Univerzitet Singidunum
www.singidunum.ac.rs

Funkcionalno programiranje

Školska 2024/25 godina

Zimski semestar

Tema 2: Funkcije u programskom jeziku JavaScript – napredno korišćenje

Sadržaj

- O prosleđivanju podataka - da malo sistematizujemo
- Još o streličastoj sintaksi
- Povratni poziv
- Parcijalna aplikacija i kuriranje

O prosleđivanju podataka: da malo sistematizujemo

Prosleđivanje podataka: sadržaj

- Prametri/argumenti
- Objekat `arguments` i `globalni objekat`
- Funkcijski objekat, imenovani funkcijski izraz
- `new function` konstrukt

Prosleđivanje podataka funkciji: parametri/argumenti

- Funkcija prima ulazne podatke putem parametara/argumenata
  - Parametri se navode u deklaraciji funkcije
  - Argumenti se navode pri pozivanju funkcije
- Argumenti se funkciji mogu prosleđivati na dva osnovna načina
  - **Po vrednosti**: pravi se lokalna kopija vrednosti u dosegu funkcije (lokalnoj memoriji funkcije) i funkcija radi sa tom kopijom ne "dirajući" originalnu vrednost.
  - **Po referenci**: funkciji se prosleđuje referenca na vrednost koja (može da) se nalazi u dosegu van funkcije (recimo, globalnom) i funkcija radi sa tom originalnom vrednošću.
- Prosleđivanje vrednosti funkciji u JavaScript-u
  - JavaScript prosleđuje argumente **po vrednosti** za **primitivne tipove** (String, Numeric. Boolean), ali se **neprimitivni tipovi (objekti)** prosleđuju **po referenci**.
  - JavaScript je vrlo fleksibilan po pitanju broja i tipa parametara i argumenata.

Fleksibilnost parametri/argumenti (broj): situacije i posledice

- Fleksibilnost u odnosu broja parametara i argumenata rezultuje sledećim situacijama:
  - Da se prosledi **baš onoliko argumenata koliko ima parametara** – ovde je sve “po pravilima službe”
  - Da se prosledi **više argumenta nego što ima parametara** – ovde se nešto mora uraditi sa “viškom” argumenata
  - Da se prosledi **manje argumenata nego što ima parametara** – ovde se mora rešiti pitanje “manjka” argumenata
  - Da treba prosleđivati različit broj argumenata pri različitim pozivima – ovde se mora smisliti **nešto** što će u sebi da čuva **onoliko argumenat** koliko je **potrebno za svaki pojedinačni poziv**.

Prosleđivanje podataka: višak argumenata

- **Argumenti koji su "višak" se, prosto, IGNORIŠU.**
- Primer:

```
function sum(a, b) {// Funkcija ima 2 parametra a i b
  return a + b;
}
// Prosleđeno je 5 argumenata 1, 2, 3, 4, 5:
alert( sum(1, 2, 3, 4, 5) ); /* vratiće vrednost 3
(3=1+2), a argumenti 3, 4, i 5 će biti ignorisani */
```

Prosleđivanje podataka: manjak argumenata

- Ako argument nije prosleđen niti mu je na bilo koji drugi način dodeljena vrednost, njegova podrazumevana vrednost je **`undefined`**.
- Podrazumevani argumenti
  - Mogu se i eksplicitno zadati podrazumevane vrednosti parametara/podrazumevani argumenti koje se koriste u slučaju da argument nije prosleđen. Sintaksa je:

```
imeParametra = podrazumevana_vrednost
```

- Primer

```
function showMessage(from, text = "Nema teksta") {
  alert( from + ": " + text );
}

showMessage("Marija"); // Marija: nema teksta
showMessage("Petar", "ima teksta" ); // Petar:
                                     // ima teksta
```

Prosleđivanje podataka: varijabilan broj argumenata - `rest` sintaksa$_{1/3}$

- Korišćenjem konstrukta **`...imeNiza`** funkciji se može proslediti proizvoljan broj argumenata:

```
function sumAll(...args) { // args je ime niza
  let sum = 0;

  for (let arg of args) sum += arg;

  return sum;
}

alert( sumAll(1) ); // Vraća 1
alert( sumAll(1, 2) ); // Vraća 3
alert( sumAll(1, 2, 3) ); // Vraća 6
```

- `rest` parametri se koriste za kreiranje funkcije koja prima proizvoljan broj argumenata.

Prosleđivanje podataka: varijabilan broj argumenata - `rest` sintaksa 2/3

- Dozvoljeno je i “mešanje” ; u sledećem primeru prva dva argumenta su varijable, ostali se prosleđuju putem niza:

```
function showName(firstName, lastName, ...titles) {
  console.log( firstName, ' ', lastName ); // Julius
Caesar

  // Ostali idu u niz
  // t.j. titles = ["Consul", "Imperator"]
 console.log( titles.length ); // 2

 console.log( titles[0] ); // Consul
 console.log( titles[1] ); // Imperator
}

showName("Julius", "Caesar", "Consul", "Imperator");
```

Prosleđivanje podataka: varijabilan broj argumenata - `rest` sintaksa 3/3

- **Ograničenje**: `rest` parametri moraju da budu **na kraju liste parametara**:
  `function f(arg1, ...rest, arg2)` – neispravno
  `function f(arg1, arg2, ...rest)` - ispravno
- Neispravno:

```
function showName(firstName, ...titles, lastName,) {
  console.log( firstName, ' ', lastName ); // Julius Caesar

  // Ostali idu u niz
  // t.j. titles = ["Consul", "Imperator"]
 console.log( titles.length ); // 2

 console.log( titles[0] ); // Consul
 console.log( titles[1] ); // Imperator
}

showName("Julius", "Consul", "Imperator ", "Caesar");
```

- Vraća grešku: **Uncaught SyntaxError: Rest parameter must be last formal parameter**

Prosleđivanje podataka: varijabilan broj argumenata - **spread** sintaksa 1/2

- Sintaksa **rest** parametara omogućuje da pojedinačne argumente "spakujemo" u niz. Primer:

```
function saberiNiz (...ulazniNiz) {
     return ulazniNiz[0] + ulazniNiz[1]
}
let suma = saberiNiz (5,18) // ulazniNiz[0]=5, ulazniNiz[1]=18
console.log(suma) => 23
```

- Potrebno nam je i obrnuto: da od niza (elemenata niza) "napravimo" pojedinačne argumente: Primer:

```
function saberiVrednosti (v1, v2) {
    return v1 + v2
}
let mojNiz = [5, 18]
let suma = saberiVrednosti (mojNiz) // =>  v1=[5, 18]  v2 = undefined ???
console.log(suma) // => 5,18undefined ???

suma = saberiVrednosti (...mojNiz) // v1=5, v2=18 !!!
console.log(suma) // => 23 !!!
```

- **spread** sintaksa se koristi da se funkciji koja po definiciji zahteva pojedinačne argumente ti argumenti proslede u obliku niza.

Prosleđivanje podataka: varijabilan broj argumenata - **spread** sintaksa 2/2

- Kada se **...ime** navede u pozivu funkcije, ona "razvija" iterabilni objekat **ime** u listu argumenata.
- Na taj način može se prosleđivati više iterabilnih objekata:

```
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];

alert( Math.max(...arr1, ...arr2) ); // Vraća: 8
```

- Mogu se kombinovati **spread** i "normalne" vrednosti:

```
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];

alert( Math.max(1, ...arr1, 2, ...arr2, 25) ); // Vraća: 25
let merged = [0, ...arr1, 2, ...arr2]; // Šta će da vrati?
```

Prosleđivanje varijabilnog boja podataka: opcije za pretvaranje u niz

- **Opcija 1**: Sintaksa spread :

```
let str = "Hello";
// spread sintaksa
alert( [...str] ); // Ispis: H,e,l,l,o
```

- **Opcija 2**: `Array.from(str)`:

```
// Array.from ()
alert( Array.from(str) ); // Ispis: H,e,l,l,o
```

- Ali postoji suptilna razlikaizmeđu `Array.from(obj)` i `[...obj]`:
  - `Array.from` radi i nad nizolikim i nad iterabilnim.
  - `spread` sintaksa radi samo nad iterabilnim.
- Dakle, bolje je da za pretvaranje nečega u `array`, koristite `Array.from` (jer je univerzalniji).

Prosleđivanje podataka: objekat `arguments`

- U JavaScript-u postoji i specijalni nizoliki objekat koji se zove **`arguments`** koji sadrži sve argumente uređene po indeksu.
- U starijim verzijama jezika nije bilo **`rest`** sintakse i jedini način za prosleđivanje proizvoljnog broja argumenata funkciji bio je objekat **`arguments`** koga još uvek ima u starijem kodu (i to radi).
- Nedostatak ovoga objekta je u sledećem:
  - on nije tipa **`array;`** iako liči na niz i iterabilan je, ne podržava **`array`** metode kao što je n.pr., **`arguments.map()`**.
  - Ne dozvoljava "mešanje" sa pojedinačnim argumentima; u stvari, mora da sadrži sve argumente u sebi.
  - Streličaste funkcije NEMAJU objekat **`arguments`**.
- Dakle, ako nam je potrebno ovakvo upravljanje prosleđivanjem argumenata, bolje je koristiti **`rest/spread`** sintaksu.

Prosleđivanje podataka: Globalni objekat 1/4

- Globalni objekat obezbeđuje (čuva u sebi) varijable i funkcije koje su svuda/svima dostupne. Ugrađen je u jezik ili okruženje.
- Globalni objekat u JavaScript okruženju:
  - U brauzeru se ovaj objekat zove **`window`**, za node.js ime je **`global`**, druga okruženja (mogu da) koriste i drugačija imena.
  - Odnedavno je jeziku JS dodat **`globalThis`** kao standardizovano ime za globalni objekat, ali to još nije univerzalno podržano. U svakom slučaju, može se lako napraviti odgovarajući polifil.
  - Ukoliko se ne zna sigurno u kom će se okruženju izvršavati skript, najbolje je koristiti ime **`globalThis`**.

Prosleđivanje podataka: Globalni objekat$_{2/4}$

- **Svim svojstvima** globalnog objekta **može se pristupiti direktno**:

```
alert("Zdravo");
// je isto što i
window.alert("Zdravo");
globalThis.alert ("Zdravo");
```

- Varijable deklarisane sa **`var` postaju globalno dostupne**, ali **to nije slučaj sa `let`**:

```
var gVar = 5;
alert(window.gVar); // 5 (postala je svojstvo
                    // globalnog objekta)
let gLet = 5;
alert(window.gLet); // undefined (nije postala
                    // svojstvo globalnog objekta)
```

Prosleđivanje podataka: Globalni objekat$_{3/4}$

```
// Učini  informaciju o tekućem korisniku globalnom,
// tako da svi skriptovi mogu da joj pristupe jer lokalne varijable sa
  istim imenom “zaklanjaju” globalne varijable

window.tekuciKorisnik = {
  ime: "Jovan"
};

/* Ako postoji lokalna varijablu sa imenom "tekuciKorisnik */
function ispisiIme (){
  let tekuciKorisnik = { // lokalna varijabla
    ime: "Petar"
    };
   console.log (tekuciKorisnik.ime) // Petar
/* ona je zaklonila globalnu varijablu pa globalna varijabla mora da se
  uzme eksplicitno (bezbedno!) iz objekta window */
   console.log (window.tekuciKorisnik.ime) // Jovan
   }
ispisiIme();
```

Napredno korišćenje funkcija: objekat `Function`$_{1/2}$

- Funkcija u JavaScript-u je vrednost
  - Svaka vrednost ima svoj tip
  - Kojeg je tipa funkcija?
- U JavaScript-u, funkcija je `Function` objekat:

```
(function(){}).constructor === Function // => true
```

- Funkcije su posebni objekti *akcionog tipa* – mogu se pozivati
- Ali imaju i druge karakteristike objekata
  - Imaju svojstva koja im se mogu dodavata/uklanjati
  - Prosleđuju se po referenci, itd.

Napredno korišćenje funkcija: Svojstva objekta `Function` 2/2

- Objekat `Function` nema sopstvenih svojstava i metoda .
- Kako je sam funkcija, on nasleđuje neke (ugrađene) metode i svojstva putem prototipskog lanca od objekta `Function.prototype`.
- Ovde ćemo objasniti ona koja se najčešće koriste:
  - **`Function.prototype.name`**
  - **`Function.prototype.length`**

Napredno korišćenje funkcija: svojstvo `Function.prototype.name`

- Sadrži **ime** funkcije
  - Deklaracija sa naredbom `function` vratiće ime koje sledi ključnu reč `function` iz deklaracije

```
function kaziZdravo() {
  console.log("Zdravo!");
}
console.log(kaziZdravo.name); // Ispis: kaziZdravo
```

  - Neimenovani funkcijski izraz vratiće ime varijable sa leve strane znaka jednakosti u naredbi dodele:

```
let kaziZdravo1 = function() {// nema imena
 console.log("Zdravo!");
};
console.log(kaziZdravo1.name); // Ispis: kaziZdravo1
```

Napredno korišćenje funkcija: svojstvo `Function.prototype.name`

- Imenovani funkcijski izraz

```
let kaziZdravo1 = function kaziZdravo2() { console.log("Zdravo!");
  console.log(kaziZdravo1.name); // Ispis: kaziZdravo2
};

  kaziZdravo1()
  console.log(kaziZdravo1.name) // Ispis: kaziZdravo2
```

- Dodela putem podrazumevane vrednosti

```
function f(kaziZdravo2 = function() {}) {
 console.log(kaziZdravo2.name); /* Ispis: kaziZdravo2
  (ovo znači da funkcija deklarisana kao podrazumevana
  vrednost parametra kaziZdravo2 dobija isto ime kao i
  ime parametra)*/
}

f();
```

I metode objekta imaju ime

```
let user = {
  kaziZdravo(){
    // ...
  },
  kaziCiao: function() {
    // ...
  },
  kaziPrivjot: function kaziZdravo1() {
    // ...
  }
}
   alert(user.kaziZdravo.name); //=> kaziZdravo
   alert(user.kaziCiao.name); //=> kaziCiao
   alert(user.kaziPrivjot.name); //=> kaziZdravo1
```

Ali ima i situacija kada se ime ne može odrediti

- Funkcija definisana unutar niza

```
let arr = [function() {},function mojaFunkcija() {} ];

/* <prazan string>, endžin nema načina da postavi ispravno ime, onda
  imena nema */
console.log ('tip od arr[0]: ',
              typeof(arr[0])) //=> tip od arr[0]: function
console.log('a ime od arr[0]: ',
             arr[0].name ); => a ime od arr[0]:

/* mojaFunkcija, endžin ima načina da postavi ispravno ime, onda
  imena ima */
console.log ('tip od arr[1]: ',
              typeof(arr[1])) //=> tip od arr[1]: function
console.log('a ime od arr[1]: ',
             arr[1].name ); //=> a ime od arr[1]: mojaFunkcija
```

Napredno korišćenje funkcija: svojstvo `Function.prototype.length`

- Sadrži podatak o broju parametara funkcije

```
function f1(a) {}
function f2(a, b) {}
function many(a, b, ...more) {}

alert(f1.length); // Ispis: 1
alert(f2.length); // Ispis: 2
alert(many.length); // Ispis: 2 (rest
                    // parametri se ne
                    // broje
```

Svojstvo `Function.prototype.length`: jedno često korišćenje

- Za introspekciju u funkcijama koje operišu nad drugim funkcijama.
- U računarstvu, ***introspekcija tipa*** je sposobnost ispitivanja tipa ili svojstava objekta u vreme izvršenja programa

Svojstvo `length`: primer korišćenje

- Zadatak:

Napraviti funkciju **`pitaj`** koja prima argument **`pitanje`** kao pitanje i proizvoljan broj hendlerskih funkcija koje će se pozivati (po potrebi). Kada korisnik da svoj odgovor, funkcija poziva hendlere. Mogu se proslediti dve vrste hendlera:

- Funkcija bez argumenata koja se poziva samo ako odgovor korisnika predstavlja potvrdu.
- Funkcija sa argumentima koja se poziva u svakom slučaju i zahteva potvrdu.

Svojstvo `length`: ideja rešenja za primer korišćenja

- Funkcija `pitaj()` je funkcija sa dva parametra
  - Parametra `pitanje` kojim se prosleđuje tekst pitanja. Ovaj prametar može da bude primitivni tip `String`.
  - Parametra `hendleri` kojim se prosleđuje **lista** funkcija (hendlera). Pošto je broj hendlera proizvoljan, njega ćemo predstaviti nizom `hendleri` korišćenjem rest sintakse.
- Način pozivanja hendlera (potvrda ili hendler koji zahteva potvrdu) odredićemo ispitivanjem svojstva **`length`** svakog prosleđenog hendlera na sledeći način:
  - Ako to svojstvo ima vrednost 0, znači da je u pitanju funkcija bez argumenata, odnosno hendler kojim se reaguje na zahtev za potvrdu.
  - Ako to svojstvo ima vrednost različitu od 0, znači da je u pitanju funkcija sa argumenatima, odnosno hendler koji zahteva potvrdu.

Svojstvo `length`: rešenje primera korišćenja - kod

```
function pitaj(pitanje, ...hendleri) {
  let jePotvrda = confirm(pitanje);
  for(let hendler of hendleri) {
    if (hendler.length == 0) {
      if (jePotvrda) hendler(); // hendler bez argumenata
    } else {
      hendler(jePotvrda);
    }
  }
}
// za potvrdan odgovor se pozivaju oba hendlera
// za negativan odgovor se poziva samo drugi hendler
pitaj("Pitanje?",() => alert('Potvrda'), result => alert(result));
```

Dodavanje svojstava `Function` objektu

- Objektu `Function` mogu se dodati svojstva
- Na primer, ako želimo da brojimo pozive funkcije, to može da se uradi dodavanjem svojstva **`brojac`** objektu **`kaziZdravo`** koji je tipa `Function`:

```
function kaziZdravo() {
  console.log("Zdravo!");
  // hajde da brojimo koliko se puta izvršavamo
 kaziZdravo.brojac++; // ažurira vrednost svojstva brojac
}
console.log( `Tip objekta kaziZdravo: `, typeof(kaziZdravo) ); /* => Tip objekta kaziZdravo: function */
kaziZdravo.brojac = 0; // postavlja početnu vrednost svojstva brojac
kaziZdravo(); // Ispis: Zdravo!
kaziZdravo(); // Ispis: Zdravo!
console.log( `Pozvao ${kaziZdravo.brojac} puta` ); // => Pozvao 2 puta
```

Svojstvo NIJE varijabla

```
function kaziZdravo() {
  console.log("Zdravo!");
  // hajde da brojimo koliko se puta izvršavamo
 kaziZdravo.brojac++; // ažurira vrednost svojstva
}

let brojac = 0; // postavlja početnu vrednost varijable
console.log(`brojac = `, brojac) //=> 0
console.log(`kaziZdravo.brojac = `, kaziZdravo.brojac)// => undfined

kaziZdravo(); // Ispis: Zdravo!
kaziZdravo(); // Ispis: Zdravo!

console.log( `Pozvao ${kaziZdravo.brojac} puta` ); // Ispis: Pozvao NaN puta
```

Svojstva ponekad mogu da zamenu zatvaranja

- Ovde je **broj** uskladišten direktno u funkciju **brojac**, ne u njen spoljašnji leksički doseg.

```
function napraviBrojac() {
// umesto inicijalizacije
// let broj = 0
  function brojac() {
    return brojac.broj++;
  };
  brojac.broj = 0; // inicijalizuje se ovde
  return brojac;
}

let brojac = napraviBrojac();
console.log( brojac() ); // Ispis: 0
console.log( brojac() ); // Ispis: 1
console.log( brojac() ); // Ispis: ??
```

Da li je to baš prava zamena ili nije?

- Glavna razlika u odnosu na pravo zatvaranje je što eksterni kod ima pristup svojstvu funkcijskog objekta `brojac()` u ovom slučaju, pa je moguće i sledeće:

```
function napraviBrojac() {
  function brojac() {
    return brojac.broj++;
  };
  brojac.broj = 0;
  return brojac;
}

let brojac = napraviBrojac();
console.log( brojac() ); // => 0
brojac.broj = 10;
console.log( brojac() ); // => 10
```

Funkcijski objekat: `new Function` konstrukt

- Ovo je još jedan način za kreiranje funkcije
- Ne koristi se previše često, ali po nekad ne može drugačije
- Glavna razlika od drugih načina koje smo videli je u tome što se **funkcija kreira doslovce iz stringa koji se prosleđuje u vreme izvršavanja**.
  - Sve prethodno viđene deklaracije zahtevaju da kod funkcije bude zapisan u okviru skripta.
  - `new Function` omogućuje da se svaki string “pretoči” u funkciju.

`new Function` konstrukt: sintaksa

- Sintaksa je sledeća:

```
let func = new Function ([arg1, arg2, ...argN], functionBody);
```

- Primer 1:

```
let sum = new Function('a', 'b', 'return a + b’);

console.log( sum(1, 2) ); // Ispis: 3
```

- Primer 2:

```
let str = 'console.log ("Neko mi je ovo poslao")' /* Primljeno
  dinamički odnekud */

let func = new Function(str);

func(); // ispis: Neko mi je ovo poslao
```

`new Function`: Zatvaranje

- Videli smo da funkcija obično pamti gde je "rođena". To se memoriše u specijalnom svojstvu `[[Environment]]`.
- A sama vrednost je pokazivač na leksički doseg u kome je funkcija kreirana.
- Međutim, kada se funkcija kreira pomoću `new Function`, njeno `[[Environment]]` svojstvo se **ne postavlja na tekući leksički doseg**, već **se postavlja na globalni doseg**.
- To ima i loše i dobre strane:
  - **Loše:** Takva funkcija nema pristup varijablama roditeljske funkcije, već samo globalnim varijablama. To može da pravi problem zbog korišćenja minifikatora (programi koji "komprimuju" JS kod tako što vrše izbacivanje suvišnih stvari u šta spada i preimenovanje lokalnih varijabli u "kratka" imena).
  - **Dobro:** Ova karakteristika funkcije je korisna u praksi jer zahteva da se za prosleđivanje nečega funkciji kreiranoj pomoću `new Function` eksplicitno koriste njeni argumenti a ne zatvaranja, što je arhitekturalno mnogo čistije.

Funkcije: Još malo o streličastoj sintaksi

Neke važne specifičnosti streličaste sintakse

- Streličasata sintaksa je nesporno vrlo zgodna za korišćenje i ima prednosti nad kanoničkom sintaksom (naredba `function`, funkcijski izraz).
- Međutim, streličasta sintaksa ima specifičnosti koje valja imati na umu pri njenom korišćenju:
  - Streličaste funkcije nemaju `this`.
  - Streličaste funkcije ne mogu se koristiti kao metode.
  - Streličaste funkcije ne mogu se koristiti kao konstruktori.
  - Streličaste funkcije nemaju varijablu `arguments`.

Streličaste funkcije NEMAJU **this** 1/3

```
/* Ovde je anonimna funkcija deklarisana naredbom function i ima svoje this
ali ono ima vrednost undefined*/
"use strict";
let group = {
  title: "Naša grupa",
  students: ["Jova", "Pera", "Vera"],
  showList() {
  console.log ('pokazivač this za showList():', this)
    this.students.forEach(function(student) {
      console.log(this.title, ': ', student)
    });
  }
};
console.log ('pokazivač this:', this)
let hocuPoziv = prompt ('da li želite da pozovete funkciju showList()?',
true)
if (hocuPoziv)
group.showList(); // ne radi
```

Streličaste funkcije NEMAJU `this` 2/3

- Ako se koristi this, ono se uzima spolja (`this.title` u primeru je isti kao i `tihs` spoljašnje metode `showList`, dakle - `group.title`):

```
"use strict";
let group = {
  title: "Naša grupa:",
  students: ["Jova", "Pera", "Vera"],
  showList() {
/* uzima this spolja (iz showList() gde je definisana, dakle:
group.title, group.students  */
      this.students.forEach(student => console.log(this.title,
                                          ":", student)
    );
  }
};

group.showList();/* => Naša grupa: Jova,
                      Naša grupa: Pera,
                      Naša grupa: Vera */
```

Streličaste funkcije NEMAJU **this** 3/3

```
const module = {
  x: 42,
  getX: function() {
    return this.x;
  }
}

const unboundGetX = module.getX;
console.log('poziv unbound: ' + unboundGetX()); /* => undefined jer
                                                    nema varijable x u
                                                    globalnom dosegu */

const boundGetX = unboundGetX.bind(module); /* pozivom bind ()
                                               ograničavamo x na module
                                               gde x postoji */
console.log('poziv bound: ' + boundGetX()); // Očekivani izlaz: 42
```

Streličaste funkcije NE MOGU se koristiti kao metode jer nemaju **this**

```
"use strict";
const obj = {
  i: 10,
  b: () => console.log(this.i, this), /* ovo this pokazuje na globalni
  objekat jer je pruzeto od objekta obj koji je definisan u
                                        globalnom objektu*/
  c() {
    console.log(this.i, this); /* ovo this pokazuje na obj objekat jer je
                                  pruzeto od objekta c() koji je definisan
                                  u objektu obj*/
  },
};

/* this.i je undefined, this pokazuje na globalni objekat (Window) */
obj.b();
/* this.i je 10, jer ovo this pokazuje na objekat obj */
obj.c();
```

Streličaste funkcije ne mogu se koristiti kao konstruktori

- Streličaste funkcije **ne mogu se koristiti kao konstruktori**, odnosno **ne mogu se pozvati sa new**.
- One nemaju ni `prototype` svojstvo.
- Ilustracija:

```
const Foo = () => {};

console.log("prototype" in Foo); // false

const foo = new Foo(); /* TypeError: Foo is not a
constructor */
```

Streličaste funkcije nemaju svojstvo/objekat `arguments`

- Objekat `arguments` je nizoliki objekat dostupan unutar funkcije koji sadrži vrednosti argumenata prosleđenih toj funkciji.
- Streličaste funkcije nemaju sopstveni `arguments` objekat nego ga preuzimaju iz roditeljskog dosega:

```
function foo(n) {
  const f = () => arguments[0] + n; /* ovde je arguments
                                       objekat koji sadrži
                                       argumente funkcije
                                       foo, odnosno
                                       arguments[0] je n */
  return f();
}

foo(3); // 3 + 3 = 6
```

Streličaste funkcije nemaju svojstvo/objekat `arguments`

- Varijabla `arguments` je zgodna za dekoratorski obrazac kada treba proslediti poziv sa tekućim `this` i `arguments`.
- Na primer, `defer(f, ms)` prima funkciju `f` i vraća omotač (zapisan u streličastoj sintaksi) oko nje koji će odložiti njeno izvršavanje za `ms` milisekundi:

```
function defer(f, ms) {
  return function() {
    setTimeout(() => f.apply(this, arguments), ms)
  };
}
function kaziZdravo(kome) {
  console.log('Zdravo, ' + kome);
}
let kaziZdravoOdlozeno = defer(kaziZdravo, 5000);
kaziZdravoOdlozeno("Jovo"); // Nakon 5 sec
ispis:Zdravo, Jovo
```

Funkcije: Povratni poziv

Povratni poziv: sadržaj

- Šta je povratni poziv
- Jednostavni povratni pozivi
  - Funkcija
  - Funkcijski izraz
  - Poziv sa parametrima
- Vremensko raspoređivanje izvršavanja
  - Ugrađene funkcije
  - Gubljenje `this` pokazivača
  - Povratni poziv i obrazac Dekorator

Šta je povratni poziv$_{1/2}$

- U računarskom programiranju, ***povratni poziv*** ili ***naknadni poziv*** (engl. *callback*, *call-after*) je bilo koja referenca na **izvršiv kod** koja se **prosleđuje kao argument drugom kodu** od koga se očekuje da u nekom trenutku (odmah – sinhrono, odloženo - asinhrono) ***pozove*** **(u stvari, *izvrši*) kod na koji pokazuje referenca**.
- Nije isto ako se funkcija “direktno” pozove unutar druge funkcije (na primer, navođenjem imena funkcije iz koga sledi par malih zagrada) i ako se poziv izvrši prosleđivanjem reference putem argumenta

Šta je povratni poziv$_{2/2}$

- "Direktni" pozivi funkcija `dva()` i `tri()` unutar funkcije `jedan()`:

```
function jedan (){
    console.log('jedan')
    dva()
    tri()}
function dva (){console.log('dva');}
function tri (){console.log('tri')}
```

- Pozivanje funkcija `dva()` i `tri()` unutar funkcije `jedan()` prosleđivanjem reference putem argumenta:

```
function jedan (fn, fn1){
    console.log('jedan')
    fn()
    fn1()}

    jedan(dva, tri)
```

Funkcija povratnog poziva: Primer 1 - zadatak

- Napisati funkciju `ask(question, yes, no)` sa tri parametra:
  - `question` – tekst pitanja
  - `yes` – funkcija koja će da se izvrši ako je odgovor “Yes”
  - `no` - funkcija koja će da se izvrši ako je odgovor “No”
- Funkcija treba da postavi pitanje i da, u zavisnosti od odgovora korisnika, pozove funkciju `yes()` ili `no()`

Funkcija povratnog poziva: Primer 1 – ideja rešenja

- Definisaćemo funkcije **`showOk`** i **`showCancel`** koje se **prosleđuju kao argumenti** funkciji **`ask`** da ih ona kasnije izvrši.
- Ideja je da se funkcija prosledi i da se očekuje da bude "povratno pozvana" kasnije, ako bude potrebno
- U primeru, **`showOk`** će se izvršiti za "yes" odgovor, a **`showCancel`** za odgovor "no".
- Argumenti **`showOk`** i **`showCancel`** funkcije **`ask`** su ***funkcije povratnog poziva*** (***povratni pozivi***).

Funkcija povratnog poziva: Primer 1 – rešenje sa **function** naredbom

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no(); }

function showOk() {
   console.log("Saglasili ste se." );
}
function showCancel() {
 console.log("Otkazali ste." );
}

/* Korišćenje: funkcije showOk, showCancel se prosleđuju
  kao argumenti funkciji ask */

ask("Da li ste saglasni?", showOk, showCancel);
```

Funkcija povratnog poziva: Primer 1 – rešenje sa funkcijskim izrazom

```
function ask(question, yes, no) {
    if (confirm(question)) yes()
   else no(); }
  ask("Da li ste saglasni?",
  function() {console.log("Saglasili ste se."); },
  function() {console.log("Otkazali ste."); } );
```

Funkcije povratnog poziva: Primer funkcije povratnog poziva sa parametrom

```
function pozdrav(ime) {
  console.log(' Zdravo ' + ime);
}

function obradiUlaz(funPP) {
  var ime = prompt('Molim, unesite ime');
 funPP(ime); // povratni poziv funkcije funPP
}
// poziv funkcije obradiUlaz
obradiUlaz(pozdrav); /* pozdrav je funkcija koja
  će biti povratno pozvana u funkciji obradiUlaz
  */
```

Funkcije povratnog poziva: **vremensko raspoređivanje izvršavanja**

- Moguće je funkciju izvršiti odloženo – ne odmah, već nakon određenog vremena.
- To se zove ***raspoređivanje poziva*** (“scheduling a call”).
- Za to u JavaScript-u postoje dve funkcije:
  - `setTimeout` omogućuje da se funkcija pokrene jednom nakon zadatog vremenskog intervala.
  - `setInterval` omogućuje da se funkcija izvršava ponovljeno počevši od nekog vremenskog intervala i kontinuirano ponavljajući izvršavanje sa učestalošću određenom tim vremenskim intervalom sve dok se izvršavanje eksplicitno ne zaustavi.
- Ove metode nisu deo JavaScript specifikacije, ali većina implementacija ima interni raspoređivač i pruža te metode. Posebno, metode su podržane u baruzerima i okruženju `Node.js`.

Funkcija `setTimeout`

- Sintaksa

```
let timerId = setTimeout(func|code, [delay], [arg1], [arg2], ...)
```

- Parametri:
  - `func|code` - Funkcija ili string koda za izvršavanje. Obično je to funkcija. Iz istorijskih razloga može se proslediti i string koda, ali se to ne preporučuje.
  - `delay` - vreme do pokretanja, u milisekundama; pretpostavljena vrednost 0 (ako je 0, ne znači da će se baš odmah izvršiti).
  - `arg1, arg2, ...` - Argumenti funkcije `func`

`setTimeout`: primer

```
function kaziZdravo(fraza, ko) {
  console.log( fraza, ' ', ko );
}

let timerId1 = setTimeout(kaziZdravo, 1000,
 "Zdaravo", "Pero"); // Zdravo Pero
```

Funkcija `setInterval`

- Sintaksa

```
let timerId = setInterval(func|code, [delay], [arg1], [arg2], ...)
```

- Parametri:
  - `func|code` - Funkcija ili string koda za izvršavanje. Obično je to funkcija. Iz istorijskih razloga može se proslediti i string koda, ali se to ne preporučuje.
  - `delay` - vreme do pokretanja, u milisekundama, pretpostavljena vrednost 0.
  - `arg1, arg2, ...` - Argumenti funkcije `func()`.

- `clearInterval(timerId)` - zaustavlja dalje pozive

Funkcija **setInterval** primer

```
// ponavljaj svake sekunde
let timerId = setInterval(() =>
console.log('tik-tak'), 1000);


// Nakon 5 sekundi prestani
setTimeout(() => {
clearInterval(timerId);
console.log('Dosta je bilo!'); }, 5000);
```

Prosleđivanje povratnih poziva: povezivanje – gubljenje `this`

- Pri prosleđivanju metoda objekta kao povratnih poziva (n.pr. funkciji `setTimeout`) javlja se problem ***gubljenja pokazivača*** **`this`** ("*losing* `this`“).
- Primer:

```
let user = {
  firstName: "Jovo",
  kaziZdravo() {
    alert(`Zdravo ${this.firstName}!`);
  }
};
/* Ne radi jer je f van konteksta user, this pokazuje na
  window u kome nema svojstva firstName, ono je u objektu
  user*/
let f = user.kaziZdravo;
setTimeout(f, 5000); // Zdravo undefined!
```

Povezivanje: Zašto se gubi `this`

- Metoda `setTimeout` u brauzeru je malo specijalna: ona postavlja `this` = `window` za poziv funkcije(za Node.js, to postaje `timer` objekat).
- Dakle, za `this.firstName` pokušava se dobavljanje `window.firstName`, što ne postoji pa `this` (uobičajeno u ovakvim slučajevima) postaje `undefined`.
- Zadatak je tipičan – želimo da prosledimo metodu objekta negde drugde (ovde – raspoređivaču) gde će ona biti pozvana. Kako obezbediti da se ona pozove u pravom kontekstu?

Gubljenje `this`: rešenja

- Rešenje 1: Wraper – omotačka funkcija koja prima kontekst (željeni objekat) i zatim normalno poziva metodu tog objekta.
- Rešenje 2: `bind` metoda – funkcije imaju ugrađenu metodu koja se zove `bind` i koja rešava ovaj problem

Rešenje 1: Omotačka funkcija

```
let user = {
  firstName: "Jovo",
  kaziZdravo() {
    console.log(`Zdravo, ${this.firstName}!`);
  }
};

setTimeout(function() { // omotačka funkcija
  user.kaziZdravo();}, 5000); // Zdravo, Jovo!
```

Rešenje 1: potencijalni problem

- Šta ako se pre trigerovanja funkcije `setTimeout` (a ima **3** sekunde kašnjenja!) promeni vrednost `user`?

```
let user = {
  firstName: "Jovo",
  kaziZdravo() {
    console.log(`Zdravo, ${this.firstName}!`);
  }
};
user.kaziZdravo () // Zdravo Jovo
setTimeout(() => user.kaziZdravo(), 3000);
// ...vrednost promenljive user se promeni u toku te tri sekunde
user.firstName = "Drugi korisnik u setTimeout!";
// Ispis: Zdravo Drugi korisnik u setTimeout!
```

Rešenje 2: Ugrađena metoda **bind** 1/2

```
let user = {
  firstName: "Jova"
};

function func() {
  console.log(this.firstName);
}


// this → window
func(); // => undefined
```

```
let user = {
  firstName: "Jova"
};

function func() {
 console.log(this.firstName);
}

let funcUser = func.bind(user);
// this → user
funcUser(); // => Jova
```

Rešenje 2: Ugrađena metoda `bind` 2/2

- `func.bind(user)` je "bound varijanta" od `func`, gde `this` pokazuje na `user`.
- Svi **argumenti** se prosleđuju originalnoj funkciji `func` "kakvi jesu" :

```
let user = {
  firstName: "Jovo"
};
function func(phrase) {
  alert(phrase + ', ' + this.firstName);
}
// fiksiraj this na user
let funcUser = func.bind(user);

funcUser("Zdravo"); // => Zdravo, Jovo
/* argument "Zdravo" se prosleđuje; this pokazuje na user) */
```

Ugrađena metoda **bind** – slučaj objekat

```
let user = {
  firstName: "Jovo",
  kaziZdravo() {
    console.log(` Zdravo ${this.firstName}! `);
  }
};
let kaziZdravo = user.kaziZdravo.bind(user); // (*)
kaziZdravo(); // Zdravo, Jovo! - može da radi, zato što this pokazuje
              // gde treba (na user)

setTimeout(kaziZdravo, 1000); // Zdravo, Jovo!

/* čak i ako se vrednost user promeni u toku te sekunde čekanja,
kaziZdravo koristi prethodno uvezanu vrednost */
user = { // promenjena vrednost user
  kaziZdravo() {console.log("Drugi user u setTimeout!"); }
};
console.log(user.kaziZdravo)

/* Tek novi bind menja rezultat */
kaziZdravo = user.kaziZdravo.bind(user); // (**)
setTimeout(kaziZdravo, 3000); // Zdravo, Drugi user u setTimeout!
```

Ugrađena metoda `bind` – slučaj objekat sa više metoda

- Za objekat sa više metoda koje želimo da prosleđujemo naokolo, možemo vezivanje uraditi u petlji:

```
for (let key in user) {
  if (typeof user[key] == 'function') {
    user[key] = user[key].bind(user);
  }
}
```

Povratni poziv: Dekoratori

- *Dekorator* (*Decorator*) je specijalna funkcija koja **prima drugu funkciju** (dekorisana funkcija) i **menja ponašanje primljene funkcije**.
  - Pri tome, originalna funkcija ostaje neizmenjena, novo ponašanje implementira dekorator.
  - U stvari, **poziv dekoratora** *rezultuje* **povratnim pozivom dekorisane funkcije** plus **ono što implementira dekorator**.
  - Pri takvim pozivima, potrebno je drugoj (dekorisanoj) funkciji proslediti **sve argumente zajedno sa kontekstom**.

Primer dekoratora – keširanje 1/5

- Neka imamo funkciju `sporaFunkcija(x)` koja vrši zahtevna računanja koja troše puno procesorskog vremena.
- Da bi se uštedelo, strategija je izbegavanje nepotrebnih poziva tako što će se argumenti i povratne vrednosti skladištiti i, u slučaju poziva funkcije sa argumentima za koje u skladištu postoje sračunate povratne vrednosti, rezultat preuzimati iz skladišta umesto da se ponovo poziva funkcija.
  - Jedan način je da se ta strategija ugradi u samu funkciju `sporaFunkcija(x)`.
  - Drugi, bolji način je da se funkcija `sporaFunkcija(x)` dekoriše i da se u dekoratoru implementira strategija izbegavanja nepotrebnih poziva. Dekorator je u tom slučaju omotačka funkcija oko funkcije `sporaFunkcija(x)`.

Primer dekoratora – keširanje 2/5

```
}
function keshDekorator(func) {
  let kesh = new Map();
  return function(x) {
    if (kesh.has(x)) {    // ako ključ postoji u kešu
      return kesh.get(x); // vrati njegovu vrednost
    }
    let rezultat = func(x); // ako ključ ne postoji u kešu, pozovi
                            // dekorisanu funkciju

    kesh.set(x, rezultat);  // i zapiši rezultat u keš
    return rezultat;
  };
}
```

Primer dekoratora – keširanje 3/5

```
sporaFunkcija = keshDekorator(sporaFunkcija);

/* Pozivanje funkcije sporaFunkcija() za vrednosti argumenata 1 i 2 i
keširanje rezultata. */
sporaFunkcija(1);
sporaFunkcija(2);
console.log( "Uzeto iz keša: " + sporaFunkcija(1) ); // vrednost
sporaFunkcija(1) je preuzeta iz keša i vraćena.
console.log( "Uzeto iz keša: " + sporaFunkcija(2) ); // vrednost
sporaFunkcija(2) je preuzeta iz keša i vraćena.
```

Dekorator i metode objekta 1/4

- U slučaju primene dekoratora na metode objekta, potrebno je razrešiti problem obezbeđivanja ispravnog konteksta (ključna reč `this` i argumenti) za dekorisanu funkciju.
- JS ima dve ugrađene metode za tu namenu `call()` i `apply()`.
  - Obe metode rade istu stvar – pozivaju funkciju u zadatom kontekstu.
  - Razlika je u načinu zadavanja argumenata: metodi `call()` argumenti se prosleđuju pojedinačno, a metodi `apply()` kao niz.

Dekorator i metode objekta 2/4

```
let worker = {
  nekaMetoda() {
    return 1;
  },

sporaFunkcija(x) {
// Ovde se vrši masivno računanje
   console.log(`Ja nešto dugo računam za vrednost argumenta ${x}`);
          return x * this.nekaMetoda(); // (*);
   }
}
```

Dekorator i metode objekta 3/4

```
function keshDekorator(func) {
  let kesh = new Map();

  return function(x) {
    if (kesh.has(x)) {    // ako ključ postoji u kešu
      return kesh.get(x); // iščitaj rezultat iz njega
    }

    let rezultat = func.call(this, x); /* ako ključ ne postoji u kešu,
                                          pozovi funkciju sa korektnim
                                          "this" i argumentom x*/

    kesh.set(x, rezultat);  // i zapiši rezultat u keš
    return rezultat;
  };
}
```

Dekorator i metode objekta 4/4

```
// Dekoriši funkciju keširanjem
worker.sporaFunkcija = keshDekorator(worker.sporaFunkcija);

console.log( worker.sporaFunkcija(2) ); // računanje i keširanje
console.log( worker.sporaFunkcija(2) ); /* nema računanja - vađenje
                                           vrednosti iz keša */
```

Parcijalna aplikacija i kuriranje

- Ima još dve situacija koje su u vezi sa pozivanjem funkcije i veoma su važne za funkcionalno programiranje:
  - Parcijalna aplikacija funkcije,
  - Kurirane funkcije
- Ovim pitanjem ćemo se još detaljno baviti u toku kursa a sada samo da vrlo sažeto kažemo o čemu se radi:
  - Parcijalna aplikacija funkcije je situacija u kojoj se u pozivu funkcije zadaje samo deo njenih argumenata, a deo argumenata se zadaje kasnije (pri drugim pozivima).
  - Kurirane funkcije su funkcije koje u jednom pozivu prihvataju samo jedan argument.
- I parcijalna aplikacija i kuriranje su u vezi sa redukcijom *arnosti* funkcije i tesno su povezane sa zatvaranjem.

Sažetak

- Funkcija je jedan od glavnih koncepata u programskom jeziku JavaScript.
- Funkcija u jeziku JavaScript, ako se koristi na način saglasan sa principima funkcionalnog programiranja, je ekvivalentna sa funkcijom definisanom u matematici: prima jedan ili više ulaznih podataka i vraća jedan izlaz.
- Podaci mogu da budu i druge funkcije, odnosno funkcija može da primi drugu funkciju kao ulaz a funkcija može i da vrati drugu funkciju kao izlaz.
- Mehanizmi za prosleđivanje podataka među funkcijama u jeziku JavaScript su veoma raznovrsni: argumenti, dosezi, zatvaranja, eksplicitna vezivanja.
- JavaScript podržava (u određenoj meri) veoma važan koncept FP-a - ***rekurziju*** (mehanizam koji omogućuje da funkcija pozove samu sebe).

Literatura za predavanje

1. Z. Konjović, Funkcije u programskom jeziku JavaScript - napredno korišćenje, Kurs Funkcionalno programiranje školska 2024/25 (slajdovi sa predavanja), Univerzitet Singidunum, dostupno na Teams platformi
2. Z.Konjović, Skripta Uvod u funkcionalno programiranje, autorski rukopis, Univerzitet Singidunum, dostupno na Teams platformi
3. I. Kantor, **Advanced working with functions**, dostupno na: https://javascript.info/advanced-functions
4. K. Simpson, **Lagano-funkcionalni JavaScript** (Poglavlje 2), dostupno na stranici predmeta
5. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions