Objektno orijentirano programiranje

13: Napredne funkcionalnosti kolekcija. Kolekcijski tokovi (Stream API)

Creative Commons

You are free to

- **Share** — copy and redistribute the material in any medium or format
- **Adapt** — remix, transform, and build upon the material

under the following terms

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **NonCommercial** — You may not use the material for commercial purposes.
- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

- https://creativecommons.org/licenses/by-nc-sa/4.0/

Klasični ispis članova mape

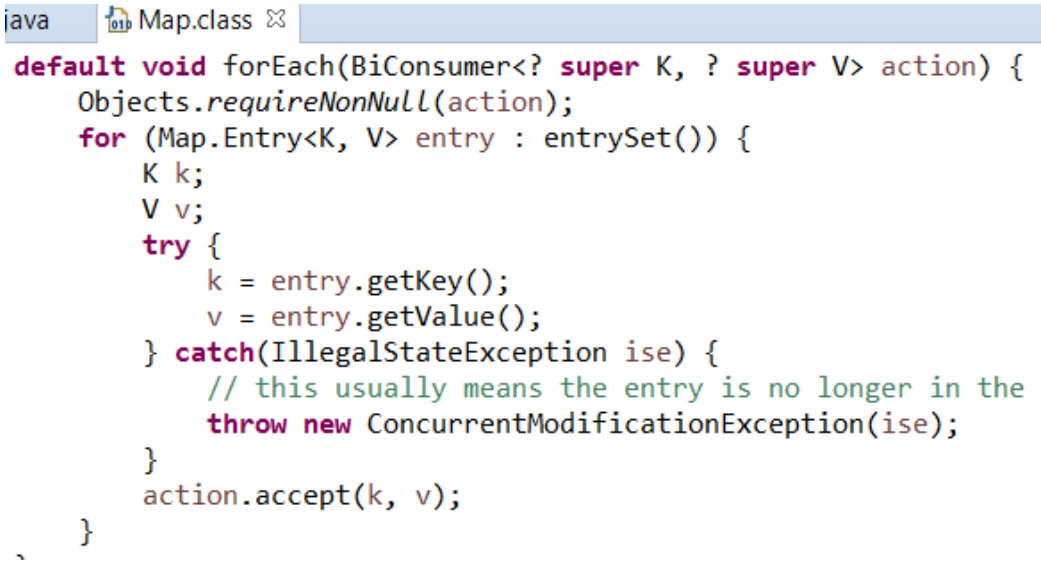
- Neka imamo mapu `Map<String,Integer> grades = …;` i trebamo nad svakim od parova ključ-vrijednost obaviti neki posao (npr. ispisati ih).
  - klasično rješenje:

```
for(Map.Entry<String, Integer> entry : grades.entrySet()) {
  System.out.format("%s ==> %s%n",
                    entry.getKey(), entry.getValue());
}
```

  - svaki puta moramo pisati ovakav kôd što je redundantno
- Od Jave 8 u sučelja pojedinih kolekcija dodan je niz korisnih *default* metoda
  - upoznat ćemo se s nekima od njih na primjeru sučelja `Map`

Metoda *forEach* u sučelju *Map*

- Sučelje *Map* sadrži *default* metodu *forEach* koja sadrži implementaciju petlje nalik prethodnoj.
- Umjesto konkretnog ispisa izvršava se određena akcija opisana argumentom metode (tipa *BiConsumer <T, U>*)
  - akcija se izvršava pozivom metode *void accept(T t, U u)*
  - *T* i *U* su u ovom slučaju *K* (tip ključa), odnosno *V* (tip vrijednosti) ili njihove natklase u hijerarhiji
  - ključ para proslijeđen kao prvi parametar
  - vrijednost para proslijeđena kao drugi parametar
- Konkretnu akciju piše pozivatelj metode



Primjer korištenja metode *forEach* iz sučelja *Map*

- Ispis svakog člana mape možemo izvesti npr. pozivom metode *forEach* i proslijeđivanjem odgovarajuće anonimne klase

```
grades.forEach(new BiConsumer<String, Integer>() {
                @Override
                public void accept(String key, Integer value) {
                        System.out.format("%s => %d%n", key, value);
                }
        });
```

13_CollectionsAdvancedTopics/hr/fer/oop/defmethods/ExampleMapForEach.java

  - ugrađena metoda *forEach* s prethodnog slajda obavlja šetnju po mapi i poziva metodu koju smo napisali

- Ili kraće i čitljivije upotrebom lambda izraza

```
grades.forEach((key,value) ->
System.out.format("%s => %d%n",key, value));
```

Promjena vrijednosti svim članovima mape

- Neka imamo istu prethodnu mapu, ali sada vrijednost koja je pridružena nekom ključu želimo zamijeniti nekom drugom vrijednošću koja se računa temeljem stare vrijednosti
  - Sjetite se primjera s brojanjem imena i mapom u kojoj su ključevi imena, a vrijednosti broj pojava tog imena *(predavanja 9b, primjer 5)*
    - nailaskom na sljedeće ime u datoteci u mapu želimo upisati 1 ako ime već ne postoji, odnosno želimo ga povećati za jedan ako postoji
    - taj kôd smo također već napisali **09_Collections/hr/fer/oop/maps/example5.java**
- Java 8 uvodi funkcijsko sučelje *BiFunction<T, U, R>* s metodom *R apply(T t, U u)* kojom se piše kôd funkcije f:T, U → R koja treba vratiti vrijednost f(t, u)
  - sučelje *Map* koristi ovo sučelje u svojoj default metodi *compute*

Metoda *compute* u sučelju *Map*

- *compute* novu vrijednost ključa stavlja u mapu umjesto stare (i vraća novu vrijednost) ili briše ključ iz mape ako je vrijednost *null*
- Izračun se vrši pomoću argumenta tipa *BiFunction <T, U, R>*
  - nova vrijednost je rezultat poziva metode *R apply(T t, U u)*
  - ključ para proslijeđen kao prvi parametar
  - vrijednost proslijeđena kao drugi parametar
  - *T* je istog tipa kao ključ mape K ili natklase u hijer.
  - *U* istog tipa kao i vrijednost V ili natklase u hijerarhiji
  - *R* tipa *V* ili izveden iz njega

ava Map.class

```
default V compute(K key,
        BiFunction<? super K, ? super V, ? extends V> remappingFunction) {
    Objects.requireNonNull(remappingFunction);
    V oldValue = get(key);

    V newValue = remappingFunction.apply(key, oldValue);
    if (newValue == null) {
        // delete mapping
        if (oldValue != null || containsKey(key)) {
            // something to remove
            remove(key);
            return null;
        } else {
            // nothing to do. Leave things as they were.
            return null;
        }
    } else {
        // add or replace old mapping
        put(key, newValue);
        return newValue;
```

Primjer korištenja metode *compute* iz sučelja *Map*

- Primjer kojim se Anti povećava ocjena za 1 (ili postavlja na 1 ako je nema)

```
Integer newGrade = grades.compute("Ante",
        new BiFunction<String, Integer, Integer>() {
                @Override
                public Integer apply(String key, Integer value) {
                        return value==null ? 1 : value+1;
                }
});
System.out.println("Now Ante has grade: " + newGrade);
```

  - Argumenti za *apply* su u ovom primjeru ključ i stara vrijednost, a rezultat nova vrijednost

- Ili kraće i čitljivije upotrebom lambda izraza

```
Integer newGrade2 = grades.compute("Ante",
                (key, value) -> value==null ? 1 : value+1);
```

13_CollectionsAdvancedTopics/hr/fer/oop/defmethods/ExampleMapCompute.java

Zašto *compute* koristi *super* i *extends* za parametrizaciju *BiFunctiona*?

- Kad bi se *Integer* mogao naslijediti, mogli bi npr. napisati klasu *MojInt*
  - hijerarhija bi tada bila *Object – Number – Integer – MojInt*
- Tada bi drugi argument u prethodnom primjeru mogao biti i objekt klase koja implementira

  `BiFunction<String, Number, MojInt>`

  npr.

```
Integer newGrade = grades.compute("Ante",
        new BiFunction<String, Number, MojInt>() {
                @Override
                public MojInt apply(String t, Number u) {
                        return u==null ? 1 : u+1;
                }
});
```

Metoda *merge* iz sučelja *Map*

- Sličnu funkcionalnost iz prethodnog primjera može se ostvariti i funkcijom *merge* čiji je prototip

```
default V merge(K key, V value,
  BiFunction<? super V, ? super V, ? extends V> remappingFunction);
```

  - ako vrijednost za ključ (*key*) ne postoji, treba je postaviti na predanu vrijednost (*value*), a inače je treba zamijeniti transformacijom koja se računa na temelju stare vrijednosti i predane vrijednosti (*remmapingFunction*)
  - Funkcija je opet *BiFunction,* ali su sada argumenti stara vrijednost i predana vrijednost, a rezultat nova vrijednost (svi tipovi su tipovi vrijednosti)

```
Integer newGrade = grades.merge("Ante", 1, (oldValue,value) ->
oldValue + value);
```

13_CollectionsAdvancedTopics/hr/fer/oop/defmethods/ExampleMapMerge.java

    - *oldvalue* će biti stara vrijednost, *value* će biti 1 (vrijednost 2. argumenta metode *merge*)

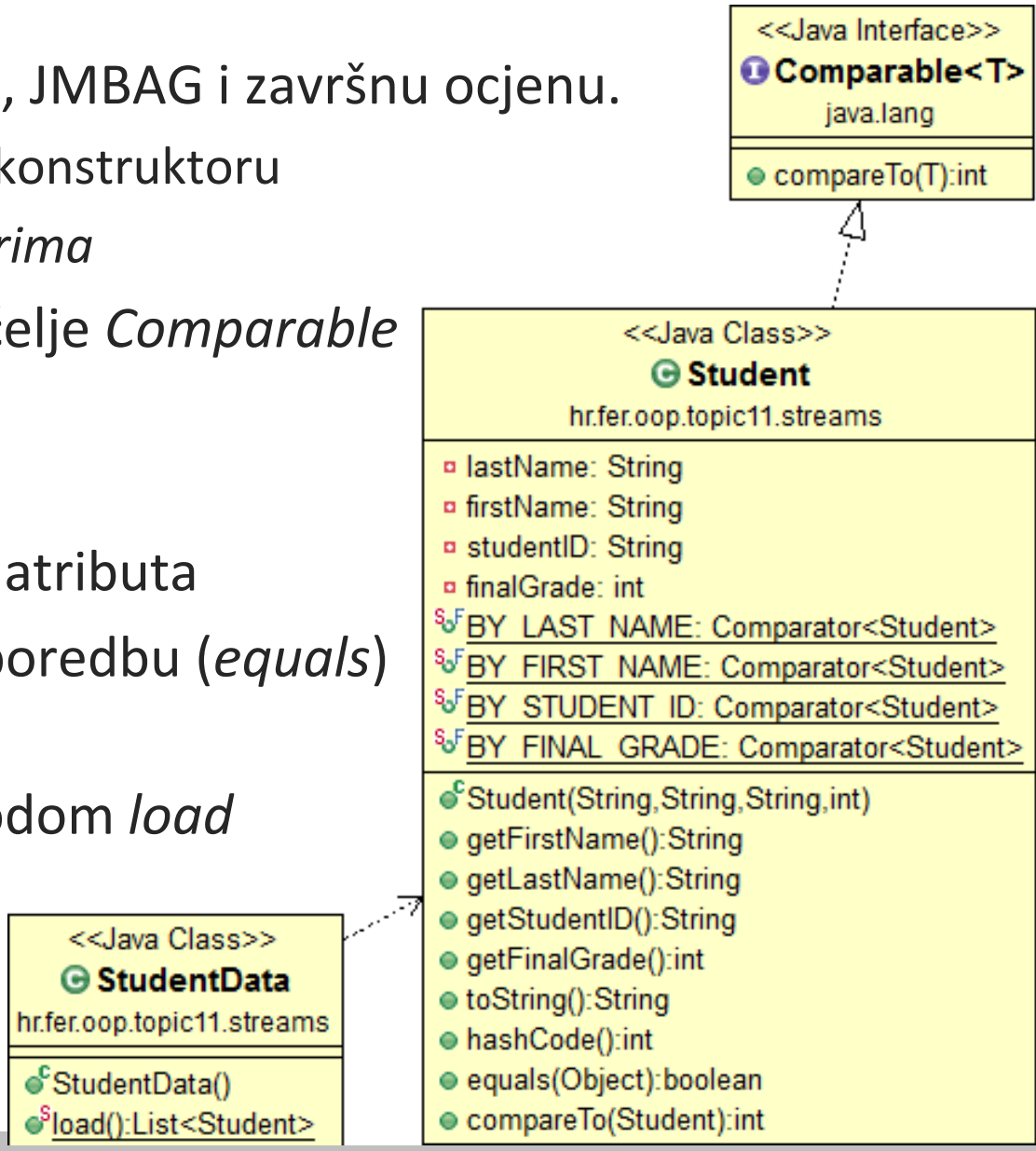
- Ovakvih pomoćnih funkcija ima još puno i u ostalim sučeljima
  - nećemo ih sve navoditi, ali bilo bi korisno upoznati se s njima

Kolekcijski tokovi

- Kolekciju možemo promatrati kao tok (engl. *stream*) pohranjenih elemenata
- Tok se **prijelaznim** (engl. *intermediate*) **metodama** može transformirati u tok nekih drugih elemenata koji će se računati temeljem originalnih elemenata
  - ove transformacije se mogu ulančavati
- Na kraju se tok **terminalnom** (engl. *terminal*) **metodom** pretvara u kolekciju ili reducira u neki drugi rezultat (npr. u sumu ili prosjek elemenata, …)
  - „konzumiramo” tok – tek tada počinje preuzimanje podataka iz kolekcije i primjena postupaka u sredini toka
- Tok ima ulogu cjevovoda, nije spremište podataka – konzumiranje „puštamo” podatke u cjevovod koji smo prethodno složili

Klasa Student koja se koristi u primjerima s tokovima

- *Student* ima ime, prezime, JMBAG i završnu ocjenu.
  - atributi se postavljaju u konstruktoru
  - dohvat vrijednosti *getterima*
- *Student* implementira sučelje *Comparable*
  - usporedba po JMBAG-u
  - prirodni komparator
- Komparator za svaki od 4 atributa
- Nadjačane metode za usporedbu (*equals*) i sažetak (*hashcode*)
- Klasa *StudentData* s metodom *load* za dohvat liste testnih podataka



Komparator specifičan za pojedini jezik

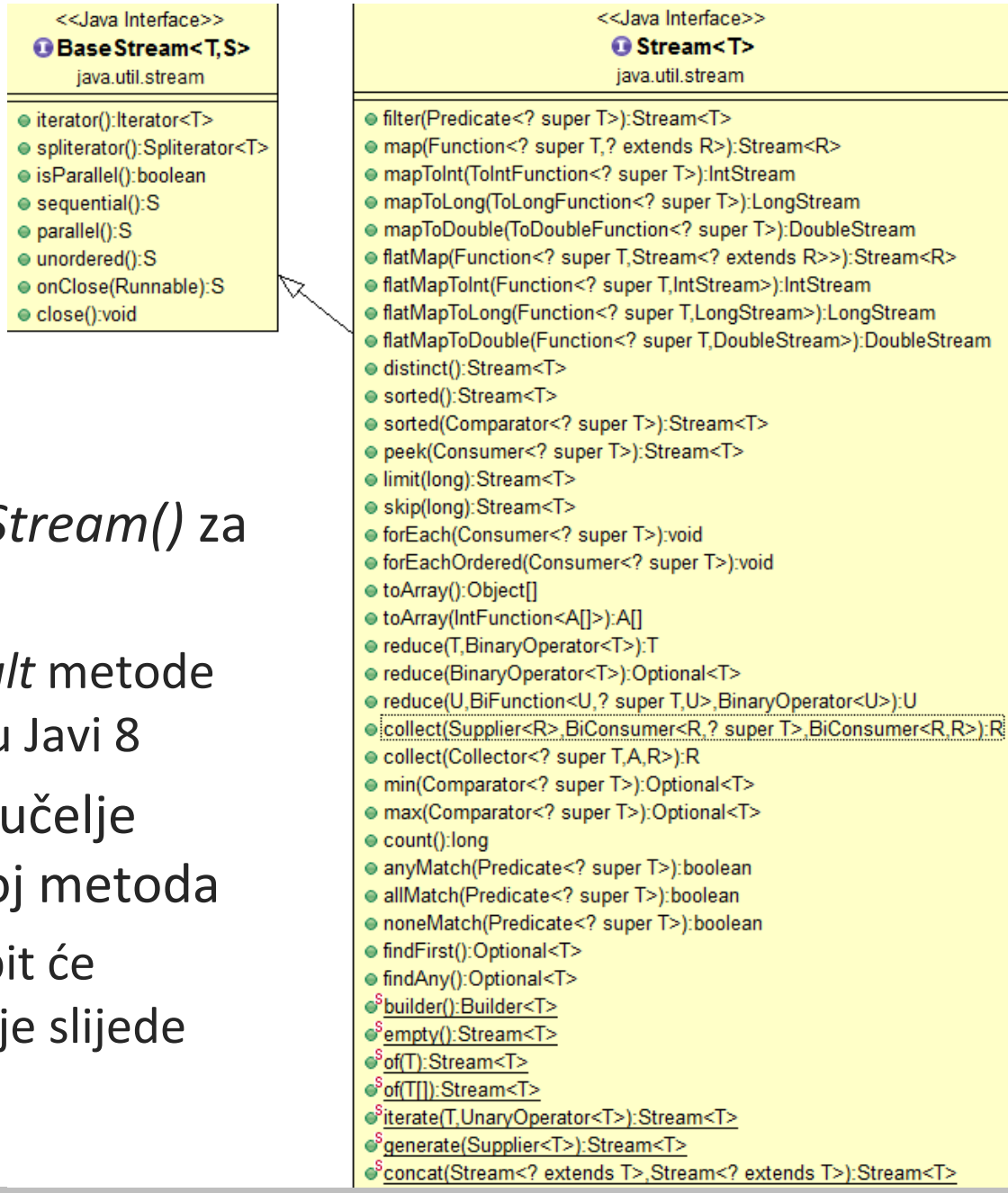
- Definira li se usporedba prezimena s komparatorom iz prethodnih predavanja kao `(o1, o2) -> o1.compareTo(o2)` pojavit će se problem u slučaju hrvatskih slova.
  - Npr. `"Č".compareTo("Ć")` → 6
    ili `"Č".compareTo("D")` → 200 umjesto vrijednosti < 0
- Komparator za hrvatski jezik može se dobiti s `Collator.getInstance(Locale.forLanguageTag("hr"))`

  pa je tako u klasi *Student* definirana usporedba po prezimenima na sljedeći način

```
private static Comparator<Object> hrcomparator =
Collator.getInstance(Locale.forLanguageTag("hr"));
public static final Comparator<Student> BY_LAST_NAME =
  (o1, o2) -> hrcomparator.compare(o1.lastName, o2.lastName);
```

13_CollectionsAdvancedTopics/hr/fer/oop/streams/Student.java

Kreiranje toka neke kolekcije



- Novi tok može se stvoriti iz kolekcije pozivom metode *stream()*, odnosno *parallelStream()* za paralelno izvođenje
  - ove dvije metode su *default* metode sučelja *Collection* nastale u Javi 8
- Sučelje *Stream* nasljeđuje sučelje *BaseStream* i nudi veliki broj metoda
  - neke od češće korištenih bit će prikazane u primjerima koje slijede

Primjer upotrebe tokova

- Ispisati sve studente iz liste korištenjem tokova i metode *forEach*
  - terminalna metoda – ne vraća novi tok, već ga konzumira
  - izvršava akciju predanu kroz argument tipa `Consumer<? super T>`
  - napomena: sučelje *List* ima metodu *forEach*, ali ne radi se o istoj metodi

```
List<Student> students = StudentData.load();
// using anonymous class
students.stream().forEach(new Consumer<Student>() {
        @Override
        public void accept(Student t) {
                System.out.println(t);
        }
});
// using lambda
students.stream().forEach(t -> System.out.println(t));
```

13_CollectionsAdvancedTopics/hr/fer/oop/streams/Example1.java

Primjer prijelazne operacije na tokovima

- Jedna od operacija koju tokovi podržavaju je filtriranje toka
- Filtar se postavlja metodom *filter* i pisanjem odgovarajućeg predikata

```
Stream<T> filter(Predicate<? super T> predicate);
```

- Primjer: Ispisati sve studente koji su odlikaši

```
List<Student> students = StudentData.load();
students.stream().filter(s -> s.getFinalGrade()==5)
        .forEach(t -> System.out.println(t));
```

13_CollectionsAdvancedTopics/hr/fer/oop/streams/Example2.java

- Metoda *filter* je prijelazna operacija
  - vraća novi tok na kojem se mogu primijeniti ostale operacije
  - samo definira filtar i ne „konzumira” tok, tj. ne uzrokuje šetnju po listi
- Terminalnom metodom *forEach* počinje uzimanje podataka iz toka
  - uzrokuje šetnju po listi i primjenu postavljenog filtra

Podsjetnik na funkcioniranje tokova

- Važno je napomenuti da princip rada tokova nije slijedna izgradnja kompletnih novih kolekcija ili promjena postojeće kolekcije
- Umjesto toga, bolja predodžba je koncept cjevovoda: tek kada se neki element zatraži terminalnom operacijom, krenut će se u njegov izračun: oblilaskom kroz kolekciju ne upravljate eksplicitno iteratorom već se iteracija događa implicitno, po potrebi
  - prijelazne metode koje vraćaju nove tokove možemo predočiti kao različite vrste cijevi od kojih radimo cjevovod
  - npr. metoda za vraćanje sortiranog toka ne sortira ulaznu kolekciju niti vraća sortiranu kolekciju – ona uzrokuje da će se podaci uzimati sortirano (jednom kad krene obilazak kolekcije)
- Jednom iskorišten (konzumiran) tok neke kolekcije ne može se ponovo koristiti, već se mora stvoriti novi tok iz te kolekcije

Pokušaj korištenja iskorištenog toka

- Pokušaj ponovnog korištenja iskorištenog (konzumiranog) toka uzrokuje iznimku *IllegalStateException*

```
List<Student> students = StudentData.load();
Stream<Student> st = students.stream();
st.forEach(t -> System.out.println(t)); //OK

//st.forEach(t -> System.out.println(t));
//uzrokovao bi IllegalStateException, jer je st
//već konzumiran

students.stream().forEach(t -> System.out.println(t)); //OK
//.stream() na kolekciji stvara novi tok
```

13_CollectionsAdvancedTopics/hr/fer/oop/streams/Example3.java

Ispis sadržaj toka sortiranog po nekom komparatoru

- Prijelazna metoda `sorted()` za sortiranje po prirodnom komparatoru ili `sorted(Comparator<? super T>)` za sortiranje po proizvoljnom komparatoru.

```
List<Student> students = StudentData.load();
//ispiši sve s ocjenom 5 sortirano po prezimenu
students.stream()
        .filter(s -> s.getFinalGrade() == 5)
                .sorted(Student.BY_LAST_NAME)
                .forEach(t -> System.out.println(t));

//ispisuje sve iz liste studenata (nesortirno)
//originalna kolekcija nije promijenjena
students.stream().forEach(t -> System.out.println(t));
```

13_CollectionsAdvancedTopics/hr/fer/oop/streams/Example4.java

Preslikavanja jednog toka u drugi tok

- Preslikavanje (transformacija) jednog toka u drugi tok vrši se metodom map

```
public interface Stream<T> extends BaseStream<T, Stream<T>> {
  ...
  <R> Stream<R> map(Function<? super T, ? extends R> mapper);
```

- Tok elemenata tipa *T* transformira se u tok elemenata tipa *R* korištenjem funkcije koja prima argument tipa *T* ili viši u hijerarhiji, a vraća element tipa *R* ili izveden iz *R*.
- Primjer: tok studenata možemo transformirati u tok njihovih prezimena (tok *Stringova*)
  - kompletan primjer slijedi u nastavku nakon primjera stvaranja kolekcije iz toka

Pretvaranje toka u kolekciju

- Operacija *collect* je terminalna operacija koja ne vraća novi tok već konačni rezultat obrade
- Razred *Collectors* sadrži nekoliko statičkih funkcija koje vraćaju gotove kolektore; primjerice, kolektor koji elemente toka transformira u listu, skup ili mapu
  - moguće je napisati vlastite kolektore implementacijom sučelja *Collector* ili koristiti preopterećenu verziju metode *collect* s 3 parametra
  - pisanje takvih metoda izlazi iz okvira ovih predavanja

Primjer korištenja više prijelaznih funkcija i kolektora

- Toku iz liste studenata postavlja se filtar, transformira ga se u tok *Stringova* (s prezimenima studenata), a zatim se njegov sadržaj stavlja u listu (*Stringova*).

13_CollectionsAdvancedTopics/hr/fer/oop/streams/Example4.java

```
List<Student> students = StudentData.load();
List<String> studentIDs = students.stream()
  .filter(new Predicate<Student>() {
          @Override
          public boolean test(Student t) {
                return t.getFinalGrade()>3;
           }
        })
        .map(new Function<Student, String>(){
           @Override
           public String apply(Student t) {
                 return t.getLastName();
            }
         })
         .collect(Collectors.toList());
```

Kraće korištenjem labda izraza:

```
studentIds = students.stream()
        .filter(s ->
s.getFinalGrade()>3)
        .map(s ->
s.getLastName())
.collect(Collectors.toList());
```

Preslikavanje toka u tok primitivnih vrijednosti

- Za preslikavanje toka u tok podataka tipa *Integer*, *Long* ili *Double* postoje metode *mapToInt*, *mapToLong*, *mapToDouble* koje vraćaju *IntStream*, *LongStream*, *DoubleStream* – izvedene varijante tokova s nekim dodatnim metodama za izračun minimalne, maksimalne, srednje vrijednosti i slično…
  - Metoda *average*() vraća primjerak *OptionalDouble* koji pamti ima li pohranjenu double vrijednosti (metoda *isPresent*()) te nudi metodu *getAsDouble*() koja vraća taj double ako postoji odnosno baca *NoSuchElementException* ako ga nema
- Primjer: izračun prosječne vrijednosti studenata s ocjenom većom od 2

```
List<Student> students = StudentData.load();
double avgGrade2 = students
                .stream()
                .filter(s -> s.getFinalGrade()>2)
                .mapToInt(s -> s.getFinalGrade())
                .average()
                .getAsDouble();
```

13_CollectionsAdvancedTopics/hr/fer/oop/streams/Example6.java

Operacije redukcije

- Metoda *average* iz prethodnog primjera i slične (min, max, count, …) pripadaju **redukcijskim** metodama koje tok svode na jednu opcionalnu vrijednost
  - ta vrijednost može biti bilo kojeg tipa *Optional<T>*
- Korištenjem metode *reduce* moguće je napisati vlastite redukcijske metode
  - `Optional<T> reduce(BinaryOperator<T> accumulator)`
  - pisanje takvih metoda izlazi iz okvira ovih predavanja

Primjer korištenja kolekcijskih tokova sa zip datotekama (1/2)

- Primjer – ispis prva 3 retka svake tekstualne datoteke unutar neke zip datoteke
  - iz zip datoteke možemo dobiti kolekcijski tok

```
try(ZipFile zip = new ZipFile(filename)){
        zip.stream()
     .filter(entry ->
           entry.getName().toLowerCase().endsWith(".txt"))
          .forEach(entry -> write3LinesWithScanner(zip, entry));
} ...
```

13_CollectionsAdvancedTopics/hr/fer/oop/streams/ZipExample.java

  - metoda *write3LinesWithScanner* je vlastita metoda unutar iste klase
    - Nema veze s kolekcijskim tokovima, ali svejedno proučite kod kao podsjetnik na Scanner i datotečne tokove

Primjer korištenja kolekcijskih tokova sa zip datotekama (2/2)

- Kao parametar moguće je predati i referencu na postojeću metodu nekog objekta (ili statičku metodu klase) koja potpisom odgovara traženom parametru
  - za *forEach* to je Consumer<? super T>, a u ovom primjeru T je *ZipEntry*

```
zip.stream()
   ...
        .forEach(entry -> write3LinesWithScanner(zip, entry));
```

13_CollectionsAdvancedTopics/hr/fer/oop/streams/ZipExample.java

```
ZipContentWriter zcw = new ZipContentWriter(zip);
zip.stream()
   ...
        .forEach(zcw::writeContent);

...
private static class ZipContentWriter {
       private void writeContent(ZipEntry entry){ ...
```

13_CollectionsAdvancedTopics/hr/fer/oop/streams/ZipExample2.java