

# Funkcionalno programiranje

Školska 2023/24 godina

Letnji semestar

# Tema 7: Funktori, aplikativni funktori i monade

# Sadržaj

- Slikovnica:
  - O kontekstu - tipu
  - funktori,
  - aplikativni funktori,
  - monade
- Manje slika, više koda:
  - Funktor
    - Šta je
    - Zašto
    - Kako
  - Monada
    - Šta je
    - Zašto
    - Kako
  - Aplikativ – između funktora i monade
- Da uopštimo

# Slikovnica

# O kontekstu -tipu

# Vrednosti i tip

- Počinjemo sa jednostavnom vrednošću

2  
↑  
VALUE

- Znamo kako da primenimo funkciju (funkcija je: **dodaj 3**) na tu jednostavnu vrednost:



# A sada kontekst -tip

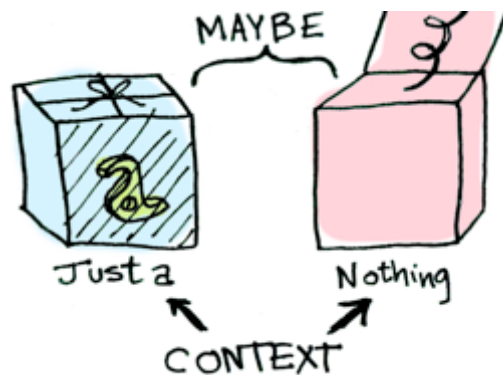
- **Svakoj vrednosti dodeljuje se tip - kontekst.** Za početak, zamislimo da je **kontekst** “kutija” u koju se **vrednost** može staviti:



- Ako se sada primeni funkcija na tu vrednost, mogu da se dobiju **različiti rezultati** u zavisnosti od **kutije u kojoj je vrednost sadržana**.
- To je ideja na kojoj su zasnovani *funktori*, *aplikativi* (*aplikativni funktori*), *monade*, itd.

# Tip Maybe (iz jezika Haskell)

- Enkapsulira opcionu vrednost koja je ili **vrednost tipa a** (Just a) ili prazno (Nothing).
- On predstavlja kontejner ( “kutiju”) u koju može da se stavi nešto.

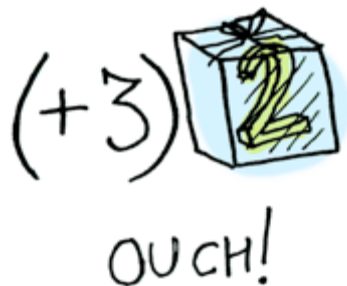


- U jeziku JavaScript nema tipa kao što je Maybe, ali se on, naravno, može posebno napraviti.
- Za ilustraciju kontekstne “kutije” u JavaScript-u dobro može da posluži i tip Array.



# Funktor: “kutija” + map

- Kada su vrednosti u kontekstu ne može se na njih direktno primeniti funkcija:



- **map** je otvarač za kontekstnu “kutiju”. **map** zna kako da primeni funkcije na vrednosti koje su spakovane u “kutiju”.

# Šta radi fmap() iz Haskell-a



$\text{fmap} :: (a \rightarrow b) \rightarrow f a \rightarrow f b$

1. `fmap` TAKES A  
FUNCTION  
(LIKE `(+3)`)

`+3`

2. AND A  
FUNCTOR  
(LIKE `Just 2`)

`[2]`

3. AND RETURNS  
A NEW FUNCTOR  
(LIKE `Just 5`)

`[5]`

# Funktor

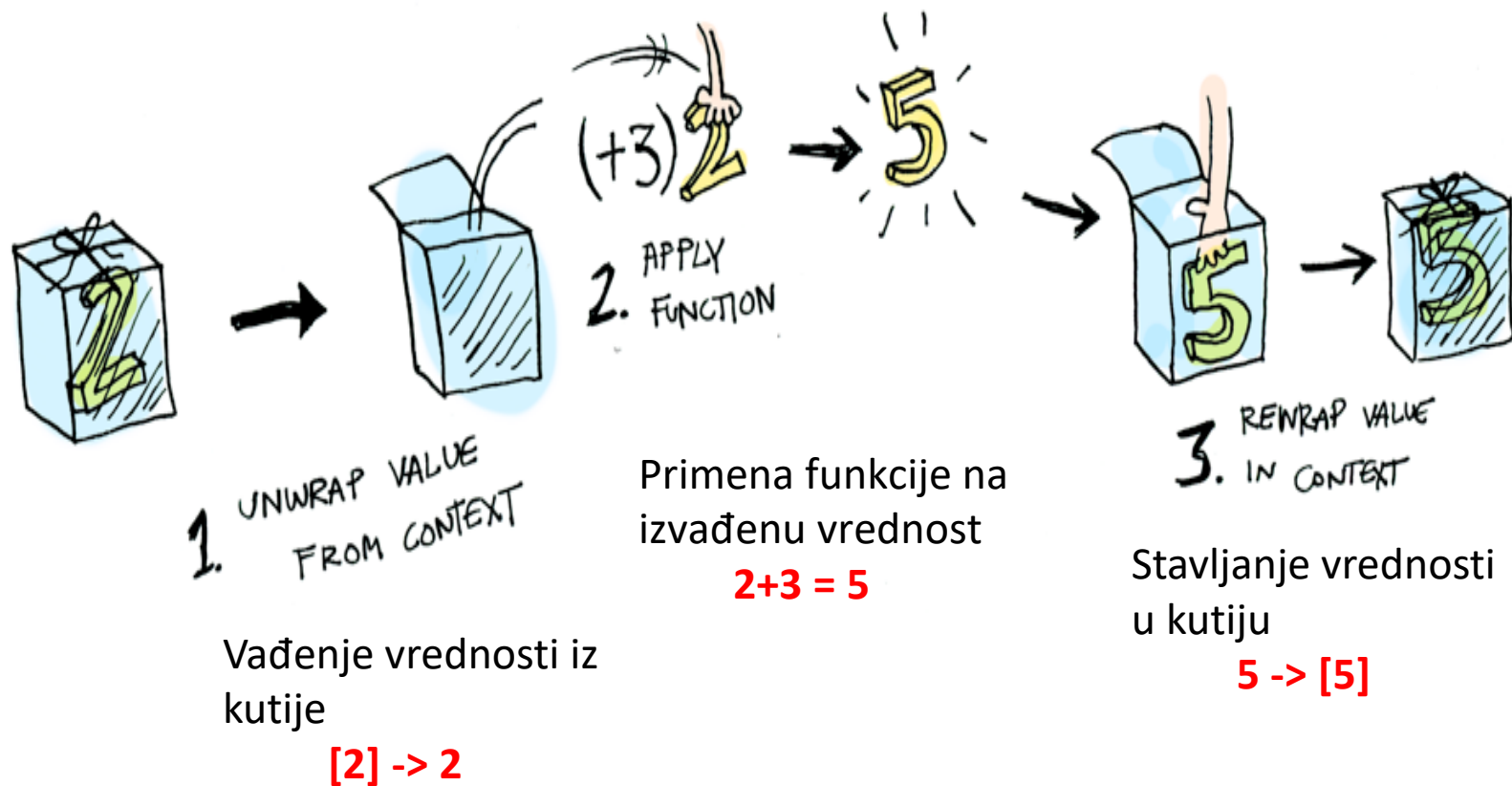
```
class Functor f where  
    fmap :: (a -> b) -> fa -> fb
```

```
fmap (+3) (Just 2)  
→ Just 5
```



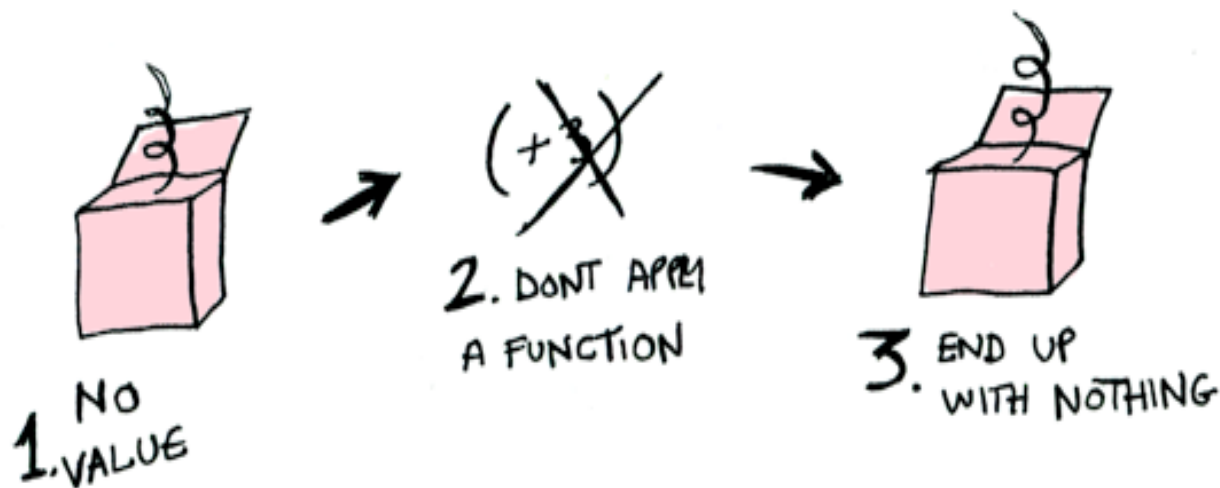
# A šta je “magija”?

- Nema nikakve magije, dešava se ono što može:



# Prazan kontejner

- Ako je **kontejner prazan**, maperska funkcija **se ne izvršava**:

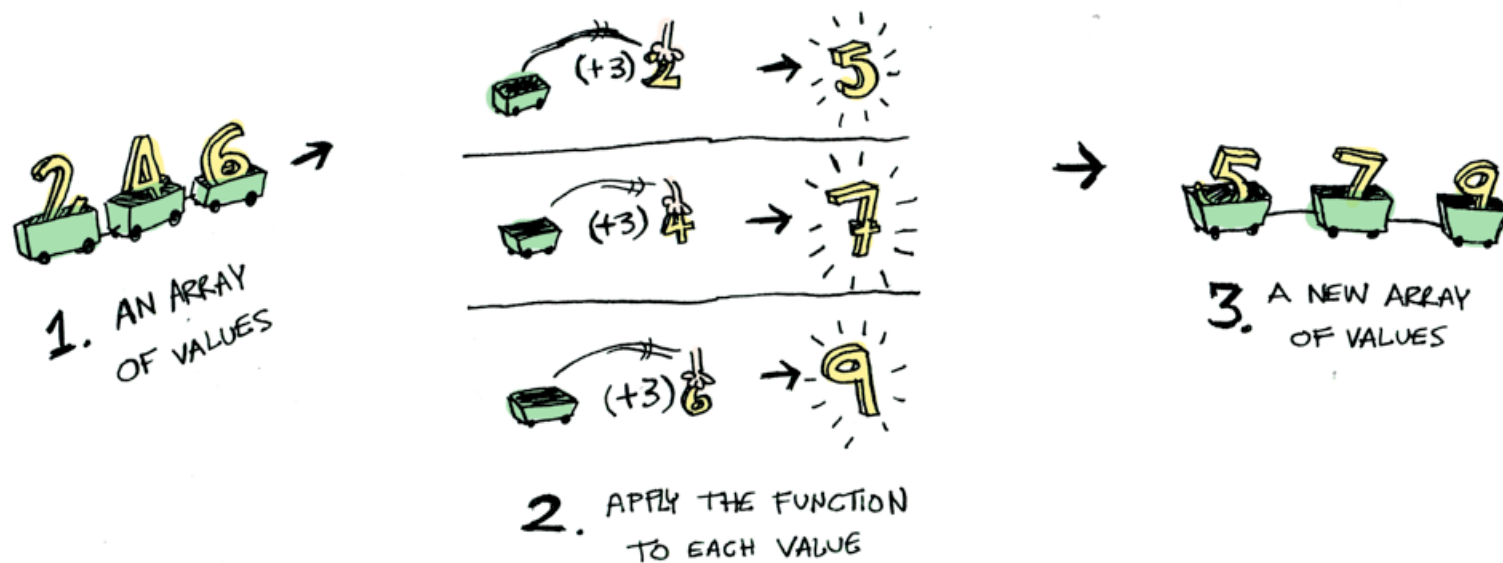


# Prazan kontejner: stvarno se ne primenjuje

```
var plus3AndPrintValue = function(val) {  
  var result = val + 3;  
  console.log(result);  
  return result;  
}  
const arr1 = [2]  
const arr2 = []  
arr1.map(plus3AndPrintValue); // '5'  
arr2.map(plus3AndPrintValue); // []
```

# Array tip

- Kada imamo Array tip sa više vrednosti, prirodno se primenjuje funkcija na svaku vrednost ulaznog niza i pravi se novi niz sa elementima koji su rezultat primene funkcije:



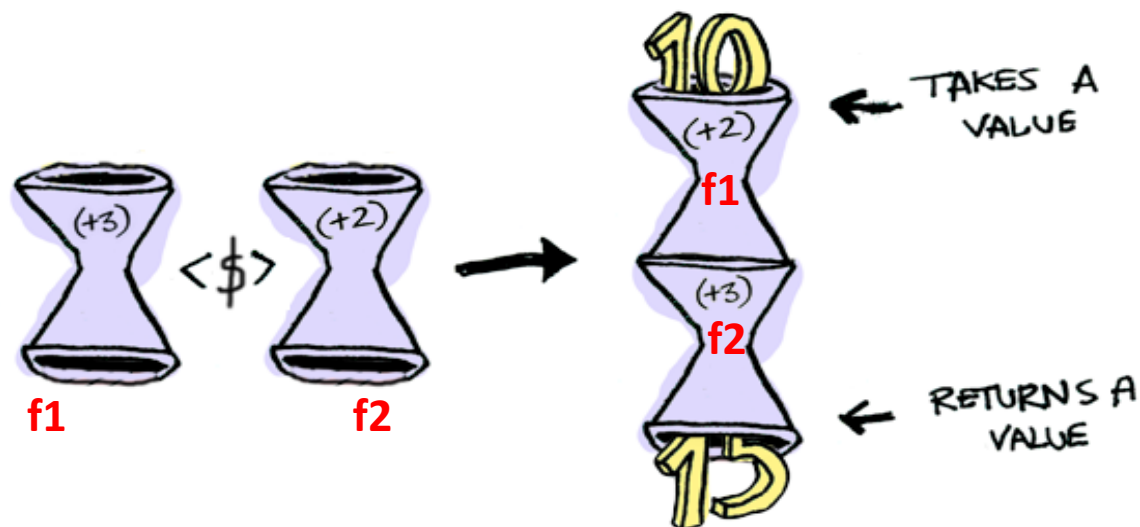
# Prazan kontejner: stvarno se ne primenjuje

```
const plus3AndPrintValue = function(val) {  
  var result = val + 3;  
  console.log(result);  
  return result;  
}  
const arr1 = [2, 4, 6]  
arr1.map(plus3AndPrintValue); // [5, 7, 9]  
const arr2 = []  
arr2.map(plus3AndPrintValue); // []
```



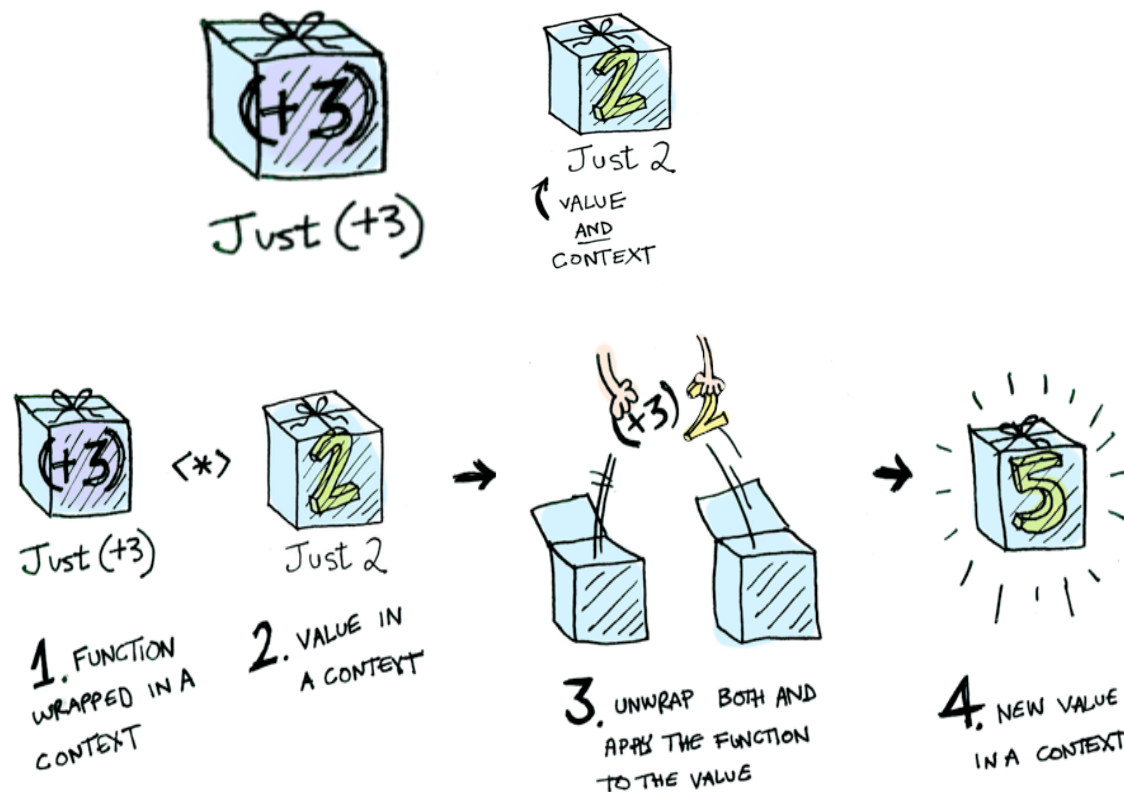
# Kompozicija funktora

- Ulaz u funktor može da bude i vrednost tipa **Function** – funkcija
- Rezultat primene je prosto druga funkcija. U JavaScript-u to je ***kompozicija funkcija***.



# Aplikativi (aplikativni funktori)

- Funkcija "spakovana" u kontekst, vrednost "spakovana" u kontekst



# Aplikativni funktor: samo malo detalja

- Dakle, aplikativni funktor je, u stvari, **funktor koji radi nad fuktorom**.

- Prima: **funktor**
- Vraća: **funktor**

- Signatura:

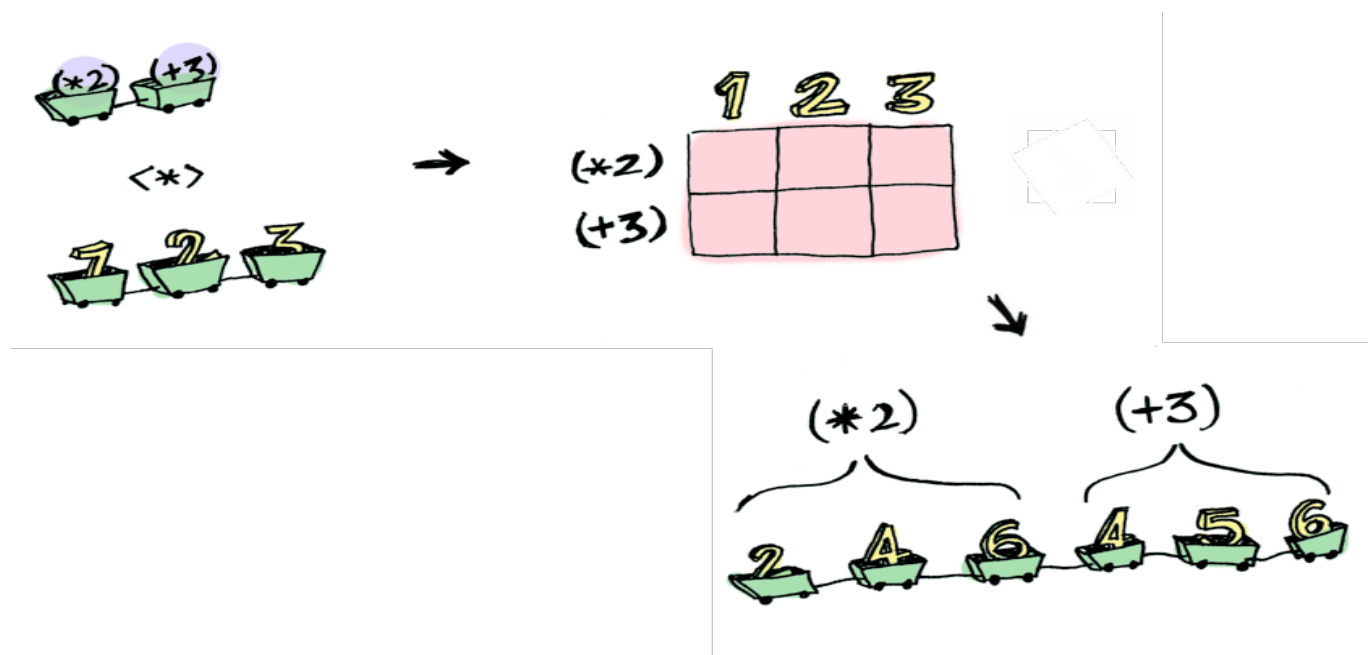
`fmap :: (a -> b) -> fa -> fb`

`fap1 :: f (a -> b) -> f a -> f b`

Takva stvarčica nam omogućuje da funkciju koja je, na primer, “spakovana” u niz (recimo, `[f1, f2, f3]`) direktno (nama to tako izgleda) primenimo na vrednost spakovanu takođe u niz (recimo, `[3, 3, 5, 6, 7]`).

- Nešto ovako (nije kod, samo ilustracija):  
`[fn]([3])`

# Aplikativni funktor: više “upakovanih” funkcija - kompozicija

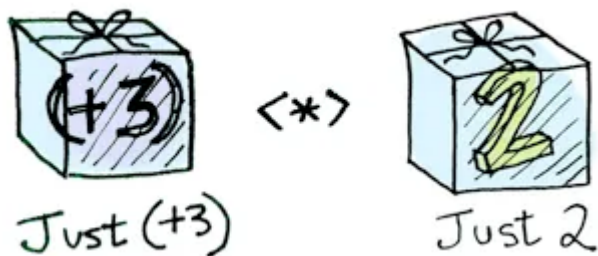


$$[f1, f2] \langle * \rangle [1, 2, 3] = [ [f1] \langle * \rangle [1], [f1] \langle * \rangle [2], [f1] \langle * \rangle [3], [f2] \langle * \rangle [1], [f2] \langle * \rangle [2], [f2] \langle * \rangle [3] ]$$

# Monada<sub>1/2</sub>



Funktor primenjuje neupakovanu funkciju na upakovanu vrednost i vraća upakovan rezultat. **Kompozicija** se direktno izvodi ako je ulaz tipa Function.

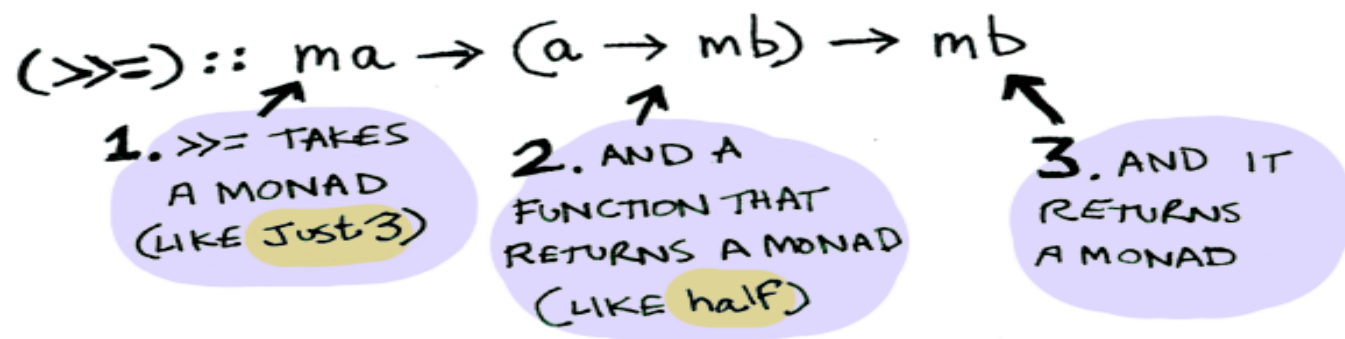


Aplikativni funktor primenjuje upakovanu funkciju na upakovanu vrednost i vraća upakovan rezultat

Monade na upakovanu vrednost primenjuju funkciju koja prima neupakovanu vrednost i vraća izlaz u upakovanom obliku, i vraćaju rezultat u upakovanom obliku. Monade imaju funkciju bind ( $>>=$ ) kojom to rade.

# Monada<sub>2/2</sub>

- Monada prima dva argumenta: (1) "upakovanu" ulaznu vrednost i (2) funkciju koja kao ulaz očekuje "neupakovanu" primljenu vrednost i vraća "upakovanu" vrednost svoje primene. Monada treba da vrati "upakovanu" vrednost koja se dobija primenom primljene funkcije na „neupakovanu“ primljenu vrednost.



1. Prima monadičku vrednost tipa **ma**
2. Prima funkciju koja prima jednostavnu vrednosti tipa **a** vraća monadičku vrednost tipa **mb**
3. Primenjuje tu funkciju na vrednost **ma** i vraća vrednost **mb**

# Rezime iz slikovnice

- Funktori primenjuju "neupakovane" funkcije na „upakovane“ vrednosti: moraju da "otpakuju" vrednosti i da na "otpakovane" vrednosti primene funkciju.
- Aplikativni funktori primenjuju "upakovanu" funkciju na "upakovanu" vrednost: moraju da "otpakuju" i vrednosti i funkcije i da na "otpakovane" vrednosti primene "otpakovanu" funkciju.
  - Ni funktori ni aplikative ne mogu da "raspakuju" ono što je spakovano u više "kutija" koje su jedna u drugoj.
- Monade na "upakovanu" vrednost primenjuju funkciju koja prima "neupakovanu" vrednost i vraća "upakovanu" vrednost.
  - Monade mogu da "raspakuju" ono što je spakovano u više "kutija" koje su jedna u drugoj i da to na kraju spakuju na odgovarajući način.

# Manje slika, više koda: Funktori



# Šta je fundamentalni problem

- U računarskim programima u suštini se radi samo jedna stvar: *neke operacije* se primenjuju na *neke vrednosti*.
- Pri tome, ima različitih vrednosti i različitih operacija i ne mogu se sve operacije primenjivati na sve vrednosti.
- Te stvari su u jeziku regulisane *sistemom tipiziranja i tipovima*
- Ipak, ostaju neke situacije koje ugrađeni mehanizmi jezika ne mogu da razreše (baš adekvatno), već ih je potrebno posebno rešavati na nivou aplikacije.
- Sećate se: to je rukovanje greškama
- Naravno, bilo bi dobro ako bismo imali neke instrumente koji bi nam omogućili da *neke operacije* primenjujemo na *neke vrednosti* i da, pri tome, ne strepimo od ishoda.

# Prikladan primer: ne može jednostavije, a odmah problem

- Inkrementiranje vrednosti

```
const increment = v => v + 1;
```

```
const a = 5; console.log( increment(a));
```

```
// 6 - ovo je OK
```

```
// a šta ako je ovako?
```

```
console.log(increment("5"));
```

```
// 51 - Šta je sada ovo?
```

```
// ili ovako?
```

```
console.log( increment({ v: 5 }));
```

```
// [object Object]1 - A šta je tek ovo?
```



# Predlog rešenja: da li je dobar?

```
const increment = v => {  
  if (typeof v !== "number") {  
    return NaN; // NaN ovde odgovara,  
                // proverava se da li je broj  
  }  
  
  return v + 1; // ako je broj, uvećaj ga za 1  
};  
console.log(increment(150)); // 151  
console.log(increment("5")); // NaN  
console.log(increment({ v: 5 })); // NaN
```

# Predlog rešenja: radi i za drugu funkciju – čini se dobar

```
const double = v => {  
  if (typeof v !== "number") {  
    return NaN; // NaN ovde odgovara,  
                // proverava se da li je broj  
  }  
  
  return v * 2; // ako je broj, udvostruči ga  
};  
console.log(double(150)); // 300  
console.log(double("5")); // NaN  
console.log(double({ v: 5 })); // NaN
```

# A zašto to nije dobro ako radi

## Prva funkcija

```
const increment = v => {  
  if (typeof v !== "number") {  
    return NaN;  
  }  
  return v + 1;  
};
```

## Druga funkcija

```
const double = v => {  
  if (typeof v !== "number") {  
    return NaN;  
  }  
  return v * 2;  
};
```

# Da malo analiziramo

- U oba slučaja se manipuliše vrednošću (na nju se primenjuje operacija) i, pri tome se rade neke verifikacije:
  - Da li je tip dobar?
  - Da li je vrednost raspoloživa?
  - Da li se desila greška?
- Nisu retke situacije da vrednost nije na raspolaganju; recimo, pribaviće se iz povratnog poziva a možda i neće, zbog greške.

# A kada još i razmislimo, zaključujemo:

- Mi imamo mentalni model rada sa funkcijama u kome **funkciji prosleđujemo ulazne vrednosti**.
- Te vrednosti su negde uskladištene – postoje mesta/stvari gde se one nlaže.
- Šta bi bilo da stvar obrnemo: da **funkcije prosleđujemo skladištima vrednosti**, odnosno onim mestima/stvarima što te vrednosti sadrže i da omogućimo da to “skladište” vodi računa o onome šta će sa vrednostim dalje da se dešava.
- Zašto bismo tako nešto radili?
- Zato što vrednost već i sama “zna” neke važne stvari o sebi koje utiču na ono što će se desiti kada se funkcija na nju primeni.

# Rezultat je: Funktor

- Ovo što smo videli/zaključili je osnovna ideja programskog obrasca zvanog **Funktor**.
- A ta osnovna ideja je: **“skladište” za vrednosti tako da se funkcija prosleđuje “skladištu” koje zna šta da uradi sa funkcijom.**
- A šta se, uopšte, može uraditi sa funkcijom?
  - Funkcija se može primeniti na vrednost/pozvati sa svojim argumentima
  - Skladište treba da ima instrument kojim će da primenjuje funkciju koja mu je prosleđena.
  - Taj instrument je opet funkcija, funkcija **map()** koja ume da primeni/pozove drugu funkciju.





# Da vidimo na primeru: **kutijaZaBroj**

```
const kutijaZaBroj = number => ({  
  primeniFunkciju: fn =>  
    kutijaZaBroj(fn(number)),  
  value: number,  
});
```

```
kutijaZaBroj(5)  
  .primeniFunkciju(v => v * 2)  
  .primeniFunkciju(v => v + 1).value; // 11
```



# Sada sve važne stvari mogu da budu na jednom mestu

```
const kutijaZaBroj = number => ({  
  primeniFunkciju: fn => {  
    if (typeof number !== "Number") {  
      return kutijaZaBroj(NaN);  
    }  
    return kutijaZaBroj(fn(number));  
  },  
  value: number,  
});  
  
console.log(kutijaZaBroj(17) // value je tipa Number  
  .primeniFunkciju(v => v * 2)  
  .primeniFunkciju(v => v + 1).value); // 35  
  
console.log(kutijaZaBroj("17") // value nije tipa Number  
  .primeniFunkciju(v => v * 2)  
  .primeniFunkciju(v => v + 1).value); // NaN
```



# A fn nije ništa drugo nego map()

```
const kutijaZaBroj = number => ({  
  map: fn => {  
    if (typeof number !== "number") {  
      return kutijaZaBroj(NaN);  
    }  
    return kutijaZaBroj(fn(number));  
  },  
  value: number,  
});
```



# Upotrebljiva primena: provera tipa pre mapiranja

```
const TypeBox = (predicate, defaultValue) =>
    {const TypePredicate = value =>
        ( {map: fn => predicate(value) ?
            TypePredicate(fn(value))
            :
            TypePredicate(defaultValue),
            value,}
        );
        return TypePredicate;
    };
```

# Provera tipa: Number

```
const kutijaZaBroj = TypeBox(value =>
  typeof value !== "Number", NaN);
console.log(kutijaZaBroj(5)
  .map(v => v * 2)
  .map(v => v + 1).value); // 11
console.log(kutijaZaBroj({ v: 5 })
  .map(v => v * 2)
  .map(v => v + 1).value); // NaN
```

# Provera tipa: String

```
const StringKutija = TypeBox(value =>
  typeof value !== "String", null);
StringKutija("world")
  .map(v => "Hello " + v)
  .map(v => "***" + v + "***").value;
// '**Hello, world**'
StringKutija({ v: 5 })
  .map(v => "Hello " + v)
  .map(v => "***" + v + "***").value;
// null
```



# map() omogućuje i kompoziciju funkcija

```
const double = v => v * 2;
const increment = v => v + 1;
// Poziv
kutijaZaBroj(5)
  .map(double)
  .map(increment).value; // izlaz:11
// daje isti rezultat kao:
kutijaZaBroj(5).map(v =>
  increment(double(v))).value; // izlaz: 11
```

# map() ne sme da proizvodi bočne efekte

- Ima još jedna stvar koju map() treba da zadovolji: ***ne sme da pravi bočne efekte.***
- Sme samo da menja *vrednosti* u funktoru i ništa drugo.
- Kako se to može proveriti?
  - Proverom koja pokazuje da mapiranje funkcije identiteta ( $v \Rightarrow v$ ) vraća isti funktor kao što je bio ulazni t.j. mora da važi:  

```
kutijaZaBroj(5).map(v => v) // {value: 5, map: f}  
kutijaZaBroj(5)             // {value: 5, map: f}
```



# Šta je funktor: rezime

- Funktor (functor) je **tip** nad kojim se može mapirati.
- U JavaScript-u funktor je običan objekat (ili klasni tip u drugim jezicima) koji implementira funkciju map koja se izvršava nad **svakom vrednošću u objektu** da bi **proizvela novi objekat**.
- Možemo ga tumačiti kao **kontejner** sa map operacijom koja se može koristiti da bi se mapiranje primenilo na vrednosti u kontejneru.
- U JavaScript-u, functor tipovi su obično predstavljeni kao objekti sa `.map()` metodom koja mapira ulaze na izlaze, n.pr., `Array.prototype.map()`.

# Zašto funktor

- Funktor apstrahuje detalje implementacije strukture podataka koji se mapiraju - programer ne mora da zna da li je potrebno iteriranje, niti na koji način se ono realizuje ako jeste potrebno.
- Funktor sakriva tip podataka koje sadrži – programer na isti način vidi nizove brojeva i nizove stringova, jedino mora da prosledi `map()` funkciju primenljivu na taj tip podatka.
- Mapiranje nad praznim funktorom za programera je isto kao mapiranje i nad nepraznim funktorom.
- Funktori omogućuju jednostavno komponovanje funkcija nad podacima koji su u funktoru.



# Kako funktor: Kontejner

- Kontejner u sebi čuva vrednost:

```
const Container = function(val) {  
  this.value = val;  
}  
// Poziva se sa new da se kreira novi objekat  
let testValue = new Container(3)  
let testObj = new Container({a:1})  
let testArray = new Container([1,2])  
console.log(testValue) // Container{value:3}  
console.log(testObj) // Container{value:{a:1}}  
console.log(testArray) // Container{value:[1,2]}
```

# Kako funktor: tačkasti kontejner

```
/* Dodaje se metoda of Container prototipu da se  
   izbegne new pri pozivanju, u stvari omogućuje se  
   smeštanje vrednosti u kontejner */
```

```
Container.of = function(value) {  
    return new Container(value);  
}
```

```
// Pozivi su sada
```

```
testValue = Container.of(3) // Container{value:3}
```

```
testObj = Container.of({a:1})
```

```
// Container {value:{a:1}}
```

```
testArray = Container.of([1,2])
```

```
// Container {value:[1,2]}
```



# Kako funktor: ugnježdjeni kontejneri

- Container može da sadrži ugnježdjene Container-e:

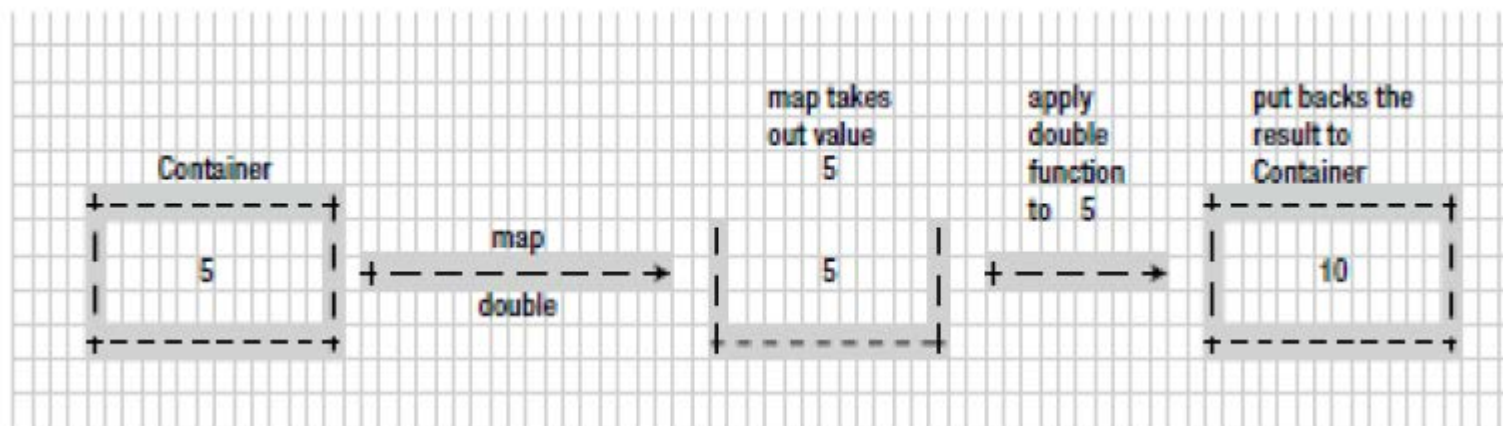
```
Container.of(Container.of(3));
```

- Vratíće:

```
Container {  
  value: Container {  
    value: 3 }  
}
```

# Kako funktor: Kontejner i funkcija map

- Funkcija map omogućuje da se pozove bilo koja funkcija nad vrednošću koja se trenutno nalazi u kontejneru.
- Funkcija map uzima vrednost iz kontejnera (Container), *primenjuje prosleđenu funkciju* na **tu vrednost** i **rezultat vraća** u kontejner(Container).





# Metoda map Container prototipa

```
const Container = function(val) {  
  this.value = val;  
}  
Container.of = function(value) {  
  return new Container(value);  
}  
Container.prototype.map = function(fn){ /* ono što  
  kontejner čini fuktorom */  
  return Container.of(fn(this.value));  
}  
let double = (x) => x + x;  
Container.of(3).map(double)  
/* Vraća: Container { value: 6 } */
```

# Container prototip omogućuje lako ulančavanje

```
const Container = function(val) {  
  this.value = val;  
}  
Container.of = function(value) {  
  return new Container(value);  
}  
Container.prototype.map = function(fn){  
  return Container.of(fn(this.value));  
}  
let double = (x) => x + x;  
Container.of(3).map(double) // Vraća: Container { value: 6 }  
// Omogućuje lako ulančavanje  
Container.of(3).map(double) )// Vraća 12+12 = 24  
                .map(double) )// Vraća 6+6 = 12  
                .map(double) // Vraća 3+3 = 6
```



# Ugovorni dizajn

- ***Ugovorni dizajn*** (Design by contract (DbC), contract programming, programming by contract i design-by-contract programming) je pristup dizajnu softvera.
- Taj pristup nalaže da se definiše formalna, precizna specifikacija interfejsa koja se može verifikovati, što proširuje običnu definiciju apstraktnog tipa *preduslovima*, *postuslovima* i *invarijantama*.
- Te specifikacije se nazivaju ***ugovori*** ("contracts").



# Funktor i ugovorni dizajn

- Funktor je običan objekat (ili klasni tip u drugim jezicima) koji implementira funkciju `map` koja se izvršava nad svakom **vrednošću u objektu** da bi proizvela **novi objekat**.
- Funktor je objekat koji implementira **map** ugovor!
- Funktor je koncept koji zahteva Ugovor, a Ugovor je implementacija funkcije **map**!
- Različiti načini implementacije `map` funkcije daju različite tipove funktora.



# Različite `map()` implementacije => različiti funktori (primeri)

- Funktor `Maybe`
  - Da bi se napravila robusna funkcija nije dovoljno proveriti samo tip ulazne vrednosti. Potrebno je, pre svega, proveriti da li ulazna vrednost uopšte postoji. To se može uraditi pomoću funktora `Maybe`.
- Funktor `Eather`
  - Ako funkcija mapirana na funktor generiše grešku, funktor bi trebao da bude u stanja da sa tom greškom nekako rukuje. To se može uraditi pomoću funktora `Eather`.

# Specifična `map()` implementacija => funktor **Maybe**

- Primer: Funkcija pristupa svojstvu

```
const street = user.address.street;
```

- Da ne bi bilo neprijatnih iznenađenja u situacijama kada vrednosti za `user` ili `user.address` nisu postavljene, treba dodati uslov za proveru:

```
const street = user && user.address &&  
user.address.street;
```

- Za to se može iskoristiti obrazac functor.
- Ideja je da se kreira kontejner koji će **u nekim situacijama skladištiti vrednost**, a **u nekim će biti prazan**.

# Funktor Maybe je jednostavan

```
const isNothing = value => value === null  
|| typeof value === "undefined";
```

```
const Maybe = value => ({  
  map: fn => (isNothing(value) ?  
    Maybe(null) : Maybe(fn(value))),  
  value,  
});
```



# map funkcija funktora Maybe

```
Maybe.prototype.isNothing = function() {  
    return (this.value === null ||  
            this.value === undefined);  
};  
Maybe.prototype.map = function(fn) {  
    return this.isNothing() ?  
        Maybe.of(null) :  
        Maybe.of(fn(this.value));  
};
```

# Funktor Maybe – kako se koristi

```
const isNothing = value => value === null || typeof value === "undefined";
const Maybe = value => ({
  map: fn => (isNothing(value) ? Maybe(null) : Maybe(fn(value))),
  value,
});

const user = {
  name: "Holmes",
  address: { street: "Baker Street",
    number: "221B"
  },
};

Maybe(user)
  .map(u => u.address)
  .map(a => a.street).value; // 'Baker Street'

const homelessUser = { name: "The Tramp" };
Maybe(homelessUser)
  .map(u => u.address)
  .map(a => a.street).value; // null
```

# Maybe malo više funkcionalno

- Pozivanje svojstva `value` na kraju nije baš nešto što funkcionalni stil preferira; malo “funkcionalniji” funktor `Maybe` bi izgledao ovako:

```
const isNothing = value => value === null || typeof  
value === 'undefined';
```

```
const Maybe = value => ({  
  map: fn => isNothing(value) ? Maybe(null) :  
    Maybe(fn(value)),  
  getOrElse : defaultValue => isNothing(value) ?  
    defaultValue : value,  
});
```



# Maybe korišćenje malo više funkcionalno<sub>1/2</sub>

- A i svojstvima objekta se može lepše pristupiti pomoću getera:

```
const get = key => value => value[key];
```

```
const getStreet = user =>
```

```
  Maybe(user)
```

```
    .map(get("address"))
```

```
    .map(get("street"))
```

```
    .getOrElse("unknown address");
```

# Maybe korišćenje malo više funkcionalno<sub>2</sub>

```
getStreet({  
  name: "Holmes",  
  firstname: "Sherlock",  
  address: {  
    street: "Baker Street",  
    number: "221B",  
  },  
}); // 'Baker Street'
```

```
getStreet(); // 'unknown address'
```

```
getStreet({  
  name: "Moriarty",  
}); // 'unknown address'
```

```
getStreet(); // 'unknown address'
```

# Rukovanje greškama/izuzecima

- Rukovanje greškama je programska tehnika za implementiranje postupaka koji se sprovode u slučaju grešaka u nekoj aplikaciji.
  - Ima situacija koje se unapred mogu predvideti kada postoji mogućnost da se program neće ponašati na željeni način.
  - U takvim slučajevima, ukoliko je to ikako moguće, treba definisati očekivano ponašanje programa.
- Način rukovanja greškama/izuzecima u funkcionalnom programiranju je drugačiji od načina koji se primenjuju u imperativnom programiranju i tu se koriste funktori.
- Dva specifična funktora koji se vrlo često koriste baš za rukovanje greškama su `Maybe` i `Either`.

# Funktor $\text{Either}_1$

- Ako funkcija mapirana na funktor generiše grešku, funktor bi trebao da bude u stanja da sa tom greškom nekako rukuje.
- Opet ćemo da počnemo sa primerom: Treba proveriti ispravnost email adrese ako je zadata, a u protivnom vrednost ignorisati i vratiti `null`.

# Funktor Eather: provera email-a

- Da li bi moglo ovako:

```
const validateEmail = value => {  
  if (!value.match(/\S+@\S+\.\S+/)) {  
    throw new Error(`Dati email je nevalidan`);  
  }  
  return value;  
};
```

```
const validEmail =  
validateEmail("foo@example.com");  
console.log(`Validan email: `, validEmail)  
// Validan email: foo@example.com
```

```
validateEmail("foo@example"); /* generiše grešku sa  
porukom `Dati email je nevalidan` */
```



# Moglo bi, ali ima nedostataka

- Ovakav način nije baš najbolji jer prekida tok izvršavanja (generiše prekid).
- Bolja bi bila funkcija `validateEmail` koja ne generiše prekid nego umesto prekida vraća nešto sa čim program dalje može da “živi” – recimo, poruku o grešci.
- Može li to sa funktorom i, ako može, kako može?

# Rukovanje greškama: Ideja

- Ideja je sledeća: Vratiti *različite* funktore na bazi try/catch:
  - Ili funktor koji će dozvoliti da se nastavi map (obično se zove Right),
  - Ili funktor koji vrednost neće uopšte manjati (ako je greška) (obično se zove Left)
- Funkcija `validateEmail` bi tada mogla da ima sledeći izgled:

# Nova funkcija **validateEmail**

```
const validateEmail = value => {  
  if (!value.match(/\S+@\S+\.\S+/)) {  
    return Left(new Error("Uneti email  
nije validan"));  
  }  
  return Right(value);  
};
```



# Kako izgledaju funktori

```
const Left = value => ({  
  map: fn => Left(value),  
  value,  
});
```

```
const Right = value => ({  
  map: fn => Right(fn(value)),  
  value,  
});
```

# Provera e-mail-a:

```
validateEmail("foo@example") /* nevalidan  
email, vrati Left functor */
```

```
    .map(v => "Email: " + v).value; /*  
Error('Uneti email nije validan') */
```

```
validateEmail("foo@example.com") /* validan  
email, vrati Right functor */
```

```
    .map(v => "Ispravan Email: " + v).value; /*  
' ispravan Email: foo@example.com' */
```



# Hvatanje grešaka

- Funkcija `validateEmail` vraća ili rezultat ili grešku, ali takva kakva je ne omogućuje da se uradi nešto specifično u slučaju greške.
- Dodajemo zbog toga novu funkciju `catch()` i funktoru `Left` i funktoru `Right`
- Kako treba da se ponaša funkcija `catch()` u funktoru `Left` a kako u funktoru `Right`?

# Funkcija `catch()` u funktoru `Right`

// U funktoru `Right`, `catch()` ne radi ništa

```
const Right = value => ({  
  map: fn => Right(fn(value)),  
  catch: () => Right(value),  
  value,  
});
```

// `catch` se ignoriše u `Right`

```
Right(5).catch(error =>  
  error.message).value; // 5
```

# Funkcija `catch()` u funktoru `Left`

/\* U funktoru `Left`, `catch()` treba da primeni funkciju i da vrati `Right` da se obezbedi dalje mapiranje \*/

```
const Left = value => ({  
  map: fn => Left(value),  
  catch: fn => Right(fn(value)),  
  value,  
});
```

```
Left(new Error('Raspadam se! ')).catch(error  
=> error.message).value; // 'Raspadam se! '
```



# Na kraju: **tryCatch** je kombinacija

```
const tryCatch = fn => value => {  
  try {  
    return Right(fn(value)); /* sve je  
    dobro krenulo, idemo desno */  
  } catch (error) {  
    return Left(error); /* oops! desila  
    se greška, idemo levo. */  
  }  
};
```

# Provera mail-a sa tryCatch

```
const validateMail = tryCatch(value => {  
  if (!value.match(/\S+@\S+\.\S+/)) {  
    throw new Error(`Mail je nevalidan`);  
  }  
  return value;  
});
```

```
validateMail("foo@example")  
  .map(v => "Email: " + v)  
  .catch(get("message")).value; // Mail je nevalidan  
validateMail("foo@example.com")  
  .map(v => "Email: " + v)  
  .catch(get("message")).value; // Email: foo@example.com
```

# Kompozicija funktora

- Zadatak: Validirati email u objektu koji MOŽDA IMA a MOŽDA i NEMA email vrednost.
- Dakle, prvo treba videti IMA li mail-a
- Zatim, ako ga IMA, validirati njegovu ispravnost (u odnosu na zadatu sintaksu)
- Funktori koje smo do sada napravili umeju sledeće:
  - Da vrate punu ili praznu kutiju-kontekst (funktor **Maybe**)
  - Da uhvate/otkriju grešku i vrate poruku (funktor **Either**, t.j. funkcija **tryCatch**)
- Pa da ih kombinujemo





# Da vidimo kako to izgleda: funkcija **validateUser**

```
const validateUser = user =>  
  Maybe(user)  
    .map(get(`email`))  
    .map(v =>  
      validateMail(v).catch(get(`message`)));
```

# Funkcija `validateUser`: korišćenje

```
validateUser({ // ispravan email  
  firstName: "John",  
  email: "foo@example.com",  
}); // Maybe(Right('foo@example.com'))
```

```
validateUser({ // neispravan email  
  firstName: "John",  
  email: "foo@example",  
}); // Maybe(Left('The given email is invalid'))
```

```
validateUser({ // nema email-a  
  firstName: "John",  
}); // Maybe(null)
```



# Osvetoljubiv kod

```
const validateUserValue = user => {  
  const result = validateUser(user).value;  
  if (value === null || typeof value ===  
    `undefined`) {  
    return null;  
  }  
  return result.value;  
};
```



# Kako se spasti: `chain()`

- Nama za konačnu transformaciju treba vrednost koja je umotana u `Maybe()`, ili neki drugi omotač – funktor.
- Dakle, treba nam alatka kojom ćemo moći da ekstrahujemo vrednost funktora i u situacijama kada je ona spakovana u drugi funktor.
- To se može uraditi jedino ako se ono što nam smeta ukloni – ako se ukloni “unutrašnji” funktor
- To je metoda `flatten()`

# Kako to izgleda

```
const Maybe = value => ({  
  /* mogla bi se vratiti vrednost, ali  
    bismo tada trebali i da se bavimo  
    različitim tipovima funktora. Na ovaj  
    način Maybe.flatten će uvek da vrati  
    tip Maybe */  
  flatten: () => isNothing(value) ?  
    Maybe(null) : Maybe(value.value),  
  ... ,  
});
```

# Na dalje: samo jednostavna Maybe

```
const validateUser = user => Maybe(user)
    .map(get('email'))
    .map(v => validateMail(v))
    .catch(get('message'))
    )
    .flatten()

/* od sada uvek imamo samo jednostavan
Maybe, pa se može iskoristiti getOrElse
da se dobije vrednost */
    .getOrElse('Korisnik nema mail');
```



# Primeri korišćenja

```
validateUser({  
  firstName: 'John',  
  email: 'foo@example.com',  
}); // 'foo@example.com'
```

```
validateUser({  
  firstName: 'John',  
  email: 'foo@example',  
}); // ' email je neispravan'
```

```
validateUser({  
  firstName: 'John',  
}); // ' korisnik nema mail'
```

# Evo ga: `chain()`

```
const Maybe = value => ({  
  flatten: () => isNothing(value) ?  
    Maybe(null) : Maybe(value.value),  
  // ovde se koristi imenovana funkcija a ne  
  // streličasta jer treba pokazivač this */  
  chain(fn) {  
    return this.map(fn).flatten();  
  },  
  ...,  
});
```



# A evo i novi **validateUser**

```
const validateUser = user =>
  Maybe(user)
    .map(get('email'))
    .chain(v => validateMail(v)
      .catch(get('message')))
    )
    .getOrElse('Korisnik nema mail');
```

# Koristimo ga na isti način

```
validateUser({  
  firstName: 'John',  
  email: 'foo@example.com',  
}); // 'foo@example.com'
```

```
validateUser({  
  firstName: 'John',  
  email: 'foo@example',  
}); // ' email je neispravan'
```

```
validateUser({  
  firstName: 'John',  
}); // ' korisnik nema mail'
```

# A to je: monada

- Funktor koji može sam sebe da poravna pomoću metode `chain()` zove se **Monada**.
- Dakle, Maybe je monada.
- Kao i `map()`, i `chain()` mora da poštuje određene zakone da bi se monade korektno komponovale:
  - Levi identitet
  - Desni identitet
  - Asocijativnost

# Levi identitet

```
monad(x).chain(f) === f(x);
```

// f je funkcija koja vraća monadu

- Ulančavanje funkcije u monadu je isto što i prosleđivanje vrednosti funkciji.
- Na taj način se osigurava potpuno uklanjanje obuhvatajuće monade pomoću metode `chain`.

# Desni identitet

`Monad(x).chain(Monad) === Monad(x);`

- Ulančavanje konstruktora monada treba da vrati istu monadu.
- Ovim se osigurava da ulančavanje ne proizvede bočni efekat

# Asocijativnost

```
monad.chain(f).chain(g) == monad.chain(x =>  
  f(x).chain(g));
```

// f i g su funkcije koje vraćaju monadu

- `chain()` mora biti asocijativna operacija, odnosno mora da važi:  
`.chain(f).chain(g)` kao i `.chain(v => f(v).chain(g))`
- Time se osigurava da se mogu ulančavati funkcije koje i same koriste `chain()`.

# Monada je važna: da još jednom ponovimo

- **Monada** je način kompozicije funkcija gde se, pored vrednosti, zahteva i nekakav kontekst (poput računanja, grananja ili efekata). Monada vrši mapiranje (`map`) i poravnavanje (`flatten`)/"raspakivanje".
- Šta to, u stvari, znači? Znači sledeće: Mapiranje jednostavne vrednosti **a** na **b** u situaciji gde je **a** dato u nekom kontekstu **M(a)** na način da se rezultat **b** takođe dobija u kontekstu, odnosno dobija se **M(b)**.
- Vrlo dobro objašnjenje koncepta monada imate u knjizi Erika Eliota *Composing Software*

# Sušтина monade

- Funkcije se mogu komponovati:

$a \Rightarrow b$  i  $b \Rightarrow c$  može da napravi  $a \Rightarrow c$

- Funktori mogu da komponuju funkcije sa kontekstom:

Ako je zadato  $F(a)$  i dve funkcije,  $a \Rightarrow b$  i  $b \Rightarrow c$ , ume da vrati  $F(c)$ .

- Monade mogu da komponuju funkcije type lifting-a:

$a \Rightarrow M(b)$ ,  $b \Rightarrow M(c)$  ume da vrati  $a \Rightarrow M(c)$





# Komponovanje: funkcije

- $g: a \Rightarrow b$
- $f: b \Rightarrow c$
- $h: a \Rightarrow c$



# Komponovanje: jednostavne funkcije

$$a \xrightarrow{g} b$$

$$b \xrightarrow{f} c$$

$$a \xrightarrow{\text{compose}(f, g)} c$$

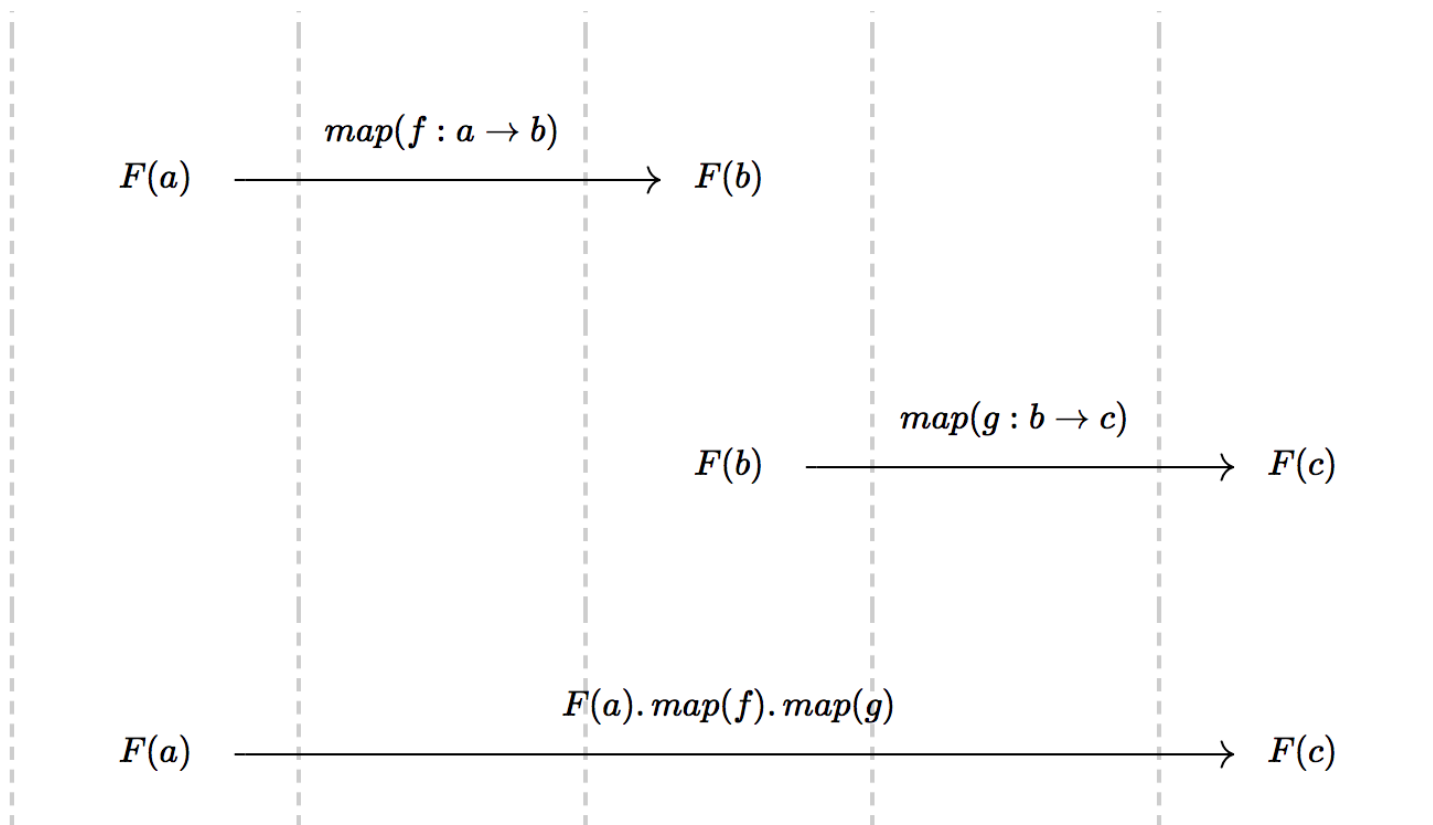


# Komponovanje: funktori

- $f: F(a) \Rightarrow F(b)$
- $g: F(b) \Rightarrow F(c)$
- $h: F(a) \Rightarrow F(c)$



# Komponovanje: funktori



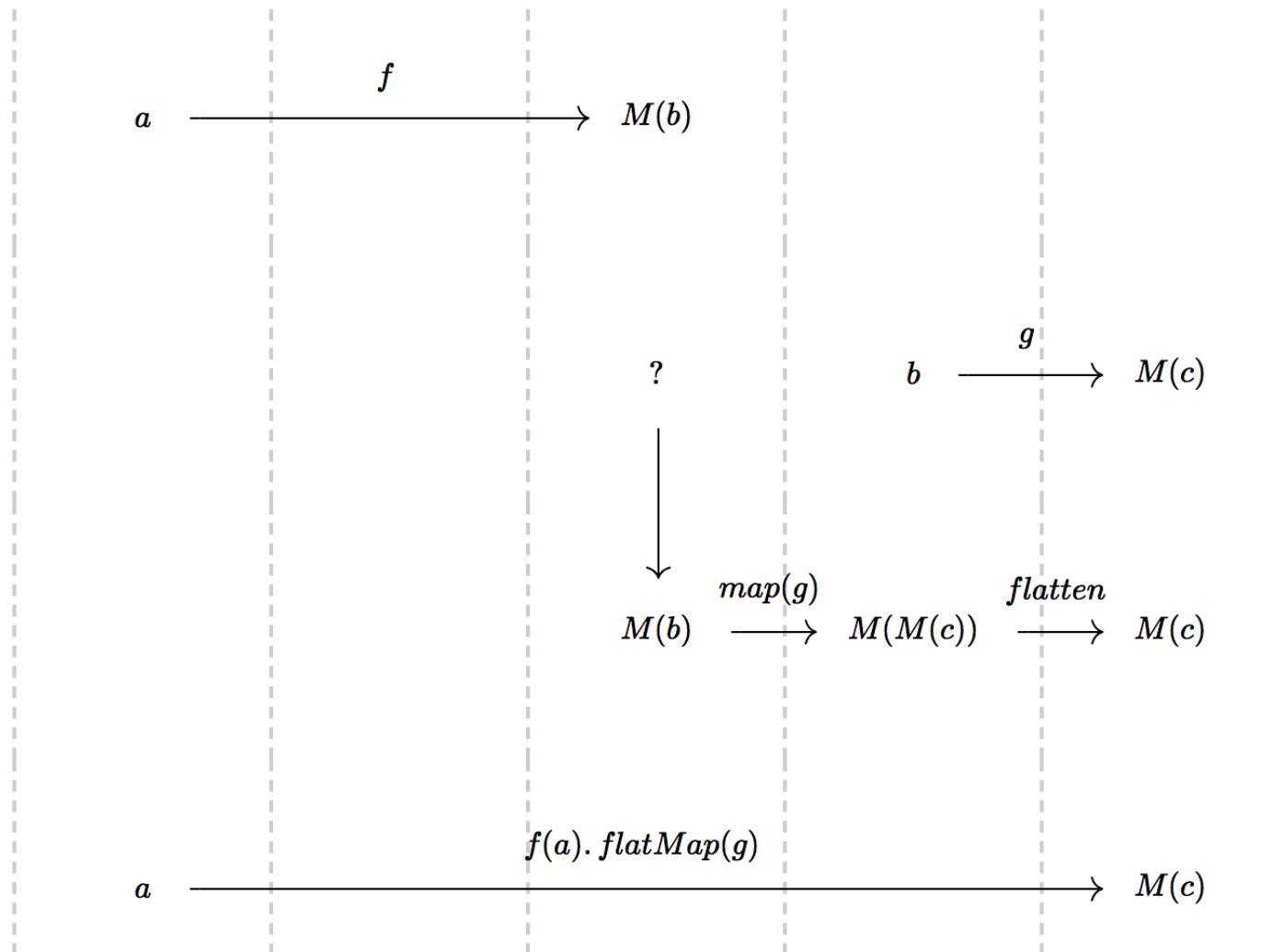


# Komponovanje: monade

- $f: a \Rightarrow M(b)$
- $g: \quad \quad \quad ??? \quad b \Rightarrow M(c)$
- $h: a \quad \quad \quad \Rightarrow \quad \quad M(c)$



# Komponovanje: monade



# Aplikativ (Aplikativni funktor)

# Šta je aplikativni funktor<sub>1/2</sub>

- U funkcionalnom programiranju, aplikativni funktor, ili skraćeno aplikativ, je struktura koja je između funktora i monada.
- Aplikativni funktori dozvoljavaju sekvencioniranje funktorskih izračunavanja (za razliku od običnih funktora), ali ne dozvoljavaju korišćenje rezultata prethodnih izračunavanja u definiciji narednih (za razliku od monada).



# Šta je aplikativni funktor<sub>2/2</sub>

- U Haskell-u, aplikativ je parametrizovani tip (tip koji je zadat parametrima i instancira se po potrebi) koji se sastoji od kontejnera za podatke tipa parametra i dve metode **pure** i **<\*>**.
- Metoda **pure** može se tumačiti kao unošenje vrednosti u aplikativ. Za aplikativ parametarizovanog tipa  $f$  ima tip:  
$$\text{pure} :: a \rightarrow f a$$
- Metoda **<\*>** može se tumačiti kao ekvivalent aplikaciji funkcije unutar aplikativa. Za aplikativ parametarizovanog tipa  $f$  ima tip:  
$$(<*>) :: f (a \rightarrow b) \rightarrow f a \rightarrow f b$$

# Zakoni aplikativnog funktora

- Identitet:  
 $\text{pure id } \langle * \rangle v = v$
- Kompozicija:  
 $\text{pure } (.) \langle * \rangle u \langle * \rangle v \langle * \rangle w = u \langle * \rangle (v \langle * \rangle w)$
- Homomorfizam:  
 $\text{pure } f \langle * \rangle \text{pure } x = \text{pure } (f x)$
- Zamenljivost:  
 $u \langle * \rangle \text{pure } y = \text{pure } (\$ y) \langle * \rangle u$

# Aplikativni funktor vs. funktor

- Svaki aplikativ je funktor.
- To znači da se ako su zadate metode **pure** i **<\*>**, funktor **fmap** može implementirati kao

$$\text{fmap } g \ x = \text{pure } g \ \langle * \rangle \ x$$

# Aplikativ - primer

- In Haskell-u, tip `Maybe` može postati instanca klase tipa `Applicative` korišćenjem definicije:

```
instance Applicative Maybe where
```

```
  -- pure :: a -> Maybe a
```

```
  pure a = Just a
```

```
  -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
```

```
  Nothing <*> _ = Nothing
```

```
  _ <*> Nothing = Nothing
```

```
  (Just g) <*> (Just x) = Just (g x)
```

- Definicija kaže: **pure** pretvara **a** u **Maybe a**, a **<\*>** primenjuje funkciju **Maybe** na vrednost **Maybe**.

# Aplikativ Maybe – šta omogućuje

- Korišćenje aplikativa **Maybe** za tip **a** omogućava da se radi na vrednostima tipa **a** tako da se greške automatski obrađuju mašinerijom aplikativa.
- Na primer, u Haskell-u je za sabiranje dva broja **m :: Maybe Int** i **n :: Maybe Int**, dovoljno napisati:  
$$(+) \langle \$ \rangle m \langle * \rangle n$$
- U slučaju kada nema greške, sabiranje **m=Just i** i **n=Just j** daje rezultat **Just(i+j)**. Ako je **m** ili **n** **Nothing**, i rezultat je **Nothing**.

Da uopštimo

# Matematika iza funktora i monada

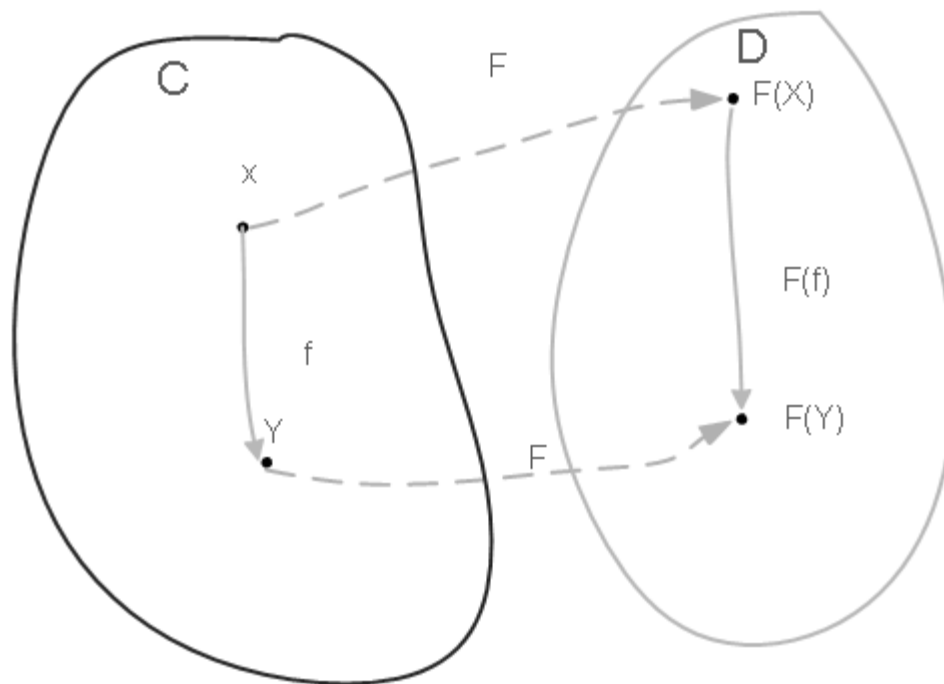
- **Teorija kategorija** formalizuje matematičku strukturu i njene koncepte putem označenog usmerenog grafa koji se zove *kategorija*, čiji se čvorovi zovu *objekti*, a označene usmerene veze *strelice* ili *morfizmi*.
- Objekti mogu da budu bilo šta ali je generalno korisno o njima razmišljati kao o skupovima nekih elemenata (kao što su, na primer, tipovi u programskim jezicima).
- Morfizmi/strelice su funkcije koje vrše mapiranja među objektima.
- Kategorija ima dva osnovna svojstva:
  - Asocijativno komponovanje morfizama, i
  - Postojanje identitetskog morfizma za svaki objekat.

# Teorija kategorija i funktor

- U teoriji kategorije **funktor** je struktura koja očuvava mapiranje sa kategorije na kategoriju, a gde
- “očuvanje strukture” znači da se očuvavaju relacije među objektima i morfizmima.
- Sve funktor map operacije moraju da poštuju dva aksioma koji se zovu “zakoni funktora”:
  - **Identitet:** Za svaki objekat  $A$  u kategoriji  $C$ , mora da postoji identitetski morfizam koji mapira na isti objekat (t.j. mapira objekat  $A$  na smog sebe). Taj morfizam se označava sa  $id_A$  ili  $1_A$ .
  - **Kompozicija:** Za sve parove morfizama  $g : A \rightarrow B$ ; i  $f : B \rightarrow C$ , postoji morfizam  $h : A \rightarrow C$ ,  $h = f \circ g$ .



# Slika koja sve kaže



$F$  mapira objekte  $X, Y$  iz kategorije  $C$  na objekte  $F(X)$  i  $F(Y)$  u kategoriji  $D$  i morfizam  $f$  iz kategorije  $C$  na  $F(f)$  u kategoriji  $D$

# Jedan mali primer

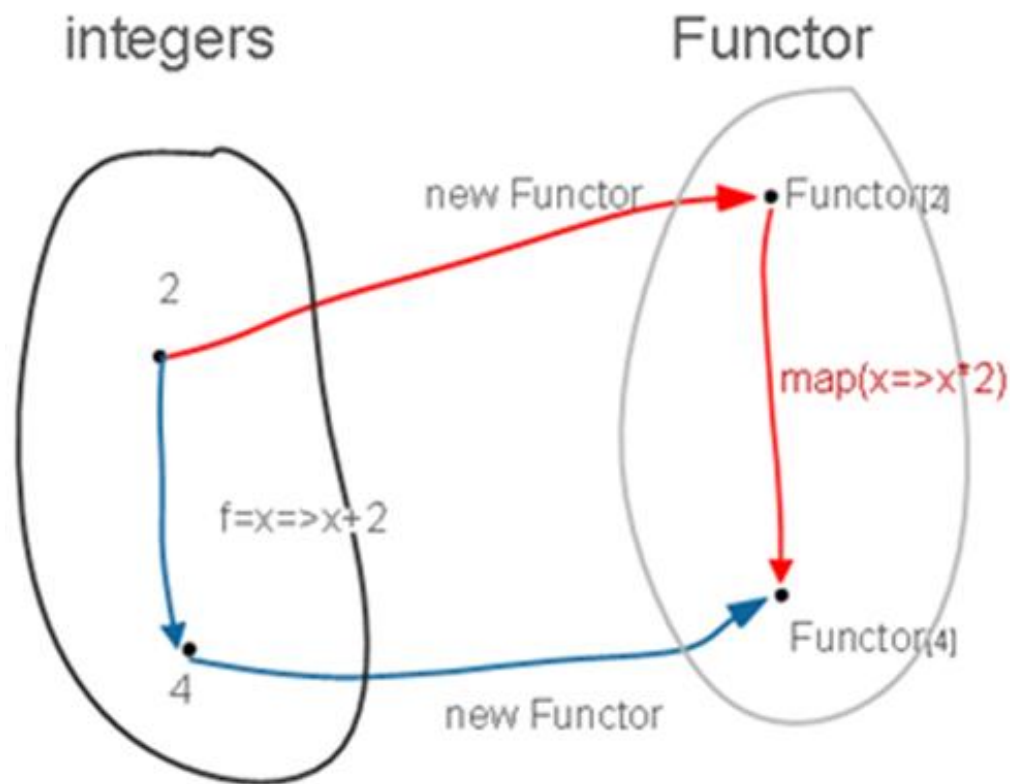
```
//Izvor: The definite guide to Functors in Js  
//https://medium.com/@dimpapadim3/the-definite-guide-  
to-functors-in-js-6f5e82bd1dac
```

```
const Functor = (v) => ({  
  value: v,  
  map: (mapping) => Functor(mapping(v))  
});
```

```
var s = Functor(2)  
  .map(x=>x*x)  
  .map(x=>x.toString());
```

```
console.log(s.value);// ispisaće: 4
```

# Još jedna slika za bolje razumevanje prethodnog koda - Funktora



# Šta je ovde korektno mapiranje

- Lako je videti da je korektno mapiranje  
`this.map = (mapping) => Functor(mapping(value));`  
zato što:  
`var f = x => x * x;`  
`var paht1 = Functor(2).map(f);`  
`var path2 = Functor(f(2));`  
`paht1.value == path2.value; // Vraća true`
- Ovde je zadovoljen još jedan bitan uslov za očuvanje strukture: komutativnost putanja - svejedno je da li se prvo izvrši Functor nad 2, pa se zatim primeni map funkcija ili obrnuto.

# Identitet

- Formalna definicija:

$$F(a) \xrightarrow{\text{map}(\text{id}: a \rightarrow a)} F(a) = F(a)$$

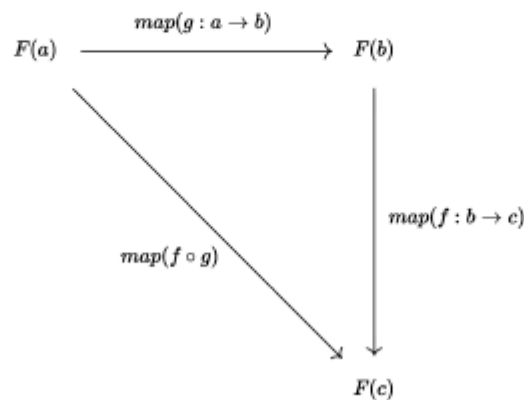
- Ako se funkcija identiteta ( $x \Rightarrow x$ ) prosledi u `a.map()`, gde je `a` funktor proizvoljnog tipa, rezultat treba da bude ekvivalentan sa `a`:

```
const a = [20];  
const b = a.map(a => a);
```

```
console.log(  
  a.toString() === b.toString() // Izlaz: true  
);
```

# Kompozicija

$$F(a) \xrightarrow{\text{map}(g)} F(b) \xrightarrow{\text{map}(f)} F(c) = F(a) \xrightarrow{\text{map}(f \circ g)} F(c)$$



- Funktori moraju da zadovoljavaju zakon kompozicije :  
 $a.\text{map}(g).\text{map}(f)$  je ekvivalentno sa  $a.\text{map}(x \Rightarrow f(g(x)))$ .

# Kompozicija

- Kompozicija morfizama je asocijativna:  
 $(f \circ g) \circ h = f \circ g \circ h = f \circ (g \circ h)$
- Funktorsko mapiranje je oblik kompozicije funkcija.
- U sledećem kodu je `mappable.map(g).map(f)` ekvivalentno sa `mappable.map(x => f(g(x)))`

```
const g = n => n + 1;  
const f = n => n * 2;  
const mappable = [20];
```

```
const a = mappable.map(g).map(f);  
const b = mappable.map(x => f(g(x)));
```

```
console.log(  
  a.toString() === b.toString() // izlaz: true  
);
```



# Zakoni funktora

```
// Primer funktora  
const Identity = value => ({  
  map: fn => Identity(fn(value))  
});
```

- Funkcija **Identity** prima vrednost (**value**) i vraća **objekat** sa **.map()** metodom.
- Metoda **.map()** prima **funkciju** i vraća **rezultat primene funkcije na vrednost value** unutar **Identity**. Vraćena vrednost se umotava u kontekst **Identity**.



# Zakoni funktora: identitet

```
// trace() je pomoćna funkcija za laku inspekciju  
sadržaja.
```

```
const trace = x => {  
  console.log(x);  
  return x;  
};
```

```
const u = Identity(2);
```

```
// Zakon identiteta
```

```
const r1 = u; // Identity(2)
```

```
const r2 = u.map(x => x); // Identity(2)
```

```
r1.map(trace); // 2
```

```
r2.map(trace); // 2
```

# Zakoni funktora: asocijativnost kompozicije

```
// Zakon kompozicije (asocijativnost)
const r3 = u.map(x => f(g(x))); // Identity(5)
const r4 = u.map(g).map(f); // Identity(5)

const f = n => n + 1;
const g = n => n * 2;
r3.map(trace); // 5
r4.map(trace); // 5
```

# Sažetak<sub>1/4</sub>

- Funktor (**functor**) tip podatka je **nešto nad čim se može mapirati**.
- Funktor je u JS-u običan objekat (ili klasni tip u drugim jezicima) koji implementira funkciju map koja se izvršava nad svakom vrednošću u objektu da bi ***proizvela novi objekat***.
- Možemo ga tumačiti kao **kontejner** sa map operacijom koja omogućuje da se funkcija prosleđena map operaciji primeni na vrednosti u kontejneru.

# Sažetak<sub>2/4</sub>

- Prednosti funktora
  - Apstrahuju detalje implementacije osnovne strukture podataka – iteriranje se dešava “samo po sebi”.
  - Sakrivaju tipove podataka koje sadrže – funkcije mapiranja su generičke i ne treba da se vodi računa nad kojim tipom će se izvršavati.
  - Ne zahtevaju logiku za kontrolisanje prazne kolekcije niti za praćenje stanja iteracije – mapiranje nad praznom kolekcijom je isto kao nad nepraznom;
  - **NAJVAŽNIJE: Funktor omogućuje da se lako komponuju funkcije mapiranja.**

# Sažetak<sub>3/4</sub>

- **Monada**

- Monada vrši mapiranje (map) i poravnavanje (flatten)/"raspakivanje" što znači da omogućuje mapiranje jednostavne vrednosti **a** na **b** u situaciji u kojoj je **a** dato u nekom kontekstu **M(a)** a rezultat **b** se dobija takođe u kontekstu, odnosno dobija se **M(b)** .
- Na taj način omogućuje se kompozicija funkcija gde se, pored vrednosti, zahteva i komunikacija nekakvih dodatnih sadržaja (poput dodatnih računanja, uslovnih primena funkcije ili drugih boćih efekata poput IO operacija i slično).
- Monada je osnovni instrument za rukovanje boćnim efektima u FP-u.

# Sažetak<sub>4/4</sub>

- **Aplikativ**

- U funkcionalnom programiranju, aplikativni funktor, ili skraćeno aplikativ, je struktura koja je između funktora i monada.
- Može se razumeti kao funktor koji se primenjuje na funktor, odnosno kao kontejner sa map operacijom pri čemu su elementi u kontejneru funktori.
- Dozvoljava sekvencioniranje funktorskih izračunavanja (za razliku od običnih funktora), ali ne dozvoljava korišćenje rezultata prethodnih izračunavanja u definiciji narednih (za razliku od monada).

# Sažetak<sub>5/5</sub>

- Monade, aplikativni funktori i funktori utemeljeni su u najapstraktnijoj grani matematike koja se zove **Teorija kategorija**.
- Teorija kategorija formalizuje matematičku strukturu i njene koncepte putem označenog usmerenog grafa koji se zove kategorija, čiji se čvorovi zovu objekti, a označene usmerene veze strelice ili morfizmi.
  - Objekti mogu da budu bilo šta ali je generalno korisno o njima razmišljati kao o skupovima nekih elemenata (kao što su, na primer, tipovi u programskim jezicima).
  - Morfizmi/strelice su funkcije koje vrše mapiranje među objektima.
- Kategorija ima dva osnovna svojstva:
  - Asocijativno komponovanje morfizama, i
  - Postojanje identitetskog morfizma za svaki objekat.

# Literatura za predavanje

1. Z. Konjović, Funkcionalno programiranje, **Tema 07 – Funktori, aplikativni funktori i monade**, slajdovi sa predavanja, dostupni na folderu **FP kurs 2023-24 → Files → Slajdovi sa predavanja**
2. E. Elliot, **Composing Software - An Exploration of Functional Programming and Object Composition in JavaScript**, Leanpub, 2019, Poglavlje “**Functors and Categories**”