function() {  
   counter++;  
   *console*.log(counter);  
   }  
  }  
    
  const *test* = func();  
    
  *test*()  
  *test*()  
  *test*()

Cand a fost apelata func(), in callback stack exista context global, si context al functiei func. Apoi,functia a returnat in global o noua functie si context al functiei func a fost sters din stack

Totusi, apelarea functiei test() de 3 ori a returnat

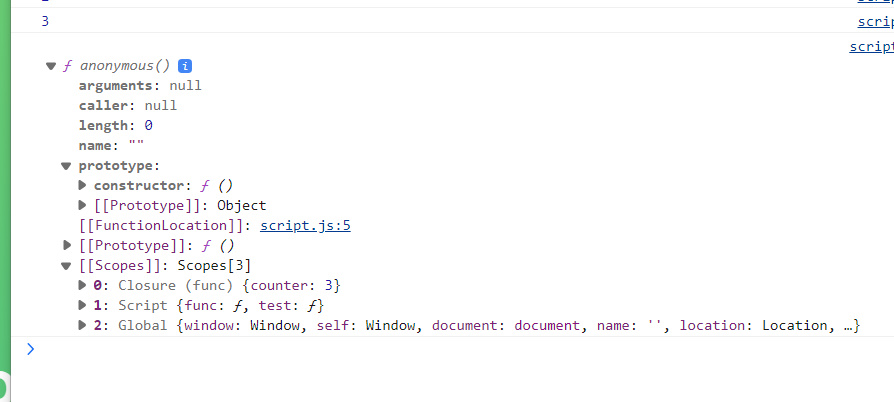
1

2

3

Partea stranie e ca varabila counter a fost din functia func(), a carui context a fost sters, insa functia test inca are acces la ea

* Anume aici si apare closure. Closure e doar referinta functiei la scope al functiei parinte, sau mai bine zis la context
* Chiar daca context al functiei ce a creat-o a fost scos din stack, JS nu l-a sters, ca stie ca functia test va avea nevoie de el, si functia test va continua sa pastreze referinta la el, dar noi nu il putem acces nicidecum!
* Daca folosim:
* *console*.dir(*test*)





* **Functia nu trebuie sa fie neaparat creata in return, poate fi creata oriunde in metoda, si apoi sa fie returnata sa o variabila globala sa o ia:**
* let *test*;  
  const func = function() {  
   let counter = 0;  
   *test* = function() {  
   counter++;  
   *console*.log(counter);  
   }  
  }  
    
  func()  
    
  *test*()  
  *test*()  
  *test*()

**test e variabila globala**