

Objektno orijentirano programiranje

13. 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/>



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*

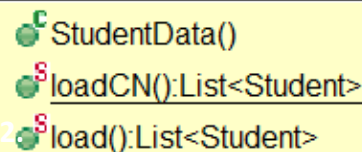
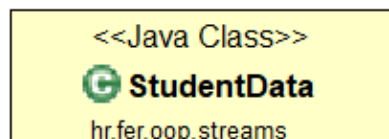
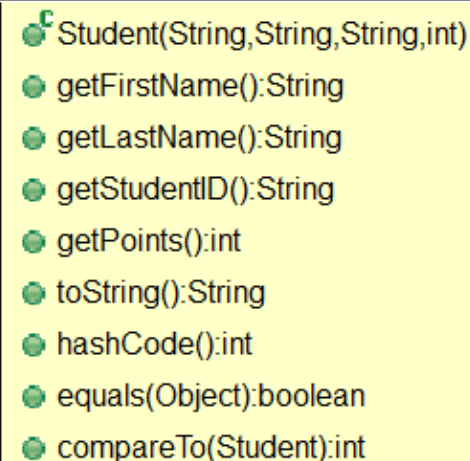
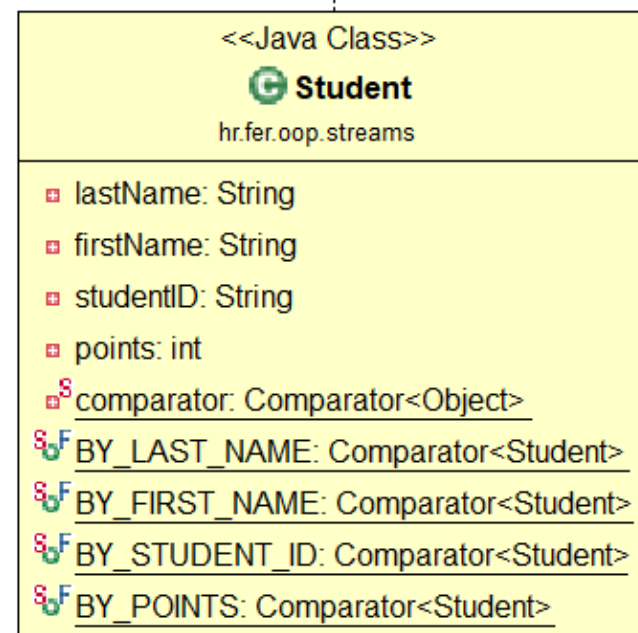
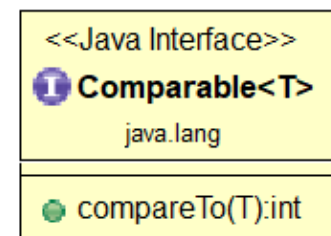
- usporedba po JMBAG-u

- 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 (1)

- 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_StreamAPI/hr/fer/oop/streams/Student.java

Komparator specifičan za pojedini jezik (1)

- Moguće definirati komparator za pojedini jezik

```
Comparator<Object> compHR = Collator.getInstance(  
    Locale.forLanguageTag("hr"));  
Comparator<Object> compChina = Collator.getInstance(Locale.CHINA);  
  
String s1 = "王"; //Wáng je ispred Zhōu  
String s2 = "周"; //Zhōu  
System.out.println(s1.compareTo(s2)); // 7971  
System.out.println(compChina.compare(s1, s2)); // -1  
System.out.println(compHR.compare(s1, s2)); // 1  
  
s1 = "č";  
s2 = "ć";  
System.out.println(s1.compareTo(s2)); // 6  
System.out.println(compChina.compare(s1, s2)); // 1  
System.out.println(compHR.compare(s1, s2)); // -1
```

13_StreamAPI/hr/fer/oop/streams/CompareTest.java

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

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
 - postoje i specifičniji kao što su *IntStream*, *LongStream* i *DoubleStream* s dodatnim metodama

```
<<Java Interface>>
1 BaseStream<T,S>
  java.util.stream

• iterator():Iterator<T>
• spliterator():Spliterator<T>
• isParallel():boolean
• sequential():S
• parallel():S
• unordered():S
• onClose(Runnable):S
• close():void
```

```
<<Java Interface>>
1 Stream<T>
  java.util.stream

• filter(Predicate<? super T>):Stream<T>
• map(Function<? super T,? extends R>):Stream<R>
• mapToInt(ToIntFunction<? super T>):IntStream
• mapToLong(ToLongFunction<? super T>):LongStream
• mapToDouble(ToDoubleFunction<? super T>):DoubleStream
• flatMap(Function<? super T,Stream<? extends R>>):Stream<R>
• flatMapToInt(Function<? super T,IntStream>):IntStream
• flatMapToLong(Function<? super T,LongStream>):LongStream
• flatMapToDouble(Function<? super T,DoubleStream>):DoubleStream
• distinct():Stream<T>
• sorted():Stream<T>
• sorted(Comparator<? super T>):Stream<T>
• peek(Consumer<? super T>):Stream<T>
• limit(long):Stream<T>
• skip(long):Stream<T>
• forEach(Consumer<? super T>):void
• forEachOrdered(Consumer<? super T>):void
• toArray():Object[]
• toArray(IntFunction<A[]>):A[]
• reduce(T,BinaryOperator<T>):T
• reduce(BinaryOperator<T>):Optional<T>
• reduce(U,BiFunction<U,? super T,U>,BinaryOperator<U>):U
• collect(Supplier<R>,BiConsumer<R,? super T>,BiConsumer<R,R>):R
• collect(Collector<? super T,A,R>):R
• min(Comparator<? super T>):Optional<T>
• max(Comparator<? super T>):Optional<T>
• count():long
• anyMatch(Predicate<? super T>):boolean
• allMatch(Predicate<? super T>):boolean
• noneMatch(Predicate<? super T>):boolean
• findFirst():Optional<T>
• findAny():Optional<T>
• builder():Builder<T>
• empty():Stream<T>
• of(T):Stream<T>
• of(T[]):Stream<T>
• iterate(T,UnaryOperator<T>):Stream<T>
• generate(Supplier<T>):Stream<T>
• concat(Stream<? extends T>,Stream<? extends T>):Stream<T>
```

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_StreamAPI/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 imaju 40 ili više bodova

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

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

- Metoda *filter* je prijelazna operacija
 - vraća novi tok na kojem se mogu primijeniti ostale operacije
 - samo definira filtari 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: obilaskom 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_StreamAPI/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 40 ili više bodova sortirano po prezimenu
students.stream()
    .filter(s -> s.getPoints() >= 40)
    .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_StreamAPI/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

13_StreamAPI/hr/fer/oop/streams/Example5.java

```
lastNames = students.stream()  
    .filter(s -> s.getPoints() > 30)  
    .map(s -> s.getLastName())  
    ...
```

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 (1)

13_StreamAPI/hr/fer/oop/streams/Example5.java

- Toku iz liste studenata postavlja se filter, transformira u tok prezimena (*Stringova*), sortira i sprema u listu (*Stringova*).

```
List<Student> students = StudentData.load();
List<String> lastNames = students.stream()
    .filter(new Predicate<Student>() {
        @Override
        public boolean test(Student t) {
            return t.getPoints() > 30;
        }
    })
    .map(new Function<Student, String>() {
        @Override
        public String apply(Student t) {
            return t.getLastName();
        }
    })
    .sorted()
    .collect(Collectors.toList());
```

```
lastNames = students.stream()
    .filter(s -> s.getPoints() > 30)
    .map(s -> s.getLastName())
    .sorted()
    .collect(Collectors.toList());
```

Kraće korištenjem lambda izraza:

Primjer korištenja više prijelaznih funkcija i kolektora (2)

- Tip rezultata ovisi o kolektoru
 - U slučaju toka Stringova koristan može biti kolektor koji spaja elemente u jedan string koristeći navedeni string između

13_StreamAPI/hr/fer/oop/streams/Example5b.java

```
List<Student> students = StudentData.load();
String lastNames = students.stream()
    .filter(s -> s.getPoints() > 30)
    .map(s -> s.getLastName())
    .sorted() // ili sorted (comp)
    .collect(Collectors.joining(", "));

System.out.println(lastNames);
```

- Metoda *sorted* može imati komparator kao argument
 - npr. *Comparator<Object> comp = Collator.getInstance(Locale.CHINA);*

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

```
double avgGrade = students.stream()
    .filter(s -> s.getPoints() > threshold)
    .mapToInt(s -> s.getPoints()) // IntStream
    .average() //OptionalDouble
    .getAsDouble(); // double or exception throws
```

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

- varijanta s *idPresentOrElse* prikazana u

13_StreamAPI/hr/fer/oop/streams/Example6b.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_StreamAPI/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_StreamAPI/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_StreamAPI/hr/fer/oop/streams/ZipExample2.java