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/

Korištenje vlastitih klasa u kolekcijama

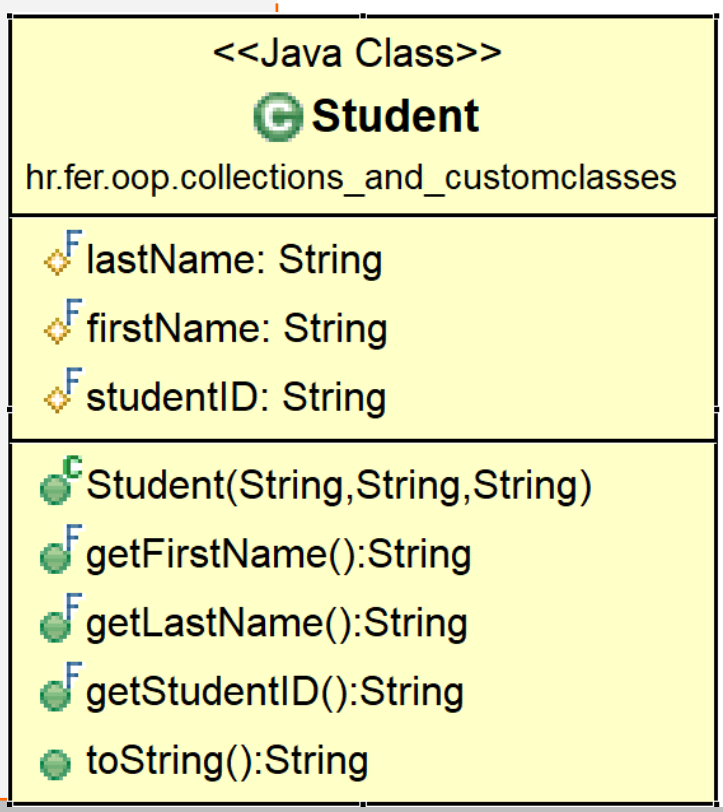
- U prethodnim primjerima ključevi u mapama te elementi lista i skupova bili su ugrađeni tipovi (String, Integer, …)
- U primjerima koji slijedi bit će prikazano što je potrebno da bi se u kolekcijama mogle koristiti vlastite klase
  - Primjeri započinju jednostavnom implementacijom klase Student koja se kroz primjeri proširuje nadjačavanje metoda equals, hashcode, implementacijom sučelja Comparable, …

Inicijalna verzija klase Student

12_.../hr/fer/oop/collections_and_customclasses/Student.java

- Inicijalna verzija sadrži konstruktor s 3 argumenta (prezime, ime te id studenta) i nadjačanu metodu *toString.* Naknadne verzije će imati i neke druge članove, ali umjesto pisanja klase ispočetka, u primjerima će klasa Student biti naslijeđena, varijable označene *protected final*, a *getteri* označeni s *public final*

```
public class Student {
...
   public Student(String lastName, String
        firstName, String studentID) {
               this.lastName = lastName;
               this.firstName = firstName;
               this.studentID = studentID;
        }
   @Override
   public String toString() {
        return String.format("(%s) %s %s",
           studentID, firstName, lastName);
   } ...
```



Zajedničke metode u primjerima – ispis kolekcije

- U primjerima se obavlja ispis skupa ili liste studenata pa je ispis premješten u parametriziranu statičku metodu klase *Common*
  - Primijetiti da je parametrizirana samo metoda *printCollection*, a ne i sama klasa *Common*
  - U kolekcijama će biti objekti klasa izvedenih iz klase Student, ali budući da se ispis svodi na iteriranje kroz kolekciju i poziv metode *toString()* što je primijenjivo na sve klase nije potrebno stavljati ograničenje *T extends Student*

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

12_CollectionsAndCustomClasses/…/collections_and_customclasses/Common.java

Zajedničke metode – punjenje kolekcije (1)

- U svakom od primjera kolekciju treba popuniti podacima, primjerice kodom nalik sljedećem

```
public static void fillStudentsCollection(Collection<Student> col) {
   Student s1 = new Student("Black", "Joe", "1234567890");
   Student s2 = new Student("Poe", "Edgar Allan", "2345678901");
   ...
   col.add(s1);
   col.add(s2);
...
```

- Problem s ovim pristupom je što primjeri koriste drugačiju verziju klase Student, preciznije primjer 2 će koristiti klasu *Student2* koja je naslijedila klasu *Student*, primjer X će koristi klasu *StudentX* i sl.
  - Posljedično konstruktor new Student nije dobar za primjer 2, new Student2 nije dobar za primjer x itd…, pa bi u svakom primjeru morali pisati new StudentX gdje je StudentX klasa specifična za taj primjer iako svi konstruktori imaju iste argumente i rade isto

Zajedničke metode – punjenje kolekcije (2)

- Želimo postići sljedeće

```
public static <S extends Student> void
                        fillStudentsCollection(Collection<S> col) {
   Student s1 = new S("Black", "Joe", "1234567890");
   Student s2 = new S("Poe", "Edgar Allan", "2345678901");
   ...
   col.add(s1);
   col.add(s2);
...
```

  gdje svaka potklasa klase Student ima konstruktor s 3 argumenta

- Ideja (u ovom obliku) nije provediva, jer (u Javi) nije moguće instancirati generički tip s *new*

Zajedničke metode – punjenje kolekcije (3)

- Rješenje je definirati funkcijsko sučelje s metodom čiji je smisao da kreira novi objekt (klase izvedene iz klase Student) temeljem tri argumenta.

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

- Metode koje kreiraju novi objekte kao alternative korištenju operatora *new* nazivaju se metode tvornice (engl. factory methods)
- Ovo sučelje može se implementirati lambda izrazom, npr.

```
StudentFactory<Student> factory =
       (last, first, id) -> new Student(last, first, id);
```

12_.../example1/ArrayListMain.java

Zajedničke metode – punjenje kolekcije (4)

- Definirano funkcijsko sučelje *StudentFactory*

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

može se implemtirati lambda izrazom,

```
StudentFactory<Student> factory =
       (last, first, id) -> new Student(last, first, id);
```

ali i referenciranjem odgovarajuće metode (tj. metode koja prima 3 stringa te vraća „studenta”

12_.../example1/ArrayListMain.java

- Na prvu izgleda kao da takva metoda ne postoji, ali upravo to će biti konstruktori koji se nalaze u primjerima

```
StudentFactory<Student> factory = Student::new;
```

Zajedničke metode – punjenje kolekcije (5)

- Koristeći prethodno definirano sučelje, sad je moguće napisati generičku varijantu metode za popunjavanje sadržaja kolekcije

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

```
public static <S extends Student> void fillStudentsCollection(
               Collection<S> col, StudentFactory<S> factory){
   S s1 = factory.create("Black", "Joe", "1234567890");
   S s2 = factory.create("Poe", "Edgar Allan", "2345678901");
   ...
   col.add(s1);
   col.add(s2);
...
```

12_.../Common.java

Primjer 1. Pretraživanje liste

- Tražimo element za kojeg mislimo da bi trebao biti u listi, ali on ne biva pronađen
  - Taj element nije onaj kojeg smo stavili u listu, već je samo identičnog sadržaja!

```
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<>();
  StudentFactory<Student> factory =
        (last, first, id) -> new Student(last, first, id);
  // StudentFactory<Student> factory = Student::new;
  Common.fillStudentsCollection(students, factory);
  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_.../collections_and_customclasses/example1/ArrayListMain.java

Kako radi pretraga u listi?

- Klase *ArrayList* (i *LinkedList*) implementira *contains*(x) na način da prolazi kroz sve elemente liste i nad svakim elementom *e* te 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 klasi *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, treba nadjačati metodu *equals(x)***
  - inače se koristi usporedba referenci
- 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 *identifikator*

```
public class Student2 extends Student {
    public Student2(String lastName, String firstName, String id) {
        super(lastName, firstName, id);
    }
    @Override
    public boolean equals(Object obj) {
        if(!(obj instanceof Student2)) return false;
        Student2 other = (Student2)obj;
        return this.studentID.equals(other.studentID);
    ...
```

12_…/collections_and_customclasses/example2/Student2.java

Pretraživanje liste - „popravljena verzija”

- ArrayList i dalje koristi *equals*, ali klasa *Student2* ima nadjačanu metodu equals, tako da uspoređuje studente po id-u

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

Poe present: true
```

12_... example2/ArrayListMain.java

```
List<Student2> students = new ArrayList<>();
Common.fillStudentsCollection(students, Student2::new);

System.out.println("I have following students:");
Common.printCollection(students);

Student2 s = new Student2("Poe", "Edgar Allan", "2345678901");
System.out.println("Poe present: " + students.contains(s));
```

Primjer 2. Pretraživanje kolekcije HashSet

- Klasa Student2 ima implementiran equals, ali se elementi umjesto u listu pospremaju u skup implementiran kao *HashSet*.
- Program ispisuje false
  - Razlog leži u načinu kako *HashSet* određuje gdje tražiti element
  - Elementi su smješteni po „pretincima”, a pretinac se određuje temeljem 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
```

12_... /example2/HashSetMain.java

```
Set<Student2> students = new HashSet<>();
Common.fillStudentsCollection(students, Student2::new);

System.out.println("I have following students:");
Common.printCollection(students);

Student2 s = new Student2("Poe", "Edgar Allan", "2345678901");
System.out.println("Poe present: " + students.contains(s));
```

Uporaba vlastitih klasa u Javinom okviru kolekcija: pravilo 2

- **Da bi se primjerci vlastitih 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
- *hashCode* treba implementirati tako da ravnomjerno raspoređuje elemente po pretincima

Primjer 3. – *HashSet* + *hashCode*, ali bez *equals*

- *Student3* ima *hashCode()*, ali nema equals
  - Posao izračuna sažetka svedemo na sažetak id-a
  - Pretinac je ispravno određen, ali on može imati više elemenata, a za pretragu unutar pretinca koristi se *equals*

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

Poe present: false
```

12_... example3/HashSetMain.java

```
Set<Student3> students = new HashSet<>();
Common.fillStudentsCollection(students, Student3::new);

System.out.println("I have following students:");
Common.printCollection(students);

Student3 s = new Student3("Poe", "Edgar Allan", "2345678901");
System.out.println("Poe present: " + students.contains(s));
```

Pravilo 3. Nadjačati i *equals* i *hashcode*

- **Uz metodu *hashCode()* nužno je imati uparenu i metodu *equals(x)* pri čemu obje razmatraju identične atribute**

```
public class Student4 extends Student {
   public Student4(String lastName, String firstName, String id) {
       super(lastName, firstName, id);
   }
   @Override
   public boolean equals(Object obj) {
       if(!(obj instanceof Student4)) return false;
       Student4 other = (Student4)obj;
       return this.studentID.equals(other.studentID);
   }
   @Override
   public int hashCode() {
       return this.studentID.hashCode();
   ...
```

12_... example4/Student4.java

- Razvojne okoline omogućavaju brzo generiranje obje metode

Java *records*

- U primjerima klase Student* imaju nepromjenjive varijable, odgovarajuće *gettere*, i nadjačane *equals, hashcode* i *toString()*
- Od Jave 14 za takve situacije može se koristiti posebna, ograničena varijanta klase koja se naziva *record*
  - *Record* je klasa koja se ne može naslijediti i s nepromjenjivim atributima za koju prevodilac automatski generira članske varijable i istoimene gettere (bez prefiksa get) te konstruktor koji ih inicijalizira

```
record Student(String lastName, String firstName, String studentID) {
  ... ostalo prema potrebi
}
```

12_... records/Student.java

  - *equals* se svodi na usporedbu svih članova, *toString* vraća string koji u uglatim zagradama navodi vrijednosti varijabli, a *hashcode* ima implementaciju kompatibilnu s equals.
  - Više na https://docs.oracle.com/en/java/javase/14/language/records.html i https://www.baeldung.com/java-record-keyword

Primjer 4. 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<Student4> students = new TreeSet<>();
        Common.fillStudentsCollection(students, Student4::new);
        System.out.println("I have following students:");
        Common.printCollection(students);

        Student4 s = new Student4("Poe", "Edgar Allan", "2345678901");
        System.out.println("Poe present: " + students.contains(s));
    }
}
```

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

Pretraga kolekcije *TreeSet* zahtijeva usporedbe

- Rezultat je iznimka:

```
Exception in thread "main" java.lang.ClassCastException:
hr.fer.oop.collections_and_customclasses.example4.Student4 cannot be
cast to java.lang.Comparable
        at java.base/java.util.TreeMap.compare(TreeMap.java:1291)
        ...
        at
hr.fer.oop.collections_and_customclasses.Common.fillStudentsCollection
(Common.java:20)
        at
hr.fer.oop.collections_and_customclasses.example4.TreeSetMain.main(Tre
eSetMain.java:13)
```

- 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

Dva način izvedbe uspoređivanja

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***
   - Prirodni komparator je konzistentan s *equals* ako i samo ako *e1.compareTo(e2) == 0* vraća istu logičku vrijednost kao i *e1.equals(e2)* za svaki *e1 i e2* klase čiji je to prirodni komparator.
2) Konstruktoru klase *TreeSet* možemo dati referencu na vanjski komparator
   - 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 (bilo koju) negativnu vrijednost
  - ako su jednaki, treba vratiti 0
  - ako je prvi veći od drugog, treba vratiti (bilo koju) pozitivnu vrijednost

Primjer 5. Implementacija prirodnog poretka

- Nadograđujemo klasu *Student* implementacijom sučelja *Comparable* tako da prirodni poredak bude konzistentan s metodom *equals*

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

```
public class Student5 extends Student4 implements
Comparable<Student5> {
    ...

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

- Ostatak koda ostaje nepromijenjen

Primjer 6. Pisanje vlastitog komparatora

- Komparator nastaje implementacijom sučelja *Comparator što se može napraviti* zasebnom ili anonimnom klasom 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<Student4> students = new TreeSet<>(new StudentComparator());
```

- Umjesto posebne klase, mogli smo koristiti lambda izraz

```
Set<Student4> students = new TreeSet<>((s1, s2) ->
s1.getStudentID().compareTo(s2.getStudentID()));
```

Složenije usporedbe objekata

- U prethodnim primjerima usporedba se vršila samo po JMBAG-u. Ovaj komparator usporedbu vrši prvo po prezimenu, pa potom po imenu, pa ako je i to isto, onda po id-u?
  - Omogućit će ispis studenata sortiranih po prezimenu pri iteriranju po *TreeSetu*

12_CollectionsAndCustomClasses/…/example7/StudentComparator.java

```
public class StudentComparator implements Comparator<Student4> {
    @Override
    public int compare(Student4 s1, Student4 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());

    }
}
```

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/example7/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<Student4> reverse = new ReverseComparator<>(comparator);
  Set<Student4> students = new TreeSet<>(reverse);

  Common.fillStudentsCollection(students, Student4::new);
  System.out.println("I have following students:");
  Common.printCollection(students);
}
```

12_CollectionsAndCustomClasses/hr/fer/oop/collections_and_customclasses/example7/Main.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/example8/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/…/collections_and_customclasses/example8/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/example8/CompositeComparator.java

Primjer višestrukog sortiranja na klasi Student

- Osim prirodnog komparatora (iz klase *Student5* implementacijom sučelja *Comparable* na način da uspoređuje identifikatore) mogu se definirati i pojedinačni komparatori
  - 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 Student8 extends Student5 {
   ...
   public static final Comparator<Student8> BY_LAST_NAME =
              (s1,s2) -> s1.lastName.compareTo(s2.lastName);
   public static final Comparator<Student8> BY_FIRST_NAME =
               Comparator.comparing(Student8::getFirstName);
   public static final Comparator<Student8> BY_STUDENT_ID =
              (s1,s2) -> s1.studentID.compareTo(s2.studentID);

}
```

12_...example8/Student8.java

Primjer 8. upotreba 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 identifikatora)
    - Prirodni komparator može se dobiti korištenjem statičke metode na sučelju *Comparator*

`Comparator.<Student8>naturalOrder()`

```
Comparator<Student8> comparator = new CompositeComparator<>(
        Student8.BY_FIRST_NAME.reversed(),
        Student8.BY_LAST_NAME,
        Comparator.naturalOrder()
                        //same as Comparator.<Student8>naturalOrder()
);
Set<Student8> students = new TreeSet<>(comparator);
...
```

12_...example8/Main.java

Primjer 8. Ugrađeni kompozitni komparatori

- Umjesto pisanja kompozitnog operatora može se koristiti *default* metoda *thenComparing* iz sučelja *Comparator*
- Ista funkcionalnost s prethodnog slajda mogla se zapisati i ovako:

```
Comparator<Student> comparator =
                        Student.BY_FIRST_NAME
                       .reversed()
                       .thenComparing(Student.BY_LAST_NAME)
                       .thenComparing(Comparator.naturalOrder());
Set<Student8> students = new TreeSet<>(comparator);
...
```

12_...example8/Main.java