1. **GLOBAL SCOPE Promenljivih**

Sve sto je definisano kao GLOBALNO njemu se moze pristupiti odakle god!!!

Odnosno sve sto **nije** definisano u funkciji je globalno.

var name = “PERA”;

function cao(){

alert(“Kako i mi danas” + name); => OVO JE OK I radi

}

------------------------------------------------------------------------------------------------------------------------------------------

1. **LOCAL SCOPE Promenljivih**

Sve sto se definisu unutar funkcija je dostupno SAMO unutar te funkcije

Function cao(){

var name = “PERA”;

alert(“kako si mi danas” + name); => OVO JE OK I radi

}

alert(“kako si mi danas” + name);=> ERROR :name is not defined

Lexical Scope(STATICKI) u vecini slucajeva se korist dok DINAMICKI se ne koristi. Jedini nacin da prevarimo lexical Scope je koriscenje funkcije eval() ali njiu treba IZBEGAVATI

Ako postoji promenljiva sa ISTIM nazivom kako globalan I kao promenljiva, kompajler ce koristiti

OVAJ PRIMER **NE RADI**

var x = 'vrednost spoljne promenjive';

(function() {

console.log(x**); // Izlaz: undefined zato sto je promenljiva definosana posle console.log**

var x = "promenjiva unutar IIFE"; })();

OVAJ PRIMER **RADI**

var x = 'vrednost spoljne promenjive';

(function() {

var x = "promenjiva unutar IIFE";

console.log(x); // promenjiva unutar IIFE je rez jer je promenljiva definisana pre console.log

})();

function testScope() {

var unutrasnja = "Ja sam lokalna promenjiva";

console.log (unutrasnja);

}

testScope(); // Izlaz: Ja sam lokalna promenjiva

console.log(unutrasnja); // Izlaz: Error: unutrasnja is not defined

Lokalna promenjiva (promenjiva unutar funkcija) i fuction scope postoje samo pri pozivanju funkcije (invoke function). Zbog toga posledanja naredba iz prethodnog primera izbacuje grešku, Gledano iz globalnog scopa, ako zanemarimo granice scopa, promenjiva “unutrašnja” ne postoji, jer je funkcija “odavno” izvršena tj. promenjiva je postojala u trenutku izvršenja ali nakon toga više ne postoji.

**Childe and Parent Scope**

Ako postoji funkcija(parent funkcija) u funkciji(childe funkcija) promenljive koje su definisane u parent mogu se videti u childe funkciji ALI ne vazi obrnuto, childe promenljive nisu vidljive u parent funkcijama

function parent() {

var name = 'Foobar'; // Scope A

function child() {

function grandChild() {

alert('My father is ' + name); // My father is Foobar

}

}

}

**RODITELJ NE MOZE pristupiti CHILD varijablama**

var Laza = function(){

alert(name); // greska jer ne moze da pristupi childe varijabli

var Pera = function(){

var name =”Mika”;} }

Ako postoji varijabla u parent I u childu sa istim imenom funkcija ce uvek pristupiti scopu funkcije za koju je pozvana

**THIS**

[**http://www.webprogramiranje.org/this-u-javascriptu-bind-call-apply/**](http://www.webprogramiranje.org/this-u-javascriptu-bind-call-apply/)

**THIS nije promenljiva vec SPECIJALNA rec koja predstavlja VEZU koja ukazuje na neku reference(OBJEKAT). Aktivira se pri budjenju(invoke) funkcije**

Postoje 4ri pravila na osnovu kojih se moze odrediti na **KOJU REFERNCU**  ukazuje *THIS* , sva pravila zavise od mesta I nacina pozivanja

1. Pozivanje konstruktorske funkcije sa operatorom **new**
2. **Explicitno povezivanje** (direktno povezivanje) – pozivanje sa metodama bind(), call() i apply()
3. **Implicitno povezivanje** (posredno povezivanje) – pozivanje metode objekta
4. **Podrazumevano povezivanje**

**POZIVANJE FUNKCIJE SA OPERATORON “NEW”**

**Nakon pozivanaj konstruktorke funkcije sa operatorom NEW , “*THIS*”ukazuje na instancu objekta**

function osoba(ime, prezime){

this.firstName = ime;

this.lastName = prezime;

}

var pera = new osoba(“Laza”,”Lazarevic”); // mora pod navodnicima “”

alert(pera.lastName);

function Laza(){

return this.val;} // ovo THIS ukazuje na OBJEKAT koji je pozvao metodu

var obj={

pera:Laza,// ovde se ne stavljaju ()

val:5

}

document.write(obj.pera()); => rez je 5

Da sam imao X objekata I u svakom istu fun LAZA zbog THIS bi se uvek ukazivalo na OBJ koji poziva tu fun I vraca VAL tog objekta

POZIVANJE METODE OBJEKTA

Kod prototipskog nasledjivanja je sledeca situacija:

var parent={

val:50,

myFun:function(){

return this.val;

}}

var childe=Object.create(parent);

childe.val=100;

document.write(childe.myFun());//=>rez 100;

var gChilde=Object.create(childe);

document.write(gChilde.myFun());//=>rez 100;

Kada pozovemo **gChilde.myFun()** dolazimo do gChilde OBJEKTA on trazi myFun u tom objektu.Posto ne nalzi ide do childe OBJ ali nit u ne nalzi pa odlazi do parent OBJ I tu nailazi na myFun koja kaze “return this.val” ovo this UKAZIJE na OBJ koji ga je pozvao I vracu vrednost val koju on ima(u ovom slucaju 100 jer je nasledio od childe OBJ )

function foo (){

console.log(this.a);}

spoljniObjekat = {

a: 2,

foo:foo

}; spoljniObjekat.foo();

Na prethodnom primeru funkcija foo() nije u direktnom vlasništvu objekta **“spoljniObjekat”**, već svojstvo objekta ima spoljnu referencu na tu funkciju. Za odredjivanje objekata na koji ukazuje operator this je bitno mesto pozivanja funkcije koja angažuje this, tako da u ovom slučaju pošto se funkcija poziva iz objekta “spoljniObjekat”, ona se ponaša kao metoda objekta pa thisukazuje na objekat!

function foo (){

//var a = 7;

console.log(this.a);}

spoljniObjekat = {

a: 2,

foo:foo};

spoljniObjekat.foo(); // rez =>2

veliki={

a:55,

spoljniObjekat:foo}

veliki.spoljniObjekat();// rez =>55

------------------------------------------------------------------------------------------------------------------------------------------

**POZIVANJE SA call(),apply() ili bind()**

const objj={

ime:"Pera",

prezime:"Peric", // mora da se koristi knjicna rec CONST

}

const r = function(){

console.log(this.ime);

}r.bind(objj)**()**;

Bind() sluzi da bi mogli da povezemo funkciju I objekat na koga ce da ukazuje kljucna rec “this”

Kod metoda **call()** i **apply()** prvi argument definiše na koji objekat će ukazivati this, dok ostali argumenti prosledjuju parametre potrebne osnovnoj funkciji. Jedina razlika izmadju metoda apply() i call() je način u kom metoda prihvata argumente koji se prosledjuju osnovnoj funkciji. Metoda call() dodaje ostale **argumente sa zarezom**, dok metoda apply() dodaje ostale elemente **kao niz elemenata**.

nekaFunkcija.call(objekatNaKojiUkazujeThis, drugiArgument, treciArgument...)

nekaFunkcija.apply(objekatNaKojiUkazujeThis, [drugiArgument, treciArgument...])

I