Javascript osnove II

**Skraceno pisanje operatora dodijeljivanja**

Ukoliko nad nekom varijablom zelimo da izvrsimo neku matematicku operaciju i novodobijenu vrijednost ponovo dodijelimo toj varijabli, mozemo koristiti skraceni oblik:  
  
let stepCounter = 10;

umjesto stepCounter = stepCounter + 10;

mozemo napisati

stepCounter += 10;

Isti koncept vazi i za ostale matematicke operacije.   
Ukoliko se vrijednost varijable povecava za 1, onda moze jos krace:

let i = 1;

i++;

odnosno

i—;

**Funkcije**

Funkcija je blok koda koji izvrsava odredjeni zadatak. Funkcije se najcesce koriste kada je potrebno neke operacije izvrsiti vise puta, tako da umjesto ponavljanja koda, mi samo “pozovemo funkciju”.  
  
Na primjer, imamo zahtjev za sabiranjem negdje u nasoj aplikaciji. To mozemo rjesiti na jednostavan nacin tako sto saberemo vrijednost varijabli:  
  
const a = 10;

const b = 5;

const result = a + b;

Ovdje je vrlo jasno ogranicenje - izvrsili smo sabiranje i dobili rezultat i ako je potrebno jos negdje u aplikaciji da uradimo isto - ovaj kod moramo ponoviti (koristeci drudge varijable).

Funkcije olaksavaju rjesavanje ovakih problema:

function add(first, second) {

return first + second;

}

Sada ovu funkciju mozemo ‘pozivati’ n puta sa razlicitim ulaznim vrijednostima (argumenti funkcije).

const result1 = add(10, 5);

const result2 = add(20, 100);

const result3 = add(50, -10);

Iz prethodnog mozemo zakljuciti:

* funkcije imaju svoju definiciju, gdje im se daje naziv, odredjuju argumenti i pise logika koju ce izvrsavati)
* funkcije se moraju “pozvati” kako bi se kod u njima izvrsio.

Funkcije mogu imati neogranicen broj argumenata (ali ovo treba drzati u granicama normale, ako broj prelazi 3, vjerovatno je bolje prosledjivati objekat).

Funkcije mogu imati obavezne i opcionalne (default) argumente. Obavezni su svi oni argumenti koji nemaju zadatu podrazumijevanu vrijednost prilikom definisanja funkcije. Opcionalni imaju “default” vrijednost, koja ce biti koriscena ukoliko nista nije prosledjeno na toj poziciji prilikom izvrsavanja funkcije.

function calculateVat(total, vatPercentage = 0.17) {

return total \* vatPercentage;

}

Ovu funkciju mozemo bezbjedno pozvati prosledivsi samo jedan parametar:  
  
calculateVat(100); // 0.17 ce biti korisceno kao drugi parametar

ali:

calculateVat(100, 0.2) // 0.2 ce biti korisceno kao drugi parametar

U prethodnim primjerima smo vidjeli kljucnu rijec “return”. Ona zapravo vraca vrijednost nazad iz funkcije, zato smo mogli da kazemo const result = add(1, 1);

Funkcije ne moraju ovo raditi nuzno, tj. ne moraju vratiti vrijednost i tada se za fnkciju kaze da je tipa “void”.

**For petlja**

For petlja se najcesce koristi za prolazak kroz niz. Recimo da imamo sledeci niz:

const sentenceParts = [“My”, “name”, “is”, “Branimir”];

Ukoliko bismo zeljeli da od ovog niza napravimo string, morali bismo nekako da prodjemo kroz sve element i spojimo ih u recenicu.  
  
let sentence = “”;

for(let i = 0; i < sentenceParts.lenght; i++) {

sentence += sentenceParts[i] + “ “;

}

console.log(sentence);

sentenceParts.lenght je duzina niza. Prethodnim kodom smo rekli da zelimo proci kroz sve element do ne dodjemo do poslednjeg i dodavati ih na sentence string.Pojedinacnom elementu pristupamo pomocu negovog ineksa (u nasem slucaju i). Indeksi pocinju sa 0.Dakle, ako kazemo:

console.log(sentenceParts[2]);

dobicemo “is” kao rezultat.

**For in petlja**

For in se koristi za prolazenje (iteraciju) kroz polja nekog objekta. Recimo da imamo:

const person = {

firstName: “Branimir”,

lastName: “Dragicevic”

};

iteraciju kroz ovaj objekat mozemo uraditi na sledeci nacin:

for(let field in person) {

console.log(person[field]); // filed ce zapavo biti kljuc (firstName, lastName), tako da sa person[field] dobijamo vrijednosti koje se nalaze pod tim kljucevima

}

Postoji takodje i **for of** petlja. Ona se koristi isto kao for in, ali moze se primjeniti i na druge tipove koji podrzavaju iteraciju (nizovi, mape itd).

**While petlja**

Za for petlju mozemo lako zakljucit da se se zavrsiti onda kada prodje kroz sve elemente niza (osim ako se eksplicitno prekinula upotrebom kljucne rijeci **break**!).

While petlju koristimo najcesce kada inicijalno ne znamo koliko iteracija treba uraditi, tj. izvrsavano je sve dok je odredjeni uslov zadovoljen.

Na primjer:

let guessWord = process.argv[0];

const SECRET\_WORD = “LUCK”;

while(guessWord !== SECRET\_WORD) {

console.log(“Entered word is not correct. Please try again”);

guessWord = readline.question(“Enter word:”);

}

Kod while petlji moramo biti narocito oprezni, jer se moze desiti da uslov nikad ne bude zadovoljen i tada cemo uci u takozvanu beskonacnu petlju (blokiran cijeli sistem).

**do-while petlja**

Za while petlju se moze desiti da se nikada ne izvrsi. Ukoliko imamo slucaj da se while petlja mora bar jednom izvrsiti, do-while petlja je rjesenje. Njena sintaksa izgleda ovako:

do {

console.log(“I will be printed at least once!”);

} while (someCondition);

Ovakvi zahtjevi su vrlo rijetki u praski i male su sanse da cete se cesto susretati sa do-while.

**break i continue kljucne rijeci**

ponekad imamo potrebu da eskplicitno zavrsimo neku petlju ili da preskocimo izvrsavanje neke logike u odredjenim slucajevima. Za tu svrhu koristimo break i continue kljucne rijeci.  
break ce prekinuti kompletno izvrsavanje, a continue “ce skociti” na naredi element.  
  
Na primjer:

const arr = [10, 20, 60, 80, 100];

const guess = 80;

for(let i = 0; i < arr.length; i++) {

if(arr[i] === guess) {

console.log(“You guessed it right!”);

break;

}

}

const arr = [10, 20, 60, 80, 100];

const excludeValue = 60;

let sum = 0;

for(let i = 0; i < arr.length; i++) {

if(arr[i] === excludeValue ) {

continue;

}

sum += arr[i];

}

console.log(sum);

**Bitne funkcije i karakteristike nizova**

Pored “rucnih” iteracija koje smo prethodno prosli, Javascript ima ugradjene funkcije i karakteristike za operacije nad nizovima.

Neke od najbitnijih su:

* length (property) - ovo vraca duzinu niza. Koristi se cesto kod iteracija, za provjeru da li je niz prazan, za pristup poslednjem elementu niza (arr[arr.length - 1] - posto su indeksi bazirani na 0) itd.
* forEach (function) - prolazi kroz sve elemente niza (ne vraca rezultat). npr

const arr = [1, 2, 3];

arr.forEach(function(element) {

console.log(element);

});

arr.forEach((element) => {

console.log(element);

});

ali ako probamo

const newArr = arr.forEach(function(element) {

return arr;

});

newArr ce imati vrijednost undefined, posto forEach ne vraca nista.

* push (function) - dodaje novi element na kraj niza.

arr.push(4);

* pop (function) - isjeca (i vraca) posledni element iz niza.

const lastElement = arr.pop(); // lastElement = 4; arr = [1, 2, 3];

* unshift (function) - dodaje element na prvu poziciju u nizu.

arr.unshift(0); // [0, 1, 2, 3];

* shift (function) - isjeca (i vraca) prvi element iz niza.

const firstElement = arr.shift(); // firstElement = 0; arr = [1, 2, 3];

* toString (function) - pretvara niz u string
* join (function) - pretvara niz u string, ali ima opciju da se prosledi delimitar,

arr.join(“, “); // 1, 2, 3

- delete (keyword) - brise ordedjeni element iz niza. Napomena! - na tom indeksu ostaje undefined.

* concat (function) - spaja nizove u jedan.

arr.concat([4, 5]); // [1, 2, 3, 4, 5]

* max (function) - vraca najvecu vrijednost iz niza.
* min (function) - vraca najmanju vrijednost iz niza.

Zadatak 1:

Napisati sopstvenu verziju max i min funkcija.

function newMax(arr) {

let max = arr[0];

if(arr.length === 0) {

return “Invalid input”;

}

for(let i = 1; i < arr.length; i++) {

if(arr[i] > max) {

max = arr[i];

}

}

return max;

}

console.log(newMax([23, 5, 2]))

Zadatak 2:

Kreirati safeDelete funkciju. Zadatak ove funkcije je da omoguci brisanje elementa iz niza sa odredjenog indeksa i da ne ostavi undefined “rupe”.

Uzmimo niz const arr = [1, 2, 3, 4, 5];

Kada se pozove safeDelete(arr, 2) nazad treba da dobijemo [1, 2, 4, 5]

function safeDelete(arr, deleteIndex) {

const newArr = [];

for(let i = 0; i < arr.length; i++) {

if(i !== deleteIndex) {

newArr.push(arr[i]);

}

}

return newArr;

}

Zadatak 3:

Napisati funkciju sum koja Izracunava zbir elemenata sledeceg niza koristeci for loop:  
  
const arr = [10, 20, 60, 80];

function sum(arr) {

let total = 0;

const len = arr.length

for(let i = 0; i < len; i++) {

total += arr[i];

}

return total;

}

Zadatak 4:

Napisati funkciju getSubarray(arr, index) koja vraca novi niz koji se sastoji od elemenata od pozicije nula do zadate pozicije index; ili vraca sve elemente (originalni niz) ako je index van granica niza.

function getSubarray(arr, index) {

const arrLength = arr.length;

if(arrLength < index) {

return arr;

}

const subarray = [];

for(let i = 0; i <= index; i++) {

subarray.push(arr[i]);

}

return subarray;

}

Zadatak 5:

Napisati funkciju koja “rotira” elemente niza. Npr, niz [5, 2, 3, 7] treba da rezultira sa [7, 3, 2, 5]

function rotateArray(arr) {

const rotatedArr = [];

const arrLength = arr.length - 1;

for(let i = arrLength; i >= 0; i–) {

rotatedArr.push(arr[i]);

}

return rotatedArr;

}

Zadatak 6:

Napisati funkciju koja racuna broj parnih elemenata u nizu (podazumijeva se da je niz sastavljen od cjelobrojnih brojeva, tako da ovo ne moramo provjeravati).

function countEven(arr) {

let counter = 0;

for(let i = 0; i < arr.length; i++) {

if(arr[i] % 2 === 0) {

counter++;

}

}

return counter;

}

Zadatak 7:

Napisati searchArray(value) funkciju koja pretrazuje niz i vraca index ako je trazena vrijednost pronadjena u nizu - ili -1 ako vrijednost ne postoji.

function searchArray(value) {

const arr = [2, 5, 8, 1];

for(let i = 0; i < arr.length; i++) {

if (value === arr[i]) {

return i;

}

}

return -1;

}