

# Java tečaj

4. dio

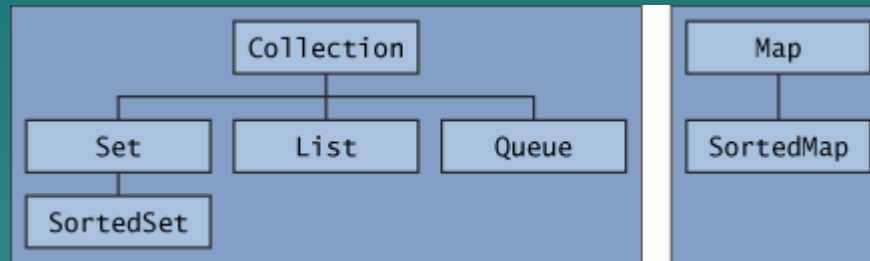
Kolekcije, 2. dio

# Kolekcije

- ◆ Nastavimo dalje s kolekcijama...
- ◆ Prošli puta smo vidjeli osnovne tipove kolekcija u Javi, te standardne implementacije, kao i njihova svojstva
  - Npr. složenost pojedinih operacija

# Kolekcije: sučelja

- ◆ Sučelja koja definiraju kolekcije



- ◆ Kolekcija – najopćenitija grupa elemenata

# Kolekcije

- ◆ Kako se kolekcije ponašaju s novim razredima?
- ◆ Primjerice, definirajmo razred Zaposlenik
- ◆ Svaki zaposlenik ima svoju sifru (jedinствeno), prezime, ime te plaću

# Kolekcije

```
package hr.fer.zemris.java.tecaj_4;
```

```
public class Zaposlenik {  
    private String sifra;  
    private String prezime;  
    private String ime;  
    private double placa;  
  
    public Zaposlenik(String sifra, String prezime, String ime) {  
        super();  
        this.sifra = sifra;  
        this.prezime = prezime;  
        this.ime = ime;  
    }  
  
    public double getPlaca() {  
        return placa;  
    }  
  
    public void setPlaca(double placa) {  
        this.placa = placa;  
    }  
    ...  
}
```

# Kolekcije

- ◆ Zbog lakšeg ispisa dodajmo metodu toString() u Zaposlenik

```
public String toString() {  
    return "Zaposlenik: sifra="+sifra  
        +", prezime="+prezime  
        +", ime="+ime+", placa="+placa;  
}
```

# Kolekcije

- ◆ Zanima nas kako će se kolekcija “lista” ponašati s ovim novim razredom
- ◆ Napravimo program koji će dodati nekoliko zapisa u listu, i potom ih sve ispisati

# Kolekcije

```
package hr.fer.zemris.java.tecaj_4;

import java.util.ArrayList;
import java.util.List;

public class DohvatZaposlenika {

    public static void main(String[] args) {
        List<Zaposlenik> lista = new ArrayList<Zaposlenik>();

        lista.add(new Zaposlenik("1","Peric","Pero"));
        lista.add(new Zaposlenik("2","Agic","Agata"));
        lista.add(new Zaposlenik("3","Ivic","Ivana"));

        for(Zaposlenik zap : lista) {
            System.out.println(zap);
        }
    }
}
```



# Kolekcije

## ◆ Rezultat izvođenja

Zaposlenik: sifra=1, prezime=Peric, ime=Pero, placa=0.0

Zaposlenik: sifra=2, prezime=Agic, ime=Agata, placa=0.0

Zaposlenik: sifra=3, prezime=Ivic, ime=Ivana, placa=0.0

# Kolekcije

- ◆ Proširimo program još s nekoliko linija koda:

```
Zaposlenik zaposlenik =  
    new Zaposlenik("1","Peric","Pero");  
boolean sadrziZaposlenika = lista.contains(zaposlenik);  
System.out.println("SadrziZaposlenika = "  
    + sadrziZaposlenika);
```

- ◆ Što će biti rezultat?

# Kolekcije

- ◆ Rezultat je **false**!
- ◆ Zašto? Razmislite!
- ◆ Ekvivalentno pitanje: jesmo li mogli zaposlenika tražiti metodom `indexOf`, i pronaći ga?

# Kolekcije

- ◆ U razred Zaposlenik treba dodati metodu equals() koja govori kada su dva primjerka ista (ma što termin **ista** mogao značiti)!

```
public boolean equals(Object arg0) {  
    if(arg0==null) return false;  
    if(!(arg0 instanceof Zaposlenik)) return false;  
    Zaposlenik drugi = (Zaposlenik)arg0;  
    return  
        Long.valueOf(sifra).equals(Long.valueOf(drugi.sifra))  
        && prezime.equals(drugi.prezime)  
        && ime.equals(drugi.ime);  
}
```


# Kolekcije

- ◆ Ponovimo li sada program, rezultat izvođenja je **true**!

- ◆ Pravilo:

Kako bi se omogućilo ispravno pretraživanje kolekcija metodom usporedbe na jednakost, potrebno je implementirati metodu equals

# Kolekcije

- ◆ Što se događa ako želimo zaposlenike dodavati u, primjerice, TreeSet?
  - ◆ Pokušajmo!
  - ◆ Program: ZaposleniciTree.java
- 
- A stylized, layered mountain range graphic in shades of teal and blue, located in the bottom right corner of the slide.

# Kolekcije

## ◆ Koji je rezultat izvođenja?

```
Exception in thread "main" java.lang.ClassCastException:  
    hr.fer.zemris.java.tecaj_4.Zaposlenik  
at java.util.TreeMap.compare(Unknown Source)  
at java.util.TreeMap.put(Unknown Source)  
at java.util.TreeSet.add(Unknown Source)  
at  
    hr.fer.zemris.java.tecaj_4.ZaposleniciTree.main(Zaposlenici  
    Tree.java:16)
```

## ◆ Zašto?

# Kolekcije

- ◆ TreeSet objekte pokušava sortirati!
- ◆ Da bi to mogao, mora znati kako usporediti dva objekta – a u našem slučaju to nije jasno.
- ◆ Možemo napraviti vlastiti komparator zaposlenika
  - Razred koji implementira `java.util.Comparator`



# Kolekcije

```
interface java.util.Comparator<T> {  
    int compare(T arg0, T arg1);  
}
```

## ◆ Metoda vraća:

- Negativan broj ako je  $\text{arg0} < \text{arg1}$
- Pozitivan broj ako je  $\text{arg0} > \text{arg1}$
- 0 inače (dakle, ako je  $\text{arg0}$  jednak  $\text{arg1}$ )

# Kolekcije

- ◆ Primjer: ZaposleniciTree2.java
- ◆ Eksplicitno se definiraju dva komparatora: jednostavan i kompozitni (složen)
- ◆ Konstruktor TreeSet prima argument na komparator koji zna usporediti objekte

# Kolekcije

- ◆ Međutim, što je s prirodnim poretkom? Zašto smo Integere i Stringove mogli dodavati u TreeSet?
- ◆ Ovi razredi imaju definiran “prirodni” poredak → implementiraju sučelje `java.lang.Comparable`!

# Kolekcije

```
interface java.lang.Comparable {  
    int compareTo(T arg);  
}
```

## ◆ Metoda vraća:

- Negativan broj ako je  $this < arg$
- Pozitivan broj ako je  $this > arg$
- 0 inače (dakle, ako je  $this$  jednak  $arg$ )

# Kolekcije

- ◆ Dakle, Zaposlenik-a treba proširiti:

```
public class Zaposlenik implements Comparable {  
...  
}
```

# Kolekcije

## ◆ I potom dodati compareTo:

```
public int compareTo(Object arg0) {  
    if(arg0==null) return 1;  
    Zaposlenik drugi = (Zaposlenik)arg0;  
    long res = Long.parseLong(sifra)  
                - Long.parseLong(drugi.sifra);  
    if(res<0) return -1;  
    else if(res>0) return 1;  
    else return 0;  
}
```

# Kolekcije

- ◆ Ovime smo definirali poredak koji je određen kao uzlazni slijed šifri
- ◆ Sada možemo napraviti isti primjer dodavanja u TreeSet bez da sami navodimo komparator.
- ◆ Primjer: ZaposleniciTree3.java

# Kolekcije

## ◆ Pravilo:

Kako bi se omogućio ispravan rad kolekcija koje za svoj rad koriste usporedbu objekata (veće, manje, jednako), potrebno je implementirati sučelje Comparable, ili ponuditi vanjski Comparator.



# Kolekcije

- ◆ Što nam treba za ispravan rad s mapama?
- ◆ Pogledajmo primjer: Bonus.java

# Kolekcije

```
public class Bonus {  
    public static void main(String[] args) {  
        Map<Zaposlenik,Double> bonusi = new HashMap<Zaposlenik,Double>();  
  
        bonusi.put(new Zaposlenik("1","Peric","Pero"),Double.valueOf(100.00));  
        bonusi.put(new Zaposlenik("2","Agic","Agata"),Double.valueOf(200.00));  
        bonusi.put(new Zaposlenik("3","Ivic","Ivana"),Double.valueOf(300.00));  
  
        for(Zaposlenik zap : bonusi.keySet()) {  
            Double bonus = bonusi.get(zap);  
            System.out.println(zap.toString()+" ima bonus od "+bonus+" kn.");  
        }  
    }  
}
```

# Kolekcije

- ◆ Što je rezultat dodavanja sljedećeg koda?

```
Double bonusOd1 =  
    bonusi.get(new Zaposlenik("1","Peric","Pero"));  
System.out.println(  
    "Bonus zaposlenika cija je sifra 1 je "  
    +bonusOd1+" kn.");
```

# Kolekcije

- ◆ Rezultat je **false**!
- ◆ Zašto? Razmislite!

# Kolekcije

- ◆ Radi se, naravno, o metodi hashCode koju je potrebno implementirati u Zaposleniku
- ◆ Npr:

```
public int hashCode() {  
    return Long.valueOf(sifra).hashCode()  
        ^ prezime.hashCode()  
        ^ ime.hashCode();  
}
```

# Kolekcije


- ◆ Da bi sve radilo kako spada, treba paziti na vezu između metoda equals i hashCode
  - Za objekte koje equals proglasi istima, hashCode mora također dati identične vrijednosti
- ◆ Ako sada ponovimo prethodni primjer, rezultat će biti **true**

# Kolekcije

## ◆ Pravilo:


Da bi mogli koristiti vlastite razrede u mapama, nužno je ispravno implementirati metode equals i hashCode.

# Kolekcije

- ◆ Za rad s poljima također imamo na raspolaganju gotove metode!
  - ◆ `java.util.Arrays`
  - ◆ Primjerice, sortiranje polja...
  - ◆ Primjer: `SortiranjePolja.java`
- 
- A stylized, layered mountain range graphic in shades of teal and blue, located in the bottom right corner of the slide.



# Kolekcije

- ◆ Uočiti da se u primjeru koriste anonimni razredi!
  - ◆ Drugi argument metode sort je primjerak nekog razreda koji implemetira sučelje Comparator<T>
  - ◆ Stoga možemo na mjestu gdje je to potrebno stvoriti anonimni razred...
- 
- A stylized, layered mountain range graphic in shades of teal and blue, located in the bottom right corner of the slide.

# Kolekcije

```
Arrays.sort(  
    zaposlenici,  
    new Comparator<Zaposlenik>() {  
        // implementacija metoda ovog  
        // razreda  
    }  
)
```

# Kolekcije

- ◆ Često ćemo trebati komparator koji uspoređuje ili u jednom "smjeru" ili u suprotnom (npr. Prezimena od A prema Z ili od Z prema A)
  - Umjesto pisanja dva gotovo identična komparatora, problem možemo riješiti tako da napišemo dekorator: novi generički komparator koji u konstruktoru prima referencu na postojeći komparator, pamti ga i u metodi compare njega pita za usporedbu pa vrati minus dobivenu vrijednost (čime okreće poredak)

```
public static class ReverseComparator<T> implements Comparator<T> {  
    private Comparator<T> original;  
    public ReverseComparator(Comparator<T> original) {  
        this.original = original;  
    }  
    public int compare(T o1, T o2) {  
        int r = original.compare(o1, o2);  
        return -r;  
    }  
}
```

# Kolekcije

- ◆ Često ćemo trebati komparator koji uspoređuje ili u jednom “smjeru” ili u suprotnom (npr. Prezimena od A prema Z ili od Z prema A)
  - Tada dalje možemo pisati:

```
Comparator<Student> comparator = new Comparator<Student>() {...};  
  
Set<Student> students = new TreeSet<>(  
    new ReverseComparator<>(comparator)  
);
```

- ◆ Dobiveni komparator može se primjenjivati na bilo koji prethodno napisani!

# Kolekcije

- ◆ S obzirom da je obrtanje rezultata usporedbe česta operacija, ne trebamo sami pisati generički razred koji to radi
  - Razred `Collections` nudi statičku metodu `reverseOrder` koja prima referencu na komparator i vraća novi komparator koji obrće njegov rezultat
  - Dapače, od Java 8 u sučelje `Comparator` dodana je defaultna metoda koja omogućava uporabom ove funkcionalnosti direktno dobivanje reverznog komparatora uz vrlo malo pisanja kôda

```
public interface Comparator<T> {  
    ...  
  
    default Comparator<T> reversed() {  
        return Collections.reverseOrder(this);  
    }  
}
```

# Kolekcije

- ♦ želimo eksplicitni komparator koji najprije usporedi po prezimenu, pa potom po imenu, pa ako je i to isto, onda po šifri

```
public static void main(String[] args) {  
    Comparator<Zaposlenik> comparator = new Comparator<Zaposlenik>() {  
        @Override  
        public int compare(Zaposlenik o1, Zaposlenik o2) {  
            int r = o1.getPrezime().compareTo(o2.getPrezime());  
            if(r!=0) return r;  
            r = o1.getIme().compareTo(o2.getIme());  
            if(r!=0) return r;  
            return o1.getSifra().compareTo(o2.getSifra());  
        }  
    };  
    Set<Zaposlenik> zaposlenici = new TreeSet<>(comparator);  
    ...  
}
```

# Kolekcije

- ◆ Ponekad je u programima potrebno podržati sortiranje po više kriterija koje korisnik može podesiti tijekom izvođenja
- ◆ Razmotrimo razred koji ima 4 članske varijable (označimo ih A, B, C i D)
  - Sortiranje možemo napraviti na  $4!$  načina ako gledamo samo redosljed varijabli koje ćemo razmatrati
  - Ako uzmemo u obzir da po svakoj varijabli možemo još sortirati "prirodno" ili obrnutim poretком, broj kombinacija se penje na  $2^4 * 4!$ , što je 384
  - Nema smisla pisati toliko različitih (gotovo identičnih) komparatora

# Kolekcije

- ◆ Ideja: dovoljno je napisati
  - Po jedan *prirodni komparator* za svaku od varijabli (uz pretpostavku da su različitog tipa) ili jedan generički komparator koji se za usporedbu oslanja na prirodan poredak samih objekata
  - Dekorator: generički komparator koji možemo koristiti za okretanje redoslijeda usporedbe
  - Dekorator: generički komparator kojemu možemo predati listu drugih komparatora i koji za usporedbu proziva svaki od predanih komparatora
- ◆ Imamo li ovo, u kôdu možemo trivijalno složiti bilo koju usporedbu
- ◆ Do sada smo prikazali gotovo sve od navedenoga; napišimo još što nedostaje...



# Kolekcije

## ◆ Usporedba prema prirodnom poretku:

```
public static class NaturalComparator<T extends Comparable<T>>
implements Comparator<T> {

    @Override
    public int compare(T o1, T o2) {
        if(o1==null) {
            return o2==null ? 0 : -1;
        } else if(o2==null) {
            return 1;
        }
        return o1.compareTo(o2);
    }
}
```

# Kolekcije

## ◆ Kompozitna usporedba

```
public static class CompositeComparator<T> implements Comparator<T> {  
    List<Comparator<T>> comparators;  
    public CompositeComparator(List<Comparator<T>> comparators) {  
        this.comparators = new ArrayList<>(comparators);  
    }  
    @SafeVarargs  
    public CompositeComparator(Comparator<T> ... comparators) {  
        this.comparators = new ArrayList<>(comparators.length);  
        Collections.addAll(this.comparators, comparators);  
    }  
    @Override  
    public int compare(T o1, T o2) {  
        for(Comparator<T> c : comparators) {  
            int r = c.compare(o1, o2);  
            if(r!=0) return r;  
        }  
        return 0;  
    }  
}
```

# Kolekcije

- ◆ Primitivni komparatori: dodati kao statičke konstante u razred Zaposlenik
- ◆ Koristit ćemo kraće definicije uporabom lambda (uočite, to su i dalje primjerci anonimnih razreda):

```
private static class Zaposlenik implements Comparable<Zaposlenik> {  
  
    public static final Comparator<Zaposlenik> BY_PREZIME =  
        (o1,o2) -> o1.prezime.compareTo(o2.prezime);  
  
    public static final Comparator<Zaposlenik> BY_IME =  
        (o1,o2) -> o1.ime.compareTo(o2.ime);  
  
    public static final Comparator<Zaposlenik> BY_SIFRA =  
        (o1,o2) -> o1.sifra.compareTo(o2.sifra);  
  
}
```

# Kolekcije

- ◆ Primjer uporabe: komparator koji najprije uspoređuje po imenu studenta i to reverzno, a u slučaju istih imena dalje uspoređuje po JMBAG-u

```
Comparator<Zaposlenik> comparator = new CompositeComparator<Zaposlenik>(
    new ReverseComparator<>(Zaposlenik.BY_IME),
    Zaposlenik.BY_SIFRA
);
```

# Kolekcije

- ◆ Sada kada razumijemo način izrade i uporabu dekoratora:
  - obrišimo ih sve iz kôda – već su napisani za nas (ali nemojmo zaboraviti što je u pozadini svega i kako je to implementirano)!
  - koristimo gotove dekoratore za sve:

```
public static void main(String[] args) {  
  
    Comparator<Zaposlenik> comparator =  
        Zaposlenik.BY_IME  
        .reversed()  
        .thenComparing(Zaposlenik.BY_SIFRA);  
  
    Set<Zaposlenik> zaposlenici = new TreeSet<>(comparator);  
  
    ...  
  
}
```

# Nove funkcionalnosti kolekcija: primjer 1

---

- Od Java 8 u sučelja pojedinih kolekcija dodan je niz defaultnih (korisnih) metoda
  - Upoznat ćemo se s nekima od njih, na primjeru sučelja Map
  - Neka imamo mapu `Map<String,Integer> mapa = ...;` i neka trebamo nad svakim od parova ključ-vrijednost obaviti neki posao (npr. ispisati ih). Klasično rješenje:

```
for (Map.Entry<String, Integer> entry :  
mapa.entrySet()) {  
    System.out.format("%s ==> %s%n",  
                        entry.getKey(), entry.getValue());  
}
```
  - Svaki puta moramo pisati ovakvu strukturu što je redundantno
-

# Nove funkcionalnosti kolekcija: primjer 1

---

- Java 8 definira sučelje `BiConsumer`

`@FunctionalInterface`

```
public interface BiConsumer<T, U> {  
    void accept(T t, U u);  
}
```

- To sučelje možemo iskoristiti za enkapsulaciju posla, gdje će prvi argument metode `accept` biti ključ (i njegov tip) a drugi vrijednost (i njezin tip) a sama metoda `accept` sadrži kôd koji radi obradu
  - Sučelje `Map` sadrži defaultnu metodu `forEach` koja sadrži implementaciju petlje i za svaki par ključ-vrijednost poziva metodu `accept` predanog objekta koji je razreda koji implementira sučelje `BiConsumer`
-

# Nove funkcionalnosti kolekcija: primjer 1

---

- Tada kraće možemo napisati obradu:

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

- Što uporabom lambda izraza postaje još kraće i čitljivije:

```
map.forEach((t,u) -> System.out.format("%s => %d%n", t, u));
```

---



## Nove funkcionalnosti kolekcija: primjer 2

---

- 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
  - 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
-

# Nove funkcionalnosti kolekcija: primjer 2

---

## ■ Java 8 uvodi sučelje `BiFunction`

`@FunctionalInterface`

```
public interface BiFunction<T, U, R> {
```

```
/**
```

```
 * Applies this function to the given arguments.
```

```
 *
```

```
 * @param t the first function argument
```

```
 * @param u the second function argument
```

```
 * @return the function result
```

```
 */
```

```
    R apply(T t, U u);
```

```
}
```

## ■ Sučelje modelira primjenu funkcije `apply` nad argumentima $u$ i $v$

## ■ Metoda `compute` prima tu funkciju i primjenjuje je na vrijednosti koje zamjenjuje rezultatom

---

## Nove funkcionalnosti kolekcija: primjer 2

---

- Evo primjera koji Anti povećava ocjenu za 1:

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

- Argumenti su ključ i stara vrijednost, a rezultat nova vrijednost
- Napomena: promjena je evidentirana i u mapi (uz to što je vraćena)!
- Metoda ima i dodatnu semantiku: ako se kao vrijednost funkcije vrati `null`, par se briše iz mape
- Uporabom lambde:

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

---

## Nove funkcionalnosti kolekcija: primjer 3

---

- Slična situacija: ako vrijednost ne postoji, treba je postaviti na predanu vrijednost, a inače je treba zamijeniti transformacijom koja se računa na temelju stare vrijednosti i predane vrijednosti
  - Koristimo defaultu metodu `merge` čiji je prototip:  

```
default V merge(K key, V value,  
    BiFunction<? super V, ? super V, ? extends V> remappingFunction);
```
  - Funkcija je opet `BiFunction` ali su sada argumenti stara vrijednost i predana vrijednost a rezultat nova vrijednost (svi tipovi su tipovi vrijednosti)
  - Evo primjera (*sljedeći slide*)
-

# Nove funkcionalnosti kolekcija: primjer 3

---

```
grades.merge(  
    "Ante",  
    Integer.valueOf(1),  
    new BiFunction<Integer, Integer, Integer>() {  
        @Override  
        public Integer apply(Integer t, Integer u) {  
            return t+u;  
        }  
    }  
);
```

- Ovo uporabom lambda izraza postaje još kraće:

```
grades.merge("Ante", 1, (o,i) -> o+i);
```

---

## Nove funkcionalnosti kolekcija: drugi primjeri

---

- Ovakvih pomoćnih funkcija ima još niz pa ih nećemo sve navoditi
- Bilo bi dobro upoznati se s njima i sada malo detaljnije ponovno proći kroz defaultne metode svih obrađenih sučelja kolekcija!

- Java 8 nudi još jednu nadogradnju rada s kolekcijama koja je temeljena na apstrakciji toka
    - Pogled koji je posebno pogodan **za modeliranje obrade podataka** koji su sadržani u kolekciji
    - Kolekciju gledamo kao tok pohranjenih elemenata
    - Taj tok možemo transformirati u tok drugih elemenata koji se računaju temeljem originalnih elemenata, ili ga možemo filtrirati (definirati koje elemente propuštamo)
    - Ove operacije možemo po potrebi ulančavati
    - Jednom kad smo gotovi, tok ponovno
      - Ili pretvaramo u kolekciju
      - Ili “kondenziramo” u neki drugi rezultat (primjerice, operacija koja uzima tok brojeva i kondenzira ga u jedan broj: njihovu sumu, ili njihov prosjek, ili ...)
-

# Uporaba tokova

---

- Kolekciju u tok pretvaramo pozivom metode `stream()` odnosno `parallelStream()`
  - Ovaj posljednji operacije izvodi paralelno (višedretveno)
  - Tokovi su po prirodi “lijeni”: ništa se ne događa sve dok konačni konzument (zadan na kraju obrade) ne zatraži elemente
    - Na svaki zahtjev konzumenta, zahtjev za elementom se propagira unatrag prema početnoj kolekciji koja u tok ubacuje elemente
-



# Uporaba tokova

---

- Radit ćemo na primjeru kolekcije studenata
  - Razred Student definira ime, prezime, JMBAG te završnu ocjenu studenta
  - Sučelje `Consumer` apstrahira obradu jednog elementa:  
`@FunctionalInterface`  
`public interface Consumer<T> {`  
    `void accept(T t);`  
`}`
  - Slično već viđenom `BiConsumer`-u
  - Najjednostavnija obrada: ispis studenta – terminalna operacija (ne vraća `stream()`)
-

# Uporaba tokova

---

Metoda `forEach` prima referencu na obradu (posao) modeliranu sučeljem `Consumer` i poziva je nad svakim elementom kolekcije:

```
public interface Stream<T> extends BaseStream<T, Stream<T>> {  
    void forEach(Consumer<? super T> action);  
    ...  
}
```

# Uporaba tokova

---

```
public static void main(String[] args) {  
  
    List<Student> students = StudentData.Load();  
  
    // Uz puno pisanja:  
    students.stream().forEach(new Consumer<Student>() {  
        @Override  
        public void accept(Student t) {  
            System.out.println(t);  
        }  
    });  
  
    // Ili kraće lambdom:  
    students.stream().forEach(t -> System.out.println(t));  
  
}
```

---

# Uporaba tokova

---

- Jedna od operacija koju tokovi podržavaju je filtriranje toka
- Filtar se definira kao objekt razreda koji implementira sučelje `Predicate` i u metodi `test` vraća `true` ili `false` za pozvani element
- Sučelje je:  

```
@FunctionalInterface  
public interface Predicate<T> {  
    boolean test(T t);  
}
```
- Primjer: ispisati sve studente koji su odličaši

# Uporaba tokova

---

- Tok nudi metodo `filter` koja prima referencu na predikat i vraća tok koji sadrži samo elemente koji su zadovoljili test

```
public interface Stream<T> extends BaseStream<T, Stream<T>> {  
    Stream<T> filter(Predicate<? super T> predicate);  
    ...  
}
```

# Uporaba tokova

---

// Uz puno pisanja:

```
students.stream()
    .filter(new Predicate<Student>() {
        @Override
        public boolean test(Student t) {
            return t.getFinalGrade()==5;
        }
    })
    .forEach(new Consumer<Student>() {
        @Override
        public void accept(Student t) {
            System.out.println(t);
        }
    });
```

// Ili kraće lambda:

```
students
    .stream()
    .filter(s -> s.getFinalGrade()==5)
    .forEach(t -> System.out.println(t));
```

---

# Uporaba tokova

---

- Prilikom obrade elemenata toka, važno je još jednom istaknuti da princip rada nije slijedna izgradnja kompletnih novih kolekcija po kojima se onda ponovno operira tokom
  - 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
-

# Uporaba tokova

---

- Operacija map elemente toka prevodi u transformirane elemente predanom funkcijom i daje tok tih novih elemenata

@FunctionalInterface

```
public interface Function<T, R> {  
    R apply(T t);  
}
```

- Metoda `map`:

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

- Primjerice tok studenata možemo transformirati u tok njihovih JMBAG-ova (primjer u nastavku)



# Uporaba tokova

---

- Operacija `collect` je terminalna operacija koja ne vraća tok već konačni rezultat obrade uporabom predanog objekta
- Razred `Collectors` sadrži nekoliko statičkih funkcija koje vraćaju gotove kolektore; primjerice, kolektor koji elemente toka transformira u listu ili skup
- Evo primjera mapiranja i pretvorbe u listu (sljedeći slide):

# Uporaba tokova

---

```
List<String> studentIDs =  
students.stream()  
    .filter(new Predicate<Student>() {  
        @Override  
        public boolean test(Student t) {  
            return t.getFinalGrade()>2;  
        }  
    })  
    .map(new Function<Student, String>() {  
        @Override  
        public String apply(Student t) {  
            return t.getStudentID();  
        }  
    })  
    .collect(Collectors.toList());
```

// Kraće:

```
List<String> studentIDs = students  
    .stream()  
    .filter(s ->  
s.getFinalGrade()>2)  
    .map(s -> s.getStudentID())  
  
    .collect(Collectors.toList());
```

# Uporaba tokova

---

- Ako tok mapiramo u primitivne vrijednosti, tada nam na raspolaganju stoje metode koje znaju računati vrijednosti poput minimalne, maksimalne, srednje i slično; evo izračuna prosječne ocjene svih studenata koji imaju ocjenu barem 3 (dan kraći prikaz):

```
double avgGrade2 = students
    .stream()
    .filter(s -> s.getFinalGrade()>2)
    .mapToInt(s -> s.getFinalGrade())
    .average()
    .getAsDouble();
```

- Metoda `.average()` vraća primjerak `OptionalDouble` koji pamti ima li pohranjene double vrijednosti (metoda `isPresent()`) te nudi metodu `getAsDouble()` koja vraća taj double ako postoji odnosno baca `NoSuchElementException` ako ga nema
-

- Detalje oko pisanja samih kolektora (koji su modelirani sučeljem `java.util.stream.Collectors`) nećemo dalje obrađivati
  - Zainteresiranog čitatelja upućujemo na dodatnu literaturu (službena Oracle-ova dokumentacija)