Biti ili ne biti funkcionalan

Ante Smolčić Ericsson Nikola Tesla d.d. 14.5.2019

O meni

— Arhitekt rješenja u Ericssonu Nikoli Tesli.

— Funkcionalnim programiranjem se bavim 2 godine.
— Uveo FP u nekoliko projekata u Javi 8.
— Nemam iskustva u pravim FP jezicima poput Haskella, Lispa ili Erlanga.

— Pokušat ću približiti realne probleme s kojima sam se susreo i neka rješenja koja su mi pomogla.

Mala povijest funkcionalnog programiranja

- Podloga u matematičkoj teoriji (lambda račun)
- LISP, prvi jezik s podrškom za FP, 1958. godina
- Jezik ML, 1973. godina
- Jezik Scheme, 1985. godina
- Mooreov zakon značajno slabi iza 2010. godine, zbog čega jača trend paralelne obrade podataka.
- Funkcionalno programiranje ima dobru podršku za paralelizam.
- FP polako ulazi i u popularne jezike.
- Google nije mogao čekati – biblioteka Guava ima podršku za FP već od 2011.
- Java 8 izlazi 2014. godine i uvodi podršku za FP.
- Scala, Clojure – FP jezici na JVM-u.

Zašto FP?

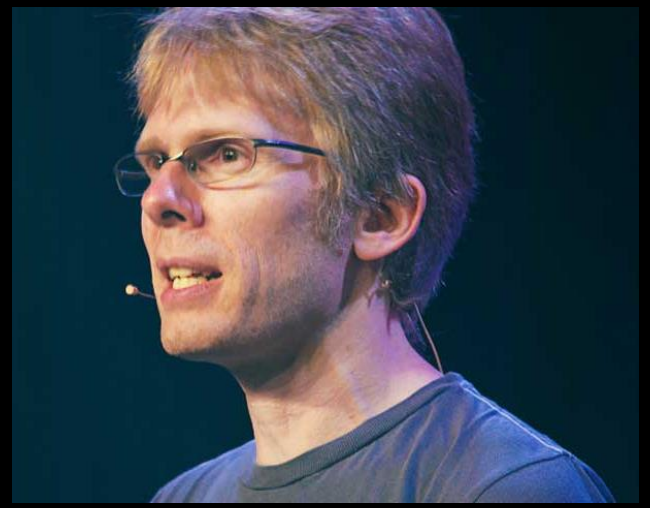
— Svaki kod je predivan... na početku.
— Zatim krenu izmjene, rokovi, komplikacije, 100 ljudi - 200 rješenja!
— Kod postaje neodrživ.

Glavni uzroci nastajanja lošeg koda:
— Promjena podataka s različitih mjesta u kodu.
— Presloženi algoritmi.
— Rješenja „na prvu loptu”.

FP otežava nastajanje svih ovih scenarija. Zašto onda ne koristiti FP?

No matter what language you work in, programming in a functional style provides benefits. You should do it whenever it is convenient, and you should think hard about the decision when it isn't convenient.

John Carmack
(Doom, Quake, Oculus Rift)



Ključne karakteristike FP

— Deklarativni stil.
— Naglasak na čistim funkcijama, bez nuspojava (side-effects).
— Naglasak na nepromjenjivim (immutable) objektima.
— Izbjegavanje NULL vrijednosti.
— Modeliranje transformacija podataka, umjesto procesa obrade podataka.

Deklarativni stil

— Sličan SQL-u:

SQL	Java API	
FROM customer c	customers.stream()	Izvor podataka
WHERE c.name LIKE 'A%'	filter(c -> c.getName().startsWith(„A“))	Filtar (uvjet)
SELECT c.name	map(c -> c.getName())	Ciljani podaci

— Fokus na na slijed operacija, a ne na kontrolne strukture (while/for/if) koje mijenjaju tijek izvođenja.
— Nema grananja pa je kod jednostavniji.
— Metoda mjerenja složenosti „McCabe Cyclomatic Complexity“ uvijek vraća 1.

Naglasak na čistim funkcijama (1)

- Nuspojave (side-effects) su promjene podataka van opsega funkcije.
- Samim tim nisu uočljive i jasne - kod je „zamršen".
- Nuspojave nisu nešto što možemo izbjeći jer svrha svih programa je da transformiraju ulazne podatke u izlazne. Korištenjem FP smanjujemo nuspojave na minimum.

- Primjer metode koja je nekad možda radila nešto drugo, a sada mijenja vrijednost u „x":

```
void doSomething(Entry<String, String> entry) {
        entry.setValue("x");
}
```

Naglasak na čistim funkcijama (2)

- Čiste funkcije
  - Kao i sve ostale funkcije, vraćaju jednu vrijednost.
  - Koriste samo lokalno stvorene vrijednosti i ulazne parametre funkcije.
  - Mijenjaju samo lokalno stvorene vrijednosti.
  - Za iste parametre vraćaju istu vrijednost - determinističke su.
  - Primjer čiste funkcije:

```
String formatDate(Date d, String format)
```

- Prethodni primjer, sada riješen preko čiste funkcije – vraća se novi objekt, a stari se ne mijenja:

```
Entry<String, String> getSomething(Entry<String, String> entry) {
        return new SimpleEntry<>(entry.getKey(), "x");
}
```

Naglasak na čistim funkcijama (3)

— Testiranje je jednostavnije jer argumenti funkcije ne bi trebali biti veliki „mock“ objekti.
— Moguće je koristiti priručnu memoriju (cache) kako bi se preskočilo ponovljeno izvođenje funkcije (tzv. „memoizacija“). U Javi taj korak moramo eksplicitno programirati.

Nepromjenjivi objekti (1)

- Nepromjenjivi objekti (immutable) postavljaju svoje vrijednosti samo jednom, prilikom stvaranja.
- Na taj način povećavaju robusnost koda jer eliminiraju potencijalne izmjene na drugim mjestima u kodu.
- Manje izmjena -> manje grešaka.
- Java nudi određene mehanizme zaštite objekata od promjene:
  - **java.lang.String** je sam po sebi nepromjenjiv.
  - Privatna polja
  - Finalna polja - moraju biti inicijalizirana u konstruktoru.
  - Read-only polja (izostavljanjem setter-metoda).
  - Za kolekcije koristimo „**Collections.unmodifiable***“.

Nepromjenjivi objekti (2)

```
class Immutable {
        private final String value;
        private final Collection<String> list;
        public Immutable(String value, Collection<String> list) {
                this.value = value;
                this.list = Collections.unmodifiableList(new ArrayList<>(list));
        }
        public String getValue() {
                return this.value;
        }
        public Collection<String> getList() {
                return this.list;
        }
}
```

— Da bi spriječili promjenu kolekcije izvana, moramo ju i kopirati i zabraniti slučajne promjene unutar objekta.

Nepromjenjivi objekti (3)

- Java 9 uvodi immutable-kolekcije: **List.of, Set.of. Map.ofEntries**.
- S obzirom na to da bi kolekcije i složene objekte trebali u potpunosti kopirati kako bi ostali nepromijenjeni, prednost se daje jednostavnim strukturama:
  - Jednostavne kolekcije (list, map, set)
  - POJO objekti,
  - Generički „tuple“ objekti.

- Java je vrlo „rječita“ (verbose), u odnosu prema drugim jezicima:
  - Scala: **val tuple = (100, "hello");**
  - Java: **Entry<Integer, String> tuple = new SimpleEntry<>(100, „hello“);**
  - Za tuple od 3 polja klasa AbstractMap.SimpleEntry neće poslužiti. ☹
  - Preporuča se koristiti vlastite klase: Pair, Tuple2, Tuple3 i slično.

Izbjegavanje NULL vrijednosti (1)

- Ako je element kolekcije NULL, možda ne bi ni trebao biti u kolekciji?
- Trend u programiranju - NULL se normalizira u manje osjetljiv tip podatka.
- NULL zahtijeva dodatnu obradu zbog potencijalnih grešaka.
- Povećava se složenost zbog:
  - Testiranja NULL vrijednost (IF grananje)
  - Dodatne logike u slučaju da vrijednost ipak jest NULL.
  - Inicijalizacije elemenata samo da ne budu NULL!

Izbjegavanje NULL vrijednosti (2)

- Java uvodi tip **java.util.Optional<T>** koji nudi nekoliko ključnih operacija:
  - Inicijalizacija iz potencijalne NULL vrijednosti: **Optional.ofNullable**.
  - Konverzija NULL u „praznu“ vrijednost: **Optional.empty**.
  - Transformacija vrijednosti: **Optional.map/flatMap**.
  - Transformacija prazne vrijednosti u zamjensku, odnosno u grešku: **Optional.orElseGet/orElseThrow**.
- Neke dijelovi funkcionalnog API-ja izazivaju greške u radu s NULL vrijednost:
  - **Collectors.toMap** – odbija NULL i za *key* i za *value*.
  - **Optional.of** – nužno je koristiti **ofNullable**.

Korištenje Optional tipa olakšava uočavanje takvih slučajeva u kojima vrijednost ne postoji ili nije definirana.

Transformacija podataka, umjesto procesa obrade (1)

- Klasično programiranje prolazi kroz dvije faze: modeliranja podataka i modeliranje procesa obrade.
- U FP-u se ne pristupa direktno podacima pa je fokus na modeliranju pomoćnih i privremenih struktura koje spremaju stanje između pojedinih koraka obrade.
- Treba paziti da pomoćne strukture ne narastu na glomazne „master"-objekte.

Transformacija podataka, umjesto procesa obrade (2)

— Primjer klasičnog programa koji se zapravo sastoji od dva procesa – provjera limita i promjena stanja računa:

```
void withdrawAmount(Account account, int amount) {
        int newBalance = account.getBalance() - amount;
        boolean allowed = checkLimit(newBalance, account.getLimit());
        if (allowed) account.setBalance(newBalance);
        else System.out.println("Not sufficent funds!");
}
```

— Proces promjene stanja ovisi o rezultatu procesa provjere limita i ta međuovisnost stvara određenu složenost.

Možemo li razbiti tu međuovisnost? Možemo li reći da promjena računa ide u svakom slučaju, ma što bilo s limitom?

Transformacija podataka, umjesto procesa obrade (3)

— Novo rješenje bez IF grananja, a ako je limit prekoračen, novo stanje:

```
void withdrawAmount(Account account, int amount) {
        BalanceStatus balanceStatus = newBalanceStatus(
                account.getBalance(),
                account.getLimit(),
                amount);
        checkError(balanceStatus.getError());
        account.setBalance(balanceStatus.getBalance());
}
```

— Da bi se ovo postiglo bilo je potrebno modelirati transformaciju strukture:
(**balance, limit, amount**) -> (**balance, error**)

— Proces izračunavanja smo preselili u metodu čija nas implementacija nas zapravo ne zanima u trenutku dizajniranja ovog rješenja.

Transformacija podataka, umjesto procesa obrade (4)

Postavlja se pitanje kako obraditi eventualnu grešku.

— Ovdje se radi o obradi transakcijskog naloga, pa neka kod koji generira transakciju također i obradi grešku!

— Sljedeće rješenje vraća grešku van, a i stvara novi objekt tipa Account jer ne želimo imati nuspojave.

```
OperationStatus withdrawAmount(Account account, int amount) {
        BalanceStatus balanceStatus = newBalanceStatus(
                account.getBalance(),
                account.getLimit(),
                amount);
        Account a2 = new Account(acount.getLimit(), balanceStatus.getBalance());
        return new OperationStatus(a2, balanceStatus.getError());
}
```

Transformacija podataka, umjesto procesa obrade (5)

Cijela obrada transakcije se može opisati ovako

transactions: (***account, amount, operationType***)
withdrawAmount: (***account, error?***)
skipOnErorr: (**true/false**)
applyChange (**account**)

— Različiti koraci koriste različite strukture podataka.

Temeljni pojmovi (1)

- Immutable objekt – nepromjenjivi objekt, koriste se za prijenos informacija između koraka obrade.
- Tuple – Generički immutable objekt koji sadrži mali broj polja 2, 3, 4...
  - **Tuple2**<T1, T2>**, Entry**<K, V>**, Pair**<L, R>
  - **Tuple3**<T1, T2, T3>
  - ...
- Statički import – Za lakše korištenje statičkih metoda, kojih ima puno u funkcionalnom API-ju, možemo si olakšati život korištenjem statičnog importa.
  - **import static** java.util.functinal.Collectors.toMap;

Temeljni pojmovi (2)

Lambda – jednostavnija sintaksa za anonimne klase

- Anonimna klasa:

```
Comparator<String> myCompare = new Comparator<String>() {
        @Override
        public int compare(String o1, String o2) {
                return o1.compareTo(o2);
        }
};
```

- Identična lambda-sintaksa:

```
Comparator<String> myCompareLambda = (o1, o2) -> o1.compareTo(o2);
```

- Pozivanje je identično:

```
myCompare.compare("a", "b");
myCompareLambda.compare("a", "b");
```

Temeljni pojmovi (3)

- Lambde se masovno se koriste u funkcionalnom API-ju, ali preporučam koristiti ih svugdje gdje možete. ☺
- Možemo raditi delegiranje funkcija, ovisno o nekom uvjetu:

```
Comparator<String> selectedComp = descending ? descComp : ascComp;
selectedComp.compare("a", "b");
```

- Ili koristiti kao mini-factory:

```
Map<Direction, Comparator<String>> comparatorMap;
comparatorMap.get(Direction.DESCENDING).compare("a", "b");
```

Temeljni pojmovi (4)

Možemo li još skratiti sintaksu?

- Method-reference
  - Još jednostavnija sintaksa za lambde
  - Ne moramo implementirati interface ako implementacijska metoda ima isti potpis kao i metoda funkcionalnog interfacea.

```
private int myCompare(String o1, String o2) {
        return o1.compareTo(o2);
}

Comparator<String> myComp = this::myCompare;
myComp.compare("a", "b");

(this::myCompare).compare("a", "b"); // ovo ne radi...

((Comparator<String>) this::myCompare).compare("a", "b"); // ...ali ovo radi!
```

Temeljni pojmovi (5)

- Closure – lambda funkcija koja dohvaća i vanjske varijable
  - U primjeru, „**descending**" je vanjska varijabla, a ograničenje je da mora biti finalna (ili efektivno finalna).

```
boolean descending = true;
Comparator<String> myComp = (o1, o2) -> {
        int result = o1.compareTo(o2);
        return descending ? -result : result;
};
```

Osnove Java API-a

- Paketi:
  - **java.util.functional** i **java.util.stream**
- Kolekcije implementiraju metodu **stream** koja vraća objekt tipa **Stream** koji jedan po jedan element kolekcije koristi kao parametar sljedeće metode
  - **[1, 2, 3].stream()** -> **doSomething**(**int** element) -> **doSomethingElse**(**T** element)
- Postoji i **parallelStream**
  - Nisam koristio!
  - Neke metode tipa **Stream** neće raditi očekivano, poput **sorted** i **forEach**, ali postoje alternativne metode.
- Za primitivne tipove int, long i double postoje posebne implementacije: **IntStream, LongStream** i **DoubleStream**.
  - Posebna metoda **boxed** - za pretvaranje u regularni **Stream**.

API: Stream (1)

- Stream tip ima dvije vrste metoda
  - *intermediate* - ne proizvode konačni rezultat, već vrše neku operaciju s ulaznim elementom
  - *terminal* – proizvode konačni rezultat obrade ulaznih elemenata

- Naizgled, intermediate metode ne mogu imati stanje (*state*) jer koriste lambda funkcije. No lambde su zapravo klase pa ipak možemo imati stanje.
- Dokaz su metode **sorted** i **distinct** koje elemente prikupljaju u internu strukturu.
- Ako radimo vlastitu „statefull" lambdu moramo paziti na paralelno izvršavanje pa interne strukture moraju biti *thread-safe*.

API: Stream (2)

- Česta greška je promjena podataka izvan lambda funkcija.
- Intermediate metode se izvršavaju odgođeno (lazy evaluation) tj. tek kada se pozove terminalna metoda.
- U donjem primjeru se kolekcija temp neće popuniti dok se ne pozove **collect** na kraju.

```
List<String> temp = new ArrayList<>();

Stream<String> mainItems = source.stream()
        .map(e -> temp.add(e));

System.out.print("First item: "  + temp.get(0)); // IndexOutOfBoundsException!

// ...
Stream.concat(mainItems, Stream.of(„extra item")).collect(toList());
```

API: forEach()

- **forEach** je terminalna metoda koja je definirana na klasama tipa Collection i Stream.
- Restriktivnija, sigurnije, implementacija **for-each** petlje jer koristi lambdu.
  - **items.forEach(item -> {...})**
- Pogodna za finalnu obradu svakog elementa.
- Nije namijenjena ažuriranju vanjskih varijabli, iako je moguće.

```
Stream<User> users;
users.forEach(user ->
        user.setFullName(user.getFirstName() + " " + user.getLastName())
);
```

API: collect()

- Skuplja vrijednosti u konačnu kolekciju – rezultat obrade.
- Za uvrštavanje u elementa u konačnu kolekciju se koriste objekti tipa **Collector**. Oni znaju kako:
  - Stvoriti konačnu kolekciju.
  - Rasporediti ulazne elemente u konačnu kolekciju.
- Standardni kolektori se nalaze u klasi **java.util.stream.Collectors**.
- Najčešće se koriste: **toList** i **toMap**.
- Primjer:

```
List<String> result = source.collect(Collectors.toList());
```

API: Collectors.toMap()

```
Map<String, String> userAddress = users.collect(
        toMap(
                user -> user.getName(),
                user -> user.getAddress()));
```

— Ako želimo da kolektor stvori neku drugu implementaciju mape, možemo koristiti proširenu metodu:

```
toMap(keyMapper, valueMapper,
        (u, v) -> {
                throw new IllegalStateException(String.format("Duplicate key %s", u));
        },
        LinkedHashMap::new);
```

— Kolektor **toMap** odbija NULL-vrijednosti. Ovo se također može riješiti vlastitim kolektorom. Postoje rješenja na internetu.

API: Collectors.groupingBy()

- Jedna od varijacija na temu skupljanja elementa u mapu su kolektori **groupingBy** i **partitioningBy**.
  - **groupingBy** – proizvodi mapu elemenata grupiranih u listu po nekom ključu: **Map<K, List<E>>**
  - **partitioningBy** – proizvodi mapu elemenata grupiranih u listu po nekom uvjetu: **Map<Boolean, List<E>>**
- Ponašanje je slično SQL direktivi GROUP BY.

```
Map<String, List<User>> usersByAddress =
        users.collect(groupingBy(User::getAddress));
```

Međurezultati obrade ulaznih elemenata

- Ponekad se ulazni elementi skupljaju u jednu strukturu, da bi se onda ponovo obradili i završili u drugoj strukturi, npr.: **source -> list -> map**
- Klasično rješenje:
  - **List list = source.collect(toList)**
  - *(prostor za neplanirane greške☺)*
  - **Map map = list.stream()... collect(toMap)**
- Manjkavost ovakvog rješenja je da se vrlo lako da izmijeniti kod i pokvariti pomoćna struktura „**list**“, npr.: **source -> list -> list' -> map**

Osim spajanjem dviju transformacija u jednu, postoji i poseban kolektor **collectingAndThen** koji može povezati te dvije operacije:

```
source.collect(
        collectingAndThen(
                toList(),
                list -> list.stream().collect(toMap(…))));
```

API: map/flatMap

- **map -** pretvara ulazni element u novi element
  - ABC, df, xyzuw
    -> map(addLengthInfo)
    -> (ABC, 3), (df, 2), (xyzuw, 5)
  - Novi element je novog tipa, što mijenja tip izvora za slijedeće operacije:
  - **Stream<T>.map(t -> u) -> Stream<U>**
- **flatMap -** pretvara ulazni element u niz (!) novih ulaznih elemenata
  - ABC, df, xyzuw
    -> flatMap(toCharacterStream)
    -> A, B, C, d, f, x, y, z, u, w

```
List<List<String>> batchCommands;
batchCommands.stream()
        .flatMap(batch -> batch.stream())
        .map(cmd -> cmd + ";")
        .forEach(cmd -> execute(cmd));
```

API: pretraživanje

— **filter** – propušta samo ulazne elemente koji zadovoljavaju zadani uvjet
— **findFirst/findAny** - vraća prvi odnosno bilo koji ulazni element, tip Optional
— **anyMatch/noneMatch** - vraća **true** ako ima bar jednog odnosno nema nijednog elementa koji zadovoljavaju zadani uvjet.

```
String message = commands
        .map(this::execute)
        .filter(errorMsg -> !errorMsg.isEmpty())
        .findFirst()
        .orElseGet(() -> "success");
```

API: još neke pomoćne metode

- Još malo metoda:
  - **distinct** – propušta samo jedinstvene elemente.
  - **sorted -** sortira ulazne elemente prema zadanom „komparatoru".
  - **limit** – vraća prvih N zadanih elemenata
  - **skip** – preskače prvih N zadanih elemenata
  - **concat** – spaja dva Stream objekta u jedan niz.

- Java 9 uvodi i operacije:
  - **takeWhile** – vraća ulazne elemente dok je ispunjen uvjet
  - **dropWhile** – preskače ulazne elemente dok je ispunjen uvjet

Praktični primjeri

Što očekivati kada ljudi počnu pisati kod u funkcionalnom stilu?

Primjer: Zbroj ocjena (1)

Zadatak:

Potrebno je zbrojiti sve jedinstvene pozitivne brojeve u ulaznoj listi. Lista može sadržavati NULL.

Ulazna struktura:

```
Collection<Integer> BROJEVI = Arrays.asList(null, -1, 1, 1, 1, 2);
```

Primjer: Zbroj ocjena (2)
klasični primjer

```
public int zbrojiPozitivneJedinstvene(Collection<Integer> brojevi) {
        Set<Integer> temp_Jedinstveni = new HashSet<>();
        int sum = 0;
        for (Integer broj : brojevi) {
                if (broj != null && broj > 0 /* && new condition? */)
{
                        boolean isNew = temp_Jedinstveni.add(broj);
                        if (isNew) {
                                sum += broj;
                        }
                }
        }
        return sum;
}
```

- Broj razina = 4
- Novo pravilo bi proširilo izraz za uvjet obrade.

Primjer: Zbroj ocjena (3)
funkcionalni primjer

```
public int zbrojiPozitivneJedinstvene (Collection<Integer> brojevi) {
        return brojevi.stream()
                .filter(broj -> broj != null)
                // new condition?
                .distinct()
                .mapToInt(broj -> broj)
                .filter(broj -> broj > 0)
                .sum();
}
```

- Broj razina = 1
- Novo pravilo filtera dodaje samo jednu liniju

Primjer: Natjecanje

Struktura rezultat predstavlja zapis o rezultatu nekog učenika na nekom testu.

Sljedeći primjeri transformiraju listu ovih zapisa u različite tražene izlazne strukture.

```
class Rezultat {
        public String getSkola();
        public String getTest();
        public String getUcenik();
        public Integer getOcjena();
}
Collection<Rezultat> REZULTATI = new ArrayList<>();
```

Primjer: Natjecanje

Zadatak 1:
Iz liste rezultata razdvojiti rezultate po školama i po testu.

- *Izlazna struktura*

```
Map<String, Map<String, List<Rezultat>>> rezultatiPoSkolamaPoTestu;
```

OŠ Kraljevac	Matematika	Tomislav	3
OŠ Knežija	Matematika	Zdeslav	5
OŠ Knežija	Matematika	Trpimir	4

OŠ Kraljevac	Matematika	[3]
OŠ Knežija	Matematika	[5, 4]

Primjer: Natjecanje – Zadatak 1
klasični primjer

```
Map<String, Map<String, List<Rezultat>>> skolaRezultati = new HashMap<>();
        for (Rezultat rezultat : REZULTATI) {
                String skola = rezultat.getSkola();
                Map<String, List<Rezultat>> testoviSkole = skolaRezultati.get(skola);
                if (testoviSkole == null) {
                        testoviSkole = new HashMap<>();
                        skolaRezultati.put(skola, testoviSkole);
                }
                String test = rezultat.getTest();
                List<Rezultat> rezultatiTestaSkole = testoviSkole.get(test);
                if (rezultatiTestaSkole == null) {
                        rezultatiTestaSkole = new ArrayList<>();
                        testoviSkole.put(test, rezultatiTestaSkole);
                }
                rezultatiTestaSkole.add(rezultat);
        }
        return skolaRezultati;
```

Primjer: Natjecanje – Zadatak 1
funkcionalni primjer

```
return REZULTATI.stream()
        .collect(
                groupingBy(
                        Rezultat::getSkola,
                        groupingBy(Rezultat::getTest, toList())));
```

Poprilično je zahvalan primjer, no za očekivati je da ćete u prvoj verziji dobiti ovo...

Primjer: Natjecanje – Zadatak 1
funkcionalni primjer 2

```
Map<String, List<Rezultat>> skoleRezultati = REZULTATI.stream()
        .collect(groupingBy(Rezultat::getSkola, toList()));
return skoleRezultati.entrySet().stream()
        .collect(toMap(
                Entry::getKey,
                skoleRezultatiEntry -> {
                        Collection<Rezultat> rezultatiSkole = skoleRezultatiEntry.getValue();
                        return rezultatiSkole.stream()
                                .collect(groupingBy(
                                        Rezultat::getTest,
                                        toList()));
                }));
```

Dakle, više iskustva donosi bolji kod. Zato je potrebno odmah krenuti s učenjem! ☺

Primjer: Natjecanje – Zadatak 2

Zadatak 2:

Iz liste rezultata razvrstati ocijenjene rezultate (1-5) prema kategoriji ocjene (pozitivan i negativan). Preskočiti diskvalificirane, pogrešne unose ocjena ili neocijenjene učenike (npr. 0, 8, NULL).

```
enum KategorijaOcjene {
        POZITIVAN, NEGATIVAN, DISKVALIFICIRAN, NEOCIJENJEN, GRESKA
}
```

- *Izlazna struktura*

```
Map<KategorijaOcjene, List<Rezultat>> rezultatiPoKategoriji
```

Postoje dva načina da se ovo riješi.

Primjer: Natjecanje – Zadatak 2
način 1

```
return REZULTATI.stream()
        .filter(rezultat -> {
                KategorijaOcjene kategorija = kategoriziraj(rezulat.getOcjena());
                return isOcijenjen(kategorija);
        })
        .collect(groupingBy(rezultat -> kategoriziraj(rezulat.getOcjena())));
```

- Problem je što se metoda kategoriziraj zove 2 puta. Performanse?

Možemo li izbjeći 2 poziva?

Primjer: Natjecanje – Zadatak 2
način 2

```
return REZULTATI.stream()
        .map(rezultat -> new SimpleEntry<>(rezultat, kategoriziraj(rezulat.getOcjena())))
        .filter(e -> isOcijenjen(e.getValue())
        .collect(groupingBy(Entry::getValue, mapping(Entry::getKey, toList())));
```

Ovdje transformiramo ulazne elemente u „tuple" objekt Entry kako bi uparili ocjenu s izračunatom kategorijom, po kojoj zatim filtriramo i grupiramo.

Rez (4)
Rez (1)
Rez (-1)

Rez (4)	POZITIVAN
Rez (1)	NEGATIVAN
Rez (-1)	DISKVALIFICIRAN

POZITIVAN	[Rez (4)]
NEGATIVAN	[Rez (1)]

Primjer: Natjecanje – Zadatak 3

Zadatak 3:
Razvrstati rezultate za svaku kategoriju ocjene. Ako nije bilo rezultata u određenoj kategoriji uvrstiti praznu listu.

Izlazna struktura – ista kao i kod prethodnog zadatka.

Opet, nekoliko načina...

Primjer: Natjecanje – Zadatak 3, način 1

```
Stream<…> prazneKategorije = Arrays.stream(KategorijaOcjene.values())
                .map(kategorija -> new SimpleEntry<>(kategorija, Collections.emptyList()));

Stream<…> rezultatiPoKategoriji = REZULTATI.stream()
                .map(rezultat -> new SimpleEntry<>(
                        kategoriziraj(rezultat.getOcjena(),
                        Arrays.asList(rezultat))));

return Stream.concat(prazneKategorije, rezultatiPoKategoriji).collect(
        HashMap::new,
        (map, e) -> {
                List<Rezultat> list = map.get(e.getKey());
                if (list == null) {
                        list = new ArrayList<>();
                        map.put(e.getKey(), list);
                }
                list.addAll(e.getValue());
        },
        (left, right) -> {});
```

Primjer: Natjecanje – Zadatak 3
način 2

```
Supplier< Map<KategorijaOcjene, List<Rezultat>> > getResultMap =
         () -> Arrays.stream(KategorijaOcjene.values())
                 .collect(toMap(k -> k, k -> Collections.emptyList()));

return REZULTATI.stream()
        .collect(groupingBy(
                rezultat -> kategoriziraj(rezultat.getOcjena()),
                getResultMap,
                toList()
        ));
```

Primjer: Natjecanje – Zadatak 3
klasični način

```
Map<KategorijaOcjene, List<Rezultat>> rezultatiPoKategoriji = new HashMap<>();
for (KategorijaOcjene kategorija : KategorijaOcjene.values()) {
        rezultatiPoKategoriji.put(kategorija, new ArrayList<>());
}
for (Rezultat rezultat : Data.REZULTATI) {
        KategorijaOcjene kategorija = kategoriziraj(rezultat.getOcjena());
        List<Rezultat> rezultatiKategorije = rezultatiPoKategoriji.get(kategorija);
        // optmizacija: lista sigurno nije null !
        rezultatiKategorije.add(rezultat);
}
return rezultatiPoKategoriji;
```

— Preuranjena optimizacija?

Debugiranje u Eclipseu (1)

- Za sada je nemoguće debugirati lambde ako su napisane istoj liniji kao i metoda koja ih koristi.

```
map(k -> new SimpleEntry<>(k, new ArrayList<>()
```

  - Rješenje je dodati novu linije tako da tijelo lambda-funkcija bude na novoj liniji.

```
map(k ->
        new SimpleEntry<>(k, new ArrayList<>()
```

Debugiranje u Eclipseu (2)

— Kod pozivanja stream metoda Eclipse je osjetljiv na najmanju greškicu, a poruka ne mora imati nikakve veze sa stvarnim problemom:

```
Arrays.stream(KategorijaOcjene.values())
        map(k -> new SimpleEntry<>(k, new ArrayList<>()));
```

— Problem: fali točka prije poziva metode.

— Poruka:

Type mismatch: cannot convert from Stream<KategorijaOcjene> to Stream<Map.Entry<KategorijaOcjene,List<Rezultat>>>

Debugiranje u Eclipseu (3)

— Uobičajena greška koju ćete dobiti je da se ne može konvertirati Object u neki ciljani tip.

Type mismatch: cannot convert from Map<Object,Object> to Map<KategorijaOcjene,List<Rezultat>>

— Problem je redovito krivi tip koji se vraća u jednoj od lambda ili krivi tip u varijabli koja prima rezultat obrade.

Memoizacija

- Želimo rezultat neke lambde „keširati" (cache).
  - Klasične opcije – cache mapa kao parametar, AOP rješenje poput Spring Cache
  - Klasa „Memoizer" – najelegantnije ako radimo s lambdama:

```
Function<String, String> doSomethingComplex = e -> e;
Function<String, String> memoSomethingComplex = Memoizer.of(doSomethingComplex);

Stream.of("a").map(memoSomethingComplex);

public class Memoizer<T,  U> {
        private final Map<T, U> cache = new ConcurrentHashMap<>();

        private Function<T, U> apply(final Function<T, U> function) {
                return input -> cache.computeIfAbsent(input, key -> function.apply(input));
        }

        public static <T, U> Function<T, U> of(final Function<T, U> function) {
                return new Memoizer<T, U>().apply(function);
        }
}
```

Što smo preskočili?

- Rekurzije
- Currying – sintaksa za ulančavanje funkcija: **f**(**g**(**h**(x))
- Generiranje streamova - **Stream.iterate**
- **IntStream.range** umjesto **for (int i = 0; i < n; i++)**
- Monad – pattern za čuvanje stanje, incijalizaciju i još neke stvari.
  - Tip Optional je monad.

Neki drugi put... ☺

Zaključak

— Funkcionalno programiranje donosi neke moćne koncepte za pisanje i dizajn koda.
— Java nije idealan FP jezik, zbog nezgrapnog API-ja i velike baze koda pisanog na klasični način što otežava usvajanje.
— Scala je dobra alternativa, ali ne znam koliki je problem koristiti ju paralelno na Java projektima.
— Većina popularnih jezika i frameworka koriste FP. Znanje stečeno u Javi mi je dosta pomoglo u radu s RxJS-om i Angularom.
— Trendovi, poput evet-driven developmenta i reaktivnog programiranja većinom imaju rješenja koja koriste FP.

— Pitanja?

— Hvala