Univerzitet Singidunum
www.singidunum.ac.rs

Funkcionalno programiranje

Školska 2024/25 godina

Letnji semestar

Tema 6: Funkcionalno programiranje sa listama

Sadržaj

- Zašto su operacije sa listama važne
- Kako operacije sa listama (mogu da) rade
- Osnovne operacije sa listama
  - Mapiranje
  - Filtriranje
  - Redukcija
- Naprednije operacije sa listama
- Metode i samostalne funkcije nad listama

Zašto su operacije sa listama važne

- Liste i operacije sa listama primenjuju se u najrazličitijim programerskim zadacima, kao što je upravljanje procesima u programu, pravljenje interpretera i kompjalera, itd.
  - Većina uobičajenih ilustracija operacija sa listama prikazuje trivijalne zadatke koji se izvršavaju nad listama vrednosti (n.pr., udvostručavanje svake vrednosti u nizu brojeva, transformisanje stringa u velika slova, itd.); to je jeftin i lak način da se shvati poenta.
  - U stvari, najvažnije vrednosti operacija sa listama u FP-u potiču iz mogućnosti da se deklarativno modeluju sekvence imperativnih naredbi kao *operacije nad listom umesto nad individualnim njenim elementima.* Na primer, funkcija `forEach()` koju smo već videli a još ćemo da je pogledamo.
  - Veoma je korisno *ulančavanje/kompozicija operacija nad listama* što omogućuje da se međurezultati prate implicitno i relaksira probleme slučajnih mutacija vrednosti i bočnih efekata.

Operacije nad listama u duhu FP-a

- Kada se kodiraju operacije u FP-u, one se predstavljaju u obliku funkcija.
- Funkcije kao i ostali kod pisan u duhu FP-a treba da zadovoljavaju principe FP-a: referencijalnu transparentnost, odsustvo bočnih efekata, imutabilnost, uniformno ponašanje nad tipovima, očuvanje kompozicije.
- Kao ilustraciju, pokazaćemo neke implementacije funkcija i komentarisati ih iz aspekta saglasnosti sa principima FP-a.
  - `forEach()`
  - `some()`, i
  - `every()`

FP-nesaglasno procesiranje liste – funkcija `forEach()`

```
const forEach = (array,fn) => {
    let result = [];
    var i;
    for (i=0; i<array.length; i++){
        result [i] = fn(array[i])
    }
        return result;
}
```

- Ova implementacija nije u duhu FP-a zato što je dizajnirana tako da za svaki poziv operiše sa bočnim efektom – poziva funkciju `fn()` koja može da proizvodi bočni efekat, na primer:

```
forEach([1,2,3], x => console.log(x))
```

FP-nesaglasno procesiranje liste – funkcije `some()` i `every()`

```
const some = (arr,fn) => {
   let result = false;
   for(const value of arr)
     result = result || fn(value)
   return result
}
const every = (arr,fn) => {
   let result = true;
   for(const value of arr)
     result = result && fn(value)
   return result
}
```

- Iako su formalno čiste: ulazi (`arr` i `fn`) se prosleđuju kroz listu parametara, u suštini redukuju ulaznu listu na jedan rezultat(`true`/`false`), što nije u duhu principa očuvavanja kompozicije.

Mapiranje

- Mapiranje je transformacija jedne vrednosti u drugu vrednost.
  - Na primer, ako uzmete broj 2 i pomnožite ga sa 3, vi ste broj 2 mapirali na 6 i to tako što ste primenuli funkciju koja svoj ulaz množi sa 3 .
- Pri tome, o mapiranju ne govorimo kao mutaciji ulazne vrednosti ili ponovnoj dodeli, već kao o *projektovanju* **vrednosti** iz **jedne lokacije** u **novu** vrednost **na drugoj lokaciji** – znači, poštovanje **FP principa imutabilnosti**.
- Dakle:

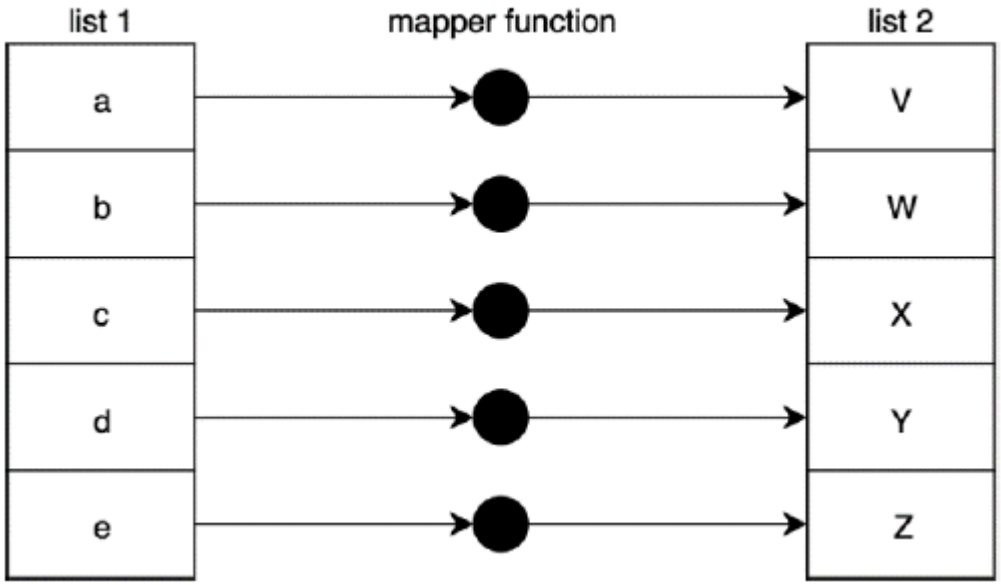
```
var x = 2, y;
y = x * 3;//ovo je ispravno (transformacija/projekcija)
x = x * 3; //ovo nije ispravno (mutacija, ponovna dodela)
```

- Odnosno, ovo je dobro:

```
var multipleBy3 = v => v * 3;
var x = 2, y;
// transformacija / projekcija
y = multiplyBy3( x );
```

`map` nad listom

- Možemo prirodno da proširimo mapiranje jedne vrednosti na mapiranje kolekcije vrednosti: `map()` nad listom je operacija koja transformiše **sve vrednosti iz liste** i projektuje (smešta) **nove vrednosti** u **novu listu**.



mapSimple (): naivna implementacija map-a

```
const mapSimple = (array,fn) => {
  let results = []
        for(const value of array)
          results.push(fn(value))
   return results;
}
// pozivi
let inList = [1,2,3]
console.log (' Ulazna lista pre poziva map: ' + inList);  // [1,2,3]
let outList = mapSimple(inList, (x) => x + 2); // mapiranje x+2
console.log ( ' Izlazna lista nakon map: ' + outList); // [3,4,5]
console.log (' Ulazna lista nakon poziva map: ' + inList);  // [1,2,3]
```

Funkcija **map** nad listom: implementacija slična bibliotečkoj

```
function map(mapperFn,arr) {
var newList = [];
  for (let [idx,v] of arr.entries()) {
    newList.push(
          mapperFn( v, idx, arr )
    );
  }
  return newList;
}
```

Funkcija `map` nad listom: implementacija slična bibliotečkoj

```
function map(maperskaFn,ulazniNiz) {
    var novaLista = [];
    for (let [indeks,vrednost] of ulazniNiz.entries()){
        novaLista.push(
            maperskaFn(vrednost, indeks, ulazniNiz)
        );
    }
    return novaLista;
}
console.log (map((x) => {return 2*x}, [1,2,3]))
```

Korišćenje: "škakljivi" slučajevi

- "Klasičan" primer je

```
["1", "2", "3"].map(parseInt)
// Očekivano: [1, 2, 3]
// Dobijeno: [1, NaN, NaN]
```

- Razlog: funkcija `parseInt` prima 2 argumenta (string i osnovu broja koji će biti vraćen) a `map()` prosleđuje 3 argumenta pa se dešava sledeće:

```
// parseInt(string,radix) -> map(parseInt(value,index))
/*  prva iteracija(index 0): */ parseInt("1", 0)  // 1
/*  druga iteracija(index 1): */ parseInt("2", 1) // NaN
/*  treća iteracija(index 2): */ parseInt("3", 2) // NaN
```

"Škakljivi" slučajevi: Rešenje je kuriranje

- Napravi se funkcija koja od zadate funkcije napravi funkciju koja ima jedan argument:

```
function unary(fn) {
  return function onlyOneArg(arg){
    return fn(arg);
  };
};

["1", "2", "3"]. map(unary(parseInt))// radi dobro
```

Korišćenje: škakljivi slučajevi – rešenje sa detaljima

```
let inArray1 = ["1","2","3"]
console.log(' ulazna lista 1: ' + inArray1) /* Ispis:
  ulazna lista 1: [1,2,3] */

console.log(' ulazna lista 1 tipovi elemenata: ' +
  typeof (inArray1[0]) + ' ' + typeof (inArray1[1]) + '
  ' + typeof (inArray1[2])) /* Ispis: ulazna lista 1
  tipovi elemenata: string string string */

let mapiranaLista1 = map(unary(parseInt), inArray1);

console.log(' mapirana lista 1: ' + mapiranaLista1) /*
  Ispis: mapirana lista 1: [1,2,3] */

console.log(' mapirana lista 1 tipovi elemenata: ' +
  typeof (mapiranaLista1[0]) + ' ' + typeof
  (mapiranaLista1[1]) + ' ' + typeof
  (mapiranaLista1[2])) /* Ispis: mapirana lista 1
  tipovi elemenata: number number number */
```

Primer : Knjige izdavača Apress – zadatak 1

Zadatak: Knjige koje je izdala izdavačka kuće ***Apress*** opisane su sledećim atributima:

identifikator (numerik), naslov (string), autor (string), ocena (više mogućih vrednosti, numerik razlomljen), broj recenzija po kategorijama (neobavezno, celobrojna vrednost, moguće kategorije su "bad", "good", "excellent")

a) Predložiti strukturu za organizovanje ovih podataka i konstruisati reprezentaciju sledećih ulaznih podataka.

111, "C# 6.0", "ANDREW TROELSEN", 4.7, good : 4 , excellent : 12

222, "Efficient Learning Machines", "Rahul Khanna", 4.5,

333, "Pro AngularJS", "Adam Freeman", 4.0

444, "Pro ASP.NET", "Adam Freeman", 4.2 , good : 14 , excellent : 12

b) Napisati program primenom FP-a koji vraća podatke koji sadrže naslov i autora knjige

Primer : Knjige izdavača Apress – analiza i rešenje

- Struktura podataka je **lista** čiji su elementi **objekti** sa svojstvima koja odgovaraju atributima iz opisa knjiga, pri čemu je atribut ocena takođe lista.
- Izlazni rezultat je **lista** čiji su elementi **objekti** sa dva svojstva: `autor` i `naslov`.
- Izlazna lista formira se iz ulazne liste mapiranjem, pri čemu je funkcija mapiranja funkcija koja mapira svojstva `title` i `author` ulaznog niza na svojstva `naslov` i `autor` izlaznog niza.

Rešenje: Knjige izdavača Apress – podaci

```
let ulaznaListaKnjiga = [
{"id": 111,"title": "C# 6.0","author": "ANDREW TROELSEN","rating": [4.7],"reviews": [{good : 4 , excellent : 12}]},
{"id": 222,"title": "Efficient Learning Machines","author": "Rahul Khanna","rating": [4.5],"reviews": []},
{"id": 333,"title": "Pro AngularJS","author": "Adam Freeman","rating": [4.0],"reviews": []},
{"id": 444,"title": "Pro ASP.NET","author": "Adam Freeman","rating": [4.2],"reviews": [{good : 14 , excellent : 12}]}
];
```

Rešenje: Knjige izdavača Apress – kod

```
// funkcija map u kućnoj radinosti
const map = (array,fn) => {
let results = []
for(const value of array)
   results.push(fn(value))
return results;
}

// transformacija
const fn = (knjiga) => {
return {naslov: knjiga.title,
        autor:  knjiga.author}
}

let izlaznaLista = map (ulaznaListaKnjiga, fn)
IspisiObjekat(izlaznaLista) // funkcija IspisiObjekat je jedna od
funkcija iz helpera za ispisivanje (listing je u delu notes ovoga slajda)
```

Rešenje: Knjige izdavača Apress – rezultat

[{ "naslov": "C# 6.0", "autor": "ANDREW TROELSEN" },
{ "naslov": "Efficient Learning Machines", "autor": "Rahul Khanna" },
{ "naslov": "Pro AngularJS", "autor": "Adam Freeman" },
{ "naslov": "Pro ASP.NET", "autor": "Adam Freeman" }]

Ugrađena funkcija **`Array.prototype.map()`**

- Sintaksa:

```
let new_array = arr.map(function callback(
  currentValue[, index[, array]]) { // return
  element for new_array }[, thisArg])
```

- Parametri:
  - ***`callback`*** – funkcija koja se poziva nad svakim elementom liste *`arr`*. Svaki put kada se *`callback`* izvrši, vraćena vrednost se dodaje u listu *`new_array`*. Ova funkcija prihvata sledeće argumente:
    - *`currentValue`* – vrednost tekućeg elementa u listi koja se mapira.
    - *index* (*n*eobavezan) – indeks tekućeg elementa liste koja se mapira.
    - *`array`* (*n*eobavezan) - lista nad kojom je pozvan map
  - **`thisArg`** (*n*eobavezan) – vrednost koja se koristi kao pokazivač `this` kada se izvršava *callback*.

`Array.prototype.map()`: detalji$_{1/3}$

- `map` poziva prosleđenu *`callback`* funkciju **po jednom za svaki element** u listi, po redu, i pravi novu listu u kojoj su rezultati.
- Funkcija *callback* poziva se samo za indekse niza koji imaju dodeljenu vrednost (uključujući i `undefined`).
- Ne poziva se za nedostajuće elemente liste, što znači indekse:
  - koji nikada nisu bili postavljeni,
  - koji su izbrisani, ili
  - kojima nikada nije dodeljena vrednost.

`Array.prototype.map()`: detalji$_{2/3}$

- *callback* se poziva sa tri argumenta: vrednost elementa, indeks elementa, i array objekat koji se mapira.
- Ako se zada parametar `thisArg`, on se koristi kao `this` vrednost *callback*-a.
  - U protivnom, `this` vrednost za *callback* je `undefined`.
  - Konačna `this` vrednost koju vidi *callback* određuje se u skladu sa uobičajenim načinom za funkciju.
  - Podestimo se: streličaste funkcije **nemaju `this` pokazivač**
- `map` ne mutira ulaznu listu nad kojom se poziva (iako *callback*, ako se pozove, može to da uradi).

`Array.prototype.map()`: detalji$_{3/3}$

- Opseg elemenata koje će map da obradi postavlja se pre prvog poziva funkcije *callback*.
- Elementi koji se listi dodaju nakon što započne poziv, neće biti obrađeni od strane tog poziva.
- Ako se izmenu postojeći elementi liste, njihova vrednost koja se prosleđuje *callback*-u biće vrednost koju imaju u trenutku kada ih map uzme u obradu.
- Elementi koji su izbrisani nakon poziva map-a ali pre no što budu uzeti u obradu, se ne uzimaju u obradu.
- Algoritam je takav da održava "retkost" ulazne liste – nema nikakvog "pakovanja" .
- Sve detalje o metodi map možete naći na https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

`Array.prototype.map()`: Primer generičkog korišćenja

```
// Korišćenje map nad String tipom
let map = Array.prototype.map
console.log(map) // ƒ map() { [native code] }
let a = map.call('Hello World', function(x) {
  return x.charCodeAt(0)
})
console.log (a)
/* Ispis:[72, 101, 108, 108, 111, 32, 87, 111,
 114, 108, 100] */
```

map(): imutabilnost$_{1/2}$

```
let numbers = [1, 4, 9]
console.log(numbers) // [1, 4, 9]
let roots = numbers.map(function(num) {
  return Math.sqrt(num)})
console.log(roots) // [1, 2, 3]
console.log(numbers)// [1, 4, 9]
```

map(): imutabilnost 2/2

```
let kvArray = [{key: 1, value: 10},
               {key: 2, value: 20},
               {key: 3, value: 30}]

let reformattedArray = kvArray.map(obj => {
   let rObj = {}
   rObj[obj.key] = obj.value
   return rObj
})
/* reformattedArray je [{"1": 10}, {"2": 20},
  {"3": 30}] */

/* kvArray je: [{key: 1, value: 10},{key: 2,
  value: 20},{key: 3, value: 30}] */
```

`map()`: može da radi svašta

```
// Da mapira funkciju na vrednost funkcije
var one = () => 1;
var two = () => 2;
var three = () => 3;
[one,two,three].map( fn => fn() ); // [1,2,3]

// Da transformiše listu funkcija tako što će
//svaku od njih komponovati sa nekom funkcijom
// i zatim izvršiti funkciju koja je rezultat kompozicije
var increment = v => ++v;
var decrement = v => --v;
var square = v => v * v;
var double = v => v * 2;
[increment,decrement,square]
.map( fn => compose( fn, double ) )
.map( fn => fn( 3 ) ); // [7,5,36]
```

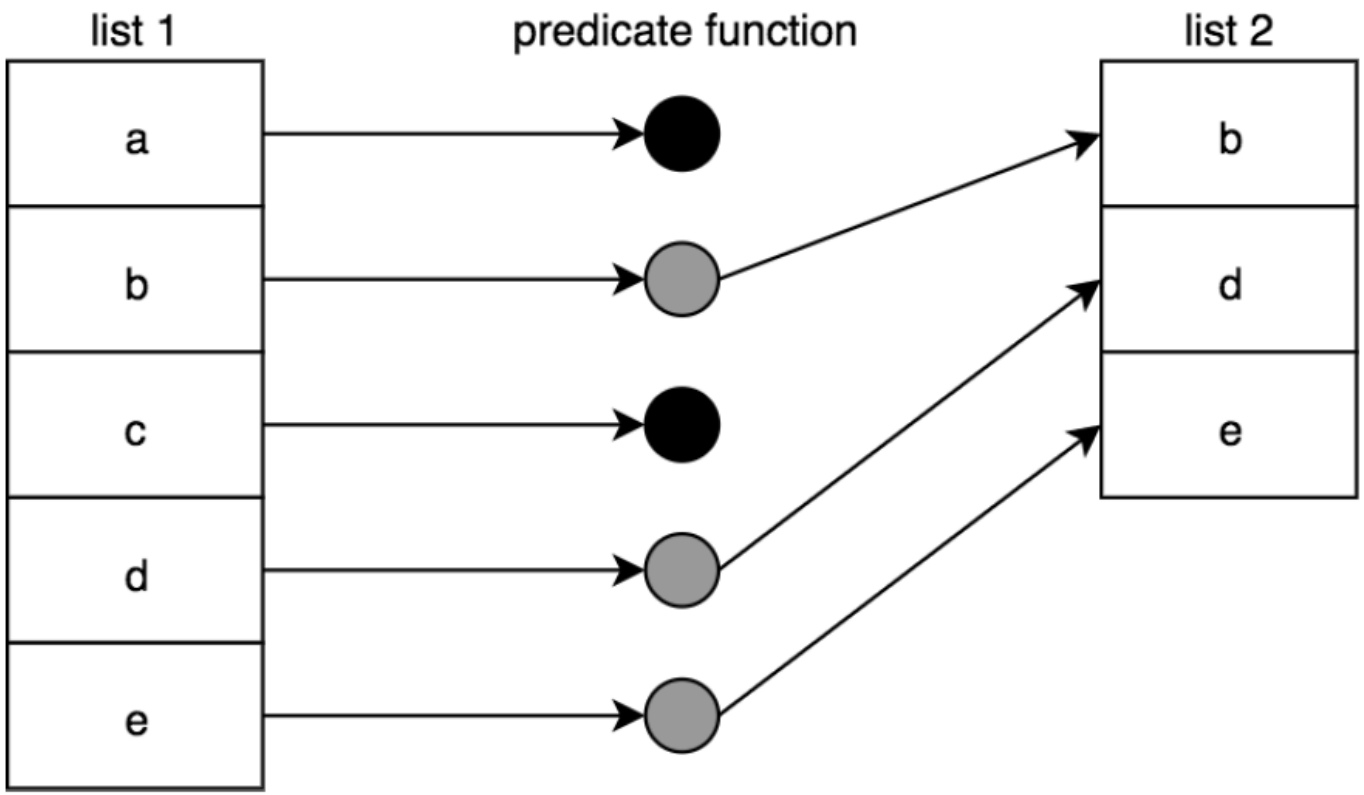
Filtriranje

- Operacija filtriranja nad listom (`filter()`) primenjuje funkciju da bi odlučila da li će svaka vrednost u originalnom nizu biti u novoj listi ili neće.
- Ta funkcija treba da vrati vrednost `true` ako vrednost treba da se uključi, ili vrednost `false` ako vrednost treba da se preskoči.
- Funkcija koja vraća vrednost `true`/`false` za takvu vrstu odlučivanja zove se ***predikatska funkcija***.

Predikatska funkcija

- Predikatska funkcija treba da odluči da li će element iz ulazne liste biti "propušten" u izlaznu listu
  - Ako razmišljate o vrednosti `true` kao o indikatoru pozitivnog signala, definicija funkcije `filter()` je da Vi kažete "propusti" vrednost (za filtriranje **u**), a ne "nemoj da propustiš" vrednost (za filtriranje **iz**).
  - Da bi se operacija `filter()` koristila kao isključujuća akcija, potrebno je da razmišljate o pozitivnom signalu kao indikaciji isključenja koja vraća `false` i pasivnom propuštanju vrednosti vraćanjem `true`.
- Važno je da ovo bude jasno zbog načina na koji ćete želeti da imenujete funkciju koja se koristi kao `predicateFn()`, i zbog značenja za čitljivost koda.

Filtriranje liste vrednosti



Funkcija `filter()`: sasvim naivna implementacija

```
const filter = (array,fn) => {
  let results = []
      for(const value of array)
        (fn(value)) ? results.push(value)
                    : undefined
         return results;
}
```

Funkcija `filter ()`: malo manje naivna implementacija

```
function filter(predicateFn,arr) {
var newList = [];
 for (let [idx,v] of arr.entries()) {
     if (predicateFn( v, idx, arr )) {
          newList.push( v );
     }
 }
     return newList;
}
```

Knjige izdavača Apress – zadatak 2

Zadatak: Koristeći strukturu podataka iz primera **Knjige izdavača Apress – zadatak 1**, napisati program koji će za sve knjige koje imaju vrednost ocene (`rating`) veću od zadate vrednosti vratiti sve podatke koji opisuju knjigu.

Knjige izdavača Apress – zadatak 2: rešenje

```
// podaci isti kao za zadatak 1
// funkcija filter u domaćoj izvedbi
const filter = (array,fn) => {
   let results = []
   for(const value of array)
      (fn(value)) ? results.push(value) :
undefined
   return results;
}
// poziv
filter(apressBooks, (book) => book.rating[0]
> 4.5)
```

Knjige izdavača Apress – zadatak 2: rezultat


```
{"id": 111,"title": "C# 6.0","author":
"ANDREW TROELSEN","rating": [4.7],
"reviews": [{good : 4 , excellent :
12}]}
```

Ugrađena funkcija `Array.prototype.filter()`

- Sintaksa:

```
let newArray = arr.filter(callback(element[, index, [array]])[, thisArg])
```

- Parametri:
  - *`callback`* – funkcija je predikat za testiranje svakog elementa niza. Vraća `true` za zadržavanje elementa, `false` u protivnom.
  - Funkcija *`callback`* prihvata sledeće argumente:
    - *`element`* – vrednost tekućeg elementa u listi koja se filtrira.
    - *`index`* (*n*eobavezan) – indeks tekućeg elementa liste koja se obrađuje.
    - *`array`* (*n*eobavezan) - lista nad kojom je pozvan filter
    - `.thisArg` (*n*eobavezan) – vrednost koja se koristi kao pokazivač `this` kada se izvršava *callback*.
  - Povratna vrednost je nova lista sa elementima koji su prošli test. Ako ni jedan element nije prošao test, vraća se prazna lista.

`Array.prototype.filter()`: Detalji$_{1/3}$

- `filter()` poziva prosleđenu *callback* funkciju po jednom za svaki element liste, i pravi novi niz od svih vrednosti za koje *callback* vrati vrednost koja se svodi (koercijom) na `true`.
- *callback* se poziva samo za indekse koji imaju dodeljene vrednosti; ne poziva se za indekse koji su brisani ili kojima nikada nije dodeljena vrednost. Elementi liste koji ne prođu *callback* test se prosto preskaču i ne uključuju se u novu listu/niza.
- *callback* se poziva sa tri argumenta:
  - Vrednost elementa
  - Indeks elementa
  - Objekat tipa `Array` kroz koji se prolazi

`Array.prototype.filter()`: Detalji$_{2/3}$

- Ako se prosledi vrednost za parametar `thisArg`, ta se vrednost koristi kao `this` vrednost za *callback*. U protivnom, `this` za *callback* je `undefined`. Konačna vrednost `this` za *callback* se određuje po pravilima za određivanje `this` vrednosti za funkciju.
- `filter()` **ne mutira** ulaznu listu nad kojom je pozvan.

`Array.prototype.filter()`: Detalji$_{3/3}$

- Opseg elemenata koje će `filter()` da obradi postavlja se pre prvog poziva funkcije *callback*.
- Elementi koji se listi dodaju nakon što započne poziv, neće biti obrađeni od strane tog poziva.
- Ako se izmenu postojeći elementi liste, njihova vrednost koja se prosleđuje *callback*-u biće vrednost koju imaju u trenutku kada ih `filter()` uzme u obradu.
- Elementi koji su izbrisani nakon poziva `filter()`-a ali pre no što budu uzeti u obradu, se ne uzimaju u obradu.
- Sve detalje o metodi `filter ()` možete naći na https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

Array.prototype.filter(): Primer 1

```
function isBigEnough(value) {
  return value >= 10
}

let inArray = [12, 5, 8, 130, 44]
console.log (' Ulazni niz: ' + inArray) /*
 Ulazni niz: [12, 5, 8, 130, 44] */
let filtered = inArray.filter(isBigEnough)
console.log (' Propušteni veći od 10 : ' +
 filtered) /* Propušteni veći od 10 :
 [12, 130, 44] */
```

Array.prototype.filter(): Primer $2_{1/2}$

- Invalidne JSON stavke: prebrajanje stavki koje nisu tipa Number u JSON objektu ili im je vrednost 0

```
function isNumber(obj) { // Provera da li je tip Number
  return obj !== undefined && typeof(obj) === 'number' &&
  !isNaN(obj)
}
function filterByID(item) {
  if (isNumber(item.id) && item.id !== 0) { // Provera na 0
    return true
  }
    invalidEntries++
  return false;
}
```

Array.prototype.filter(): Primer $2_{2/2}$

```
// Primer invalidne JSON stavke;  prebrajanje stavki koje nisu tipa
  Number u JSON objektu ili im je vrednost 0

// Nefiltrirani niz
let arr = [  { id: 15 },   { id: -1 },   { id: 0 },   { id: 3 },   { id:
  12.2 },  { },  { id: null },  { id: NaN },  { id: 'undefined' }]
console.log('Nefiltrirani niz\n', arr)

let invalidEntries = 0
let arrByID = arr.filter(filterByID)
console.log('Filtrirani niz\n', arrByID)
// Filtrirani niz
// [{ id: 15 }, { id: -1 }, { id: 3 }, { id: 12.2 }]
console.log('Broj nevalidnih stavki = ', invalidEntries) /* Broj
  nevalidnih stavki = 5 */
```

Array.prototype.filter(): Primer 3

```
const voce = ['jabuka', 'banana', 'grožđe', 'jagoda',
  'pomorandža']


/*
 * Filtriraj elemente niza na bazi upita
 */
const filterItems = (arr, query) => {
  return arr.filter(el =>
  el.toLowerCase().indexOf(query.toLowerCase()) !== -1)
}


console.log(filterItems(voce, 'ja'))
// ['jabuka', 'jagoda']
console.log(filterItems(voce, 'gr'))
// ['grožđe']
```

Ulančavanje operacija 1/2

- Najčešće nam je potrebno da *primenimo više funkcija da bismo dostigli krajnji cilj.*
- Na primer, zamislite problem pravljenja liste koja sadrži naslove i imena autora knjiga iz primera apressBooks koje imaju prvu vrednost obeležja `rating` veću od 4.5.
- To bismo, naravno, mogli da uradimo pomoću jedne funkcije koja bi radila sledeće:
  - Za svaki objekat proveravala da li ima prvu vrednost obležja `rating` veću od 4.5.
  - Ako je uslov zadovoljen, pravila bi novi objekat koji ima dva svojstva `title` i `author`.
- Ova funkcija, u suštini, radi dve stvari: (1) filtrira objekte koji zadovoljavaju zadati uslov i (2) mapira zadata obeležja objekata koji zadovoljavaju uslov na novi objekat koji ima samo dva obeležja.
- Zato je prirodno da se problem rešava pomoću `map` i `filter`, recimo ovako:

```
let goodRatingBooks =
  filter(apressBooks, (book) => book.rating[0] > 4.5)
  map(goodRatingBooks,(book) => {
      return {title: book.title,author:book.author}
    })
```

Ulančavanje operacija 2/2

- Ili još bolje ovako - bez potrebe za dodatnom varijablom goodRatingBooks:

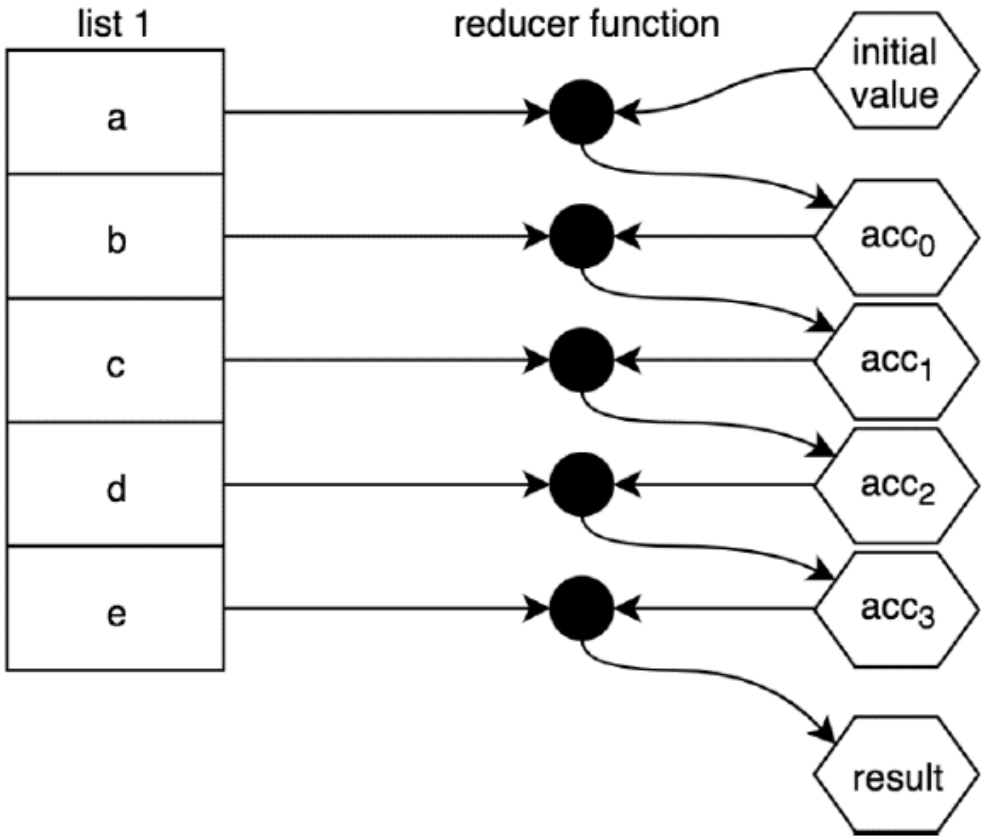
```
map(filter(apressBooks, (book) =>
  book.rating[0] > 4.5),(book) => {
return
  {title:book.title,author:book.author}
})
```

- Poslednji kod doslovce ŠTA TREBA da se uradi: "**Mapiraj** nad **filtriranim** nizom čija je **prva ocena > 4.5** i vrati **title** i **author** ključeve u objektu!"

Redukcija: funkcija `reduce()`

- Jedna od čestih operacija nad listama je **redukovanje liste na jednu vrednost**; primeri su računanje zbira elemenata liste, računanje proizvoda elemenata liste, računanje intenziteta vektora, formiranje jednog stringa od podstringova, itd.
- Operator `reduce()` kombinuje ("redukuje") listu na jednu vrednost (recimo, određivanje skalarne vrednost za neki vektor).
- Kombinacija/redukcija se apstraktno definiše kao uzimanje dve vrednosti i pravljenje jedne vrednosti od njih.
- Kao i kod mapiranja i filtriranja, način kombinovanja je u potpunosti na Vama i, generalno, zavisan od tipova vrednosti u listi.
- U nekim situacijama, redukcija specificira početnu vrednost, `initialValue`, i radi tako što kombinuje tu početnu vrednost sa prvim elementom u listi i zatim se spušta kaskadno kroz sve ostale vrednosti u listi.

Redukcija: vizualizacija



Redukcija: implementacija

```
const reduce = (array,fn,initialValue) => {
let accumlator;
// proverava se da li postoji početna vrednost
        if(initialValue != undefined)
                accumlator = initialValue;
        else
                accumlator = array[0];
        if(initialValue === undefined)
                for(let i=1;i<array.length;i++)
        accumlator = fn(accumlator,array[i])
        else
                for(const value of array)
                accumlator = fn(accumlator,value)
return [accumlator]
}
```

Ugrađena funkcija: `Array.reduce()`

- Izvršava funkciju **reducer** (koju vi zadajete) nad svakim elementom liste i daje jednu izlaznu vrednost (skalar).
- Funkcija **reducer** prima 4 argumenta:
  - Accumulator (*acc*)
  - Current Value (*cur*)
  - Current Index (*idx*)
  - Source Array (*src*)
- Vrednost koju vraća vaša **reducer** funkcija dodeljuje se promenljivoj `accumulator`, čija se vrednost pamti kroz iteriranje i na kraju je to rezultujuća vrednost.

Array.reduce(): primer

```
const array1 = [1, 2, 3, 4];
const reducer = (accumulator, currentValue)
                => accumulator + currentValue;

// 1 + 2 + 3 + 4
console.log(array1.reduce(reducer));
// Očekivani izlaz: 10

// 5 + 1 + 2 + 3 + 4
console.log(array1.reduce(reducer, 5));
// Očekivani izlaz: 15
```

Array.prototype.reduce()

- Sintaksa:

```
arr.reduce(callback(accumulator, currentValue[, index[, array]] )[, initialValue])
```

- Parametri
  - `callback` – funkcija koja se izvršava nad svakim elementom liste (osim prvog, ako nije prosleđen argument za *initialValue*). Funkcija prima 4 argumenta.
  - `initialValue` (opcioni) - Vrednost koja će biti korišćena kao prvi argument pri prvom pozivu *callback*-a.

`Array.prototype.reduce()`: parametri callback fukcije

- Funkcija prima 4 argumenta:
  - `Accumulator` – akumulira povratne vrednosti `callback` funkcije. Predstavlja ili akumuliranu vrednost prethodno vraćenu iz poslednjeg poziva funkcije `callback` ili prosleđenu `initialValue`.
  - `currentValue` – Tekući element liste koji se obrađuje
  - `currentIndex` (opcioni) – indeks tekućeg elementa. Počinje od indeksa 0 ako je prosleđen `initialValue` , inače od indeksa 1.
  - `array` (opcioni) – lista nad kojom je pozvan `reduce()`.

`Array.prototype.reduce()`: parametar `initialValue`

- `initialValue` (opcioni argument) - Vrednost koja će biti korišćena kao prvi argument pri prvom pozivu *callback*-a.
  - Ako se `initialValue` ne prosledi, prvi elemenat u nizu se koristi kao inicijalna vrednost za `accumulator` i preskače se kao tekuća vrednost (*currentValue*) pri pozivu *callback*-a.
  - Ako se reduce() pozove sa praznim nizom i bez prosleđenog argumenta za *initialValue* generisaće se TypeError.

Array.prototype.reduce()
Kako radi

```
[0, 1, 2, 3,4]
    .reduce(function(accumulator,
     currentValue, currentIndex, array) {
     return accumulator + currentValue })
```

callback iteracija	accumulator	currentValue	currentIndex	array	vraćena vrednost
prvi poziv	0	1	1	[0, 1, 2, 3, 4]	1
drugi poziv	1	2	2	[0, 1, 2, 3, 4]	3
treći poziv	3	3	3	[0, 1, 2, 3, 4]	6
četvrti poziv	6	4	4	[0, 1, 2, 3, 4]	10

Primer 1: Elementi niza su objekti

```
let initialValue = 0
let sum = [{x: 1}, {x: 2}, {x: 3}].reduce(function
(accumulator, currentValue) {
    return accumulator + currentValue.x
}, initialValue)
 console.log(sum) // 6


let initialValue = 0
let sum = [{x: 1}, {y: 2}, {x: 3}].reduce(function
(accumulator, currentValue) {
    return accumulator + currentValue.x
}, initialValue)
 console.log(sum) // NaN
```

Primer 2: Brojanje pojava

```
let imena = ['Mara', 'Boba', 'Mika', 'Mile', 'Mara','Mara', 'Vera', 'Vera' ]
let izbrojanaImena = imena.reduce(function (svaImena, ime) {
  if (ime in svaImena) {
    svaImena[ime]++
 }
   else {    svaImena[ime] = 1
}
return svaImena
}, {})
// izbrojanaImena je:
izbrojanaImena;
// Vraća: {Mara: 3, Boba: 1, Mika: 1, Mile: 1, Vera: 2}
```

`reduce()` je specijalna funkcija

- Ispostavlja se da funkcija `reduce()` ima poseban značaj.
- Razlog je što se pomoću nje mogu predstaviti i `map()` i `filter()`.

`map()` kao `reduce()`

- Operacija `map()` je iterativna po svojoj prirodi, tako da se može predstaviti i kao redukcija (`reduce()`).
- Poenta je da se shvati da početna vrednost `initialValue` funkcije `reduce()` može i sama da bude (prazan) niz u kom slučaju rezultat redukcije može biti druga lista!

map() kao reduce(): primer

```
var double = v => v *2;
 // ovo je "čist" map
let inList = [1,2,3,4,5]
console.log(inList)
console.log(inList.map(double)); // [2,4,6,8,10]
console.log(inList) // [1,2,3,4,5]

// a ovo je takodje mapiranje, ali sa reduce()
var double = v => v *2;
let inList = [1,2,3,4,5]
console.log(inList) // [1,2,3,4,5]
console.log(
inList
  .reduce((list,v) => (list.push(double(v)),
                                  list),
                      [])); // [2,4,6,8,10]
console.log(inList) // [1,2,3,4,5]
```

`filter()` kao `reduce()`

- Baš kao i `map()`, i `filter()` se može realizovati pomoću `reduce()`:

```
var isOdd = v => v % 2 == 1;

[1,2,3,4,5].filter(isOdd); // [1,3,5]

[1,2,3,4,5].reduce(
    (list,v) => (
            isOdd(v) ? list.push(v) : undefined,
            list),
            []
    ); // [1,3,5]
```

filter() kao reduce(): primer

```
const isOdd = v => v % 2 == 1;
let inArray = [1,2,3,4,5]

console.log ('inArray pre filter: ' + inArray); /* inArray pre
  filter: [1,2,3,4,5] */

console.log('inArray.filter(isOdd): ' + inArray.filter(isOdd)); /*
  inArray.filter(isOdd): [1, 3, 5] */

console.log('inArray posle filter: ' + inArray); /* inArray posle
  filter: [1, 2 3, 4, 5] */

console.log ('inArray pre filter by reduce: ' + inArray); /* inArray
  pre filter by reduce: [1,2,3,4,5]  */
console.log ('inArray.reduce: ' + inArray1.reduce(
            (list,v) => (
                isOdd(v) ? list.push(v) : undefined,
                list),
                []
    )); /* inArray.reduce: [1, 3, 5] */
console.log ('inArray posle filter by reduce: ' + inArray); /*
  inArray posle filter by reduce: [1,2,3,4,5]  */
```

Naprednije operacije sa listama

- Postoji još mnogo operacija sa listama. Pokazaćemo najvažnije:
  - Operacije sa jednom listom
    - Eliminisanje duplikata
    - Poravnavanje
    - Mapiranje pa poravnavanje
  - Operacije sa više lista
    - Zipovanje
    - Spajanje lista

Eliminisanje duplikata

- Ima puno algoritama za eliminsanje duplikata iz liste
- Dva ćemo da prikažemo
  - Prvi koristi `filter()`
  - Drugi koristi `reduce ()`

Brisanje duplikata: pomoću `filter()`

```
var uniquef =
    arr => arr.filter(
            (v,idx) => arr.indexOf(v) == idx
        );
inArr =[1,4,7,1,3,1,7,9,2,6,4,0,5,3];
console.log(inArr);
//[1,4,7,1,3,1,7,9,2,6,4,0,5,3]
console.log(uniquef( inArr ));
// [1, 4, 7, 3, 9, 2, 6, 0, 5]
console.log(inArr);
// [1,4,7,1,3,1,7,9,2,6,4,0,5,3]
```

Brisanje duplikata: pomoću `reduce()`

```
var uniquer =
  arr =>
    arr.reduce(
      (list,v) =>
         list.indexOf( v ) == -1?
           ( list.push( v ), list ):list
           , [] );
inArr =[1,4,7,1,3,1,7,9,2,6,4,0,5,3];
console.log(inArr);
console.log(uniquer( inArr ));
console.log(inArr);
```

Poravnavanje

- Transformacija:

`[ [1, 2, 3], 4, 5, [6, [7, 8]] ] → [ 1, 2, 3, 4, 5, 6, 7, 8 ]`

- Može se implementirati operatorom `reduce()`:

```
var flatten =
  arr =>
      arr.reduce(
        (list,v) =>
          list.concat( Array.isArray(v) ? flatten(v) : v )
                    , [] );

  flatten(  [[0,1],2,3,[4,[5,6,7],[8,[9,[10,[11,12],13]]]]] );
  // [0,1,2,3,4,5,6,7,8,9,10,11,12,13]
```

Poravnavanje: može i sa `reduce()`

```
let flattened =
[[0, 1], [2, 3], [4, 5]].reduce((
accumulator,
currentValue ) =>
accumulator.concat(currentValue),
[])
console.log (flattened) // [0, 1, 2, 3, 4, 5]
```

Delimično poravnavanje$_{1/3}$

- Po nekad je potrebno je raditi i delimično poravnavanje, na primer:

```
[[0,1],2,3,[4,[5,6,7],[8,[9,[10,[11,12],13]]]]] →
[0,1,2,3,4,5,6,7,8,9,10,[11,12],13]
```

- Za to je potrebno uvesti mehanizam kojim bi se kontrolisala "dubina" poravnavanja. U primeru implementacije koji sledi to je `depth`.

Delimično poravnavanje 2/3

```
var flatten = (
// depth – dubina poravnavanja
      arr,depth = Infinity) =>
          arr.reduce(
               (list,v) =>
                    list.concat(
                        depth > 0 ?
                               (depth > 1 && Array.isArray( v ) ?
                                      flatten( v, depth - 1 ) :
                                      v
                               ) :
                               [v]
                    )
           , [] );
```

Delimično poravnavanje$_{3/3}$

```
flatten( [[0,1],2,3,[4,[5,6,7],[8,[9,[10,[11,12],13]]]]], 0 );
// [[0,1],2,3,[4,[5,6,7],[8,[9,[10,[11,12],13]]]]]
flatten( [[0,1],2,3,[4,[5,6,7],[8,[9,[10,[11,12],13]]]]], 1 );
// [0,1,2,3,4,[5,6,7],[8,[9,[10,[11,12],13]]]]
flatten( [[0,1],2,3,[4,[5,6,7],[8,[9,[10,[11,12],13]]]]], 2 );
// [0,1,2,3,4,5,6,7,8,[9,[10,[11,12],13]]]
flatten( [[0,1],2,3,[4,[5,6,7],[8,[9,[10,[11,12],13]]]]], 3 );
// [0,1,2,3,4,5,6,7,8,9,[10,[11,12],13]]
flatten( [[0,1],2,3,[4,[5,6,7],[8,[9,[10,[11,12],13]]]]], 4 );
// [0,1,2,3,4,5,6,7,8,9,10,[11,12],13]
flatten( [[0,1],2,3,[4,[5,6,7],[8,[9,[10,[11,12],13]]]]], 5 );
// [0,1,2,3,4,5,6,7,8,9,10,11,12,13]
```

Mapiranje pa poravnavanje: primer

```
var firstNames = [
  {name:"Jonathan", variations: ["John", "Jon", "Jonny"]},
  {name:"Stephanie", variations: [ "Steph", "Stephy" ] },
  {name:"Frederick", variations: ["Fred", "Freddy"]}
];

firstNames.map( entry => [entry.name, ...entry.variations]);
/* [  ["Jonathan","John","Jon","Jonny"],
      ["Stephanie","Steph","Stephy"],
      ["Frederick","Fred","Freddy"] ] */
```

Mapiranje pa poravnavanje: `flatMap()` (ulančano)

```
//Prva – naivna - implementacija:
var flatMap =
  (mapperFn,arr) =>
        flatten(arr.map(mapperFn), 1);

// Opet pomaže reduce():
var flatMap =
    (mapperFn,arr) =>
       arr.reduce(
         (list,v) =>
                  list.concat(mapperFn(v))
              , [] );
```

Operacije sa više lista: `zip()`

- Postoje situacije u kojima je potrebno procesirati više lista istovremeno.
- Jedna od poznatih operacija, zvana `zip()`, alternativno selektuje vrednosti iz jedne i druge ulazne liste u pod-listu:

[1,3,5,7,9], [2,4,6,8,10]→[[1,2], [3,4], [5,6], [7,8], [9,10]]

- Ako su ulazni nizovi različite dužine, završava kada iscrpe kraći niz:

[1,3,5], [2,4,6,8,10]→[[1,2], [3,4], [5,6]]

Implementacija: `zip()`

```
function zip(arr1,arr2) {
   var zipped = [];
   arr1 = [...arr1];
   arr2 = [...arr2];

      while (arr1.length > 0 && arr2.length > 0) {
      zipped.push([ arr1.shift(), arr2.shift() ]);
      }

   return zipped;
}
let zipovano = zip([1,3,5,7,9],[2,4,6,8,10])
// [ [1,2], [3,4], [5,6], [7,8], [9,10] ]
```

Operacije sa više lista: Spajanje lista

- Spajanje lista je transformacija koja formira "ravnu" rezultujuću listu preplitanjem vrednosti iz dve liste:

([1,3,5,7,9], [2,4,6,8,10])→[1,2,3,4,5,6,7,8,9,10]

- To je slično kompoziciji funkcija `flatten()` i `zip()`:
- Prvo `zip()` :

```
zip([1,3,5,7,9], [2,4,6,8,10]);
// [ [1,2], [3,4], [5,6], [7,8], [9,10] ]
```

- ,Onda `flatten()`:

```
flatten([[1,2], [3,4], [5,6], [7,8], [9,10]]);
// [1,2,3,4,5,6,7,8,9,10]
```

Spajanje lista – jedna implementacija

```
function mergeLists(arr1,arr2) {
   var merged = [];
       arr1 = [...arr1];
       arr2 = [...arr2];

       while (arr1.length>0 || arr2.length>0) {
          if (arr1.length>0) {
              merged.push( arr1.shift() );
          }
          if (arr2.length>0) {
              merged.push( arr2.shift() );
          }
       }

   return merged;
}
let spojeno = mergeLists([1,2,3,5,7,7,9],[2,4,5,6,7,8,10])
```

Liste: metode tipa `Array` i samostalne funkcije

- Unificiranje strategije za rad samostalnim funkcijama i metodama prototipa `Array` u JavaScript-u su čest izvor frustracije za programere.
- Tu oba API stila rade isti zadatak ali imaju vrlo različitu "ergonomiju":

```
// Stil 1: metode
[1,2,3,4,5]
   .filter(isOdd)
   .map(double)
   .reduce(sum, 0);  // 18

// Stil 2: samostalne funkcije
reduce(map(filter([1,2,3,4,5],isOdd ), double),
                    sum,
                    0); // 18
```

Komponovanje lanaca metoda

- Metode prototipa `Array` primaju implicitno argument `this` pa se, bez obzira na svoj formalni oblik, ne mogu tretirati kao unarne.
  - Da bi se izborilo sa tim, prvo je potrebna `this`-svesna verzija funkcije `partial()`.
  - Kao drugo, potrebna je i verzija funkcije `compose()` koja poziva svaku parcijalno primenjenu metodu u kontekstu lanca - njena ulazna vrednost koja je "prosleđena" (putem implicitnog `this`) iz prethodnog koraka.

`this`-svesna verzija funkcije `partial()`

```
var partialThis = (fn,...presetArgs) =>
/* function da bi se omogućilo 'this'-povezivanje */
    function partiallyApplied(...laterArgs ) {
      return fn.apply(this,[...presetArgs,...laterArgs]);
   };
```

`this`-svesna verzija funkcije `compose()`

```
var composeChainedMethods =
    (...fns) =>
        result =>
            fns.reduceRight(
                (result,fn) =>
                    fn.call(result)
                , result
            );
```

Kako se to koristi

```
composeChainedMethods(
partialThis( Array.prototype.reduce, sum, 0),
partialThis( Array.prototype.map, double),
partialThis( Array.prototype.filter, isOdd)
)
( [1,2,3,4,5] );   // 18
```

- Kako radi

```
partialThis( Array.prototype.filter, isOdd) /* [1,2,3,4,5]→
[1,3,5] */
partialThis( Array.prototype.map, double), /* [1,3,5] →[2,6,10]
*/
partialThis( Array.prototype.reduce, sum, 0),/* [2,6,10] →
2+6+10 = 18 */
```

Komponovanje samostalnih pomoćnih funkcija

- Kompozicija samostalnih funkcija ne zahteva sve te "`this` bravure", što je i najvažnija prednost.
- Primer definicije samostalnih komponenti:

```
var filter = (arr,predicateFn) =>
  arr.filter(predicateFn);

var map = (arr,mapperFn) => arr.map(mapperFn);

var reduce = (arr,reducerFn,initialValue) =>
        arr.reduce( reducerFn, initialValue );
```

Kompozicija: mora desna parcijalna aplikacija

```
compose(
    partialRight(reduce, sum, 0),
    partialRight(map, double),
    partialRight(filter, isOdd)
)
( [1,2,3,4,5] );   // 18
```

Prilagođavanje metoda samostalnim komponentama

- U definicijama `filter()`/`map()`/`reduce()` ima zajednički obrazac: one se sve dispečiraju na odgovarajući nativni `Array` metod.
- Bilo bi zgodno generisati te samostalne adaptacije nekim pomoćnim programom.
- Sledi taj pomoćni program zvani `unboundMethod()`

Adapter unboundMethod()

```
var unboundMethod =
    (methodName,argCount=2) =>
        curry(
          (...args) => {
              var obj = args.pop();
              return obj[methodName]( ...args );

            },
              argCount
        );
```

Adapter **unboundMethod()**: **korišćenje**

```
var filter = unboundMethod( "filter", 2 );
var map    = unboundMethod( "map", 2 );
var reduce = unboundMethod( "reduce", 3 );

compose(
reduce( sum )( 0 ),
map( double ),
filter( isOdd )
)
( [1,2,3,4,5] ); // 18
```

Prilagođavanje samostalnih komponenti metodama

- Ako se daje prednost radu samo sa `Array` metodama, postoje dva izbora:
  1. Proširiti ugrađeni `Array.prototype` dodatnim metodama. **NIKADA OVO NEMOJTE DA RADITE!!!**
  2. Adaptirati samostalni pomoćni program da radi kao reduktorska funkcija i proslediti ga instanci metode `reduce ()`.

Prilagođavanje samostalnih komponenti metodama: Adaptacija komponente

- Komponenta koju adaptiramo

```
var flatten =
  arr =>
    arr.reduce(
      (list,v) =>
list.concat( Array.isArray(v) ?
                    flatten(v) : v )
              , [] );
```

Adaptacija komponente: Kako

```
/* namerno function da bi se omogućila
  rekurzija po imenu */

function flattenReducer(list,v) {
return list.concat(
      Array.isArray( v ) ? v.reduce(
      flattenReducer, [] ) : v
   );
}
//  Korišćenje nakon adaptacije (koristi se kao metoda):
console.info(
[ [1, 2, 3], 4, 5, [6, [7, 8]] ]
.reduce( flattenReducer, [] )
) // [1, 2, 3, 4, 5, 6, 7, 8]
```

Primeri korišćenja lista i operacija nad njima

Primer deklarativno modelovanje koda 1/8

- Prikazaćemo primer kojim se imperativni kod zapisuje u deklarativnom obliku koristeći operacije sa listama.
- Imperativni kod koji posmatramo je:

```
var getSessionId = partial( prop, "sessId" );
var getUserId = partial( prop, "uId" );

var session, sessionId, user, userId, orders;
session = getCurrentSession();
if (session != null) sessionId = getSessionId(session);
if (sessionId != null) user = lookupUser(sessionId);
if (user != null) userId = getUserId(user);
if (userId != null) orders = lookupOrders(userId);
if (orders != null) processOrders(orders);
```

Primer deklarativno modelovanje koda 2/8

- Ovde ima baš mnogo if-ova sa istim uslovom:

```
if (session != null) sessionId = getSessionId(session);
if (sessionId != null) user = lookupUser(sessionId);
if (user != null) userId = getUserId(user);
if (userId != null) orders = lookupOrders(userId);
if (orders != null) processOrders(orders);
```

Možemo da napravimo funkciju koja će da proverava if uslov:

```
var guard =
    fn =>
       arg =>
           arg != null ? fn( arg ) : arg;
```

Primer deklarativno modelovanje koda 3/8

- Ovde ima baš mnogo if-ova sa istim uslovom:

```
if (session != null) sessionId = getSessionId(session);
if (sessionId != null) user = lookupUser(sessionId);
if (user != null) userId = getUserId(user);
if (userId != null) orders = lookupOrders(userId);
if (orders != null) processOrders(orders);
```

- Sada možemo ovo da zapišemo kao niz funkcija nad kojim se vrši provera uslova:

```
[getSessionId,lookupUser,getUserId,lookupOrders, processOrders]
    .map(guard) // proverava if uslov za svaku funkciju u nizu i
                // vraća poziv funkcije ako je uslov true
```

Primer deklarativno modelovanje koda 4/8

- Šta smo ovim dobili: Dobili smo niz poziva koje treba komponovati.
- Namerno ćemo ih komponovati korišćenjem operacije nad listama `reduce()` koristeći vrednost sesije iz `getCurrentSession()` kao početnu vrednost:

```
reduce(
    (result,nextFn) => nextFn( result )
    , getCurrentSession()
))
```

Primer deklarativno modelovanje koda 5/8

- `getSessionId()` i `getUserId()` mogu se predstaviti kao mapiranje odgovarajućih vrednosti "sessId" i "uId":

```
["sessId", "uId"].map(propName =>
partial(prop,propName))
```

- Obratite pažnju na imperativni kod koji poziva funkcije

```
if (session != null) sessionId = getSessionId(session);
if (sessionId != null) user = lookupUser(sessionId);
if (user != null) userId = getUserId(user);
if (userId != null) orders = lookupOrders(userId);
if (orders != null) processOrders(orders);
```

- On zahteva da se pozove **`getSessionId`** pre nego što se pozove **`lookupUser`**, odnosno da se pozove **`getUserId`** pre nego što se pozove **`lookupOrders`** koji, zauzvrat, mora biti pozvan pre nego što se pozove **`processOrders`** .

Primer deklarativno modelovanje koda$_{6/8}$

- Naš kod zahteva da se pozove **`getSessionId`** pre nego što se pozove **`lookupUser`**, odnosno da se pozove **`getUserId`** pre nego što se pozove **`lookupOrders`** koji, zauzvrat, mora biti pozvan pre nego što se pozove **`processOrders`** .
- To znači da ćemo funkcije **`getSessionId()`** i **`getUserId()`** morati da preplićemo sa druge tri funkcije (`lookupUser()`, `lookupOrders()` i `processOrders()`) da bi se dobio niz od pet funkcija koje treba proveravati na != null uslov i komponovati.
- Pitanje: Kako modelovati to preplitanje?

Primer deklarativno modelovanje koda 7/8

- Pitanje: Kako modelovati to preplitanje?
- Odgovor: Spajanjem lista za šta je kod

```
const mergeReducer =
          (merged,v,idx) =>
                (merged.splice( idx * 2, 0, v ), merged);
```

- Koristimo `reduce()` da se „ubaci“ `lookupUser()` u niz između funkcija `getSessionId()` i `getUserId()`, spajanjem dve liste:

```
reduce( mergeReducer, [ lookupUser ] )
```

- Na kraju se još `lookupOrders()` i `processOrders()` dodaju na kraj niza funkcija:

```
concat( lookupOrders, processOrders ).
```

Primer deklarativno modelovanje koda 8/8

- Rezultat je deklarativni kod:

```
// Lista funkcija koje treba komponovati
    [ "sessId", "uId" ].map( propName => partial( prop, propName
    ) )
       .reduce( mergeReducer, [ lookupUser ] )
       .concat( lookupOrders, processOrders )
// Provera null uslova i kompozicija
[ "sessId", "uId" ].map( propName => partial( prop, propName ) )
   .reduce( mergeReducer, [ lookupUser ] )
   .concat( lookupOrders, processOrders )
   .map( guard )
   .reduce(
      (result,nextFn) => nextFn( result )
      , getCurrentSession()
);
```

Fuzija susednih operacija

```
// Stvari na koje često nailazite u kodu:
..
.filter(..)
.map(..)
.reduce(..);

// A još češće nešto ovako
NekaLista
.filter(..)
.filter(..)
.map(..)
.map(..)
.map(..)
.reduce(..);
```

Šta je tu dobro a šta nije

- Dobro: stil lanca je deklarativan i lako je pročitati konkretne korake koji će se izvršavati po zadatom redosledu (`filter`, pa `filter`, pa `map`, ...).
- Loše: svaka od ovih operacija prolazi kroz celu listu, što smanjuje performansu, posebno ako je lista duža.
- A mi hoćemo da nam programi budu performantni.

Operacije sa listama: fuzija

- Fuzija je postupak spajanja susednih operacija sa ciljem smanijvanja broja prolazaka kroz listu.
- Dakle: da umesto

```
NekaLista
   .filter(..)
   .filter(..)
   .map(..)
   .map(..)
   .map(..)
   .reduce(..);
```

- dobijemo nešto ovako

```
NekaLista
   .filter1(..)
   .map1(..)
   .reduce(..);
```

Operacije sa listama: fuzija **map** operacija

```
const removeInvalidChars = str => str.replace( /[^\w]*/g, "" );
const upper = str => str.toUpperCase();
const elide = str =>
   str.length > 10 ?
      str.substr( 0, 7 ) + "..." :
      str;

var words = "Mr. Jones isn't responsible for this disaster!"
   .split( /\s/ );
words;
// ["Mr.","Jones","isn't","responsible","for","this","disaster!"]

words
   .map( removeInvalidChars )
   .map( upper )
   .map( elide );
// ["MR","JONES","ISNT","RESPONS...","FOR","THIS","DISASTER"]
```

Operacije sa listama: fuzija `map` operacija

- Šta se dešava u ovom toku transformacija?
  - Prva vrednost u spisku reči počinje kao "Mr.", postaje "Mr", zatim "MR", a zatim prelazi kroz `elide()` nepromenjena.
  - Još jedan tok podataka: "responsible" -> "responsible" -> "RESPONSIBLE" -> "RESPONS..."..
- Dakle, ove transformacije podataka mogle bi se formulisati ovako:

```
elide( upper( removeInvalidChars( "Mr." ) ) ); //
"MR"
elide( upper( removeInvalidChars( "responsible" ) )
); // "RESPONS..."
```

Operacije sa listama: fuzija **map** operacija

- Transformacije podataka formulisane su ovako:

```
elide( upper( removeInvalidChars( "Mr." ) ) ); // "MR"
elide( upper( removeInvalidChars( "responsible" ) ) ); // "RESPONS..."
```

- Šta je tu poenta?
  - Mogu se izraziti tri odvojena `map()` poziva kao kompozicija dve funkcije mapiranja (funkcije `upper()` i `removeInvalidChars()`) jer su to unarne funkcije i svaka vraća vrednost koja je prikladan ulaz za sledeću.
  - Mogu se spojiti funkcije mapiranja koristeći `compose()`, a zatim komponovana funkcija proslediti u jedan `map()` poziv:

```
words
    .map(
        compose( elide, upper, removeInvalidChars )
/* može i ovako
        pipe(removeInvalidChars, upper, elide) */

    );
```

Operacije sa listama: fuzija `filter()` i `reduce()` operacija

- Iako spajanje dva ili više `filter()` predikata tipično tretiranih kao unarne funkcije izgleda pogodno za kompoziciju, nije baš sasvim jednostavno zbog toga što svaki predikat vrati tip `boolean` koji ne odgovara sledećoj predikatskoj funkciji kao ulaz.
- Spajanje susednih `reduce()` poziva je takođe moguće, ali je sa reduktorima problem što reduktor prima dve vrednosti na ulazu (inicijalnu vrednost i objekat nad kojim se vrši redukcija) a vraća samo jednu izlaznu vrednost.
- Sve to dovodi do obrasca ***transdjuser*** koga ćemo detaljno obraditi u posebnom predavanju.

Sažetak 1/2

- Za FP su posebno značajne sledeće operacije sa listama
  - `map()`
  - `filter()`
  - `reduce()`
- Ima i naprednijih operacija nad pojedinačnom listom ili nad više lista, na primer:
  - Eliminisanje duplikata iz liste
  - Poravnavanje liste
  - Mapiranje pa poravnavanje liste
  - Zipovanje lista
  - Spajanje lista
- Jedna od najvažnije vrednosti operacija sa listama u FP-u potiču iz mogućnosti da se kao *operacije sa listama* modeluju sekvence – serije naredbi koje inače *ne izgledaju kao lista.*

Sažetak 2/2

- U jeziku JavaScript operacije nad listama javljaju se u dva oblika: (1) kao samostalne funkcije i kao metode prototipa `Array`.
- Zarad udobnijeg programiranja i kvalitetnijeg koda u jeziku JavaScript potrebno je unificiranje strategije za rad sa samostalnim funkcijama i metodama prototipa `Array` što zahteva:
  - Prilagođavanje metoda na samostalne funkcije
  - Prilagođavanje samostalnih funkcija na metode
- Korišćenje lista može da doprinese podizanju performanse koda
  - To se postiže tako što će se uzastopne (iste) operacije nad listama spajati tako da se minimizuje broj prolazaka kroz listu.

Literatura za predavanje

1. Z. Konjović, Funkcionalno programiranje, **Tema 06 – Uvod u FP**, slajdovi sa predavanja, dostupni na folderu **FP kurs 2024-25→Files →Slajdovi sa predavanja**
2. A. Aravinth, Beginning Functional JavaScript Functional Programming with JavaScript Using EcmaScript 6, Apress, 2017. (Poglavlje **Being Functional on Arrays**),
3. Kyle Simpson, **Functional-Light JavaScript - Balanced, Pragmatic FP in JavaScript**, Manning, 2018., (Poglavlje **List Operations**)