

03.03.2025.

Funkcije

- **I deo**
 - O jeziku JavaScript
 - Sintaksa jezika JS i struktura koda JS programa
 - Varijable, izrazi, operatori, tipovi
 - Kontrola toka programa
 - **Funkcije**
- **II deo**
 - **Izvršavanje JS koda**
- **III deo**
 - JS strukture podataka i strukturni tipovi: objekat, niz

Šta je funkcija

1. Funkcija je modul koda koji izvršava specifičan zadatak.
2. Funkcija, obično (ne nužno), "prima" podatke, obrađuje te podatke i "vraća" rezultat.
3. Jednom napisana funkcija može se pozivati više puta.
4. Funkcija se može pozivati i iz druge funkcije.

Funkcije u JavaScript-u

- Funkcije su osnovni “gradivni elementi programa” u jeziku JS. Omogućuju da se **kod poziva više puta** bez ponovljenog pisanja. Kao kada koristite neku matematičku formulu da nešto sračunate više puta, za različite vrednosti “opštih brojeva” u formuli (prosek ocena).
- Funkciji se moraju proslediti neke ulazne vrednosti i one se prosleđuju (najčešće) putem liste parametara.
- Kao rezultat svog rada, funkcija vraća jedan izlaz
- U JS-u **funkcije su građani prvog reda** što, između ostalog, znači da se sa njima može raditi sve što se radi sa bilo kojom drugom varijablom – **funkcije mogu da se prosleđuju drugim funkcijama kao ulazi i funkcija može da vrati drugu funkciju kao izlaz.**

Funkcije za interakciju sa programom

- Za obezbeđivanje interakcije JS nudi gotove, ugrađene funkcije `alert`, `prompt` i `confirm`.

- `alert(poruka)` - ispisuje sadržaj promenljive `poruka`
`alert("Hello");`

- `result = prompt(title, [default])` – ispisuje tekst i **prihvata ulaznu vrednost**

`title` – tekst koji se ispisuje

`default` – pretpostavljena vrednost (neobavezno)

```
let age = prompt('Koliko Vam je godina?', 100);
```

```
alert(`Vama je ${age} godina!`);
```

- `result = confirm(question)` – prikazuje modalni prozor **sa pitanjem (question) i dva dugmeta**: OK i Cancel. Ako se odabere OK, vrednost promenljive `result` je `true`, ako se odabere Cancel, vrednost je `false`.

```
let isBoss = confirm("Da li ste Vi Gazda?");
```

```
alert( isBoss ); // true ako se odabere dugme OK, inače false
```

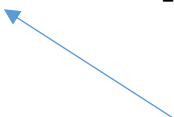
```
let odgovor = confirm("Da li ste sigurni da želite nastaviti?");  
if (odgovor) {  
    alert("Kliknuli ste OK!");  
} else {  
    alert("Kliknuli ste Cancel!");  
}
```

Varijabla i funkcija

- “Regularne” vrednosti kao što su **string** i **numerik** predstavljaju *podatke* koji se dodeljuju varijablama.
- **Funkcija** se može posmatrati kao *akcija* opisana/predstavljena odgovarajućim **izvornim kodom**.
- **Izvorni kod** nije ništa drugo do vrednost tipa **string**, a rezultat izvršavanja funkcije može da bude različitog tipa.
- Opis funkcije (izvorni kod) se može prosleđivati kao varijabla i izvršavati kada mi to želimo.
- U tom smislu, u JavaScript-u **funkcija je varijabla**.

Primer: Varijabla u kojoj je zapisana funkcija

```
let funcs = []; // ovde se deklariše da je funcs šta ?
//niz
for (let i = 0; i < 3; i++) {
    funcs[i] = function() { // ovde se deklariše funkcija
        // svaka treba da na konzoli ispiše svoju vrednost indeksa.
        alert(" Ja sam " + " funcs [" + i + "]. Moj rezultat je: " + i);
    };
}
// Šta znači ovaj kod a šta znači ovo : funcs [" + i + "]?
// Kreiraju se 3 funkcije i smeštaju se u niz funcs
for (let j = 0; j < 3; j++) {
    let vratio = funcs[j](); // moglo je da stoji i samo funcs[j]();
}
// Sada pokrećemo svaku funkciju da uradi ono što treba
alert (" U funcs [0] piše : " + funcs[0])
alert (" U funcs [1] piše : " + funcs[1])
alert (" U funcs [2] piše : " + funcs[2])
// Sada ispisujemo elemente niza funcs da vidimo šta je u njima
```



Zaključak:

- **funcs[0]** je funkcija, a ne broj, pa kada je konvertujemo u string, dobijamo njen kod.
- Ako želimo da dobijemo rezultat funkcije, moramo je **pozvati** sa funcs[0](), a ne samo ispisati funcs[0].
- Ako želimo da svaka funkcija ispiše svoju vrednost i, možemo je promeniti ovako:

```
alert(" U funcs [0] piše : " + funcs[0]()); // sada će ispisati 0  
alert(" U funcs [1] piše : " + funcs[1]()); // sada će ispisati 1  
alert(" U funcs [2] piše : " + funcs[2]()); // sada će ispisati 2
```


Pisanje funkcija

- Videli smo da postoje gotove funkcije (ugrađene) koje možemo da koristimo. **koristimo.Primer?**

```
a=Math.sqrt(81);
```

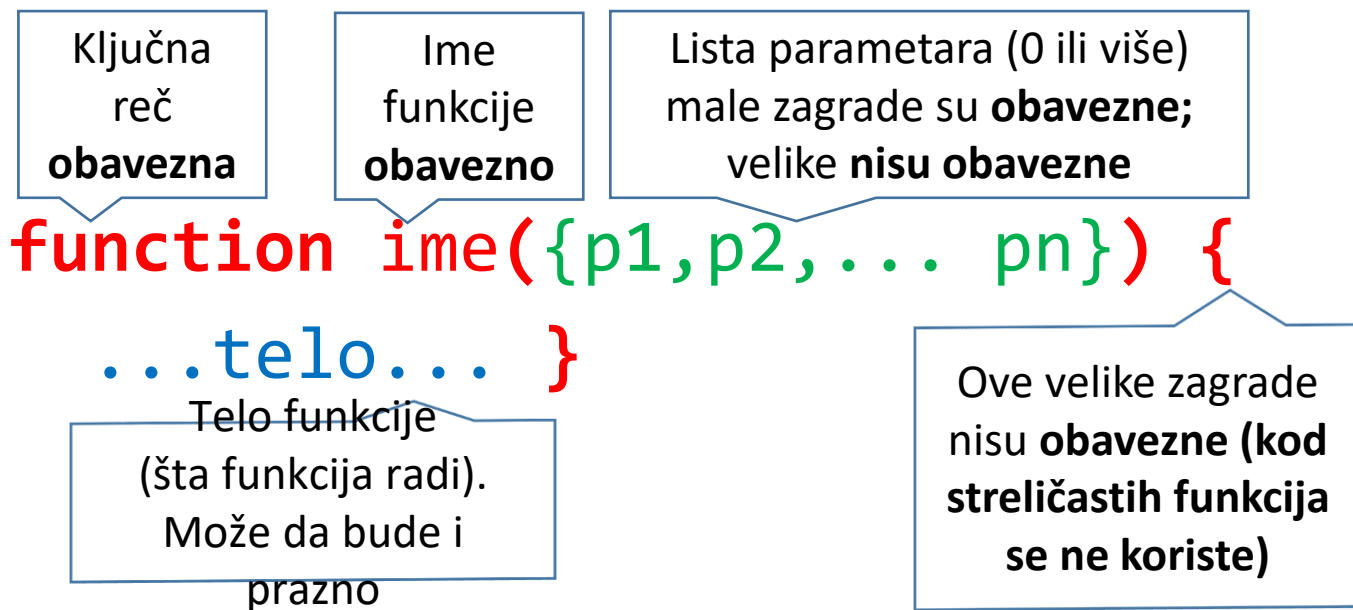
```
alert(a);
```

- Funkcije, naravno, i sami možemo da pišemo.
- Da bi nešto bilo prepoznato kao funkcija, potrebno je to nešto ***proglasiti/deklarirati funkcijom***.
- JavaScript ima tri načina deklarisanja funkcija
 - **Kanonička deklaracija** (Function Declaration) – samostalna deklaraciona naredba
 - **Funkcijski izraz** (Function Expression) – deklaracija unutar izraza
 - **Function** konstruktor (o njemu ćemo kasnije)

Deklaracija funkcije: naredba **function**

Ovo je osnovni, kanonički oblik za deklarisanje funkcije.

U njemu ključna reč **function** kaže da će to biti tip funkcija.



```
function kaziZdravo() {  
    alert('Pozdrav svima!');  
}
```

- **ime** je identifikator funkcije. Pomoću njega se može referisati funkcija (recimo, pri pozivanju). Posebno je zgodno za debugovanje jer se na steku vidi tačno o kojoj je funkciji reč.
- Postoje i anonimne funkcije (funkcije bez imena) i one se sve na steku identifikuju istim imenom **anonymous** (što baš nije neka velika radost za onoga ko debugira program).
- **Parametri** (može i da bude funkcija bez parametara kao u primeru) su mehanizam kojim se funkciji prosleđuju ulazne vrednosti.
- To su placeholder-i u koje se smeštaju argumenti.
- Postoje dve vrste parametara: **pozicioni** i **imenovani** (imenovani su zatvoreni u veliku zagradu).
- U slučaju pozicionih parametara, pri pozivu funkcije redosled navođenja argumenata mora da odgovara redosledu parametara. **Ako su u pitanju imenovani parametri, redosled navođenja je argumenata je proizvoljan.**
- Telo funkcije je kod koji opisuje šta funkcija radi.

Deklaracija funkcije: Funkcijski izraz

- Druga sintaksa za kreiranje funkcije je ***funkcijski izraz*** (*Function Expression*).
- U ovoj sintaksi, funkcija se kreira i eksplicitno dodeljuje varijabli - nije važno kako je funkcija definisana, ona je samo vrednost (string koji sadrži izvorni kod) smeštena u neku varijablu.

```
let kaziZdravo = function () {  
    alert('Pozdrav svima!' )  
}
```

Kako ćemo pozvati funkciju?

kaziZdravo()

 file://

Pozdrav svima!

OK

Deklaracija funkcije: Imenovani funkcijski izraz

- Funkcijski izraz se koristi za deklarisanje anonimne funkcije (funkciju bez imena)
- Funkcijski izraz može da deklariše i neanonimnu funkciju (funkciju sa imenom)
 - To ime je *lokalno*, važi samo u telu funkcije
 - Omogućuje referisanje funkcije imenom u telu funkciji (pogodno za rekurziju)

Imenovani funkcijski izraz: primer

```
let mat = {  
  'faktor': function factorial(n) {  
    alert(n)  
    if (n <= 1) {  
      return 1;  
    } return n * factorial(n - 1);  
  }  
};
```

```
alert(mat.faktor(3))  
Šta će se ispisati?  
Ispis : 3;2;1;6
```

Python:

```
def faktorijel(n):  
    if n == 0:  
        return 1  
    else:  
        return n*faktorijel(n-1)  
print(faktorijel(5))
```

Ovaj JavaScript kod definiše objekat mat, koji sadrži jednu metodu pod imenom 'faktor'. Ta metoda je funkcija factorial(n), koja jednostavno prikazuje vrednost n pomoću alert(n).

Deklaracija funkcije i ime: primer

```
var foo = function(){} // nema napisanog imena
```

```
    alert(foo.name); // Ispiše se : foo
```

```
    alert(typeof foo); // Ispiše se : function
```

```
var foo2 = foo
```

```
    alert(foo2.name) // Ispiše se : foo
```

```
var bar = function baz() {} // ima deklarirano ime baz
```

```
    alert(typeof baz); // Ispiše se : undefined
```

```
    alert(typeof bar); // Ispiše se : function
```

```
    alert(bar.name) // Ispiše se : baz
```

```
    alert(foo === foo2); // Ispiše se : true
```

```
var bar1 = function baz() {}
```

```
    alert(bar1.name) // Ispiše se : baz
```

Pozivanje funkcije bez parametara

- Da bi funkcija uradila svoj posao, ona se mora **pozvati**.
- Ako funkcija nema parametre, poziva se navođenjem svog imena iz koga sledi par malih zagrada
- Primer : poziv funkcije bez parametara:

```
function showMessage() {  
    alert('Pozdrav svima!' );  
}  
showMessage(); // Pozdrav svima!
```


Pozivanje funkcije sa pozicionim argumentima_{1/2}

- Ako funkcija ima parametre, poziva se navođenjem svog imena iz koga sledi par malih zagrada u kojima su navedene vrednosti parametara (argumenti) za taj poziv
- Primer: funkcija sa parametrima – 2 parametra

```
function sendMessage(from, text) {  
  // Parametri: from, text  
  return from + ': ' + text  
}
```

Kako ćemo pozvati funkciju?

```
alert(sendMessage("Petar", "Dobar"));
```

 file://

Petar: Dobar

Pozivanje funkcije sa pozicionim argumentima ^{2/2}

Argumenti: from='Ana', text = 'Zdravo, Soko!'

```
function sendMessage(from, text) {  
    return from + ': ' + text  
}
```

```
let od='Ana', poruka = 'Zdravo Soko!'  
alert(sendMessage(od, poruka));  
Ipisaće se:      Ana: Zdravo, Soko!
```

/ Redosled navođenja argumenata mora da odgovara redosledu parametara:*/*

```
alert(sendMessage(poruka, od));  
Ipisaće se: Zdravo, Soko!: Ana
```

Pozivanje funkcije sa imenovanim argumentima_{1/2}

- Primer: funkcija sa parametrima – 2 parametra

```
function sendMessage({from, text}) {  
  // Parametri: from, text  
  return from + ': ' + text;  
}
```

U čemu je razlika?

Zagradama: {}

Ovde se parametri prosleđuju kao objekat (unutar zagrade {}) čija svojstva imaju ključeve koji odgovaraju imenima parametara. Argumentima se pristupa putem tog ključa pa zato nije važan redosled navođenja argumenata.

Pozivanje funkcije sa imenovanim parametrima_{2/2}

```
function sendMessage({from, text}) {  
    return from + ': ' + text;  
}
```

```
const from='Ana', text = 'Zdravo Soko!'  
alert(sendMessage({from, text}));
```

Šta se dobija kada je poslednji red

 file://

Ana: Zdravo Soko!

```
alert(sendMessage({text, from}));
```

 file://

Ana: Zdravo Soko!

Podrazumevani argumenti

- Pri pozivu funkcije ne moraju se navesti svi argumenti iz liste parametara – moguće je pri deklarisanju funkcije definisati podrazumevanu vrednost koja će biti prosleđena ako se argument ne navede.

- Primer:

```
function sendMessage(from = 'podrazumevana  
osoba', text = 'podrazumevana poruka') {  
    // Parametri: from, text  
    return from + ': ' + text  
}  
alert(sendMessage())
```

⊕ file://

- .

podrazumevana osoba: podrazumevana poruka

- Ako se ne prosledi vrednost za parametar koji nema definisan podrazumevani argument, biće prosleđena vrednost `undefined`. Kako to izgleda?

```
function sendMessage(from ='podrazumevana osoba', text) {  
    // Parametri: from, text  
    return from + ': ' + text  
}  
alert(sendMessage())
```

 file://

podrazumevana osoba: undefined

☐ Don't allow this site to prompt you again

Pozivanje funkcijskog izraza

- Šta će sledeći kod da ispiše?

```
let kaziZdravo = function() {  
  alert(' Pozdrav svima! ');  
};  
let func = kaziZdravo;
```

Ništa

Dodati : `alert('Sadrzaj func: ' + func);` // ispis: ??

 file://

```
Sadrzaj kaziZdravo: function() {  
  alert(' Pozdrav svima! ');  
}
```

```
Sadrzaj func: function() {  
  alert(' Pozdrav svima! ');  
}
```

1. Funkcijska deklaracija kreira funkciju i skladišti je (sam kod funkcije, ne rezultat njenog izvršavanja) u varijablu sa imenom **kaziZdravo**.
2. Linija **let func = kaziZdravo;** kopira sadržaj varijable **kaziZdravo** u varijablu sa imenom **func**.
Napomena: ovde nema zagrada iza **kaziZdravo**. Da ih ima, ova naredba bi u varijablu **func** upisala **rezultat poziva funkcije kaziZdravo()**, a ne **samu funkciju kaziZdravo**.

`alert('Sadrzaj kaziZdravo: ' + kaziZdravo);` //ispis: ??

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

 `file://`

Pozdrav svima!

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

 `file://`

Pozdrav svima!

Sada se ista funkcija može pozvati i naredbom **kaziZdravo()** i naredbom **func()**.

Prosleđivanje različitog broja argumenata

- Česte su situacije u kojima se javlja potreba da se pri različitim pozivima **prosledi različit broj argumenata** ili da se **elementi niza proslede kao pojedinačni argumenti**.
- JS obezbeđuje mehanizam za to koji se zove **rest parametri i spread sintaksa**.
- **rest** od argumenata pravi niz, a **spread** sintaksa radi obrnuto: od niza pravi pojedinačne argumente.
- **rest** se primenjuje pri definisanju funkcije, a **spread sintaksa** pri pozivanju funkcije sa **rest parametrima**

Prosleđivanje različitog broja argumenata: rest parametri_{1/3}

```
function suma(a, b) {  
    return a + b;  
}  
alert( suma(1, 2, 3, 4, 5) );
```

3

```
function sumaSvih(...args) { // args je ime niza  
    let sum = 0;  
    for (let arg of args) sum += arg;  
  
    return sum;  
}
```

```
alert(sumaSvih(1));  
alert(sumaSvih(1, 2));  
alert(sumaSvih(1, 2, 3) );  
alert(sumaSvih(1, 2, 5) );
```

1

3

6

8

2.čas

Prosleđivanje različitog broja argumenata: `rest` parametri_{2/3}

```
function prikaziIme(ime, prezime, ...titule) {  
    alert( ime + ' ' + prezime );  
    alert( titule[0] );  
    alert( titule[1] );  
    alert( titule.length );  
}
```

```
prikaziIme("Julije", "Cezar", "Konzul", "Imperator");  
Julije Cezar           Konzul           Imperator           2
```

```
ime = "Julije", prezime = "Cezar"; ostali idu u niz titule =  
["Konzul", "Imperator"]
```

Šta se dobije sada?

```
function prikaziIme(ime, prezime, ...titule) {  
    alert( ime + ' ' + prezime );  
    alert( titule[1] );  
    alert( titule[0] );  
    alert( titule.length );  
}
```

```
prikaziIme("Julije", "Cezar", "Konzul",  
"Imperator");
```

Julije Cezar	Imperator	Konzul	2
--------------	-----------	--------	---

Šta će sada ispisati?

```
prikaziIme("Julije", "Cezar", "Imperator");
```

Julije Cezar	undefined	Imperator	1
--------------	-----------	-----------	---

Prosleđivanje različitog broja argumenata – neispravno_{3/3}

```
function prikaziIme(...titule ,ime, prezime) {  
  // ne radi  
}
```

```
function prikaziIme(ime, ...titule , prezime) {  
  // ne radi  
}
```

- rest parametri moraju biti na kraju liste parametara

Prosleđivanje različitog broja argumenata – spread sintaksa_{1/2}

```
alert( Math.max(3, 5, 1) );
```

5

```
let arr = [3, 5, 1];
```

```
alert( Math.max(arr) )
```

NaN

vraća “pogrešan” rezultat

Zato što funkcija `Math.max()` zahteva pojedinačne argumente

Što može da se postigne spread sintaksom:

```
let arr = [3, 5, 1];
```

```
alert( Math.max(...arr) );
```

5

Prosleđivanje različitog broja argumenata – **spread** sintaksa_{2/2}

- U pozivu može biti više rest argumenata

```
let arr1 = [1, -2, 3, 4];  
let arr2 = [8, 3, -8, 1];  
alert( Math.max(...arr1, ...arr2) );
```

8

- U pozivu se rest argumenti mogu kombinovati sa “običnim” argumentima

```
let arr1 = [1, -2, 3, 4];  
let arr2 = [8, 3, -8, 1];  
let devet = 9  
alert( Math.max(1, ...arr1, 2, ...arr2, devet) );
```

9

Funkcija i varijable: lokalne varijable i spoljašnje (globalne) varijable_{1/4}

- Varijabla deklarirana unutar funkcije, vidljiva je samo unutar te funkcije:

```
function showMessage() {  
    // varijabla message je lokalna za funkciju  
    let message = "Zdravo, ja sam JavaScript!";  
    alert( message );  
}
```

showMessage();

Zdravo, ja sam JavaScript!

Dodati sledece :

```
alert( message );
```

Uncaught ReferenceError: message is not defined at
<anonymous>:6:8

PYTHON-šta će ispisati?

```
def f1():
```

```
    a=2
```

```
    def f2():
```

```
        a=3
```

```
        print(a)
```

```
    f2()
```

```
    print(a)
```

```
a=1
```

```
f1()
```

```
print(a)
```

3

2

1

>>>

Funkcija i varijable: lokalne varijable i spoljašnje (globalne) varijable_{2/4}

Funkcija može da pristupi i varijablama koje su deklarisanе van nje (i da ih menja):

```
let userName // varijabla deklarisanа izvan funkcije
userName = 'Jova' // vrednost dodeljena izvan funkcije
function showMessage() {
    userName = 'Pera'; /* spoljašnja varijabla izmenjena unutar
    funkcije */
    let message = 'Zdravo, ' + userName;
    alert(message);
}
alert( userName + ' pre poziva funkcije');
showMessage();
alert( userName+ ' vrednost je modifikovala funkcija' );
```

Jova pre poziva funkcije
Zdravo Pera
Pera vrednost je modifikovala funkcija

Funkcija: lokalne varijable i spoljašnje varijable istog imena_{3/4}

- Ako postoje lokalna i spoljašnja varijabla sa istim imenom, lokalna varijabla “zaklanja” spoljašnju:

```
let userName = "Jovo"; // Spoljašnja varijabla
function showMessage() {
    let userName = "Pero"; // Lokalna varijabla istog imena
    let message = "Zdravo, " + userName;
    alert(message);
}
showMessage();           /* → Zdravo, Pero jer je funkcija kreirala i
                           koristila sopstvenu varijablu userName */
alert(userName);         /* → Jovo, jer nije pozvana funkcija pa se
                           koristi spoljašnja varijabla userName */
```

Zdravo, Pero
Jovo

Funkcija: lokalne varijable, spoljašnje (globalne) varijable_{4/4}

- Varijable deklarisanе van (bilo koje) funkcije (kao spoljašnja `userName`) zovu se ?

globalne varijable.

- Globalne varijable su vidljive iz svake funkcije (ako nisu “zaklonjene” lokalnim).
- Preporuka je da se minimizira korišćenje globalnih varijabli (zbog bočnih efekata – o tome kasnije).

Python - return

```
def maksimum(n1,n2):  
    if n1>n2:  
        rezultat=n1  
    else:  
        rezultat=n2  
    return rezultat  
veci=maksimum(15,16)  
print(veci)
```

16

Funkcija: naredba `return`_{1/4}

- Funkcija može (a ne mora) da vrati u kod koji je poziva vrednost kao rezultat.
- Naredba `return` ima dvojaku namenu u funkciji.
 - Da vrati vrednost
 - Da kontroliše tok izvršavanja programa

- Sintaksa je

`return` `[izraz]`; // `izraz` je vraćena vrednost

- Naredba `return` može da bude bilo gde u funkciji.
- Kada se u izvršavanju dođe do `return`, kontrola toka se vraća kodu koji je funkciju pozvao (na naredbu koja sledi poziv funkcije) i vraća se vrednost
- Primer:

```
function sum(a, b) {  
    return a + b;  
}
```

```
let result = sum(1, 2);  
alert( result );
```

// Ispis: 3

```
function sum(a, b) {  
    return a + b;  
    alert("šta će napisati")  
}
```

```
let result = sum(1, 2);  
alert( result );  
3
```

Funkcija: naredba `return`_{2/4}

- **return bez vrednosti rezultuje izlaskom iz funkcije:**

```
function showMovie(age) {
```

```
  if ( !(age > 12) ) {
```

```
    alert (" Imate manje od 12 godina ")
```

```
    return;
```

```
  }
```

```
  alert(" Prikazujem Vam film" ); // (*)
```

```
}
```

```
showMovie(22);
```

Prikazujem Vam film

```
showMovie(2);
```

Imate manje od 12 godina

Šta bi se desilo da nema Return?

Za showMovie(2) bi prikazalo:

Imate manje od 12 godina

Prikazujem Vam film

Funkcija: naredba `return`_{3/4}

- **Izlaz funkcije bez `return` ili sa praznim `return` je?**

```
function doNothing() { /* bez return */ }  
let fempty = doNothing (); // Ovo je poziv!!  
alert ('funkcija bez return vraća: ' + fempty)
```

nedefinisan (funkcija vraća vrednost **undefined**):

funkcija bez return vraća: undefined

```
function doNothing1() { /* sa praznim return*/  
  return;  
}  
let fempty1 = doNothing1 (); // Ovo je poziv!!  
alert ('funkc.sa praznim return vraća: ' + fempty1)
```

funkc.sa praznim return vraća: undefined

Funkcija: naredba `return`_{4/4}

- **Nikada nemojte stavljati novi red između `return` i povratne vrednosti** (; se pretpostavlja iza `return` pa se izlazi iz funkcije bez povratne vrednosti)

```
function sum(a, b) {  
    return
```

```
    a + b;  
}
```

```
let result = sum(1, 2);  
alert( result );  
undefined
```

- Kada će sračunati sumu?

```
function sum(a, b) {  
    return(a + b);  
}
```

```
let result = sum(1, 2);  
alert( result );
```

- **Višelinijski povratni izrazi se mogu stavljati u zatvorenu malu zagradu:**

```
return (nekidugiizraz + nesto * f(a) + f(b) )
```

```
function sum(a, b) {  
    return(a + b);  
}
```

```
let result = sum(1, 2) + 5*sum(3,4);  
alert( result );
```

Razlike između kanoničke deklaracije naredbom **function** i funkcijskog izraza

1. [Sintaksa](#)
2. [Kada](#) JavaScript endžin kreira funkciju
3. [Blokovsko dosezanje](#)

- JS sintaksa dozvoljava kanoničku deklaraciju samo u programu ili u telu funkcije.

1. **Kanonička deklaracija**: posebna naredba u kodu:

```
// Function Declaration
```

```
function sum(a, b) {  
    return a + b;  
}
```

```
alert(sum(3,4));
```

Nije dozvoljeno da se ova sintaksa koristi u konstruktu blok (`{ ... }`) u naredbama `if`, `while` ili `for`. NE RADI BAŠ UVEK!

2. **Funkcijski izraz**: reč `function` unutar izraza ili drugog sintaktičkog konstrukta. Na primer, funkcija kreirana na desnoj strani “izraza dodele” =:

```
// Function Expression
```

```
let sum = function(a, b) {  
    return a + b;  
};
```

```
alert(sum(2,3));
```

Za ovakvu sintaksu dozvoljeno je korišćenje u konstruktu blok (`{ ... }`) u naredbama `if`, `while` ili `for`.

Kada endžin kreira funkciju

- **Funkcijski izraz** se kreira kada se u izvršenju dođe do njega i upotrebljiv je tek od tog momenta.

```
sayHi("Pero"); // greška – ovde još ne može! Mi smo pozvali  
funkciju sa parametrom „Pero“, ali je još nismo definisali  
Da li ovo radi?
```

```
sayHi("Pero");  
let sayHi = function(name) { // (*) odavde može  
  alert( `Zdravo, ${name}` );  
};
```

```
Da li ovo radi?  
let sayHi = function(name) { // (*) odavde može  
  alert( `Zdravo, ${name}` );  
};  
sayHi("Pero");  
Zdravo Pero
```

- **Kanonički deklarisaná funkcija** može se pozvati pre nego što je definisana:

```
sayHi("Pero");  
function sayHi(name) {  
    alert( `Zdravo, ${name}` );  
}
```

Zdravo Pero

Blokovsko dosezanje₁

- U režimu strict, kada je *kanonička deklaracija* unutar bloka koda, **vidljiva je unutar toga bloka a nije vidljiva izvan bloka:**

```
"use strict";
```

```
let age = prompt("Koliko Vam je godina?", 18);
```

```
// Uslovno deklarisanje funkcije (u bloku)
```

```
if (age < 18) {
```

```
    function welcome() {
```

```
        alert("Hello!");
```

```
    }
```

```
} else {
```

```
    function welcome() {
```

```
        alert("Greetings!");
```

```
    }
```

```
}
```

```
// ...a poziva se izvan bloka
```

```
welcome();
```

```
// Greška: funkcija welcome nije definisana
```

```
ZNAČI OVO NE RADI
```


"use strict";

U JavaScriptu, **"use strict"** je fraza koja se koristi na:

- početku skripte ili
- funkcije

kako bi se omogućio "strict mode" ili **"strogi režim"**.

- Ovaj režim omogućava pisanje JavaScript koda na strožiji način, **primenjujući dodatna pravila i ograničenja, što pomaže u izbegavanju grešaka i loše prakse u kodiranju.**

Blokovsko dosezanje₁

U režimu **strict**, kada je kanonička deklaracija unutar bloka koda, **vidljiva je unutar toga bloka a nije vidljiva izvan bloka**:

"use strict";

```
let age = prompt("Koliko Vam je godina?", 18);
```

```
// Uslovno deklarisanje funkcije (u bloku)
```

```
if (age < 18) {
```

```
    alert("manje od 18");
```

```
    alert(age);
```

```
    function welcome() {
```

```
        alert("Hello!");
```

```
    }
```

```
} else {
```

```
    alert("veće od 18");
```

```
    alert(age);
```

```
    function welcome() {
```

```
        alert("Greetings!");
```

```
    }
```

```
}
```

```
// ...a poziva se izvan bloka
```

```
alert("test pre");
```

```
welcome();
```

```
alert("test");
```

Kada „use strict“ postoji:

Za broj manji od 18 ispiše:

Manje od 18

8

Test pre

Pukne. Zašto?

funkcija welcome nije definisana

Za broj veći od 18 ispiše:

45

Veće od 18

45

Test pre pa pukne isto kao i gore

Kada „use strict“ ne postoji :

Za broj manji od 18 ispiše:

Manje od 18

8

Test pre

Hello!

test

Ne pukne. Zašto?

Obratiti pažnju da je editor
podvukao ime funkcije:

Za broj veci od 18 ispiše:

45

Veće od 18

45

Test pre

Greetings!

test

Ne pukne.

```
function welcome() {  
    alert("Hello!");  
}
```

Blokovsko doseganje₂

```
"use strict";
```

```
let age = 16; // uzmimo 16 za primer
```

```
if (age < 18) {
```

```
    welcome();
```

```
    function welcome() {
```

```
        alert("Ćao!");
```

```
    }
```

```
    welcome();
```

```
} else {
```

```
    welcome();
```

```
    function welcome() {
```

```
        alert("Dobar dan!");
```

```
    }
```

```
    welcome();
```

```
}
```

```
// \    (radi)
```

```
// |    ovo je KANONOČNI OBLIK
```

```
// |    Funkcija welcome je dostupna
```

```
// |    u bloku u kome je deklarisan
```

```
// /    (radi)
```

```
//Ovo radi: za broj 16 ispiše 2x Ćao a za broj 26 ispiše 2x Dobar dan
```

```
// A ovde smo izvan bloka (crvene vitičaste zagrade),
```

```
// pa ne vidimo funkciju welcome koja je unutar bloka.
```

```
welcome(); // Zato dobijamo grešku: welcome is not defined
```

```
//znači ovo ne radi. Nije treći put ispisan tekst iz funkcije
```

Kada se u kodu programa umesto 16 upiše broj veći od 18 dobije se ispis dva puta Dobar dan i onda program pukne

Kada se ukloni : „use strict“

Ispiše se tri puta „Ćao“ ili tri puta „Dobar dan“

3.čas

Kako to ipak izmeniti: prvi način - FUNKCIJSKI IZRAZ sa **let** deklaracijom i dodela u bloku

```
"use strict";
```

```
let age = prompt("Koliko Vam je godina?", 18);
```

```
let welcome; // deklaracija izvan bloka. Ovo je dodato u odnosu na prošli primer
```

```
if (age < 18) {
```

```
    welcome = function() { //pre je pisalo function welcome()
```

```
        alert("Ćao!");
```

```
    };
```

```
} else {
```

```
    welcome = function() {
```

```
        alert("Dobar dan!");
```

```
    };
```

```
}
```

```
welcome();
```

```
alert("ovo sada radi");
```

Za broj manji od 18 ispiše se:

Ćao

Ovo sada radi

Za veće od 18 ispiše se:

Dobar dan!

Ovo sada radi iako je strict mod

Isto radi i sa "use strict" i bez

Kako to izmeniti: drugi način - *FUNKCIJSKI IZRAZ SA USLOVNIM IZRAZOM (?)*

```
"use strict";  
let age = prompt("Koliko Vam je godina?", 18);  
  
let welcome = (age < 18) ?  
    function() { alert("Ćao!"); } :  
    function() { alert("Dobar dan!"); };  
  
welcome();  
alert(" i sada radi!");
```

Ispiše : Ćao!, pa onda, I sada radi! za manje od 18
i : Dobar dan!, pa onda, I sada radi! za veće od 18
Isto radi i sa “use strict” i bez

Koju deklaraciju koristiti?

- **Funkcijski izraz** (**welcome = function()**) samo u situacijama kad kanonička deklaracija ne može da se upotrebi iz nekog razloga i, posebno, kada je potrebna **uslovna deklaracija**.
- **Kanoničku deklaraciju** (**function welcome()**) u svim ostalim situacijama, zato što
 - Daje više slobode u organizaciji koda jer se može pozvati pre pojave naredbe deklaracije u kodu.
 - Čitljivija je.

Funkcije: Streličasta sintaksa

Šta je streličasta sintaksa

- To je nova, **veoma jednostavna i koncizna sintaksa za kreiranje funkcija koja je često bolja od funkcijskih izraza.**

- Zove se “streličasta funkcija”, jer izgleda ovako:

```
let func = (arg1, arg2, ...argN) => izraz
```

- Gornja definicija je isto što i:

```
let func = function(arg1, arg2, ...argN) {  
    return izraz;  
};
```

Nema reči function i return!

Streličasta sintaksa: konkretan primer

```
let sum = (a, b) => a + b;
```

To je, u stvari, kraći oblik klasične definicije?:

```
let sum = function(a, b) {  
    return a + b;  
};
```

```
alert( sum(1, 2) );
```

Ispis: 3

Streličasta sintaksa: još kraći zapis

- Funkcija sa jednim parametrom dozvoljava još skraćivanja:

Recimo:

```
let double = function(n) { return n * 2 }  
alert(double(5));
```

10

Može da se piše ovako:

```
let double = (n) => n * 2; // bez ključne reči function  
alert(double(5));
```

A mogu se izostaviti zagrade oko liste parametara :

```
let double = n => n * 2;  
alert(double(5));
```

- Izuzetak: Ako je funkcija bez parametara, mora se navesti zagrada:

```
let sayHi = () => alert("Hello!");  
sayHi();
```

Hello!

Streličasta sintaksa i funkcijski izrazi

- Streličaste funkcije mogu se koristiti na isti način kao i funkcijski izrazi, na primer, za dinamičko kreiranje funkcije:

```
let age = prompt("Koliko Vam je godina?", 18);
```

```
let welcome = (age < 18) ?
```

```
  () => alert('Ćao!') :
```

```
  () => alert("Dobar dan!");
```

```
welcome(); // ok radi
```

Višelinijske streličaste funkcije

- Složenije stvari (n.pr., višestruki izrazi ili naredbe) se zatvaraju u vitičastu zagradu.
- U tom slučaju **obavezna je eksplicitna return naredba**:

```
let sum = (a, b) => { // zagrada koja otvara
                      // višelinijisku funkciju
  let result = a + b;
  return result; // ako se koristi vitičasta zagrada,
                 // obavezan je eksplicitan "return"
};               // zagrada koja zatvara višelinijisku
                 // funkciju

alert( sum(1, 2) ); // 3
```

Za izražavanje složenijih stvari koje ne mogu da stanu u jednu liniju streličasta sintaksa nalaže da se one zatvore u vitičastu zagradu

Da li je ovo kraj priče o funkcijama u JavaScript-u?

- Funkcija je centralni koncept JavaScript-a koji pruža još puno mogućnosti u radu sa funkcijama.
- Naš kurs se zove Funkcionalno programiranje, a u funkcionalnom programiranju kod se pravi komponovanjem funkcija.
- Dakle, nismo ni blizu kraja:
 - Ima još puno da se kaže i o funkcijama u JavaScript-u.
 - A još više o funkcijama u funkcionalnom programiranju.
- Zbog toga ćemo se vraćati na funkcije u različitim kontekstima u temama koje slede.

Izvršavanje JS koda

Deo koji smo upravo završili imao je za cilj da nas uvede u osnovne stvari (**koncepti**, **sintaksa** ...) jezika JavaScript.

U delu koji sledi pokušaćemo da objasnimo kako to sve radi “ispod haube”.

Ne baš do poslednjeg šrafića, ali da bar budemo u stanju da prepoznamo da li “auto” neće da krene zato što nema goriva, ili zato što nam je podignuta ručna kočnica.

Objasnićemo kako se izvršava naš izvorni kod.

Šta je izvršavanje koda?

U najogoljenijoj formi možemo ga posmatrati kao sledeću sekvencu

1. **Pribaviti neki raspoloživ podatak** (recimo, nešto binarno zapisano/kodirano), na primer brojnu vrednost 5.

`var pet = 5;`

2. **Izvršiti nad tim podatkom neku raspoloživu operaciju** (recimo množenje sa brojnom vrednošću 2)

`2*pet;`

3. **Rezultat operacije smestiti na neko mesto** na kome ćemo kasnije moći da ga nađemo u slučaju potrebe.

`let duplo = 2*pet;`

Trivijalno računanje u JS-u

```
var pet = 5;  
let duplo = 2*pet;  
let dvadeset = duplo*2;  
alert(pet); alert(duplo);  
alert(dvadeset);
```

ispisaće: 5 10 20

// A šta se dešava ako to uradimo ovako:

```
{var pet = 5;  
let duplo = 2*pet;}  
let dvadeset = duplo*2;  
alert(pet); alert(duplo);  
alert(dvadeset);
```

Ispisaće: ??

Za čega se koriste
{ } u JS ?

koriste se za

1. definisanje objekta i
2. Definisanje blokova

Kontekst izvršavanja JS koda

- Cilj: da vam bude jasno šta JS endžin radi, posebno zašto se neke funkcije/varijable mogu koristiti pre nego što se deklarišu, i kako se njihove vrednosti stvarno određuju.
- Koncept na koji se sve oslanja je KONTEKST IZVRŠAVANJA
- Šta je to: **kontekst izvršavanja** je apstraktni koncept koji **sadrži informacije o okruženju u kome se tekući kod izvršava**.
 - Osnovni konteksti (okruženja) u kojima se JS kod izvršava su:
 - **Globalni?**
pretpostavljeno okruženje koje “obuhvata” izvršavanje “celog programa” .
 - **Funkcijski?**
okruženje kada tok izvršavanja uđe u telo funkcije.
 - Postoji i okruženje **eval**
 - **Eval kod** — Tekst (kod) koji treba da se izvrši unutar interne **eval** funkcije;
 - funkcija **eval** evaluira string u kome je kod koji će se izvršiti, prosto napravi funkciju od teksta;
 - smatra se VELIKIM BEZBEDNOSNIM RIZIKOM i NE PREPORUČUJE SE ZA KORIŠĆENJE.
- Mi ćemo da objasnimo globalni i funkcijski kontekst

eval

```
a=eval("1 + 1");
```

```
alert(a);
```

2

U Pythonu, funkcija eval() izvršava izraze koji su predstavljeni kao stringovi.

```
var x = 10;
```

```
var y = 20;
```

```
var code = "x + y"; // string koji sadrži izraz koji želimo da evaluiramo
```

```
alert(code);
```

```
var result = eval(code); // Evaluacija stringa kao JavaScript koda
```

```
alert(result);
```

x + y

30

Funkcija **eval()** u JavaScriptu je ugrađena funkcija koja omogućava interpretaciju stringa kao JavaScript koda i izvršavanje tog koda.

eval() funkcija se koristi sa oprezom zbog potencijalnih sigurnosnih rizika i performansi.

Kada se koristi nepromišljeno, može izložiti vašu aplikaciju ranjivostima kao što su "injection" napadi.

Ako koristimo **eval()** sa korisničkim unosom, napadač može ubaciti maliciozni kod.

Kontekst izvršavanja: globalni i funkcijski

```
// global context

var sayHello = 'Hello';

function person() { // execution context

    var first = 'David',
        last = 'Shariff';

    function firstName() { // execution context
        return first;
    }

    function lastName() { // execution context
        return last;
    }

    alert(sayHello + firstName() + ' ' + lastName());
}
```

Uvek može da postoji SAMO JEDAN globalni kontekst i PROIZVOLJAN (konačan) BROJ funkcijskih konteksta. Kontekst može biti ugnježđen u drugi (“roditeljski”) kontekst:

1. *Globalni kontekst nije ugnježđen u drugi kontekst.*

2. *Funkcijski kontekst može biti ugnježđen u globalni kontekst (n.pr., kontekst funkcije person()) ili u kontekst druge funkcije (n.pr., konteksti funkcija firstName() i lastName() su ugnježđeni u kontekst funkcije person())*

Kontekst izvršavanja: globalni i funkcijski

```
var kaziZdravo = 'Zdravo,'          /* Globalni kontekst */
```

```
function osoba (){ // kontekst funkcije osoba()
```

```
    var imeVrednost = "Petar", prezimeVrednost = "Petrović"
```

```
    function ime() { // kontekst funkcije ime()
```

```
        return imeVrednost
```

```
    }
```

```
    function prezime() { // kontekst funkcije prezime()
```

```
        return prezimeVrednost
```

```
    }
```

```
    alert(kaziZdravo); alert(ime()); alert(prezime());
```

```
}
```

```
osoba()          /* Globalni kontekst */
```

Ispiše: Zdravo

ispiše : Petar

ispiše: Petrović

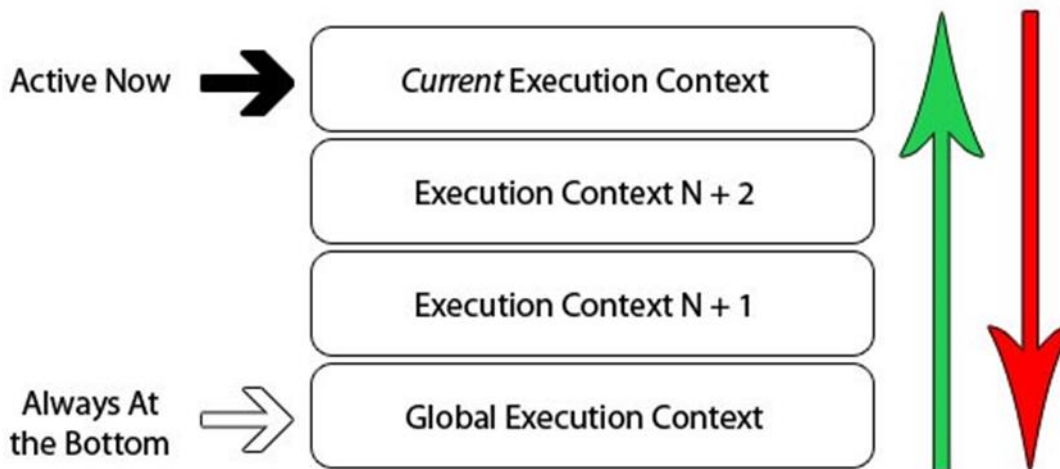
Šta će ispisati kada se poslednji red stavi u komentar?

Evaluiranje koda: stek (konteksta) izvršavanja

- Kada se program izvršava u računaru se mora obezbediti memorijski prostor u kome će biti smešteno sve ono što je potrebno za izvršavanje programa.
- Za te namene koristi se struktura podataka?

STEK.

- JavaScript **engžin** je u brauzeru implementiran kao jedna nit, pa je i stek sa jednom niti. Šta znači jedna nit?
- To znači da se u brauzeru u **jednom trenutku može dešavati samo jedna stvar**, a osta **izvršavanja**:



Evaluiranje koda: stek konteksta izvršavanja

- Kada endžin prvi put napuni skript, on ulazi u **globalni kontekst izvršavanja** po pretpostavci.
- Kada se u kodu poziva funkcija, sekvenca toka programa ulazi u pozvanu funkciju i pri tome kreira **novi kontekst izvršavanja** koji postavlja na vrh **steka izvršavanja**.
- Kada se pozove druga funkcija unutar tekuće funkcije, dešava se ista stvar. Tok izvršavanja ulazi u unutrašnju funkciju koja kreira **novi kontekst izvršavanja** koji se postavlja na vrh postojećeg steka.
- Svaki poziv funkcije kreira novi kontekst, odnosno novi element (zove se uobičajeno **frejm**) u steku izvršavanja

Global Execution Context

Execution Context N + 1

Global Execution Context

Execution Context N + 2

Execution Context N + 1

Global Execution Context

Stek konteksta izvršavanja: primer

```
(function foo(i) {  
    alert(i)  
    if (i === 3) {  
        return;  
    }  
    else {  
        foo(++i);  
    }  
})(0));
```

Šta kada se stavi 2
umesto 0 prilikom poziva
funkcije?

Rezultat je: 2,3

A kada se stavi 5?

Rezultat je: 5,6,7,.....besk.

Zagrada na početku ovog JavaScript koda
(function foo(i) { ... }(0)); označava da se
funkcija odmah izvršava

//Šta će ispisati

0,1,2,3

Funkcija foo() poziva se za vrednosti i 0,1,2,3. Svaki put
kada se pozove, kreira se novi kontekst izvršavanja

Stek konteksta izvršavanja: kreiranje

EC4 kontekst - poziv foo (3)

EC3 kontekst - poziv foo (2)

EC2 kontekst - poziv foo (1)

EC1 kontekst - poziv foo (0)

Globalni kontekst

```
(function foo(i) {  
    alert(i)  
    if (i === 3) {  
        return;  
    }  
    else {  
        foo(++i);  
    }  
})(0));
```

Globalni kontekst je uvek na dnu,
Ostali (0,1,2,3) se slažu na vrh steka.

Stek konteksta izvršavanja: ažuriranje steka pri izvršavanju

EC4 kontekst - poziv foo (3)

EC3 kontekst - poziv foo (2)

EC2 kontekst - poziv foo (1)

EC1 kontekst - poziv foo (0)

Globalni kontekst

Kada se kod izvršava, skida se sa vrha steka

Ključne stvari konteksta izvršavanja JS-a

1. Jedna nit.

JavaScript, kao jednonitni jezik, izvršava se na jednoj glavnoj niti u okviru svog okruženja, što znači da se kod izvršava sekvencijalno, a ne paralelno (C++ i Java).

2. Sinhrono izvršavanje.

Odnosi se na sekvencijalno izvršavanje koda, gde se svaki red koda izvršava jedan za drugim, bez ikakvih prekida ili čekanja na završetak nekog zadatka.

U sinhronom modelu izvršavanja, ako se izvršava neki dugotrajan zadatak, cela aplikacija će biti blokirana dok se taj zadatak ne završi.

3. Jedan globalni kontekst.

Globalni kontekst u JavaScriptu predstavlja najviši nivo izvršavanja koda. To je kontekst u kojem se izvršava JavaScript kod kada se pokrene aplikacija ili kada se JavaScript datoteka učitava u web pregledaču.

U globalnom kontekstu, glavna stvar koja se dešava je definisanje globalnih promenljivih i funkcija koje će biti dostupne iz svih delova koda aplikacije. Ovaj kontekst obuhvata sve što nije ugnježđeno unutar funkcija ili blokova koda.

4. Više (konačno mnogo) funkcijskih konteksta.

Sposobnost jezika da podrži ugnježdene funkcije i funkcije koje se pozivaju rekurzivno ili rekurzivno-nakon-povratka, što dovodi do stvaranja više funkcionalnih konteksta tokom izvršavanja koda.

5. Svaki poziv funkcije kreira novi kontekst, pa i rekurzivni poziv (kada funkcija poziva samu sebe).

- U osnovi, JavaScript izvršava kod **redom, liniju po liniju**. **Primer:** `alert("Prvi log"); alert("Drugi log");`
- Ako imamo dugotrajnu operaciju **bez asinhronog koda**, aplikacija može biti **blokirana**.

```
alert("Početak");
```

```
for (let i = 0; i < 1e9; i++) {} // Simulacija dugog  
procesa
```

```
alert("Kraj");
```

Probat i sa : 1e10

JS kontekst izvršavanja: konceptualni model

- Svaki **kontekst izvršavanja** se može konceptualno predstaviti kao **objekat** sa tri **svojstva** (koja su i sama objekti):

```
executionContextObj = {  
  'scopeChain': { /* variableObject + svi  
    variableObject-i roditeljskih konteksta  
    izvršenja */ },  
  'variableObject': { /* argumenti funkcije /  
    parametri, unutrašnje deklaracije varijabli i  
    funkcija */ },  
  'this': {} // pokazivač this  
}
```


1. **scopeChain** – prevod?

Ilanac dosezanja - u JavaScriptu **označava hijerarhiju opsega** (scope-ova) **koja se koristi za pronalaženje vrednosti promenljivih** tokom izvršavanja koda.

- Kada se traži vrednost promenljive u JavaScript kodu, interpreter prvo proverava :
 - lokalni opseg (scope) trenutnog izvršnog konteksta.
 - Ako promenljiva nije pronađena u lokalnom opsegu, interpreter zatim traži u opsegu roditeljske funkcije (ako postoji),
 - pa zatim u opsegu roditeljske funkcije te funkcije
 - i tako dalje, sve dok se ne stigne do globalnog opsega.

Hijerarhija opsega koja se formira od trenutnog opsega do globalnog opsega naziva se **scopeChain**

2. `variableObject` - prevod?

objekat promenljivih koji čuva informacije o svim promenljivima koje su definisane unutar tog konteksta.

- Objekat promenljivih uključuje informacije kao što su:
 - Deklarisane promenljive i njihove vrednosti.
 - Reference na funkcije koje su definisane unutar konteksta.
 - Parametri funkcije itd
- Kada JavaScript interpreter izvršava kod, koristi objekat promenljivih da pristupi i upravlja promenljivama u trenutnom opsegu (scope-u)

3. This - u JavaScriptu je specijalna ključna reč koja se koristi za **referenciranje trenutnog objekta** u kontekstu u kome se koristi

- Tačno značenje **this** zavisi od načina na koji se funkcija poziva:
 1. **U globalnom kontekstu:** Ako se funkcija poziva direktno u globalnom kontekstu, **this se odnosi na globalni objekat**, što u većini web pregledača predstavlja **window objekat u okviru browsera**,
 2. Kada se funkcija poziva kao metoda objekta, **this** se odnosi na sam objekat na kojem je pozvana ta funkcija.

Detalji JS konteksta izvršavanja

- Svaki put **kada se pozove funkcija**, radi se sa **novim kontekstom izvršavanja** u dve faze:
 - Faza 1 - ***kreiranje***: ?
dešava se kada se funkcija pozove, ali **pre izvršavanja bilo kakvog koda** unutar funkcije
 - Faza 2 – ***aktivacija/izvršavanje***: ?
dodeljuju se vrednosti referenci funkcijama i **kod se interpretira/izvršava**.

JS kontekst kreiranja: Faza 1

1. Kreira se lanac dosezanja (**scopeChain**).
2. Kreiraju se varijable, funkcije i parametri (**variableObject**).
3. Određuje se vrednost pokazivača **this**.

Dešava se kada se funkcija kreira, ali PRE NEGO ŠTO POČNE IZVRŠAVANJE.

JS kontekst izvršavanja: Faza 2

- U ovoj fazi se dodeljuju vrednosti funkcijama i kod se interpretira/izvršava.
- Pokreće se/ interpretira kod funkcije u kontekstu, i dodeljuju se vrednosti varijablama kako se kod izvršava, liniju po liniju.

Faza aktivacije/izvršenja koda: primer

```
function foo(i) {  
    var a = 'hello';  
    var b = function privateB() {    };  
    function c() {    }  
}  
alert('poziv foo(22) vraća: ',foo(22));
```

Rezultat: **poziv foo(22) vraća:** i pukne....

Poziv **foo(22)** ovde će da vrati **undefined**, jer funkcija **foo()** nema return a vrednost varijable pri kreiranju se postavlja na **undefined**.

Faza aktivacije/izvršenja koda: primer

```
function foo(i) {  
    var a = 'hello';  
    var b = function privateB() {    };  
    function c() {    }  
    return a  
}  
alert(foo(22));
```

Šta će biti ispisano u ovom slučaju?

Hello!

Faza aktivacije/izvršenja koda: primer

```
function foo(i) {  
    var a = 'hello';  
    var b = function privateB() {    };  
    function c() {    }  
    return b  
}
```

```
alert(foo(22));
```

A šta će biti ispisano u ovom slučaju?

```
function privateB() {    }
```

Šta se dešava u ovom kodu?

1. Deklaracija funkcije foo(i)

- Prima argument i, ali ga ne koristi. Zašto?
- Deklariše promenljivu a sa vrednošću 'hello' (ali se ne koristi).
- Deklariše funkciju b (nazvanu **privateB**) unutar foo().
- Deklariše funkciju c() (koja se takođe ne koristi).
- Vraća funkciju b.**

```
function foo(i) {  
    var a = 'hello';  
    var b = function  
};  
    function c() {  
        return b  
    }  
    alert(foo(22));
```

2. Poziv foo(22) unutar alert()

- Funkcija foo(22) se poziva.
- Vraća funkciju b**, ali je ne izvršava.
- alert() prima **funkciju** kao povratnu vrednost i pokušava je konvertovati u string.
- JavaScript **konvertuje funkcije u string tako što ispisuje njihov izvorni kod!**

Primer 2

```
let x = 10;  
function timesTen(a){  
    return a * 10;  
}  
let y = timesTen(x);  
alert(y); // ispis: 100
```

Šta radi kod?

- Dodeljuje vrednost 10 varijabli **x**. U kom doseg se nalazi varijabla **x**?
- Deklariše funkciju **timesTen()** koja svoj argument množi sa 10.
- Poziva funkciju **timesTen()** prosleđujući joj vrednost **x** kao argument i vraćenu vrednost skladišti u varijablu **y**. U kom doseg se nalazi varijabla **y**?
- Ispisuje vrednost varijable **y** na konzolu.

Primer 2: konteksti – faza kreiranja

Prvo se kreira globalni kontekst:

1. Kreira se globalni objekat, n.pr., `window` u brauzeru ili `global` u Node.js.
2. Kreira se vezivanje `this` za taj objekat, `this` pokazuje baš na kreirani globalni objekat iz tačke 1.
3. Uspostavlja se hip (struktura) za skladištenje referenci na varijable i funkcije.
4. Skladište se deklaracije funkcija i varijable unutar globalnog konteksta izvršavanja, pri čemu se varijable inicijalizuju na vrednost `undefined`.

Primer 2: konteksti - faza kreiranja

1. Kreira se globalni objekat, window u brauzeru.
2. Kreira se vezivanje `this` za taj objekat, `this` pokazuje objekat iz tačke 1.
3. Uspostavlja se prostor za skladištenje referenci na varijable i funkcije.
4. Skladište se deklaracije funkcija i varijable unutar globalnog konteksta izvršavanja, pri čemu se varijable inicijalizuju na vrednost `undefined`.

```
let x = 10;  
function timesTen(a){  
    return a * 10;  
}
```

```
let y = timesTen(x);  
alert(y); // ispis: 100
```

Global Execution Context

Creation Phase (Web Browser)

Global Object: window

Korak 1

this: window

Korak 2

x: undefined

timesTen: function(){...}

Koraci
3 i 4

y: undefined

Primer 2: konteksti - faza izvršavanja

```
let x = 10;  
function timesTen(a){  
    return a * 10;  
}  
  
let y = timesTen(x);  
alert(y); // ispis: 100
```

Global Execution Context

Execution Phase (Web Browser)

Global Object: window

this: window

x: 10

timesTen: function(){...}

y: timesTen(x)

Primer 2: konteksti – faza kreiranja konteksta funkcije **timesTen()**

Poziv funkcije



Kreira novi
kontekst

```
let x = 10;  
function timesTen(a){  
    return a * 10;  
}
```

```
let y = timesTen(x);  
alert(y); // ispis: 100
```

Global Execution Context

Execution Phase (Web Browser)

Global Object: window

this: window

x: 10

timesTen: function(){...}

y: timesTen(x)



Function Execution Context

Creation Phase

Global Object: arguments

this: window

a: undefined

Primer 2: faza izvršavanja funkcije

U toku izvršavanja funkcijskog konteksta:

- dodeljuje se vrednost 10 parametru **a**,
- evaluiira se izraz **a * 10** i
- vraća se rezultat (**100**).

Naredba **return** vraća kontrolu niti izvršavanja u globalni kontekst.

U globalnom kontekstu se :

- vraćena vrednost vezuje sa varijablom **y**
- poziva se funkcija **alert**, za koju se kreira i izvršava poseban kontekst.
- Kada se završi izvršavanje funkcije **alert**, kontrola se opet vraća globalnom kontekstu i program se terminira.

