Objektno orijentirano programiranje

12. Upotreba vlastitih klasa s Javinim okvirom kolekcija. Komparatori.

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/

Važnije metode i sučelja za kolekcije

- Implementacije kolekcija uobičajeno se oslanjaju se na dvije metode klase *Object* za ispravan rad s elementima:
  - `boolean equals(Object o);`
  - `int hashCode();`
- Neke implementacije oslanjaju se na sučelja *Comparable* i *Comparator*

- U nastavku (nakon kratke diskusije o zajedničkom kodu za sve primjere) slijede primjeri u kojima se demonstrira rad s kolekcijama vlastitih klasa
  - primjeri se razlikuju u ovisnosti o (ne)postojanju nadjačavanja i implementacija gore navedenih metoda odnosno sučelja

Zajednički ispis u primjerima

- U svim primjerima koristi se ispis skupa ili liste studenata, pa je ispis premješten u parametriziranu statičku metodu klasu *Common*
  - Primijetiti da je parametrizirana samo metoda *printCollection*, a ne i sama klasa *Common*

```
public static <T> void printCollection(Collection<T> col) {
   for (T element : col) {
     System.out.println(element);
   }
   System.out.println();
}
```

12_CollectionsAndCustomClasses/hr/fer/oop/collections_and_customclasses/Common.java

Vlastita klasa korištena u primjerima

- Definiramo klasu Student koja ima atribute *firstName*, *lastName*, *studentID* (*i* odgovarajuće *gettere*)
  - U svakom primjeru postoji konstruktor s 3 parametra
  - Dodatni kôd razlikuje se od primjera do primjera

```
public class Student {
    private String lastName;
    private String firstName;
    private String studentID;

    public Student(String lastName, String firstName, String
studentID) {
        super();
        this.lastName = lastName;
        this.firstName = firstName;
        this.studentID = studentID;
    }
...
}
```

12_CollectionsAndCustomClasses/hr/fer/oop/collections_and_customclasses/example*/Common.java

Stvaranje parametriziranog objekta (1)

- U svim primjerima stvaraju se kolekcije testnih studenata, ali klase nisu iste zbog ilustracije različitih koncepata
  - stoga će metoda za stvaranje testnih podataka biti parametrizirana po tipu S(tudenta)
- Kako stvoriti novi objekt tipa S u metodi parametriziranoj po S?
  - U Javi nije dopušteno *new S()* pa posljedično ni *new S(ime, prezime, jmbag)*
- Rješenje:
  - definirati opis metode koja bi stvarala novi objekt (novog studenta)
  - prilikom poziva postupka za punjene kolekcije poslati referencu na implementaciju metode koja stvara primjerak S(tudenta) na osnovu 3 parametra (ime, prezime, jmbag)
    - Već viđeni koncept u primjerima sa sučeljima *Consumer*, *Predicate* i slično iz prethodnih predavanja
- Pišemo parametrizirano funkcijsko sučelje za tu svrhu

```
public interface StudentFactory<S> {
    S create(String lastName, String firstName, String studentID);
}
```

12_CollectionsAndCustomClasses/hr/fer/oop/collections_and_customclasses/StudentFactory.java

Stvaranje parametriziranog objekta (2)

- Kolekciju punimo testnim podacima u metodi *fillStudentsCollection* klase *Common*

```
public static <S> void fillStudentsCollection(Collection<S> students,
StudentFactory<S> factory) {
  S s1 = factory.create("Black", "Joe", "1234567890");
  S s2 = factory.create("Poe", "Edgar Allan", "2345678901");
  S s3 = factory.create("Kant", "Immanuel", "3456789012");
  S s4 = factory.create("Rock", "Joe", "0123456789");
  S s5 = factory.create("Black", "Joe", "5687461359");

  students.add(s1);
  students.add(s2);
  students.add(s3);
  students.add(s4);
  students.add(s5);
}
```

12_CollectionsAndCustomClasses/hr/fer/oop/collections_and_customclasses/Common.java

Primjer poziva parametrizirane metode za punjenje podataka

- Metodi *fillStudentsCollection* koja stvara probnu kolekciju potrebno je predati implementaciju funkcijskog sučelja *StudentFactory* s metodom *create* koja prima tri parametra i vraća primjerak studenta
  - posebna javna klasa, anonimna klasa, lambda izraz…
    - Npr. *Common.fillStudentsCollection(students, (ln, fn, id) -> new Student(ln, fn, id))*
  - može se predati i referenca na postojeću metodu npr. *NazivKlase::nazivMetode,* ali i referenca na konstruktor (ako postoji takav da prima upravo tražene argumente)

```
public static <S> void fillStudentsCollection(Collection<S> students,
StudentFactory<S> factory) {
...
}
```

12_CollectionsAndCustomClasses/hr/fer/oop/collections_and_customclasses/Common.java

Primjer 1. Pretraživanje liste

- Napravimo nekoliko studenata, dodajmo ih u kolekciju
  tipa *ArrayList* pa potražimo jednog od tih studenata
- Što će biti ispis?

```
I have following students:
(1234567890) Joe Black
(2345678901) Edgar Allan Poe
(3456789012) Immanuel Kant
(0123456789) Joe Rock
(5687461359) Joe Black

Poe present: false
```

```
public static void main(String[] args) {
  List<Student> students = new ArrayList<>();
  Common.fillStudentsCollection(students, Student::new);
  System.out.println("I have following students:");
  Common.printCollection(students);

  Student s = new Student("Poe", "Edgar Allan", "2345678901");
  System.out.println("Poe present: " + students.contains(s));
}
```

12_CollectionsAndCustomClasses/hr/fer/oop/collections_and_customclasses/ArrayListMain.java

Kako radi pretraga u listi?

- Klasa *ArrayList* metodu *contains*(x) izvodi tako da prolazi kroz sve elemente liste i nad svakim elementom *e* liste poziva *e.equals*(x)
  - ako metoda equals vrati *true*, metoda contains vrati *true*
  - ako niti jedan element nije jednak traženom, metoda vraća *false*
  - pogledati ovo direktno u kôdu klase *ArrayList*!
  - pogledajte kako su u klase *ArrayList* implementirane metode *indexOf(x)* te *lastIndexOf(x)*
    - Hoće li te metode raditi?
- Metoda *equals* je naslijeđena iz klase *Object* i u primjeru 1 nije bila nadjačana

Uporaba vlastitih klasa u Javinom okviru kolekcija: pravilo 1

- **Da bi se primjerci naših klasa mogli ispravno koristiti u Javinom okviru kolekcija, nužno je da klasa nadjača metodu *equals(x)* i time specificira kada su dva primjerka klase jednaka**
- Pitanje „kada su dva primjerka jednaka” je pitanje koje treba razriješiti tijekom modeliranja objekata domene
  - Nije nužno da su svi atributi jednaki, npr. može se pretpostaviti da su dva studenta jednaka ako im je jednak *StudentId* (metoda *equals* u klasi *String*)
  - Primjer 2 → Metoda *main* u *ArrayListMain* sada ispisuje *true*

```
@Override
public boolean equals(Object obj) {
  if (!(obj instanceof Student)) {
    return false;
  }
  Student other = (Student) obj;
  return this.studentID.equals(other.studentID);
}
```

12_CollectionsAndCustomClasses/hr/fer/oop/collections_and_customclasses/example2/Student.java

Primjer 3. Pretraživanje kolekcije *HashSet*

- Umjesto u listu, elementi se pospremaju u skup implementiran kao *HashSet*.
  - Klasa *Student* ima verziju *equals* iz primjera 2.
- Program ispisuje *false*
  - Razlog leži u načinu kako *HashSet* određuje gdje traži predani objekt
  - koristi se raspršeno adresiranje pomoću metode *hashCode*

```
I have following students:
(0123456789) Joe Rock
(2345678901) Edgar Allan Poe
(1234567890) Joe Black
(3456789012) Immanuel Kant
(5687461359) Joe Black

Poe present: false
```

```
public class HashSetMain {
    public static void main(String[] args) {
        Set<Student> students = new HashSet<>();
        Common.fillStudentsCollection(students, Student::new);
        System.out.println("I have following students:");
        Common.printCollection(students);

        Student s = new Student("Poe", "Edgar Allan", "2345678901");
        System.out.println("Poe present: " + students.contains(s));
    }
}
```

12_CollectionsAndCustomClasses/hr/fer/oop/collections_and_customclasses/example3/HashSetMain.java

Uporaba vlastitih klasa u Javinom okviru kolekcija: pravilo 2

- **Da bi se primjerci naših klasa mogli ispravno koristiti u Javinom okviru kolekcija, nužno je da klasa nadjača metodu *hashCode*() i time specificira način na koji se temeljem "sadržaja" generira sažetak**
- Pitanje "što se uzima u obzir pri računanju sažetka" je pitanje koje treba razriješiti tijekom modeliranja objekata domene
  - Važno je pri tome poštivati ugovor između metoda hashCode i equals: **ako equals kaže da su dva objekta jednaka, tada njihovi sažetci moraju biti jednaki**
- Uočiti da treba vrijediti:
  - ako su sažetci jednaki, objekti ne moraju biti
  - ako sažetci nisu jednaki, tada objekti sigurno nisu jednaki

Primjer 4. *HashSet* + *hashCode*, ali bez *equals*

- Napišimo metodu *hashCode()* kako je prikazano u nastavku
  - Posao izračuna sažetka delegiramo atributima

```
@Override
public int hashCode() {
  return this.studentID.hashCode();
}
```

12_CollectionsAndCustomClasses/hr/fer/oop/collections_and_customclasses/example4/Student.java

- U ovom primjeru klasa *Student* nema svoju metodu *equals*
- Što će sada biti rezultat ispisa metode *main* u *HashSetMain* ?

Primjer 5. *HashSet + hashCode + equals*

- Rezultat u primjeru 4 je i dalje biti *false*
- Jednom kada je *HashSet* pronašao slot u kojem bi objekt morao biti, pozivom metode equals(x) provjerava se je li on doista tamo
  - Više različitih objekata može biti preslikano u isti slot!
- **Stoga je nužno uz metodu *hashCode()* imati uparenu i metodu *equals(x)* pri čemu obje razmatraju identične atribute**
- Dodamo li odgovarajuću implementaciju metode *equals(x)*, kôd će konačno proraditi u skladu s očekivanim ponašanjem

  12_CollectionsAndCustomClasses/hr/fer/oop/collections_and_customclasses/example5/*.java

  - Razvojne okoline uobičajeno nude “čarobnjaka” za istovremeno stvaranje obje metode

Primjer 6. Pretraživanje kolekcije *TreeSet*

- Ponovimo prethodni primjer, ali sada kao implementaciju skupa odaberemo *TreeSet*
  - Klasa *Student* ima i *hashCode* i *equals*
- Što će biti rezultat ispisa metode *main* u *TreeSetMain* ?

```
public class TreeSetMain {
    public static void main(String[] args) {
        Set<Student> students = new TreeSet<>();
        Common.fillStudentsCollection(students, Student::new);
        System.out.println("I have following students:");
        Common.printCollection(students);

        Student s = new Student("Poe", "Edgar Allan", "2345678901");
        System.out.println("Poe present: " + students.contains(s));
    }
}
```

12_CollectionsAndCustomClasses/hr/fer/oop/collections_and_customclasses/example6/TreeSetMain.java

Pretraga kolekcije *TreeSet* zahtijeva usporedbe

- Rezultat je iznimka:

```
Exception in thread "main" java.lang.ClassCastException:
hr.fer.oop.collections_and_customclasses.example6.Student cannot be
cast to java.lang.Comparable
        at java.base/java.util.TreeMap.compare(TreeMap.java:1291)
        at java.base/java.util.TreeMap.put(TreeMap.java:536)
        at java.base/java.util.TreeSet.add(TreeSet.java:255)
        at
hr.fer.oop.collections_and_customclasses.Common.fillStudentsCollection(Common.ja
va:21)
        at
hr.fer.oop.collections_and_customclasses.example6.TreeSetMain.main(TreeSetMain.j
ava:14)
```

- Klasa *TreeSet* nekako mora moći uspoređivati objekte (u smislu veći, manji, jednak) kako bi izgradio stablo
  - stoga pretpostavlja da predani objekt implementira sučelje *Comparable* koje je osmišljeno upravo u tu svrhu i pokušava objekt ukalupiti
  - student ne implementira to sučelje pa ukalupljivanje pukne uz iznimku

Izvedba uspoređivanja

- Problem možemo riješiti na dva načina

1) Možemo definirati prirodan poredak studenata
   - klasa *Student* mora implementirati sučelje *Comparable* i metodu *compareTo*
   - time se definira način na koji se studenti uspoređuju
   - ovakav poredak nazivamo ***prirodan poredak*** *objekata* te vrste, odnosno kažemo da je definiran ***prirodni komparator***

2) Konstruktoru klase *TreeSet* možemo dati referencu na vanjski komparator
   - to je objekt čija klasa implementira sučelje *Comparator* te ima metodu *compare* koja prima reference na dva objekta, uspoređuje ih i vraća rezultat usporedbe

Sučelja *Comparable* i *Comparator*

- Sučelje *Comparable*

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

- Sučelje *Comparator*

```
public interface Comparator<T> {
    int compare(T o1, T o2);

    //te još niz defaultnih metoda od Jave 1.8
}
```

- Metoda *compareTo* uspoređuje trenutni objekt s predanim, a metoda *compare* uspoređuje prvi argument s drugim
  - ako je prvi (trenutni) manji od drugog (predanog), treba vratiti negativnu vrijednost (iznos nije bitan)
  - ako su jednaki, treba vratiti 0
  - ako je prvi veći od drugog, treba vratiti pozitivnu vrijednost (iznos nije bitan)

Primjer 7. Rješenje primjera 6 upotrebom sučelja *Comparable*

- Nadograđujemo klasu *Student* implementacijom sučelja *Comparable*

```
public class Student implements Comparable<Student> {
    ...

    @Override
    public int compareTo(Student other) {
        return this.studentID.compareTo(other.studentID);
    }
}
```

12_CollectionsAndCustomClasses/hr/fer/oop/collections_and_customclasses/example7/Student.java

- Ostatak koda ostaje nepromijenjen

Primjer 8. Rješenje primjera 6 upotrebom sučelja *Comparator*

- Novi komparator možemo stvoriti kao zasebnu klasu, anonimnu klasu ili korištenjem lambda izraza
  - Primjer sa zasebnom klasom

```
public class StudentComparator implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        return s1.getStudentID().compareTo(s2.getStudentID());
    }
}
```

12_CollectionsAndCustomClasses/hr/fer/oop/collections_and_customclasses/example8/*.java

- U glavnom programu *TreeSet* stvorimo predajući instancu komparatora

```
Set<Student> students = new TreeSet<>(new StudentComparator());
```

- Umjesto posebne klase, mogli smo koristiti lambda izraz

```
Set<Student> students = new TreeSet<>((s1, s2) ->
s1.getStudentID().compareTo(s2.getStudentID()));
```

Složenije usporedbe objekata

- U prethodnim primjerima usporedba se vršila samo po JMBAG-u
- Izmijenimo komparator na način da se usporedba vrši prvo po prezimenu, pa potom po imenu, pa ako je i to isto, onda po JMBAG-u?
  - Ovakav komparator omogućit će nam ispis studenata sortiranih po prezimenu ako iteriramo po *TreeSetu*

```
public class StudentComparator implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        int r = s1.getLastName().compareTo(s2.getLastName());
        if (r != 0) {
            return r;
        }
        r = s1.getFirstName().compareTo(s2.getFirstName());
        if (r != 0) {
            return r;
        }
        return s1.getStudentID().compareTo(s2.getStudentID());

    }
}
```

12_CollectionsAndCustomClasses/hr/fer/oop/collections_and_customclasses/example9/StudentComparator.java

Novi komparator kao dekorator postojećeg

- Što ako želimo usporedbu u suprotnom smjeru tako da iteriranje daje poredak studenata silazno, a ne uzlazno?
  - Možemo napisati novi (skoro isti) komparator
- Bolje rješenje → napisati novi (generički) komparator koji u konstruktoru prima referencu na postojeći komparator, pamti ga (omata ga) i u nadjačanoj metodi compare pita omotani komparator za usporedbu, ali vrati suprotnu vrijednost čime okreće poredak
- Ovaj način proširenja funkcionalnosti definiranjem nove klase koja nasljeđuje klasu ili sučelje čiju instancu prima u konstruktoru i pamti, a zatim u nadjačanim metodama koristi omotanu instancu uz dodatni kod za novu funkcionalnost poznat je u oblikovnim obrascima pod nazivom **dekorator**

Primjer dekoratora za usporedbu studenata

- Kôd je jednostavan i primijenjiv za bilo koji komparator, ne samo za komparator studenata, pa je napisani generički komparator

```
public class ReverseComparator<T> implements Comparator<T> {
    private Comparator<T> original;

    public ReverseComparator(Comparator<T> original) {
        this.original = original;
    }

    @Override
    public int compare(T o1, T o2) {
        int r = original.compare(o1, o2);
        return -r;
    }
}
```

12_CollectionsAndCustomClasses/hr/fer/oop/collections_and_customclasses/example9/ReverseComparator.java

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

Primjer upotrebe dekoriranog komparatora

- Stvorimo komparator za studente, a onda na osnovu toga stvorimo primjerak reverznog komparatora

```
public static void main(String[] args) {
  StudentComparator comparator = new StudentComparator();
  Comparator<Student> reverse = new ReverseComparator<>(comparator);
  Set<Student> students = new TreeSet<>(reverse);

  Common.fillStudentsCollection(students, Student::new);
  System.out.println("I have following students:");
  Common.printCollection(students);
}
```

12_CollectionsAndCustomClasses/hr/fer/oop/collections_and_customclasses/example9/ReverseComparator.java

- Studenti su sada ispisani sortirano silazno po prezimenu
  - obrnuto od onoga kako je navedeno u metodi *compare* u *hr.*.example9.StudentComparator*

```
I have following students:
(0123456789) Joe Rock
(2345678901) Edgar Allan Poe
(3456789012) Immanuel Kant
(5687461359) Joe Black
(1234567890) Joe Black
```

Ugrađeni reverzni komparatori

- Umjesto pisanja vlastitog reverznog komparatora mogu se koristiti već ugrađene metode u Javi
  - od Jave 8 sučelje *Comparator* sadrži *default* metodu koja vraća reverzni komparator postojećeg komparatora, pa se može pisati:

```
Comparator<Student> reverse = comparator.reversed();
```

  - klasa *Collections* nudi statičku metodu *reverseOrder* koja prima referencu na komparator i vraća reverzni komparator

```
Comparator<Student> reverse =
        Collections.reverseOrder(comparator);
```

  - ako klasa implementira sučelje *Comparable*, onda možemo koristiti *Collections.<T>reverseOrder()*

```
Comparator<Student> reverse = Collections.reverseOrder();
```

  - što vraća reverzni komparator komparatora definiranim kroz implementaciju sučelja *Comparable*

Višestruki kriteriji sortiranja

- Ponekad je u programima potrebno podržati sortiranje po više kriterija koje korisnik može podesiti tijekom izvođenja
- Razmotrimo klasu koji ima 4 atributa
  - sortiranje možemo napraviti na 4! načina ako gledamo samo redoslijed atributa koje ćemo razmatrati
  - ako uzmemo u obzir da po svakom atributu možemo još sortirati "prirodno" ili obrnutim poretkom, broj kombinacija se penje na 2^4*4!, što je 384
  - nema smisla pisati toliko različitih (gotovo identičnih) komparatora

Kako realizirati višestruku usporedbu po nepoznatim kriterijima?

- Kako realizirati višestruku usporedbu, ako u trenutku izrade klase nije poznati kriterij za usporedbu?
- 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

Konstruktori kompozitnog komparatora (1)

- Modeliramo kompozitni komparator da može primiti varijabilni broj komparatora istog tipa koje onda pohrani u vlastitu listu.

```
public class CompositeComparator<T> implements Comparator<T> {
  private List<Comparator<T>> comparators;

  @SafeVarargs
  public CompositeComparator(Comparator<T>... comparators) {
    this.comparators = new ArrayList<>(comparators.length);
    Collections.addAll(this.comparators, comparators);

    // or instead we can do this like
    //(Comparator<? super T> c : comparators) {
    //  comparators.add(c);
    //}
    }
...
}
```

12_CollectionsAndCustomClasses/hr/fer/oop/collections_and_customclasses/example10/CompositeComparator.java

Konstruktori kompozitnog komparatora (2)

- Modeliramo kompozitni komparator da može primiti listu bilo kojih komparatora istog tipa.

```
public class CompositeComparator<T> implements Comparator<T> {
...
  public CompositeComparator(List<Comparator<T>> comparators) {
    this.comparators = new ArrayList<>(comparators.size());
    this.comparators.addAll(comparators);
  }
...
}
```

12_CollectionsAndCustomClasses/hr/fer/oop/collections_and_customclasses/example10/CompositeComparator.java

  - Načelno, svugdje u klasi *CompositeComparator<T>* umjesto `Comparator<T>` moglo je pisati `Comparator<? super T>`
    - na taj način bi omogućili da se može predati i lista komparatora neke nadređene klase. Npr. da smo *CompositeComparator* parametrizirali po klasi *ForeignStudent* koji nasljeđuje klasu *Student*, onda bi valjan komparator u parametrima bio i *Comparator<Student>*, a ne samo *Comparator<ForeignStudent>*

Implementacija usporedbe u kompozitnom komparatoru

- Kompozitni komparator uzima jedan po jedan komparator iz liste i završava s usporedbom kad se pojavi prvi komparator po kojem elementi nisu isti ili kad iscrpi sve komparatore

```
public class CompositeComparator<T> implements Comparator<T> {
...
  @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;

  }
...
}
```

12_CollectionsAndCustomClasses/hr/fer/oop/collections_and_customclasses/example10/CompositeComparator.java

Primjer višestrukog sortiranja na klasi Student

- Neka klasa *Student* ima prirodni komparator (implementacija sučelja *Comparable* na način da uspoređuje JMBAG-ove) i tri statičke varijable koje predstavljaju primitivne komparatore po pojedinom atributu
  - Modelirani su kao lambda izrazi (kraće i jednostavnije u odnosu na zasebne klase)

```
public class Student implements Comparable<Student> {
...

 @Override
    public int compareTo(Student o) {
        return this.studentID.compareTo(o.studentID);
    }

    public static final Comparator<Student> BY_LAST_NAME = (s1, s2) ->
s1.lastName.compareTo(s2.lastName);
    //public static final Comparator<Student> BY_FIRST_NAME = (s1,s2) ->
s1.firstName.compareTo(s2.firstName);
    public static final Comparator<Student> BY_FIRST_NAME =
Comparator.comparing(Student::getFirstName);
    public static final Comparator<Student> BY_STUDENT_ID = (s1, s2) ->
s1.studentID.compareTo(s2.studentID);
}
```

12_CollectionsAndCustomClasses/hr/fer/oop/collections_and_customclasses/example10/Student.java

Primjer upotrebe kompozitnog komparatora

- Studente treba poredati silazno po imenu
  - Ako dva studenta imaju isto ime, treba ih poredati po prezimenu uzlazno
  - Ako dva studenta imaju isto ime i prezime, poredati po prirodnom komparatoru (u ovom slučaju to je usporedba JMBAG-a)
    - Prirodni komparator može se dobiti korištenjem statičke metode na sučelju *Comparator*

`Comparator.<Student>naturalOrder()`

```
public static void main(String[] args) {
  Comparator<Student> comparator = new CompositeComparator<>(
    Student.BY_FIRST_NAME.reversed(),
    Student.BY_LAST_NAME,
    Comparator.naturalOrder() //same as Comparator.<Student>naturalOrder()
  );
  Set<Student> students = new TreeSet<>(comparator);
...
}
```

12_CollectionsAndCustomClasses/hr/fer/oop/collections_and_customclasses/example10/Main.java

Ugrađene metode umjesto kompozitnog komparatora

- Umjesto pisanja kompozitnog operatora mogli smo koristiti *default* metodu *thenComparing* u sučelju *Comparator*
- Ista funkcionalnost s prethodnog slajda mogla se zapisati i ovako:

```
public static void main(String[] args) {
  Comparator<Student> comparator = Student.BY_FIRST_NAME.reversed()
   .thenComparing(Student.BY_LAST_NAME)
   .thenComparing(Comparator.naturalOrder());

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

12_CollectionsAndCustomClasses/hr/fer/oop/collections_and_customclasses/example10/Main.java