**Funkcije**

**00:19:40**

Jedinica: 25 od 30

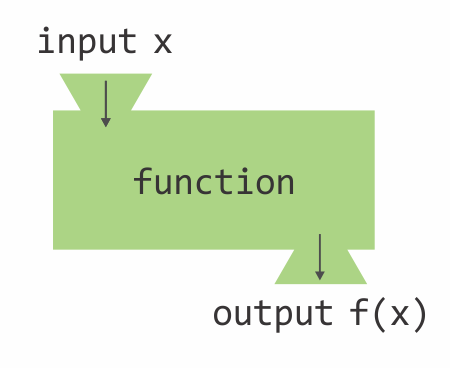
**Rezime**

* Funkcija je specijalni blok koda, zadužen za izvršenje određene logike.
* Funkcija prilikom izvršenja logike može da koristi određene vrednosti, koje se nazivaju ulazni parametri, a može i emitovati svoj rezultat kao povratnu vrednost.
* U programskom jeziku JavaScript funkcija se definiše korišćenjem ključne reči function.
* Logika funkcije se definiše unutar njenog tela.
* Kako bi se logika funkcije uposlila, nju je potrebno pozvati.
* Funkcija može imati ulazne parametre i povratnu vrednost, ali to nije obavezno.
* Povratna vrednost funkcije se emituje ključnom rečju new.
* U JavaScript jeziku, funkcija se može definisati i bez naziva i tada se govori o takozvanim anonimnim funkcijama.
* JavaScript poznaje i takozvane samopozivajuće funkcije koje se aktiviraju automatski onoga trenutka kada izvršavanje koda dođe do njih.
* Kada se kreira jedna funkcija, s obzirom da je reč o bloku koda, takva funkcija postaje jedna leksička celina.
* Promenljive kreirane unutar funkcije nazivaju se lokalne promenljive i one su vidljive samo funkciji unutar koje su kreirane.

Kada je u prethodnim lekcijama bilo reči o blokovima koda, pomenut je pojam funkcija. Funkcije su specijalni blokovi koda, koji su, iako to možda niste znali, korišćeni u gotovo svakoj lekciji za nama. Funkcije su osnovni gradivni blok gotovo svakog jezika, i ne može se zamisliti pisanje koda bez njihove upotrebe. U jeziku JavaScript funkcije zauzimaju posebno mesto značajem koji im je u samom jeziku dodeljen. Stoga će lekcija pred vama biti posvećena pojmu funkcije.

**Šta su funkcije?**

Funkcija je specijalni blok koda, zadužen za izvršenje određene logike. Pritom, funkcija prilikom izvršenja takve logike može da koristi određene vrednosti, koje se nazivaju ulazni parametri, a može i emitovati svoj rezultat kao povratnu vrednost. Uloga funkcije može se predstaviti kao na slici 25.1.



*Slika 25.1 – Funkcija*

Na slici 25.1 slovom x je označen ulazni parametar, odnosno vrednost koju funkcija dobija na obradu. Nakon završene obrade, funkcija emituje rezultat izvršavanja. Ipak, kao što je rečeno, nije obavezno da funkcija ima ulazne i izlazne parametre.

Sa slike 25.1 se još može videti da je funkcija jedna zaokružena celina. To je vrlo važna osobina funkcije iz koje proizilaze brojne prednosti [funkcionalnog programiranja](https://www.link-elearning.com/linkdl/opisPojma.php?id=144318). Najznačajnije prednosti su:

* Enkapsulacija
* smanjenje redundantnosti koda
* brzina kodiranja
* portabilnost

Funkcija je jedna zaokružena celina, koja je u potpunosti sposobna da izvrši jedan deo posla, zato što sadrži sve potrebne elemente za tako nešto. Grupisanje svih potrebnih elemenata u jednu celinu drugačije se naziva enkapsulacija.

Takođe, prilikom pisanja koda, često se javlja potreba da se određeni deo koda izvrši nekoliko puta na različitim mestima, odnosno u različitim situacijama. Da funkcije ne postoje, za rešavanje pomenutog problema bilo bi neophodno jedan isti kod za rešavanje problema kopirati na sva ona mesta na kojima se javi potreba za određenom funkcionalnošću. Na taj način bi se značajno povećala redundantnost koda, smanjila preglednost i otežalo održavanje. Funkcije ovakve probleme elegantno rešavaju tako što omogućavaju da se jednom definisana logika pozove onoliko puta koliko je to potrebno.

**Definisanje funkcije**

U programskom jeziku JavaScript funkcija se definiše korišćenjem ključne reči function. Nakon ove ključne reči navode se naziv funkcije, eventualni parametri i blok koda oivičen uglastim zagradama (slika 25.2.).



*Slika 25.2 – Sintaksa funkcije u JS-u*

Zamislite sada situaciju po kojoj je potrebno u nekoj aplikaciji računati površinu pravougaonika. Površina pravougaonika se računa tako što se pomnože dužine njegovih stranica a i b. JavaScript kod koji bi tako nešto obavljao mogao bi da izgleda ovako:

|  |  |
| --- | --- |
| 1  2  3  4 | var a = 5;  var b = 10;    var rect\_area = a\*b; |

Prikazani kod je potpuno funkcionalan, odnosno, bez problema računa površinu pravougaonika sa dužinama stranica 5 i 10. Ali, šta ukoliko je potrebno površinu računati često, i to za pravougaonike različitih veličina. U takvoj situaciji, neophodno bi bilo ponavljati uvek iznova logiku za računanje površine. Stoga se takva logika može enkapsulirati unutar jedne funkcije:

|  |  |
| --- | --- |
| 1  2  3 | function calculateRectArea(a, b){   return a\*b;  } |

Kreirana funkcija je nazvana calculateRectArea, prihvata dva parametra, obavlja računanje površine i emituje rezultat kao povratnu vrednost. Novina u primeru je upotreba ključne reči return, što je naredba kojom funkcija emituje svoju povratnu vrednost.

Na ovaj način, ne samo da je izvršena enkapsulacija logike za računanje površine pravougaonika već je funkcionalnost generalizovana, pa je tako primenljiva na pravougaonicima svih veličina.

**Pozivanje funkcije**

Definisanje funkcije, samo po sebi, ne pokreće izvršavanje njene logike, već kao što i sam naziv kaže, isključivo vrši njeno definisanje. Kako bi se logika funkcije uposlila, nju je potrebno pozvati. Sledeća linija koda ilustruje pozivanje malopre kreirane funkcije:

|  |  |
| --- | --- |
| 1 | calculateRectArea(5, 10); |

Na ovaj način, poziva se funkcija sa nazivom calculateRectAreai njoj se prosleđuju ulazni parametri 5 i 10. Tako se aktivira logika funkcije, koja je definisana unutar njenog tela. Funkcija obavlja posao definisan njenom logikom, i vraća vrednost izvršavanja. Ukoliko se izvrši testiranje prikazanog primera, sve ovo što je navedeno ostaće nevidljivo za posmatrača, odnosno povratna vrednost funkcije nigde neće biti vidljiva. To je potpuno logično, s obzirom da u prikazanoj liniji u kojoj se poziva funkcija nije precizirano šta uraditi sa rezultatom koji funkcija isporučuje. Stoga će primer biti modifikovan na sledeći način:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | function calculateRectArea(a, b){   return a\*b;  }    var result = calculateRectArea(5, 10);  console.log(result); |

Sada je rezultat funkcije uhvaćen unutar promenljive sa nazivom result. Da je to stvarno tako, ilustruje ispis koji proizvodi prikazani kod:

50

**Ulazni parametri i povratna vrednost**

Funkcija može imati ulazne parametre i povratnu vrednost, ali to nije obavezno. Prethodni primer, u kome je prikazana funkcija za računanje površine pravougaonika, ilustrovao je upotrebu ulaznih parametara i povratne vrednosti. Ipak, za ispravno korišćenje funkcija, potrebno je poznavati neke važne osobine ulaznih i izlaznih vrednosti.

Parametri se funkciji mogu proslediti po vrednosti ili referenci. Svi parametri koji su prostih tipova uvek se prosleđuju po vrednosti. To praktično znači da se funkciji prosleđuje kopija vrednosti promenljive, a ne sama promenljiva. Tako funkcija dobija potrebnu vrednost, ali nema mogućnost da izvrši promenu originalne vrednosti. To ilustruje sledeći primer:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | var a = 5;  var b = 10;    function calculateRectArea(a, b){     var result = a\*b;     a = 30;     return result;  }    var result = calculateRectArea(a, b);    console.log('Result is: ' + result);  console.log('a = ' + a); |

Primer ilustruje definisanje i pozivanje funkcije za računanje površine pravougaonika. Funkcija dobija dva parametra, odnosno vrednosti 5 i 10 i računa površinu. Ipak, pored računanja površine, unutar funkcije se obavlja i postavljanje vrednosti promenljive a na 30. Na kraju se unutar konzole ispisuje rezultat računanja, ali i vrednost promenljive a:

|  |  |
| --- | --- |
| 1  2 | Result is: 50  a = 5 |

Potpuno je jasno da se vrednost promenljive a nije promenila, odnosno da promena vrednosti unutar funkcije nije imala nikakav efekat, s obzirom da funkcija rukuje samo kopijom vrednosti stvarne promenljive.

Situacija je značajno drugačija kada se funkciji prosledi neki objektni tip podatka. Tada se takav parametar prosleđuje po referenci i funkcija ima mogućnost da promeni izvornu vrednost promenljive.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | function calculateRectArea(theObject){     var result = theObject[0]\*theObject[1];   theObject[0] = 30;   return result;  }    var rectDimensions = [5, 10];    var result = calculateRectArea(rectDimensions);  console.log('Result is: ' + result);  console.log('a = ' + rectDimensions[0]); |

Ovoga puta je funkcija transformisana tako da prihvata jedan parametar. Parametar je objektni tip Array sa dva člana koji predstavljaju dužine stranica pravougaonika. Unutar funkcije se čitaju vrednosti dva člana prosleđenog niza i računa se površina. Nakon toga, menja se vrednost prvog člana niza na 30.

Izvan funkcije, prvo se kreira jedan niz sa nazivom rectDimensions i dva člana 5 i 10. Ovaj niz se prosleđuje funkciji prilikom poziva, a povratna vrednost se hvata unutar promenljive result. Na kraju se u konzoli ispisuje povratna vrednost funkcije, ali i vrednost prvog člana niza rectDimensions. Rezultat prikazanog koda je:

|  |  |
| --- | --- |
| 1  2 | Result is: 50  a = 30 |

Može se primetiti da je vrednost prvog člana niza 30, odakle se zaključuje da je promena izvršena unutar funkcije, promena na originalnoj vrednosti ulaznog parametra.

**Da li je neophodno da funkcija vraća neku vrednost?**

 Da

 Ne

**Anonimne funkcije**

Ukoliko pažljivije obradite pažnju na prethodne primere u ovoj lekciji, možete da vidite da je svaka kreirana funkcija imala svoj naziv. U JavaScript jeziku, funkcija se može definisati i bez naziva i tada se govori o takozvanim anonimnim funkcijama. Sledeći primer ilustruje jednu anonimnu funkciju:

|  |  |
| --- | --- |
| 1  2 | var a = function(x){return x\*x;};  console.log(a(5)); |

Kao što se može videti, funkcija je kreirana bez naziva i dodeljena promenljivoj a. Nadalje, promenljiva a će služiti kao identifikator kreirane funkcije. To dokazuje druga linija koda u kojoj se funkcija poziva korišćenjem naziva promenljive kojoj je dodeljena. Česta upotreba anonimnih funkcija jeste omogućavanje prosleđivanja jedne funkcije drugoj:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | function doSomeStuff(a, b){     if(a == 13){    b(a);   }    }    doSomeStuff(13, function(num){   console.log(num + " is not allowed!");  }); |

U primeru je kreirana funkcija doSomeStuff. Ova funkcija prihvata dve vrednosti a i b. Unutar tela funkcije utvrđuje se da li je prvi parametar jednak broju 13. Ukoliko jeste, korišćenjem promenljive b, izvršava se pozivanje funkcije, koja se anonimno definiše prilikom poziva funkcije doSomeStuff.

Prilikom poziva funkcije doSomeStuff kao prvi parametar se prosleđuje broj 13. Drugi parametar je anonimna funkcija, čija je uloga da ispiše jednu poruku. Na ovaj način se ovakva anonimna funkcija prosleđuje kao parametar funkciji doSomeStuff, a njeno izvršavanje se aktivira iz tela funkcije doSomeStuff.

**Samopozivajuće funkcije**

Pored anonimnih funkcija, JavaScript poznaje i takozvane samopozivajuće funkcije. To su funkcije koje se aktiviraju automatski onoga trenutka kada izvršavanje koda dođe do njih, bez potrebe da budu eksplicitno pozvane. Primer jedne takve funkcije je sledeći:

|  |  |
| --- | --- |
| 1  2  3 | (function () {      console.log("Hello from self-invoking function");  })(); |

Kada izvršavanje koda dođe do ovako definisane funkcije, ona će odmah biti izvršena.

**Function scope**

Kada se kreira jedna funkcija, s obzirom da je reč o bloku koda, takva funkcija postaje jedna leksička celina. Takva leksička celina se drugačije naziva *Function scope* ili oblast važenja jedne funkcije. Oblast važenja jedne funkcije poseban značaj ima za promenljive koje se deklarišu unutar, ali i izvan takvih funkcija. Naime, ukoliko se u razmatranje uzme samo jedna funkcija, neka promenljiva može biti deklarisana unutar funkcije ili izvan funkcije.

Kada se promenljiva deklariše izvan funkcije, takva promenljiva postaje globalna promenljiva. S druge strane, kada se promenljiva deklariše unutar jedne funkcije, ona je vidljiva samo unutar takve funkcije, a drugačije se naziva lokalna promenljiva.

Sledeći primer bavi se ovom problematikom:

|  |  |
| --- | --- |
| 1  2  3  4 | function myFunction() {  var x = 4;  }  console.log(x); |

Unutar funkcije myFunction, deklariše se i inicijalizuje promenljiva sa nazivom x i vrednošću 4. Izvan funkcije, pokušava se ispisivanje vrednosti promenljive x. Primer proizvodi sledeći rezultat:

|  |  |
| --- | --- |
| 1 | ReferenceError: x is not defined |

Naravno, kod proizvodi grešku, zato što promenljiva x nije vidljiva na globalnom nivou.  
S druge strane, globalna promenljiva je vidljiva unutar bilo koje funkcije:

|  |  |
| --- | --- |
| 1  2  3  4  5 | var x = 4;  function myFunction() {   console.log(x)  }  myFunction(); |

Promenljiva x je globalna promenljiva i kao takva dostupna i unutar funkcije myFunction. Zato primer bez problema ispisuje vrednost promenljive x:

|  |  |
| --- | --- |
| 1 | 4 |

Zanimljivo je da je i unutar funkcija moguće kreirati globalne promenljive. Dovoljno je promenljivu kreirati bez ključne reči var, i ona će postati globalna. Drugim rečima, inicijalizovanje promenljive koja prethodno nije deklarisana automatski takvu promenljivu čini globalnom. To ilustruje sledeći primer:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | function myFunction() {   x = 4;  }    myFunction();    console.log(x); |

Unutar funkcije myFunction inicijalizovana je promenljiva sa nazivom x, koja prethodno nije deklarisana. To će ovakvu promenljivu učiniti globalnom, ali u onom trenutku kada se takva linija izvrši, odnosno kada se pozove funkcija myFunction. U primeru se ova funkcija poziva, a zatim se ispisuje vrednost promenljive x i na izlazu se dobija:

|  |  |
| --- | --- |
| 1 | 4 |

Da je kojim slučaj promenljiva x i deklarisana unutar funkcije, primer bi proizveo grešku.

**Funkcija unutar funkcije**

Potpuno je legitimno pisati jednu funkciju unutar druge. Na taj način, unutrašnja funkcija je privatni član spoljne funkcije i ne može joj se pristupiti na globalnom nivou. Sledeći primer ilustruje funkciju unutar funkcije:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | function subtractSquares(a,b) {   function square(x) {    return x \* x;   }   return square(a) - square(b);  }    console.log(subtractSquares(6,3)); |

Primer ilustruje funkcionalnost za oduzimanje kvadrata dva broja. Naziv glavne funkcije je subtractSquares i ona prihvata dva parametra. Unutar ove funkcije, definisana je još jedna funkcija square, čija uloga je podizanje broja na kvadrat. Ova funkcija se koristi interno, unutar funkcije subtractSquares. Na kraju, funkcija subtractSquares se upošljava sa vrednostima 6 i 3, i dobija se sledeći odgovor:

*27*

Već je rečeno da se funkcija unutar funkcije ne može pozvati na globalnom nivou. Zato sledeći kod proizvodi grešku:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | function subtractSquares(a,b) {   function square(x) {    return x \* x;   }   return square(a) - square(b);  }    console.log(square(5)); |

Rezultat:

|  |  |
| --- | --- |
| 1 | ReferenceError: square is not defined |

**Closure**

Funkcije unutar funkcija imaju još jednu vrlo korisnu upotrebnu vrednosti. Već je rečeno da sve promenljive deklarisane unutar jedne funkcije postoje samo unutar takve funkcije.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | function iterate() {     var counter = 0;     return counter++;    }    counter++; |

S obzirom da je promenljiva counter lokalnog tipa, kod proizvodi izuzetak:

|  |  |
| --- | --- |
| 1 | ReferenceError: counter is not defined |

Nekada ćemo želeti da lokalnu promenljivu učinimo dostupnom spoljnom svetu. Upravo to se može postići korišćenjem funkcije unutar funkcije, odnosno upotrebom closurea.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | function iterate() {     var counter = 0;     function increment() {    return counter++;   }     return increment;  } |

Sada je unutar funkcije iterate definisana unutrašnja funkcija sa nazivom increment. Reč je o closure funkciji. Ova funkcija vraća vrednost counter promenljive uvećanu za jedan. Spoljašnja funkcija iterate kao svoju povratnu vrednost ima closure increment.

Na ovaj način je moguće na globalnom nivou ne samo dobiti vrednost promenljive counter, koja je lokalnog tipa, već i izvršiti njeno uvećanje. Sve što je potrebno jeste funkciju iterate dodeliti nekoj globalnoj promenljivoj, i nadalje će svaki poziv takve promenljive u stvari biti poziv closure funkcije increment.

|  |  |
| --- | --- |
| 1  2  3  4  5 | var incrementFromOutside = iterate();    console.log(incrementFromOutside());  console.log(incrementFromOutside());  console.log(incrementFromOutside()); |

Ovakav kod proizvodi efekat:

|  |  |
| --- | --- |
| 1  2  3 | 0  1  2 |

**Podizanje funkcija na početak dokumenta**

U svim dosadašnjim primerima, definicije funkcija uvek su prethodile pozivima takvih funkcija. Drugim rečima, funkcije su uvek prvo bile definisane, a tek onda i pozivane:

|  |  |
| --- | --- |
| 1  2  3  4  5 | function myFunction(x) {      return x \* x;  }    myFunction(6); |

Ovo je potpuno logičan sled, jer je očekivano da se ne može pozvati nešto što prethodno nije deklarisano. Da li je stvarno tako? Odgovor na ovo pitanje daje sledeći primer:

|  |  |
| --- | --- |
| 1  2  3  4  5 | console.log(myFunction(6));    function myFunction(x) {      return x \* x;  } |

Ovoga puta, funkcija je pozvana pre nego što je definisana. Potpuno neočekivano, ovakav kod proizvodi sledeći rezultat:

*36*

Iako se deklaracija nalazi ispod poziva, kod funkcioniše bez problema. Zašto je to tako?  
Odgovor na ovo pitanje leži u specijalnoj funkcionalnosti JavaScript parsera, koji vrši pomeranje svih deklaracija na početak tekućeg opsega. Ovakva funkcionalnost ne odnosi se samo na funkcije već i na deklaracije promenljivih:

|  |  |
| --- | --- |
| 1  2 | console.log(x);  var x; |

Rezultat:

|  |  |
| --- | --- |
| 1 | Undefined |

Kao što se može videti, ne dobija se izuzetak, već vrednost undefined, s obzirom da promenljiva nije inicijalizovana.

|  |
| --- |
| **Napomena**    *Opisana funkcionalnost ne važi za promenljive koje su deklarisane i inicijalizovane.* |

**Funkcije su objekti**

U JavaScript jeziku funkcije su još jedna vrsta objekata, tako da se kreiranjem funkcije zapravo kreira objekat. Reč je  o objektu koji se zove Function i može se kreirati korišćenjem istoimenog konstruktora:

|  |  |
| --- | --- |
| 1 | var myFunction = new Function('a', 'b', 'return a + b'); |

Ovoga puta, funkcija je kreirana u objektom maniru, korišćenjem konstruktora i ključne reči new. Konstruktoru su prosleđeni nazivi promenljivih, koji će predstavljati ulazne parametre funkcije i kod koji je potrebno izvršiti kada se funkcija pozove. Prva dva parametra se odnose na nazive parametara funkcije, a poslednji parametar na telo funkcije. Ovakvu funkciju je moguće pozvati na sledeći način:

|  |  |
| --- | --- |
| 1 | myFunction(10, 15); |

Naravno, povratna vrednost funkcije će biti 25.

Kreiranje funkcija korišćenjem konstruktora i bez korišćenja konstruktora proizvodi identičan efekat kada se govori o logici funkcije. Ipak, funkcije koje se kreiraju korišćenjem objektnog konstruktora parsiraju se kada se funkcija kreira.

Takođe, korišćenjem konstruktora za kreiranje funkcija, nije moguće funkciju deklarisati kao closure. Jednostavno, na ovaj način kreirane funkcije uvek se smeštaju na globalni nivo. Tako se iz njih može pristupiti samo promenljivama njihovog i globalnog sklopa.