

JavaScript

Osnove JavaScript jezika Vanilla JS

Tamara Naumović
tamaranaumovic@gmail.com

Beograd, 2018

Korisni linkovi

How it feels to learn JavaScript in 2016:

<https://hackernoon.com/how-it-feels-to-learn-javascript-in-2016-d3a717dd577f>

The State Of JavaScript: Front-End Frameworks:

<https://medium.com/@sachagreif/the-state-of-javascript-front-end-frameworks-1a2d8a61510>

The State Of JavaScript: Javascript Flavors:

<https://medium.com/@sachagreif/the-state-of-javascript-javascript-flavors-1e02b0bfefb6>

My journey of becoming a web developer:

<https://medium.freecodecamp.org/my-journey-to-becoming-a-web-developer-from-scratch-without-a-cs-degree-2-years-later-and-what-i-4a7fd2ff5503>

ECMAScript: <https://www.ecma-international.org/ecma-262/8.0/index.html>

MDN: <https://developer.mozilla.org/en-US/>

DOM: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

JS patterns: <https://addyosmani.com/resources/essentialjsdesignpatterns/book/>

Principles: <https://github.com/rwaldron/idiomatic.js>

You might not need JS: <http://youmightnotneedjquery.com/>

First-class functions:

https://developer.mozilla.org/en-US/docs/Glossary/First-class_Function

Difference between of first-class and high-order functions:

<https://stackoverflow.com/questions/10141124/any-difference-between-first-class-function-and-high-order-function>

JavaScript

Osnovne karakteristike JavaScript jezika

Java Script pripada grupi jezika za skriptovanje, pre svega klijentske strane, mada se može izvršavati i na serveru (runat="server"). Java Script je najpopularniji jezik na Internetu, koji je dizajniran da poveća interaktivnost HTML strana. JavaScript jezik je nastao u kompaniji Netscape pod prvobitnim imenom LiveScript koje mu je dodelio direktor projekta Brendan Eich. Ovaj jezik je razvijen za dve namene.

Prva namena mu je bila da posluži kao skript jezik koji će administrator Web servera moći da koristi za upravljanje serverom i povezivanje njegovih strana sa drugim servisima, kao što su programi za pretraživanje koje korisnici koriste kako bi pronašli željenu informaciju.

Druga namena je bila da ovi skriptovi budu korišćeni na strani klijenta, u HTML dokumentima, kako bi na mnoge načine poboljšali Web strane. Na primer, autor bi pomoću LiveScript jezika mogao da kontroliše da li je korisnik pravilno uneo tražene podatke, pre nego što se ti podaci pošalju do servera i server pošalje odgovor. Time se štedi vreme i povećava efikasnost stranice, a sva izračunavanja i provere se vrše na klijentu tako da se zapošljava neiskorišćena snaga računara klijenta.

Početkom decembra 1995. godine, Netscape i Sun su zajednički objavili da će se skript jezik ubuduće zvati JavaScript. Iako je Netscape imao nekoliko dobrih marketinških razloga za usvajanje ovog imena, izmena je možda izazvala više konfuzije nego što je iko očekivao. Ispostavilo se da je prava poteškoća bila da se svetu objasni razlika između jezika Java i jezika JavaScript.

Nakon raznih poboljšanja i nestandardnih kopija, međunarodna organizacija za standarde u informacionim i komunikacionim sistemima (European Computer Manufactures Association - ECMA international) standardizovala je JavaScript 1997. godine pod nazivom ECMAScript. Pod ovom organizacijom je nastavio da se razvija do danas i imao je veliki broj značajnih poboljšanja. Trenutno, poslednja standardizovana verzija JavaScripta je 1.8.5.

JavaScript programi se koriste da bi detektovali i reagovali na događaje koji su inicirani od strane korisnika, kao npr. prelazak mišem preko linka i grafičkog objekta na strani.

JavaScript kod može da poboljša kvalitet sajta navigacionim pomagalima, skrolovanim porukama, dialog prozorima, dinamičnim slikama, kolicima za kupovinu (shopping cart), itd. JavaScript omogućava korisnicima da kontrolišu izgled veb strane dok se dokument „tumači“ od strane čitača. Uz pomoć JavaScripta, možemo da proverimo tačnost podataka koje je korisnik ukucao u formu, pre nego što se ti podaci prebace na server. JavaScript testira brauzer korisnika da vidi da li ima neophodne dodatke (plugins) za učitavanje tražene strane i ako ih nema, JavaScript usmerava korisnika na drugi sajt da ih nabavi. Njegovo jezgro sadrži osnovne elemente programskih jezika kao što su promenljive, tipovi podataka, kontrolne petlje, if/else izjave, switch izjave, funkcije i objekte. JavaScript se koristi za aritmetičke operacije, manipulacije datumom i vremenom, nizovima, stringovima i objektima.

On takođe čita i upisuje vrednosti kuki fajlova i dinamički kreira HTML na osnovu vrednosti kukija.

Kada brauzer korisnika prihvati stranu koja sadrži JavaScript kod, kod se šalje JavaScript interpreteru, koji izvršava skriptu. Budući da svaki čitač ima svoj interpreter, često postoje razlike u načinu na koji se skript izvršava. I dok se konkurentske kompanije stalno utrukuju poboljšavajući i modifikujući njihove čitače, novi problemi se javljaju. Ne što samo različite verzije čitača dovode do inkompatibilnosti, već se problemi javljaju i sa različitim verzijama istog brauzera. JavaScript engine (JavaScript interpreter) je interpreter koji interpretira JS source kod i izvršava skript.

Od 2009.godine svi važniji browseri u svojim novim verzijama podržavaju JS engine, koji je standardizovan u okviru ECMAScript (ECMA-262) Edition 3.

ECMAScript standard se godinama menjao i na taj način proširivao i dopunjavao JavaScript kao jezik i tako sa početkom 2012. svi moderni pretraživači u potpunosti podržavaju ECMAScript 5.1 koji je u korelaciji sa standardom ISO/IEC 16262:2011.

U junu 2015. godine objavljena je šesta verzija ECMAScript, poznata kao ES6 ili ES2015 koja donosi velike sintaksne promene kada je u pitanju pisanje kompleksnih aplikacija, uljučujući klase i module, petlje, nizove i poznate arrow funkcije. Od tada, svake godine ECMAScript standardi se objavljuju jednom godišnje.

Više informacija možete videti na : <https://developer.mozilla.org/bm/docs/Web/JavaScript>

Neke od mogućnosti koje Java Script pruža svojim korisnicima /programerima su:

1. Kada Web strana treba da odgovori ili direktno reaguje na korisnikovu interakciju sa formama (polja za unos, dugmad, radio dugmad...) i hipertekstualnim vezama
2. Kada je potrebno da se kontroliše kretanje kroz više okvira, priključke ili Java aplete u zavisnosti od korisnikovog izbora u HTML dokumentu
3. Kada je potrebno da se na računaru klijenta izvrši validacija unetih podataka pre nego se ti podaci pošalju serveru
4. Kada je potrebno da se stranica učini lepšom i organizovanijom pomoću padajućih menija, dinamičke promene izgleda i rasporeda elemenata na stranici (uz pomoć CSS-a i DOM-a)
5. Reagovanje na određene događaje (klik mišem, učitavanje stranice...)
6. Čitanje i pisanje HTML stranica
7. Čitanje i menjanje sadržaja HTML strane

Razlika između jezika JavaScript i Java

Bez obzira na nazive, Java i JavaScript su dva potpuno različita jezika po konceptu i dizajnu. Oni predstavljaju dve različite tehnike programiranja na Internetu. Java je programski jezik. JavaScript je (kako mu i samo ime kaže) script jezik. Razlika je u tome što se u Javi mogu kreirati pravi programi, koji mogu da se izvršavaju potpuno nezavisno od WWW čitača, ili Java apleti koji se mogu pozivati iz HTML dokumenta i koji se dovlače preko mreže i onda izvršavaju u okviru WWW čitača (browser-a).

Tabela 1:Poređenje JavaScript-a i Java-e

Java Script	Java
Ne kompajlira se, već samo interpretira kod klijenta.	Kompajlira se na serveru pre izvršenja kod klijenta.
Objektno zasnovan. Koristi postojeće objekte, bez klasa ili nasleđa.	Objektno orjentisan. Apleti sadrže objekte sastavljene od klasa sa nasleđem.
Kod je integrisan u HTML.	Applet je izdvojen iz HTML-a.
Tip promenljive se ne deklarise.	Tip promenljive mora biti deklarisan.
Dinamička povezanost. Objektne reference proveravaju se u vreme izvršavanja.	Statička povezanost. Objektne reference moraju postojati u vreme kompajliranja.

Gde smestiti JavaScript kod i kako se izvršava

Komande JavaScript-a su u tekstualnom formatu, tako da se za pisanje skriptova može koristiti bilo koji tekst editor, npr. Visual Studio Code.

Da bi browser prepoznao JavaScript kôd, potrebno je da se taj kôd postavi unutar para oznaka `<script>` i `</script>`.

```
<script>
<!--znak za komentar u HTML-u
      Naredbe JavaScript-a
-->
</script>
```

Da bi se izbeglo prikazivanje JavaScript kôda, odnosno ako želimo u okviru HTML dokumenta da sakrijemo, iskomentarišemo delove JavaScript kôda, naredbe skripta se smeštaju između simbola za **HTML komentar** (`<!-- -->`).

Kôd za povezivanje stranice sa eksternom bibliotekom funkcija može da se nađe u zaglavlju stranice (između tagova `<head>` i `</head>`), ako je potrebno da se biblioteka učitava pre nego što se izvrše drugi skriptovi na stranici, ili u telu stranici (između tagova `<body>` i `</body>`).

Sledi Primer 1. JavaScript-a.

```
<html>
<head>
<title>Jednostavan JavaScript</title>
</head>
<body>
Tekst prikazan na uobicajen nacin.
<br>
<script>
document.write("Zdravo svima!")
</script>
</body>
</html>
```

Reč *document.write* je standardna JavaScript komanda za ispisivanje izlaza na stranici pa će gornji kôd prikazati na stranici sledeći izlaz: „Zdravo svima!“.

Drugi način za uključivanje kôda napisanog JavaScript-om je povezivanje spoljašnje datoteke koja sadrži naredbe ili funkcije. Povezivanje u ovom slučaju se obavlja uz pomoć **atributa src** oznake `<script>` i to na sledeći način:

```
<script src="mojskript.js"></script>
```

Primer 2. Menjanje sadržaja tagova

index.html

```
<!DOCTYPE html>
<html lang="sr">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Prvi heading</title>
</head>
<body>
  <h1>Prvi heading</h1>
  <br>
  <script src="mojskript.js"></script></body>
</body>
</html>
```

Osvežiti stranicu i konstatovati prikaz sadržaja h1 taga, zatim u fajl skripte upisati sledeći kôd :

mojskript.js

```
var mojHeading = document.querySelector('h1');
mojHeading.textContent = 'Drugi heading!';
```

Ako osvežimo našu stranicu možemo videti da se sadržaj h1 taga promenio.

Kontekst Izvršavanja i Leksičko okruženje

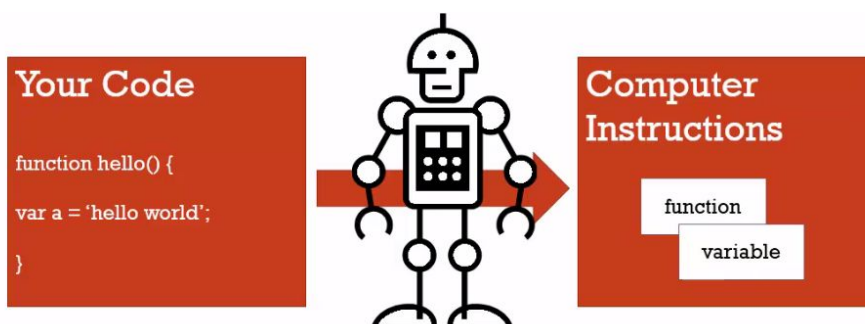
Sintaksni Parseri, Kontekst Izvršavanja i Leksičko Okruženje

(Syntax Parsers, Execution Contexts, and Lexical Environments)

SYNTAX PARSER: A PROGRAM THAT READS YOUR CODE AND DETERMINES WHAT IT DOES AND IF ITS GRAMMAR IS VALID

Your code isn't magic. Someone else wrote a program to translate it for the computer.

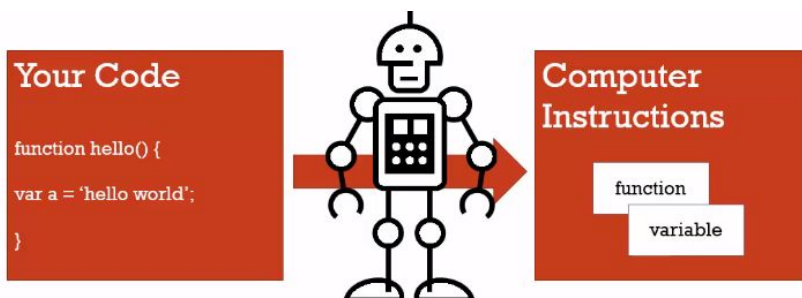
[Sintaksni parser](#) je deo JavaScript engine-a. On čita naš kôd karakter po karakter, zaključuje šta naš kôd treba da uradi i proverava gramatiku ispisanog kôda.



LEXICAL ENVIRONMENT: WHERE SOMETHING SITS PHYSICALLY IN THE CODE YOU WRITE

'Lexical' means 'having to do with words or grammar'. A lexical environment exists in programming languages in which **where** you write something is *important*.

Leksičko okruženje, definisano u oficijalnoj dokumentaciji ES5, opisuje asocijaciju identifikatora ka vrednostima promenljivih i funkcija baziranu strukturama leksičkog ugnježdavanja ES kôda. Zapravo, leksičko okruženje predstavlja mesto gde nešto fizički nalazi u našem kôdu.



EXECUTION CONTEXT: A WRAPPER TO HELP MANAGE THE CODE THAT IS RUNNING

There are lots of lexical environments. Which one is currently running is managed via execution contexts. It can contain things beyond what you've written in your code.

Kontekst izvršavanja je apstraktni koncept okruženja u kojem se trenutni kôd evaluira. Možemo ga posmatrati na globalnom nivou ili na nivou funkcije.

Promenljive

Definisanje promenljivih

Postoji nekoliko načina za definisanje promenljivih u JavaScript-u. Najčešće se promenljiva deklarise uz pomoć ključne reči `var`, `let` ili `const` iza koje sledi ime promenljive. Dakle, ako hoćemo da definišemo promenljivu godine, naredba JavaScript-a glasi:

```
var godine;
```

Promenljivoj se može dodeliti vrednost pri deklarisanju promenljive (inicijalizacija) ili kasnije pomoću operatora dodele. Najčešće korišćeni operator za tu svrhu je znak jednakosti. U sledećem primeru, pri deklarisanju promenljive izvršena je i inicijalizacija promenljive (dodeljivanje početne vrednosti):

```
var godine = 22;
```

Ako se vrši deklaracija promenljive bez inicijalizacije, tada ta promenljiva dobija vrednost koju joj dodeljuje čitač. Najčešće je to vrednost praznog objekta. Kada vrednost promenljivoj želimo dodeliti kasnije u okviru naredbi skripta to se postiže navođenjem imena deklarisanе promenljive i dodeljivanjem željene vrednosti pomoću operatora `=`. Na primer:

```
godine = 22;  
tekst = „Vrednost promenljive je ovaj tekst“;
```

Ključna reč `var` se koristi samo pri inicijalizaciji, odnosno deklarisanju promenljive, tj. samo jednom za vreme postojanja jedne promenljive. Za razliku od strogo tipiziranih jezika, u

JavaScript-u, pri deklaraciji promenljive ne definiše se njen tip. U toku izvršavanja skripta jedna promenljiva, u stvari, može uzeti vrednosti različitih tipova.

Promenljivoj se kao vrednost može dodeliti i rezultat nekog izraza. Na primer:

```
var prviSabirak = 1;  
var drugiSabirak = 2;  
var zbir = prviSabirak + drugiSabirak;
```

Java Script promenljiva može da sadrži bilo koji tip vrednosti. Što se tiče imena promenljivih, postoje određena pravila koja moraju da budu zadovoljena. U JavaScriptu imena promenljivih se sastoje od sekvenci slova (a-z, A-Z), cifara (0-9) i donje crte (_). JavaScript razlikuje velika i mala slova. Ključne reči (for, if, else, class, byte, int...) ne mogu se koristiti u nazivu identifikatora. U JavaScriptu ne moramo eksplicitno da deklariramo promenljivu (mada je to dobra praksa). Promenljiva ne može da sadrži razmak. Najsigurnije je da ime sadrži samo jednu reč. Ako postoji potreba za imenom od dve reči, preporučuje se korišćenje jedne od dve konvencije. Prva je da se te dve reči razdvoje donjom crtom, dok u drugoj kombinacija reči počinje malim slovom, a početno slovo druge reči je veliko.

Prilikom deklarisanja promenljivih treba obratiti pažnju na sledeće :

- Pokušati da se deklariraju sve varijable koje će se koristiti na početku programskog koda, čak iako još nema konkretnih vrednosti za njih.
- Koristiti ispravna imena varijabli. Ne koristiti rezervisane reči i reči koje su predugačke ili teške za pamćenje.
- Imena varijabli su osetljiva na velika i mala slova (case sensitive), MojeIme nije isto što i mojeime.
- Ne treba davati slična imena različitim varijablama.
- Iako nije uvek obavezno da se koristi var ključna reč prilikom deklarisanja promenljivih, dobro je koristiti je, sigurnije je.

Doseg promenljivih

U JavaScript-u se promenljive mogu definisati izvan ili unutar funkcije (o funkcijama će kasnije biti reči). Promenljive koje su definisane izvan funkcija zovu se globalne promenljive; one koje su definisane unutar funkcija zovu se lokalne promenljive. Globalne promenljive u JavaScript-u donekle se razlikuju od onih u većini drugih jezika. U JavaScript-u termin 'globalno' odnosi se na tekući dokument koji je učitao u prozor ili okvir čitača. Prema tome, kada se inicijalizuje jedna promenljiva, to znači da sve naredbe na toj stranici, uključujući i one unutar funkcija, imaju direktan pristup vrednosti te promenljive.

Važno je napomenuti da onoga trenutka kada se stranica ukloni iz čitača, sve globalne promenljive koje su definisane za tu stranicu brišu se iz memorije. Ako nam je potrebna vrednost koja će biti postojana iz stranice u stranicu, koriste se druge tehnike da uskladište tu vrednost (na primer, globalnu promenljivu u podešavanju okvira, ili kolačić).

Nasuprot globalnoj promenljivoj, lokalna promenljiva je definisana unutar funkcije. Doseg lokalne promenljive su samo naredbe u sklopu funkcije. Nijedna druga funkcija niti naredba izvan funkcije nemaju pristup lokalnoj promenljivoj.

Treba naglasiti još jednu bitnu stvar u vezi sa lokalnim i globalnim promenljivama. Naime, JavaScript dozvoljava da se na istoj stranici koriste globalne i lokalne promenljive sa istim imenom. Međutim, kada se to radi treba biti veoma obazriv i treba znati sledeće. Kada se unutar funkcije inicijalizuje vrednost lokalne promenljive istog imena kao i globalna promenljiva, mora se koristiti ključna reč `var` ispred imena lokalne promenljive. U suprotnom, interpreter će tu naredbu shvatiti kao da treba da dodeli vrednost globalnoj promenljivoj koja ima to ime i dobiće se neočekivani rezultat.

Hoisting promenljivih



Jedna od neobičnih stvari kod promenljivih u JavaScript jeziku je pojava koja se zove **hoisting**, koja dozvoljava pristup promenljivama definisanim kasnije u kodu, bez pojave greške ili exception-a. Ovo znači da su sve definicije promenljivih vidljive JavaScript engine-u (koji se nalazi u pretraživaču) pri samom učitavanju skripte, "podizaju se na početak skripte", i time se dozvoljava njihovo pozivanje u kodu iako im je kasnije dodeljen tip. Međutim, promenljive do kojih se nije došlo u izvršavanju imaju vrednost `undefined` (o kojoj ćemo kasnije pričati više). To znači da JS engine u memoriji pamti deklaraciju ovih promenljivih ali ne i njihovu vrednost pre izvršavanja te linije koda.

Primer Hoisting

mojskript.js

```
/**
 * Primer 1
 */
console.log(x === undefined); // true
var x = 3;

/**
 * Primer 2
 */

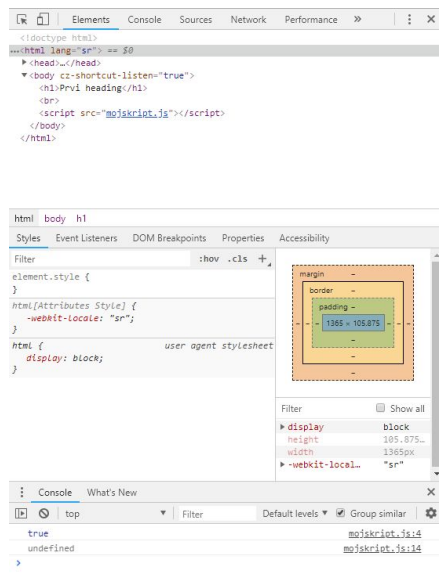
var myvar = 'globalna varijabla';

function funkcija() {
  console.log(myvar); // undefined
  var myvar = 'lokalna varijabla';
};

funkcija(); //poziv funkcije
```

U konzoli pretraživača možemo videti rezultat izvršavanja ove skripte.

Desnim klikom kliknemo na stranicu i odaberemo opciju `Inspect` i u delu `Console` možemo videti ispisane rezultate:



Prvi heading

Prethodni primer je identičan sledećem kodu:

```
/**
 * Primer 1
 */
var x;
console.log(x === undefined); // true
x = 3;

/**
 * Primer 2
 */

var myvar = 'globalna varijabla';

function funkcija() {
  var myvar;
  console.log(myvar); // undefined
  myvar = 'lokalna varijabla';
};

funkcija(); //poziv funkcije
```

Ovde smo deklarirali promenljive bez dodela njihovih vrednosti i zbog toga konzola ispisuje vrednosti undefined.

Konstante

Ako želimo da definišemo promenljive čije vrednosti se ne mogu menjati njihovom daljom manipulacijom, koristimo rezervisanu reč const. Konstante moraju biti inicijalizovane sa nekom vrednošću (dodela se ne može vršiti kasnije). Konstante moraju imati unikatna imena, što znači da ne možemo imati promenljive ili funkcije definisane istim imenom kao i konstante.

Imena promenljivih

Pri dodeljivanju imena promenljivama, važe ista pravila kao i kod većine drugih programskih jezika. Preporuka je da se dodeljuju opisna imena promenljivama kako bi se olakšalo čitanje i održavanje kôda. Pri dodeljivanju imena promenljivama u JavaScript-u važe sledeća ograničenja:

- Ne mogu se koristiti ključne reči JavaScript-a za dodeljivanje imena. To uključuje sve ključne reči koje se sada koriste i sve koje su u planu da budu uvedene.
- Ime promenljive ne sme da sadrži razmak
- JavaScript je case sensitive, što znači da promenljive godine i Godine nisu jedno te isto
- Ime promenljive ne sme da počne brojem
- Ime promenljive ne sme da sadrži znake interpunkcije, osim znaka za podvlačenje

Tipovi promenljivih

Svaka napisana naredba JavaScript-a na neki sebi svojstven način obrađuje podatke koji joj se proslede, na primer prikaz teksta na ekranu uz pomoć Java Scripta ili uključivanje i isključivanje radio dugmeta u obrascu. Svaki pojedinačni deo informacije u obrascu se naziva vrednost. U JavaScriptu postoji nekoliko tipova vrednosti. Formalni tipovi podataka su:

1. Primitivni tipovi:
 - a. String - niz znakova unutar navodnika
 - b. Number - bilo koji broj koji nije pod navodnicima
 - c. Boolean - logičko istinito ili neistinito
 - d. Null - lišeno vrednosti
 - e. Undefined – primitivna vrednost koja se automatski dodeljuje promenljivama koje su deklarisanе, ako im nije eksplicitno dodeljena druga vrednost
 - f. Symbol (novo u ES6) – unikatne nepromenljive (immutable) primitivne vrednosti koje se najčešće koriste kao ključevi pri kreiranju Objekata.
2. Kompleksni tipovi:
 - a. Object - sva svojstva i metodi koji pripadaju objektu ili nizu
 - b. Array – strukture podataka koje omogućavaju skladištenje više vrednosti unutar jedne reference
3. Function - definicija funkcije

Metoda kojom se može videti dodeljeni tip promenljivoj je **typeof**(naziv_promenljive).

Undefined

Primer 4:

```
var a;  
console.log(a);
```

```
if (a === undefined) {  
    console.log('a is undefined!');
```

```

}
else {
  console.log('a is defined!');
}

```



Undefined je vrednost ili recimo tip promenljive, kao sto je number, string itd... **Ne koristiti ovu vrednost kao vrednost promenljive, tj rucno postavljati undefined kao vrednost promenljivih. Koristiti ovo kao feature za debug.**

Stringovi

String je niz karaktera sačinjen od nula ili više karaktera zatvorenih u jednostrukim (') ili dvostrukim (") navodnicima.

Primeri stringova:

```

"prvi"
'1234'
"PRVA LINIJA \n druga linija"

```

Takođe je često neophodno koristiti navodnike unutar niza karaktera. To se može uraditi korišćenjem obrnute kose crte. Na primer:

```

var tekst="<P>On je isao u skolu \"Naziv skole \". "
document.write(tekst)

```

U stringovima je dozvoljeno koristiti sledeće specijalne karaktere:

- \b = pomeraj za jedno mesto ulevo (backspace)
- \f = pomeraj jedan red dole (form feed)
- \n = na početak novog reda (new line character)
- \r = return (carriage return)
- \t = tabulator (tab).

JavaScript je slabo tipiziran jezik, dinamički kucan. Tipovi podataka će biti automatski konvertovani zavisno od mesta njihove upotrebe u programu. Tako na primer možemo definisati i inicijalizovati sledeću promenljivu:

```

var promenljiva=11

```

a kasnije je možemo predefinisati, dodeljujući joj niz karaktera, na primer

```

promenljiva="Osnove JavaScript-a"

```

U kombinaciji broja i stringa, JavaScript konvertuje broj u string. Na primer:

```

x = "Primer za kombinaciju " + 11
y = 11 + " jos jedan primer."

```

Dobra je praksa definisati promenljivu sa **var**, kada je uvodimo. Specijalna ključna reč **null** ukazuje da je promenljivoj dodeljena null vrednost. To ne znači da je promenljiva nedefinisana. Promenljiva je nedefinisana kada joj nije dodeljena nikakva vrednost i tada je ne možemo dodeliti drugoj promenljivoj ili koristiti u izrazima, a da ne dobijemo run-time grešku (greška pri izvršavanju programa).

Celi brojevi

Celi brojevi u JavaScriptu mogu biti predstavljeni u tri osnove: u decimalnom (baza 10), u oktalnom (baza 8) i heksadecimalnom (baza 16) formatu.

Decimalni celi brojevi se predstavljaju kao niz cifara (0-9) bez vodeće nule.

Oktalni celi brojevi se predstavljaju kao niz cifara (0-7) predvođen sa nulom ("0").

Heksadecimalni celi brojevi

Heksadecimalni celi brojevi se predstavljaju kao niz cifara (0-9) i slova (a-f i A-F) predvođen sa nulom koju sledi slovo x ("0x" ili "0X").

Brojevi u pokretnom zarezu

Brojevi u pokretnom zarezu imaju sledeće delove: decimalni ceo broj, decimalnu tačku ("."), deo iza decimalnog zareza (decimalni ceo broj), eksponent ("e" ili "E", praćen decimalnim celim brojem).

Primeri brojeva u pokretnom zarezu:

1.1234 .1E23 -1.1E12 2E-10

Boolean

Promenljive tipa Boolean može da ima samo dve vrednosti: *true* (tačno) i *false* (netačno).

Dinamičko kucanje

(Dynamic typing)

DYNAMIC TYPING:

**YOU DON'T TELL THE ENGINE WHAT
TYPE OF DATA A VARIABLE HOLDS,
IT FIGURES IT OUT WHILE YOUR
CODE IS RUNNING**

Variables can hold different types of values because it's all figured out during execution.

Svi tipovi se dodeljuju dinamički dok pišete kod a ne predefinisano kao kod većine objektno orjentisanih programskih jezika.

Ključ-vrednost par i objekti

(Name-value pairs and objects)

NAME/VALUE PAIR:
**A NAME WHICH MAPS TO A UNIQUE
VALUE**

The name may be defined more than once, but only can have one value in any given **context**.

That value may be more name/value pairs.

OBJECT:
A COLLECTION OF NAME VALUE PAIRS

The simplest definition when talking about **Javascript**.

```

Address:
{
  Street: 'Main',
  Number: 100
  Apartment:
  {
    Floor: 3,
    Number: 301
  }
}

```



Globalno okruženje i Globalni

objekat

(The Global Environment and The Global Object)

Osnovni *kontekst izvršavanja* je Globalni kontekst izvršavanja i on obezbeđuje dve stvari: Globalni Objekat i "this"

Primer 1:

HTML:

```

<html>
  <head>

  </head>
  <body>

    <script src="app.js"></script>
  </body>
</html>

```

JS:

//bez koda

Pokrenućemo nasu aplikaciju. U konzoli kucamo **this** -sto predstvalja promenljivu koju JS engine kreira pri pokretanju bilo kog js fajla, i vidimo da postoji objekat, globalni objekat - window u nasem slucaju, jer se radi o prozoru koji otvorio nas kod. U prvom slucaju `this===objekat===window`

Primer 2:

HTML:

```

<html>
  <head>

```



```

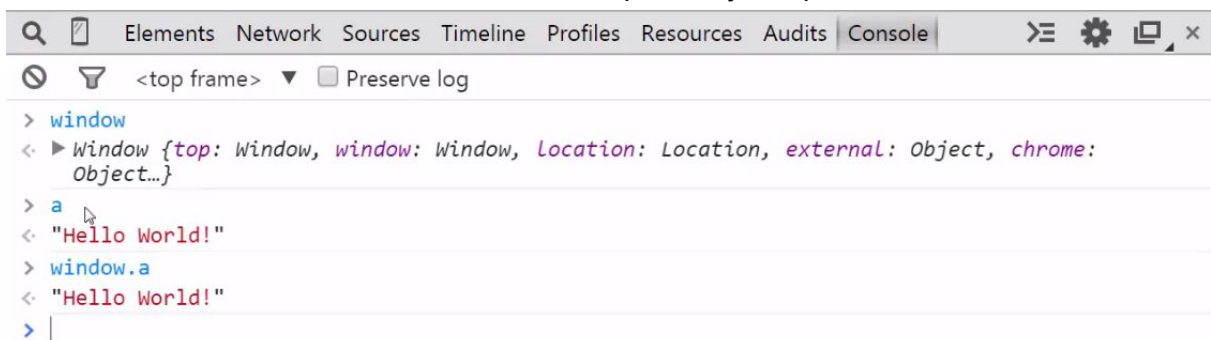
</head>
<body>

    <script src="app.js"></script>
</body>
</html>
JS:
var a = 'Hello World!';

function b() {

```

U konzoli mozemo pristupiti promenljivoj a i metodi b. U JSu kada kreiramo promenljive i funkcije, a nismo unutar neke funkcije, one budu prikacene za globalni objekat. Ako kucamo a ili window.a dobicemo vrednost nase promenljive ispisanu u konzoli .



Sinhrono izvršavanje na jednoj niti

(Single Threaded, Synchronous Execution)

SINGLE THREADED: ONE COMMAND AT A TIME

Under the hood of the browser, maybe not

Možda u brauzeru JS zapravo nije na jednoj niti, ali iz programerske perspektive to tako izgleda.

SYNCHRONOUS: ONE AT A TIME

And in order...

Znači, jedna komadna u jednom momentu i ne prelazi se na sledeću dok trenutna ne završi. Jedna nit, sinhrono.

```
function a() {  
  b();  
  var c;  
}
```

```
function b() {  
  var d;  
}
```

```
a();  
var d;
```

b()
Execution Context
(create and execute)

a()
Execution Context
(create and execute)

Global Execution Context
(created and code is executed)

```
function a() {  
  b();  
  var c;  
}
```

```
function b() {  
  var d;  
}
```

```
a();  
var d;
```

a()
Execution Context
(create and execute)

Global Execution Context
(created and code is executed)

```
function a() {  
  b();  
  var c;  
}
```

```
function b() {  
  var d;  
}
```

```
a();  
var d;
```

Global Execution Context
(created and code is executed)

Lanac Obima

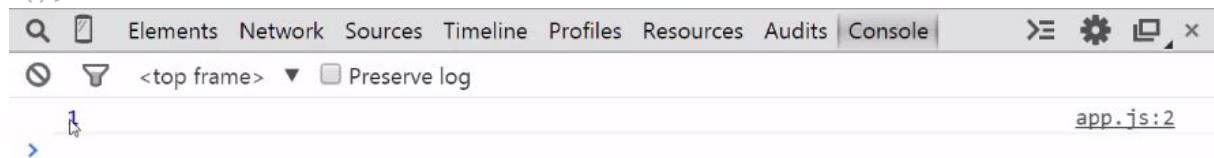
(The Scope Chain)

Primer 7:

```
function b() {  
  console.log(myVar);  
}
```

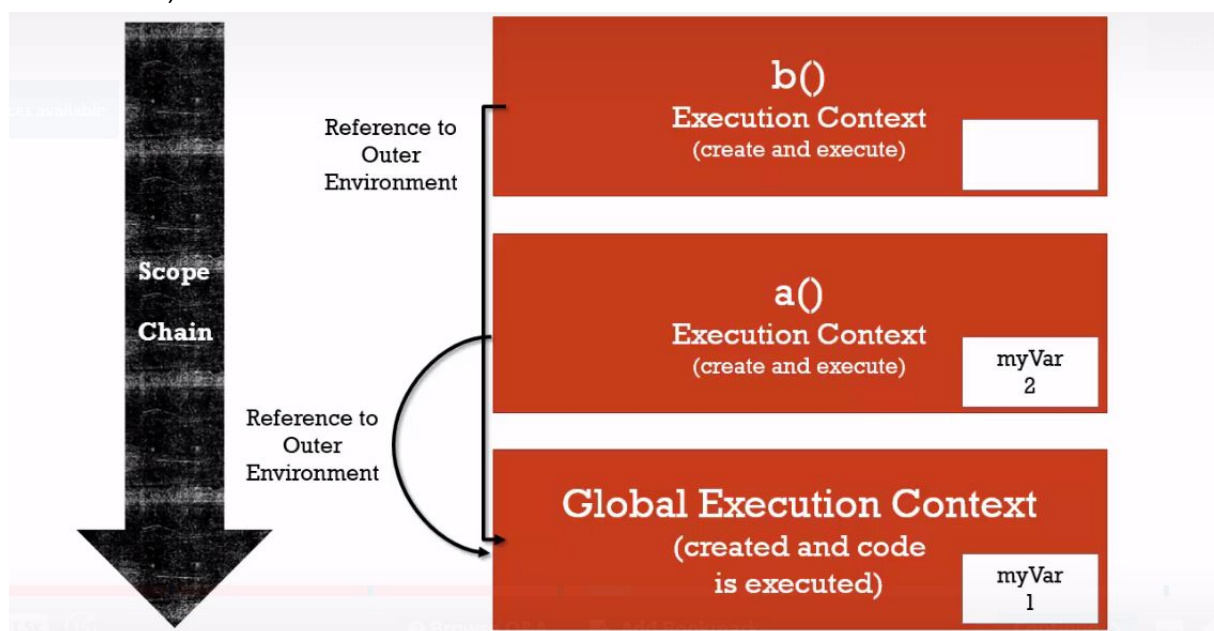
```
function a() {  
  var myVar = 2;  
  b();  
}
```

```
var myVar = 1;  
a();
```



Zasto 1?

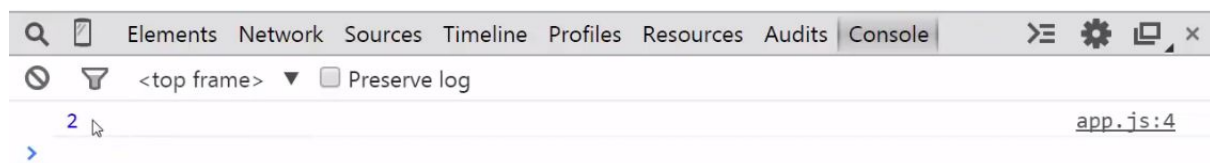
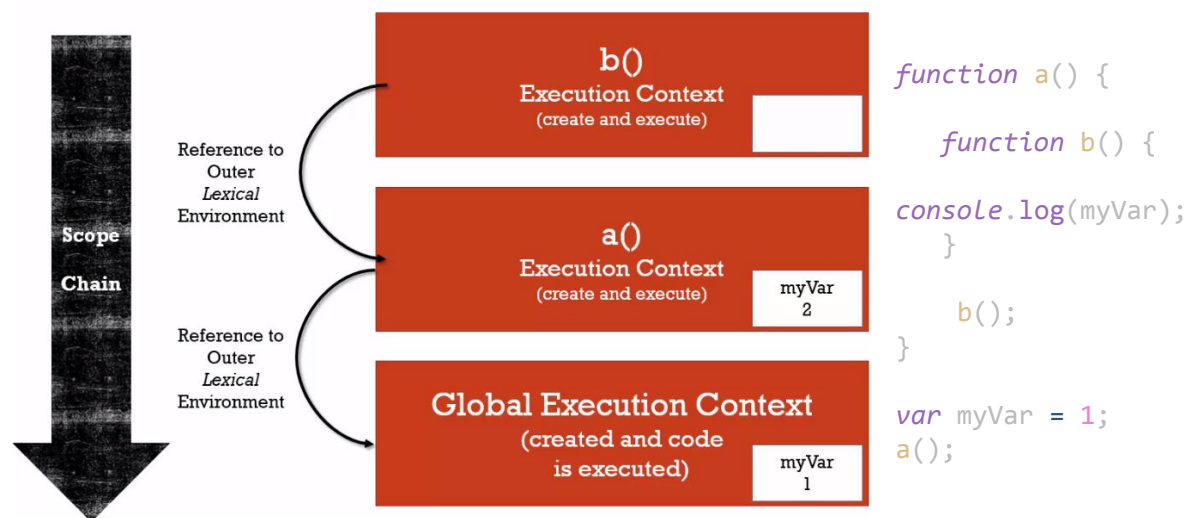
Svaki kontekst izvršavanja ima referencu ka svom **spoljašnjem okruženju** (outer environment)



svom spoljašnjem **leksičkom okruženju** (pričali smo o njemu na početku). Ako JS ne može da nađe promenljive koje pozivamo u našem okruženju tražiće ih u prvom iznad.



Primer 8:



Spoljno okruženje/ Doseg promenljivih (Outer environment):

<https://stackoverflow.com/questions/500431/what-is-the-scope-of-variables-in-javascript>

My bad, sry guys. Ali ovde imate super kul primere :) OBAVEZNO Pročitajte

Obim

(Scope)

SCOPE: WHERE A VARIABLE IS AVAILABLE IN YOUR CODE

And if it's truly the same variable, or a new copy

ES6

(Imate link na početku skripte ka EcmaScript-u :) **read it!**)

EcmaScript 6 je uveo nov način deklarisanja promenljive 2015godine i to je **let** - dozvoljava JS engine-u da koristi nešto što se zove **block scoping**. Koristi se deklarisanje varijable. Možemo deklarirati promenljivu bilo gde u kodu, ali ako je pozovemo pre trenutka njenog izvršavanja, JS engine će vratiti grešku, a ne undefined, iako joj je ovo defaultna vrednost, kao ranije. I dalje postoji sačuvano mesto u memoriji za tu promenljivu, ali ovim se ograničava njeno korišćenje pre same inicijalizacije promenljive. Pogledajmo sledeći primer:

```
if (a>b) {  
    let c = true;  
}
```

Druga stvar koja je bitna, jeste da je ova promenljiva vidljiva samo u okviru svog bloka - blok predstavlja bilo koji deo koda koji je oivičen uglastim zagradama, u if-u, petlji ... Na primer, ako imamo let u okviru for petlje, svaki put kada se prolazi kroz petlju kreiraće se nova promenljiva sa istim nazivom ali je ograničena samo za korišćenje u petlji u tom momentu inkrementa.

SAVRŠENO OBJAŠNJENJE BLOCK SCOPING-A:

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/block>

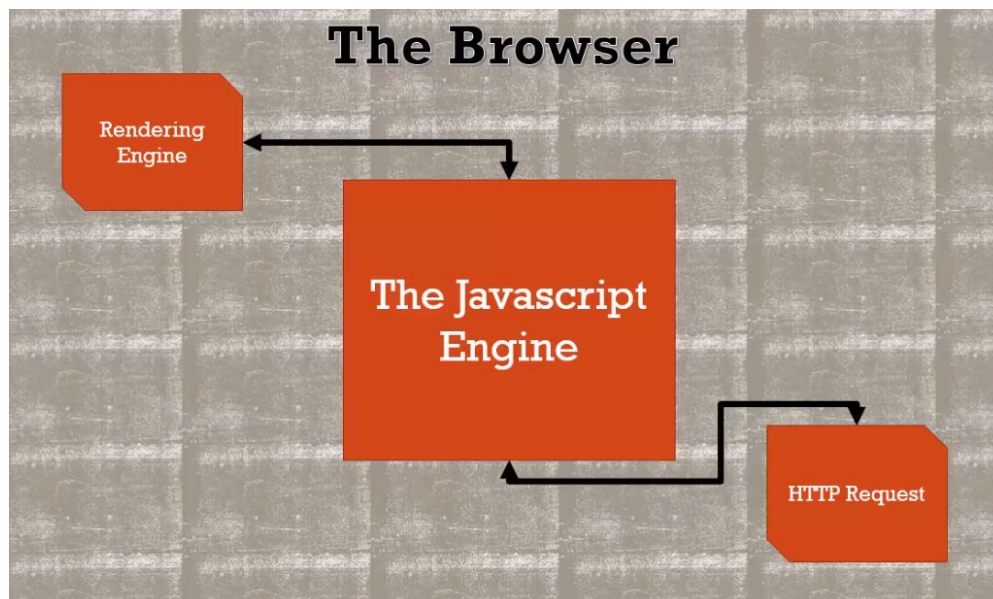
Asinhroni callback

(Asynchronous callback)

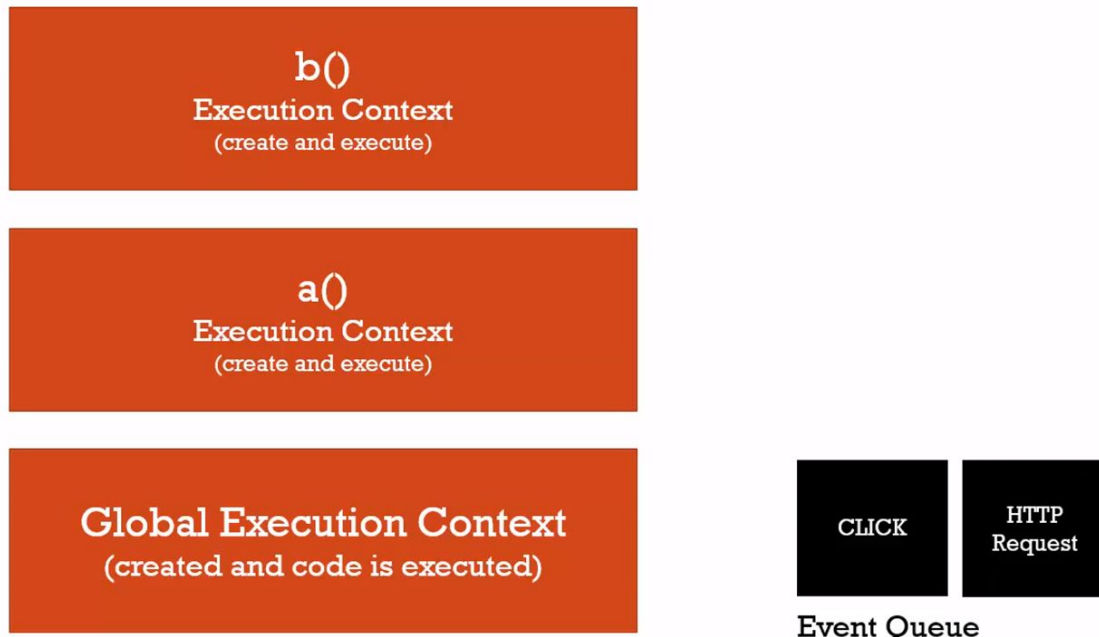
Asinhrono - Više od jedne stvari u isto vreme

Kako JS kao sinhroni jezik onda rešava asinhronu pozive.

JS engine nije jedini engine koji postoji u browser-u. Engine ima kuke(hooks) preko kojih se povezuje sa drugim elementima.



Imali smo Execution stack, gde se slažu jedan na drugi naši konteksti izvršavanja određenih delova koda. Pored njega postoji još jedan red, ali zove se **Event Queue**. On se puni događajima, DOM događajima, klik ili HTTP zahtev i slično, koji se okidaju tokom rada aplikaciju.



JS engine će gledati Event Queue tek kada se Execution Stack isprazni. Kada vidi da je za neki event potrebno izvršiti određeni kod, funkciju koja stoji iza događaja, on kreira novi Exe. Context i obrađuje taj događaj. I tako dalje dok ne obradi sve događaje.



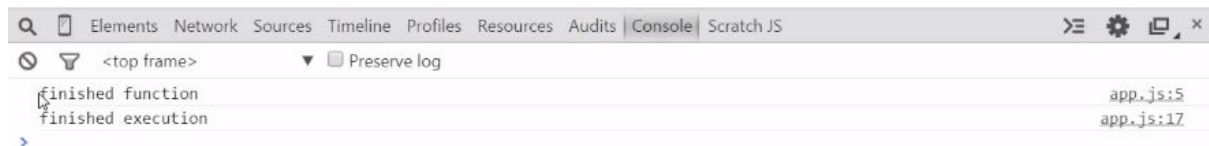
Iako se čini da JS radi asinhrono on i dalje u pozadini izvršava liniju po liniju koda. Hajde da vidimo na primeru

```
function waitThreeSeconds() {  
    var ms = 3000 + new Date().getTime();  
    while (new Date() < ms){}  
    console.log('finished function');  
}  
function clickHandler() {  
    console.log('click event!');  
}  
// osluškujemo klik događaj  
document.addEventListener('click', clickHandler);
```

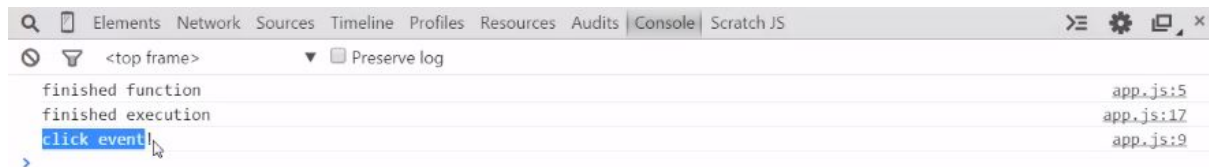


```
waitThreeSeconds();  
console.log('finished execution');
```

Rezultat:



Šta će se desiti ako za vreme loadovanja stranice, kliknemo na stranicu?



Event Queue se izvršava tek kada svi drugi konteksti u steku se završe.

Operatori

(Operators)

Specijalne funkcije koje sintaksno se drugačije pišu (dummy definition)

[Lista svih operatora u JSu](#)

Postoje tri vrste operatora:

1. Binarni
`operand1 operator operand2 : a+b`
2. Unarni
`operator operand : ++a` ili `operand operator : a++`
3. Samo jedan ternarni/trojni - kondicioni ili uslovni operator ?
`condition ? val1 : val2 : var status = (age >= 18) ? 'adult' : 'minor';`

Pored prethodne podele postoji i sledeća:

1. Operatori dodele vrednosti
2. Operatori za upoređivanje
3. Aritmetički operatori
4. Logički operatori
5. String operatori
6. Kondicioni operator
7. Comma (zarezni) operator
8. Bitwise operatori*
9. Unarni operator (delete, typeof, void, in, instanceof)

Prednost operatora i asocijativnost

Prednost operatora bukvalno se odnosi na to koji operator ima prednost kada se u jednoj liniji koda nađe niz operatora, odnosno koji se prvi poziva.

$a+b=c$

Operator sabiranja ima prednost u odnosu na operator dodele vrednosti, zato što neće reći da je b jednako c pre nego što ja saberemo sa a.

Asocijativnost određuje na koji način će se prasiirati operatori sa istim ponderom prednosti.

$a=b=5$

Znak jednakosti, kao dodele vrednosti, odnosno znak dodela (znak jednakosti u JSu je ==) ima sa desna na levo asocijativnost, što znači da će se u našem primeru prvo dodeliti 5 promenljivoj b, pa će se vrednost b dodeliti promenljivoj a, znači obe će imati vrednost 5.

[Listu operatora sortiranih po prednosti](#) sa svojim asocijativnim vrednostima.

Najveću prednost od operatora imaju zagrade ;)

Koercija - prinudna promena tipa
(Coercion)

COERCION: CONVERTING A VALUE FROM ONE TYPE TO ANOTHER

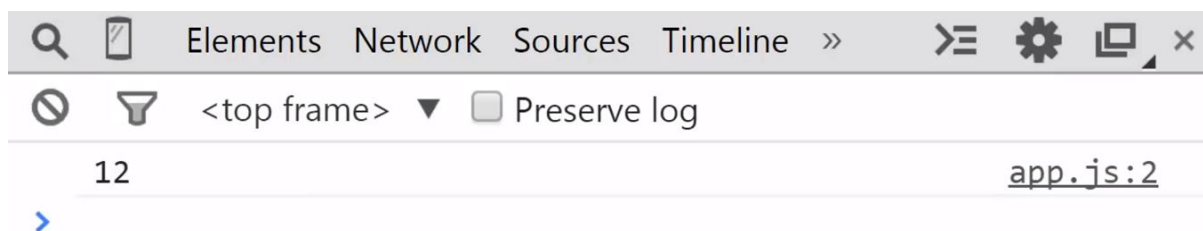
This happens quite often in Javascript because it's dynamically typed.

Prevedimo ga kao prinudno menjanje tipa promenljive.

```
var a = 1 + '2'  
console.log(a);
```

Šta će se odštampati u konzoli ?

Rezultat:



Zašto 12? Pa JS Engine je skontao da mi nešto sabiramo, ali nije bio siguran šta, pošto ne zna da li sa sigurnošću može da parsira string u broj, on radi ono što sigurno neće izbaciti grešku, a to je da parsira broj u string. Ovo je jedna od posledica dinamičkog kucanja, jer JS je kreiran kao takav da može dosta toga da dozvoli umesto što bi izbacio grešku.

Operatori komparacije

Iako su intuitivno jasni, postoje neke začkoljice kod operatora komparacije

Primer 1:

```
console.log(1<2<3);
```

Rezultat:



Nema puno razmišljanja, jer to je logično da je tačno. A šta ako zamenimo mesto 3 i 1

```
console.log(3<2<1);
```

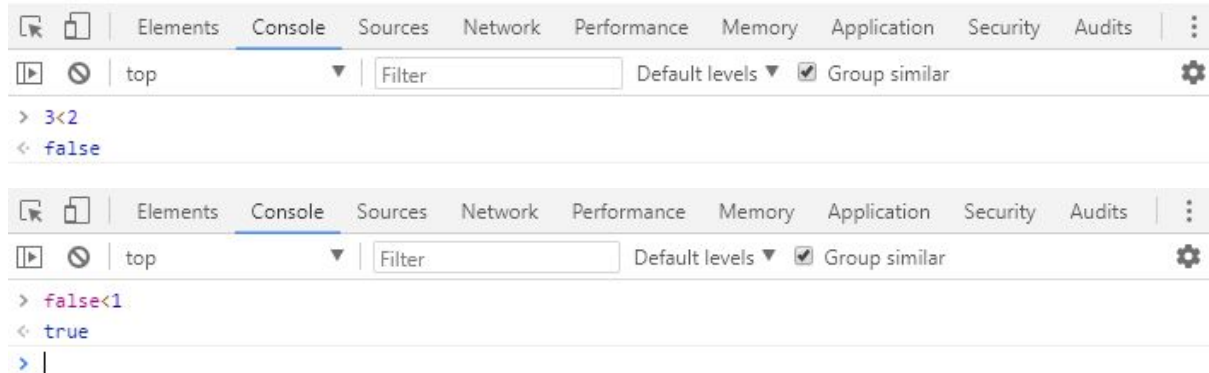
Šta bi bio logičan odgovor?

False

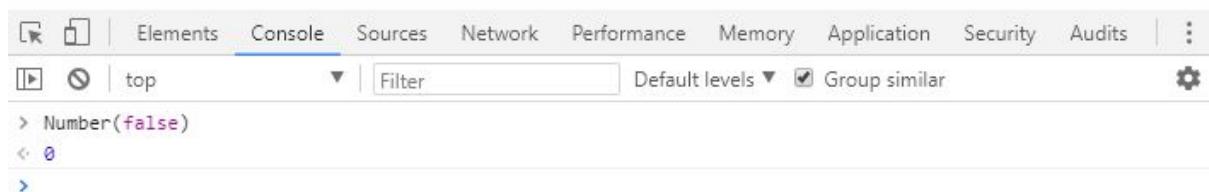
Rezultat:



Zašto je true? Pogledajmo tabelu prednosti operatora i asocijativnost za operator <. Gleda se sa leva na desno.



Zašto je false manje od 1? Zato što je koercija prinudno pretvorila false u njegovu numeričku reprezentaciju što je 0.



Number je built-in funkcija JSa, koja se uglavnom pozadinski koristi pri izvršavanju funkcija, ali gotovo nikad u samom kucanju koda. Ako pogledamo prvi primer još jednom, možemo zaključiti da ono nije jednako zbog same logike znaka manje, već zbog koercije.

1 < 2 je true, a Number(true) je 1, pa je samim tim 1 < 3.

Jednakost

U tabeli operatora možemo videti da postoje četiri vrste jednakosti:

1. Jednakost ==
2. Nejednakost !=
3. Striktna jednakost ===
4. Striktna nejednakost !==

Striktna (ne)jednakost poredi vrednosti, ali ne vrši prinudnu promenu tipa.

[Na ovom linku](#) možete da nađete više informacija o jednakosti i poređenju

Kada je koerzija korisna?

```
var a;  
if (a){  
    console.log("nesto")  
}
```

Rezultat:



If se izvršava samo ako ono što je unutar zagrade vraća true. Kao što smo imali numeričke ugrađene funkcije imamo i Boolean.

```
> Boolean(a)  
< false  
> |
```

Što znači da se naš uslov ne prolazi i zbog toga se ništa ne štampa.

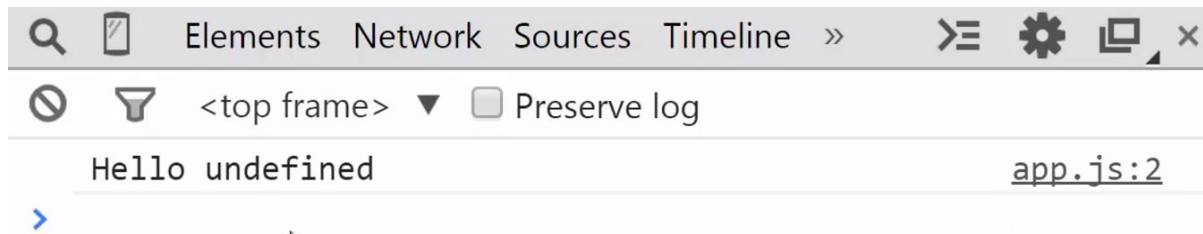
```
var a;  
a=0;  
if (a || a === 0){  
    console.log("nesto")  
}
```

Ova upotreba koerzije da bi se ustanovilo postojanje nečega u memoriji, se često koristi kao dobra praksa u mnogim frejmrvcima. ;)

Difoltne vrednosti

```
function greet(name){  
    console.log('Hello '+name);  
}  
greet();
```

Rezultat:



JS ne interesuje što vi niste prosledili argumente, koje neka funkcija treba da bi se pravilno izvršila. On će parametrima dodeliti difoltne vrednosti i nastaviti kao da je sve okej.

Postoji par načina na koji možete sami dodeliti određenim promenljivama difoltne vrednosti, a da one ne budu undefined.

```
function greet(name){  
    name = name || '<Your name here>';  
    console.log('Hello '+name);  
}  
greet();
```

|| operator ne vraća samo true ili false, već vraća i vrednost koja se može koerzijom prinudno promeniti u true.



U ovom slučaju vraća hello, jer undefined sigurno ne može da vrati true. Ako stavimo dve vrednosti koje mogu vratiti true, vrati će prvu, jer to je način na koji || operator funkcioniše - prekida sa prolaženjem kroz ostale uslove kada naiđe na prvi koji može biti pretvoren u true.



Tako da u našem primeru `name = name || '<Your name here>'`; JS engine će logovanju proslediti prvu vrednost koja bude vratila true, što znači ako unesemo neko ime kao parametar, vrati će nam to ime kao vrednost parametra u suprotnom će nam vratiti `'<Your name here>'`.

Difoltne vrednosti u frejmrvcima

Napravimo tri JS fajla i jedan dodajmo ih u naš index.html:

lib1.js:

```
var libraryName= "Lib1";
```

lib2.js:

```
var libraryName= "Lib2";
```

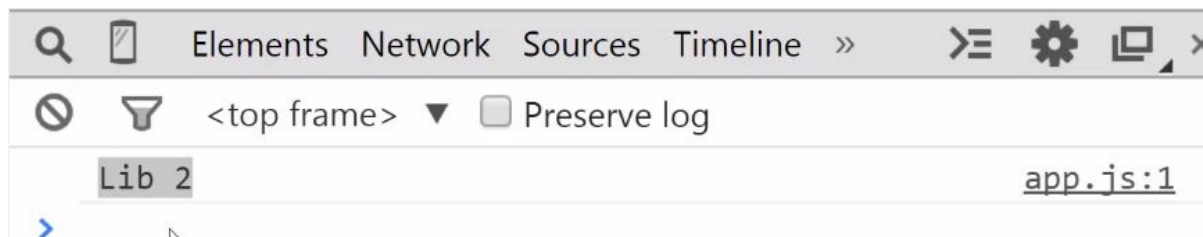
app.js:

```
console.log(libraryName);
```

index.html:

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>
  <script type="text/javascript"
src="lib1.js"></script>
  <script type="text/javascript"
src="lib2.js"></script>
  <script type="text/javascript"
src="app.js"></script>
</body>
</html>
```

Rezultat:



Skripte se stekuju jedna na drugu i JS engine ih pakuje u jedan veliki fajl, tako čak i ako imate uključene neke biblioteke i frejmrke, JS engine sav taj JS kod strpava u jedan veliki JS fajl. Šta je keč? Šta ako nazove promenljivu isto kao što se ona zove u nekoj biblioteci. Pa možemo uvek da proverimo da li neka promenljiva postoji, ako postoji tu vrednost ćemo koristiti, ako ne postoji kreiraćemo je i dodeliti joj novu:

lib2:

```
window.libraryName =window.libraryName || "Lib2";
```

Kontrolne strukture i petlje

Naredbe i struktura JavaScripta veoma podsećaju na onu koja se koristi u jezicima Java, C++ i C.

JavaScript program je izgrađen iz funkcija, naredbi, operatora i izraza. Osnovna jedinica je naredba ili izraz koji se završava sa tačkom-zarezom.

```
document.writeln("Pocetak!<BR>");
```

Prethodna komanda poziva writeln() metod, koji je deo document objekta. Tačka-zarez ukazuje na kraj komande. JavaScript komanda se može prostirati u više redova. Slično, može se više naredbi naći u jednom redu, dokle god se završavaju tačkom-zarezom.

Možemo grupisati naredbe u blokove naredbi, izdvojene velikim zagradama:

```
{
    document.writeln("Da li ovo radi? ");
    document.writeln("Radi!<BR>");
}
```

Blokovi naredbi se koriste u definiciji funkcija i kontrolnim strukturama.

Jedna od osnovnih mogućnosti svakog programskog jezika je da pošalje tekst na izlaz. U JavaScriptu izlaz može biti preusmeren na nekoliko mesta uključujući trenutni prozor dokumenta i pop-up dijalog. Osnovni izlaz je preusmeravanje teksta u prozor WWW klijenta, što se obavlja prosleđivanjem HTML koda. Rezultujući tekst će biti interpretiran kao HTML kod i prikazan u prozoru WWW klijenta. To ostvarujemo sa metodima write (šalje tekst u prozor WWW čitača bez pomeranja) i writeln (isto kao write(), s tim što se posle ispisa teksta kurzor pomera u sledeći red) objekta document:

```
document.write("Test");
document.writeln('Test');
```

Vrste naredbi koje donose odluke i ponavljaju se u petlji se nazivaju kontrolne strukture. Važan deo komandne strukture je uslov. Svaki uslov je jedan logički izraz koji dobija vrednost true ili false.

Konstrukcija if i if...else.Switch

if

Najjednostavnija odluke u programu jeste praćenje neke grane ili putanje programa ako je ispunjen određen uslov. Sintaksa za ovu konstrukciju je:

```
If(uslov){  
    //Kod koji se izvršava ako je vrednost izraza true  
};
```

Sledi primer kako napisati if naredbu:

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
    <meta charset="UTF-8">  
    <meta name="viewport" content="width=device-width, initial-scale=1.0">  
    <meta http-equiv="X-UA-Compatible" content="ie=edge">  
    <title>Document</title>  
</head>  
<body>  
    <script>  
        var d = new Date()  
        var vreme = d.getHours()  
        if (vreme < 10){  
            document.write("<b>Dobro jutro!</b>")  
        }  
    </script>  
    <p>Ovo je primer If naredbe.</p>  
    <p>Ukoliko je na vašem računaru manje od 10 sati,  
    dobićete poruku: “ Dobro jutro!”.</p>  
</body>  
</html>
```

If...else

Ako su umesto jedne grane potrebne dve ili više koje obrada treba da prati koristi se if...else tj. If...else if...else konstrukcija.

Sintaksa za konstrukciju if...else je:

```
if(uslov){  
    //kod koji se izvršava ako je vrednost izraza true  
}else {  
    //kod koji se izvršava ako je vrednost izraza false  
}
```

Sledi primer za konstrukciju if...else:

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
    <meta charset="UTF-8">  
    <meta name="viewport" content="width=device-width, initial-scale=1.0">  
    <meta http-equiv="X-UA-Compatible" content="ie=edge">  
    <title>Document</title>  
</head>  
<body>  
    <script>  
        var d = new Date()  
        var vreme = d.getHours()  
        if (vreme < 10) {  
            document.write("<b> Dobro jutro!</b>")  
        } else {  
            document.write("<b>Dobar dan!</b>")  
        }  
    </script>  
  
    <p>Ovaj primer ilustruje If ... else naredbu</p>  
    <p>  
        Ukoliko je na vašem računaru manje od 10 sati,  
        dobićete poruku: "Dobro jutro!".  
        U suprotnom, dobićete poruku: "Dobar dan!".  
    </p>  
  
</body>  
</html>
```


if..else if...else

Konstrukcija if..else if...else pogodna je kada je potrebno pratiti nekoliko izvršnih linija.

Sintaksa:

```
if(uslov1){  
    //Kod koji se izvrsava ako je vrednost izraza uslov1 true  
}else if (uslov2) {  
    //Kod koji se izvrsava ako je vrednost izraza uslov2 true  
}else{  
    //kod koji se izvrsava ako ni jedan od izraza uslov1 I uslov2 nema vrednost  
    true  
}
```

Primer za konstrukciju if...else if...else:

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
    <meta charset="UTF-8">  
    <meta name="viewport" content="width=device-width, initial-scale=1.0">  
    <meta http-equiv="X-UA-Compatible" content="ie=edge">  
    <title>Document</title>  
</head>  
<body>  
    <script>  
        var d = new Date()  
        var vreme = d.getHours()  
        if (vreme < 10) {  
            document.write("<b> Dobro jutro!</b>")  
        }else if (vreme < 18){  
            document.write("<b>Dobar dan!</b>")  
        }else {  
            document.write("<b>Dobro večer!</b>")  
        }  
    </script>  
    <p>  
        Ovaj primer ilustruje If ... else If ... else naredbu  
    </p>  
</body>  
</html>
```

Switch

Pod nekim okolnostima, odluka tipa true ili false nije dovoljna za obradu podataka u skriptu. Svojstvo objekta ili vrednost promenljive mogu sadržati bilo koju od nekoliko vrednosti i potreban je poseban put izračunavanja za svaku od njih. U JavaScriptu postoji kontrolna struktura koju koriste mnogi jezici. Na početku strukture se identifikuje o kom izrazu se radi i svakoj putanji izvršavanja dodeljuje se oznaka koja odgovara određenoj vrednosti. U pitanju je switch naredba.

```
switch(n) {  
    case 1:  
        //izvrši blok1 koda  
        break  
    case 2:  
        //izvrši blok2 koda  
        break  
    default:  
        //kod koji se izvršava ako je n različito od vrednosti datih u  
        slučajevima 1 i 2  
}
```

Sledeći primer pokazuje kako napisati switch naredbu:

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
    <meta charset="UTF-8">  
    <meta name="viewport" content="width=device-width, initial-scale=1.0">  
    <meta http-equiv="X-UA-Compatible" content="ie=edge">  
    <title>Document</title>  
</head>  
<body>  
    <script>  
        var d = new Date()  
        dan=d.getDay()  
        switch (dan)  
        {  
            case 5:  
                document.write("<b>Konačno petak!</b>")  
                break  
            case 6:  
                document.write("<b>Subota URAAA!</b>")  
            }  
        }  
    </script>  
</body>  
</html>
```

```

        break
    case 0:
        document.write("<b>Nedelja odmor!</b>")
        break
    default:
        document.write("<b>Kad će vikend?!</b>")
    }
</script>
<p>U ovom primeru se generiše različita poruka u zavisnosti od dana u
nedelji. Primetite da je nedelja=0, ponedeljak=1, utorak=2, itd.</p>
</body>
</html>

```

Naredba default obezbeđuje nastavak po putanji izvršavanja kada vrednost izraza ne odgovara ni jednoj oznaci naredbe case.

Naredba break koja služi za izlazak iz petlje, ovde ima značajnu ulogu. Naime, ako nije navedeno break posle svake grupe naredbi u case granama, izvršiće se sve naredbe iz svake case grane bez obzira na to da li je nađena odgovarajuća oznaka.

For petlja. Konstrukcije while i do..while

Često kada se piše kod, želi se da se neki deo koda ponovi više puta zaredom. Umesto dodavanja nekoliko gotovo identičnih linija koda u script-u, koriste se petlje da bi se postigao identičan rezultat.

U Java Scriptu postoje dve vrste petlji:

- for-kada želimo da se deo koda izvrši tačno određen broj puta
- while-kada želimo da se određen deo koda izvršava sve dok je određen uslov zadovoljen.

for (; ;)

Formalna sintaksa za for petlju je:

```

for (početna_vrednost;uslov; uvećanje/smanjenje){
    //kod koji se izvršava
}

```

Primer ispisuje tri nivoa naslova:

```

for(i=1;i<=3;i++){
    document.write("<H"+i+"> Naslov na nivou " + i+ "</H"+i+">")
}

```

Sledeći primer označava petlju koja počinje od vrednosti $i=0$. Petlja će se ponavljati sve dok i ima vrednost manju ili jednaku sa 10. Svaki put kada se petlja izvrši vrednost i će se povećati za 1.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>
<body>
  <script>
    var i=0
    for (i=0;i<=10;i++){
      document.write("Broj je: " + i)
      document.write("<br />")
    }
  </script>
</body>
</html>
```

for .. in

Pomoću nje možemo proći kroz sve osobine (properties) nekog objekta. Koliko jedan objekat ima osobina, toliko puta će se izvršiti ova for petlja.

```
for (promenljiva in objekat){
  //naredbe
}
```

while (var<=endvalue)

Formalna sintaksa za while petlju je:

```
while (uslov){
  //kod koji se izvršava
}
```

Ova petlja izvodi akciju sve dok izraz uslov ne dobije vrednost false.

do..while

JavaScript nudi još jednu konstrukciju petlje zvanu do..while. Formalna sintaksa za ovu konstrukciju je sledeća:

```
do{  
    //naredbe  
}  
while(uslov)
```

Razlika između while i do..while petlje je ta što se u do..while petlji naredbe izvršavaju bar jednom pre nego što se uslov ispita, dok u petlji while to nije slučaj.

Break i continue

Break naredba se koristi da bi se iskočilo iz petlje.

Continue naredba se koristi da bi se iskočilo iz tekuće petlje i nastavilo sa narednom vrednošću.

Break

Naredba break će prekinuti petlju i nastaviti izvršenje od prve linije koda koja sledi nakon petlje.

Primer:

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
    <meta charset="UTF-8">  
    <meta name="viewport" content="width=device-width, initial-scale=1.0">  
    <meta http-equiv="X-UA-Compatible" content="ie=edge">  
    <title>Document</title>  
</head>  
<body>  
    <script>  
        var i=0  
        for (i=0;i<=10;i++){  
            if (i==3){break}  
            document.write("Broj je: " + i)  
            document.write("<br />")  
        }  
    </script>  
</body>  
</html>
```

continue

Nareba continue će prekinuti tekuću petlju i nastaviti sa sledećom vrednošću.

Primer:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>
<body>
  <script>
    var i=0
    for (i=0;i<=10;i++){
      if (i==3){continue}
      document.write("Broj je: " + i)
      document.write("<br />")
    }
  </script>
</body>
</html>
```

Funkcije

Funkcija je deo koda koja se izvršava kada se dogodi neki događaj ili kada je funkcija pozvana.

Primer za poziv funkcije:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
  <script>
    function mojaFunkcija (){
```

```

        alert("Zdravo!!!")
    }
</script>
</head>
<body>
    <form>
        <input type="button" onClick=" mojaFunkcija ()" value="Pozovi
funkciju">
    </form>
    <p>Pritiskom dugmeta pozvaće se funkcija. Funkcija će izbaciti alert prozor
sa porukom</p>
</body>
</html>

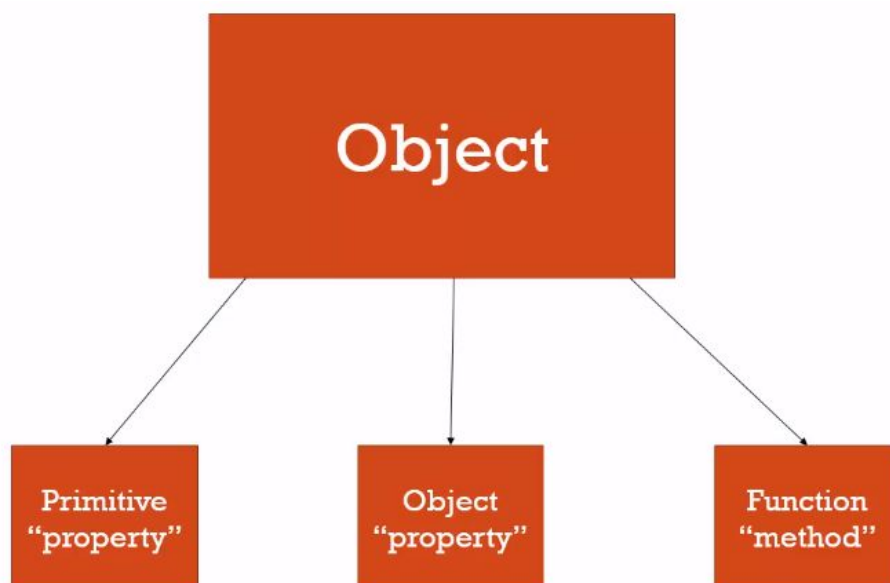
```

Objekti i funkcije

U JS, ono što je bitno da znamo jeste da su funkcije i objekti slična, ako ne ista, stvar. Jer, ako izuzmemo mogućnost kreiranja objekta u JSu, kao u bilo kom drugom OO jeziku (*new Object*), JS je kreiran kao funkcionalni jezik i sve je zamišljeno da se kreira preko funkcije pa čak i objekti.

Objekti i tačka

Objekti su kolekcije name-value parova, gde value može biti kolekcija name-value parova. Objekat može imati metode i atribut (property)



Primer:

```
var person = new Object();
```

Kreirali smo objekat i sad želimo da mu dodamo par atributa. Postoji više načina za to.

```
person["firstname"] = 'Tamara'
```

Ovime smo kreirali novi atribut i dodelili mu vrednost 'Tamara'. Ovo je jedan od načina dinamičkog kreiranja atributa.

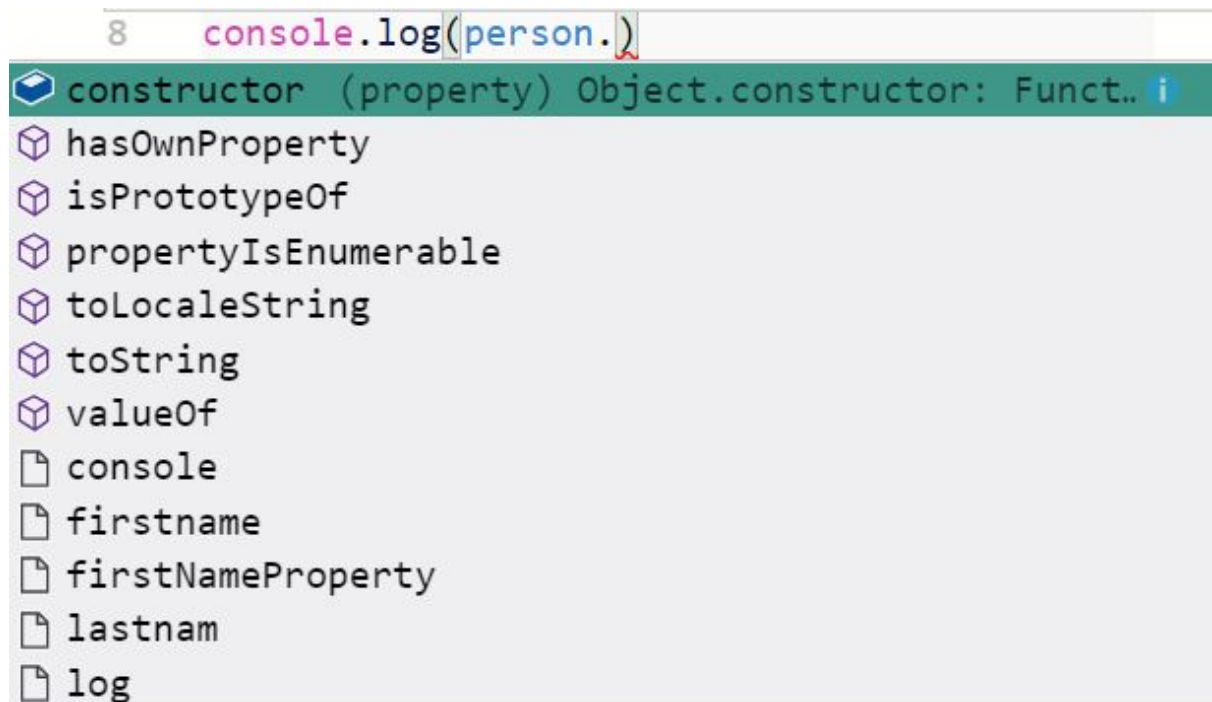
[] su [operator pristupa atributima](#). Tačka radi istu stvar.

Ako hoću da pristupim nekom atributu lako to mogu da uradim, takođe sa uglastim zagradama i unosom atributa kojem želim da pristupim, ili ako hoću da zakomplikujem mogu da

```
var firstNameProperty = "firstname";  
console.log(person[firstNameProperty]);
```

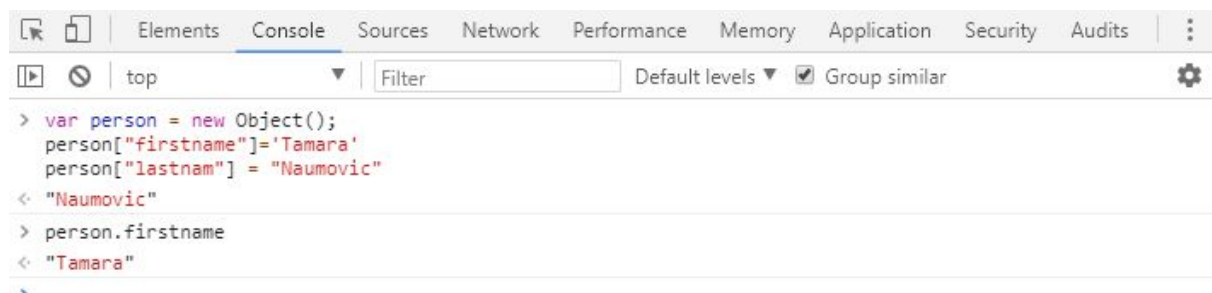
Nekoj promenljivoj sam dodelila vrednost "firstname" koja kad se prosledi samom objektu, zapravo traži atribut sa tim nazivom.

Ali ovaj pristup nećete često viđati. Ono što hoćete je tačka.



Ako ukucate tačku posle objekta person, pojaviće se lista svih atributa i metoda koje su dostupne ovom objektu. Tačkom pristupamo svemu što jedan objekat poseduje.

```
console.log(person.firstname)
```



Tačka i [] su operatori koji traže članove (Computed Member Access).

Objekti i Literali

```
var person = {};  
console.log(person);
```

{ } omogućavaju kreiranje literala. JS Engine to vidi kao kreiranje objekta, praznog objekta. Sa tim što nam je na ovaj način lakše da kreiramo attribute koje ćemo vezati za naš objekat

```
var person = {  
  firstname: 'Tamara',  
  lastname: 'Naumovic',  
  address: {  
    street: 'Jove Ilica',  
    Number: 154,  
  }  
};  
console.log(person.firstname);
```

Hajde da vidimo sledeći primer i još jednu upotrebu dinamičkog kucanja

```
function greet(person){  
  console.log('Hello ' + person.firstname);  
}  
greet({  
  firstname: 'Marko',  
  lastname: 'Markovic'  
});
```

Ne moramo kreirati objekat pre nego što ga prosledimo nekoj funkciji ili metodi, ali zapamtimo samo da nemamo fizičku referencu ka objektu koji kreiramo pri inicijalizaciji koda.

JSON i Literal

JSON i literal nisu isto. JSON je inspirisan i kreiran na osnovu sintakse literala ali nije identično literalu.

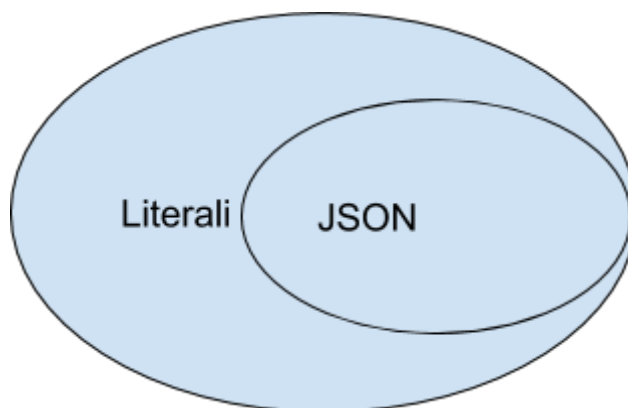
Literal

```
var personLiteral = {  
  firstname:"Tamara",  
  yearsOld:24  
};
```

```
var personJSON = {  
  "firstname":"Tamara",  
  "yearsOld":24  
};
```

JSON

Kod JSONa svi atributi moraju da se deklariraju sa navodnicima, gde to kod literala nije potrebno, ali je dozvoljeno i moguće. Tako da JSON mora da ima navodnike, literal može ali ne mora.



Svaki JSON je validan literal, ali svaki literal nije validan JSON format.

Zbog ovoga postoje neke ugrađene funkcije u JS koje bi obezbedile lakšu konverziju između dva.

```
var jsonValue = JSON.parse('{"firstname":"Tamara","yearsOld":24}');  
console.log(jsonValue);  
var person = {firstname:'Tamara',yearsOld:24}  
console.log(JSON.stringify(person));
```

```
▶ {firstname: "Tamara", yearsOld: 24}
```

VM404:2

```
{"firstname":"Tamara","yearsOld":24}
```

VM404:4

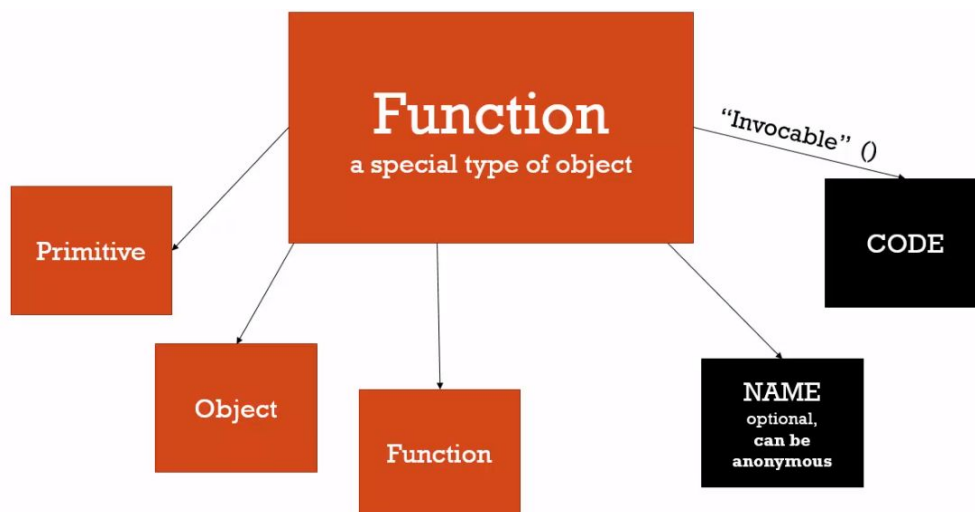
Dobijamo objekat, literal, nasto parsiranjem JSONa i dobijamo JSON parsiranjem objekta person.

Funkcije prve klase

FIRST CLASS FUNCTIONS: EVERYTHING YOU CAN DO WITH OTHER TYPES YOU CAN DO WITH FUNCTIONS.

Assign them to variables, pass them around, create them on the fly.

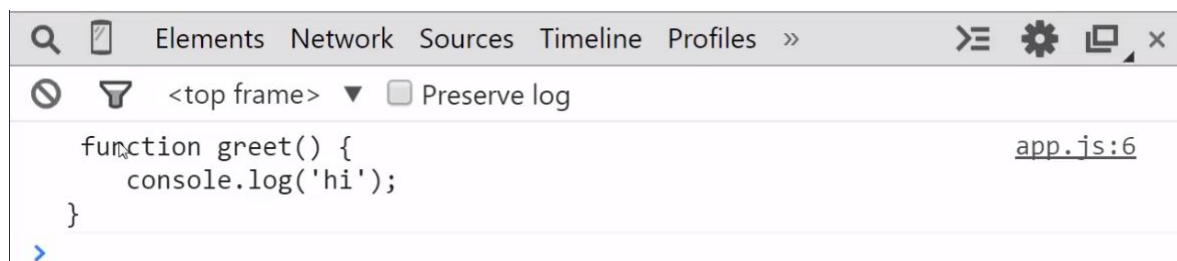
Na prvoklasne funkcije možemo dodati bilo šta, atribut, metode ili druge funkcije (što se često koristi, pričaćemo o tome)



Kod je samo još jedan od atributa funkcije. ;)

```
function greet(){  
  console.log('hi');  
}  
greet.language = 'english';  
console.log(greet);
```

Rezultat:



```
console.log(greet.language);
```

Rezultat:



U JSu funkcije su objekti.

Funkcije kao izjave i kao izrazi

(Function statements and Functions expressions)

EXPRESSION: A UNIT OF CODE THAT RESULTS IN A VALUE

It doesn't have to save to a variable.

Primer:

```
var a;
```



Izjave, samo odrađuju posao, ne vraćaju nikakvu vrednost

```
if (a===3){  
}
```

If je izjava (if statement) , uslovna izjava, kojoj se ne može dodeliti nikakva vrednost, već samo izvršava određeni deo koda.

Primer Izjavne funkcije:

```
function greet(){  
  console.log('hi')  
}
```

Primer Izrazne funkcije:

```
var anonymousGreet = function(){
```

```
    console.log('hi')
}
```

Ovo je anonimna funkcija i možda ste čuli za njih, funkcija koja nema ime ali ima svoje mesto u memoriji, ime joj u ovom slučaju nije potrebno jer smo je dodelili kao vrednost promenljivoj koja ima svoje mesto i referencu ka tom mestu u memoriji. Kako pozivamo ovu funkciju da se izvrši?

```
anonymousGreet();
```

Šta će se desiti ako funkciju pozovemo pre njene deklaracije?

```
anonymousGreet();
```

```
var anonymousGreet = function(){
    console.log('hi')
}
```

Rezultat:

hi	app.js:4
✖ ▶ Uncaught TypeError: undefined is not a function	app.js:7

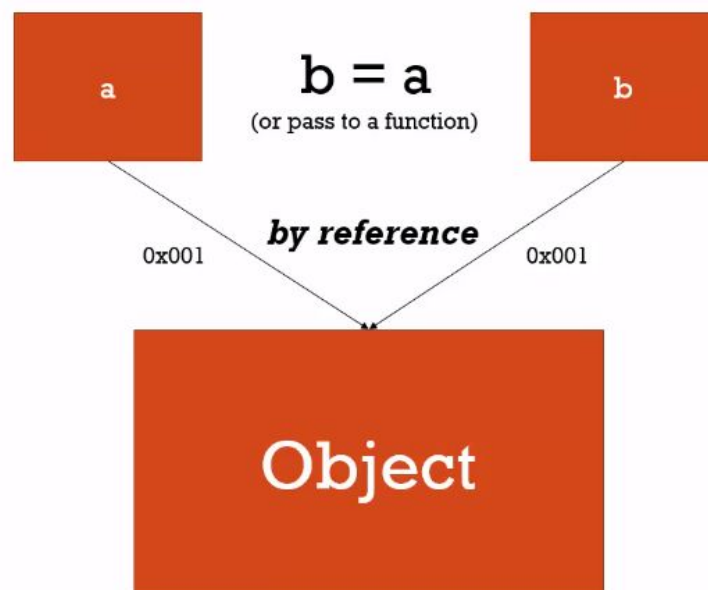
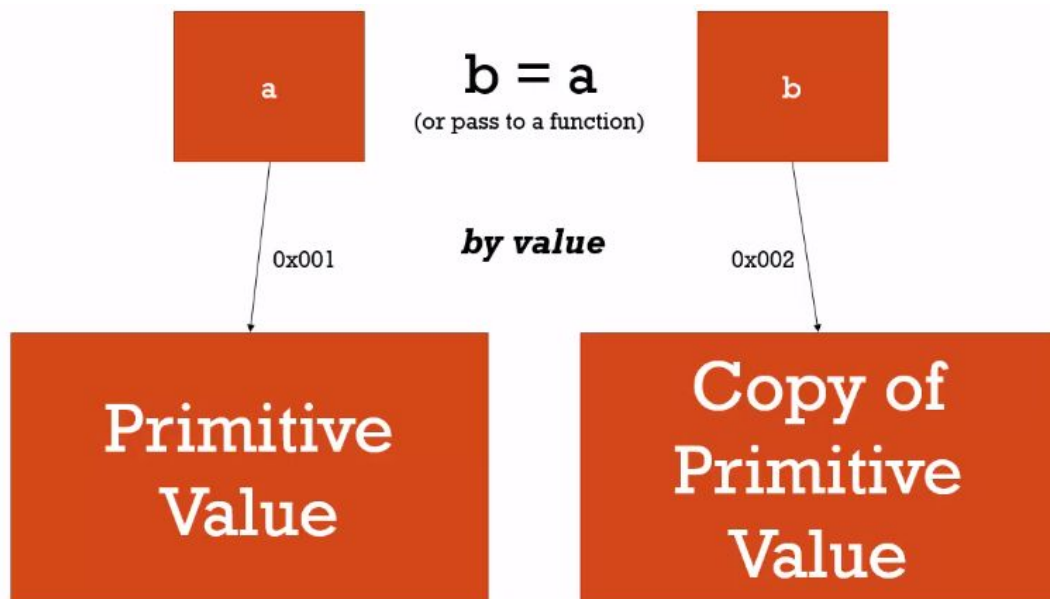
Zato što je za `anonymousGreet` u memoriji rezervisao memoriju i predefinisano vrednost `undefined`

Hajde da pogledamo jos jedan zanimljiv primer

```
function log(a){
    console.log(a);
    a();
}
log(function(){
    console.log('hi')
})
```

Anonimna funkcije se može kreirati dinamički, kao i sve do sada što smo uspevali da kreiramo, ovo je sam kor funkcionalnog programiranja i vrlo često se koristi kod kreiranja frejmvrkova.

Po vrednosti / Po referenci



Kada imate dve varijable koje pokazuju kao u gornjem slučaju na isti objekat, istu memorijsku lokaciju, svaka izmena a i b će se primenjivati nad istim objektom. Ali ako su u pitanju prosti tipovi, izmene nad a i b neće menjati isti objekat, već svako svoju vrednost na koju upućuje.

```
var c = {greeting: 'hi'};
var d;
d=c;
c.greeting = 'hello';
console.log(c);
console.log(d);
```

Rezultat: hello, hello

```

var c = {greeting: 'hi'};
var d;
d=c;
c.greeting = 'hello';
console.log(c);
console.log(d);
c={greeting:'howdy'}
console.log(c);
console.log(d);

```

Rezultat:

▶ {greeting: "hello"}	VM406:5
▶ {greeting: "hello"}	VM406:6
▶ {greeting: "howdy"}	VM406:8
▶ {greeting: "hello"}	VM406:9

Ako je vrednost primitivni tip u objektu onda je to atribut, ako je funkcija onda je to metoda.

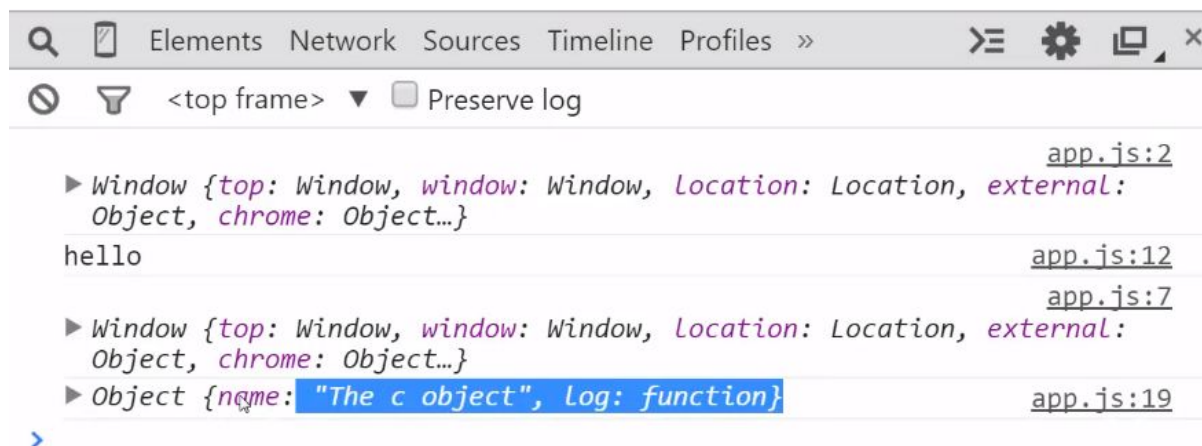
Pogledajmo sledeći primer sa korišćenjem **this**:

```

function a() {
    console.log(this);
    this.newvariable = 'hello';
}
var b = function() {
    console.log(this);
}
a();
console.log(newvariable); // ne bi trebalo ovo da radimo! Ali to je
promenljiva na globalnom nivou
b();
var c = {
    name: 'The c object',
    log: function() {
        console.log(this);
    }
}
c.log();

```

Rezultat:

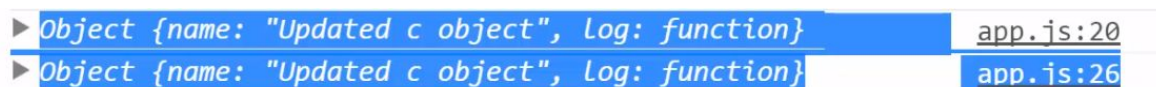


Ako izmenimo malo kod, možemo videti nešto što mnogi ljudi smatraju da je bug, greška u samom JSu

```
var c = {
  name: 'The c object',
  log: function() {
    this.name = 'Updated c object';
    console.log(this);

    var setname = function(newname) {
      this.name = newname;
    }
    setname('Updated again! The c object');
    console.log(this);
  }
}
c.log();
```

Rezultat:



Zašto je ovo greška? Pa, pri kreiranju anonimne funkcije `setName`, njen kontekst kreiranja se našao na samom globalu, kao što to i biva, i time je globalnom objektu dodelio novi atribut `name` koji ima vrednost `'Updated again! The c object'`. Mnogi ljudi misle da je ovo pogrešno, pa se to rešava na sledeći način, kako se ne bi zbunili:

```
var c = {
  name: 'The c object',
  log: function() {
    var self = this;

    self.name = 'Updated c object';
    console.log(self);

    var setName = function(newname) {
      self.name = newname; // ne moramo da brinemo što self nije tehnički
definisan u okviru ovog objekta jer će ga JS engine tražiti šire
    }
    setName('Updated again! The c object');
    console.log(self);
  }
}
c.log();
```

I onda kada god treba da koristimo `this`, zapravo koristimo `self` prethodno kreiran u objektu.

Function overloading

```
function greet(firstname, lastname, language) {
  language = language || 'en';
  if (language === 'en') {
    console.log('Hello ' + firstname + ' ' + lastname);
  }
  if (language === 'es') {
    console.log('Hola ' + firstname + ' ' + lastname);
  }
}

function greetEnglish(firstname, lastname) {
  greet(firstname, lastname, 'en');
}

function greetSpanish(firstname, lastname) {
  greet(firstname, lastname, 'es');
}

greetEnglish('John', 'Doe');
```

```
greetSpanish('John', 'Doe');
```

// umesto da ovde uvek kao parametar funkcije greet prosleđujemo i jezik, mi kreiramo dve funkcije

Overloadovanje je koncept koji dozvoljava programeru da definiše više funkcija istog imena u istom obimu.

Immediately invoked function expressions IIFE

// izjavna funkcija

```
function greet(name) {  
    console.log('Hello ' + name);  
}  
greet('John');
```

// izrazna funkcija

```
var greetFunc = function(name) {  
    console.log('Hello ' + name);  
};  
greetFunc('John');
```

// Immediately Invoked Function Expression (IIFE)

```
var greeting = function(name) {  
  
    return 'Hello ' + name;  
}  
}('John');
```

```
console.log(greeting);
```

// IIFE

```
var firstname = 'John';
```

```
(function(name) {  
  
    var greeting = 'Inside IIFE: Hello';  
    console.log(greeting + ' ' + name);  
  
})(firstname); // IIFE
```

Closures

Poznata je činjenica da kada se završi funkcija, sve njene lokalne promenljive pokupi garbage collector i one prestaju da postoje u memoriji [MDN](#). Međutim to nije slučaj za funkciju koja unutar sebe sadrži *closure* funkciju.

Closure je kombinacija funkcije i leksičkog okruženja u kojem je ta funkcija definisana. Closure su funkcije koje imaju pristup promenljivima koje se nalaze u domenu druge funkcije. Treba napomenuti da se promenjive spoljne funkcije ne brišu po izvršavanju same funkcije, već se čuvaju u memoriji da bi bile dostupne closure funkciji. Nakon izvršenja closure funkcije, zatvara se i spoljna funkcija (odatle i naziv “closure”, *zatvaranje*). Dok god se closure ne izvrše, JavaScript će čuvati i potrebne promenjive iz domena drugih funkcija, stoga one zauzimaju više memorije nego obične funkcije. [Webprogramiranje](#)

Primer:

```
function greet(whattosay) {  
    return function(name){  
        console.log(whattosay+ ' ' + name)  
    }  
}  
greet('Hi')('Tamara');
```

Rezultat



Ako malo izmenimo kod šta će se desiti

```
function greet(whattosay) {  
    return function(name){  
        console.log(whattosay+ ' ' + name)  
    }  
}
```

```
var sayHi = greet('Hi')  
sayHi('Tamara');
```

Rezultat



Hajde da vidimo šta se u pozadini dešava



```
function greet(whattosay) {  
  return function(name){  
    console.log(whattosay+ ' ' + name)  
  }  
}
```



```
}  
  
var sayHi = greet('Hi')  
sayHi('Tamara');
```

Pozivanjem `var sayHi = greet('Hi')` funkcija `sayHi` je kreirala svoj kontekst izvršavanja u okviru scope chain-a, odnosno za funkciju `greet()` i kada se izvršila ona se sa steka briše kao i kontekst izvršavanja. Kako je moguće da `sayHi` u svom drugom pozivu imam referencu na to šta je to `whattosay` promenljiva kada se njeno izvršavanje završilo? Mogli smo da imamo milione linija koda između koji se ne odnose na ove dve funkcije i ono bi i dalje moglo da se izvrši.

Šta se onda desi kada se izvrši `sayHi('Tamara')`? Kreira se kontekst izvršavanja za samu funkciju i ono se kao anonimna funkcija vezuje sada za globalni kontekst izvršavanja. Iako je `greet` prošao i završio se, ostao je pokazivač na njegovu memorijsku lokaciju, i tako JS engine obezbedio funkciji `sayHi` da može da ide niz scope chain i traži potrebne informacije. Kažemo da je kontekst izvršavanja zatvorio sve njegove spoljne varijable, one na kojem bi u svakom slučaju imao reference, da se izvršavanje nije podelilo u dve funkcije.

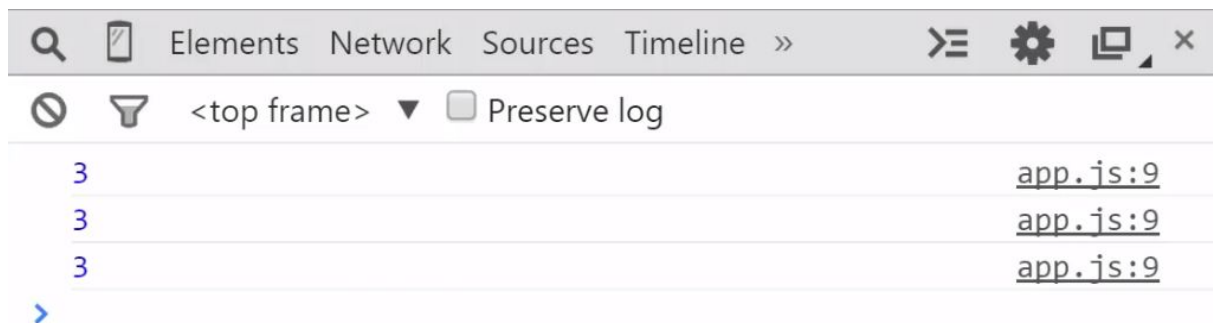


Hajde da pogledamo jedan primer. Najpoznatiji there is što se closure-a tiče:

```
function buildFunctions(){
  var arr = [];
  for(var i =0;i<3; i++){
    arr.push(
      //ovo ne predstavlja pozivanje fje vec samo njeno kreiranje
      function(){
        console.log(i)
      }
    )
  }
  return arr;
}
```

```
var fs = buildFunctions();
//ovde pozivamo funkciju
fs[0]();
fs[1]();
fs[2]();
```

Rezultat:



Očekivali smo da će rezultat biti 0,1,2.

Znači kada se izvršila `buildFunctions()` ono je kreirala dve stvari nakon svog izvršena `i=3` i `arr[f0,f1,f2]`. Zašto `i = 3`, pa for petlja ovog tipa na kraju svakog prolaza povećava vrednost `i`, onda pita da li je ta vrednost manja od 3, u poslednjem krugu, ona će `i` povećati na tri, ali neće ući u sam kod petlje. Što znači da je u memoriji ostalo zapamćeno da je `i` 3. Onog momenta kada funkcije u nizu krenu da se izvršavaju (naše poslednje tri linije koda), tek tog momenta

one traže da dodele i neku vrednost i odštampaju je. A šta su našle? Ono što je petlja poslednje upisala, jer se u vrednost i ništa nije upisalo pri momentu kreiranja i dodavanja funkcija samom nizu. Kada se `buildFunctions()` iza nje je ostao pokazivač na njenu memoriju i ono što je ona sadržala u sebi.

Ali šta ako smo hteli da naš kod zapravo radi - ispiše 0,1,2? Šta mislite kako bismo to postigli?

```
function buildFunctions(){
  var arr = [];
  for(var i =0; i<3; i++){
    let j=i
    arr.push(
      function(){
        console.log(j)
      }
    )
  }
  return arr;
}
```

```
var fs = buildFunctions();
fs[0]();
fs[1]();
fs[2]();
```

Jedno let nam rešava problem.

Ili drugi način je preko IIFE.

```
function buildFunctions(){
  var arr = [];
  for(var i =0; i<3; i++){
    arr.push(
      (function(j){
        return function(){
          console.log(j)
        }
      })(i))
    )
  }
  return arr;
}
```

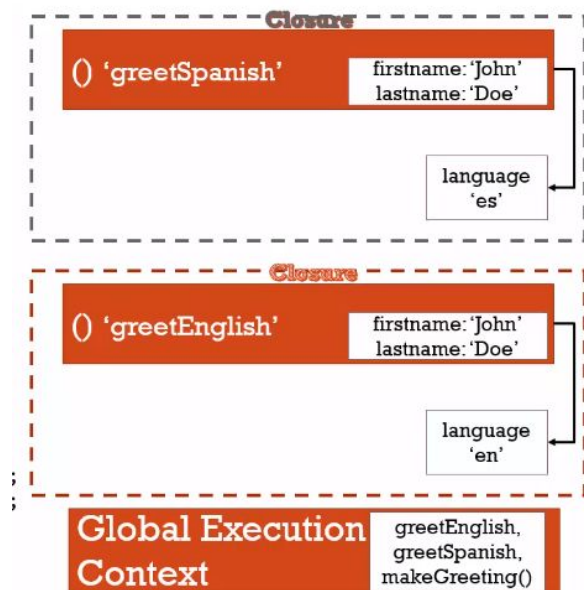
```
var fs = buildFunctions();
fs[0]();
fs[1]();
fs[2]();
```

Malo zbunjujuće izgleda, ali ništa posebno. Na naš niz push-ujemo funkciju koja će se izvršiti i vratiti nam funkciju koja se tog momenta kreira a ne izvršava, kao i u prethodnom primeru, samo što sad sadrži referencu na i u trenutku izvršavanja IIFE.

Još jedan primer koji se često koristi u frejmvcima:

```
function makeGreeting(language) {
  return function(firstname,
    lastname) {
    if (language === 'en') {
      console.log('Hello ' +
        firstname + ' ' + lastname);
    }
    if (language === 'es') {
      console.log('Hola ' +
        firstname + ' ' + lastname);
    }
  }
}

var greetEnglish = makeGreeting('en');
var greetSpanish = makeGreeting('es');
greetEnglish('John', 'Doe');
greetSpanish('John', 'Doe');
```



Closure i Callback

[Callback](#) je mehanizam, poznat i u drugim jezicima, koji omogućava da se funkcija prosledi kao parametar, da bi kasnije bila pozvana po potrebi. Ova praksa ima korene u funkcionalnom programiranju gde je prosleđivanje funkcija kao parametara sasvim uobičajena stvar. To je deo srži samog funkcionalnog programiranja

CALLBACK FUNCTION: A FUNCTION YOU GIVE TO ANOTHER FUNCTION, TO BE RUN WHEN THE OTHER FUNCTION IS FINISHED

So the function you call (i.e. invoke), 'calls back' by calling the function you gave it when it finishes.

Primer:

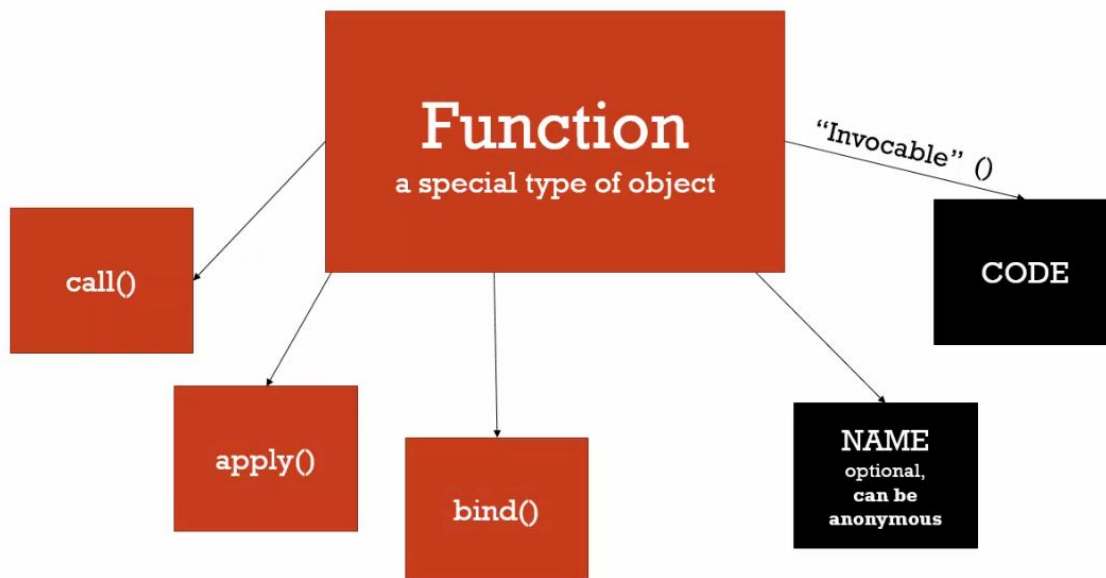
```
function sayHiLater(){  
    var greetings = 'Hi';  
  
    //ovo je ugrađena funkcija za timeout  
  
    // koja prima 2 parametra  
  
    // ono što treba da pauzira i koliko dugo u milisekundama  
  
    setTimeout(function(){  
        console.log(greetings);  
    }, 3000)  
}  
  
sayHiLater();
```

Rezultat:

Hi	VM50:7
>	

Posle tri sekunde ispisao je Hi. Šta nam to govori, da je ova izrazna funkcija prosleđena kao parametar zapamtla referencu na samu sayHiLater funkciju i ono što se u njoj izvršilo pre tri sekunde.

Call() Apply() Bind()



Sve funkcije u JSu pored svog koda i imena metode i propertije koji su im dodeljeni pri kreiranju i jedni od tih su: call, apply i bind metode. Sve tri metode se vezuju za **this**.

[Bind\(\)](#)

```
bind(thisArg: any, ...argArray: any[]): any
```

An object to which the this keyword can refer inside the new function.

For a given function, creates a bound function that has the same body as the original function.

The this object of the bound function is associated with the specified object, and has the specified initial parameters.

```
15    }.bind()
```

Preko **bind()** funkcije, *this* menjamo sa prosleđenim objektom.

```
var person= {  
  firstname: 'Tamara',  
  lastname: 'Naumovic',  
  age: 24,  
  getFullName: function(){
```

```

        var fullname = this.firstname+' '+this.lastname;

        return fullname;
    }
}

// u nasem slucaju this ce pokazivati na sam objekat person
// jer se funkcija u kojoj se poziva kreira i izvrsava u okviru samog objekta

var logname = function(lang1, lang2){

    console.log('Logged: '+ this.getFullName());

} // ovde ce da fejljuje, jer ovo this sad pokazuje na globalni objekat
// koji ne poseduje getFullName property

logname(); //izbaci ce gresku

```

```

//bind vraca novu funkciju, odnosno kopiju Logname fje
//kad god se pokrene JS engine vidi bind metodu
//i this setuje na vrednost paramentra bind metoda

var logPersonName = logname.bind(person)

logPersonName() //ovo ce nam dati pravi rezultat

```

Mogli smo i pri kreiranju same funkcije da pozovemo bind te funkcije

```

var logname = function(lang1, lang2){
    console.log('Logged: '+ this.getFullName());
}.bind(person);

```

Bind ne poziva i izvršava fju, već kreira njenu kopiju i dodeljuje proprijetu *this* vrednost parametra prosleđenog.

[Call\(\)](#)

(thisArg: any, ...argArray: any[]): any

The object to be used as the current object.

Calls a method of an object, substituting another object for the current object.

```
26 logname.call()  
  
var logname = function(lang1, lang2){  
    console.log('Logged: ' + this.getFullName());  
}  
logname.call(person, 'en', 'es');
```

Razlika je u tome što `call` ne kreira kopiju, kao što je to radio `bind` i dodeljivao je nekoj promenljivoj. `Call` pri samom pozivu izvršava u tom trenutku funkciju dodeljivajući joj parametar na koji će *this* da se odnosi, i pored njega prosleđuje parametre funkcije koja se izvršava, odvajajući ih zarezom.

[Apply\(\)](#)

(thisArg: any, argArray?: any): any

The object to be used as the this object.

Calls the function, substituting the specified object for the *this* value of the function, and the specified array for the arguments of the function.

```
27 logname.apply()
```

Razlika između `call` i `apply` je ta što lista argumenata ne može da se prosledi samo navodeći ih jedan za drugim iza zareza, već je potrebno listu argumenata navesti kao niz.

```
logname.apply(person, ['es', 'en'])
```

Funkcionalno programiranje

Hajde da vidimo na delu lepotu funkcionalnog programiranja

```
var arr1 = [1,2,3];
console.log(arr1);
var arr2 = [];
for (var i=0; i < arr1.length; i++) {
    arr2 .push(arr1[i]*2);
}
console.log(arr2);
```

Rezultat:

▶ (3) [1, 2, 3]	VM900:2
▶ (3) [2, 4, 6]	VM900:7

Napisali smo dosta veliki kod kako bismo napravili da ovo radi. Međutim znajući koliko smo lenji kao programeri, hoću da napišemo ovo malo lepše. Sa prvoklasnim funkcijama ćemo postići upravo to. Napravićemo jednu funkciju za mapiranje:

```
function mapForEach(arr, fn) {
    var newArr = [];
    for (var i=0; i < arr.length; i++) {
        newArr.push(
            fn(arr[i])
        )
    };
    return newArr;
}
var arr2 = mapForEach(arr1, function(item) {
    return item * 2;
});
console.log(arr2);
```

Onu gužvu oko for petlje smo gurnuli u funkciju. Što nam dozvoljava da se sa ovom funkcijom igramo a različite načine.

```
var arr3 = mapForEach(arr1, function(item) {
    return item > 2;
});
console.log(arr3);
Hajde da vidimo još neki primer:
var checkPastLimit = function(limiter, item) {
    return item > limiter;
}
```

```
var arr4 = mapForEach(arr1, checkPastLimit.bind(this, 1));
```

Ovime smo difoltno podesili parametar limiter na 1

```
console.log(arr4);
```

Uprošćena verzija, gde ne moramo da prosleđujemo this u bind-u:

```
var checkPastLimitSimplified = function(limiter) {  
    return function(limiter, item) {  
        return item > limiter;  
    }.bind(this, limiter);  
};
```

```
var arr5 = mapForEach(arr1, checkPastLimitSimplified(1));  
console.log(arr5);
```

Ali opet imamo bind???!!!! Da imamo, ali zamislite da ste uzeli da probate novu biblioteku JSa i da u toj biblioteci metode nisu definisane kao `checkPastLimitSimplified` već od tebe zahtevaju da nekad negde setuješ **this** na neki objekat negde prosleđen, ono daj makse. Zato tamo negde u definisanju metoda i njihovom kreiranju treba sebi/drugima da maksimalno olakšamo korišćenje. Jedno pojašnjenje samo

```
var checkPastLimitSimplified = function(limiter) {  
    return function(limiter, item) {  
        return item > limiter;  
    }.bind(this, limiter);  
};
```

Arrow funkcija

```
var checkPastLimitArrow = function(limiter) {  
    return (item)=> item>limiter  
}
```

Da pojasnim šta se dešava u gornjem primeru.

Kada smo nizu arr5 rekli da je jednak mapForEach funkciji - dali smo jo parametar niz i funkciju checkPastLimiterSimplified. Ali tu grešimo, nismo joj mi prosledili checkPastLimiterSimplified funkciju već njen rezultat - kako to znam? Pa drugi argument u zagradi je checkPastLimiterSimplified funkcija koja se izvršava sa parametrom 2 i vraća kao rezultat funkciju kojoj je 2 difoltno postavljen parametar i ONA se onda prosledjuje kao parametar funkcije mapForEach.

Sedi udahni, pročitaj opet, i opet i pokušaj da shvatiš šta smo ovde uradili.

Praksa je da menjanje promenljivih, atributa, funkcija, nikada se ne dešava u samom koru mesta gde ga menjamo, već što je moguće više u lancu ili da ne menjamo uopšte već da kreiramo kopije ili nove promenljive koje će biti rezultat promene nekog tipa.

Primeri funkcionalnog programiranja

[Underscore.js](#)*** ili [lodash](#)

Biblioteka koja olakšava rad sa nizovima, kolekcijama i objektima

```
// underscore
```

```
var arr6 = _.map(arr1, function(item) { return item * 3 });  
console.log(arr6);
```

```
var arr7 = _.filter([2,3,4,5,6,7], function(item) { return item % 2 === 0; });  
console.log(arr7);
```

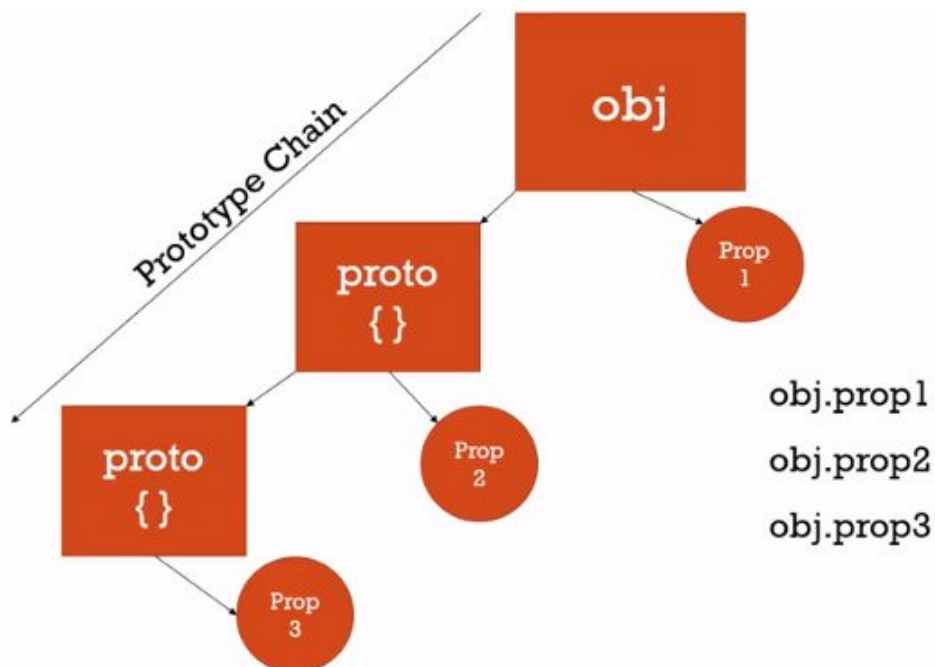
Objektno orijentisani JS i prototipno nasleđivanje

INHERITANCE:
ONE OBJECT GETS ACCESS TO
THE PROPERTIES AND METHODS
OF ANOTHER OBJECT.

Klasično nasleđivanje - Java i C# koriste ovaj način nasleđivanja

[Prototipno nasleđivanje](#) (prototypal inheritance)

Prototip je properti svakog objekta. Svi objekti poseduju PROTO kao svoj atribut.



obj objekat ne sadrži prop2, zato JS ide dublje u nasleđivanje i gleda da li proto ima taj traženi atribut. Nasleđe kao proto objektu je fiksno. Proto može imati svoj prototip i tako dalje i tako dalje i tako dalje. Posao JS engine-a jeste da ide niz sam Chain i nađe atribut koji tražimo, sve do momenta dokle on zaista ne postoji.

Hajde da vidimo par primera kako ovo funkcioniše:

```
var person = {  
  firstname: 'Default',  
  lastname: 'Default',  
  getFullName: function() {  
    return this.firstname + ' ' + this.lastname;  
  }  
}
```

Napravimo jedan objekat koji ima recimo te difoltne vrednosti. I onda napravimo još jedan:

```
var john = {  
  firstname: 'John',  
  lastname: 'Doe'  
}
```

Koji ima setovane vrednosti za ime i prezime. Nigde trenutno ne navodimo da su john objekat i person objekat isto.

// nikad nikad nikad NIKAD ne raditi ovo! Samo za demonstraciju

```
john.__proto__ = person;  
console.log(john.getFullName());  
console.log(john.firstname);
```

Imamo John Doe i John kao ispis. Zašto ako smo nasledili person koji ima setovan Default kao vrednost za ime i prezime. Prototype Chain upravo tome i služi, da prvo gleda u samom objektu pa onda dalje ako tu ne može da nađe.

Evo sledećeg primera:

```
var jane = {  
  firstname: 'Jane'  
}  
  
jane.__proto__ = person;
```

```
console.log(jane.getFullName());
```

Ovd imamo Jane Default, jer nasleđuje od person objekta i nema definisano prezime.

Sve je objekat

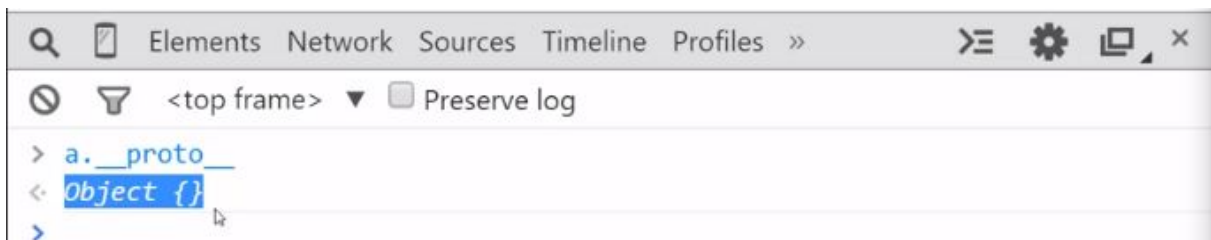
Svaki objekat, funkcija, promenljiva... sve u JS ima prototip osim baznog elementa

```
var a = {} //objekat
```

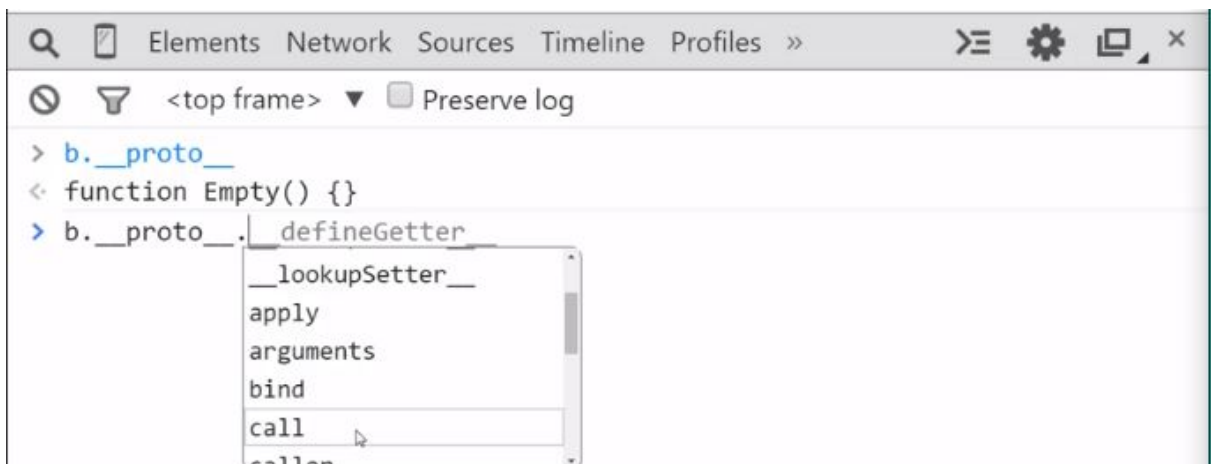
```
var b = function(){}; //funkcija
```

```
var c = [] //niz
```

Kada ranujemo ovo ne postoji nikakva greška, ali u konzoli imamo pristup svim elementima kreiranim u skripti.

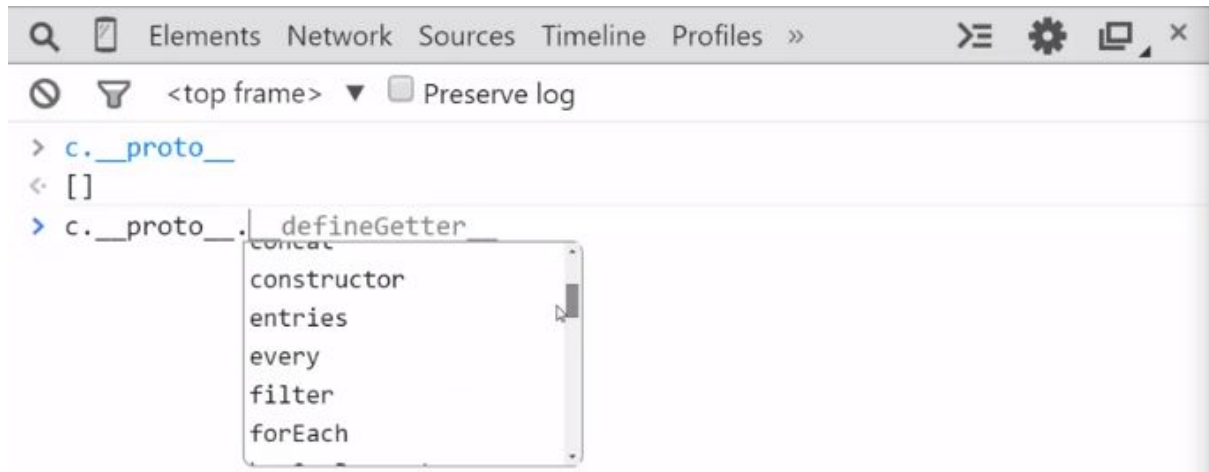


Object{} je bazični objekat i sve na kraju se sliva do njega. Sa svojim funkcijama i atributima.

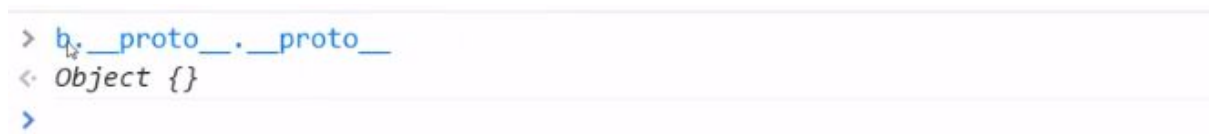


Sve funkcije imaju bazični prototip funkcije kao prototip i to nam zapravo dozvoljava da imamo pristup onom `apply`, `bind`, `call`.

Za niz važi ista stvar



Prototip svakog prototipa je Object{} zato kažemo da je sve u JSu objekat.



Refleksija i extend

REFLECTION: AN OBJECT CAN LOOK AT ITSELF, LISTING AND CHANGING ITS PROPERTIES AND METHODS.

JS objekat ima mogućnost da gleda samog sebe i svoje metode i atribut. Što nam dozvoljava da koristimo jedan feature koji se zove **extend** - produživanje.

Pogledajmo naš prethodni primer i dodajmo jednu petlju

```
var person = {  
  firstname: 'Default',  
  lastname: 'Default',  
  getFullName: function() {  
    return this.firstname + ' ' + this.lastname;  
  }  
}
```

```
var john = {
```

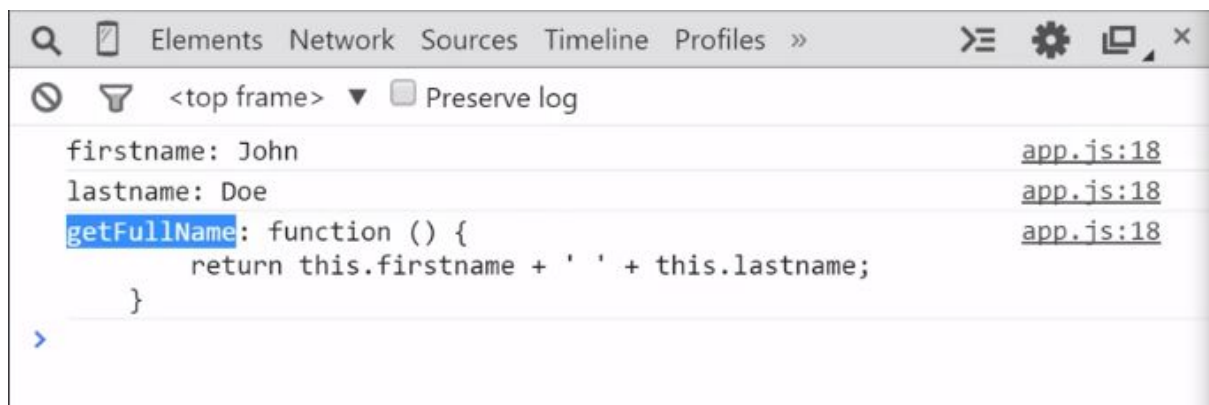
```

    firstname: 'John',
    lastname: 'Doe'
}

john.__proto__ = person;

for (var prop in john){
    console.log(prop + ': ' + john[prop]);
}

```

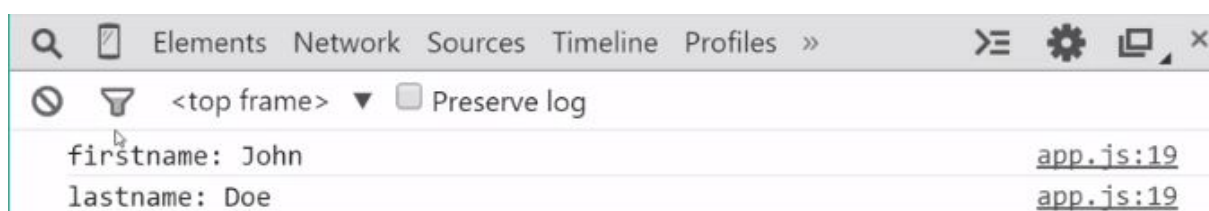


Ako malo izmenimo naš kod

```

for (var prop in john){
    if (john.hasOwnProperty(prop)){
        console.log(prop + ': ' + john[prop]);
    }
}

```



Hajde da vidimo kako extend radi. Koristićemo underscore biblioteku. Dodajmo našem kodu još dva objekta jane i jim

```

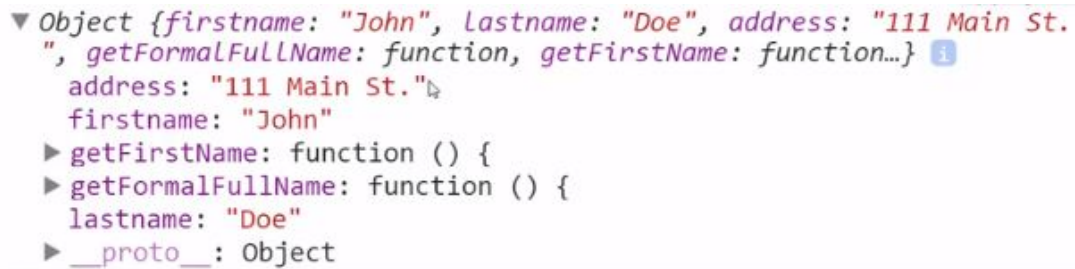
var jane = {
    address: '111 Main St.',
    getFormalFullName: function() {
        return this.lastname + ', ' + this.firstname;
    }
}

```

```
var jim = {
  getFirstName: function() {
    return firstname;
  }
}
```

```
_.extend(john, jane, jim);
```

```
console.log(john);
```



```
▼ Object {firstname: "John", lastname: "Doe", address: "111 Main St.", getFormalFullName: function, getFirstName: function...} ⓘ
  address: "111 Main St."
  firstname: "John"
  ▶ getFirstName: function () {
  ▶ getFormalFullName: function () {
    lastname: "Doe"
  ▶ __proto__: Object
```

Kako ovo radi? Hajde da pogledamo u samoj biblioteci

```
// Extend a given object with all the properties in passed-in object(s).
```

```
_.extend = createAssigner(_.allKeys);
```

```
// An internal function for creating assigner functions.
```

```
var createAssigner = function(keysFunc, undefinedOnly) {
```

```
  return function(obj) {
```

```
    var length = arguments.length;
```

```
    if (length < 2 || obj == null) return obj;
```

```
    for (var index = 1; index < length; index++) {
```

```
      var source = arguments[index],
```

```
          keys = keysFunc(source),
```

```
          l = keys.length;
```

```
    for (var i = 0; i < l; i++) {
```

```
      var key = keys[i];
```

```
      if (!undefinedOnly || obj[key] === void 0) obj[key] = source[key];
```

```
    }
```

```
  }
```

```
  return obj;
```

```
};
```

```
};
```

U 4 liniji koda `createAssigner` funkcije vidimo da ispituje da li je niz manji od 2 i ako jeste prosto mi vrati moj objekat samo. Što je i logično ako, `extend` funkcije prosledimo samo naš objekat nema šta dalje da proširuje. Ako je broj argumenata veći od jedan onda petljom prođi kroz te argumente, ali počni od prvog sledećeg, ne od nultog elementa, jer već znamo šta naš objekat ima hajde da ga proširimo sa novim atributima. *Source* je trenutni element niza, a *keys* predstavljaju ključevi iz key-value parova samog objekta kojem pristupa. I onda hoću sledećom petljom da prođem kroz sve ključeve, pitam da li zaista postoje i ako postoje hoću da `obj[key] = source[key]` - sa uglastim zagradama postavljam vrednost novog *key* atributa na vrednost njegovog izvora. Ali mi nemamo taj atribut??? Pa rekli smo ranije, da je JS dinamički kucan jezik, što znači da uglastim zagradama setujemo vrednost postojećem atributu ili prilikom kucanja ga kreiramo i zatim postavljamo vrednost. Na kraju hoćemo da vratimo sam objekat.

Od ES6 verzije JSa `extend` je uveden kao opcija za nasleđivanje, kao i u drugim objektno orijentisanim jezicima. [EXTENDS](#)

Pravljenje objekata

Možemo kreirati objekte preko literala i to je ono što smo do sada radili, međutim to nije jedini način za kreiranje objekata, pogotovu ako nam je potrebno nasleđivanje.

[Ovde](#) možeš pogledati sve načine kreiranja objekata.

[OVDE](#) je objašnjeno kako radi nasleđivanje sa različitim načinima kreiranja objekata

Konstruktori funkcije, 'new', i istorija JSa

```
function Person() {  
  this.firstname = 'John';  
  this.lastname = 'Doe.';  
}
```

```
var john = new Person();  
console.log(john);
```



new je uveden kako bi programeri koji su prešli sa Java na JS navikli i lakše uklopili u jezik, iako JS po svom tipu nije objektno orijentisan jezik.

Sa new se kreira novi prazan objekat i poziva funkciju Person(). Dokle god funkcije kojima želimo kreirati nove objekte nemaju nikakvu povratnu vrednost (return) mogu se dodeliti kao *tip objekta* nekoj varijabli.

Hajde da pogledamo šta se dešava ovde

```
function Person() {  
  console.log();  
  this.firstname = 'John';  
  this.lastname = 'Doe.';  
  console.log('This function is invoked.');
```

return {greeting:"getting in the way"};

```
}
```



Nismo dobili naš objekat jer funkcija Person vraća objekat.

Nešto još nisam napomenula, kada smo kreirali funkciju, Person sam napisala sa velikim P - imajte na umu, kad god kreirate funkciju koja je novi tip objekta, njenu deklaraciju pišemo **velikim početnim slovom!!!**

```
function Person(firstname, lastname) {  
    console.log(this);  
    this.firstname = firstname;  
    this.lastname = lastname;  
    console.log('This function is invoked.');
```

```
}
```

```
var john = new Person('John', 'Doe');  
console.log(john);
```

```
var jane = new Person('Jane', 'Doe');  
console.log(jane);
```

Ova funkcija je konstruktor funkcija! - koriteći funkciju kreiramo objekat.

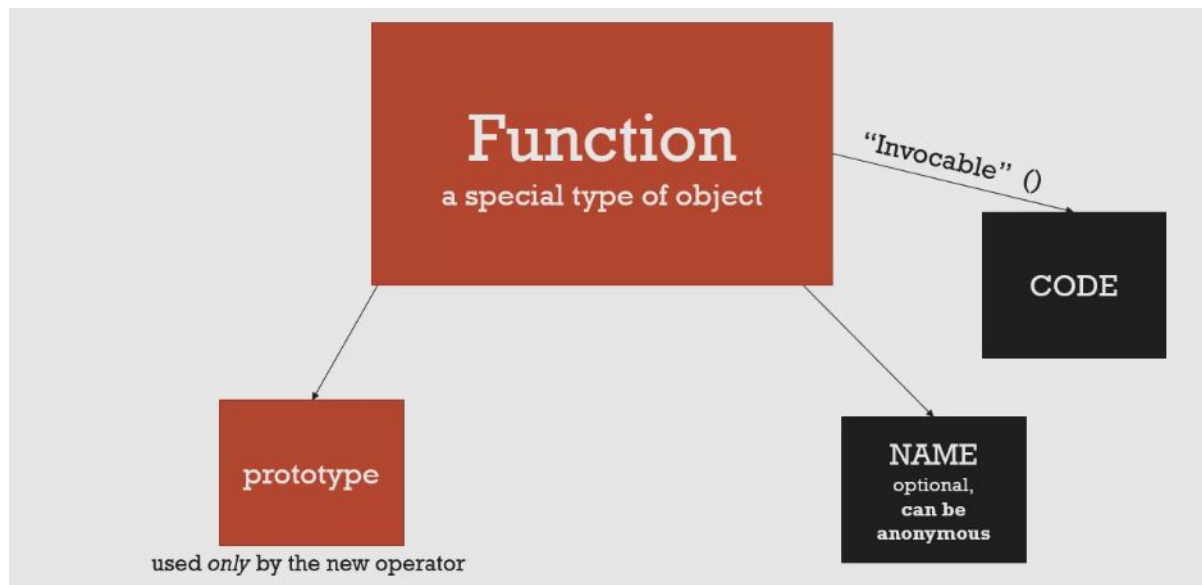
FUNCTION CONSTRUCTORS: A NORMAL FUNCTION THAT IS USED TO CONSTRUCT OBJECTS.

The 'this' variable points a new empty object, and that object is returned from the function automatically.

Konstruktori funkcije i .prototype

Kad god kreiramo objekat preko funkcije, rekli smo da ima svoje atribute, ako što su ime i kod, ali ima još jedan koji vidljiv svima i prilično koristan.

Ovaj atribut inače ne služi ničemu ako našu funkciju ne koristimo kao konstruktor za objekte, ali prvog trenutka kada pozovemo new, ovaj atribut nam je dostupan. On postoji i kod bilo koje druge funkcije, ali ne koristi se, ne služi svojoj svrsi.



Ovo nije prototip funkcije.

Treba da razlikujemo **__proto__** od **.prototype**.

.prototype je prototip objekta kreiranog putem funkcije, ne njen prototip.

```
function Person(firstname, lastname) {  
    console.log(this);  
    this.firstname = firstname;  
    this.lastname = lastname;  
    console.log('This function is invoked.');
```



```
}  
  
Person.prototype.getFullName = function() {  
    return this.firstname + ' ' + this.lastname;  
}  
  
var john = new Person('John', 'Doe');  
console.log(john);  
  
var jane = new Person('Jane', 'Doe');  
console.log(jane);
```

```
Person.prototype.getFormalFullName = function() {
    return this.lastname + ', ' + this.firstname;
}
```

```
console.log(john.getFormalFullName());
```

Mi *u hodu* dodajemo atribute našem objektu, ali to ne menja objekat i njegovu definiciju kao takvu. Bacimo pogled na ovo:

```
> a = new Person()
▼ Person {}
  firstname: undefined
  lastname: undefined
  ▶ __proto__: Object
This function is invoked.
< ▶ Person {firstname: undefined, lastname: undefined}
> a.getFullName()
< "undefined undefined"
>
```

Kada kreiramo novi prazan objekat, on kao svoje atribute ima samo *firstname* i *lastname* nigde ne piše da možemo da pristupimo metodama za *fullName* i *formalFullName*. Ali ako pokušamo da pozovemo tu metodu, nama je to dozvoljeno. U samoj memoriji, zapisano je da su ova dva svojstva dodata Person prototipu, ali u njegovoj definiciji, u njegovom kodu ne postoje kao takvi. Malo uvrnuto ali go with a flow.

Često ćemo videti, u dobrom JS kodu, da su atributi nekih objekata kreirani i definisani u okviru samog koda objekta, ali da su metode kreirane kasnije i zalepljene na sam prototip.

```
> Person.prototype
< ▼ {getFullName: f, getFormalFullName: f, constructor: f}
  ▶ getFormalFullName: f ()
  ▶ getFullName: f ()
  ▶ constructor: f Person(firstname, lastname)
  ▶ __proto__: Object
>
```

Zašto ne bismo dodali metode u sam objekat? Zauzimaju mnogo mesta, mnogo memorije, jer pri svakom kreiranju novog objekta, ova metoda bi se uvek kreirala i zadržavala mesto, u ovom slučaju, sa `.prototype`, ona postoji na prototipu, samo na jednom mestu.

NAPOMENA

```
var john = Person('John', 'Doe');
console.log(john);
```

```
var jane = Person('Jane', 'Doe');
console.log(jane);
```

Ono što mi možemo da uradimo, jer konstruktor funkcija je funkcija kao i svaka druga - možemo da kažemo da je neka promenljiva jednaka tvojoj funkciji, i to ne samo jednaka, već jednaka njenom rezultatu (s obzirom da imamo zagrade, to znači da se funkcija izvršava i da ćemo dobiti njen rezultat). S obzirom na to da naša funkcija nema nikakav rezultat, nema neki `return`, naše promenljive će biti jednake *undefined*. Ako koristimo funkcije kao konstruktore, **MORAMO ALI OBAVEZNO MORAMO DA KORISTIMO NEW!**

Ugrađeni konstruktori

Ovi konstruktori se odnose na objekte koje smo pominjali na početku: ono `Number`, `Boolean`, `String`... pokazivali smo primere sa prinudnom konverzijom tipa, koristeći ove funkcije/objekte :D

Mi možemo da kreiramo broj pomoću `Number` funkcije, prosto i jednostavno sa `new`

```
var a = new Number(3)
```

Ništa specijalno. Jel da? Ali šta je a?

```
> a
< Number {3}
  __proto__: Number
    [[PrimitiveValue]]: 3
```

Nije prost tip, a je objekat tipa `Number` koji ima primitivnu vrednost 3. To nam daje da pristupimo nekim metodama koje `Number` kao prototip nudi.

```
> a.toFixed(2)
< "3.00"
```

`Number` kao tip to ne poseduje

```
> Number.toString
  prototype
  toString
  constructor
  toLocaleString  Object
  toString
  constructor
  isPrototypeOf
  __proto__
```

Ali protip, na koji pokazuje a, da

```
> Number.prototype.toExponential
  toExponential  Number
  toFixed
  toLocaleString
  toPrecision
  toString
  constructor
  toLocaleString  Object
  toString
  constructor
```

Ovo može da bude nekad korisno, iako ne treba da kreiramo proste tipove pomoću konstruktora, hajde da pogledamo kako ovo može da nam pomogne da unapredimo naš kod.

Hoćemo da dodamo novi feature za sve `String` tipove ili za sve `Number` tipove

```
String.prototype.isLengthGreaterThan = function(limit) {
  return this.length > limit;
}
```

```
console.log("John".isLengthGreaterThan(3));
```

```
Number.prototype.isPositive = function() {
  return this > 0;
}
```

```
> b = 5
< 5
> b.isPositive()
< true
> |
```

Iako su promenljive koje kreiramo prostog tipa, one se mogu konvertovati u objekat. Ali moramo voditi računa i kada ovo koristimo, jer na primer:

```
> "Tamara".isLengthGreaterThan(5)
< true
> 7.isPositive()
✖ Uncaught SyntaxError: Invalid or unexpected token VM537:1
> |
```

String konvertuje automatski, ali broj ne, sem ako broj kao vrednost ne dodelimo nekoj promenljivoj.

NAPOMENA

```
> var a = 3
< undefined
> var b = new Number(3)
< undefined
> a == b
< true
> a === b
< false
```

U globalu, i u praksi, ugrađeni konstruktori se ne koriste kao način za kreiranje primitivnih vrednosti.

Ali šta ako nam treba Date, koji je jedan od korisnijih ugrađenih konstruktora, a nije za kreiranje prostog tipa. Bolje da koristimo neku biblioteku, jer su tu već regulisani svih problemi do koji se lako može doći korišćenjem *goli*h ugrađenih konstruktora. Jedna super biblioteka je [moment.js](#).

Možemo koristiti nekad koristiti ove funkcije, kao što smo na početku, za prinudnu konverziju i slično, ali ne kao konstruktor. **Don't mess with it. Unless you are pretty sure what are you messing with.**

NAPOMENA

```
var arr = ['tam', 'sam', 'ram'];
for (var prop in arr){
    console.log(prop + ': ' + arr[prop])
}
```

Nizovi su isto malo zeznuti, s obzirom da su objekti tipa Array, i njihova iteracija može da ide sa **for..in** petljom

```
0: tam VM548:3
1: sam VM548:3
2: ram VM548:3
```

Ovo 0,1,2 nisu samo pozicije u nizu, to su nazivi ključevi, jer Array je skup key-value parova. Tako da ako mi preko prototipa dodamo novi feature na primer

```
Array.prototype.myCustomFeature = 'cool';  
var arr = ['tam', 'sam', 'ram'];  
for (var prop in arr){  
    console.log(prop + ': ' + arr[prop])  
}
```

I onda prođemo kroz petlju

0: tam	(unknown)
1: sam	(unknown)
2: ram	(unknown)
myCustomFeature: cool	(unknown)

Dobijamo i ovaj `myCustomFeature` kao deo niza, jer naša petlja prolazi kroz sve propertyje tog niza.

Da bi ovo izbegli najbolje da koristimo standardnu for petlju

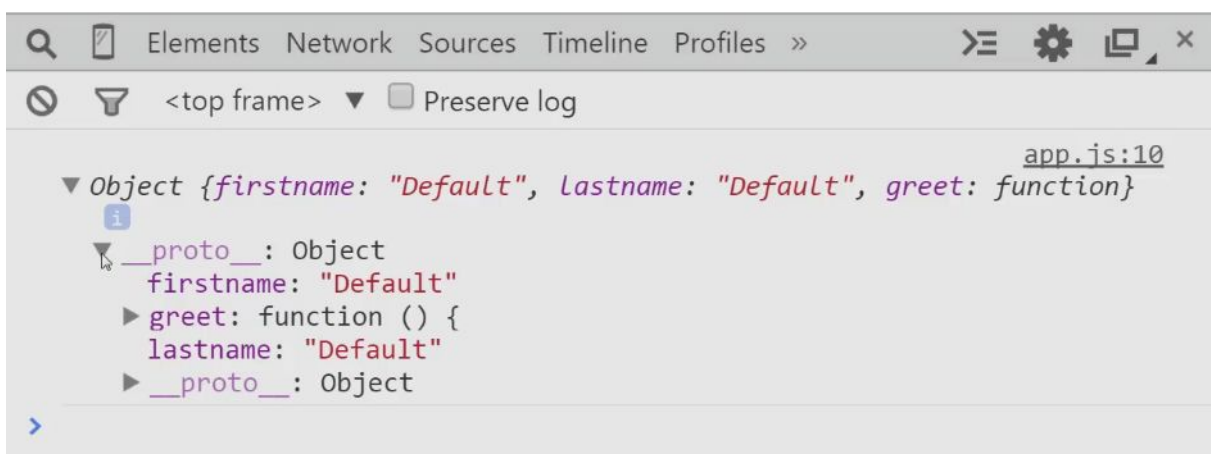
```
for(var i = 0; i<arr.length; i++){  
    console.log(i + ': ' + arr[i])  
}
```

Object.create i *pure* prototipno nasleđivanje

```
var person = {  
    firstname: 'Default',  
    lastname: 'Default',  
    greet: function() {  
        return 'Hi ' + this.firstname;  
    }  
}
```

Hajde da vidimo kako možemo kreirati objekat ovog tipa

```
var john = Object.create(person);  
console.log(john);
```



Object.create kreira prazan objekat koji referencira na svoj prototip. Kako možemo da dodamo naše vrednosti na naš objekat, pa jednostavno pozivanjem atributa na samom objektu

```
john.firstname = 'John';  
john.lastname = 'Doe';
```



Ovo se zove **pure prototypal inheritance**.

Ovo je nešto što svi današnji brauzeri podržavaju, ali ako je potrebno da, iz nekog nepoznatog razloga, ne koristimo Object.create jer brauzeri za koje kreiramo ne podržavaju našu funkciju, koristimo **Polifil**

POLYFILL:
CODE THAT ADDS A FEATURE
WHICH THE ENGINE *MAY* LACK.

```
// polyfill  
if (!Object.create) {  
  Object.create = function (obj) {  
    if (arguments.length > 1) {  
      throw new Error('Object.create implementation'  
        + ' only accepts the first parameter.');    }  
    function F() {}  
    F.prototype = obj;  
    return new F();  
  };  
}
```

Ako ovo postoji u brauzeru, samo će nastaviti sa ostatkom koda, ako ne postoji, kreira novu funkciju, koja vraća novi objekat, kao funkciju, sa prototipom obj.

Klase

Klasa kao način kreiranja objekta uveden je od ES6 verzije

```
class Person{
  constructor(firstname, lastname){
    this.firstname = firstname;
    this.lastname = lastname;
  }

  greet(){
    return 'Hi' + this.firstname;
  }
}

var tam = new Person("Tamara", "Naumovic");
```

Klasa u JS smislu nije isto što i klasa u bilo kom drugom drugom objektno-orijentisanom jeziku. U Javi ili C# klasa služi kao template po kojem se objekti kreiraju, definicija - u JSu **KLASA JE OBJEKAT**, nema tu razdvojenost, bez obzira na *class* koji služi da se njime kreira. To je samo još jedan od načina kreiranja objekta, koji tehnički pomaže onima koji u JS pristižu sa znanjem iz Jave i C#. Ovo su moderni updejti JSa, koji u nekom smislu zamagljuju koncept JSa - funkcionalno programiranje.

Ovde nasleđivanje radi sa onim **extends** koji sam ranije pominjala

```
class InformalPerson extends Person{
  constructor(firstname, lastname){
    super(firstname, lastname);
  }
  greet(){
    return 'Yo' + this.firstname;
  }
}
```

Ovo ne menja kako se ovi objekti ponašaju u pozadini, samo je još jedan od načina kako kreirati objekat.

EXTRAS

Vrlo često se u programiranju desi da misstype-ujemo nešto pre toga već definisano, ali JS kao takav dozvoljava sve što mi napišemo da se izvrši. Zbog toga su ljudi koji se bave JSom dodali jedan novi feature koji se zove [Strict mode](#), koji ne dozvoljava nikakva nedefinisana ponašanja.

Check it out.

Primeri

Greeter framework

Šta je naš frejmvrk i šta on radi

GREETR

When given a `firstname` and `lastname`, and optional `language`, it generates formal and informal greetings.

Support English and Spanish languages.

Reusable library/framework.

Easy to type `'G$()'` structure.

Koristili smo greet kroz ceo uvodni deo, pa hajde da napravimo biblioteku. Pored toga hoćemo da podržava jQuery.

Za početak hajde da definišemo strukturu našeg projekta:

index.html

app.js

Greetr.js

jquery.js

Index će kao i do sada sadržati sve naše js-ove

```
<html>

  <head>

  </head>

  <body>

    <script src="jquery-3.3.1.js"></script>

    <script src="Greetr.js"></script>

    <script src="app.js"></script>

  </body>

</html>
```

Greetr

```
(function(global, $) {
```

```
})(window, jQuery));
```

Rekli smo ranije da većina frejmvrkova i biblioteka su IIF, immediately invoked functions, iz razloga što nam je potrebno da su *ready to go* onog momenta kada se naša aplikacija učitava, bez nekog eksplicitnog pozivanja frejmvrka da se pokrene.

Dve varijable koje su nama potrebne su window i jQuery.

Kreiranjem naše biblioteke hoćemo da malo imitiramo jQuery i način na koji je on kreiran. Kada koristimo jQuery, dovoljno je da ga importujemo i bilo gde u kodu kada koristimo \$ naša aplikacija zna da koristimo jQuery, bez da smo pisali neko new i slično. E to hoćemo da postignemo sa našom bibliotekom.

Baby steps

Hajde da izmenimo malo **Greetr**

```
(function(global, $) {
```

```
    var Greetr = function(firstName, lastName, language){  
        return new Greetr.init(firstName, lastName, language);  
    }
```

```
    Greetr.prototype = {};
```

```
    Greetr.init = function(firstName, lastName, language){  
        var self = this;  
        self.firstName = firstName || "";  
        self.lastName = lastName || "";  
        self.language = language || "srb";
```



```
}
```

```
//imamo konstruktor koji kreira novi objekat sa prosleđenim vrednostima
```

```
//ili difoltnim ako nismo ništa uneli
```

```
Greetr.init.prototype = Greetr.prototype;
```

```
// prototip svakog objekta kreiranog sa Greetr.init hoću da bude Greeter.prototype
```

```
//baš kao i u jQuery-u
```

```
// sve metode vezane za Greetr, pisaću na njegovom prototipu
```

```
global.Greetr = global.G$ = Greetr;
```

```
//ovime smo prvo postavili da se Greetr poziva sa G$
```

```
// i zatim smo ga prikazali spoljašnosti, time što smo rekli
```

```
// bilo koji poziv na global.Greetr ili global.G$ vodi na samu funkciju
```

```
}(window, jQuery));
```

Kako ovo radi?

app.js

```
var g = G$('Tam', 'Naumovic');
```

```
console.log(g);
```

```
▼ Greetr.init 1 app.js:2
  firstName: "Tam"
  language: "srb"
  lastName: "Naumovic"
  ► __proto__: Object
> |
```

Atributi i metode

```
(function(global, $) {

    var Greetr = function(firstName, lastName, language){
        return new Greetr.init(firstName, lastName, language);
    }

    // dole kreiramo atribute, kojima Greeter može da pristupi
    // iako nisu u njegovom konstruktoru, pričali smo o closure-u, ali može da
    ih koristi
    // što je nama korisno jer neke metode i attribute ne želimo svima da
    prikazemo
    var supportedLangs = ['en', 'srb'];

    var greetings = {
        en: 'Hello',
        srb: 'Cao'
    };

    var formalGreetings = {
        en: 'Greetings',
        srb: 'Zdravo'
    };

    var logMessages = {
        en: 'Logged in',
        srb: 'Ulogovan/a'
    };

    //sve što je na prototipu je vidljivo korisnicima same biblioteke
    Greetr.prototype = {
        fullName: function(){

            return this.firstName + ' ' + this.lastName;
        },

        validate: function(){
            if (supportedLangs.indexOf(this.language)===-1){
                throw "Invalid language";
            }
        },

        greeting: function(){
            return greetings[this.language] + ' ' + this.firstName + '!';
        },

        formalGreeting: function(){
            return formalGreetings[this.language] + ' ' + this.fullName() +
            '!';
        },

        greet: function(formal){
```

```

    var msg;
    // ako je undefined ili null, koerzija ce ga prebaciti u false
    if (formal) {
        msg = this.formalGreeting();

    } else {
        msg = this.greeting();
    }
    if (console){
        console.log(msg)
    }

    // 'this' se odnosi na objekat koji pozivamo u vreme izvršavanja
    // na samo greet
    // time pravimo chainable metodu

    return this;
},

log: function(){
    if (console){
        console.log(logMessages[this.language] + ': ' + this.fullName());
    }

    return this;
},

setLang: function(lang){
    this.language = lang;
    this.validate();
    return this;
}

};

Greetr.init = function(firstName, lastName, language){
    var self = this;
    self.firstName = firstName || "";
    self.lastName = lastName || "";
    self.language = language || "srp";

}

Greetr.init.prototype = Greetr.prototype;

global.Greetr = global.G$ = Greetr;

})(window, jQuery));

```

app.js

```
var g = G$('Tam', 'Naumovic');  
g.greet();
```

```
Cao Tam! Greetr.js:59  
>
```

```
var g = G$('Tam', 'Naumovic');  
g.greet().setLang('en').greet(true);
```

```
Cao Tam! Greetr.js:59  
Greetings Tam Naumovic! Greetr.js:59  
>
```

Dodavanje jQuery-a

Hoćemo da prosledimo selektor nekog elementa, fizičkog elementa na našoj stranici. Ne želimo da gledamo u konzolu, hoćemo da se to zaista vidi negde. Koristićemo jQuery da olakšamo sebi baratanje sa elementima same stranice.

Prvo moramo malo da izmenimo naš HTML:

```
<html>  
  <head>  
  
  </head>  
  <body>  
    <div id="logindiv">  
      <select id="lang">  
        <option value="en">English</option>  
        <option value="srb">Srpski</option>  
      </select>  
      <input type="button" value="Login" id="login" />  
    </div>  
    <h1 id='greeting'></h1>  
    <script src="jquery-1.11.2.js"></script>  
    <script src="Greetr.js"></script>  
    <script src="app.js"></script>  
  </body>  
</html>
```

Dodajmo jednu metodu u naš prototip.

```
HTMLGreeting: function(selector, formal){  
  if(!$){  
    throw 'jQuery not loaded'
```

```

    }
    if(!selector){
        throw 'Missing jQuery selector'
    }

    var msg;

    if (formal) {
        msg = this.formalGreeting();

    } else {
        msg = this.greeting();
    }
    $(selector).html(msg);
    // ova linija koda poziva jQuery sa svojom metodom html nad elementom
    // sa prosleđenim selektorom
    // i menja njegov content
    return this;
}

```

Jedan dodatan trik, jer da stavimo ; iza poziva naše IIF. Zašto? To je da bismo zaštitili upotrebu našeg koda, nekad neke biblioteke neće raditi kako treba, nisu napisane baš kako treba i možda će ne njihov kod nastavljati na naš a to ne želimo, stoga odvajamo ono što je naše sa ;

```

app.js
var g = G$('Tam', 'Naumovic');
g.greet().setLang('en').greet(true);

$('#login').click(function(){
    var loginGrtr = G$('John', 'Doe');

    $('#logindiv').hide();

    loginGrtr.setLang($('#lang').val()).HTMLGreeting('#greeting', true).log();

}

```

Pig game

Index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <link href="https://fonts.googleapis.com/css?family=Lato:100,300,600"
rel="stylesheet" type="text/css">
    <link
href="http://code.ionicframework.com/ionicons/2.0.1/css/ionicons.min.css"
rel="stylesheet" type="text/css">
    <link type="text/css" rel="stylesheet" href="css/style.css">

    <title>Pig Game</title>
  </head>

  <body>
    <div class="wrapper clearfix">
      <div class="player-0-panel active">
        <div class="player-name" id="name-0">Player 1</div>
        <div class="player-score" id="score-0">43</div>
        <div class="player-current-box">
          <div class="player-current-label">Current</div>
          <div class="player-current-score" id="current-0">11</div>
        </div>
      </div>

      <div class="player-1-panel">
        <div class="player-name" id="name-1">Player 2</div>
        <div class="player-score" id="score-1">72</div>
        <div class="player-current-box">
          <div class="player-current-label">Current</div>
          <div class="player-current-score" id="current-1">0</div>
        </div>
      </div>
    </div>
  </body>
</html>
```

```

        <button class="btn-new"><i class="ion-ios-plus-outline"></i>New
game</button>
        <button class="btn-roll"><i class="ion-ios-loop"></i>Roll
dice</button>
        <button class="btn-hold"><i
class="ion-ios-download-outline"></i>Hold</button>

        <input type="text" placeholder="Final score" class="final-score">

        
        
    </div>

    <!--<script src="app.js"></script>-->
    <script src="js/app.js"></script>
</body>
</html>

```

Style.css

style.css

Type

Style Sheet

Size

3 KB (3,219 bytes)

Storage used

6 KB (6,435 bytes)

Location

css

Owner

me

Modified

21 Oct 2018 by me

Opened

21 Oct 2018 by me

Created

21 Oct 2018 with Google Drive Web

Add a description

Viewers can download

```
/******  
*** GENERAL  
******/
```

```
.final-score {  
    position: absolute;  
    left: 50%;  
    transform: translateX(-50%);  
    top: 520px;  
    color: #555;  
    font-size: 18px;  
    font-family: 'Lato';  
    text-align: center;  
    padding: 10px;  
    width: 160px;  
    text-transform: uppercase;  
}
```

```
.final-score:focus { outline: none; }
```

```
#dice-1 { top: 120px; }  
#dice-2 { top: 250px; }
```

```
* {  
    margin: 0;  
    padding: 0;  
    box-sizing: border-box;  
}
```

```
.clearfix::after {  
    content: "";  
    display: table;  
    clear: both;  
}
```

```
body {  
    background-image: linear-gradient(rgba(62, 20, 20, 0.4), rgba(62, 20, 20,  
0.4)), url(../img/back.jpg);
```



```
background-size: cover;
background-position: center;
font-family: Lato;
font-weight: 300;
position: relative;
height: 100vh;
color: #555;
}
```

```
.wrapper {
width: 1000px;
position: absolute;
top: 50%;
left: 50%;
transform: translate(-50%, -50%);
background-color: #fff;
box-shadow: 0px 10px 50px rgba(0, 0, 0, 0.3);
overflow: hidden;
}
```

```
.player-0-panel,
.player-1-panel {
width: 50%;
float: left;
height: 600px;
padding: 100px;
}
```

```
/******
*** PLAYERS
******/
```

```
.player-name {
font-size: 40px;
text-align: center;
text-transform: uppercase;
letter-spacing: 2px;
font-weight: 100;
```

```
margin-top: 20px;
margin-bottom: 10px;
position: relative;
}

.player-score {
  text-align: center;
  font-size: 80px;
  font-weight: 100;
  color: #EB4D4D;
  margin-bottom: 130px;
}

.active { background-color: #f7f7f7; }
.active .player-name { font-weight: 300; }

.active .player-name::after {
  content: "\2022";
  font-size: 47px;
  position: absolute;
  color: #EB4D4D;
  top: -7px;
  right: 10px;
}

.player-current-box {
  background-color: #EB4D4D;
  color: #fff;
  width: 40%;
  margin: 0 auto;
  padding: 12px;
  text-align: center;
}

.player-current-label {
  text-transform: uppercase;
  margin-bottom: 10px;
  font-size: 12px;
  color: #222;
}
```

```
}
```

```
.player-current-score {  
    font-size: 30px;  
}
```

```
button {  
    position: absolute;  
    width: 200px;  
    left: 50%;  
    transform: translateX(-50%);  
    color: #555;  
    background: none;  
    border: none;  
    font-family: Lato;  
    font-size: 20px;  
    text-transform: uppercase;  
    cursor: pointer;  
    font-weight: 300;  
    transition: background-color 0.3s, color 0.3s;  
}
```

```
button:hover { font-weight: 600; }  
button:hover i { margin-right: 20px; }
```

```
button:focus {  
    outline: none;  
}
```

```
i {  
    color: #EB4D4D;  
    display: inline-block;  
    margin-right: 15px;  
    font-size: 32px;  
    line-height: 1;  
    vertical-align: text-top;  
    margin-top: -4px;  
    transition: margin 0.3s;  
}
```

```

.btn-new { top: 45px;}
.btn-roll { top: 403px;}
.btn-hold { top: 467px;}

.dice {
  position: absolute;
  left: 50%;
  top: 178px;
  transform: translateX(-50%);
  height: 100px;
  box-shadow: 0px 10px 60px rgba(0, 0, 0, 0.10);
}

.winner { background-color: #f7f7f7; }
.winner .player-name { font-weight: 300; color: #EB4D4D; }

```

App.js

```

/*
Izazova
Izmeniti igru da ispoštuje naredna pravila:

1.Igrač gubi sav skor ako dobije 2 šestice
*/

var scores, roundScore, activePlayer, gamePlaying;

init();

document.querySelector('.btn-roll').addEventListener('click', function() {
  if(gamePlaying) {
    // 1. Random broj
    var dice1 = Math.floor(Math.random() * 6) + 1;
    var dice2 = Math.floor(Math.random() * 6) + 1;

    //2. Prikaz rezultata
    document.getElementById('dice-1').style.display = 'block';
    document.getElementById('dice-2').style.display = 'block';
  }
});

```

```

        document.getElementById('dice-1').src = '../img/dice-' + dice1 +
'.png';
        document.getElementById('dice-2').src = '../img/dice-' + dice2 +
'.png';

        //3. azuriraj skor ako kockica nije jednaka 1
        if (dice1 !== 1 && dice2 !== 1) {
            //dodaj skor
            roundScore += dice1 + dice2;
            document.querySelector('#current-' + activePlayer).textContent =
roundScore;
        } else {
            //Next player
            nextPlayer();
        }

        /* Ako ispunimo naš izazov prethodno if račvanje bi trebalo da izgleda
ovako
        if (dice1 === 6 && dice2 === 6) {
            //Igrac gubi skor
            scores[activePlayer] = 0;
            document.querySelector('#score-' + activePlayer).textContent = '0';
            nextPlayer();
        } else if (dice1 !== 1 && dice2 !== 1) {
            //dodaj skor
            roundScore += dice;
            document.querySelector('#current-' + activePlayer).textContent =
roundScore;
        } else {
            //sledeci igrac
            nextPlayer();
        }
        */
    }
});

```

```

document.querySelector('.btn-hold').addEventListener('click', function() {
    if (gamePlaying) {
        // dodaj trenutni skor na globalni

```

```

    scores[activePlayer] += roundScore;

    // ažuriraj prikaz stranice
    document.querySelector('#score-' + activePlayer).textContent =
scores[activePlayer];

    var input = document.querySelector('.final-score').value;
    var winningScore;

    // Undefined, 0, null ili "" se prevode u false
    // sve ostalo se prevodi u true
    //tako da će nam input vratiti ili true ili false
    if(input) {
        winningScore = input;
    } else {
        winningScore = 100;
    }

    // Da li je trenutni igrač pobedio
    if (scores[activePlayer] >= winningScore) {
        document.querySelector('#name-' + activePlayer).textContent =
'Winner!';
        document.getElementById('dice-1').style.display = 'none';
        document.getElementById('dice-2').style.display = 'none';
        document.querySelector('.player-' + activePlayer +
'-panel').classList.add('winner');
        document.querySelector('.player-' + activePlayer +
'-panel').classList.remove('active');
        gamePlaying = false;
    } else {
        //Ako nije idemo opet
        nextPlayer();
    }
}
});

```

```

function nextPlayer() {
    //Next player
    activePlayer === 0 ? activePlayer = 1 : activePlayer = 0;
}

```

```

roundScore = 0;

document.getElementById('current-0').textContent = '0';
document.getElementById('current-1').textContent = '0';

document.querySelector('.player-0-panel').classList.toggle('active');
document.querySelector('.player-1-panel').classList.toggle('active');

//document.querySelector('.player-0-panel').classList.remove('active');
//document.querySelector('.player-1-panel').classList.add('active');

document.getElementById('dice-1').style.display = 'none';
document.getElementById('dice-2').style.display = 'none';
}

document.querySelector('.btn-new').addEventListener('click', init);

function init() {
  scores = [0, 0];
  activePlayer = 0;
  roundScore = 0;
  gamePlaying = true;
  //getElementById je funkcija koja nam vraca neki tag(element) sa naseg
  html-a
  //ciji je id prosledjen u zagradi
  document.getElementById('dice-1').style.display = 'none';
  document.getElementById('dice-2').style.display = 'none';
  //postavljamo sve vrednosti na pocetne
  document.getElementById('score-0').textContent = '0';
  document.getElementById('score-1').textContent = '0';
  document.getElementById('current-0').textContent = '0';
  document.getElementById('current-1').textContent = '0';
  document.getElementById('name-0').textContent = 'Player 1';
  document.getElementById('name-1').textContent = 'Player 2';
  //brisemo klase active i winner, jer brisanje ne vraca gresku u slucaju da
  ne postoje
  //ne zelimo da rizikujemo da su mozda ostale prikacene na nase elemente
  document.querySelector('.player-0-panel').classList.remove('winner');
  document.querySelector('.player-1-panel').classList.remove('winner');
  document.querySelector('.player-0-panel').classList.remove('active');

```

```
document.querySelector('.player-1-panel').classList.remove('active');  
//postavljamo prvog igraca kao pocetnog igraca  
document.querySelector('.player-0-panel').classList.add('active');  
}
```

Budzet

Index.html

index.html

Type

HTML

Size

5 KB (5,547 bytes)

Storage used

5 KB (5,547 bytes)

Location

budzet

Owner

me

Modified

21 Oct 2018 by me

Opened

21 Oct 2018 by me

Created

21 Oct 2018 with Google Drive Web

Add a description

Viewers can download

```
<!DOCTYPE html>
```

```
<html lang="en">
```

```
  <head>
```

```
    <meta charset="UTF-8">
```

```
    <link
```

```
      href="https://fonts.googleapis.com/css?family=Open+Sans:100,300,400,600"
```

```
      rel="stylesheet" type="text/css">
```

```
    <link
```

```
      href="http://code.ionicframework.com/ionicons/2.0.1/css/ionicons.min.css"
```

```
      rel="stylesheet" type="text/css">
```

```
    <link type="text/css" rel="stylesheet" href="css/style.css">
```



```

        <input type="text" class="add__description"
placeholder="Add description">
        <input type="number" class="add__value"
placeholder="Value">
        <button class="add__btn"><i
class="ion-ios-checkmark-outline"></i></button>
    </div>
</div>

<div class="container clearfix">
    <div class="income">
        <h2 class="income__title">Income</h2>

        <div class="income__list">

            <!--
            <div class="item clearfix" id="income-0">
                <div class="item__description">Salary</div>
                <div class="right clearfix">
                    <div class="item__value">+ 2,100.00</div>
                    <div class="item__delete">
                        <button class="item__delete--btn">
                            <i class="ion-ios-close-outline"></i>
                        </button>
                    </div>
                </div>
            </div>

            <div class="item clearfix" id="income-1">
                <div class="item__description">Sold car</div>
                <div class="right clearfix">
                    <div class="item__value">+ 1,500.00</div>
                    <div class="item__delete">
                        <button class="item__delete--btn"><i
class="ion-ios-close-outline"></i></button>
                    </div>
                </div>
            </div>
        </div>
    </div>
-->

```

```
</div>
</div>
```

```
<div class="expenses">
  <h2 class="expenses__title">Expenses</h2>

  <div class="expenses__list">
```

```
    <!--
    <div class="item clearfix" id="expense-0">
      <div class="item__description">Apartment rent</div>
      <div class="right clearfix">
        <div class="item__value">- 900.00</div>
        <div class="item__percentage">21%</div>
        <div class="item__delete">
          <button class="item__delete--btn"><i
class="ion-ios-close-outline"></i></button>
        </div>
      </div>
    </div>

    <div class="item clearfix" id="expense-1">
      <div class="item__description">Grocery
shopping</div>

      <div class="right clearfix">
        <div class="item__value">- 435.28</div>
        <div class="item__percentage">10%</div>
        <div class="item__delete">
          <button class="item__delete--btn"><i
class="ion-ios-close-outline"></i></button>
        </div>
      </div>
    </div>
  -->
```

```
</div>
</div>
</div>
```

```
</div>

<script src="js/app.js"></script>
</body>
</html>
```

Style.css

style.css

Type

Style Sheet

Size

5 KB (5,051 bytes)

Storage used

10 KB (10,099 bytes)

Location

css

Owner

me

Modified

21 Oct 2018 by me

Opened

19:19 by me

Created

21 Oct 2018 with Google Drive Web

Add a description

Viewers can download

```
/*****
```

```
*** GENERAL
```

```
*****/
```

```
* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}
```

```
.clearfix::after {
```

```

        content: "";
        display: table;
        clear: both;
    }

    body {
        color: #555;
        font-family: Open Sans;
        font-size: 16px;
        position: relative;
        height: 100vh;
        font-weight: 400;
    }

    .right { float: right; }
    .red { color: #FF5049 !important; }
    .red-focus:focus { border: 1px solid #FF5049 !important; }

    /***** TOP PART *****/

    .top {
        height: 40vh;
        background-image: linear-gradient(rgba(0, 0, 0, 0.35), rgba(0, 0, 0,
0.35)), url(../img/back.png);
        background-size: cover;
        background-position: center;
        position: relative;
    }

    .budget {
        position: absolute;
        width: 350px;
        top: 50%;
        left: 50%;
        transform: translate(-50%, -50%);
        color: #fff;
    }

```

```
.budget__title {
    font-size: 18px;
    text-align: center;
    margin-bottom: 10px;
    font-weight: 300;
}

.budget__value {
    font-weight: 300;
    font-size: 46px;
    text-align: center;
    margin-bottom: 25px;
    letter-spacing: 2px;
}

.budget__income,
.budget__expenses {
    padding: 12px;
    text-transform: uppercase;
}

.budget__income {
    margin-bottom: 10px;
    background-color: #28B9B5;
}

.budget__expenses {
    background-color: #FF5049;
}

.budget__income--text,
.budget__expenses--text {
    float: left;
    font-size: 13px;
    color: #444;
    margin-top: 2px;
}

.budget__income--value,
```

```

.budget__expenses--value {
    letter-spacing: 1px;
    float: left;
}

.budget__income--percentage,
.budget__expenses--percentage {
    float: left;
    width: 34px;
    font-size: 11px;
    padding: 3px 0;
    margin-left: 10px;
}

.budget__expenses--percentage {
    background-color: rgba(255, 255, 255, 0.2);
    text-align: center;
    border-radius: 3px;
}

/***** BOTTOM PART *****/
/***** FORM *****/

.add {
    padding: 14px;
    border-bottom: 1px solid #e7e7e7;
    background-color: #f7f7f7;
}

.add__container {
    margin: 0 auto;
    text-align: center;
}

.add__type {
    width: 55px;
    border: 1px solid #e7e7e7;

```

```
    height: 44px;
    font-size: 18px;
    color: inherit;
    background-color: #fff;
    margin-right: 10px;
    font-weight: 300;
    transition: border 0.3s;
}
```

```
.add__description,
.add__value {
    border: 1px solid #e7e7e7;
    background-color: #fff;
    color: inherit;
    font-family: inherit;
    font-size: 14px;
    padding: 12px 15px;
    margin-right: 10px;
    border-radius: 5px;
    transition: border 0.3s;
}
```

```
.add__description { width: 400px;}
.add__value { width: 100px;}
```

```
.add__btn {
    font-size: 35px;
    background: none;
    border: none;
    color: #28B9B5;
    cursor: pointer;
    display: inline-block;
    vertical-align: middle;
    line-height: 1.1;
    margin-left: 10px;
}
```

```
.add__btn:active { transform: translateY(2px); }
```

```
.add__type:focus,
```



```
.add__description:focus,
.add__value:focus {
    outline: none;
    border: 1px solid #28B9B5;
}

.add__btn:focus { outline: none; }

/***** LISTS *****/
.container {
    width: 1000px;
    margin: 60px auto;
}

.income {
    float: left;
    width: 475px;
    margin-right: 50px;
}

.expenses {
    float: left;
    width: 475px;
}

h2 {
    text-transform: uppercase;
    font-size: 18px;
    font-weight: 400;
    margin-bottom: 15px;
}

.income__title { color: #28B9B5; }
.expenses__title { color: #FF5049; }

.item {
    padding: 13px;
    border-bottom: 1px solid #e7e7e7;
}
```

```
.item:first-child { border-top: 1px solid #e7e7e7; }  
.item:nth-child(even) { background-color: #f7f7f7; }
```

```
.item__description {  
    float: left;  
}
```

```
.item__value {  
    float: left;  
    transition: transform 0.3s;  
}
```

```
.item__percentage {  
    float: left;  
    margin-left: 20px;  
    transition: transform 0.3s;  
    font-size: 11px;  
    background-color: #FFDAD9;  
    padding: 3px;  
    border-radius: 3px;  
    width: 32px;  
    text-align: center;  
}
```

```
.income .item__value,  
.income .item__delete--btn {  
    color: #28B9B5;  
}
```

```
.expenses .item__value,  
.expenses .item__percentage,  
.expenses .item__delete--btn {  
    color: #FF5049;  
}
```

```
.item__delete {  
    float: left;  
}
```

```

.item__delete--btn {
  font-size: 22px;
  background: none;
  border: none;
  cursor: pointer;
  display: inline-block;
  vertical-align: middle;
  line-height: 1;
  display: none;
}

.item__delete--btn:focus { outline: none; }
.item__delete--btn:active { transform: translateY(2px); }

.item:hover .item__delete--btn { display: block; }
.item:hover .item__value { transform: translateX(-20px); }
.item:hover .item__percentage { transform: translateX(-20px); }

.unpaid {
  background-color: #FFDAD9 !important;
  cursor: pointer;
  color: #FF5049;
}

.unpaid .item__percentage { box-shadow: 0 2px 6px 0 rgba(0, 0, 0, 0.1); }
.unpaid:hover .item__description { font-weight: 900; }

```

App.js

```

/*Kreiranje modula za budzet, prikaz i aplikaciju
koriscenjem modularnog paterna
https://toddmotto.com/mastering-the-module-pattern/
https://dzone.com/articles/module-pattern-in-javascript
https://medium.com/@tkssharma/javascript-module-pattern-b4b5012ada9f
gde je svaki modul kreiran kao IIFE
*/

```

```

// BUDGET CONTROLLER
var budgetController = (function() {

    //kreiranje opisa objekta za Potrosnju bez metoda
    //ne zelimo da ih loadujemo pri svakom kreiranju
    var Expense = function(id, description, value) {
        this.id = id;
        this.description = description;
        this.value = value;
        this.percentage = -1;
    };

    //dodavanje metoda na prototip
    //preko prototipa moze se pristupiti bilo kojoj metodi
    Expense.prototype.calcPercentage = function(totalIncome) {
        if (totalIncome > 0) {
            this.percentage = Math.round((this.value / totalIncome) * 100);
        } else {
            this.percentage = -1;
        }
    };

    Expense.prototype.getPercentage = function() {
        return this.percentage;
    };

    //kreiranje opisa objekta za Primanja bez metoda

    var Income = function(id, description, value) {
        this.id = id;
        this.description = description;
        this.value = value;
    };

    var calculateTotal = function(type) {
        var sum = 0;
        data.allItems[type].forEach(function(cur) {
            sum += cur.value;
        });
    };
})();

```

```

    });
    data.totals[type] = sum;
};

```

//kreiramo objekat sa podacima u koji cemo upisivati svu potrosnju i sva primanja na jednom mestu

```

var data = {
  allItems: {
    exp: [],
    inc: []
  },
  totals: {
    exp: 0,
    inc: 0
  },
  budget: 0,
  percentage: -1
};

```

//closure za budget kontroler

```

return {
  addItem: function(type, des, val) {
    var newItem, ID;

    //[1 2 3 4 5], next ID = 6
    //[1 2 4 6 8], next ID = 9
    // ID = last ID + 1

    // Kreiramo novi ID
    if (data.allItems[type].length > 0) {
      ID = data.allItems[type][data.allItems[type].length - 1].id +
1;

    } else {
      ID = 0;
    }

    // Kreiramo novi item zavisno od tipa 'inc' ili 'exp'
    if (type === 'exp') {

```

```

        newItem = new Expense(ID, des, val);
    } else if (type === 'inc') {
        newItem = new Income(ID, des, val);
    }

```

```

// Pushujemo na data strukturu
data.allItems[type].push(newItem);

```

```

// i vracamo novokreirani element
return newItem;

```

```

},

```

```

deleteItem: function(type, id) {
    var ids, index;

```

```

    // id = 6
    //data.allItems[type][id];
    // ids = [1 2 4 8]
    //index = 3

```

```

    ids = data.allItems[type].map(function(current) {
        return current.id;
    });

```

//https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

//map funkcija vraca niz (slicno onome sto smo pravili u primerima)

//niz je mapiran odnosno kreira se u odnosu na prosledjenu funkciju

```

    //koja kao parametar prima trenutni clan niza
});

```

```

index = ids.indexOf(id);

```

```

if (index !== -1) {
    data.allItems[type].splice(index, 1);
    //metoda splice sece niz od zadatog indexa
    //do prosledjenog broja clanova niza
}

```

```
},
```

```
calculateBudget: function() {  
  
    // racunamo ukupne potrosnje i primanja  
    calculateTotal('exp');  
    calculateTotal('inc');  
  
    // racunamo budget: income - expenses  
    data.budget = data.totals.inc - data.totals.exp;  
  
    // racunamo procenat  
    if (data.totals.inc > 0) {  
        data.percentage = Math.round((data.totals.exp /  
data.totals.inc) * 100);  
    } else {  
        data.percentage = -1;  
    }  
  
},
```

```
calculatePercentages: function() {
```

```
    /*  
    a=20  
    b=10  
    c=40  
    income = 100  
    a=20/100=20%  
    b=10/100=10%  
    c=40/100=40%  
    */  
  
    data.allItems.exp.forEach(function(cur) {  
        cur.calcPercentage(data.totals.inc);  
    });  
}
```

[//https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach)

```

        //forEach funkcija prolazi kroz svaki clan elementa
        //i vrsi obradu tog elementa sa prosledjenom funkcijom
        //u nasem slucaju racuna pojedinačne procenete za svaki clan exp
niza
        //poziva metodu kreiranu na 15 liniji koda
    });
},

```

```

getPercentages: function() {
    var allPerc = data.allItems.exp.map(function(cur) {
        return cur.getPercentage();
    });
    return allPerc;
},

```

```

getBudget: function() {
    return {
        budget: data.budget,
        totalInc: data.totals.inc,
        totalExp: data.totals.exp,
        percentage: data.percentage
    };
},

```

```

testing: function() {
    console.log(data);
}
};

```

```

})();

```

```

// UI CONTROLLER

```

```

var UIController = (function() {

```

```

    //povlacimo sve klase definisane u indexu u jednu kolekciju stringova

```



```

var DOMstrings = {
  inputType: '.add__type',
  inputDescription: '.add__description',
  inputValue: '.add__value',
  inputBtn: '.add__btn',
  incomeContainer: '.income__list',
  expensesContainer: '.expenses__list',
  budgetLabel: '.budget__value',
  incomeLabel: '.budget__income--value',
  expensesLabel: '.budget__expenses--value',
  percentageLabel: '.budget__expenses--percentage',
  container: '.container',
  expensesPercLabel: '.item__percentage',
  dateLabel: '.budget__title--month'
};

```

```

var formatNumber = function(num, type) {
  var numSplit, int, dec, type;
  /*
    + ili - ispred broj
    zaokruzivanje na 2 decimale
    zarez (,) za odvajanje hiljada

    2310.4567 -> + 2,310.46
    2000 -> + 2,000.00
    */

  num = Math.abs(num);
  num = num.toFixed(2);

  numSplit = num.split('.');

  int = numSplit[0];
  if (int.length > 3) {
    int = int.substr(0, int.length - 3) + ',' + int.substr(int.length -
3, 3); //input 23510, output 23,510
  }

  dec = numSplit[1];

```

```

        return (type === 'exp' ? '-' : '+') + ' ' + int + '.' + dec;
    };

    //kreiranje kostura callback funkcije koju cemo koristiti kod prikaza
    var nodeListForEach = function(list, callback) {
        for (var i = 0; i < list.length; i++) {
            callback(list[i], i);
        }
    };

    //closure za UI kontroler
    return {
        getInput: function() {
            return {
                type: document.querySelector(DOMstrings.inputType).value, //
                //Will be either inc or exp
                description:
                document.querySelector(DOMstrings.inputDescription).value,
                value:
                parseFloat(document.querySelector(DOMstrings.inputValue).value)
            };
        },

        addListItem: function(obj, type) {
            var html, newHtml, element;
            // kreira HTML string sa placeholder textom

            if (type === 'inc') {
                element = DOMstrings.incomeContainer;

                html = '<div class="item clearfix" id="inc-%id%"> <div
                class="item__description">%description%</div><div class="right clearfix"><div
                class="item__value">%value%</div><div class="item__delete"><button
                class="item__delete--btn"><i
                class="ion-ios-close-outline"></i></button></div></div></div>';
                } else if (type === 'exp') {

```

```

        element = DOMstrings.expensesContainer;

        html = '<div class="item clearfix" id="exp-%id%"><div
class="item__description">%description%</div><div class="right clearfix"><div
class="item__value">%value%</div><div class="item__percentage">21%</div><div
class="item__delete"><button class="item__delete--btn"><i
class="ion-ios-close-outline"></i></button></div></div></div>';
    }

    // Zamenjuje placeholder text sa podacima iz prosledjenog objekta
    newHtml = html.replace('%id%', obj.id);
    newHtml = newHtml.replace('%description%', obj.description);
    newHtml = newHtml.replace('%value%', formatNumber(obj.value,
type));

    // ubacuje HTML u DOM
    document.querySelector(element).insertAdjacentHTML('beforeend',
newHtml);
    },

    deleteListItem: function(selectorID) {

        var el = document.getElementById(selectorID);
        el.parentNode.removeChild(el);

    },

    clearFields: function() {
        //resetovanje forme za unos
        var fields, fieldsArr;

        fields = document.querySelectorAll(DOMstrings.inputDescription + ',
' + DOMstrings.inputValue);

        fieldsArr = Array.prototype.slice.call(fields);

        fieldsArr.forEach(function(current, index, array) {
            current.value = "";

```

```

    });

    fieldsArr[0].focus();
  },

  displayBudget: function(obj) {
    var type;
    obj.budget > 0 ? type = 'inc' : type = 'exp';

    document.querySelector(DOMstrings.budgetLabel).textContent =
formatNumber(obj.budget, type);
    document.querySelector(DOMstrings.incomeLabel).textContent =
formatNumber(obj.totalInc, 'inc');
    document.querySelector(DOMstrings.expensesLabel).textContent =
formatNumber(obj.totalExp, 'exp');

    if (obj.percentage > 0) {
      document.querySelector(DOMstrings.percentageLabel).textContent
= obj.percentage + '%';
    } else {
      document.querySelector(DOMstrings.percentageLabel).textContent
= '---';
    }
  },

  displayPercentages: function(percentages) {

    var fields =
document.querySelectorAll(DOMstrings.expensesPerLabel);

    nodeListForEach(fields, function(current, index) {

      if (percentages[index] > 0) {
        current.textContent = percentages[index] + '%';
      } else {
        current.textContent = '---';
      }
    })
  }
}

```

```

    });

    },

    displayMonth: function() {
        var now, months, month, year;

        now = new Date();

        months = ['January', 'February', 'March', 'April', 'May', 'June',
'July', 'August', 'September', 'October', 'November', 'December'];
        month = now.getMonth();

        year = now.getFullYear();
        document.querySelector(DOMstrings.dateLabel).textContent =
months[month] + ' ' + year;
    },

    changedType: function() {

        var fields = document.querySelectorAll(
            DOMstrings.inputType + ',' +
            DOMstrings.inputDescription + ',' +
            DOMstrings.inputValue);

        nodeListForEach(fields, function(cur) {
            cur.classList.toggle('red-focus');
        });

        document.querySelector(DOMstrings.inputBtn).classList.toggle('red');

    },

    getDOMstrings: function() {
        return DOMstrings;
    }
}

```

```

    };

    })();

// GLOBAL APP CONTROLLER
var controller = (function(budgetCtrl, UICtrl) {

    var setupEventListeners = function() {
        //kreiramo funkciju za event listener-e
        //preko kojih cemo osluskivati da li se desavaju klikovi
        //na predefisanim poljima za unos, brisanje i izmene
        var DOM = UICtrl.getDOMstrings();

        document.querySelector(DOM.inputBtn).addEventListener('click',
ctrlAddItem);

        document.addEventListener('keypress', function(event) {
            if (event.keyCode === 13 || event.which === 13) {
                ctrlAddItem();
            }
        });

        document.querySelector(DOM.container).addEventListener('click',
ctrlDeleteItem);

        document.querySelector(DOM.inputType).addEventListener('change',
UICtrl.changedType);
    };

    var updateBudget = function() {

        // 1. Izracunaj budzet
        budgetCtrl.calculateBudget();

        // 2. Vрати budzet
        var budget = budgetCtrl.getBudget();
    };

```

```

        // 3. Prikazi budzet
        UICtrl.displayBudget(budget);
    };

    var updatePercentages = function() {

        // 1. Izracunaj procenke
        budgetCtrl.calculatePercentages();

        // 2. Procitaj procenke iz budzet kontrolere
        var percentages = budgetCtrl.getPercentages();

        // 3. Azuriraj prikaz sa novim procentima
        UICtrl.displayPercentages(percentages);
    };

    var ctrlAddItem = function() {
        var input, newItem;

        // 1. Pokupi podatke iz forme
        input = UICtrl.getInput();

        if (input.description !== "" && !isNaN(input.value) && input.value > 0)
        {
            // 2. Dodaj item u budzet kontroler
            newItem = budgetCtrl.addItem(input.type, input.description,
input.value);

            // 3. Dodaj item u UI kontroler
            UICtrl.addListItem(newItem, input.type);

            // 4. Obrisati polja
            UICtrl.clearFields();

            // 5. Izracunaj novi budzet
            updateBudget();
        }
    };

```

```

        // 6. Izracunaj i azuriraj procenke
        updatePercentages();
    }
};

//brisanje itema
var ctrlDeleteItem = function(event) {
    var itemID, splitID, type, ID;

    itemID = event.target.parentNode.parentNode.parentNode.id;

    if (itemID) {

        //Dolazak do tipa id-a iz onog placeholder html-a
        //na 261 liniji koda
        splitID = itemID.split('-');
        type = splitID[0];
        ID = parseInt(splitID[1]);
        //dobijamo odvojeno id i odvojeno br id-a

        // 1. brisemo item iz data strukture
        budgetCtrl.deleteItem(type, ID);

        // 2. brisemo ga sa prikaza
        UICtrl.deleteListItem(itemID);

        // 3. azuriramo budzet
        updateBudget();

        // 4. racunamo nove procenke
        updatePercentages();
    }
};

```

```

//closure za app kontroler
return {
    init: function() {
        console.log('Application has started. ');
        UICtrl.displayMonth();
    }
};

```



```
        UICtrl.displayBudget({
            budget: 0,
            totalInc: 0,
            totalExp: 0,
            percentage: -1
        });
        setupEventListeners();
    }
};

})(budgetController, UIController);

//inicjalizacija nase aplikacije preko init funkcije koja je jedina vidljiva spolja.
controller.init();
```