

JAVASCRIPT PROGRAMMER

PRIRUČNIK 1.0



Dobrodošao u "**JAVASCRIPT PROGRAMER**" - ovaj priručnik je kreiran na osnovu gradiva sa predavanja o JavaScript-u na platformi postani.it. Nastao je iz želje da što većem broju ljudi omogućim pristup najvažnijim konceptima programiranja i JavaScript-a.

U ovom priručniku ćeš pronaći ključne koncepte o JavaScript programiranju, koja uključuju:

- **Osnovne Tipove:** Upoznaćeš se sa različitim osnovnim vrstama informacija u JavaScript-u i kako da ih koristiš efikasno.
- **Funkcije:** Naučićeš kako da definišeš i koristiš funkcije, što je ključni koncept u svakom programskom jeziku.
- **Grananja i Petlje:** Razumećeš kako se upravlja izvršavanjem koda kroz različite putanje i ponavljanja, što omogućuje kontrolu nad tokom izvršavanja programa.
- **DOM Osnove:** Upoznaćeš se sa Document Object Modelom (DOM), kako bi mogao da komuniciraš sa HTML stranicama i manipulišeš njima.
- **Asinhroni Kod:** Naučićeš kako da efikasno radiš sa operacijama koje se obavljaju u pozadini, što je važno za obradu dugotrajnih procesa.
- **Komunikacija sa Serverom:** Otkrićeš kako da uspostaviš komunikaciju sa serverom putem internet zahteva (HTTP), što je ključno za razvoj dinamičkih web aplikacija.

Ovaj priručnik je namenjen svima koji žele da nauče programiranje kroz JavaScript, prošire svoje postojeće znanje ili samo ponove gradivo s vremena na vreme.

Ovaj dokument možeš slobodno podeliti sa prijateljima i odštampati bez ikakve naknade.

Prodaja ovog dokumenta nije dozvoljena.

Ivan | Postani IT

SADRŽAJ

1

UVOD

Osnove

Promenljive

Brojevi i NaN

Const, Let i Var

Boolean

String

Spajanje stringova - Konkatenacija

String metode

Null i Undefined

2

FUNKCIJE

Kreiranje funkcija

Callback funkcije

Anonimne funkcije

Arrow funkcije

Debugging

Hoisting

Scoping

Try and Catch

3

STRUKTURE

Poređenje

Konverzija tipova

Nizovi

Stack i Queue

Objekti

Window objekat

Vrednost i Referenca

Metode nizova

New i This

4

GRANANJA I PETLJE

Uslovi

IF, ELSE, ELSE IF

AND Logički operator

OR Logički operator

NOT Logički operator

Ugnježdeni IF

Ternarni operator

Switch

For Petlja

For sa nizovima

Ugnježdeni for

While petlja

Break i Continue

Rekuzija

5

JAVASCRIPT DOM

Uvod

Selektori

Promena stilova

Promena teksta

Promena atributa i brisanje

Prolazak kroz DOM

Događaji

Prevent Default

6

ASINHRONI KOD

Asinhronost i Event Loop

Promisi

Varijacije Promisa

Async - Await

KOMUNIKACIJA SA SERVEROM

Uvod (AJAX i API)

JSON - JavaScript Object Notation

HTTP/API

Postman

XMLHttpRequest

Fetch

Axios

UVOD

OSNOVE



[POGLEDAJ LEKCIJU](#)



Programiranje je naređivanje računaru da uradi zadatak koji se od njega traži bez greške.



Najosnovniji koncepti u programiranju upotrebom JavaScript-a:

1

PROMENLJIVE

2

STRINGOVI

3

NIZOVI

4

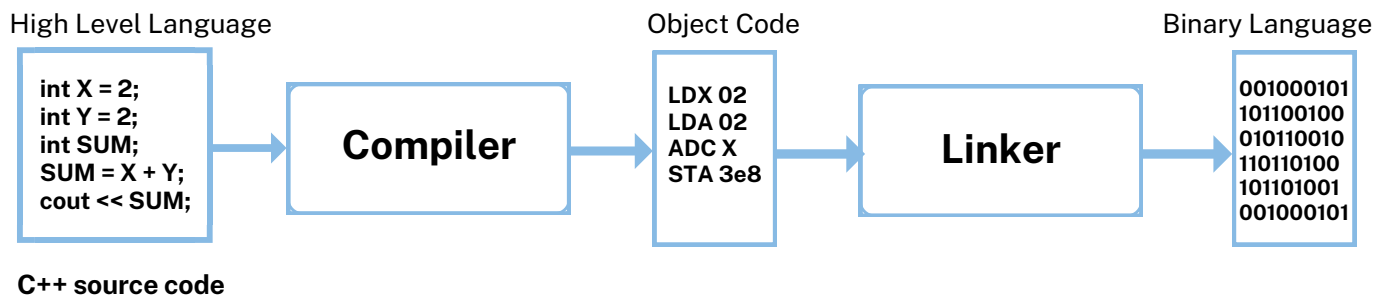
USLOVI

5

PETLJE



High Level Language - najviši nivo programiranja, jezik koji je najpribližniji čoveku, tj. jezik koji mi možemo da razumemo.



Uzmimo za primer sabiranje dva broja. Kod koji mi iskucamo (to je npr C++ kod) odlazi u kompajler. Kompajler je zapravo kao neki “filter” koji od tog koda pravi asemblerski kod (on spušta taj kod jedan nivo ispod po apstrakciji). Asemblerski kod barata sa registrima (delovima) procesora. Posle kompajlera i asemblerskog koda dolazi linker koji taj kod pretvara u mašinski kod. Mašinski ili binarni kod je niz nula i jedinica koji posle dolazi direktno u sam računar koji taj kod razume i izvršava taj program.

PRAVILA JEZIKA

Dodeljivanje vrednosti 2 u različitim programskim jezicima

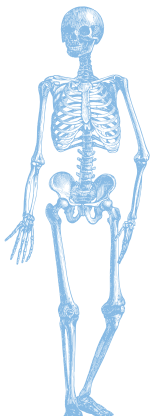
int broj = 2; (Java)
const broj = 2; (JavaScript)
broj = 2 (Python)

- JavaScript je najzahtevnija oblast kod frontend programiranja. Kad se ovo savlada sve će biti dosta lakše kasnije.
- Uz pomoć JavaScript-a mogu da se prave različite aplikacije kao što su web, desktop ili mobilne aplikacije.
- Uz pomoć HTML-a i CSS-a se prave statičke web aplikacije, aplikacije koje su samo prezentacione (vidimo strukturu i boje ali nemamo nikakvu interaktivnost između čoveka i web sajta).

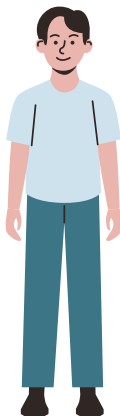
Tri glavna alata za web programiranje:

- 1** HTML ("skelet") - sadržaj sajta / content
- 2** CSS ("koža") - izgled sajta / look and fee
- 3** JAVASCRIPT ("mišići") - interaktivnost i logika

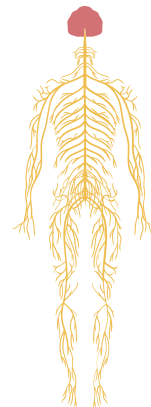
HTML



CSS



JAVASCRIPT



Izvršavanje koda u konzoli



Konzola predstavlja jedan vid interfejsa. Interfejs je sve ono što nam neka mašina omogućava da komuniciramo s njom. Npr. kada bismo otišli na bankomat, interfejs bankomata bi bila tastatura, deo gde možemo ubaciti karticu i mesto gde nam izlazi novac. Kod računara, taj interfejs je konzola, tj. način na koji mi komuniciramo sa računarom. Ona se može nalaziti i u našem pretraživaču. Svaki pretraživač (Chrome, Mozilla Firefox, Safari) ima u sebi konzolu.

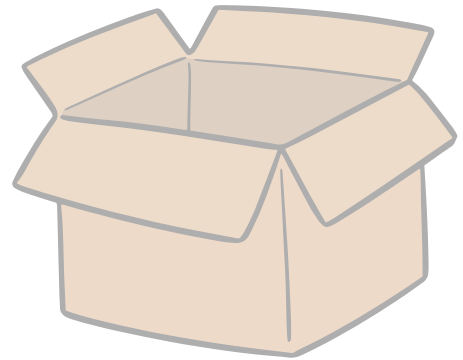
PROMENLJIVE

 [POGLEDAJ LEKCIJU](#)

-  Promenljive služe da čuvaju vrednost. Mogu se zamisliti kao kutija u koju ubacujemo nešto na čuvanje.
-  Svaka promenljiva ima tip podataka koji može da čuva. Promenljive logičkog tipa na primer govore samo da li je nešto tačna ili netačna vrednost (**true** i **false**), one ne znaju ništa više od toga da čuvaju.

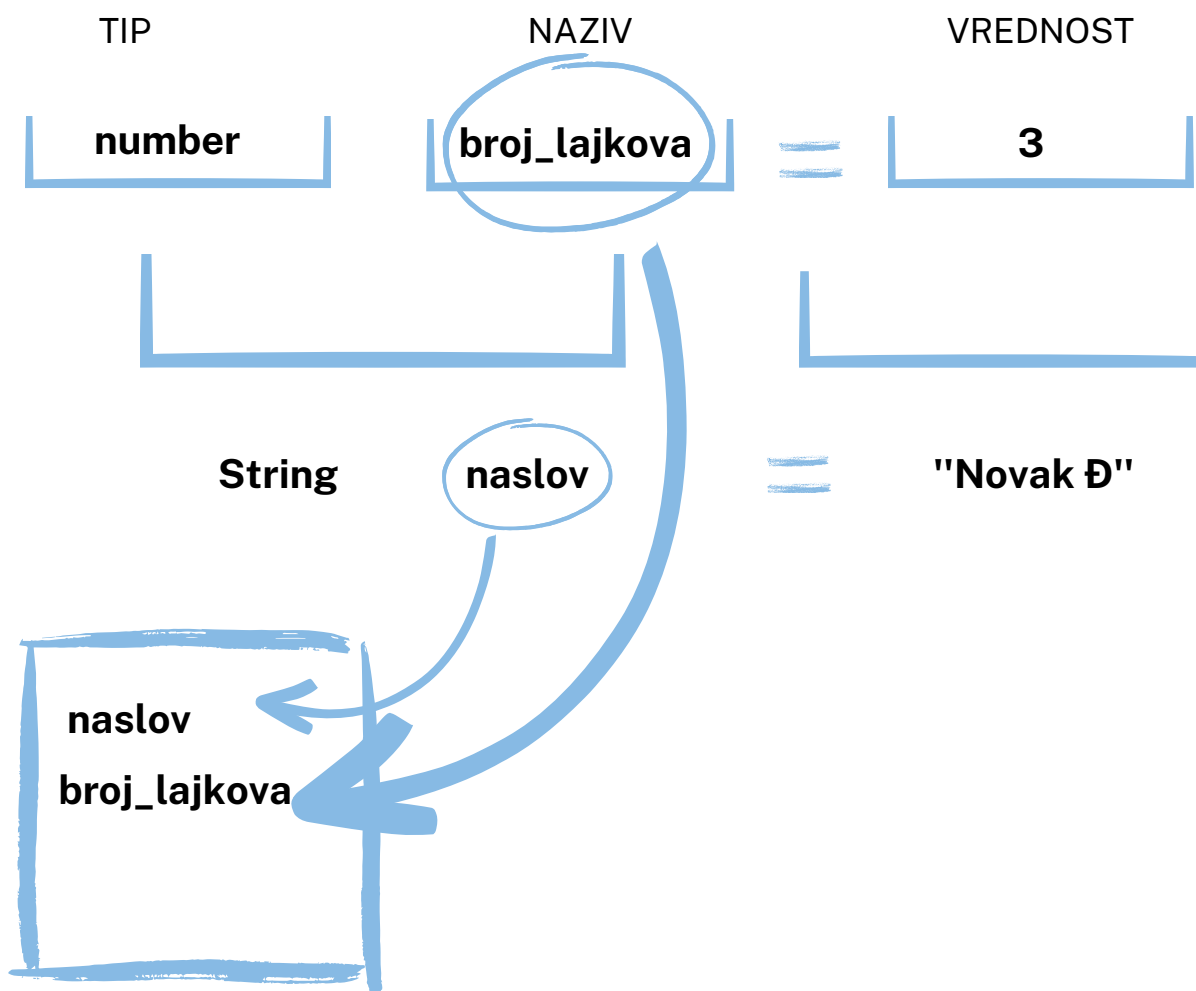
Osnovni (primitivni) tipovi u JS:

-  **Number (numerički)**
-  **String (tekstualni)**
-  **Boolean (logički)**
-  **Null (nulta vrednost)**
-  **Undefined (nedefinisano)**



Kako se koriste promenljive?

Deklaracija i definicija
Tip - Naziv - Vrednost



 Primer deklaracije promenljive bi bilo kao bila kada bismo samo nacrtali kutiju i stavili njen naziv, a ne bismo u nju ubacili ništa (to bi bila prazna kutija), dok se pod definisanjem promenljive smatra deklaracija + dodeljena vrednost toj kutiji.

 U JavaScript-u nije potrebno definisati tip promenljive, odnosno nije potrebno ispred naziva promenljive pisati number ili string jer je JavaScript slabo tipizirani, odnosno dinamički tipizirani programski jezik. JavaScript će automatski u toku izvršavanja koda odrediti kog tipa je ta promenljiva i dodeliti taj tip promenljive, tako da programer ne mora da vodi računa o tome.



DINAMIČKI TIPIZIRANI JEZIK // js, python, php



STROGO TIPIZIRANI JEZIK // c, c++, java, pascal



JS konzola radi u tzv. **repl mode-u**, što znači: pročitaj-izvrši-prikaži-vrti opet

BROJEVI I NAN



[POGLEDAJ LEKCIJU](#)



JavaScript ima samo jedan tip za brojeve: **number**. S jedne strane, to je dobro jer ne moramo da vodimo računa o različitim tipovima za broj, ali sa druge strane to je loše jer gubimo određenu preciznost. Međutim, nama ovde preciznost nije toliko važna, ona je bitna kada radimo programiranje u neke naučne svrhe za neko precizno matematičko izračunavanje.



Ukoliko želimo da stavimo neki eksponent, to bi bilo **, tako da ukoliko želimo da napišemo 2^2 to bi bilo **2**2**, 2^3 bi bilo **2**3** itd.



JavaScript podržava i decimalne brojeve u okviru tipa number.

Jačina operacije

() zagrade

a eksponent

× množenje

÷ deljenje

+ sabiranje

- oduzimanje

P	E	M D	A S
1	2	3	4
Parentheses	Exponents	Multiply or Divide	Add or Subtract
()	a ²	X or /	+ or -



$3 + 7 * 6 / 3 = 3 + 42 / 3 = 3 + 14 = 17$



$(6 * 4) / 3 - 6 + 2 = 24 / 3 - 6 + 2 = 8 - 6 + 2 = 2 + 2 = 4$



Komentari su olakšica programeru, oni se ne izvršavaju u JavaScript-u nego služe da opišu deo koda i omoguće nam da ostavimo neke beleške u kodu kako bismo se kasnije lakše snašli. Međutim, dobra praksa je da izbegavamo komentare u kodu i da pišemo kod na taj način da on jednostavno opisuje nazivima svojih promenljivih i funkcija ono što on radi.



Komentari se u praksi ne koriste toliko često.



Komentar pišemo tako što napišemo **//ovo je komentar**.



NaN - not a number

0 / 0 = NaN

1 + NaN = NaN

typeof(NaN) = number

typeof 42 = number



Kada dobijemo NaN znači da je došlo do greške prilikom izračunavanja neke operacije. Kada dolazi do te greške? Npr. ukoliko dođe do pokušaja da se izvrši deljenje sa nulom (znamo da sa nulom ne može da se deli) mi ćemo dobiti kao rezultat NaN. To zapravo nije nikakva greška nego samo jednostavan način u JavaScript-u da se prikaže kada nešto ne može da se izračuna i kada nešto nije broj.



typeof funkcija je korisna kada god nismo sigurni šta se nalazi u nekoj promenljivoj, tako da možemo na ovaj način da ispitamo kog je ona tipa.

CONST, LET I VAR



[POGLEDAJ LEKCIJU](#)

LET

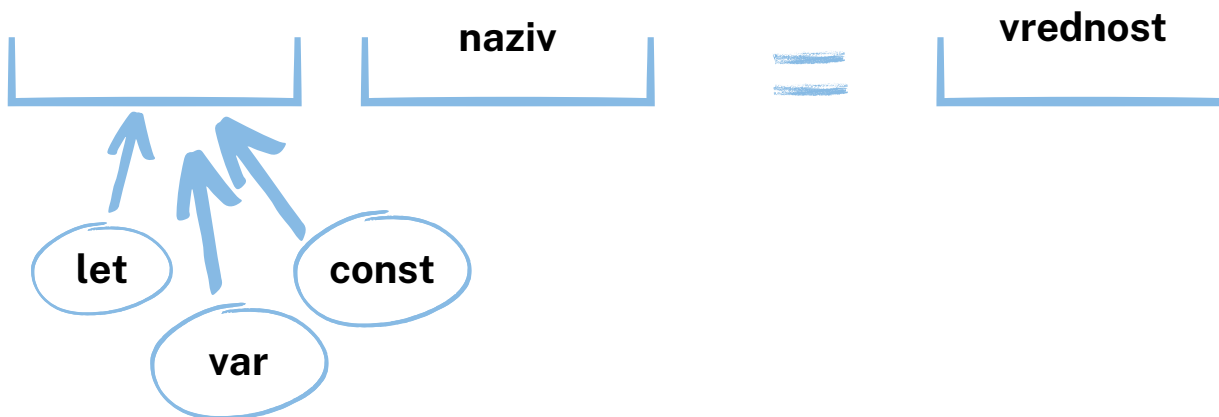
Kada nešto stavimo u let promenljivu mi možemo da menjamo njenu vrednost kasnije, ona nije fiksirana.

CONST

Kada koristimo const, to je zapravo jedna konstanta i nju ne možemo kasnije da menjamo.

VAR

Kada nešto definišemo sa var, mi možemo da menjamo vrednost. Var se ne koristi više u novijim verzijama JavaScript-a, ali je bitno da ga znamo u slučaju da naletimo na kod koji ga i dalje koristi.



```
let brKoraka = 230  
brKoraka = 250
```

👉 U programiranju težimo ka tome da pišemo stvari što jednostavnije kako bismo što manje vremena potrošili.

korpa += 1 je isto što i **korpa = korpa + 1**

korpa -= 1

korpa *= 2 je isto što i **korpa = korpa * 2**

korpa /= 2

👉 U JavaScript-u je bitno da pišemo ; na kraju svake komande, tako da bi izgled svake promenljive trebao na kraju da završava sa ;

BOOLEAN

 [POGLEDAJ LEKCIJU](#)

 Kod ovog logičkog tipa postoje samo dve vrednosti, a to su **true** i **false**.

 Svaki programski jezik ima rezervisane reči, tj. reči koje taj programski jezik razume bez prethodne definicije.

 Kada je promenljiva logičkog tipa, odnosno kada promenljivu logičkog tipa nazovemo tako da nam ona služi konkretno za true ili false, nije dobra praksa u nju stavljati nešto drugo. Ovo nije moguće uraditi u drugim programskim jezicima kao što su Java ili C++ zbog toga što tamo prilikom inicijalizacije definišemo kog je tipa promenljiva i ako stavimo da je ona logičkog tipa (boolean) dobićemo grešku kada god pokušamo da stavimo u nju nešto što nije tog tipa. U JavaScriptu to može da se radi i postoji rešenje za to koje se naziva **typescript**.

 **Typescript** je jedan omotač oko JavaScript-a koji vodi računa o ovim tipovima za nas, tako da ukoliko želimo da stavimo nešto da bude logičkog tipa pa posle toga hoćemo da promenimo i stavimo neki broj ili tekst unutar te promenljive, mi ćemo dobiti grešku.

STRING



[POGLEDAJ LEKCIJU](#)



Tekstualni tip (string) - definisanje nekog teksta u JavaScript-u.



"Ivan" - 'Ivan'



"Ivan'

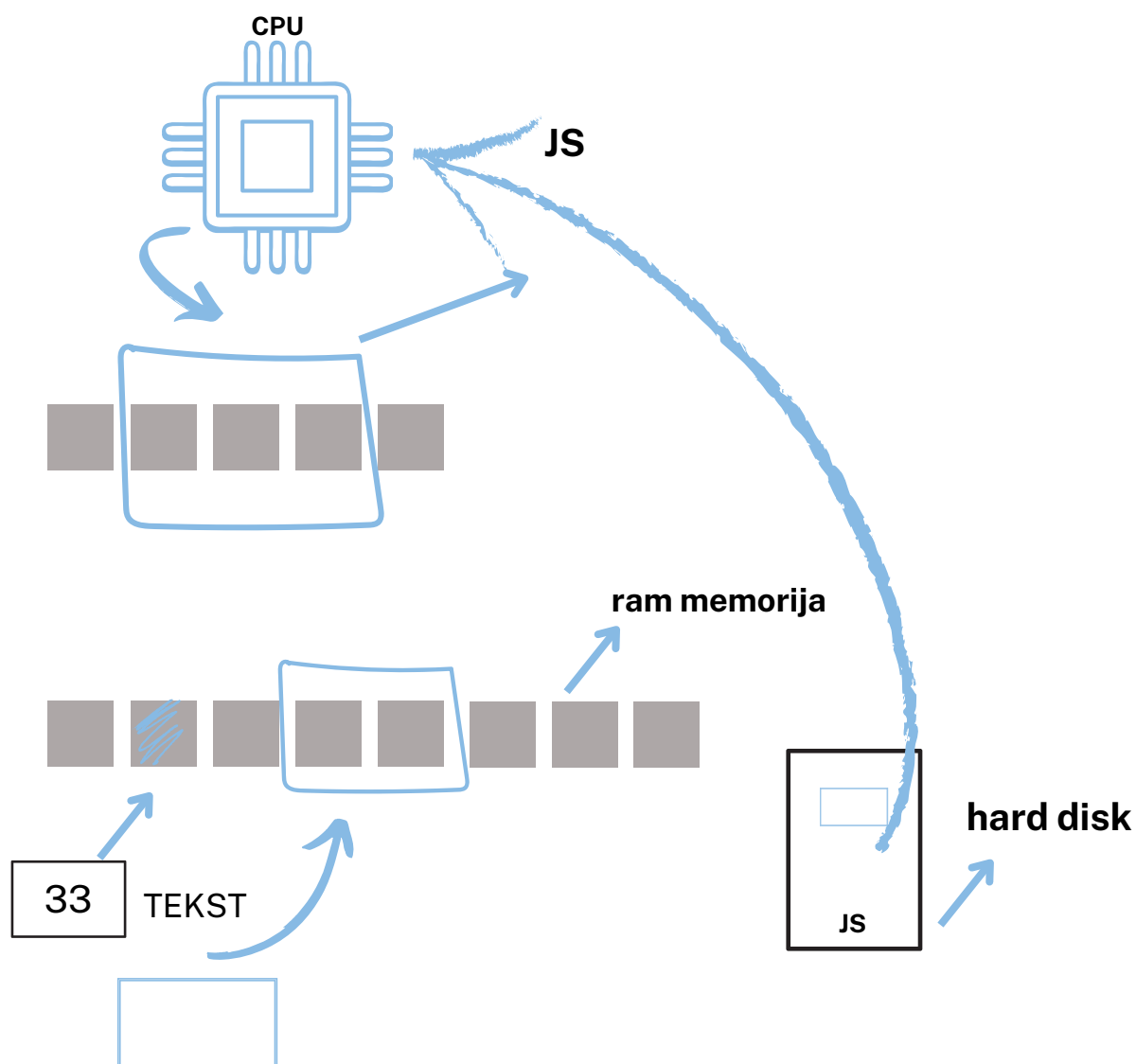


'Ivan"



Ukoliko želimo da unutar jednog stringa navodimo nešto drugo, ne možemo da koristimo iste navodnike. Zato je dobro da koristimo kombinaciju jednostrukih i dvostrukih navodnika. Npr. 'On mi je rekao: "Nauči da programiraš"' ili "On mi je rekao: 'Nauči da programiraš'".

Kako se izvršavaju programi u računaru?

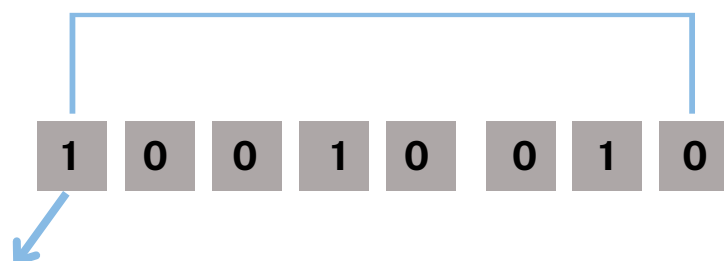


 **Hard disk** - ovde se nalazi JS fajl koji se izvršava. Mi želimo da pokrenemo ovaj JS fajl, on će da dođe do procesora, nakon toga će procesor da odvoji neki deo radne memorije (RAM) za ovaj program i ona će da pripada samo tom našem programu. Ovaj program koristi memoriju tako što u nju stavlja naše promenljive.

 Ako zamislimo memoriju kao niz nekih polja, možemo zamisliti da se neki naš broj koji definišemo kao npr. let brojKoraka uskladišti u memoriju.

 Kada napišemo neki tekst i njega stavimo u memoriju on će možda da zauzme 2 polja ove memorije.

8 x bit = bajt



bit može imati vrednost 0 ili 1

8 bita = 1 bajt

64 bita = 8 bajta

boolean zauzima 1 bit memorije

string zauzima 2 bajta po karakteru

☛ Kada bismo pogledali od čega se sastoji radna memorije koju koristi naš JS program, videli bismo da se u njoj nalazi određen broj ćelija i da svaka ćelija može da sadrži vrednost 1 ili 0. Svaka ćelija može da čuva 1 bit.

☛ Bit može biti 0 ili 1. Osam bitova je bajt. Kada kažemo bajt mislimo na 8 bitova, tj. 8 ćelija u memoriji.

☛ Svaki osnovni tip podataka u JS, bio on number, string ili boolean, zauzima određeno mesto u memoriji. Npr. number zauzima 64 bita = 8 bajta; boolean (jer on govori samo da li je nešto true ili false, tu memoriju možemo da uskladištimo kao 1 ili 0) zauzima samo 1 bit (znači samo jednu ćeliju u memoriji); string 2 bajta po karakteru. Ove vrednosti su vezane samo za JavaScript, ukoliko bismo gledali C++ ili Javu, videli bismo da tamo osnovni tipovi zauzimaju različitu količinu memorije.

☛ String je u stvari niz karaktera i kao takav on se čuva u memoriji.

0	1	2	3	4			
T	e	k	s	t			

let naziv = "Tekst"

naziv [0] = T

naziv [1] = e

 Ako zamislimo da je ovo naša memorija i da ona ima određena polja i definišemo neki string, npr. let naziv = "Tekst", svaki taj karakter će biti smešten u 1 ćeliju i mi ćemo moći da pristupimo pojedinačnim karakterima u okviru ove ćelije. Kako bismo pristupili npr. slovu T? Tako što napišemo naziv[0]. Ovde stavljamo 0 zato što u memoriji postoje tzv. **indexi**. To su pozicije u memoriji. Indexi uvek kreću od 0. To znači da svako polje iznad ima svoj index. Prvo polje je 0, drugo je 1, treće je 2... Kada bismo napisali **naziv[1]** očekujemo da nam se prikaže ovo što je na mestu prvog indexa (e). Tako možemo do kraja stringa da preuzimamo samo po 1 karakter. Ovo se ne koristi toliko često ali je bitno da znamo kako možemo da pristupimo određenom elementu tog stringa.

 Kada želimo da pristupimo indexu neke promenljive to radimo tako što napišemo uglaste zagrade [] i stavimo unutar njih index.

 Stringovi imaju svoje ugrađene funkcije. Njih možemo zamisliti kao neke alate u okviru JavaScript-a koji nam olakšavaju rad tako što za nas nešto izračunavaju.

 Funkcija **length** izračunava koliko stvarno postoji karaktera.

SPAJANJE STRINGOVA

-Konkatenacija-

 [POGLEDAJ LEKCIJU](#)

-  Konkatenacija označava dodavanje stringa na string.
-  Ovo je primenljivo u svim programskim jezicima i koristi se svakodnevno.

"test" + "test" = "testtest"

-  + spaja dva stringa.
-  String ne smemo da pišemo bez navodnika jer onda JavaScript pokušava da pronađe to kao neku ugrađenu reč, a ona ne postoji.

-Template Literal-

-  **"Kupio si" + proizvod + ". Količina" + količina + "po ceni od" + cena + "dinara"**
-  Ovo nije najsjajnije rešenje zato što imamo previše pluseva, vodimo računa gde bi trebalo da stavimo razmak i previše izgleda konfuzno kada je neki veći tekst u pitanju.
-  Ono što je bolje i novije rešenje u JS-u je tzv. **String Template Literal**. To je drugačiji način predstavljanja stringova.
-  **`Kupio si {proizvod}. Količina \${količina} po ceni od \${cena} dinara`**
-  Pišemo kao običnu rečenicu bez ikakvih pluseva, a kada hoćemo da napišemo neki izraz koji treba da se izvrši (promenljiva, sabiranje...) otvaramo vitičaste zagrade.

STRING METODE

 [POGLEDAJ LEKCIJU](#)



Kada je reč o stringovima, JavaScript nam pruža dodatne funkcionalnosti kao što su određene metode stringa i na taj način možemo da npr. skratimo string, pronađemo neku reč u stringu, spojimo stringove i još mnogo toga.

```
let myString = "Hello World!"
```



Ovaj string ima svoje metode koje dobijamo tako što posle naziva ove naše promenljive gde smo smestili string upišemo tačku i posle toga dobijamo neke funkcionalnosti tog stringa (replace, split...). Kako to funkcioniše ako smo rekli da je string primitivan tip? U pozadini JavaScript nama omogućava sve ove funkcije kada želimo da koristimo string. Svaki put prilikom korišćenje stringa, JavaScript kao da okruži taj naš string sa dodatnim funkcijama i dopusti nam da ih koristimo i da na taj način menjamo string. Kada se završi naša obrada stringa, string se i dalje čuva kao primitivni tip i nema nikakve dodatne opcije. To su neke dodatne ugrađene funkcionalnosti koje nam mnogo olakšavaju rad sa stringom, tako da ukoliko želimo da npr. uklonimo poslednji karakter ili nešto pronađemo u stringu, za sve to imamo ugrađene funkcije koje zapravo predstavljaju olakšice da ne bismo morali te stvari da pišemo ručno.

```
let myString = "Hello World!"  
myString.length  
12
```



Ukoliko želimo da znamo koliko karaktera se nalazi u našem stringu, to radimo tako što koristimo metodu **length**.

```
let myString = "Hello World!"  
myString.slice(0, 5)  
"Hello"
```



Ukoliko želimo da skratimo string i da ga isečemo te da ostavimo samo hello, to radimo tako što koristimo metodu **slice** i njoj dodajemo prvi index od koga želimo da sečemo, dakle želimo da sečemo od nultog do petog indexa, tj. dodajemo mu jedan string više u odnosu na ono što želimo da sečemo.

```
let myString = "Hello World!"  
myString.substr(6, 5)  
World
```



Substr je skraćenica za substring. Ovde kao rešenje dobijamo world a substr(6, 5) bi značilo da krećemo od indexa 6 i izvlačimo narednih 5 karaktera.

```
myString = "  Hello World!  "  
myString.trim()  
Hello World
```



Trim metoda automatski briše okolne prazne karaktere i ostavlja samo tekst. Ona ne briše prazne karaktere između, već samo sa početka i kraja.

```
let myString = "Hello World!"  
myString.charAt(4)  
o
```



Ovde smo iskoristili poziciju indexa da dobijemo karakter.

```
let myString = "Hello World!"  
myString.indexOf("World")  
6
```



Sa ovom metodom možemo da iskoristimo neki karakter ili tekst da bismo dobili poziciju indexa gde počinje taj karakter ili tekst. U ovom primeru, reč "World" kreće na šestom indexu.

```
let myString = "Hello World!"  
myString.replace("World", "Balkan")  
Hello Balkan!
```



Replace funkcija menja reč. Ovde smo promenili Hello World u Hello Balkan. Ova metoda nam vrati novi string.

```
let myString = "Hello World Hello World"  
myString.replaceAll("World", "Balkan")  
Hello Balkan Hello Balkan
```



Replace menja reč samo prvi put kada naiđe na nju, u slučaju da postoji više reči World u tekstu biće zamenjena samo prva, a ako iskoristimo **replaceAll** onda ćemo da zamenimo svako pojavljivanje te reči.

```
let myString = "Hello World!"  
myString.toUpperCase()  
HELLO WORLD
```

```
myString.toLowerCase()  
hello world
```



ToUpperCase pretvara sve karaktere u velika slova, a **toLowerCase** u mala. Ovo je korisno ukoliko na nekoj stranici ističemo naslov i želimo da on bude velikim slovima. Ili ukoliko želimo da neki tekst bude ispisan malim slovima, ne moramo da ga proveravamo reč po reč već nad tim stringom izvršimo toLowerCase i pretvorimo sve u mala slova.

NULL I UNDEFINED

(I Rad sa fajlovima)

 [POGLEDAJ LEKCIJU](#)

-  Null- nulta vrednost
-  Undefined- nedefinisana vrednost
-  Undefined predstavlja nedefinisane promenljive. Promenljiva nije definisana ukoliko joj nije dodeljena vrednost prilikom kreiranja.

Rad sa fajlovima

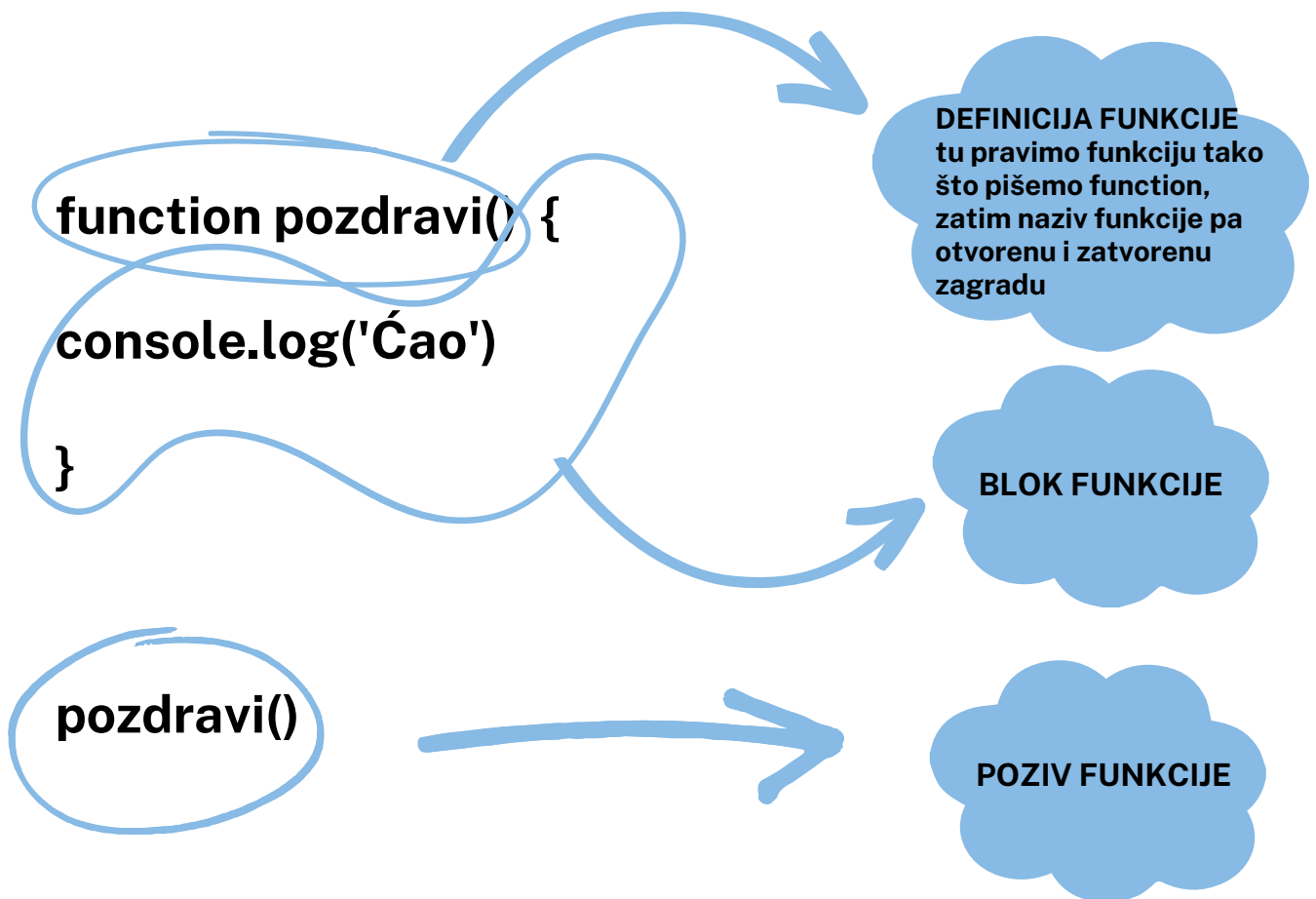
-  S obzirom da se JS fajl koristi na web stranicama, mi moramo taj JS fajl da povežemo sa HTML fajlom.
-  Kreiranje HTML fajla- **index.html**
-  Kreiranje JS fajla- **app.js**
-  JavaScript fajlovi imaju ekstenziju js, što znači JavaScript, tako da svaki fajl kome stavimo ekstenziju js postaje JavaScript fajl i automatski će biti prepoznat od strane VS Code-a.
-  Povezivanje JS fajla sa HTML-om postizemo tako što u okviru **head taga** napišemo **<script src = "app.js"></script>**.

FUNKCIJE

KREIRANJE FUNKCIJA

 [POGLEDAJ LEKCIJU](#)

 Funkcije služe da određenu logiku iskoristimo više puta, bez da je pišemo svaki put ručno ispočetka.



 Funkciju definišemo tako što pišemo **function** i dajemo **naziv** toj funkciji (npr. `pozdravi`). Funkcija je uvek neka akcija, pa je za njen naziv najbolje koristiti neki glagol. Zatim sledi **otvorena i zatvorena zagrada**, a nakon toga otvaramo sa **vitičastim zagrada** blok. Ovo što se nalazi unutar funkcije jeste **blok te funkcije** i to je ono što će se izvršiti svaki put kada budemo pozvali ovu funkciju.



SyntaxError- sintaksna greška koja znači da pravila pisanja JavaScript-a nisu ispunjena.



Kako se vraća rezultat neke funkcije? Za to postoji **return**. Kao što samo ime kaže "vrati", mi posle nje definišemo šta želimo da nam ta funkcija vrati.

www.postani.it



U programiranju ćemo često čuti **parametri** ili **argumenti**. Ispravno je da parametri služe prilikom definisanja funkcije i da se oni koriste u funkciji, a da su argumenti ono što se šalje kada se ta funkcija poziva.

CALLBACK FUNKCIJA

 [POGLEDAJ LEKCIJU](#)



Funkcije možemo da šaljemo kao parametre druge funkcije.



Callback - kada jednoj funkciji prosledimo drugu funkciju i ona odlučuje kada će tu funkciju da upotrebi.

```
function saberiCallback(a, b, callback){  
  let zbir = a + b  
  callback(zbir)  
}
```

```
function štampajCenu(cena) {  
  console.log(cena)  
}
```

```
saberiCallback(1, 5, štampajCenu)
```

rešenje je 6

ANONIMNE FUNKCIJE



[POGLEDAJ LEKCIJU](#)



Anonimna funkcija je funkcija koja nema ime. Nju koristimo kada želimo nešto da izvršimo bez da je čuvamo i kasnije pozivamo ponovo. Nije nam bitno njeno ime.



Ovde moramo voditi računa o tome da svaki put kad otvorimo neku zagradu moramo da je i zatvorimo. Ne možemo da počinjemo sa otvorenom, običnom zagradom, a da završavamo sa vitičastom, zato što ta vitičasta zagrada nema svoj početak.

```
function saberiCallback(a, b, callback){  
  let zbir = a + b  
  callback(zbir)  
}
```

```
saberiCallback(1, 5, function (cena) {  
  console.log(cena)  
})
```

ARROW FUNKCIJE

 [POGLEDAJ LEKCIJU](#)

-  Arrow funkcije nisu ništa drugo nego samo drugačije definisanje funkcije. Ovo predstavlja drugačije pisanje obične funkcije. Koristi se mnogo češće od običnih funkcija danas tako da je bitno da se dobro nauče.
-  Već smo rekli da su sve funkcije faktički neke promenljive, tako da mi arrow funkciju definišemo tako što je stavljamo u neku promenljivu.

```
function pomnozi(a, b) {  
  return a*b  
}
```

```
console.log(pomnozi(1,2))
```

```
let pomnozi = (a, b) => {  
  return a*b  
}
```

ovo su iste funkcije, samo drugačije definisane. Prva je obična a druga arrow.

-  Ovde smo napravili promenljivu pomnozi i unutar pomnozi promenljive, tj. naše funkcije, smo stavili znak jednako i parametre (a i b), zatim smo stavili arrow znak (=>) i napisali telo te funkcije.



Kada imamo jedan parametar ne moramo da stavljamo zagrade kod parametra, npr. `const pomnozi = ime=>{}` međutim ukoliko imamo 2 parametra, zagrade su obavezne, npr. `const pomnozi = (ime, prezime)=>{}`



Još jedan trik kod arrow funkcija jeste da mi arrow funkcije koje nešto vraćaju možemo da napravimo u jednom redu, npr:

<code>let podeli =(a,b)=>{</code>	<code>=====</code>	<code>let podeli =(a,b)=>a/b</code>
<code> return a/b</code>	<code>=====</code>	
<code>}</code>		



Pretvaranje običnih funkcija u arrow funkcije

```
function pozdravi(ime) {  
  return "Zdravo" + ime  
}
```

obična funkcija

```
let pozdravi=ime= {  
  return "Zdravo" + ime  
}
```

arrow funkcija

```
let pozdravi=ime=>"Zdravo" + ime
```

skraćena arrow funkcija

DEBUGGING

 [POGLEDAJ LEKCIJU](#)



Posle return-a nema više izvršavanja koda, izlazi se iz funkcije..



Debug mode - aktivira se tako što otvorimo **inspect element** -> **source tab** i upalimo ikonicu pauze i onda će kod da se pauzira čim dođe do neke greške.



Breakpoint - trenutak prilikom izvršavanja na koji hoćemo da paudiramo izvršavanje da bismo videli šta se u tom trenutku dešava. Možemo imati više breakpointa.



Postoje još neke akcije koje su mnogo korisne kada radimo debugg-ovanje našeg koda, a to su:



nastavi do sledećeg breakpoint-a



preskoči trenutnu liniju



uđi u funkciju



step out



steps

ne koristi se često



deaktivirati ili aktivirati breakpoint-e

HOISTING



[POGLEDAJ LEKCIJU](#)



Hoisting je koncept u kojem pre početka izvršavanja koda JavaScript sve funkcije podigne gore na početak fajla (to se desi bez našeg znanja) i na taj način su onda sve funkcije dostupne kada god ih pozovemo. Ovo važi samo za funkcije, ne i za promenljive.



Bitno je da se na ove stvari ne oslanjamo i da se uvek trudimo da definišemo funkcije pre poziva. S obzirom da se arrow funkcije mnogo češće koriste, jednostavno ćemo biti prisiljeni da ih definišemo pre njihove upotrebe tako da je savet da se one uvek koristile.

SCOPING

 [POGLEDAJ LEKCIJU](#)

-  Scoping u JavaScriptu se odnosi na određivanje vidljivosti i dostupnosti promenljivih u određenom delu koda.
-  Scoping se generalno koristi i u drugim programskim jezicima, ali u svakom radi drugačije.

```
function pomnozi(a, b) {  
  let test = "Proba"  
  console.log(a*b)  
}
```

```
let a = 3  
let b = 4  
let test = "Test"
```

```
pomnozi(a, b)
```

-  Scope je zapravo okruženje gde se promenljiva `let test = "Test"` nalazi. Promenljive ispod funkcije u primeru se nalaze u **globalnom** scope-u, što znači da se one nalaze van bilo koje funkcije, nalaze se na nivou fajla. Promenljive `a` i `b` su u globalnom scope-u, a promenljiva iznad `test` se nalazi u **lokalnom** scope-u, što znači u scope-u te funkcije gore. U JavaScript-u su scope-ovi unutar vitičastih zagrada, što znači da svaki put kada imamo vitičastu zagradu mi ugnjezdimo tu promenljivu.
-  Kada hoćemo da pristupimo varijablama, da pristupimo nekim promenljivama, prvo gledamo u tom scope-u u kojem jesmo, posle toga izlazimo u spoljašnje scope-ove i gledamo dok ne dođemo do scope-a file (obrnuto ne važi).



Ovo je veoma bitna veština zato što se često u programiranju desi da nešto ne radi kako treba a najbolje se otkrije uzrok kada se duboko zađe i kada se pregledaju tačno funkcije, koji su parametri poslati, gde je uzrok nastanka problema... Debugging tool je zapravo jako dobar alat u rešavanju ovih problema. Dosta programera koristi samo `console.log` i ne koriste uopšte devTools, međutim kao dobar programer mnogo je preciznije i bolje da se koristi ovaj alat koji je ugrađen u svaki pretraživač.

TRY I CATCH



[POGLEDAJ LEKCIJU](#)



Kako baratati sa greškama koje se dese prilikom izvršavanja programa?



Prvi tip grešaka su one greške koje mogu da se dese, a mi na njih ne možemo da utičemo. To su npr. greške nedostatka memorije, neki problem sa hardverom itd. Sve to dovodi do nefunkcionisanja i zaustavljanja našeg programa.

Sa druge strane, postoje greške na koje mi možemo da utičemo, kao npr. ukoliko želimo da pristupimo fajlu koji ne postoji, možemo da napišemo kod koji će taj slučaj da obradi i da ispišemo neku poruku korisniku. Ili npr. pokušavamo da se konektujemo na bazu, u slučaju da pošaljemo neke pogrešne kredencijale za logovanje možemo u tom slučaju da ispišemo neku poruku ili da se pokuša opet.



Prvi tip grešaka su **errors**, a drugi tip grešaka su **exceptions**.



Mi možemo da utičemo na **exceptions**.



Sa ovim tipom grešaka rukujemo tako što koristimo **try i catch**. Try i catch je koncept u svim programskim jezicima.

```
try {  
    zdravoTekst.toUpperCase()  
}catch{  
    console.log("Nisi izvršio metodu nad tekstem!")
```



Try blok bi bio "pokušaj da izvršiš ono što se nalazi u bloku, a ako se dogodi neka greška onda upadni u catch i izvrši to što je tu".



Catch blok treba da uhvati exceptions.



Još jedna stvar koju možemo da uradimo jeste da u catch bloku prihvatimo i grešku. Grešku prihvatamo tako što u okviru običnih zagrada napišemo naziv tog našeg elementa koji prihvatamo, tj. tu našu grešku.

```
try {  
    console.log("zdravoTekst1".toUpperCase())  
} catch(error) {  
    console.log(error)  
    console.log("Nisi izvršio metodu nad tekstom!")  
}
```

PROMENLJIVE

POREĐENJE

 [POGLEDAJ LEKCIJU
www.postani.it](http://www.postani.it)



U programiranju se često koristi koncept Truthy i Falsy vrednosti.



Truthy vrednost je svaka vrednost koja se može tretirati kao true, dok je **falsy** obrnuto, vrednost koja se može tretirati kao false. Ove vrednosti inicijalno nisu true ili false, nisu boolean, ali se na neki način tako tretiraju u programskom jeziku.

Truthy: true, "neki tekst", 4

Falsy: false, " ", 0, null, undefined



Svi brojevi različiti od 0 se tretiraju kao **true** vrednost. Isto tako i neki tekst, tj. string koji ima neku vrednost, koji nije prazan string pa čak i string sa praznim karakterima (" ") se tretira kao **true** vrednost.

Sa druge strane, falsy vrednosti bi bile false, prazan string (""), međutim isto tako bi bila i 0. Ovo su najčešće truthy i falsy vrednosti u JavaScript-u. Ove stvari dolaze do izražaja kada se radi neko poređenje.

```
let a = 0
```

```
let b = false
```

```
console.log(a==b) true
```

```
console.log(a===b) false
```



Rekli smo da i 0 i false spadaju u falsy vrednosti. Kada bismo hteli da radimo njihovo poređenje (poređenje se radi tako što stavimo dva znaka jednakosti) dobili bismo true. Dakle, poredili smo dve vrednosti, pitali smo da li su jednake i naš JavaScript je rekao da su one jednake, iako su one različitog tipa, **a** je tipa number a **b** je tipa boolean, ali ukoliko iskoristimo dva znaka jednakosti i pitamo da li su jednaki, dobićemo **true**.



U JavaScript-u postoji poređenje sa dva znaka jednakosti i sa tri znaka jednakosti. U čemu je razlika? Ako uporedimo dve promenljive iz prethodnog primera sa tri znaka jednakosti, dobićemo **false**. Poređenje sa tri znaka jednakosti se naziva **striktno poređenje** i kod ovog poređenja je bitno da ove dve promenljive budu istog tipa, dok kod nestriktnog poređenja (sa dva znaka jednakosti) je bitno da su oni truthy ili falsy, da spadaju u jednu ili drugu kategoriju.

U praksi se tri znaka jednakosti koriste gotovo uvek, stoga je dobra praksa izbegavati dva znaka jednakosti zato što uvek želimo da poredimo iste tipove.

```
let c = null
```

```
let d = undefined
```

```
console.log(c==d)
```



Ukoliko u ovom primeru pokušamo da odradimo poređenje, dobićemo **true** zato što iako nisu istog tipa, one spadaju pod falsy. Dakle, njih JavaScript tretira kao falsy vrednosti, što u neku ruku i jesu jer one ne sadrže nikakvu vrednost, ne označavaju nikakvu vrednost konkretno. Ali kada bismo ovde odradili striktno poređenje (c===d) gde se prvo pita da li su one istog tipa, odmah bismo dobili false. Tako da je preporuka da se uvek koriste tri znaka jednakosti da ne bismo razmišljali o tipu i da bismo bili sigurni da dobro pišemo kod.

KONVERZIJA TIPOVA



[POGLEDAJ LEKCIJU](#)

```
let a = 1
let b = "Zdravo"

console.log(a+b)
```

= 1Zdravo

```
let a = 1
let b = "Zdravo"

console.log(a-b)
```

=NaN



U slučaju da imamo dve promenljive, **a** koja je broj i **b** koja je string, kada bismo hteli da ispišemo njihov zbir dobili bismo 1Zdravo kao rešenje. Kako uopšte možemo da saberemo broj i string? Tako što JavaScript gleda da prevede promenljivu u odgovarajući tip kako bi izvršio operaciju nad ove dve promenljive. Šta JavaScript tu uradi? S obzirom da je logičnije da se 1 pretvori u string nego da se Zdravo pretvori u neki broj (Zdravo ne može da se pretvori ni u kakav broj), onda JavaScript pretvori broj 1 u string i kao string odradi 1+Zdravo. Nakon toga mi dobijamo spajanje stringa i to je upravo ovaj rezultat koji smo dobili. Dakle, JavaScript je za nas konvertovao ova dva tipa u odgovarajući tip koji može da podrži i jednu i drugu promenljivu i odradio je sabiranje.



Ovo ne može da važi za oduzimanje jer oduzimanje nije podržano od strane stringa. Jedino se mogu oduzeti brojevi, a s obzirom da JavaScript ne može **b** iz primera da pretvori u broj, mi ćemo tu da dobijemo NaN (not a number).

-  **Int**- ceo broj
-  **Double** ili **float**- decimalni broj
-  **parseInt**- ugrađena funkcija koju koristimo za konvertovanje stringa u broj

```
parseInt("23jdjasd")
```

```
23
```

-  Ukoliko odradimo `parseInt` u primeru koji sadrži brojeve i posle toga neke karaktere koje on ne može da konvertuje, on će krenuti redom i konvertovaće samo ono što može da pretvori u broj, a ostalo će ignorisati. Međutim, ukoliko je naš broj negde na kraju ili u sredini on neće uspeti to da konvertuje i dobićemo NaN. Dakle, da bi konverzija bila uspešna naš broj mora da bude na početku.

```
let a = "1.245"
```

```
let b = "4"
```

```
let zbir = parseFloat(a) + parseInt(b)                                = 5.245
```

```
console.log(zbir)
```

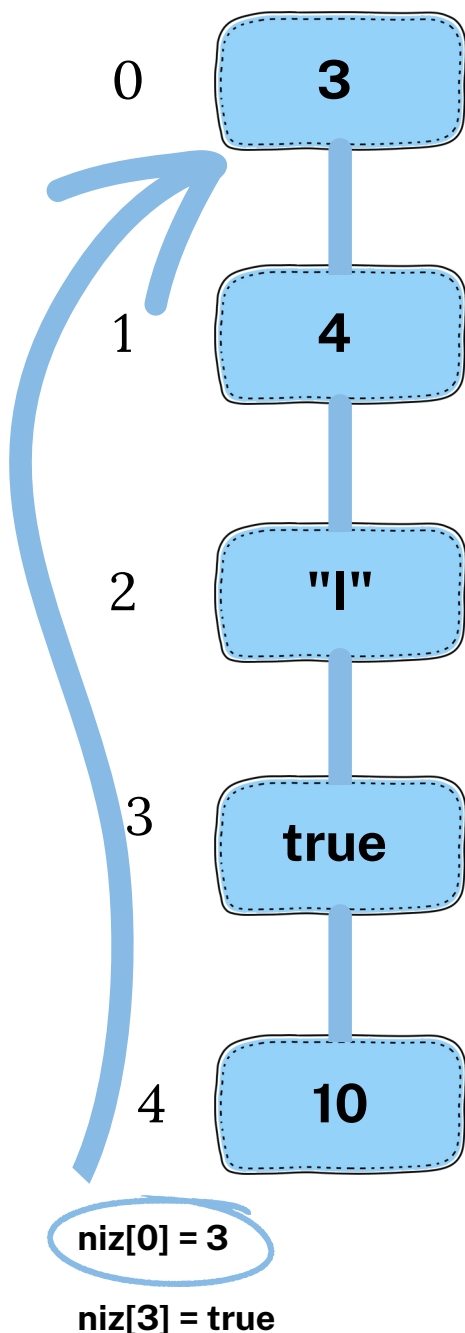
ili

```
console.log(a+b)                                                    = 1.2454
```

-  Da u ovom primeru nismo odradili konvertovanje, s obzirom da su oba stringa, dobili bismo samo broj 4 nalepljen na kraj stringa (1.2454), odradili bismo konkatenciju, tj. spajanje stringova i dobili bismo takođe string.

NIZOVI

[POGLEDAJ LEKCIJU](#)



Ovaj naš niz možemo predstaviti kao voz sa vagonima koji su povezani i gde svaki vagon sadrži neki element. Bitan nam je redosled i kao elemente tog niza možemo da stavljamo šta god poželimo. Zašto nam je bitan redosled? Jer da bismo pristupili ovom nizu, to moramo da učinimo preko indexa. Index je broj uz pomoć kojeg možemo direktno da pristupimo određenom elementu našeg niza i oni kreću od 0. Kada bismo hteli pristupiti prvom elementu našeg niza, to bismo uradili tako što bismo otvorili uglaste zagrade i napisali unutar njih naš index. Tako da ovaj **niz[0]** zapravo pristupa prvom vagonu, prvom elementu niza i dobijamo vrednost. Tako da je niz[0] = 3. Ovo je sličan koncept kao kod stringova.



Svaki niz se definiše kao i bilo koja druga promenljiva, napišemo **let niz** (ili bilo koji drugi naziv) i krećemo da mu dodeljujemo vrednost tako što stavljamo vrednosti unutar **uglastih zagrada**. Bitno je da elemente niza razdvojimo zarezom. Zarez može da se nalazi i na kraju [1, 2, 3, 4, 5,] i to ne smeta JavaScript-u, ali je bitno da se elementi odvoje zarezom jer kada bi bio samo razmak bez zareza dobili bismo grešku.

Takođe, možemo da pišemo i neki drugi niz npr. `let niz2 = ["test", true, 3, 412, false]`. Nismo ograničeni šta možemo da stavljamo u taj niz mada je preporučljivo da elementi niza budu uvek istog tipa.



Niz se na engleskom naziva **array**.



Kako se menjaju elementi niza?

```
let niz = [1, 2, 3, 4, 5]
```

```
niz[1] = 10
```

```
console.log(niz)
```

dobijamo 1, 10, 3, 4, 5



Ova promena vrednosti ne važi kod stringova.



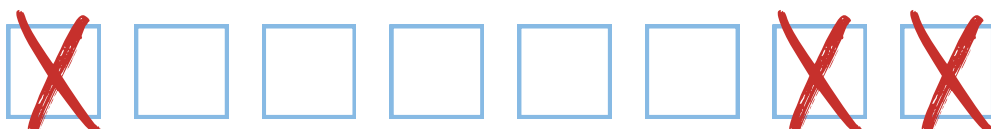
Za rad sa nizovima postoje određene ugrađene funkcije ili metode. Kako pristupamo njima? Tako što čim napravimo neki niz, posle toga napišemo **niz**. i automatski dobijamo metode koje su ugrađene i povezane sa tim nizom.

Kada kažemo funkcije ili metode, to su slične stvari. **Metoda** je zapravo funkcija unutar neke klase (klasa nizova, stringova itd). Kada napišemo **niz**. dobijamo pristup svim tim ugrađenim metodama za niz. Do sada smo koristili `length` koji nam govori koliko elemenata ima neki niz. Kako bismo pristupili poslednjem elementu? Tako što pristupamo onom indexu koji je za jedan manji od dužine niza.

STACK I QUEUE



[POGLEDAJ LEKCIJU](#)



1

PUSH(element)

2

POP()

3

SHIFT()

4

UNSHIFT(element)



Četiri popularne funkcije kada su nizovi u pitanju. One dodaju ili uklanjaju elemente niza.

1

PUSH - dodaje još jedan element na kraj niza **na kraj**

2

POP - uklanja element **sa kraja niza**. Kada bismo opet nad nizom uradili pop funkciju mi bismo sklonili još jedan element sa kraja niza.



Dakle, push funkcija dodaje element na kraj niza, pop funkcija sklanja element sa kraja niza. Kada hoćemo da dodamo neki element na početak niza mi moramo da u toj funkciji pošaljemo taj element, jer je to funkcija koja prihvata element da bi znala da doda na kraj niza. Sa druge strane, pop funkcija ne treba da prihvati ništa jer ona svakako briše element, tako da pop koristimo bez slanja argumenta.

3

SHIFT - uklanja element **sa početka niza**

4

UNSHIFT - dodaje element **na početak niza**



Unshift koji zapravo dodaje element na početak niza prihvata jedan argument, dok shift koji radi brisanje ne prihvata nikakve argumente.

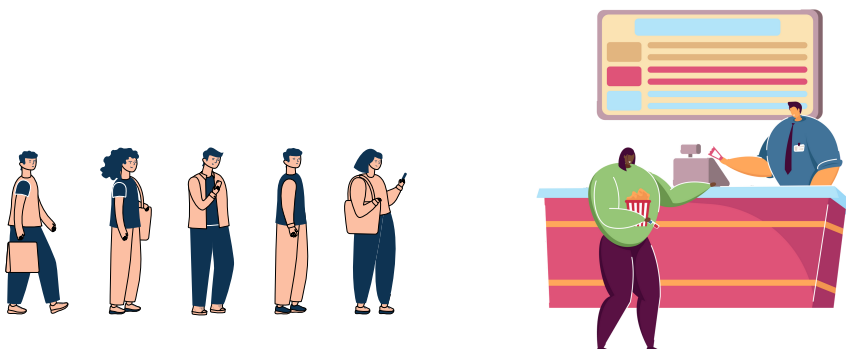
```
const niz = [1, 2, 3, 4, 5]
niz.push(10)
console.log(niz)
```

dobijamo [1, 2, 3, 4, 5, 10]

Zašto su bitne ove funkcije?



Ove funkcije su bitne jer dosta često treba da modelujemo 2 strukture kada modelujemo niz. Jedna od tih struktura se naziva red ili **queue**, a druga struktura se naziva **stack**. Koja je razlika?





Uzmimo za primer kupovinu karata za bioskop. Mi dolazimo da kupimo kartu, a ispred nas već nekoliko ljudi čeka. Prvo će biti uslužen onaj koji je prvi došao u red, posle će biti uslužen drugi, treći... Mi ćemo biti poslednji. Poslednji koji dođe će biti poslednji i uslužen. Ova struktura se naziva **red (queue)**.

Red je struktura u kojoj onaj element koji je prvi ubačen (u ovom slučaju osoba koja je prva došla), prvi i izlazi iz reda. Kako realizujemo ovu strukturu? Tako što napravimo jedan niz i u taj niz ubacimo prvu osobu koja je došla u red sa **push**, zatim ubacimo i drugu osobu koja je došla u red sa push, nakon toga treću... Međutim, kada bismo hteli da praznimo ovaj niz, praznili bismo ga tako što bismo obrisali ovu osobu koja je prva došla i koja je prva bila uslužena. Nju bismo obrisali tako što bismo iskoristili **shift**. Dakle, ukoliko bismo želeli da implementiramo ovakvu strukturu reda mi bismo koristili push i shift. Push bi bio za dodavanje na kraj reda, a shift za uklanjanje, tj. obradu prvog koji se stvorio u našem redu. Ovo je jedna struktura podataka koja je bitna da se nauči i koja je primenljiva u svim programskim jezicima. Ono što je bitno napomenuti jeste da ovu strukturu nazivamo **FIFO (First In First Out)**, što znači da u ovoj strukturi onaj koji prvi dođe prvi i izlazi.



PUSH I POP



Sa druge strane, stack struktura je malo drugačija. Ovo možemo zamisliti kao kada ređamo tegove na šipku u teretani. Poslednji izlazi sa šipke onaj teg koji je prvi ušao. Ovu strukturu nazivamo **LIFO (Last In First Out)** što znači da onaj koji je poslednji ušao prvi izlazi iz ove strukture. Kako ovu strukturu pravimo u okviru JavaScript-a? Tako što napravimo niz i ubacimo prvi element sa push, zatim ubacimo drugi element sa push, ubacimo treći element sa push... Međutim, mi ovde ne možemo da obradimo prvi element kao što je to bio slučaj kod reda, već u ovoj strukturi obrađujemo prvo ovaj poslednji element. Dakle, ovde koristimo funkciju **push** za dodavanje na kraj reda, ali koristimo i **pop** za uklanjanje. Ukoliko želimo da dođemo do prvog elementa, moramo pre toga da uklonimo sve elemente koji se nalaze posle njega.

OBJEKTI

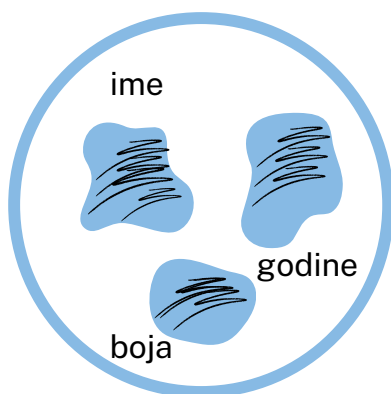
 [POGLEDAJ LEKCIJU](#)

 Objekti predstavljaju novu strukturu podataka, prvu su predstavljali nizovi.

```
const osoba = {  
  ime: "Milan",  
  godine: 30,  
  omiljenaBoja: "plava"
```

 Ovako definišemo objekat. Objekat se sastoji od tzv. parova **key-value**. Key-value par predstavlja **ključ i vrednost**. U ovom primeru, ključ bi bilo ime, a vrednost Milan, ključ bi takođe bile godine, a vrednost 30.

OSOBA



Nama objekat predstavlja samo način da grupišemo više promenljivih na neki smislen način, da ne bismo morali posle da im pristupamo preko indexa i da pamtimo na kojoj je poziciji koji element.

Kako se pristupa objektu?

```
console.log(osoba.ime)
```

 Kada god pristupamo nekom elementu objekta, uvek koristimo tačku.

 Elemente objekta nazivamo **property**.

```
console.log(osoba["ime"])
```

 Ako pristupamo objektu preko uglastih zagrada kao stringu, uvek navodimo ključ pod stringom.

Promene elemenata u objektu i još neke operacije

```
const student = {  
  ime: "Milan",  
  prezime: "Milanović",  
  položio: ["matematika", "programiranje", "filozofija"]  
  trenutniBodovi: {  
    algoritmi: 59,  
    webProgramiranje: 80  
  }  
}
```

```
student.prezime = "Milanić"  
student.položio.push("automatika")  
console.log(student.položio)
```



Push funkcija dodaje element na kraj niza.



Kod brisanja elemenata koristi se rezervisana reč **delete**. Napišemo delete, a zatim ono što želimo da obrišemo (npr. delete student.trenutniBodovi).

Čuvanje funkcija u objektima



S obzirom da smo rekli da se i funkcije čuvaju kao neke promenljive, ne postoji razlog zašto ne bismo mogli funkcije da čuvamo unutar objekta.

```
const osoba = {  
  ime: "Igor",  
  godine: 27,  
  omiljenaBoja: "plava",  
  pozdravi: function() {  
    console.log("Zdravooo")  
  }  
}
```

WINDOW OBJEKT

 [POGLEDAJ LEKCIJU](#)



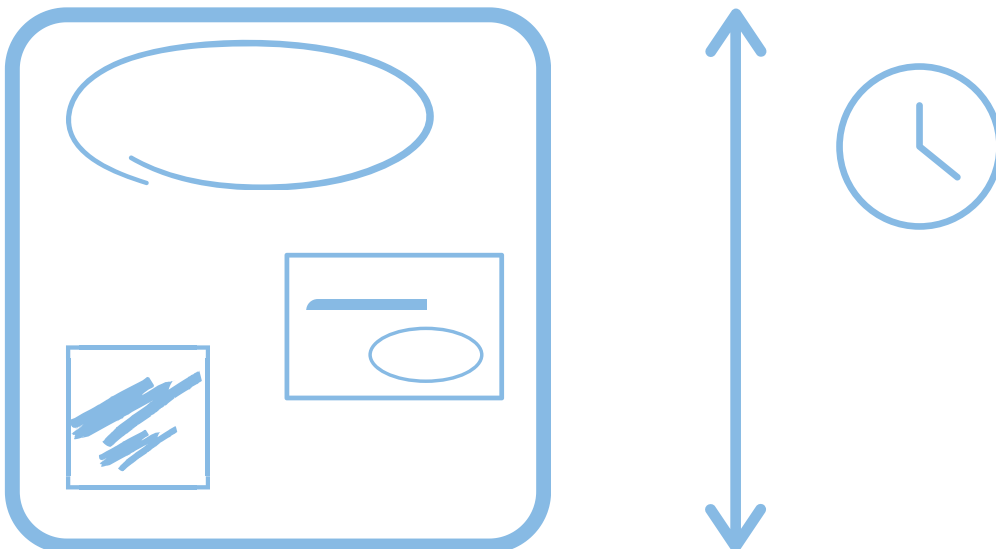
JavaScript kod živi u pretraživaču, to je okruženje u kojem se izvršava. Bitno je da imamo neki način komunikacije između JavaScript koda i pretraživača. Zbog čega nam je to bitno? Zato što od pretraživača nekad želimo da saznamo neke informacije, kao što su npr. koja je URL adresa na kojoj smo trenutno, koje je trenutno vreme i datum... Sve su to informacije koje možemo dobiti od pretraživača.



Kako izgleda ta komunikacija?



Ta komunikacija funkcioniše tako što pretraživač pošalje window objekat i mi taj window objekat možemo da koristimo iz JavaScript-a. Dakle, taj window objekat je zapravo neki interfejs, neka pristupna tačka koju mi iz JavaScript koda možemo da koristimo i možemo da koristimo sva njegova polja, kao i sve njegove metode.





Možemo zamisliti da je ovo naš pretraživač na slici iznad. Recimo da je ovo Chrome i unutar njega se nalazi naš JavaScript kod. Chrome kao pretraživač ima informacije koje je visine ovaj prozor, koje je tačno vreme, koji je tačan datum i još mnoge stvari.



Kako funkcioniše veza između pretraživača i JavaScript-a?



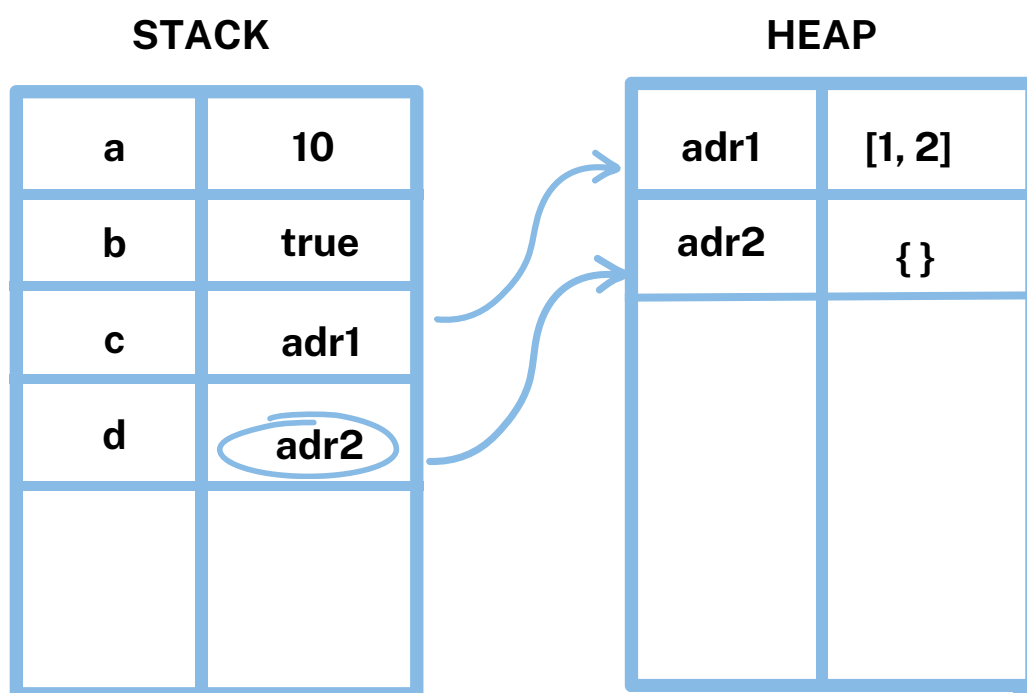
To funkcioniše tako što pretraživač koji se nalazi na slici iznad sa leve strane pošalje window objekat JS kodu i onda u okviru JS koda mi možemo da pristupimo tom objektu. On je globalni objekat koji je svuda prisutan. Kako oni komuniciraju? Tako što s leve strane imamo pretraživač koji kreira taj window objekat i pošalje JavaScript-u i na taj način mi iz konteksta JavaScript-a možemo svuda da pristupimo tom window globalnom objektu te da koristimo njegove metode i polja.

VREDNOST I REFERENCA

 [POGLEDAJ LEKCIJU](#)



Kada smo naučili šta su to objekti i nizovi koji predstavljaju složene strukture podataka u odnosu na primarne tipove kao što su string, number, boolean, undefined ili null, sada dolazimo do jednog koncepta koji je jako bitan da se nauči. Taj koncept predstavlja **razliku između vrednosti i reference**. Ovo je izuzetno važno da se savlada zato što dosta grešaka proizilazi baš iz nerazumevanja ovih konceptata. Sa druge strane, ovo je jedno od omiljenih pitanja koja se gotovo uvek postavljaju junior programerima na intervjuima.



Kod nizova i objekata kao vrednost čuvamo samo adresu koja ukazuje na deo memorije gde se stvarno nalazi taj objekat, dok kod promenljivih koje čuvaju osnovne tipove vrednosti čuvamo na samom stack-u.





JavaScript čuva promenljive u jednoj strukturi podataka koja se naziva **STACK**. Međutim, sa druge strane postoji još jedna struktura podataka koja se naziva **HEAP**.

Stack je brža struktura i služi za brže dobavljanje ili skladištenje primitivnih tipova, dok heap služi za čuvanje objekata i nizova.

Stack ima 2 dela, **naziv** i **vrednost**.



Kada kreiramo promenljivu **a** i u nju stavimo **vrednost 10**, ta vrednost se upisuje direktno na **stack**. To važi i za sve druge primitivne tipove (stringove, boolean, brojeve...). Međutim, ako želimo da kreiramo neki objekat ili niz, npr. neki **c niz** koji ima **vrednosti 1 i 2**, naziv tog niza će biti upisan na stack ali će njegova vrednost zapravo predstavljati neku adresu br. 1 na **heap-u** koja je u stvari deo radne memorije. Dakle, na toj adresi na heap-u se nalazi naš niz. Isto važi i za objekat.

METODE NIZOVA

 [POGLEDAJ LEKCIJU](#)



```
let myArray = [1, 2, 3]
```

myArray.push(4)

dobijamo 1,2,3,4

dodaje na kraj niza

myArray.pop()

dobijamo 1,2,3

uklanja parametar 4 s kraja

myArray.shift()

dobijamo 2,3

briše s početka

myArray.unshift(1)

dobijamo 1, 2, 3

dodaje na početak

myArray.indexOf(2)

dobijamo 1

vraća index nekog elementa



```
let myArray = [1, 2, 3, 2, 4]
```

myArray.lastIndexOf(2)

dobijamo 3

kada imamo ponavljanje nekog broja i želimo da saznamo index gde se taj broj koji se ponavlja proslednji put pojavio



```
myArray = [1, 2, 3]
```

```
let myArray2 = [4, 5, 6]
```

let spojenNiz = myArray.concat(myArray2)

dobijamo [1, 2, 3, 4, 5, 6]

Metoda concat spaja 2 ili više nizova u jedan novi niz i prvobitni nizovi se ne menjaju na taj način.



spojenNiz = [1, 2, 3, 4, 5, 6]

spojenNiz.slice(1, 4)

dobijamo 2, 3, 4

isekli smo deo niza



spojenNiz = [1, 2, 3, 4, 5, 6]

spojenNiz.splice(1, 4)

dobijamo 1, 6

ova metoda je isekla
te elemente iz niza i
ostavila originalni niz
bez tih elemenata



U prethodnom primeru, slice seče od 1. indexa do 4. (2, 3, 4), dok splice iz originalnog niza iseče te elemente i taj niz više ne poseduje te elemente (1, 6). Drugim rečima, splice radi brisanje elemenata, briše od indexa do indexa, dok slice izvlači te elemente i dobijamo niz tih novih elemenata.

NEW I THIS



[POGLEDAJ LEKCIJU](#)



Na koji način možemo da kreiramo objekte?



DEFINISANJEM UNUTAR VITIČASTIH ZAGRADA

```
const objekat = {  
  ime: "Milan",  
  godine: 30  
}  
console.log(objekat)
```

dobijamo {ime: "Milan", godine: 30}



PREKO FUNKCIJE

```
function kreirajKorisnika(ime, godine) {  
  return {ime: ime, godine: godine }  
}
```

```
const korisnik = kreirajKorisnika("Milan", 30)  
console.log(korisnik)
```

dobijamo {ime: "Milan", godine: 30}

```
const datum = new Date()  
console.log(datum)
```

dobijamo trenutni datum i trenutno vreme



Mi smo ovde poslali jednu konstruktorsku funkciju. Ovo je klasa **Date**, a klasa je u suštini jedan šablon i sve ono što pravimo po tom šablonu je zapravo objekat (instanca te klase). Kada kažemo `new Date` (dakle `new` pa taj šablon), mi zapravo kreiramo objekat.



Konstruktorska funkcija - funkcija koja konstruiše naš objekat i to je zapravo obična funkcija koju pišemo velikim slovom.

```
function Korisnik(ime, godine) {  
    return {ime: ime, godine: godine}  
}
```

```
const korisnik2 = new Korisnik("Ivan", 27)  
console.log(korisnik2)
```

dobijamo {ime: "Ivan", godine: 27}



Zbog čega nam je u ovom primeru bitan **new**? Kada god vidimo da se nešto poziva sa `new`, a zatim naziv klase, znamo da se zapravo od tog šablona kreira pravi objekat i da je to konstruktorska funkcija koja nam konstruiše objekat. Dakle, `new` koristimo samo kod kreiranja objekata uz pomoć konstruktorske funkcije.



This možemo zamisliti kao neki kontekst u kojem se trenutno nalazimo. `This` postoji svuda i svuda možemo pogledati šta se nalazi u okviru `this` objekta.



`This` veoma često buni programere te dolazi do puno grešaka upravo zbog nerazumevanja ovog koncepta u JavaScript-u.

```
const b = {  
  ime: "Pera",  
  godine: 30,  
  prikažiKontekst() {  
    console.log(this)  
  }  
  prikažiImeGodine() {  
    console.log(this.ime + " " + this.godine)  
  }  
}  
b.prikažiKontekst()  
b.prikažiImeGodine()
```

Pera 30

```
function prikažiNekiTekst(tekst) {  
  console.log(tekst)  
  console.log(this)  
}
```

```
prikažiNekiTekst("blabla")  
console.log(this)
```

blabla
window
window



Kada `this` koristimo u okviru objekta on prikazuje članove objekta, a kada ga koristimo van objekta on će da prikaže `window`. Neko podrazumevano ponašanje kada pozivamo funkcije i kada odradimo `this` unutar njih jeste da se taj `this` odnosi na `window` objekat.

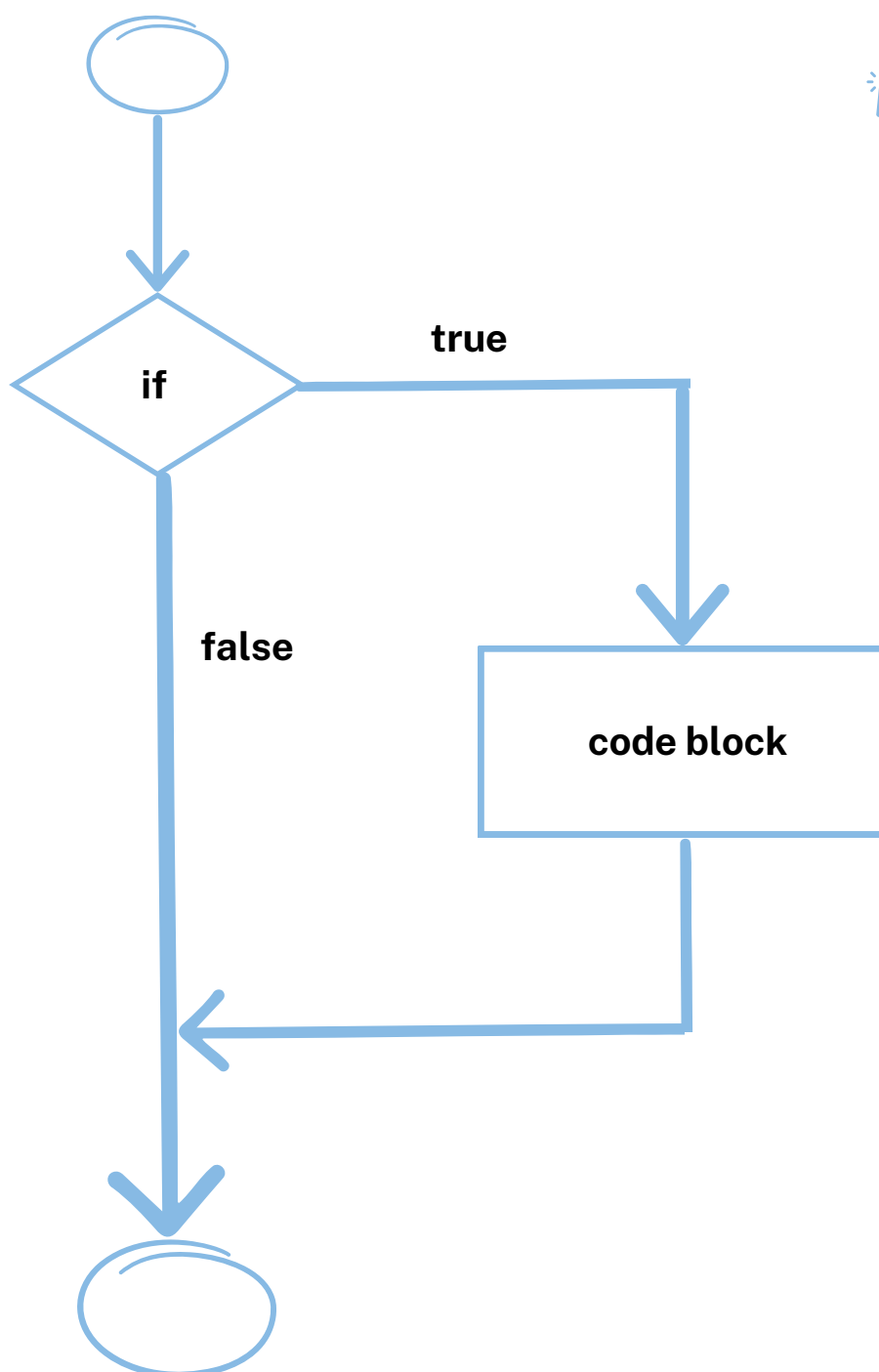
GRANANJA I PETLJE

USLOVI

-Poređenja-



[POGLEDAJ LEKCIJU](#)



Šta se ovde dešava? Polazimo od kruga, dolazimo do našeg uslova i u tom uslovu pitamo nešto, npr. da li je korisnik ulogovan ili nije. Ukoliko jeste vrednost je true, dakle ukoliko je ulogovan izvršiće se neki kod. Ukoliko nije ulogovan, ne izvršava se nikakav kod i nastavlja se dalje. U ovom primeru ukoliko je korisnik ulogovan i ode u desnu stranu true, izvršiće se neki kod pre nego što se završi program. Ukoliko nije ulogovan, neće se izvršiti ništa.

>	veće
<	manje
>=	veće ili jednako
<=	manje ili jednako
==	jednako
!=	različito
===	striktno jednako
!==	striktno različito

Bitna stvar kod poređenja jeste da nam poređenja uvek vrate true ili false. Ukoliko je neko poređenje tačno dobićemo true, ukoliko nije dobićemo false.



1 > 3 // false

2 < 5 // true

"A" < "a" // true



Možemo da poredimo i karaktere. Svaki karakter je predstavljen svojim kodom. Kada poredimo karaktere, mi zapravo poredimo njihove kodove (A₄₁, a₆₁).



Striktno i obično poređenje



Kod običnog poređenja, ukoliko npr. poredimo 100 == "100" dobićemo true, jer će JavaScript pokušati da izjednači tipove, pokušaće da prebaci string u broj. Ovo ne može da se desi kod striktnog poređenja (===) jer u ovom slučaju JavaScript gleda i kog su tipa obe strane i ne pokušava da ih izjednači po tipu. Npr. 100 === "100", JS će odmah videti da je jedno broj a drugo string, da ne može da se poredi, da to ne može nikada da bude jednako i da je false.



Isto tako kod različitog običnog i različitog striktnog poređenja, npr. 100 != "90" JavaScript će to da pretvori u 90 (100 != 90) i dobićemo true, ukoliko stavimo 100 !== 90 dobićemo true jer je stvarno različito, a kada bismo stavili 100 !== "90" opet ćemo dobiti true jer JS ovo vidi kao drugi tip i on je sigurno različit.



Ovde je najbitnija stvar da se odmah zaboravi na ovaj nestriktni način i da se uvek koristi striktni, tako da uvek kada hoćemo nešto da poredimo koristimo striktno poređenje (100 === 100 i 90 !== 100).

IF, ELSE, ELSE IF

 [POGLEDAJ LEKCIJU](#)

```
let stars = 5;
if(stars === 5){
  console.log("You are premium user!")
}
```



Kada pišemo if uslov, rezervisanu reč if, najpre u zagradi pišemo uslov, zatim otvorimo uglaste zagrade i sve što pišemo unutar uglastih zagrada će se izvršiti ako je uslov ispunjen.



Kako se definiše if? Tako što napišemo if, zatim u običnim zgradama napišemo uslov i posle toga u okviru vitičastih zagrada pišemo šta želimo da se izvrši ukoliko je uslov ispunjen.



U JavaScript-u postoji varijanta za generisanje random broja od 0 do 1, a to radimo tako što napišemo random promenljivu i u njoj dodelimo vrednost **Math.random**. Šta Math.random radi? On dodeljuje random vrednost od 0 do 1, a mi nikada ne znamo koja će vrednost biti sledeća jer kao što i sam naziv kaže, on to dodeljuje proizvoljno.

```
let stars = 5;
if(stars === 5){
  console.log("You are premium user!")
}else if(stars === 2){
  console.log("You are basic user!")
}
```

 Else if blokova može biti više, dok if blok može biti samo jedan po uslovu.

 Komentari se ne izvršavaju.

```
let stars = 5;
if(stars === 5){
  console.log("You are premium user!")
}else if(stars === 4){
  console.log("You are VIP user!")
}else{
  console.log("You are basic user!")
}
```

 Else nam služi da uhvati sve one koji nisu upali ni u jedan od prethodnih uslova.

 Else if-ova možemo da imamo koliko god želimo, to zavisi od primera, a if može da stoji sam bez ikakvog else-a ili else if-a, može da bude u kombinaciji sa else-om, a može da bude i u kombinaciji samo sa else if-om.

AND LOGIČKI OPERATOR

 [POGLEDAJ LEKCIJU](#)



Uzmimo za primer situaciju u kojoj nas neko pošalje u prodavnicu i kaže kupi jaja i mleko. Zadatak će biti izvršen ukoliko ispunimo obe stvari, ukoliko kupimo i jaja i mleko. Sa druge strane, moguća je i druga varijanta, gde nam neko kaže da kupimo jaja ili mleko, tako da u toj varijanti mi možemo da kupimo ili jedno ili drugo. A neće biti greška i ako kupimo obe stvari. Treća varijanta bi bila kada bi nam neko rekao da ne kupimo ništa. Ove tri stvari postoje i u programiranju i njih označavamo upotrebom **AND**, **OR** ili **NOT** operacija.

And bi bila varijanta gde je potrebno da se ispune oba uslova, **or** bi bila varijanta gde je potrebno da se ispuni ili jedan ili drugi uslov ili čak oba, a **not** bi bila neka negacija, dakle suprotno od nečeg što mi pošaljemo.



&& - oznaka za AND



AND je ispunjen ukoliko su obe strane tačne

```
const tekst = "tekst"  
if(tekst.length > 3 && tekst.length < 10){  
    console.log("Tekst je duži od 3")  
}
```



Prompt - funkcija koja skuplja informacije od korisnika



Modal - okvir (dialog box) koji se prikazuje na vrhu trenutne stranice

OR LOGIČKI OPERATOR

 [POGLEDAJ LEKCIJU](#)



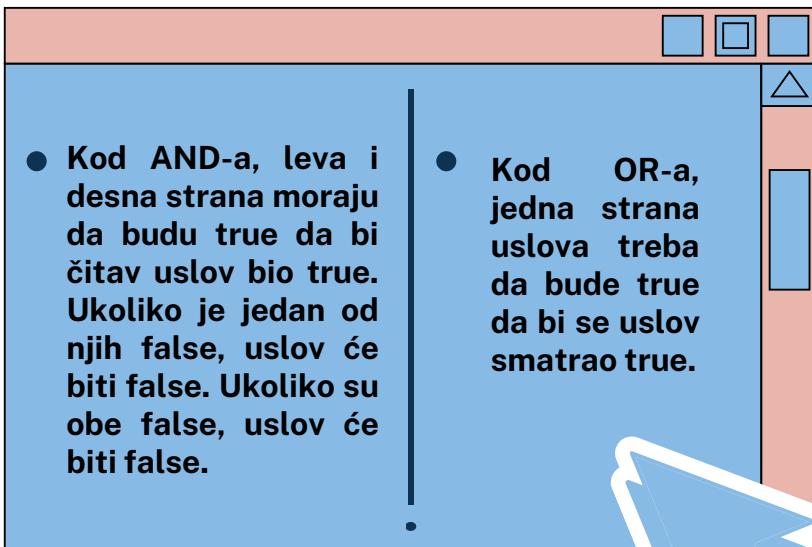
Operator “ili”



Ovaj operator se piše sa dve uspravne linije ||



Koja je razlika između operatora OR i operatora AND?



- Kod AND-a, leva i desna strana moraju da budu true da bi čitav uslov bio true. Ukoliko je jedan od njih false, uslov će biti false. Ukoliko su obe false, uslov će biti false.
- Kod OR-a, jedna strana uslova treba da bude true da bi se uslov smatrao true.

true		true	dobijamo	true
false		true	dobijamo	true
true		false	dobijamo	true
false		false	dobijamo	false

0 - 7 DECA BESPLATNO

8 - 18 ŠKOLARCI 10e

19 - 65 ODRASLI 20e

65+ PENZIONERI BESPLATNO

```
const godine = prompt("Unesite godine")
if((godine > 0 && godine < 8) || godine > 64){
  console.log("Besplatan ulaz!")
}else if(godine >=8 && godine < 19){
  console.log("10e")
}else if(godine >=19 && godine < 65){
  console.log("20e")
}else{
  console.log("Greška prilikom unosa!")
}
```

NOT LOGIČKI OPERATOR



[POGLEDAJ LEKCIJU](#)



Negacija se primenjuje tako što stavimo uzvičnik ispred neke vrednosti koju želimo da negiramo.

!true	dobijamo	false	
!1	dobijamo	false	jer se 1 tretira kao truthy vrednost
!30	dobijamo	false	
!0	dobijamo	true	jer se 0 tretira kao falsy vrednost



Negacija je zapravo pretvaranje u suprotnu vrednost.



Bilo koju vrednost možemo da negiramo, a možemo da je negiramo čak i više puta.

!true pretvaramo u **false**
!!true dobijamo **true** nazad



Mi možemo negaciju da odradimo više puta i to je zapravo više puta primenjena negacija, ali kada odradimo dva puta negaciju mi zapravo vraćamo u prvobitno stanje.



Negaciju možemo da koristimo i u nekoj kombinaciji drugih logičkih operacija.

(2 === 2) dobijamo **true**
!(2 === 2) dobijamo **false**

"a" === "b"	false
!("a" === "b")	true

!null	true
!undefined	true



Savet je da ne koristimo toliko često negaciju i da uvek pišemo uslov koji nema negaciju zato što je mnogo jednostavnije i mnogo je razumljivije da se to pročita kasnije. Negaciju koristimo kada baš moramo, dugački uslovi se izbegavaju i obično se grupišu u neke manje.

UGNJEŽDENI IF

 [POGLEDAJ LEKCIJU](#)



To je zapravo if unutar drugog if-a.

primer:

```
//korisničko ime mora imati više od 5 karaktera  
//korisničko ime ne sme sadržati razmak
```

```
const korisničkoIme = "Dušan99"  
if(korisničkoIme.length > 5){  
  if(korisničkoIme.indexOf(" ") === -1){  
    console.log("Ispravno korisničko ime!")  
  }else{  
    console.log("Neispravno korisničko ime (razmak)!")  
  }  
}else{  
  console.log("Neispravno korisničko ime (dužina)!")  
}
```

TERNARNI OPERATOR

 [POGLEDAJ LEKCIJU](#)



Jednostavniji način za pisanje if bloka.



Koristi se u radu veoma često.

```
const korisnikUlogovan = true
if(korisnikUlogovan){
  console.log("Korisnik ulogovan")
}
```

Kako bismo ovaj primer skratili i napisali u jednoj liniji?



Ovo je skraćeni način za pisanje if-a i na ovaj način možemo da izvršimo samo jednu liniju.
uslov ? kod : kod



```
const korisnikUlogovan = true
if(korisnikUlogovan) console.log("Korisnik ulogovan")
```

```
const korisnikUlogovan = false
if(korisnikUlogovan){
  console.log("Korisnik ulogovan")
}else{
  console.log("Korisnik nije ulogovan")
}
```



Sve ovo možemo da zamenimo tako što napišemo ternarni operator:

```
korisnikUlogovan ? console.log("Korisnik ulogovan") : console.log("Korisnik nije ulogovan")
```



Šta smo mi zapravo ovde uradili? Napisali smo if else skraćeno, tako što smo odmah napisali uslov i upitnik (?). To je zapravo taj ternarni operator. Zatim smo napisali prvo deo ukoliko je uslov ispunjen pa nakon toga deo ukoliko uslov nije ispunjen. Za ovo može da se koristi ternarni operator samo ukoliko je jedna linija u pitanju. Ukoliko želimo da izvršimo više linija moramo da napravimo neku funkciju i zatim tu funkciju pozvati.



Ispisivanje poruke u zavisnosti od nekog uslova

```
const korisnikUlogovan = true
const porukaZalispis = korisnikUlogovan? "Dobrodošli": "Morate se ulogovati!"
```

```
console.log(porukaZalispis)
```

dobijamo Dobrodošli



Osim što ternarni operator može da izvršava neke operacije u odnosu na if ili else (znači da li je uslov ispunjen ili nije), on takođe može da služi da dodeljujemo vrednost nekoj promenljivoj u zavisnosti od nekog uslova.



Da li treba da koristimo ternarni operator?

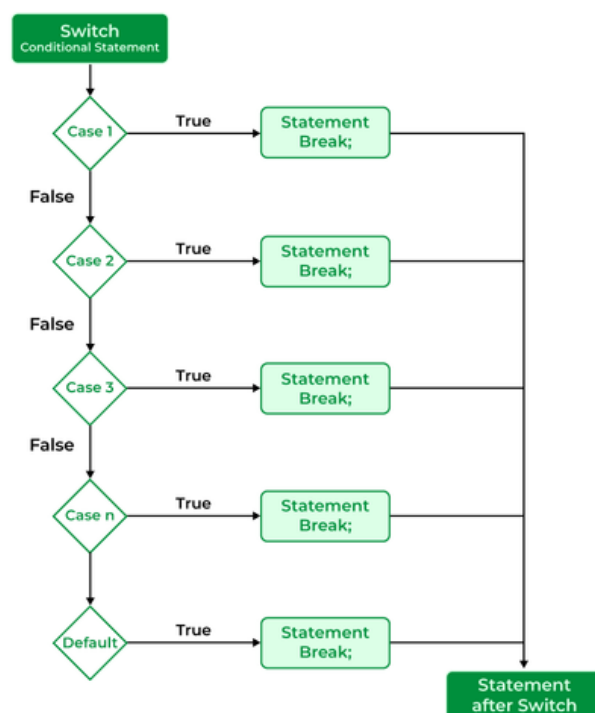


Savet je da se ternarni operator koristi kada je neko dodeljivanje vrednosti u pitanju. Za izvršavanje if else bloka je uvek mnogo bolje praviti if else, jer nekad treba ići na što čitljiviju varijantu umesto na što kraću jer posle nas treba da dođe neki drugi programer i da to pročita, tako da uvek treba da razmišljamo o dobrom imenovanju varijabli i o pisanju čitkog koda.

SWITCH

 [POGLEDAJ LEKCIJU](#)

-  Switch je konstrukcija koja omogućava selektivno izvršavanje koda na temelju vrednosti neke promjenljive, poput varijable ili izraza.
-  Koristi se u svim programskim jezicima
-  Switch se može predstaviti tako što koristimo if i else if.
-  Nekada se naziva switch, nekada switch case, oba naziva su validna i oba naziva ukazuju na istu stvar.
-  **Break;** - služi da prekine izvršavanje switch-a, jer ukoliko bismo upali u prvi case posle toga bi se svi case-ovi krenuli izvršavati.



PETLJE

-For petlja-

 [POGLEDAJ LEKCIJU](#)

 Bitno je da ponavljamo neki deo koda sve dok se neki uslov ne zadovolji.

kreni od 1	stani na 10	dodaj 1 svaki put
<pre>for(let i = 1; i < 10; i ++){ console.log (i); }</pre>		

 Svaka for petlja se definiše sa for i u okviru nje postoje 3 dela i telo te petlje (u ovom slučaju console.log(i)).

 Prvi deo je **definisanje brojača**. Dakle, mi tu definišemo jednu promenljivu koja se naziva **i** te joj dodelimo vrednost 1.

Drugi deo for petlje je **uslov**.

Treći deo je najčešće neki **inkrement ili dekrement**, tj. dodavanje ili oduzimanje vrednosti na taj brojač.

```
for(let i = 10; i >= 0; 1--){  
  console.log(i);
```

 prvi deo - definicija promenljive
drugi deo - neki logički uslov koji vrati true ili false
treći deo - inkrement ili dekrement, tj. povećanje ili smanjivanje vrednosti

for (init; uslov; inkrement)



3 dela svake petlje



inicijalna vrednost

i ++	skraćenica za	i = i + 1
i += 2	skraćenica za	i = i + 2

i ++	koristimo kada želimo da povećamo vrednost za 1
i --	koristimo kada želimo da smanjimo vrednost za 1
i = i + 2 ili i += 2	koristimo kada želimo da menjamo vrednost za 2

FOR SA NIZOVIMA

 [POGLEDAJ LEKCIJU](#)



Koristimo for petlju za prolazak kroz niz



```
let rasporedSedenja = ["Jelena", "Milan", "Dragan"]
```

```
// i= 0  
// i= 1  
// i= 2  
// i= 3 - prekid
```

Jelena
Milan
Dragan

```
for(let i = 0; i < rasporedSedenja.length; i++){  
  console.log(rasporedSedenja[i])  
}
```



```
let rasporedSedenja = ["jelena", "Milan", "Dragan"]
```

```
// i= 0  
// i= 1  
// i= 2  
// i= 3 - prekid
```

Dragan
Milan
Jelena

```
for(let i = rasporedSedenja.length - 1; i >= 0; i--){  
  console.log(rasporedSedenja[i])  
}
```

UGNJEŽDENI FOR



[POGLEDAJ LEKCIJU](#)



Matrica - niz nizova

1	2	3
4	5	6
7	8	9



Matrica je višedimenzionalni niz. Višedimenzionalni niz znači da nije samo jedan niz, već da se niz sastoji od drugih nizova.



Za prikazivanje ovakvih nizova ne možemo da koristimo samo jednu for petlju jer se jedna for petlja koristi samo u jednoj dimenziji, kada imamo jedan niz.



Ovde su nam potrebne dve for petlje.

```
let rasporedSedenja = [  
  ["Jelena", "Milan", "Dragan"],  
  ["Jovan", "Ana", "Zoran"],  
  ["Maja", "Petar", "Ivan"],  
  ["Jova", "Danilo", "Ogi"]  
]
```

```
for(let i = 0; i < rasporedSedenja.length; i++){  
  console.log(rasporedSedenja[i])  
}
```



Za ove for petlje najčešće se koriste slova **i**, **j**, **k**.

```
console.log("raspored sedenja", rasporedSedenja)

for(let i = 0; i < rasporedSedenja.length; i++){
  console.log("red: " + i)
  for(let j = 0; j < rasporedSedenja[i].length; j++){
    console.log("kolona: " + j)
    console.log[rasporedSedenja[i] [j]]
  }
}
```

WHILE PETLJA

 [POGLEDAJ LEKCIJU](#)

 Namena while petlje je da izvrši određen kod sve dok je uslov ispunjen.

```
while(true){  
  console.log("Zdravo!")  
}
```



 Ukoliko ostavimo ovde ovaj true dolazimo do beskonačne petlje koja će opteretiti naš računar. Beskonačna petlja je veliki problem kada se koristi while jer u nju možemo vrlo lako upasti ukoliko ne postavimo uslov kako treba.

```
let brojač = 1  
while(brojač <= 5){  
  console.log("Zdravo!")  
  brojač ++  
}  
  
console.log("Kraj petlje!")
```

Kada se koristi while i čemu on služi?

 While je dobar kada su u pitanju neke simulacije, kada ne znamo kad treba da završimo petlju. For koristimo kada znamo, kada npr. prolazimo kroz neki niz i znamo da on ima 10 elemenata, želimo da prođemo kroz taj niz i koristimo for koji prolazi 10 puta. Međutim, kada ne znamo do kada će uslov biti ispunjen koristimo while, npr. u nekim video igricama gde čekamo da se završi igra.

BREAK I CONTINUE



[POGLEDAJ LEKCIJU](#)



Dva koncepta koja nam služe da prekinemo petlju pre vremena ili da nastavimo sa izvršavanjem te petlje i pre nego što se sav kod unutar petlje izvrši.



Break služi da prekinemo izvršavanje određenje funkcije.

```
for(let brojač = 1; brojač <= 10; brojač++){  
  console.log("Zdravo!")  
  if(brojač === 5){  
    break  
  }  
}
```

```
console.log("Kraj petlje!")
```

```
while(brojač <= 10){  
  console.log("Zdravo!")  
  if(brojač === 5){  
    break  
  }  
  brojač++  
}
```

```
console.log("Kraj petlje!")
```



Continue nam omogućava da pre nego što se izvrše sve ove linije unutar bloka koda izvršavanja možemo da kažemo "Stani tu, pređi u sledeću iteraciju petlje, nemoj da izvršiš ono što je ispod".

```
for(let brojač = 1; brojač <= 10; brojač++){
  if(brojač === 5) {
    continue
  }
}
```

```
console.log("Kraj petlje!")
```



Continue je takođe dobar kada želimo da određenu iteraciju preskočimo unutar for-a ili da izvršimo samo 1 deo ovog bloka i da nastavimo dalje da iteriramo kroz petlju.

**Primer
beskonačne petlje**



```
let brojač = 1
```

```
while(brojač <= 10){
  console.log("Iteracija" + brojač)
  if(brojač === 9){
    continue
  }
  brojač++
}
console.log("Kraj petlje!")
```

**Evo kako ovo
treba da rešimo**



```
let brojač = 1
```

```
while(brojač <= 10){
  brojač++
  if(brojač === 9){
    continue
  }
  console.log("Iteracija" + brojač)
}
console.log("Kraj petlje!")
```

REKURZIJA



[POGLEDAJ LEKCIJU](#)

```
function ispišiZdravo(){  
  console.log("Zdravo")  
  ispišiZdravo()  
}
```

ispišiZdravo()



JavaScript nam ovde kaže da smo dostigli maksimum izvršavanja. Mi smo ovde ušli u endess loop (beskonačnu petlju), funkciju ispišiZdravo smo pozvali u funkciji ispišiZdravo i onda smo opet ušli u tu funkciju pa je ta funkcija opet pozvala ispišiZdravo itd. Vrteli smo se u toj funkciji i nikada nismo izašli iz nje. Slično je kao i kada u while petlji imamo uslov koji se nikada ne ispuni i stalno se vrti petlja.



Kada je korisna rekurzija ?



Rekurzija se koristi u kodu kada je problem prirodno podeljen na manje slične podprobleme koji se mogu rešiti istim algoritmom, međutim moramo voditi računa da napravimo uslov iz kojeg se izlazi iz te rekurzije. Ne možemo pustiti da funkcija poziva samu sebe neograničen broj puta. Umesto toga, možemo iskoristiti taj koncept za nešto što nam treba tako da ne prekršimo taj zakon završetka rekurzivne funkcije i da iskoristimo da nešto odradi za nas.

Kada se može iskoristiti rekurzivna funkcija?

```
let suma = 0
function sumaBrojeva(broj){
  suma = broj
  sumaBrojeva(broj + 1)
}
```

sumaBrojeva(0)

dobijamo endless loop



Npr. želimo da izračunamo zbir prvih 10 brojeva. Na ovaj način računamo sumu prvih 10 brojeva upotrebom rekurzivne funkcije. Funkcija poziva samu sebe i uvek dodaje broj. Problem sa ovom implementacijom može biti ukoliko nemamo izlaz (stalno će da se dodaje broj i dobićemo endless loop) i zato je bitno da ovde damo uslov kada se izlazi iz ove rekurzije.

```
let suma = 0
function sumaBrojeva(broj){
  if(broj === 10){
    return;
  }
  suma = suma + broj
  console.log(suma)
  sumaBrojeva(broj + 1)
}
```

sumaBrojeva(0)



Bitno je znati da je rekurzivna funkcija funkcija koja poziva samu sebe određen broj puta te je vrlo važno da se tu zada uslov, međutim neće se često vidati jer usled pogrešno definisanog uslova može doći do beskonačne petlje. Svaka rekurzivna funkcija se može predstaviti for petljom što je bolji izbor i treba izbegavati rekurzivne funkcije kada god je to moguće.

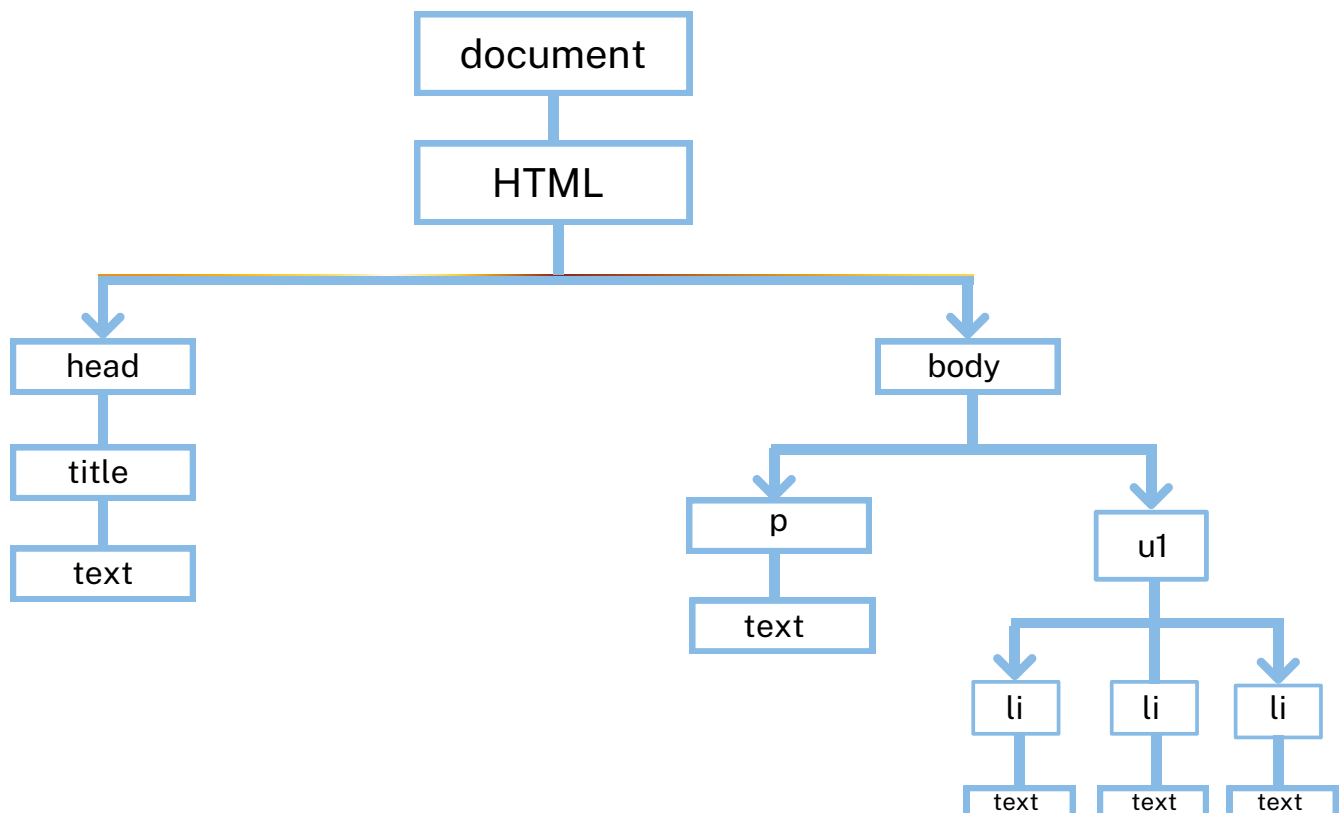
JavaScript DOM

UVOD

 [POGLEDAJ LEKCIJU](#)

 **Document Object Model** (DOM) predstavlja reprezentaciju web stranice u JavaScript-u.

 Prilikom učitavanja bilo koje web stranice u pretraživaču, pretraživač stvori internu reprezentaciju koju naziva **DOM stablo**. DOM stablo je zapravo HTML struktura pretvorena u JS objekat. JS DOM je u stvari jedan interfejs koji koristimo da promenimo elemente na web sajtu, da dodamo neke događaje kao što su npr. ponašanje na klik nekog elementa ili reakcija na promenu teksta u nekom inputu. Svu tu interaktivnost mi dodajemo upotrebom JS DOM-a.





Ova struktura se sastoji iz **čvorova** i svaki čvor može sadržati svoje **podčvorove** ili **nadčvorove**. Nadčvor bi bio **parent čvor**, tj. roditeljski čvor, dok bi podčvor predstavljao **child** tj. dete čvor kako to nazivamo u programerskom svetu. Čvorovi ili **nodovi** bi bili elementi našeg HTML fajla. Znamo da svaki HTML fajl počinje sa HTML elementom, i da se unutar HTML elementa nalaze drugi elementi kao što su head, body... U okviru head-a znamo da imamo i title i u okviru tog title-a se nalazi neki tekst koji je takođe predstavljen čvorom.

Sa druge strane, u body-u pišemo sve ono što je vidljivi deo web stranice i to su tagovi kao što je p tag, ul tag, može se tu naći i image tag i paragraf tagovi kao što su h1, h2, h3. Sve to može da se predstavi grafički, upotrebom stabla. U stablu svaki čvor može sadržati više podčvorova, ali svak čvor ima samo jedan roditeljski čvor.

SELEKTORI

 [POGLEDAJ LEKCIJU](#)

 Ovde počinjemo sa selektovanjem čvorova iz DOM stabla. Zašto je važno da selektujemo? Da bismo primenili neku promenu u HTML-u, mi prvo moramo da znamo da uhvatimo određeni element u HTML fajlu. Kako to radimo? Tako što postoje ugrađene metode u okviru našeg document objekta. Postoje ugrađene metode koje za nas vraćaju određeni element na osnovu uslova koje mi zadamo.

 **Selektor metode** - metode koje nam služe da selektujemo neke elemente.

 **QuerySelector** - univerzalni selektor koji je najčešće u upotrebi. On uvek vraća jedan element koji upada u zadati uslov. U slučaju da mu pošaljemo klasu, on počne da traži tu klasu i čim nađe prvi element sa tom klasom on će njega da vrati. Za razliku od nekih selektora koji su vraćali niz, ovaj selektor uvek vraća jedan element i dobar je za ciljano selektovanje jednog elementa.

 Za selektovanje elementa po id-u kada koristimo querySelector koristimo **#id**.

 Ukoliko želimo da pronađemo element po npr. img tagu dovoljno je da samo napišemo img.

 Ukoliko želimo da pronađemo element po klasi, u okviru stringa ćemo napisati tačku pa naziv te klase (.green-label).

 **QuerySelectorAll** - vraća kolekciju, tj. više elemenata koji zadovoljavaju određeni uslov.

PROMENA STILOVA



[POGLEDAJ LEKCIJU](#)



Kako menjamo stilove nad elementom?



Elementi su objekti, tako da ukoliko želimo da pristupamo nekim njegovim poljima, pristupamo tako što napišemo tačku pa nakon toga style (jer mi menjamo neki style atribut unutar njega).



Za razliku od CSS property-a gde su imena koja imaju dve reči u sebi bila odvojena crticom, u JavaScript-u se dve reči odvajaju tako što se prva reč napiše malim, a druga velikim slovom (border-top / borderTop). To je tzv. **camel case** i on nam služi kada želimo da prikazemo više reči, dakle te reči odvajamo velikim slovom.

odvajamoVelikimSlovom - camel case

JS

odvajamo_velikim_slovom - snake case

PYTHON

odvajamo-velikim-slovom - dash case

CSS



U CSS-u se koristi odvajanje srednjom crtom ukoliko želimo da prikazemo više reči, u JavaScript-u se koristi camel case, dok se u Python-u dosta koristi snake case.

PROMENA TEKSTA

 [POGLEDAJ LEKCIJU](#)



Append - metoda za dodavanje novog elementa



innerHTML - sa ovom metodom dobijamo naš ceo HTML sa svim tagovima.



innerText - sa ovom metodom dobijamo samo sadržaj teksta.



textContent - sa ovom metodom dobijamo isto kao i sa metodom innerHTML, ali bez tagova (dobijamo samo tekst unutar tagova).



\n - način na koji se govori stringu da se posle toga nešto prebacuje u novi red. Piše se u okviru stringa.

PROMENA ATRIBUTA I BRISANJE

 [POGLEDAJ LEKCIJU](#)



Svaki HTML element ima svoje atribute (id, atribut a taga je href, kod image taga atribut je src, kod inputa je type, name, value, kod labela je class...).



Add - metoda za dodavanje klase (`element.classList.add(nazivKlase)`)



Contains - metoda koja nam vraća true ako element sadrzi klasu ili false ako ne sadrži (`element.classList.contains(nazivKlase)`)



Remove - metoda za uklanjanje klase na elementu (`element.classList.remove(nazivKlase)`)



SetAttribute - metoda pomoću koje možemo da menjamo atribute, dodajemo nove, menjamo stare...

PROLAZAK KROZ DOM

-Traversiranje-

 [POGLEDAJ LEKCIJU](#)



Svaki element može da ima jedan parent element.



Svaki element može da ima više child elemenata, tj. više dece.



ParentElement - ukoliko nad elementom primenimo parentElement mi hvatamo njegovog roditelja.



NextElementSibling - ukoliko nad elementom primenimo nextElementSibling mi hvatamo sledeći element koji je na istom nivou.

DOGAĐAJI

-Event Listeners-

 [POGLEDAJ LEKCIJU](#)



Događaj je sve ono što korisnik uradi na stranici, dakle tu gde počinje interaktivnost- korisnik klikne na neku sliku, klikne na neki tekst, promeni neki tekst na stranici, prevuče kursor miša preko slike ili promeni veličinu ekrana. To su sve događaji na koje mi možemo da reagujemo.



Događaje možemo dodati na svakom elementu.



Obrađivači događaja:

1

EVENT LISTENER - onaj koji osluškuje (on click)

2

EVENT HANDLER - onaj koji rukuje događajem (alert)

PREVENT DEFAULT



[POGLEDAJ LEKCIJU](#)



Šta su podrazumevani događaji?



Npr. kod a tagova znamo da kada kliknemo na taj tag on nas automatski vodi na link koji smo mu zadali. To je običan klik event koji je u pozadini integrisan u tom tagu i kaže “svaki put kada se klikne na taj tekst ti mene preusmeri na ovaj portal”. Ukoliko želimo da kažemo “od sada kada neko klikne na taj a tag, nemoj da me vodiš na tu stranicu, nego uradi nešto drugo, mi moramo da imamo mehanizam koji će reći “prestani da koristiš to podrazumevano ponašanje, ja želim da ti dam drugo ponašanje koje ćeš raditi na klik”.

ASINHRONI KOD

ASINHRONOST I EVENT LOOP

 [POGLEDAJ LEKCIJU](#)



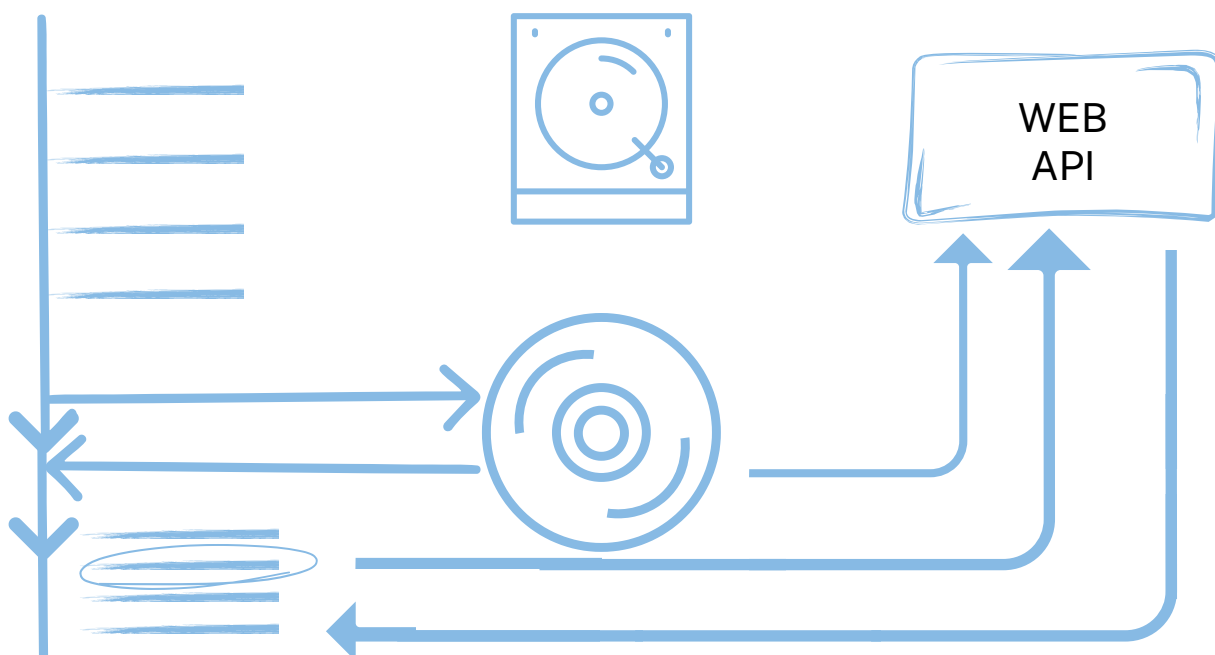
Do sada smo radili sa sinhronim kodom, što znači da se svaka linija izvršavala redom. Počevši od prve linije, JavaScript uzima fajl i kreće da ga izvršava liniju po liniju. To je redosled **top to bottom** (odozgo do dole) kojim se kod izvršava.



JavaScript ima samo **jedan thread** izvršavanja (js je single-threaded), a to znači da ne može paralelno da izvršava više stvari odjednom i da ima prostora samo za jedan prolaz, tako da tu mogu naići određeni problemi u slučaju da moramo da čekamo na izvršavanje neke operacije.



Kada god imamo potrebu da se neki kod malo odloži ili da se čeka na određeni kod da bi se on izvršio, mi onda imamo problem u JavaScript-u i moramo da koristimo asinhroni način izvršavanja.





Ovde postoji jedna nit izvršavanja koja izvršava sav kod i ukoliko bi došao neki proces kao što je npr. čitanje nekog fajla sa diska gde treba da se pristupi nekom hard disku ili da se pročita neka druga ulazna operacija, takve stvari bi mogle da uspire i blokiraju tu nit i u tom slučaju bi aplikacija bila zamrznuta zato što bi se čekalo izvršavanje nekog čitanja pa bi se tek onda nastavio dalje kod. S obzirom da to predstavlja veliki problem u JavaScript-u, mi moramo da vodimo računa o tome. S obzirom da se JavaScript izvršava u pretraživaču, pretraživač nam nudi neke opcije koje rešavaju ovo.



Web API je pomagač u ovoj situaciji. Kada bismo krenuli da izvršavamo kod i kada bismo došli do trenutka kada treba da čitamo sa hard diska ili da pozivamo neki backend servis koji treba da šalje poziv određenom backend serveru, mi u tom slučaju kontaktiramo upravo web API i kažemo mu da pročita za nas iz određenog fajla, pošalje zahtev ka web-u i dovuče određenu informaciju sa servera. Dakle, sve te stvari koje moramo da čekamo ili stvari koje odlažu izvršavanje su zapravo asinhronne. Npr. mi želimo kada korisnik klikne na neko dugme da se tek posle 2 sekunde nešto izvrši. Tada mi web API-u treba da kažemo da broji 2 sekunde pa kada te 2 sekunde prođu da nam kaže da izvršimo taj kod. To je način na koji se u JavaScript-u borimo sa tim operacijama koje traju duže ili koje treba da se izvrše tek posle nekog vremena kako ne bismo korisniku zablokirali aplikaciju. Npr. `addEventListener` je funkcija sa odloženim izvršavanjem.



Request (zahtev) ka serveru predstavlja kontaktiranje servera koji nam dobavlja neke informacije (npr. ako čekamo neke rezultate da se objave na sajtu, kada neko objavi te rezultate to će biti smešteno na jednom serveru a svi korisnici će onda sa svojih računara kroz neke frontend aplikacije da kontaktiraju taj server i da pošalju request i da kažu “daj mi tu rang listu” i posle će da prikažu tu rang listu na web sajtu).

`console.log("Šaljem request ka serveru")`

`console.log("Odgovor od servera")` dobijamo

`console.log("Završio sam sa radom")`

Šaljem request ka serveru

Odgovor od servera

Završio sam sa radom



Ovaj kod se izvršava sinhrono, ide linija po linija, ne čekamo ništa, sve se izvršava redom.



SetTimeout - odloženo izvršavanje. Ova funkcija ima 2 parametra, **handler** i **timeout**. Handler je naša callback funkcija koju želimo da izvršimo kada prođe određeno vreme. Timeout se izvršava u milisekundama (1000 milisekundi je 1 sekunda).

```
console.log("Šaljem request ka serveru")
setTimeout(()=>{
  console.log("Odgovor od servera")
}, 2000)

console.log("Završio sam sa radom")
```

dobijamo

Šaljem request ka serveru
Završio sam sa radom
Odgovor od servera
(pojavljuje se nakon 2 sekunde)



Da nismo iskoristili ovaj setTimeout i da imamo neku operaciju koja dugo traje te da smo je odmah izvršili, mi bismo blokirali aplikaciju, aplikacija ne bi mogla da nastavi dalje sa radom i sve bi bilo zamrznuto. Na ovaj način, mi smo preko setTimeout-a koji je deo web API-a rekli "zakaži nam ovo za 2 sekunde" i mi nastavljamo dalje sa radom. Dakle, asinhronost se koristi kada god imamo neke dugotrajne operacije koje čekamo, poput čitanja, pisanja, obrade slike, kontaktiranja servera, kada želimo da dodamo neki event...

```
setTimeout(()=>{
  console.log("unutra")
}, 4000)
```

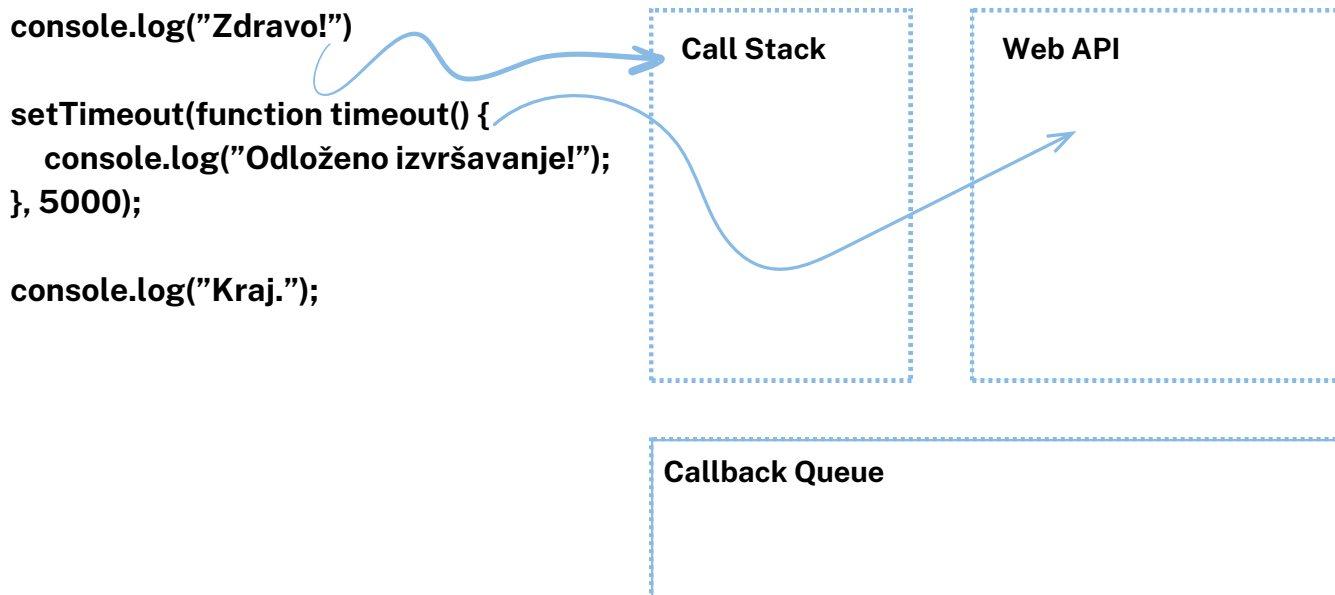
ovde smo setTimeout-u dali callback funkciju (prvi parametar)

```
const button = document.querySelector("button")
button.addEventListener("click", ()=>{
  console.log("kliknuto")
})
console.log("napolju")
```

posle toga smo i addEventListener-u dali callback funkciju, ali je ovde ta callback funkcija drugi parametar

dobijamo

napolju
unutra (nakon 4 sekunde)
kliknuto (tek kada kliknemo na to dugme)



- Call Stack -** Ovde se izvršava sinhroni kod. Kada naidje asinhroni deo za izvršavanje on se stavi u web API i zatim se izvršava (posle 5 sekundi u ovom slučaju).
- SetTimeout -** Ono što JavaScript izvršava. Sve što se izvršava sinhrono u JavaScript-u se nalazi u Call Stack-u.
- “Kraj” -** Takođe sinhrono izvršavanje i nalazi se na Call Stack-u.
- Callback Queue -** Kada prođe 5 sekundi web API kontaktira JavaScript i kaže mu da je prošlo 5 sekundi. Na koji način ga kontaktira? Tako što taj deo koji treba da se izvrši nakon 5 sekundi pošalje u Callback Queue, to je red izvršavanja. Ovaj čitav koncept se naziva Event Loop.

Kako funkcioniše Event Loop?



Imamo Call Stack, Web Api i Callback Queue. Imamo sinhrono izvršavanje i asinhrono izvršavanje. Sinhrono se izvršava u Call Stack-u. Kada se nađe na neki asinhroni deo koda pošalje se Web Api-u i Web Api onda vodi računa o tome. Kada prođe određeno vreme ili se izvrši taj deo asinhronog koda na koji se čekalo, ostatak se pošalje u Callback Queue (red) i JavaScript posle iz tog reda uzima i izvršava ga u Call Stack-u.

```
setTimeout(()=>{  
  console.log("1")  
  setTimeout(()=>{  
    console.log("2")  
    setTimeout(()=>{  
      console.log("3")  
    }, 3000)  
  }, 3000)  
, 3000)
```

Callback hell se najlakše
prepozna po svom
piramidalnom obliku



Callback hell - kada imamo mnogo callback-a i ne možemo da razumemo više šta se u tom kodu radi zato što se sve te funkcije izvršavaju asinhrono te mi ne znamo kada će se koja izvršiti, sve postaje nejasno.

PROMISI

 [POGLEDAJ LEKCIJU](#)



Promisi dolaze kao rešenje na problem koji imamo sa callback-ovima. Da bismo izbegli tu konfuziju sa previše callback-a dolazimo do promisa. To je jedan omotač oko asinhronog koda koji nam služi da imamo bolju kontrolu nad tim tokom izvršavanja.

```
const promise = new Promise((resolve, reject) => {  
  const sum = 1 + 1;  
  if(sum === 2) {  
    resolve("Uspešno")  
  } else {  
    reject("Error")  
  }  
})
```

U ovom primeru se ništa asinhrono ne dešava, ovde samo učimo kako se kreira promis



```
promise  
  .then((message) => {  
    console.log(message)  
  }).catch((errorMessage) => {  
    console.error(errorMessage)  
  })
```



Sa **then** očekujemo rezultat.



Sa **catch** hvatamo grešku.



U JavaScript-u postoji klasa koja se zove **Promise**. Mi kreiramo objekat te klase i smestamo ga u promenljivu. Ta naša promise klasa prihvata jedan atribut i to je funkcija. Ona prihvata funkciju koja ima dva parametra, **resolve parametar** i **reject parametar**. Zatim kreće telo funkcije te definišemo šta se asinhrono izvršava. Promis može da bude **uspešan** i u tom slučaju ćemo dobiti **resolve izvršavanje**, a može da bude i **neuspešan** i u tom slučaju ćemo dobiti **reject izvršavanje**.

```
function odlozenolzvršavanje() {  
  return new Promise ((resolve, reject) => {  
    setTimeout(() => {  
      resolve("Zdravo")  
    }, 2000)  
  })  
}
```

```
console.log("prvi")  
odlozenolzvršavanje().then(message) => {  
  console.log(message)  
})  
console.log("drugi")
```

U ovom primeru smo pretvorili jednu asinhronu funkciju u promis



Ovde imamo funkciju koja se naziva **odloženolzvršavanje** i koja nam vraća objekat promisa koji u sebi izvršava `setTimeout`. Ovde gledamo i kako se barata sa funkcijama koje vrata promis i u promisu asinhroni kod. Promis je tu da bi kod izgledao što čitljivije i da bismo imali utisak da se on izvršava sinhrono, iako se on izvršava asinhrono.

Pretvaranje callback hell-a u promis

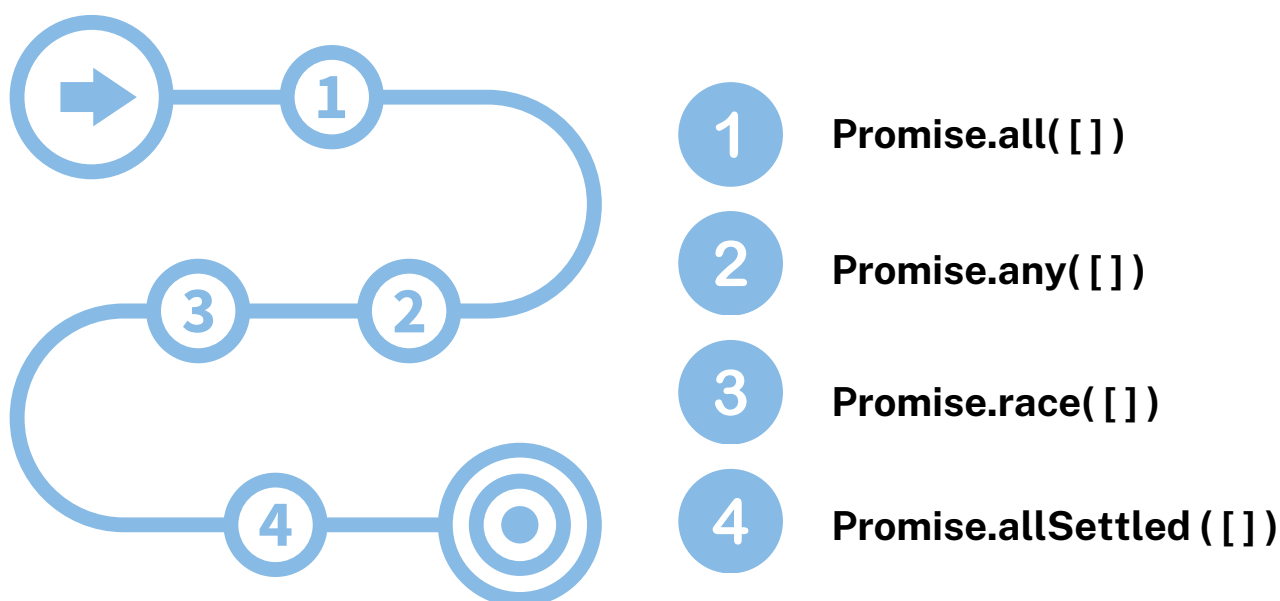
```
function odloženolzvršavanje(timeout) {  
  return new Promise ((resolve, reject) => {  
    setTimeout (() => {  
      resolve()  
    }, timeout)  
  })  
}
```

```
setTimeout (() => {  
  console.log("Ćao");  
  setTimeout(() => {  
    console.log("Ćao2");  
    setTimeout(() => {  
      console.log("Ćao3");  
    }, 3000);  
  }, 3000);  
}, 3000);
```

```
odloženolzvršavanje(3000)  
  .then (() => {  
    console.log("Ćao1")  
    return odloženolzvršavanje(3000)  
  }).then(() => {  
    console.log("Ćao2")  
    return odloženolzvršavanje(3000)  
  }).then(() => {  
    console.log("Ćao3")  
    console.log("KRAJ")  
  })
```

VARIJACIJE PROMISA

 [POGLEDAJ LEKCIJU](#)



Svaka od ovih metoda promisa prihvata listu promisa. Oni nam služe kada npr. želimo da obradimo više zahteva ka serveru, želimo da dobavimo temperaturu, vlažnost vazduha i brzinu vetra za Novi Sad. Ranije u primerima smo radili sa then, prvo smo dobili temperaturu pa kada se to izvrši tek onda se šalje zahtev za vlažnost, brzinu vetra itd. Dakle, izvršavali smo sinhorno. Međutim, desiće se da želimo da odjednom izvršimo više request-ova, da se oni paralelno izvrše, za to koristimo all, any, race i allSettled.

1 Promise.all

```
Promise.all([
  Promise.resolve("Temperatura 20"),
  Promise.resolve("Vlažnost 30"),
  Promise.resolve("Brzina vetra 5")
]).then( (message) => {
  console.log(message)
}).catch( (error) => [
  console.error(error)
])
```



Sve ovo možemo posmatrati kao jedan promis. Ova lista koja u sebi sadrži promise, na kraju vrati 1 promis, tj. 1 rešenje. Kod Promise.all u slučaju da nam server za 1 od ova 3 request-a vrati "neuspešan je", mi ćemo dobiti **catch**. Promise.all vraća takođe promis, što znači da sa njim baratamo tako što radimo **then**. Promise.all vraća **then** samo ako su svi promisi uspešno izvršeni.

2 Promise.any

```
Promise.any([
  Promise.resolve("Temperatura 20"),
  Promise.reject("Vlažnost 30"),
  Promise.reject("Brzina vetra 5")
]).then( (message) => {
  console.log(message)
}).catch( (error) => [
  console.error(error)
])
```



Za any nam je bitno da se bilo koji promis uspešno završi. Ukoliko se desi da su sva tri odbijena, u tom slučaju ćemo upasti u grešku. Dobićemo skroz novu grešku - All promises were rejected.

3 Promise.race

```
Promise.race( [
  Promise.resolve("Temperatura 20"),
  Promise.reject("Vlažnost 30"),
  Promise.reject("Brzina vetra 5")
]).then( (message) => {
  console.log(message)
}).catch( (error) => [
  console.error(error)
])
```



Race služi kada želimo rezultat prvog promisa koji nam dođe, bio on uspešan ili neuspešan.

4 Promise.allSettled

```
Promise.allSettled( [
  Promise.reject("Temperatura 20"),
  Promise.resolve("Vlažnost 30"),
  Promise.reject("Brzina vetra 5")
]).then( (message) => {
  console.log(message)
}).catch( (error) => [
  console.error(error)
])
```



Kod allSettled mi čekamo da se svi promisi završe i posle gledamo da li su oni resolve ili reject, dakle uvek će biti svi izvršeni.

```
Promise.resolve("uspešno")
  .then( (message) => console.log(message) )
  .catch( (error) => console.error(error) )
  .finally( () => console.log("uvek se izvršava!") )
```

Finally je deo koda koji će se izvršiti, nebitno da li je then ili catch

ASYNC - AWAIT

 [POGLEDAJ LEKCIJU](#)



Async - rezervisana reč koja označava da se u funkciji radi asinhrono.



Funkcije koje vraćaju promis moramo da sačekamo sa **await**.



U slučaju da želimo da radimo u nekoj funkciji sa promisima, mi tu funkciju moramo da označimo da je ona async. Na taj način JavaScript zna da se u toj funkciji radi sa asinhronim kodom i moramo te pozive funkcije koji nam vraćaju promis da uhvatimo sa await.



Async uvek pišemo pre reči function.

```
function odložilzvršavanje(delay) {  
  return new Promise( (resolve, reject) => {  
    setTimeout( () => {  
      resolve("test")  
    }  
    , delay)  
  })  
}
```

```
odložilzvršavanje(3000).then( () => {  
  console.log("Ćao1")  
  return odložilzvršavanje(3000)  
}).then( () => {  
  console.log("Ćao2")  
})
```

```
async function odloži() {  
  const result = await odložiIzvršavajne(3000)  
  console.log("Ćao1", result)  
}
```

odloži()



izvršavanje koda iz prethodnog
primera upotrebom async-a i
await-a

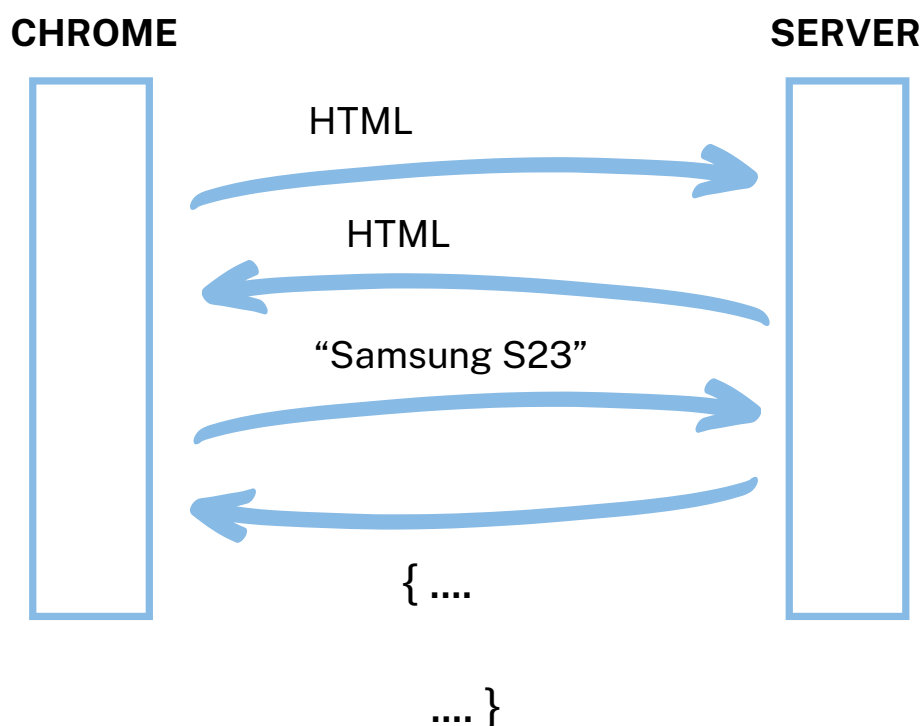
KOMUNIKACIJA SA SERVEROM

Uvod -AJAX i API-

 [POGLEDAJ LEKCIJU](#)

-  **Network tab** - snima svu komunikaciju između frontenda i backenda, između našeg pretraživača i našeg servera.
-  **Lazy loading** - tehnika čekanja da se učitaju određeni delovi web stranice.
-  **Fetch** - učitava podatke “ispod haube”, ovde nisu podaci koji podrazumevaju slike, HTML i JS, nego su isključivo podaci koji su čitljivi samo ovoj stranici i prikazani su u formatu koji ta stranica razume.

Kako izgleda komunikacija između chrome-a i servera?





Sa jedne strane imamo pretraživač, a sa druge server. Mi šaljemo serveru zahtev i tražimo HTML koji nam server vrati, a taj HTML u sebi sadrži linkove ka drugim JS fajlovima, ka slikama... Kada server počne sve to da vraća, tada se stranica učitala. Ali posle toga, ukoliko izvršimo neku akciju (npr. pretražujemo Samsung S23) kada se to upiše šalje se novi request ka serveru i to je upravo taj “zahtev ispod haube” koji se ne vidi i gde mi u pozadini iz našeg JS-a tražimo od servera da nam vrati sve što je vezano za S23 te nam on to šalje nazad u JSON formatu.



XHR - XMLHttpRequest je način na koji JS komunicira sa serverom i njegova skraćenica je **XHR** što se i nalazi u devTool-u (**Fetch / XHR**). Svaki put kada izaberemo XHR to su zahtevi koji se u pozadini šalju iz JS-a. On je u **XML** formatu. XML format se koristio pre JSON-a, a JSON je jako sličan JavaScript-u i mnogo je jasniji za čitanje.

XML možemo zamisliti kao HTML koji čuva podatke. JSON je noviji i danas se mnogo češće koristi te gotovo da i nema više toliko aplikacija koje koriste XML za prenos podataka.



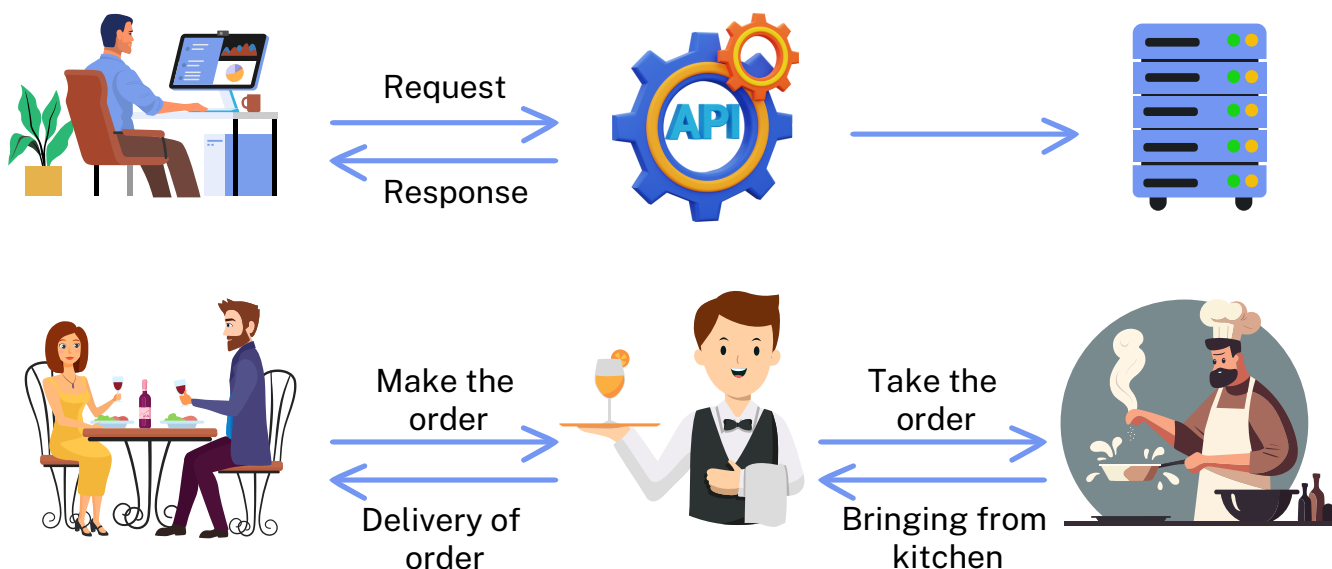
AJAX bi bila unapređena verzija XMLHttpRequest-a koja može da koristi i JSON, a ne samo XML. Od sada ćemo uvek pričati samo o AJAX-u, tj. o načinu prenošenja podataka sa frontenda i backenda upotrebom JSON-a.



API je ono što kontaktiramo kada želimo da dobijemo pozadinske zahteve. Npr. kada upišemo Samsung S23 mi kontaktiramo Samsungov API koji nam vraća rezultat u JSON-u. Praksa je da se u nazivu URL-a stavi reč API kako bi se označilo da je to API a ne web stranica.

API nam zapravo služi da dobavimo informacije od servera koje nisu u HTML formatu. Nama API vraća JSON.

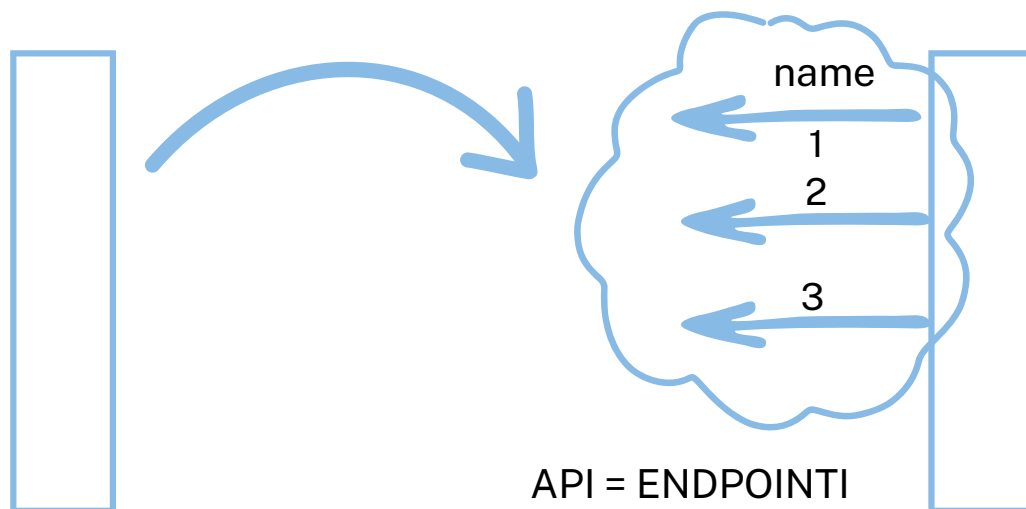
API (Application Programming Interface) je interfejs na koji se zakačimo kako bismo komunicirali sa serverom, tj. naša stranica i JS se zakače na API kako bi komunicirali sa serverom.



Primer iz realnog života bi bila situacija gde smo mi otišli u restoran. U restoranu ne komuniciramo direktno sa kuvarom, već sa konobarom, a konobar posle komunicira sa kuvarom. Konobar zna šta možemo da poručimo, a to je nešto što se nalazi u meniju. To što bi bio meni u restoranu, kod servera se zove **endpointi**.

CHROME

SERVER



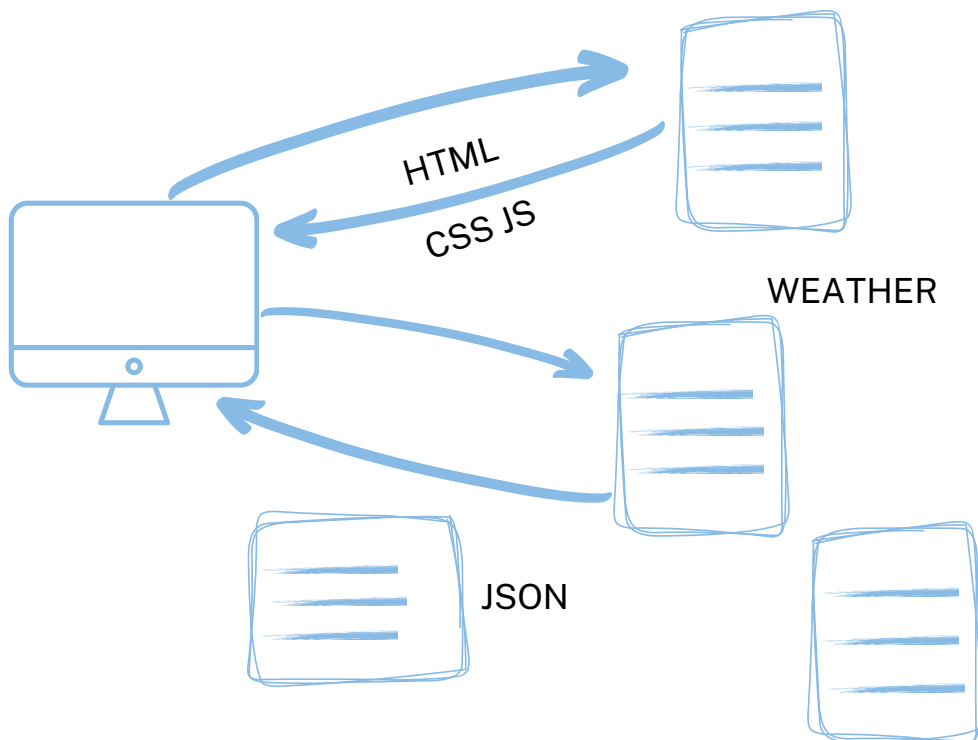
Ovde imamo server koji nam npr. dobavlja informacije o nekoj državi, gradu, rekama... Mi možemo da se povežemo sa našeg pretraživača na neki od ovih endpointa (to bi bile neke pristupne tačke) i ti endpointi su zapravo naš API. Dakle, pristupna tačka ovom našem serveru su jedino ova 3 endpointa. Ukoliko pogodimo neki od ta 3 endpointa mi ćemo dobiti informacije, ali moramo da pogodimo na način koji naš server propisuje. API zapravo nama pruža endpointe na koje se mi zakačimo preko frontenda i dobijamo informacije.

JSON

-JavaScript Object Notation-

 [POGLEDAJ LEKCIJU](#)

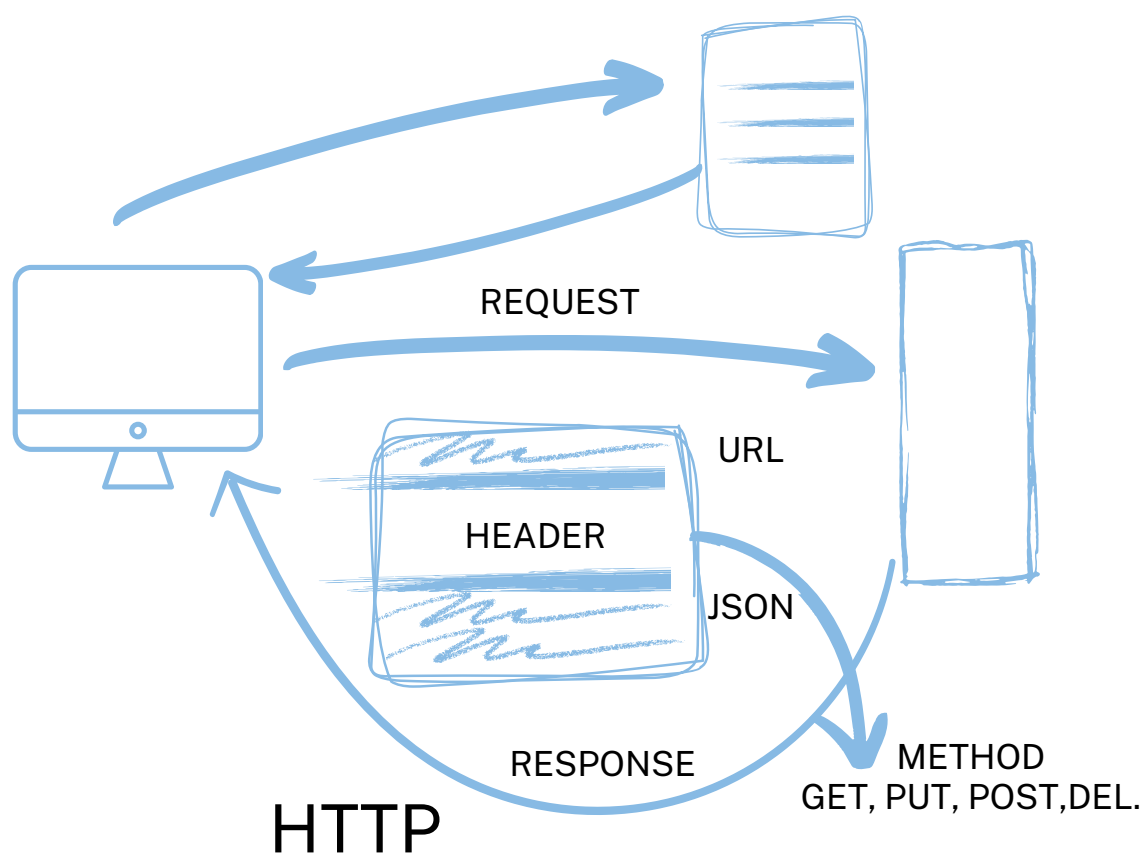
 **JSON** je format koji nam služi za prenos podataka od klijenta do servera.



 Mi kroz frontend komuniciramo sa serverom i šaljemo mu zahtev, a on nama pošalje HTML, CSS i JS fajlove i mi tada učitalamo stranicu. Ali šta u slučaju da želimo da na stranici prikazemo informacije o vremenskoj prognozi? Rekli smo da postoje servisi koje mi možemo da kontaktiramo, a to su npr. neki drugi serveri koji nama serviraju podatke. Dakle, mi iz naše frontend aplikacije koju smo učitali kontaktiramo druge servere (npr. za vremensku prognozu), gađamo određeni endpoint, tj. određenu adresu koju taj server podržava i nama se vraćaju podaci. Ti podaci koji se vraćaju su najčešće u formatu JSON.

HTTP/API

 [POGLEDAJ LEKCIJU](#)



Ovo je primer HTTP zahteva i HTTP odgovora. Aplikacija kontaktira server i dobije HTML, CSS, itd. Kada sve to dobije, kontaktira druge servere (npr. za vremensku prognozu) i šalje zahtev. **Kako taj zahtev izgleda?** To možemo zamisliti kao jedan paket koji ima nekoliko stvari: **URL**, tj. putanju na koju ide, **hedere** i **deo koji služi za podatke**. Protokol kojim se šalju ovi paketi se naziva **HTTP** (Request - Response).



Request - zahtev **response** - odgovor (ono što server vraća)

POSTMAN

 [POGLEDAJ LEKCIJU](#)



Postman je alat koji služi da kontaktiramo backend, tj. da kontaktiramo API bez da to kodiramo u našoj aplikaciji.



Swagger nam služi za dokumentovanje API-a i prikazuje listu svih endpointa nekog servera.



Path parametar - nalazi se u putanji URL-a i to je jedna stvar koja je promenljiva u okviru request-a. Druga stvar jesu **query parametri**.

Query parametri nam služe da isfiltriramo naš request. To se najčešće koristi kada radimo neke pretrage. Redaju se posle upitnika u URL-u i sastoje se od imena, zatim ide jednako pa vrednost, zatim & (nabrajanje)...

Path parametri služe za ciljano traženje određenog item-a po nekom id-u.

www.vreme.com/api/trenutna-temperatura/gradovi/1

- ovde 1 na kraju predstavlja path parametar (dovlacimo temperaturu za prvi grad)

www.vreme.com/api/trenutna-temperatura?grad=NoviSad

- ovde grad=NoviSad predstavlja query parametar (dovlacimo trenutnu temperaturu za Novi Sad)

XMLHttpRequest



[POGLEDAJ LEKCIJU](#)



Kreiranje objekta XMLHttpRequest:

```
const req = new XMLHttpRequest()
```

```
const req = new XMLHttpRequest()
```

```
req.onload = function () {  
  console.log("onload", this)  
  console.log("data", JSON.parse(this.responseText))  
}
```

```
req.onerror = function () {  
  console.log("error", this)  
}
```

```
req.open("GET", "https://swapi.dev/api/people/1")  
req.send()
```

kontaktiranje API-a preko
JavaScript-a uz pomoć
XMLHttpRequest-a



FETCH

 [POGLEDAJ LEKCIJU](#)

```
fetch("https://swapi.dev/api/people/1")  
.then(response => {  
  console.log("response", response)  
  response.json().then(data => console.log("data", data))  
})  
.catch(error => console.log("error", error))
```

kontaktiranje API-a preko
JavaScript-a uz pomoć
Fetch-a



AXIOS

 [POGLEDAJ LEKCIJU](#)

```
axios.get("https://swapi.dev/api/people/1")  
.then(response => console.log("response", response.data))  
.catch(error => console.log("error", error))
```

kontaktiranje API-a
preko JavaScript-a uz
pomoć Axios-a



```
const učitaj = async () => {  
  const res = await axios.get("https://swapi.dev/api/people/1")  
  console.log("data", res.data)  
}  
  
učitaj()
```

pretvaranje prethodnog
primera u async i await





JSON format se čuva u fajlu koji ima ekstenziju json (tekst.json). JSON je skraćenica od JavaScript Object Notation i to je format koji se ne koristi samo u JavaScript-u već se koristi generalno u web programiranju i u komunikaciji sa serverom, bilo da radimo sa node serverom, php-om... Najčešća komunikacija između frontenda i backenda je upravo JSON.

JSON je najpribližniji JavaScript-u, izgleda kao JavaScript i sadrži neke osnovne tipove kao JavaScript. JSON format počinje i završava se sa vitičastom zagradom. Pisanje JSON-a je skoro identično kao pisanje objekta u JS-u, sa tom razlikom što u JSON-u key deo moramo da stavimo pod navodnike.

JSON podržava string, number, boolean, niz, možemo da ugnježdavamo objekte unutar objekata...



JSON.parse - pretvara string u objekat



JSON.stringify - pretvara objekat u string (obrnuto)

TIPOVI HTTP METODA

GET

traži od servera da mu se nešto vrati

HEAD

proverjava najčšće dostupnost resursa

PUT

slanje nečega na server, najčešće se koristi za neku izmenu

TRACE

služe za dobijanje neke informacije o serveru

POST

slanje i kreiranje novog objekta

OPTION

koje HTTP metode server podržava za određeni resurs

DELETE

brisanje sa servera

AKCIJE KOJE MOŽEMO DA IZVRŠIMO NAD NEKIM RESURSOM



Create

C



Read

R



Update

U



Delete

D



STATUS KODOVI KOJE SERVER VRAĆA UZ RESPONSE

1xx - status kodovi informacionog tipa

2xx - status kodovi koji označavaju da je sve u redu

3xx - privremene ili trajne redirekcije

4xx - greške na strani klijenta

5xx - status kodovi koji predstavljaju greške na serveru

Čestitam, ovo je sam kraj priručnika!

Nadam se da ti je bio koristan. Bilo je potrebno dva meseca da se on napravi i mnogo godina učenja iza toga. Povremeno proveri sajt kako bi preuzeo najnoviju verziju.

Ukoliko imaš bilo kakvih pitanja, komentara ili želiš da поделиš svoje utiske, slobodno mi piši.

- **Društvene mreže:** @postani.it
- **E-mail:** ivan@postani.it

Ivan | Postani IT

P.S. Nastavi da učiš jer ne možeš da izgubiš osim ako ne odustaneš.

