10.03.2025.

Hoisting

This

Call,apply

Dosezanje

Zatvaranje

Leksički opseg

JS engine V8

Objekti: deskriptori svojstava

Hoisting (Podizanje)

```
function konfuzija() {
  alert('Tip od foo:');
  alert(typeof foo);
  alert('Tip od bar:');
  alert(typeof bar);
     var foo = 'hello', // foo je varijabla sa vrednošću 'hello'
     bar = function() { // liči kao da je bar funkcija
            return 'world';
        };
     function foo() {   // foo je funkcija
        return 'hello';
        }
  }
  konfuzija();
```

Rezultat:?

Tip od foo: function

Tip od bar: undefined

Šta ovde imamo?

Imamo funkcijski izraz u kojem je:

- **foo** deklarisana kao varijabla kojoj je dodeljena vrednost `'hello'`
- **bar** deklarisana kao funkcija koja vraća string `'world'`
- **foo** deklarisana kao funkcija koja vraća string `'hello'`.

```
var foo = 'hello', // foo je varijabla sa vrednošću 'hello'
    bar = function() { // liči kao da je bar funkcija
          return 'world';
       };
    function foo() {   // foo je funkcija
       return 'hello';
       }
```

Hoisting (Podizanje)

- Postupak kojim se deklaracije varijabli i funkcija postavljaju na vrh njihovog funkcijskog konteksta.
- Posledice su sledeće:
  - **Funkciji se može pristupiti pre nego što je deklarisana**
  - **Ono što je deklarisano kao funkcija ostaje funkcija i ako se (više puta) deklariše kao varijabla**

- Hoisting je JavaScript mehanizam u kojem deklaracije **varijabli** i **funkcija** bivaju "podignute" (hoisted) na vrh njihovog **scope-a** (oblasti važenja) pre nego što se kod izvrši.

```
alert(a); // ispis?
var a = 10;
alert(a); // ispis?
```

undefined (nije greška, ali nema vrednost)
10

- Java script to tretira ovako:

```
var a; // Hoisting - deklaracija se podiže na vrh
alert(a); // undefined
a = 10;
alert(a); // 10
```

- To ne važi za let i const:

```
alert(b); // Cannot access 'b' before initialization
let b = 20;
```

A zašto su posledice hoisting-a takve?

1. Zašto možemo da pristupimo funkciji `foo` pre nego što smo je deklarisali?
2. `foo` je dva puta deklarisana. Zašto je `foo` tipa `function` a nije `undefined` ili `string`?
3. Zašto je `bar` `undefined` ?

Odgovor:

1. Varijable su već kreirane pre ulaska u fazu izvršavanja. Dakle, kada počne da se izvršava kod iz primera, **`foo`** je već definisana u aktivacionom objektu.
2. Funkcije u aktivacionom objektu se kreiraju pre varijabli i, ako ime svojstva već postoji u aktivacionom objektu, nova deklaracija se prosto zaobilazi pa **`foo`** ostaje **`function`**.
3. Zato što je **`bar`** u stvari varijabla kojoj je dodeljena funkcija (operatorom dodele vrednosti), a varijable se kreiraju u fazi kreiranja i , pri tome, im se inicijalno dodeljuje vrednost **`undefined . Kao u prethodnom slajdu prvo alert(a)`**

Ključna reč `this`

- Ključna reč `this` javlja se i u drugim jezicima i, u većini slučajeva, ona **pokazuje na objekat koji je kreiran.**
- **U JS-u to nije jedino pravilo i ovde ključna reč `this` može često da pokazuje na različite objekte iz različitih konteksta izvršavanja.**

- **Generalno, u JS-u je pokazivač this poprilično zakomplikovan, pokazuje na razne stvari u različitim uslovima i zahteva veliku pažnju i pedantnost ako se žele izbeći zabune.**
- Ovde ćemo kazati samo neka najizraženija pravila.

1. JS `this` u globalnom kontekstu

- Kada god se koristi ključna reč **this** u globalnom kontekstu (ne unutar funkcije), ona uvek pokazuje na globalni objekat.
- Primer1:

```
// globalni doseg
foo = 'abc'; /* deklarisano u globalnom dosegu */
alert(foo); // Ispis?
```

abc

```
alert(this === window);
```

Rezultat:true

Šta bi bilo kada bi stavili:

```
alert(this)
```

JS `this` u funkcijskom kontekstu

```
var boat = { /* kontekst objekta boat */
    size: 'normal',
    boatInfo: function() { /* kontekst
                             funkcije */
        alert(this === boat);
        alert(this.size);
    }
};
boat.boatInfo(); // Ispis: true, 'normal'

var bigBoat = {/* kontekst objekta bigBoat */
    size: 'big'
};

bigBoat.boatInfo = boat.boatInfo; // preuzima...
bigBoat.boatInfo(); // Ispis: false, 'big'
Šta bi se ispisalo sa alert(this === window);
```

- Šta se ovde dešava sa **this** pokazivačem?
- Dok smo i “malom čamcu”, **this** pokazuje na **boat**; a tamo je this===boat true i boat,size je `'normal’`
- Kada pređemo na veliki čamac (**`bigBoat.boatInfo = boat.boatInfo`**), this pokazuje na **bigBoat**, a ne na **boat**, pa je **`this === boat`** `neistinito (istinito je` **`this === bigBoat`**`) i svojstvo size ima vrednost 'big' a ne 'normal’.`

2. JS `this` kao metod objekta

- **Metod objekta**: Kada se funkcija poziva kao metod objekta, **this** se odnosi na sam objekat na kojem je pozvana ta funkcija.

```
var obj = {
  name: 'John',
  greet: function() {
    alert('Hello, my name is ' + this.name);
  }
};
obj.greet();
```

Rezultat: Hello, my name is John

Šta bi se dobilo sa alert('Hello, my name is ' + obj.name);

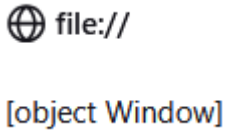
Isto:Hello, my name is John

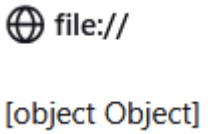
JS `this` u funkcijskom kontekstu

- U JS-u vrednost ključne reči `this` unutar funkcije **nije statična**, ona se **određuje svaki put kada se funkcija pozove**, ali **pre no što se stvarno izvrši kod funkcije**.
- Vrednost za ključnu reč **`this` unutar funkcije u stvari obezbeđuje roditeljski doseg u kome je funkcija pozvana**, a još zavisi i od načina na koji je sintaksa funkcije napisana.

JS **this** u funkcijskom kontekstu$_1$

```
function bar() {
    alert(this);
}
bar();  /* this pokazuje na globalni objekat (Window u
                brauzeru) */

var foo = {
    baz: function() {
      alert(this);
    }
}
foo.baz(); /* this pokazuje na objekat foo */
```



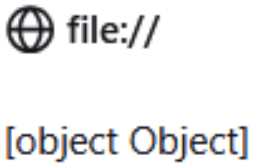


Функција bar је позвана као **обична функција** (не као метод објекта). **this** у обичној функцији упућује на **глобални објекат**. У претраживачу, глобални објекат је window, па ће **alert(this)** исписати [object Window].
Метод **baz** припада објекту foo и позван је преко **foo.baz().**
У овом случају, this упућује на објекат foo, јер је метод позван кроз тај објекат.

JS `this` : zavisi od sintakse poziva

- Ključna reč `this` može biti različita za različite sintakse pozivanja funkcije:

```
var foo = {
    baz: function() {
        alert(this);
    }
}
// prva sintaksa poziva
foo.baz(); /* foo - zato što se baz poziva kao metoda objekta foo */
```



this унутар baz() ће бити foo, јер је метод позван преко тог објекта.

```
// druga sintaksa poziva
var anotherBaz = foo.baz;
anotherBaz(); /* global - zato što se funkcija anotherBaz()
         poziva samostalno (u stvari, kao metoda
         globalnog objekta) a ne kao metoda objekta foo */
```

Када позивамо anotherBaz(), он се извршава као обична функција, а не као метод неког објекта

JS **this** : ugnježđene funkcije

```
var anum = 33;
var foo = {
    anum: 10,
    baz: {
        anum: 20,
        bar: function() {
          alert(this.anum);
        }
    }
}
foo.baz.bar(); // Ispis:20
var hello = foo.baz.bar;
hello(); // Ispis:33
```

Poziv **foo.baz.bar()** ispisuje 20, jer je **baz levo od bar ()** pa **this** pokazuje na **baz** u kome je anum 20.

-Poziv **hello();** ispisuje 33 jer **levo od hello** nema ničega pa this pokazuje na globalni objekat u kome je anum 33

apply(),call()

- Ugradjene metode **metode call() i apply(), omogućuju objektima da pozajme metode od drugih objekata i pozovu ih kao svoje.**
- Kod metoda **call()** i **apply() prvi argument definiše na koji objekat će ukazivati *this***, dok **ostali argumenti prosledjuju parametre** potrebne osnovnoj funkciji.
- Jedina razlika izmedju metoda *apply() i call()* je načinu kom metoda prihvata argumente koji se prosledjuju osnovnoj funkciji.
- Metoda *call()* dodaje ostale **argumente sa zarezom**.
- Metoda *apply()* dodaje ostale elemente **kao niz elemenata**.
- Za lakše pamćenje u kakvom obliku koja metoda prihvata argumente koristite prva slova metoda:

Call <=>**C**omma

Apply <=>**A**rray

Call()

```
var javaskript= {
  nadimak: 'JS',
  pozdrav(program) {
  alert('Zdravo programe ' + program + ', ja sam ' + this.nadimak)
                        }
                    }
javaskript.pozdrav('Skript')
```

- Zdravo programe Skript, ja sam JS

pozdrav(program) је метод (функција унутар објекта) који прихвата параметар program и исписује поруку помоћу alert().

call(guru, 'Superskript') postavlja this na guru.
this.nadimak sada pokazuje na 'Javascript guru'.
program je 'Superskript'

Umesto poslednjeg reda dodajemo:

```
var guru ={ nadimak: 'Javascript guru' }
javaskript.pozdrav('Skript')
javaskript.pozdrav.call(guru, 'Superskript')
```

- Zdravo programe Skript, ja sam JS
- Zdravo programe Super skript, ja sam Javascript guru

Apply()

- Ako umesto call stavimo apply:

javaskript.pozdrav.apply(guru, 'Super skript')

- Neće raditi posle ispisa:
- Zdravo programe Skript, ja sam JS

Ali ako stavimo:

javaskript.pozdrav.apply(guru, ['Super skript'])

- Radiće kao i pre:
- Zdravo programe Skript, ja sam JS
- Zdravoprograme Super skript, ja sam Javascript guru

Jer: Metoda *apply()* dodaje ostale elemente **kao niz elemenata**

JS `this` : eksplicitno postavljanje

- Radi se pomoću funkcija `call()` i `apply()`:

```
var person = {
  fullName: function(city, country) {
    return this.firstName + " " + this.lastName + "," + city + ","
                          + country;
  }
}
var person1 = {
  firstName:"Pera",
  lastName: "Perić"
}
var person2 = {
  firstName:"Mika",
  lastName: "Perić"
}
```

Da li će raditi ako u drugom redu stavim: **apply**?

- `alert(person.fullName.apply(person1, ["Beograd", "Srbija"])); /* this pokazuje na person1;` Ispisaće: **Pera Perić,Beograd,Srbija** */
- `alert(person.fullName.call(person2, "Novi Sad", "Srbija")); /* this pokazuje na person2;` Ispisaće: **Mika Perić,Novi Sad,Srbija** */

- Primer :

```
function greet() {
   alert('Hello, my name is ' + this.name);
}
var person = { name: 'John' };
greet(); //Šta će ispisati?
```

1.Rezultat:Hello, my name is

```
greet(person); //Šta će ispisati?
```

2.Rezultat:Hello, my name is

```
greet.call(person); //Šta će ispisati?
```

3.Rezultat:Hello, my name is John

1.This će pokazivati na globalni objekat

2.funkcija greet() nema parametre!

- JavaScript ignoriše argument

3. .call(person) eksplicitno postavlja this na objekat person

Dosezanje i zatvaranje

Terminologija

- ***Vezivanje*** (engl. *binding*) označava **povezivanje vrednosti** i **imena** putem različitih mehanizama:?
  - ključnih reči za deklarisanje varijabli (`var` , `let`, `const`)
  - funkcijskih argumenata,

  function sum(a, b): funk.arg. su a i b
  - Prosleđivanja pokazivača this, (videli ranije)
  - dodele svojstva (sledeći slajd)
- ***Dosezanje*** u najopštijem slučaju označava šemu razrešiti varijable (kako program “dolazi” do vrednosti koju će da poveže sa imenom).
- ***Slobodne varijable*** su varijable koje nisu lokalno deklarisane niti prosleđene kao parametar.

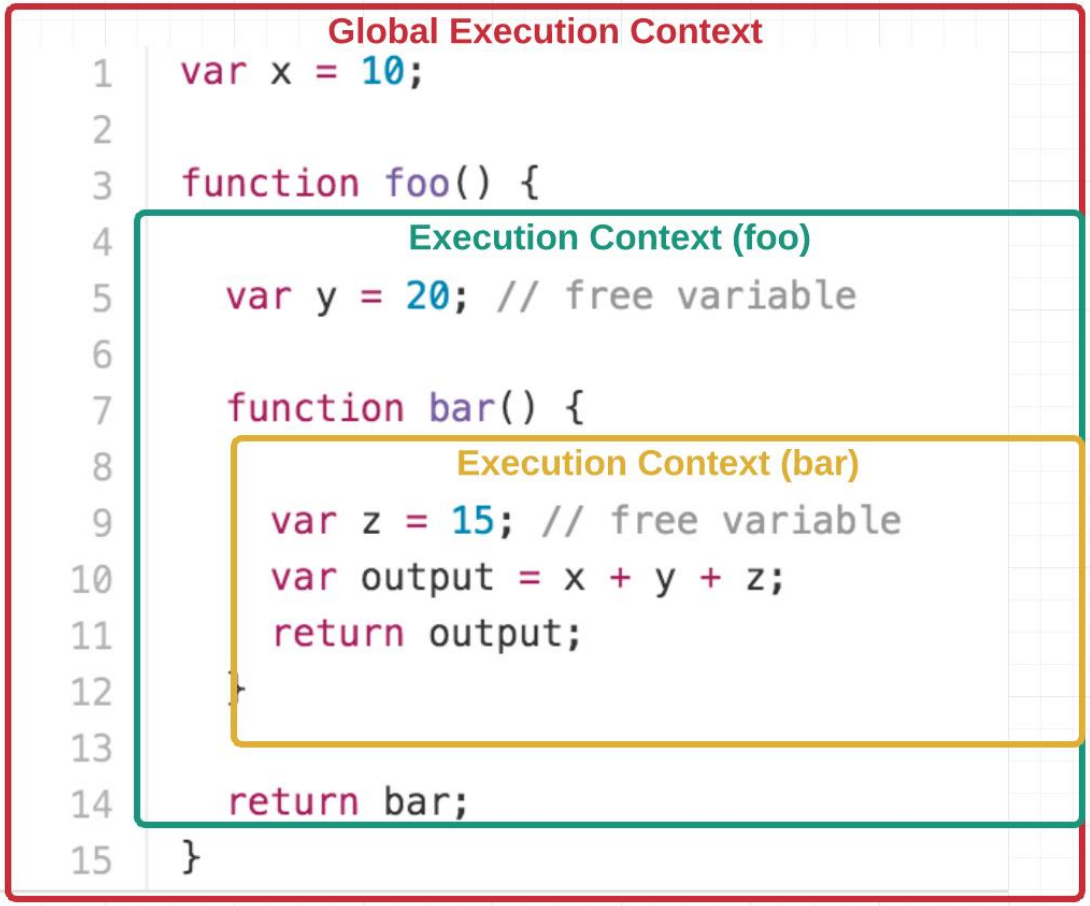
Vezivanje preko dodele svojstva:

```
// Kreiramo prazan objekat
let osoba = {};
// Dodeljujemo svojstvo 'ime' objektu 'osoba'  itd
osoba.ime = "Marko";
osoba.prezime = "Marković";
osoba.godine = 30;
```

// Ispisujemo objekat . Kako ćemo ispisati?

- alert(osoba.ime + osoba.prezime + osoba.godine);
- Rezultat?
- MarkoMarković30

Da se podsetimo: Kontekst izvršavanja$_1$




```
1  var x = 10;
2
3  function foo() {
4
5    var y = 20; // free variable
6
7    function bar() {
8
9      var z = 15; // free variable
10     var output = x + y + z;
11     return output;
12   }
13
14   return bar;
15 }
```

Varijabla x je slobodna varijabla u kontekstu `bar()` i u konekstu `foo()`. Zauzeta je u globalnom kontekstu.

Varijabla y je slobodna u kontekstu `bar()` i u globalnom konekstu. Zauzeta je u foo() kontekstu.

Varijabla z je slobodna u konekstu foo() i u globalnom konekstu. Zauzeta je u bar() kontekstu.

Slobodne varijable su varijable koje nisu lokalno deklarisane niti prosleđene kao parametar.

2.čas

Kontekst izvršavanja: Primer 1 (kod)

```
var x = 10;
function foo(a){
var b =20;

   function bar(c){
   var d =30;
      return boop(x + a + b + c + d);
   }

   function boop(e){
      return e *-1;   //šta je ova operacija?
   }

return bar;
}

var moar = foo(5); // Zatvaranje
```

alert(moar(0)); //Šta je ispis?

Rezultat: -65

alert(moar(1)); //Šta je ispis?

Rezultat: -66

0. **x=10**

1.Poziv funkcije **moar()** izvršava funkciju **foo(5). a=5**

2.U okviru funkcije **foo (5)**, (b=20), funkcija **bar** poziva funkciju **boop()** **(d=30)**

3.i u toj tački se funkcija **bar()** suspenduje a funkcija **boop()** se postavlja na vrh steka i izvršava se.

4.10+5+20+30=65 (c je nepoznato)

5.Nakon što se izvrši funkcija **boop()**, ponovo je na vrhu steka suspendovana funkcija **bar()** i ona se izvršava.

•Ovde je demonstrirana još jedan važan koncept: **zatvaranje** –sposobnost funkcije da pristupi kontekstu roditeljske funkcije – funkcije u kojoj je deklarisana kada je roditeljska funkcija završila svoje izvršavanje. U našem primeru, funkcija **foo()**je roditeljska za funkcije **bar()** i **boop()** pa one mogu da pristupe njenom kontekstu i nakon što se završi poziv

```
var x= 10;


function foo(a){
var b=20;
function bar(c){
var d=30;
return boop(x+a+b+c+d)
}
function boop(e){
return e *-1;
}
return bar;
}
var moar = foo(5);


alert(moar(0));// -65
```

Kako proveravati šta radi program?

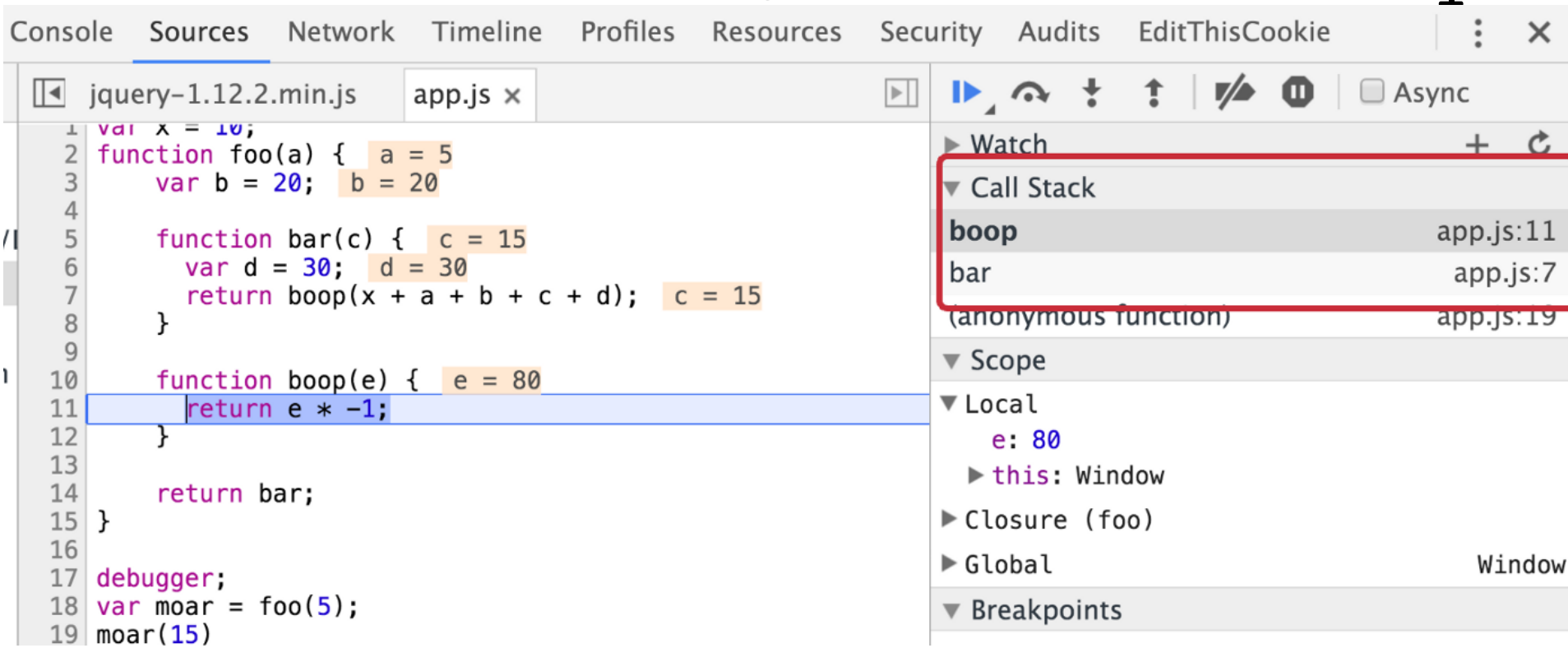
```
var x= 10;

function foo(a){
var b=20;
alert("x="+ x + "  a=" + a  + "   b=" + b)
function bar(c){
var d=30;
alert("c=" + c  + " d=" + d)
return boop(x+a+b+c+d)
}
function boop(e){
  alert("e=" + e)
return e *-1;
}
return bar;
}
var moar = foo(5);

alert(moar(0));// -65
```

- Poziv foo vraća funkciju **bar**, ali **ne izvršava bar** odmah.
- moar sada sadrži referencu na funkciju bar.

- moar(0) znači da pozivamo bar(0), gde c = 0.

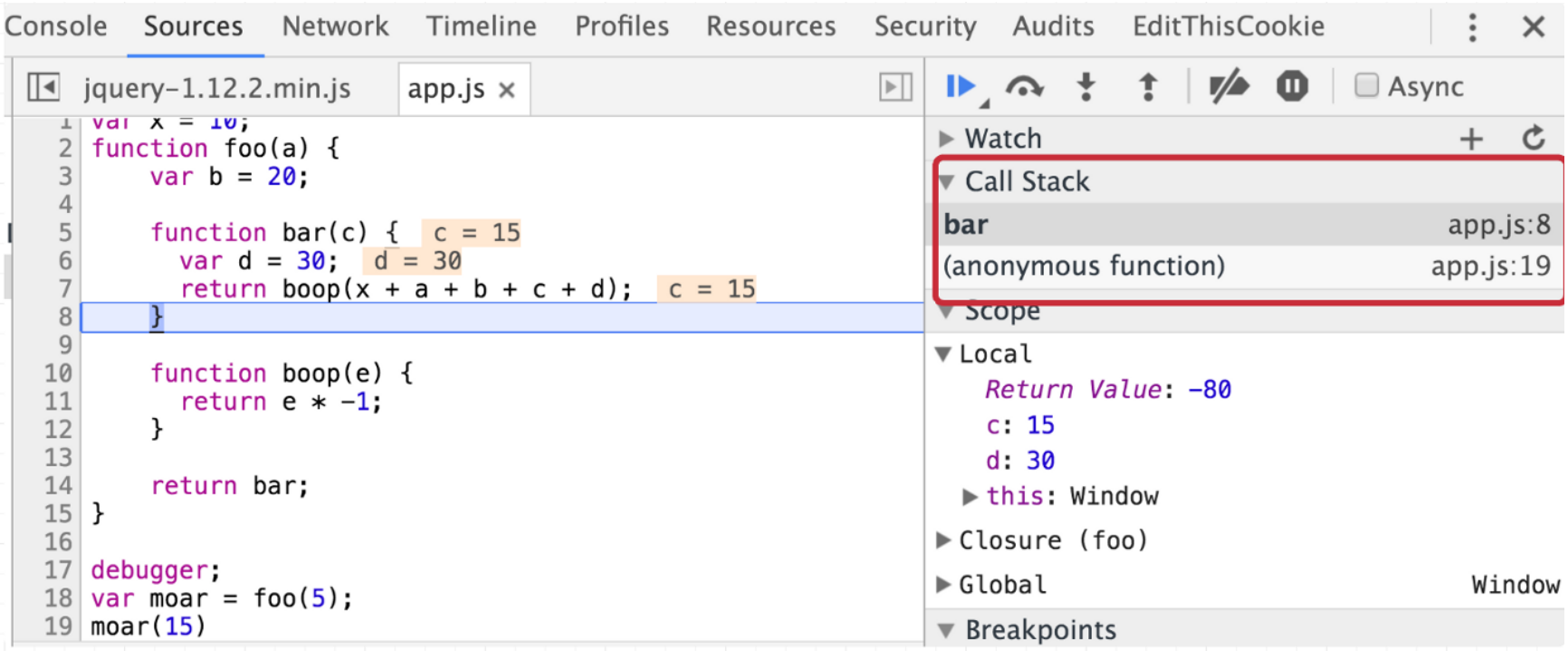
- Developer Tools, Googlov alat:
- https://developer.chrome.com/docs/devtools

Kontekst izvršavanja: Primer 1 (stek)$_1$



Poziv funkcije **`moar(15)`** izvršava funkciju **bar()** koja se vraća kada se izvrši funkcija **foo (5)**. Funkcija **bar()** poziva funkciju **boop()** i u toj tački se funkcija **bar()** suspenduje a funkcija **boop()** se postavlja na vrh steka

Kontekst izvršavanja: Primer 1 $(stek)_2$



Kontekst izvršavanja: praćenje napredovanja koda

- **Koncepti:**

1. Stanje evaluacije koda

- **Gde se trenutno nalazi izvršenje** (koja linija koda se izvršava).
- **Koje funkcije su pozvane** i gde treba da se vrati rezultat.
- **Koje su promenljive dostupne** u tom trenutku.

2. Funkcija

- Funkcija u JS je **poseban blok koda koji se može izvršiti više puta**.
- Svaka funkcija kreira **novi kontekst izvršavanja** kada se pozove.
- Unutar funkcije se čuvaju **lokalne promenljive i parametri**

3. Realm

"Realm" je okruženje u kojem se izvršava JavaScript kod.

4. Leksičko okruženje (ambijent)

Prati dostupne promenljive

5. Variable okruženje (ambijent)

Koristi za **deklarisanje varijabli sa var, let, const**.

Leksički ambijent: definicija

- Leksički ambijent (lexical environment) u JavaScriptu je koncept koji definiše okruženje u kojem se identifikatori i promenljive vezuju za njihove vrednosti tokom izvršavanja koda.

- Svaki put kada se funkcija pozove, stvara se novi leksički ambijent koji sadrži informacije o okruženju u kojem je funkcija definisana, uključujući sve spoljne (roditeljske) leksičke ambijente.

- **Reference to the Outer Environment**: Ovo je referenca na roditeljski leksički ambijent, što omogućava funkcionisanje lanca leksičkih ambijenata.
- To omogućava funkcijama da pristupe promenljivama iz okruženja u kojem su definisane (okruženja u kojem su definisane).

```
function out() {
   var x = 10;
   function inn() {
   alert(x);
   }
   inn();
}
out();
//Rezultat?
```

- Leksički ambijent funkcije **inn** sadrži referencu na leksički ambijent funkcije **out** kao njegov roditeljski ambijent.
- Kada se funkcija **inn** izvršava, traži vrednost promenljive **x**.
- Pošto **x** nije definisan u leksičkom ambijentu **inn**, traži se u roditeljskom ambijentu, tj. u leksičkom ambijentu funkcije **out.**
- Zato funkcija **inn** može pristupiti promenljivoj **x** definisanoj u funkciji **out**.

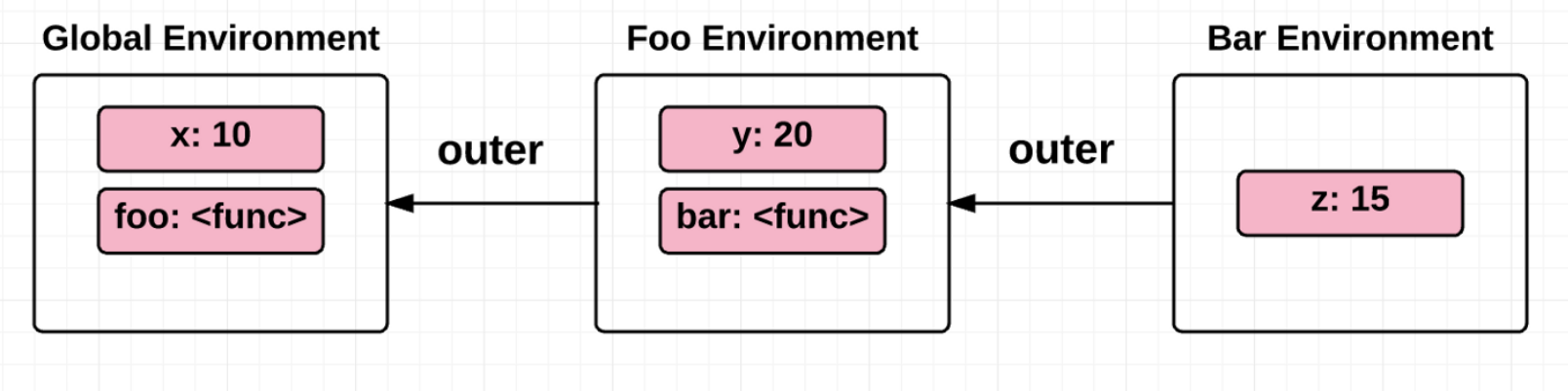
Leksički ambijent: lanac dosezanja

- Ambijent ima pristup svom roditeljskom ambijentu, a taj roditeljski ambijent ima pristup ambijentu svog roditelja, i tako do globalnog ambijenta.
- Skup identifikatora kojima svaki ambijent ima pristup zove se ***doseg***.
- Dosezi se mogu ugnježdavati u hijerarhijski lanac ambijenata a taj lanac se zove ***lanac dosezanja***.

Leksički ambijent: Primer 2 - ugnježđena struktura

```
var x = 10;

function foo(){
var y = 20;// slobodna varijabla

  function bar(){
     var z = 15;// slobodna varijabla
       return x + y + z;
   }
   return bar;
}
```

Leksički ambijent: primer ugnježdene strukture - vizualizacija




```
var x = 10;
function foo(){
var y = 20;// slobodna varijabla
    function bar(){
        var z = 15;// slobodna varijabla
            return x + y + z;
    }
    return bar;
}
```

Ovde je **bar** ugnježden unutar **foo**
a **foo** je ugnježden u globalni ambijent

Dosezanje

- Dosezanje se odnosi na životni vek varijable, to jest koliko dugo varijabla zadržava određenu vrednost.
- Najduže "žive“ varijable čiji "život" traje koliko traje i izvršavanje samog programa
- Te varijable zovu se ***globalne varijable***, ***globali***

Leksičko dosezanje: osnov vidljivosti

- Problem sa globalnim varijablama je što njih svako i u svakom trenutku izvršavanja programa može da promeni.
- Naravno, postoje i mehanizmi da se ovo spreči.
  - To što varijabla čuva određenu vrednost tokom celog svog životnog veka ne znači da će se svakim referenciranjem varijable ta globalna vrednost i dobiti.
  - Tu dolazimo do koncepta zvanog ***leksičko dosezanje***.
- ***Leksičko dosezanje*** odnosi se na ***vidljivost varijable i njene vrednosti na osnovu tekstualne reprezentacije (mesta gde se varijabla nalazi u kodu)***.

Leksičko dosezanje: Primer

```
var  outerVar= 'Outer';
function outerFunction() {
    var innerVar= 'Inner';
    function innerFunction() {
        alert(innerVar);
        alert(outerVar);
     }
   innerFunction();
}
outerFunction();
```

Rezultat:Inner pa Outer

- U ovom primeru, funkcija **innerFunction** ima pristup promenljivoj **innerVar**, kao i promenljivoj **outerVar**, iako su obe deklarisane u različitim opsezima.
- To je zbog leksičkog dosezanja-interpreter prvo traži identifikator unutar najbližeg opsega, a ako nije pronađen, nastavlja da traži u nadređenim opsezima sve dok ne stigne do globalnog opsega

Dinamičko dosezanje

- **Kod leksičkog dosezanja varijable i njihove vrednosti traže se u izvornom kodu.**
- Dinamičko dosezanje za određivanje dostupnih varijabli (i odgovarajućih vrednosti) traži varijablu i odgovarajuću vrednost u steku izvršavanja/pozivanja.

•**Statički doseg (Lexical Scope)** → Određen je prema tome gde je funkcija **deklarisana u kodu**.

•**Dinamički doseg (Dynamic Scope)** → Određen je prema tome **odakle je funkcija pozvana**. Funkcija traži promenljive na osnovu toga **odakle je pozvana**.

Leksičko i dinamičko dosezanje: Primer

```
var x =10;

function foo(){
   var y = x +5;
   return y;
}

function bar(){
   var x =2;
   return foo();
}

function main(){
foo();// Statički doseg: 15; Dinamički doseg: 15
bar();//Statički doseg: 15; Dinamički doseg: 7
   return 0;
}
main();
```

U ovom primeru, pri statičkom dosezanju **povratna vrednost funkcije bar** je bazirana na vrednosti promenljive **x** u vreme kreiranja funkcije **foo**.

Rezultujuća vrednost 15 je posledica statičke i leksičke strukture koda u kome je varijabla **x** asocirana sa vrednošću **10**, odnosno **15 = 10+5**.

Dinamičko dosezanje održava stek definicija varijable u toku izvršavanja tako da **x** koje se koristi za sračunavanje rezultata zavisi od toga šta je na vrhu tog steka u trenutku računanja što se dinamički definiše u toku izvršavanja.

Izvršavanje funkcije bar postavlja na vrh steka promenljive x vrednost 2 pa je rezultat izvršavanja funkcije foo jednak 2+5=7.

Funkcijsko dosezanje

- Osnovni model dosezanja u JS-u je ***funkcijsko dosezanje***.
- Moglo bi se reći da liči na dinamičko dosezanje, ali se od njega razlikuje u logici povezivanja i traženja
  - Umesto da se povezivanjima pristupa u globalu, funkcijski model uvodi ograničavanje svih vezivanja na najmanju moguću oblast - funkciju

JS lanac dosezanja (`scopeChain`)$_{2/2}$

- Lanci dosezanja uspostavljaju doseg zadate funkcije
- Svaka definisana funkcija ima sopstveni ugnježdeni doseg i svaka funkcija definisana unutar druge funkcije ima lokalni doseg koji je povezan sa spoljašnjom funkcijom – ta veza se zove ***lanac dosezanja***.

JS lanac dosezanja: primer $1_{1/2}$

```
function one() {
    two();
    function two() {
        three();
        function three() {
            alert('Ja sam funkcija three');
        }
    }
}

one(); // ispis: ?
```

Ja sam funkcija three

Funkcija **one()**je definisana u globalnom dosegu, funkcija two() je definisana u dosegu funkcije one(), funkcija three() je definisana u dosegu funkcije two(). Poziv funkcije one() poziva funkciju two() koja poziva funkciju three().

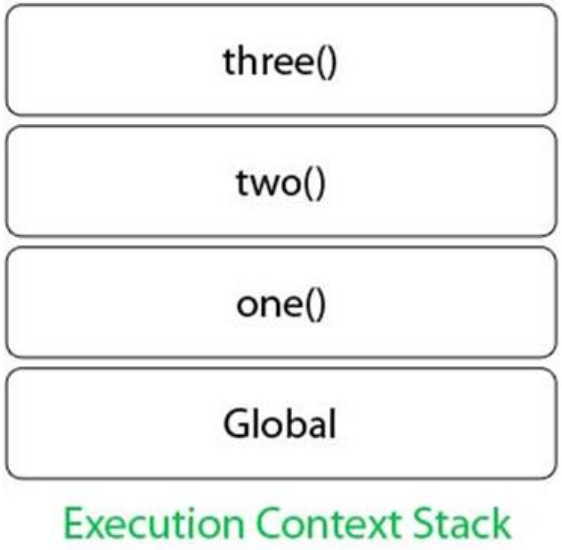
JS lanac dosezanja: primer $1_{2/2}$

```
function one() {
  two();
  function two() {
     let prefix = 'Ja sam prefix iz funkcije two';
     three();
     function three() {
        alert ('Ja sam funkcija three ');
        alert (prefix);
     }
  }
}

one();
```

```
Ja sam funkcija three
Ja sam prefix iz funkcije two
```

Funkcija **one()**je definisana u globalnom dosegu, funkcija two() je definisana u dosegu funkcije one(), funkcija three() je definisana u dosegu funkcije two(). Poziv funkcije one() poziva funkciju two() koja poziva funkciju three().

JS lanac dosezanja: primer 1



Stek konteksta izvršavanja

Lanac dosezanja pri izvršavanju funkcije `three ()`:

Razrešavanje vrednosti varijable

- Razrešavanje vrednosti varijable se vrši prolaskom kroz lanac dosezanja počevši od tekućeg i završavajući sa globalnim.
- Prvi put u lancu kada se nađe ime varijable daje se i njenu vrednost.
- Ukoliko se neka varijabla ne može razrešiti (nema njenog imena nigde u lancu), rezultat je fatalna greška sa porukom: **Uncaught ReferenceError: *ime_varijable* is not defined**

šta je ovde ispis ?

```
function one() {
    var a = 1;
    two();
    function two() {
        var b = 2;
        three();
        function three() {
            var c = 3;
            alert(a + b + c);
            // Ispis: 6
        }
    }
}
one();
```

- Pri sabiranju, prvo se vrednosti varijabli **a**, **b** i **c** traže u dosegu funkcije **three()** gde se vrši računanje.
- U tom dosegu pronalazi se varijabla **c=3**, a varijabli **a** i **b** nema.
- Zatim se ide na sledeći doseg u lancu (doseg funkcije **two()**) i tu se pronalazi varijabla **b=2**, a varijable **a** nema.
- Konačno, pretražuje se doseg **one()** i u njemu se nalazi varijabla **a**=1.
- Dakle rezulatat je: a=1+b=2+c=3 =6

Šta će ovde biti ispisano?

```
function one() {
    var a = 1;
    two();
    function two() {
        var b = 2;
        three();
        function three() {
            alert(a + b + c); // ??
        }
    }
}
one();
```

```
function one() {
    var a = 1;
    var b = 2;
    two();
    function two() {
        var c = 3;
        three();
        function three() {
            alert(a + b + c); // ??
        }
    }
}
one();
```

Ništa neće biti ispisano? Zašto?
Nigde u kodu ne postoji deklaracija C

Ispis?
6

Zatvaranje

- "JavaScript zatvaranja su jedna od velikih životnih misterija." [Fogus]
- Daglas Krokford: ***Unutrašnja funkcija uvek ima pristup varijablama i parametrima svoje spoljašnje funkcije, čak i nakon što spoljašnja funkcija završi svoje izvršavanje …*** U žargonu kaže se: ***funkcija "pamti mesto na kome je rođena".***
- U nekim (ne svim) primerima u nastavku je usvojena interna (ne odnosi se generalno na JS) konvencija da imena svih varijabli koje su zapamćene zatvaranjima pišu velikim slovima.

Funkcija prve klase je funkcija koja **može :**

1. **Biti smeštena u promenljivu:**

```
var greet = function() {
        alert("Hello!");};
greet();  //ispis je?
```

Hello!

2. da primi drugu funkciju kao argument i/ili da vrati drugu funkciju ka povratnu vrednost.

```
var greet = function() {
   alert("Hello!");};
function saySomething(func) {
func();}
saySomething(greet); //Ispis?
```

Hello!

Greet je funkcija prve klase jer:

- **je dodeljena promenljivoj** (vidi se sa: var greet).
- **je prosleđena kao argument drugoj funkciji** (što se dešava u saySomething(greet)).
- Mogao je ispis i sa: greet()

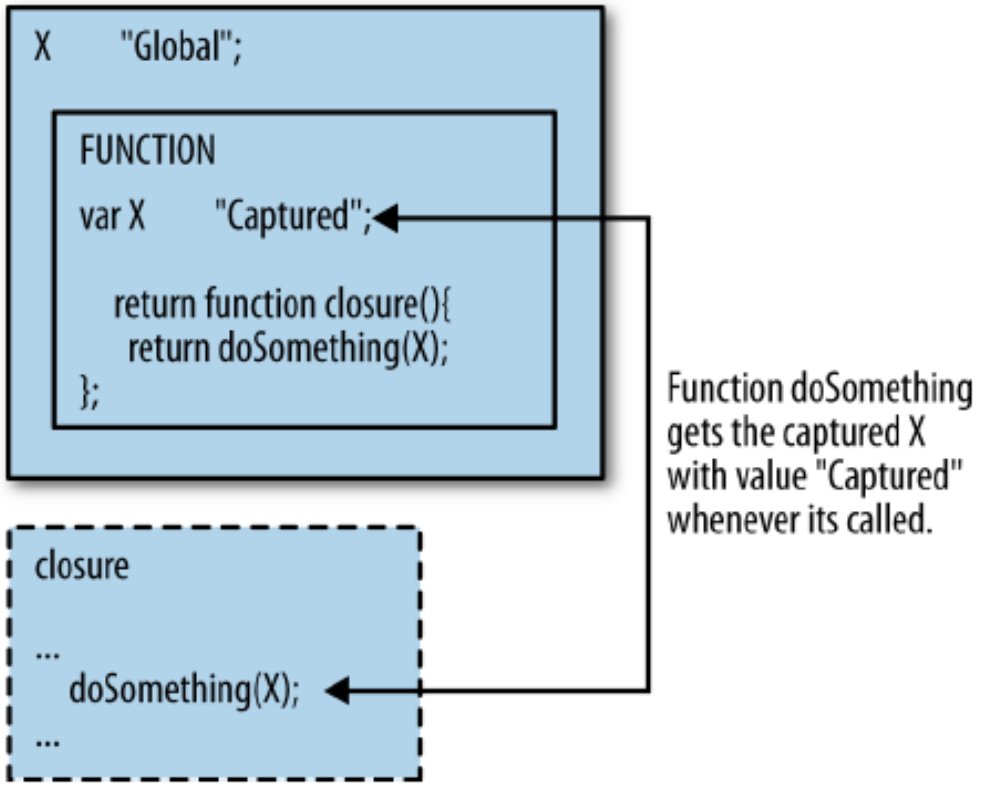
Šta je zatvaranje:

```
var X = 'GLOBAL'

function FUNCTION (){
var X = 'Captured'
  return function closure(){
    return function doSomething() {
        alert (' X u doSomething(): ');
        alert (X);
    }
  }
}
FUNCTION()()()
//Ispis?
```

```
X u doSomething():
Captured
```

Zatvaranje je **funkcija** koja "hvata" vrednost na mestu na kome je ta vrednost "rođena".

Ime **`X`** postoji na dva mesta: u globalnom dosegu gde mu je dodeljena vrednost **`„Global"`**, i u dosegu funkcije **`FUNCTION`** gde je deklarisano ključnom reči **`var`** i gde mu je dodeljena vrednost **`"Captured".`**
Funkcija **`FUNCTION`** vraća funkciju **`closure()`** koja vraća funkciju **`doSomething`** sa argumentom **`X`**.
Svaki put kada se pozove funkcija **`doSomething`** ona pristupa"uhvaćenom" **`X`** sa vrednošću**`"Captured"`**`zato što je varijabla X deklarisana.`



Šta se dešava prilikom različitih poziva?

1.Poziv FUNCTION();

NIŠTA SE NE ISPIŠE. ZAŠTO?

- Poziva se funkcija FUNCTION(), koja vraća unutrašnju funkciju closure, ALI JE NE POZIVA (ništa ne izvršava jer rezultat nije dodeljen promenljivoj niti pozvan).

2.Poziv FUNCTION()();

NIŠTA SE NE ISPIŠE. ZAŠTO?

- FUNCTION() se izvršava i vraća closure.
- closure() se odmah poziva, što vraća još jednu funkciju doSomething, ali se ona **ne POZIVA odnosno ne izvršava**.

3.Poziv FUNCTION()()();

- FUNCTION() se poziva i vraća closure.
- closure() se poziva i vraća doSomething.
- doSomething() se poziva, pa se tada izvršava alert(), i prikazuje se vrednost X (koja je 'Captured').

Simulacija zatvaranja: primer pamćenja lokalnih varijabli

```
CAPTURED = "Ćao, ja sam CAPTURED iz dosega Global";
function staBeseLokal() {
var CAPTURED = "CAPTURED, lokalno iz funkcije";
   return function() { // anonimna funkcija zatvaranja
// ovde nije deklarisana varijabla CAPTURED
   return "Ćao ja sam: " + CAPTURED;
};
}
var reportLocal = staBeseLokal();
alert (reportLocal());  // Ispis?
alert (CAPTURED);       // Ispis?
```

- Ćao ja sam: CAPTURED, lokalno iz funkcije
- Ćao, ja sam CAPTURED iz dosega Global

Zatvaranje i ambijent

- Primer (već smo ga videli)

```
var x = 10;

function foo(){
var y = 20;// slobodna varijabla
function bar(){
    var z = 15;// slobodna varijabla
    return x + y + z;
    }
    return bar;
}
foo();
```

Kako izgleda ambijent$_1$ - global

```
GlobalEnvironment =
x:10
```

Kako izgleda ambijent$_2$ - foo

```
fooEnvironment = {
    y:20, /* varijabla deklarisana u foo */
    bar:'<func>' /* funkcija deklarisana u  foo */
}
```

Kako izgleda ambijent$_3$ - bar

```
barEnvironment ={
z:15 /* varijabla deklarisana u bar */
}
/* pokazivač na roditeljski ambijent (bar je
deklarisana u foo*/
```

Natrag na kod: šta se tu dešava

```
var x = 10;

function foo(){
var y = 20; // slobodna varijabla za bar()
function bar(){
    var z = 15; // slobodna varijabla za foo()
    return x + y + z;
    }
    return bar;
}
var test = foo();
alert(test) // ispis:?
 kod funkcije bar
alert(test()) // ispis:?
 45
```

Razlika VAR i LET

Blokovski opseg znači da je promenljiva vidljiva samo unutar {} bloka u kojem je definisana.
Primer sa let (koji **ima** blokovski opseg):

```
{
  let x = 10;
  }
  alert(x);
```

Rezultat?
ništa

```
{
var x = 10;
}
alert(x);
```

Rezultat?
10

3.čas

Zatvaranje: slobodne varijable$_1$

Primer:

```
function numberGenerator(){
    var num = 1;
    function checkNumber(){
    alert(num);
    }
  num++;
return checkNumber;
}

var number = numberGenerator();
number(); //Rezultat?
```

2

U ovom primeru funkcija **numberGenerator**kreira lokalnu varijablu **num**(broj) i funkciju **checkNumber** (ispisuje **num** na konzolu) u kojoj je **num**“slobodna” varijabla .

Funkcija **checkNumber** nema svojih lokalnih varijabli, a ipak zbog zatvaranja ima pristup varijablama koje su unutar funkcije **numberGenerator**.

Zbog toga ona može da koristi varijablu **num** deklarisanu u funkciji **numberGenerator** da na konzolu ispiše “ispravnu” vrednost*i nakon što* je funkcija **numberGenerator** završila sa izvršavanjem (uradila **return**).

Zatvaranje: slobodne varijable$_2$

```
function kaziZdravo(){
      var kazi = function(){
              alert(zdravo);}
var zdravo ='Zdravo,narode!';
   return kazi;
}
var kaziZdravoZatvaranje =
  kaziZdravo();
kaziZdravoZatvaranje();
Rezultat:

'Zdravo, narode!'
```

Ovde je varijabla **zdravo** definisana *nakon* anonimne funkcije koja ispisuje njenu vrednost na konzolu, a ipak ta funkcija ima pristup **`zdravo`** varijabli. To je zato što je varijabla **`zdravo`** već definisana u "dosegu" funkcije **`kaziZdravo()`** u vreme kreiranja funkcije **`kaziZdravo()`** i time je varijabla **`zdravo`** učinjena dostupnom u vreme kada se anonimna funkcija izvršava.

Zatvaranje: Misteriozni kalkulator 1

```
function mysteriousCalculator(a, b){
  var mysteriousVariable = 3;
  return {
    add:function(){
      var result = a + b + mysteriousVariable;
      alert ("Zbir je: " + result);
      return toFixedTwoPlaces(result);
    },
       subtract:function(){
      var result = a - b -mysteriousVariable;
      alert(" Razlika je : " + result)
      return toFixedTwoPlaces(result);
    }
  }
}
function toFixedTwoPlaces(value){
  return value.toFixed(2);
}
```

mysteriousCalculator je u globalnom dosegu i vraća dve funkcije (sabiranje tri vrednosti i oduzimanje tri vrednosti).

Zatvaranje: Misteriozni kalkulator$_2$

```
// pozivanje
var myCalculator = mysteriousCalculator(10,2);
myCalculator.add() // rezultat: ?
myCalculator.subtract() // rezultat: ?
```

Rezultat:

Zbir je 15

Razlika je 5

Prototipski lanac i razrešavanje identifikatora

- JavaScript je jezik prototipski po prirodi: kada je reč o nasleđivanju, sve u JS-u, izuzev `null` i `undefined`, je tipa `object`.
- ***Prototipski lanac*** (***prototype chain***) je struktura koja predstavlja **lanac nasleđivanja objekta**.
- JavaScript endžin, pri pokušaju da pristupi nekom svojstvu u tipu `object`, to radi tako što pokušava prvo da ga razreši u samom objektu.
- Ako u tom samom objektu nema svojstva sa traženim imenom, endžin dalje pretražuje ***uz prototipski lanac*** sve dok ne nađe traženo svojstvo (u bilo kom objektu u lancu) ili dok ne ne dođe do kraja lanca (korenski tip) u kom slučaju identifikator nije razrešen.

Prototipski lanac i razrešavanje identifikatora

- Za razrešavanje identifikatora JS endžin koristi:

***lanac dosezanja* (*scope chain*)** i

***prototipski lanac* (*prototype chain*)**

- Postavlja se pitanje *na koji način* (kojim redosledom ih koristi)
- Odgovor je:
  - **prvo** koristi **lanac dosezanja** da **locira** `object`.
  - Kada je `object` **nađen**, prolazi se kroz **prototipski lanac** tog `object`-a da se **locira ime svojstva**.

Primer

```
var bar = {};

function foo() {
    bar.a = 'Postavljeno iz foo()';
    return function inner() {
                alert(bar.a);
                 }
}

foo()(); // Ispis: ?
Postavljeno iz foo()
```

Kreira svojstvo **a** u globalnom objektu **bar** i dodeljuje mu vrednost string: `Postavljeno iz foo()`

Endžin gleda u Scope chain i, kao što je i očekivano, nalazi objekat **bar** u globalnom kontekstu.

Zatim pretražuje objekat **bar** i u njemu (u samom objektu **bar**) nalazi svojstvo sa imenom **a**.

Primer

Malo pre je bilo: bar.a

```
var bar = {};
function foo() {
    Object.prototype.a =
'Postavljeno iz prototype';
        return function inner() {
        alert(bar.a);
    }
}
foo()(); // Ispis:?
Postavljeno iz prototype
```

U vreme izvršavanja, poziva se funkcija **`inner()`** koja pokušava da razreši **`bar.a`** tako što u lancu dosega traži **`bar`**.

Funkcija **`inner()`** nalazi **`bar`** u globalnom kontekstu i pretražuje **`bar`** da nađe ime **`a`**.

U objekat **`bar`** nije postavljeno ništa sa imenom **`a`**, pa endžin prolazi kroz prototipski lanac objekta i nalazi da je **`a`** postavljeno u `Object.prototype`-u i ispisuje ono što tamo stoji, a to je string `Postavljeno iz prototype`

Podsetnik za : ()()

Poziv sa **dvostrukim zagradama** foo()(); u JavaScript-u znači:
1.Prvi deo foo() → Poziv funkcije foo, koja vraća drugu funkciju.
2.Drugi deo () → Odmah poziv funkcije koju je foo() vratila.

Kada se funkcija poziva sa samo jednim parom zagrada **()**, to znači da se **funkcija izvršava sama za sebe**, dok se poziv sa dva para zagrada**()()** koristi kada se **rezultat jedne funkcije prosleđuje kao argument drugoj funkciji**.

```
function add(a, b) { return a + b; }
function double(x) { return x * 2; }
alert(add(2, 3)); // Output: 5
alert(double(add(2, 3))); // Output: 10
```

Funkcija **add** se prvo izvršava sa argumentima **2** i **3**, što rezultira vrednošću **5**. Zatim se ta vrednost prosleđuje funkciji **double**, koja je izvršava i vraća rezultat **10**.

Kada se koristi Zatvaranje

- Zatvaranje (closure) u JavaScriptu je koncept koji se odnosi na sposobnost funkcije da pristupi svojstvima i promenljivima iz spoljnih opsega (scope) u kojima je ta funkcija definisana, čak i nakon što je spoljni opseg završio svoje izvršavanje.
- Promenljive ostanu dostupne za korišćenje unutar funkcije, čak i nakon što spoljni opseg prestane da postoji.

```
function outer() {
    var x=10;
    function inner() {
       alert(x);
    }
    return inner;
}
var myFunction=outer();
myFunction(); //Output: ?

10
```

Inner je zatvorena unutar funkcije **outer**. Kada se **outer** funkcija poziva, ona definiše promenljivu **x** i definiše unutrašnju funkciju **inner**.
Kada se **inner** funkcija vrati iz **outer** funkcije i dodeli promenljivoj **myFunction**, ona zadržava pristup promenljivoj **x** iz spoljnog opsega, čak i nakon što je **outer** funkcija završila sa izvršavanjem.
Ovo se dešava zahvaljujući zatvaranju, što omogućava da se referenca na promenljivu **x** sačuva unutar unutrašnje funkcije **inner**.

Leksički doseg i zatvaranje: Primer

```
function init() {
  var name = 'Mozilla';
  function displayName()
     {  alert(name); }
  displayName();
}
init();
```

Rezultat?

Mozilla

Funkcija **`init()`** kreira lokalnu varijablu sa imenom **`name`** i funkciju sa imenom **`displayName()`**.

Funkcija **`displayName()`** je unutrašnja funkcija (zatvaranje) koja je definisana unutar funkcije **`init()`** i dostupna je samo u telu funkcije **`init()`**.

Funkcija **`displayName()`** **nema svojih lokalnih varijabli**.

Međutim, kako unutrašnje funkcije imaju pristup varijablama spoljašnjih funkcija, funkcija **`displayName()`** može da pristupi varijabli **`name`** deklarisanoj u roditeljskoj funkciji **`init()`**.

Leksički doseg i zatvaranje: isti Primer , malo drugačije

```
function makeFunc() {
  var name = 'Mozilla';
  function displayName() {
    alert(name);
  }
  displayName;
}

var myFunc = makeFunc();
myFunc(); // ispis ?
Ništa
Mozilla
```

Razlika:

Umesto : **displayName()**

sada je **displayName**

displayName nije funkcija pa se ništa ne izvršava umesto displayName staviti return displayName da bi se nešto vratilo

```
function init() {
    var name = 'Mozilla';
    function displayName() {
        alert(name); }
    var name1 = 'Mozilla1';
    function displayName1() {
        alert(name1); }
    var name2 = 'Mozilla2';
    function displayName2() {
        alert(name2); }
    return displayName();
}

var a=init()
a();
```

Šta je ispis?

Mozilla

Šta će biti ispis kada stavimo :
return displayName1?

Mozila1

Šta će biti ispis kada stavimo :
return displayName2?

Mozila2

Znači uvek se ispisuje ono što se sa return vraća. Čak je Visual Studio Code podvukao one funkcije koje se ne vraćaju jer nisu ni korisne

Zatvaranje: Primer 2 - klasika

```
function makeAdder(x) {
    alert("x= " + x);
   return function(y) {
     return x + y;
   };
}

var add5 = makeAdder(5);
var add10 = makeAdder(10);

alert(add5(2));  // Šta će biti ispisano?
alert(add10(2)); // Šta će biti ispisano?
```

X= 5

X= 10

Y=2

Y=2

Ispis: **7**

Ispis: **12**

1. Poziv"makeAdder(5)" , izvršava funkciju makeAdder koja ima u sebi alert i ispisuje na ekran 5 (to je x)

Takodje poziv makeAdder(5) vraća funkciju(y) kao tekst ili sadržaj:

function(y) {

return x+ y;

};

S tim da se zna koliko je x ali se ne ispisuje

Varijabla add5 ne sadrži rezultat, jer je add5 funkcija, a ne vrednost koju možemo prikazati.

2. Poziv **alert (add5(2))** će prikazati rezultat sabiranja 5 , što je vrednost **x** u funkciji **add5 ,**i 2 (vrednost **y** koja je prosleđena funkciji). Dakle prikaza će se 7.

JS engine

- JavaScript engine-i su programi koji izvršavaju JavaScript kod u različitim okruženjima (pregledačima, serverima, ugrađenim sistemima itd.).
- Postoji nekoliko poznatih JavaScript engine-a, koji su razvijeni od strane različitih kompanija.
- **JS engine V8** (Google Chrome, Node.js, Edge, Opera)
  - Razvijen od strane Google-a
  - Koristi **Just-In-Time (JIT) kompajler** za bržu interpretaciju koda

JS endžin V8

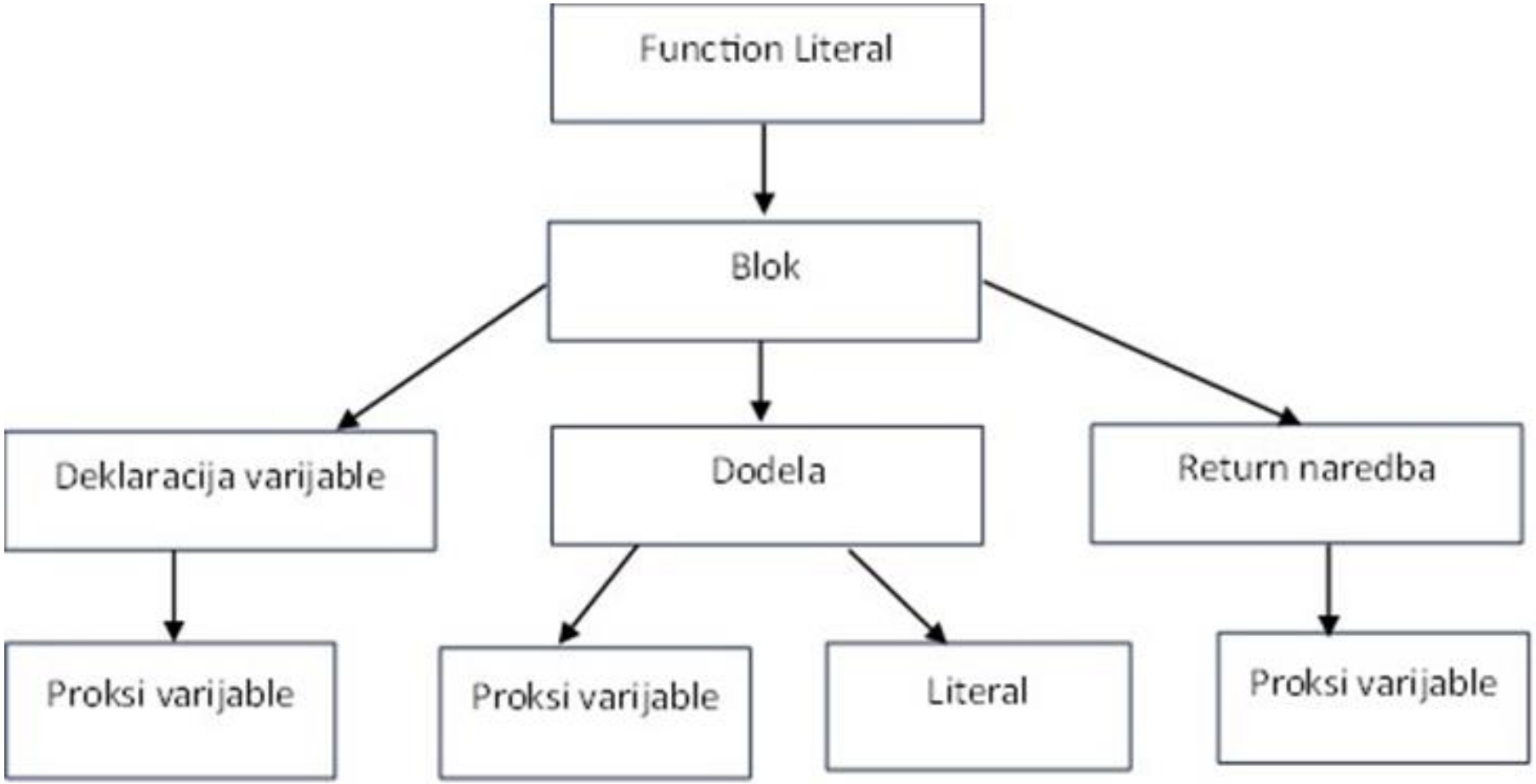
1. Započinje **pribavljanjem izvornog JavaScript koda**.
2. **Parsira izvorni kod** i transformiše ga u apstraktno sintaksno stablo (AST).
3. Na bazi AST, **Ignition** interpreter može da počne da radi svoj posao i da **pravi bajt-kod**.
4. U ovoj tački, endžin počinje da **izvršava kod** i da prikuplja povratne informacije o tipu.
5. Da bi se program ubrzao, bajt-kod se može proslediti optimizujućem kompajleru zajedno sa povratnim informacijama o tipu. Optimizujući kompajler na bazi primljenog pravi neke pretpostavke (o tipu) i **proizvodi visoko optimizovan mašinski kod.**
6. Ako se u nekoj tački izvršavanja ispostavi da je neka od pretpostavki kompajlera postala nekorektna, **kompajler vrši deoptimizaciju i vraća se na interpreter.**

1. Priprema izvornog koda

- Kada se kod pribavi, treba ga modifikovati u oblik koji kompajler može da razume. Taj proces naziva se **parsiranje** i sastoji se iz dva dela: **skenera** i **parsera**.
  - **Skener** prima JavaScript fajl sa izvornim kodom i konvertuje ga u listu poznatih tokena.
  - **Parser** prima rezultat skeniranja i kreira apstraktno sintaksno stablo (Abstract Syntax Tree, AST) što je reprezentacija izvornog koda u obliku stabla.

```
function saberi()
{   let x = 5;
    let y = 10;
    return x + y; }
```

2. Primer AST stabla



- **Function Literal** – Predstavlja definiciju funkcije.
- **Blok** – Unutrašnjost funkcije, sadrži njene naredbe.
- **Deklaracija varijable** – Definiše promenljive koje će funkcija koristiti.
- **Proksi varijable** – Varijable koje funkcija referencira.
- **Dodela (Assignment)** – Operacija dodeljivanja vrednosti varijablama.
- **Proksi varijable** – Promenljiva kojoj se dodeljuje vrednost.
- **Literal** – Konkretna vrednost (broj, string itd.).
- **Return naredba** – Vraća vrednost iz funkcije.
- **Proksi varijable** – Promenljiva čija se vrednost vraća.

```
function saberi()
{   let x = 5;
    let y = 10;
    return x + y; }
```

- **Function Literal** → saberi
- **Blok** → sadrži naredbe
- **Deklaracija varijable** → x , y
- **Dodela** → x = 5, y = 10
- **Return naredba** → return x + y

Just-in-Time (JIT) kompilacija

1.Kompilacija - izvorni kod se transformiše u mašinski kod pre započinjanja izvršenja programa.

2. Interpretacija - svaka linija koda se prevodi i odmah izvršava.

3. Kombinacija ova dva pristupa - **Just-in-Time (JIT) kompilacija.**

- Endžin V8 koristi **interpretaciju kao bazni metod**, ali ima i mogućnost da **detektuje funkcije koje se koriste češće i da ih kompajlira** koristeći informacije o tipu iz prethodnih izvršavanja.

3. Ignition interpreter

Inicijalno prima AST i generiše **bajt-kod**

[Generated Bytecode for function: saberi]

- 0x123456789 0x0d010203 LdaSmi 5 – učitava broj 5 u registar
- 0x123456790 0x0d020304 StaGlobal x – čuva 5 u promenljivoj x
- 0x0d050607 LdaSmi 10 - učitava broj 10 u registar
- 0x123456792 0x0d08090a StaGlobal y - čuva 10 u promenljivoj y
- 0x123456793 0x0d0b0c0d LdaGlobal x - učitava vrednost x
- 0x123456794 0x0d0e0f10 Add y – sabira x + y
- 0x0d111213 Return – vraća rezultat

- Ovo pokazuje kako V8 interpretuje naš kod i prevodi ga u niz jednostavnih instrukcija za brže izvršavanje

Izvršenje

U V8 engine-u, tabela rukovalaca može izgledati ovako:

Bajt-kod ključ	Instrukcija	Rukovalac (Handler)
0x01	LdaSmi 5	`DoLdaSmi()`
0x02	StaGlobal x	`DoStaGlobal()`
0x03	Add	`DoAdd()`
0x04	Return	`DoReturn()`

Nakon generisanja bajt-koda, **Ignition** interpretira instrukcije koristeći tabelu rukovalaca u kojoj se rukovaocu pristupa pomoću bajt-kod ključeva.

Tokom izvršavanja, Ignition gleda bajt-kod ključeve i poziva odgovarajuće rukovaoce:

1.0x01 → DoLdaSmi() učitava broj.
2.0x02 → DoStaGlobal() upisuje u promenljivu.
3.0x03 → DoAdd() sabira vrednosti.
4.0x04 → DoReturn() vraća rezultat.

On nalazi odgovarajuću funkciju rukovaoca i izvršava je za zadate argumente.

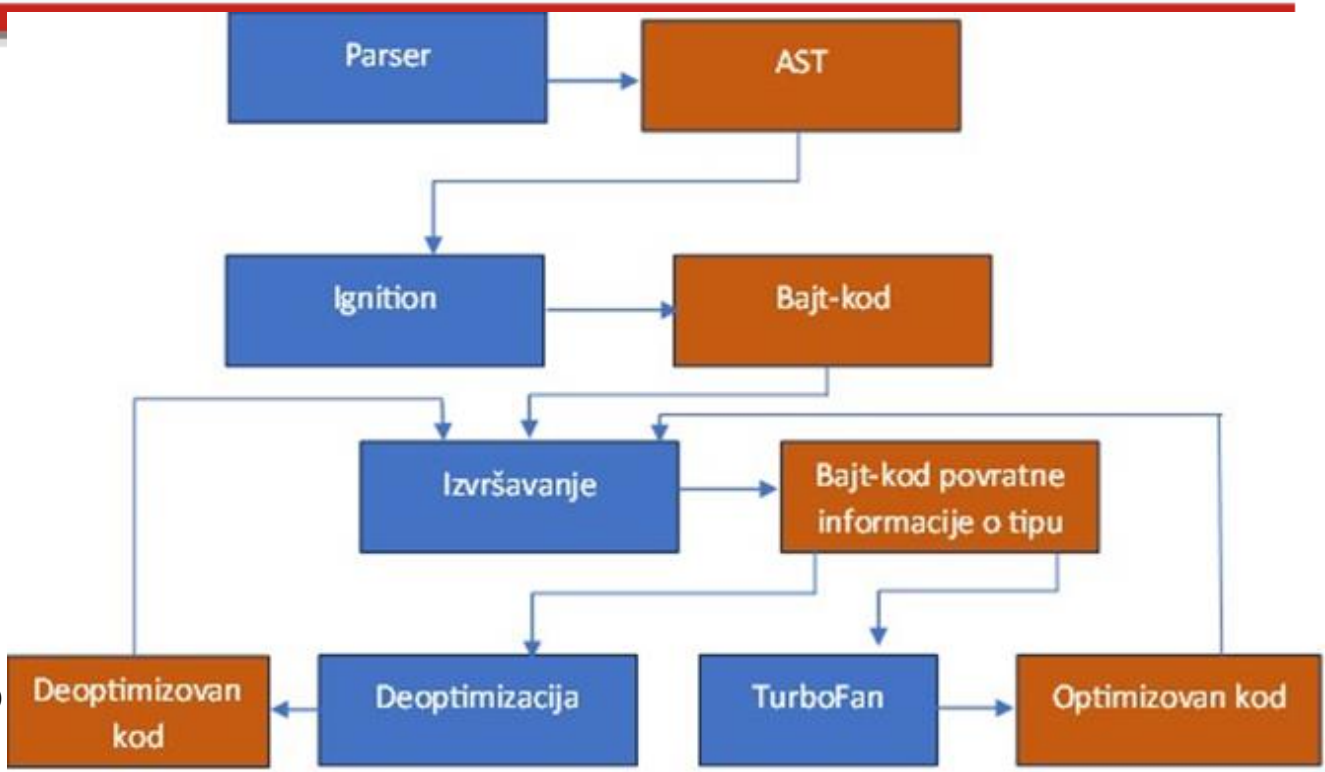
Kompajler Turbofan

- **TurboFan** je optimizovani Just-In-Time (JIT) kompajler u **V8 JavaScript engine-u** koji **pretvara JavaScript kod u visoko optimizovan mašinski kod** radi poboljšanja performansi.
- Ako funkcija postane dovoljno interesantna biće optimizovana u kompajleru **Turbofan** da bi se ubrzala
- Međutim, niko ne garantuje da se informacije o tipu neće u budućnosti promeniti. Ukoliko do promena dođe, radi se proces zvani *deoptimizacija* koji odbacuje optimizovani kod, vraća se na interpretirani kod, nastavlja izvršavanje i ažurira povratnu informaciju o tipu.

```
function saberi(a, b) {

return a + b; }
```

Ako se saberi(5, 10) često izvršava sa **brojevima**, TurboFan: ✓ Zaključi da su a i b brojevi (ne stringovi, ne objekti).
✓ Generiše brži mašinski kod specijalizovan za brojeve.
Ali, ako kasnije pozovemo saberi("5", 10), TurboFan vidi promenu tipova i **deoptimizuje kod** (vrati ga u Ignition).

Rezime



V8 endžin radi sledeće:

1. Započinje pribavljanjem izvornog JavaScript koda
2. Parsira izvorni kod i transformiše ga u apstraktno sintaksno stablo (AST).
3. Na bazi AST, **Ignition** interpreter može da počne da radi svoj posao i da pravi bajt-kod.
4. U ovoj tački, endžin počinje da izvršava kod i da prikuplja povratne informacije o tipu.
5. Da bi se program ubrzao, bajt-kod se može proslediti optimizujućem kompajleru zajedno sa povratnim podacima. Optimizujući kompajler na bazi primljenog pravi neke pretpostavke (o tipu) i proizvodi visoko optimizovan mašinski kod.
6. Ako se, u nekoj tački izvršavanja, ispostavi da je neka od pretpostavki kompajlera postala nekorektna, kompajler vrši de-optimizaciju i vraća se na interpreter.

JS strukture podataka i strukturni tipovi

Objekat

Objekat: Tip `object`$_1$

- Tip **`object`** (**`plain object`**) je strukturni tip koji služi za skladištenje kolekcija (neuređenih) različitih vrednosti i skladištenje drugih složenijih entiteta.
- **Pojedinačnim elementima kolekcije može se pristupati putem ključa.**
- Objekat ima ***ime*** i ***svojstva***
- ***Svojstvo*** ima ***ključ/ime*** (tip **`String`** ili **`Symbol`**) i ***vrednost*** (bilo koji tip)

```
let korisnik = { // objekat sa imenom korisnik
  ime: "Petar",  // ključ "ime", vrednost "Petar"
  starost: 30    // ključ "starost" ,vrednost 30
};
```

Objekat: Tip `object`$_2$

- U objektima, **imena svojstava** su **uvek** stringovi. Ako se koristi bilo koji drugi tip kao ime svojstva, on će uvek prvo biti konvertovan u string.

- **Vrednost svojstva** može da bude **bilo koja JS vrednost**, uključujući i **funkciju**.

Objekat: implementacija objektne paradigme u JS-u

- Objekat je osnovni koncept objektno orijentisanog programiranja
- U JavaScript-u implementacija objektnog programiranja se može posmatrati kroz tri sloja
  1. Pojedinačni objekat
  2. Prototipski lanac objekata
  3. Konstruktori i nasleđivanje konstruktora

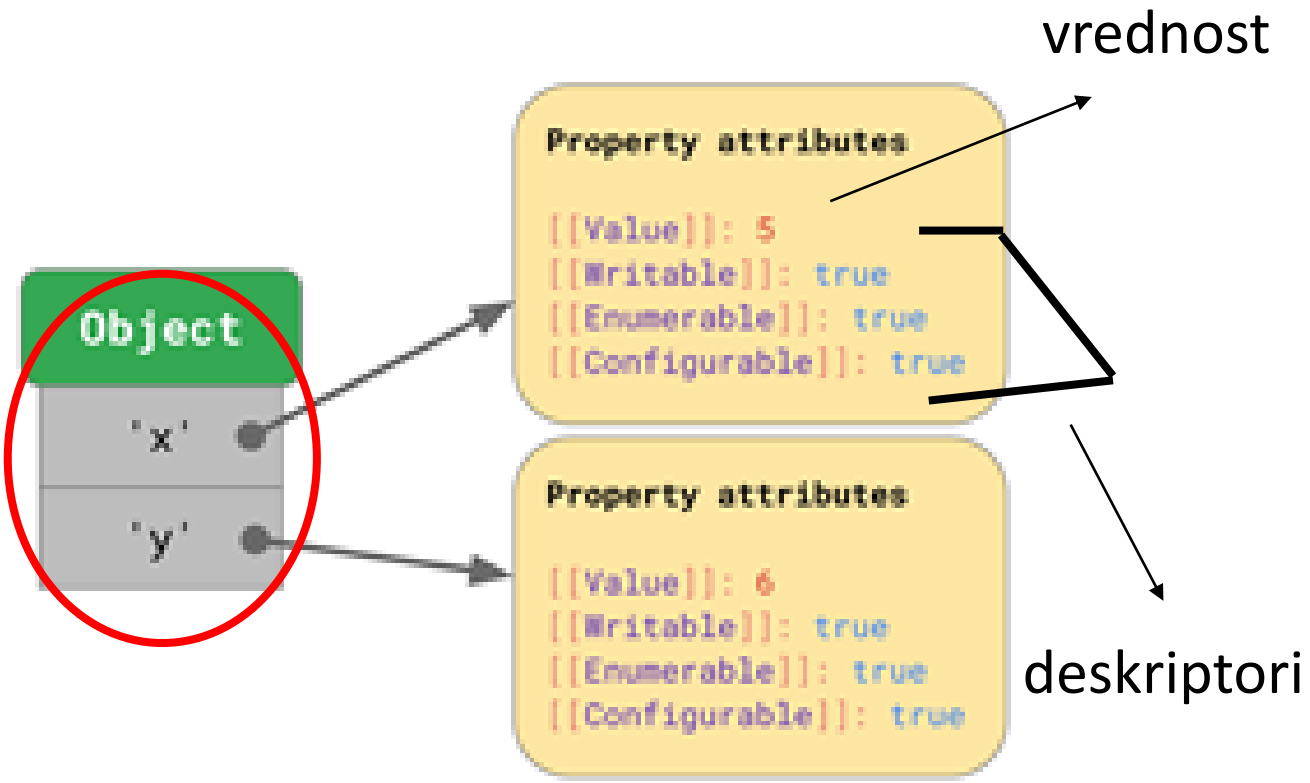
Sloj 1: Pojedinačni objekat

- U suštini, objekat se sastoji od parova (ključ, vrednost) a taj par naziva se svojstvo (*property*) objekta.
- Važi ograničenje da je ključ uvek tekstualni string.
- Vrednost svojstva može da bude bilo koja JavaScript vrednost, uključujući i funkciju.
  - Metode objekta su svojstva objekta koja za vrednosti imaju funkcije.

Objekat: Reprezentacija u memoriji

- ECMAScript specifikacija u suštini objekte definiše kao rečnike sa ključevima koji pokazuju na atribute svojstva: imena svojstava koja pokazuju na ***vrednosti*** i ***deskriptore*** tih svojstava.

```
object = {
    x: 5,
    y: 6,
};
```



Objekti: deskriptori svojstava

- Deskriptor svojstava (zvani i *deskriptor podatka*) obuhvata još tri krakteristike: `writable`, `enumerable`, i `configurable`.
- Vrednosti koje ove tri karakteristike mogu da uzmu su `true` ili `false`.
- **Deskriptori diktiraju šta se sve može raditi sa svojstvom koje opisuju**

- **Kreiranje objekta i prikaz**

1.Obično kreiranje

```
var myObject = {a:2};
alert(myObject.a);
```

Rezultat:2

2. Kreiranje sa deskriptorima:

```
var myObject = {};
Object.defineProperty(myObject, "a", { value:2});
alert(myObject.a);
```

Rezultat:2

Object.defineProperty() je metod u JavaScriptu koji se koristi za definisanje nove osobine direktno na objektu ili modifikaciju postojeće osobine na objektu. Ovaj metod omogućava precizno definisanje deskritora

Deskriptor: `writable`

- Kontroliše mogućnost da se promeni vrednost bilo svojstva objekta (u režimu strict (dodati kao prvi red:`"use strict";` dobija se poruka o grešci TypeError):

```
var myObject = {};

Object.defineProperty( myObject, "a", {
  value:2,
  writable:false, // nije writable!
  configurable:true,
  enumerable:true
} );

myObject.a  = 3
alert(myObject.a)  // ispis?
```

2

Writable:true

Koji je rezultat?

3

Deskriptor: `configurable`

- Sve dok svojstvo ima karakteristiku da je konfigurabilno (vrednost deskriptora `configurable` je `true`), moguće je modifikovati definiciju deskriptora pomoću `defineProperty(..)`.
- `configurable:false` znači da više nema menjanja
- Promena configurable na `false` je **jednosmerna akcija i ne može se opozvati!**
- Takođe, `configurable:false` sprečava mogućnost da se koristi `delete` operator za uklanjanje postojećeg svojstva.

Primer:

```
const osoba = {
  ime: 'Marko'
};
alert(osoba.ime)
// Definišemo deskriptor za 'ime' sa configurable: false
Object.defineProperty(osoba, 'ime', {
  configurable: false
});

// Pokušavamo da obrišemo svojstvo 'ime'
delete osoba.ime;  // Ovo neće uspeti, jer je 'configurable' postavljeno na false

alert(osoba.ime) //Ispis?
```

Marko

- Kada se u prethodnom primeru stavi:

configurable: true

- Ispis je:

Marko

undefined

Deskriptor: `enumerable`

- Ova karakteristika kontroliše da li će se svojstvo pojavljivati u konstruktima sa nabrajanjima svojstva u objektu, kao što je `for..in` petlja.
- Vrednost svojstva `false` znači da se ono neće pojavljivati u takvim nabrajanjima, iako je inače potpuno dostupno.
- Naravno, vrednost `true` znači da će se pojavljivati u takvim nabrajanjima.

Primer:

```
const osoba = {
  ime: 'Marko',
  prezime: 'Marković'
};

// Definišemo deskriptor za 'prezime' sa enumerable: false
Object.defineProperty(osoba, 'prezime', {
  enumerable: false
});

// Ispisujemo sve svojstva objekta koristeći for...in
for (let kljuc in osoba) {
  alert(kljuc);  // Ovo će ispisati samo 'ime', a ne 'prezime'
}   //Ispis?
```

ime

- Sada staviiti:

enumerable: true //Ispis?

ime

prezime