

# Funkcionalno programiranje

Školska 2023/24 godina

Letnji semestar

# Tema 8: Upravljanje bočnim efektima

# Sadržaj

- Bočni efekti
- Strategije za kontrolu bočnih efekata:
  - Injekcija zavisnosti
  - Odlaganje
- Funktor Effect

# Bočni efekti

- Sve što se može opaziti u stanju programa izvan funkcije koja se izvršava.
- Čiste funkcije ne prave bočne efekte
- Program bez bočnih efekata je čist, ali ništa korisno ne može da uradi: čak ni I/O operacije koje su same po sebi bočni efekat
- Šta činiti?
  - **Praviti bočne efekte, ali svesno i u kontrolisanim uslovima**



# Strategije za kontrolisanje bočnih efekata

- ***Injekcija zavisnosti*** koja bi kolokvijalno mogla da se nazove *prebacivanje problema u komšijsko dvorište*; i
- ***Odlaganje izvršavanja*** što bi kolokvijalno moglo da se nazove *ekstremnim odugovlačenjem*.
  - Korišćenje Effect funktora

# Injekcija zavisnosti<sub>1/2</sub>

```
/* Ja sam kaos po pitanju čistoće. Pozivam funkciju Date() za koju niko ne garantuje da je čista a, zatim, i metodu toISOString() kakva god ona bila. Radim I/O operaciju koja je sama po sebi bočni efekat pri čemu još izvršavam funkciju prosleđenu kao argument nad vrednošću prosleđenom kao argument. Vraćam poziv prosleđene funkcije fn koja modifikuje svoj argument */
```

```
function logSomething(something,fn) {
```

```
    const dt = (new Date()).toISOString(); /* pozv Date() a, zatim, poziv metode toISOString() */
```

```
    console.log(`${dt}: ${fn(something)}`); /* I/O operacija + izvršavanje funkcije prosleđene kao argument */
```

```
    return fn(something); // vraćanje poziva prosleđene funkcije fn
```

```
}
```

```
function fun(x) {
```

```
    x = x+12
```

```
    return x
```

```
}
```

# Injekcija zavisnosti<sub>2/2</sub>

- ***Injekcija zavisnosti*** svodi se na to ***da se svaka nečistoća u kodu smesti u parametre funkcije.*** Nakon toga, te ***nečistoće postaju odgovornost drugih funkcija.***

# Funkcija koja proizvodi bočne efekte – ali “čista”

- Signatura

OdgovornaZaSve: Console -> Date -> String -> \*

```
function OdgovornaZaSve (cns1,d,message) {  
    const dt = d.toISOString();  
    return cns1.log(`${dt}: ${message}`)  
}
```

```
OdgovornaZaSve(console,new Date(), `Pozdrav od  
mene, odgovorne za sve!`);
```



# Strategija odlaganja

- *Bočni efekat nije bočni efekat dok se stvarno ne dogodi.*
- Funkcija neće proizvesti bočni efekat dok se ne izvrši
- Strategija je da se izvršavanje funkcije odlaže do “poslednjeg trenutka”



# Idemo na primer: nuklearni rat

```
// fZero :: () -> Number
function fZero() {
    console.log('Lansiram nuklearne rakete');
    // Ovde ide kod za lansiranje
    return 0;
}
```



# Funkcija koja vraća fZero funkciju

```
// fZero :: () -> Number
function fZero() {
  console.log('Lansiram rakete');
  // Ovde ide kod za lansiranje
  return 0;
}

// returnZeroFunc :: () -> (() -> Number)
function returnZeroFunc() {
  return fZero; // ovo nije poziv, samo se vraća
                // funkcija
}

const zeroFunc1 = returnZeroFunc();
const zeroFunc2 = returnZeroFunc();
const zeroFunc3 = returnZeroFunc();

/* Nije lansirana ni jedna raketa jer nije izvršena funkcija
   fZero() */
```



# Skroz čista funkcija

```
// returnZeroFunc :: () -> (() -> Number)
function returnZeroFunc() {
  function fZero() {
    console.log('Lansiram rakete!');
    // Ovde ide kod za lansiranje
    return 0;
  }
  return fZero; // vraća se uvek nova referenca
                // to je naša prilika
}
```

# Rupa u zakonu

- `returnZeroFunc()` će uvek vratiti novu referencu funkcije.
- Međutim, inspekcijom koda vidi se da `returnZeroFunc()` vraća uvek **istu** funkciju (`fZero`).
- Kako bi se to moglo iskoristiti?



# Korišćenje rupe u zakonu: Ideja 1

- Prvo ćemo da probamo da iskoristimo nulu koju vraća `fZero()` a da ne započnemo nuklearni rat.
- Kreiraćemo funkciju koja prima nulu koju vraća `fZero()` i na nju dodaje 1:

```
// fIncrement :: (() -> Number) -> Number
function fIncrement(f) {
    return f() + 1;
}
```

```
fIncrement(fZero);
// ] Lansiram rakete
// ← 1
```

# Korišćenje rupe u zakonu: Ideja 2

- Nećemo vratiti broj, nego **funkciju** koja će u krajnjoj instanci (kada se pozove) vratiti broj:

```
// fIncrement :: (() -> Number) -> (() -> Number)
function fIncrement(f) {
    return () => f() + 1;
}
```

```
fIncrement(fZero);
let v1 = fIncrement(fZero)
console.log ('Vratio sam:',v1); // Vratio sam: () => f() + 1
/* nema loga na konzoli, nema nuklearnog rata. Prava stvar! */
```

```
let v2 = v1()
console.log ('Vratio sam:',v2);
// Lansiram nuklearne rakete
// // Vratio sam: 1
```

- Sada sa ovim funkcijama možemo da kreiramo koliko god hoćemo funkcija koje **broj vraćaju tek na kraju**.

# Množi, stepenuj, korenuj ali ne odmah

- A možemo i da napravimo gomilu funkcija koje rade sa takvim brojevima:

```
// fMultiply :: (() -> Number) -> (() -> Number) -> (() -> Number)
function fMultiply(a, b) {
    return () => a() * b();
}
```

```
// fPow :: (() -> Number) -> (() -> Number) -> (() -> Number)
function fPow(a, b) {
    return () => Math.pow(a(), b());
}
```

```
// fSqrt :: (() -> Number) -> (() -> Number)
function fSqrt(x) {
    return () => Math.sqrt(x());
}
```

.....



# Množi, stepenuj, korenuj

// Ali ne odmah

```
const fFour = fPow(fTwo, fTwo);  
console.log (fFour) //() => Math.pow(a(), b())  
const fEight = fMultiply(fFour, fTwo);  
const fTwentySeven = fPow(fThree, fThree);  
const fNine = fSqrt(fTwentySeven);
```

// A sada stvarno sračunaj

```
function fTwo () {return 2}  
console.log (fFour()) // 4
```

# Šta smo mi to radili

- Napravili ono što matematičari zovu *izomorfizam* (U matematici, **izomorfizam** je mapiranje između dve strukture istog tipa takvo da postoji inverzno mapiranje).
- Naše strukture su regularni brojevi i "odloženi" brojevi.
- Sve što možemo da radimo sa regularnim brojevima, možemo i sa "odloženim".
- Regularan broj dobijamo pozivom funkcije.
- Dakle: Vršimo mapiranje između regularnih i odloženih brojeva - i vrednosti i operacije.

# A šta smo uradili

- Ovakvo pakovanje funkcija je legitimna strategija:
  - Možemo se sakrivati iza funkcija dokle god želimo.
  - I sve dok stvarno ne izvršimo konačan poziv, one su “čiste” – preciznije, ne proizvode bočni efekat.
  - Tek kada ih pozovemo, one će proizvesti bočni efekat.
  - Ali ćemo ih pozivati samo ako želimo da proizvedemo taj bočni efekat.
- ***NAPRAVILI SMO MEHANIZAM KOJIM BOČNE EFEKTE MOŽEMO DA DRŽIMO POD KONTROLOM!***

# A šta tu nije baš najbolje i kako može bolje

- Ovakav kakav je ovaj način zahteva da stalno pravimo nove funkcije što nije baš neki posao.
- Može i lepše. Mehanizam koji bi nam omogućio da obične (ugrađene) funkcije rade sa ovim odloženim vrednostima.
- Šta bi bila osnova tog mehanizma? Pa, naravno, ***funktor***

# Funktor Effect

- Funktor Effect nije ništa drugo do objekat u koji se umeću "odložene" funkcije.
- Pokazaćemo to na primeru funkcije fZero koju ćemo umetnuti u Effect objekat.
- Za početak nam treba funkcija za umetanje fZero
- Zatim nam treba objekat u koji ćemo da je umetnemo (naš funktor Effect)

# Evo ih: fZero i Effect konstruktor

```
// zero :: () -> Number
function fZero() {
    console.log('Počinjemo ni sa čim');
    /* Definitivno ovde ne pokrećemo nuklearni
    napad. Ali ova funkcija je i dalje nečista. */
    return 0;
}

// Effect :: Function -> Effect
function Effect(f) {
    return {};
}
```



# A sada da ih “udružimo”

```
// Effect :: Function -> Effect
function Effect(f) {
  return {
    map(g) {
      return Effect(x => g(f(x)));
    }
  }
}
```



# A sada možemo i da lansiramo rakete

```
// Effect :: Function -> Effect
function Effect(f) {
  return {
    map(g) {
      return Effect(x => g(f(x)));
    },
    runEffects(x) {
      return f(x);
    }
  }
}
```





# Pa da lansiramo

```
const zero = Effect(fZero);  
const increment = x => x + 1; /* Samo  
    regularna funkcija. */  
const one = zero.map(increment);  
one.runEffects();  
// Počinjemo ni sa čim  
// 1
```



# I sada imamo funktor

```
const double = x => x * 2;  
const cube = x => Math.pow(x, 3);  
const eight = Effect(fZero)  
    .map(increment) // 1 = 0+1  
    .map(double) // 2 = 1*2  
    .map(cube); // 8 = 2*2*2  
eight.runEffects();  
// Počinjanje sa nulom  
// 8
```



# On može da radi ovakve stvari

```
const incDoubleCube = x =>  
  cube(double(increment(x)));
```

- Ako se koriste biblioteke (Ramda ili Lodash/fp) može i ovako:

```
const incDoubleCube = compose(cube, double,  
  increment);  
const eight =  
  Effect(fZero).map(incDoubleCube);
```

# Upravljanje bočnim efektima: sažetak

- Prikazali smo dve strategije za rukovanje nečistotama u funkcijama:
  - Injekcija zavisnosti; i
  - Strategija odlaganja - Effect funktor.
- Injekcija zavisnosti radi tako što nečiste delove koda izmešta iz funkcije. Dakle oni se prosleđuju kao parametri.
- Strategija odlaganja radi tako što umotava sve iza (jedne) omotačke funkcije. Za pokretanje efekata mora se učiniti nameran potez kojim se pokreće ta omotačka funkcija.
- Funktor Effect omogućuje upravo to: samo vraća funkcije (bez izvršavanja) dok se svesno ne povuče “okidač” za izvršavanje.
- Ni jedan pristup ne uklanja nečistote (bez njih programi ne bi ništa radili), već ih samo smeštaju u druge delove koda ali ih, što je važno, čine na taj način eksplicitnim i jasno vidljivim.

# Literatura za predavanje

1. Z. Konjović, Funkcionalno programiranje, **Tema 08 – Upravljanje bočnim efektima**, slajdovi sa predavanja, dostupni na folderu **FP kurs 2023-24 → Files → Slajdovi sa predavanja**
2. J. Sinclair, HOW TO DEAL WITH DIRTY SIDE EFFECTS IN YOUR PURE FUNCTIONAL JAVASCRIPT, dostupno na adresi:  
<https://jrsinclair.com/articles/2018/how-to-deal-with-dirty-side-effects-in-your-pure-functional-javascript/>