9 Nasledjivanje

Kreirajmo dve klase za rad sa osobama i učenicima. Rećićemo da svaka osoba ima:

- Ime
- Prezime
- Broj godina
  a da svaki učenik ima:
- Ime
- Prezime
- Broj godina
- Listu ocena

Osoba

```
1  package nasledjivanje;
2
3  public class Osoba {
4
5
6      private String ime, prezime;
7      private int brojGodina;
8
9      public Osoba(String ime, String prezime, int
   brojGodina) {
10         this.ime = ime;
11         this.prezime = prezime;
12         this.brojGodina = brojGodina;
13     }
14
15     public String getIme() {
```

```
16          return ime;
17      }
18
19      public String getPrezime() {
20          return prezime;
21      }
22
23      public int getBrojGodina() {
24          return brojGodina;
25      }
26
27      public void setIme(String ime) {
28          this.ime = ime;
29      }
30
31      public void setPrezime(String prezime) {
32          this.prezime = prezime;
33      }
34
35      public void setBrojGodina(int brojGodina) {
36          this.brojGodina = brojGodina;
37      }
38
39      @Override
40      public String toString() {
41          return ime + " " + prezime + " (" + brojGodina +
    ")";
42      }
43  }
```

Ucenik

```
1  package nasledjivanje;
2
3
4  import java.util.ArrayList;
```

```
5
6  public class Ucenik {
7
8      private String ime, prezime;
9      private int brojGodina;
10     private ArrayList<Integer> ocene;
11
12     public Ucenik(String ime, String prezime, int
   brojGodina, ArrayList<Integer> ocene) {
13         this.ime = ime;
14         this.prezime = prezime;
15         this.brojGodina = brojGodina;
16         this.ocene = new ArrayList<>();
17         this.ocene.addAll(ocene);
18     }
19
20     public String getIme() {
21         return ime;
22     }
23
24     public String getPrezime() {
25         return prezime;
26     }
27
28     public int getBrojGodina() {
29         return brojGodina;
30     }
31
32     public ArrayList<Integer> getOcene() {
33         return ocene;
34     }
35
36     public void setIme(String ime) {
37         this.ime = ime;
38     }
39
```

```
40      public void setPrezime(String prezime) {
41          this.prezime = prezime;
42      }
43
44      public void setBrojGodina(int brojGodina) {
45          this.brojGodina = brojGodina;
46      }
47
48      public void setOcene(ArrayList<Integer> ocene) {
49          this.ocene = ocene;
50      }
51
52      @Override
53      public String toString() {
54          StringBuilder sb = new StringBuilder();
55          sb.append(ime).append(" ").append(prezime)
56          .append(" (").append(brojGodina).append(")\n");
57          sb.append(ocene.toString()).append("\n");
58
59          return sb.toString();
60      }
61  }
```

Ako malo bolje pogledamo naš kôd, ceo deo koji se tiče polja `ime`, `prezime` i `brojGodina` u klasi `Ucenik` ima identično ponašanje kao što je definisano u klasi `Covek`. Čak i `toString()` metod serijalizuje na isti način te podatke.

Kao što smo već navikli, ovakvo redundantno prepisivanje kôda želimo da izbegnemo što je više moguće. Na drugi pogled, vidimo da naša klasa `Ucenik` samo *proširuje* klasu `Covek` sa jednim dodatim poljem, i možemo reći da je `Ucenik` `Covek` sa nekim dodatim atributima. Ovakvo ponašanje kalsa, kada jedna *proširuje* drugu, nazivamo **nasledjivanjem**.

Nasledjivanje

Definicija 9.1 (Nasledjivanje)**.**

Za klasu B kažemo da **nasledjuje**, odnosno **proširuje** klasu A, u oznaci $B \subseteq A$[1], kada klasa B sadrži sve karakteristike klase A.
Za klasu B tada kažemo da je **podklasa** klase A, tj. klasa A je **nadklasa** klase B.

Jasnu definicju **nasledjivanja** nije jednostavno iskazati, te ćemo sada na konkretnijim primerima proći tačno šta podrazumevamo pod `sve karakteristike klase A`.

U Objektno orijentisanom programiranju razlikujemo dve vrste nasledjivanja:

- Prototipno nasledjivanje
- Klasno nasledjivanje
  Kako Java realizuje nasljedjivanje korišćenjem principa klasnog nasledjivanja, to je ona vrsta na koju ćemo se fokusirati u ovom poglavlju.

Ukoliko želimo da naznačimo da jedna klasa nasledjuje drugu, koristimo ključnu reč `extends`, po šablonu:

```
1  <modifikator_klase> class <ImePodklase> extends
   <ImeNadklase> {
2      //Telo podklase
3  }
```

Time naglašavamo Javi da naša podklasa sadrži čitavu definiciju nadklase, sa potencijalno dodatnim poljima i metodama.

Ovim putem, kažemo da naša podklasa **jeste** nadklasa, tj. u navedenom primeru `Osoba` - `Ucenik`, kažemo da `Ucenik` **jeste** jedna `Osoba`, tačnije, da su objekti klase `Ucenik` takodje objekti klase `Osoba`. Primetimo da ikao se radi o klasnom nasledjivanju, jasnu primenu možemo uočiti tek na kreiranim objektima, odnosno instancama naših klasa.

Upravo zbog ovog razloga, kako je svaka instanca podklase ujedno i instanca nadklase, prilikom kreacije objekta podklase, potrebno je prvo kreirati objekat nadklase, koji se *proširuje* da obuhvati dodatne pojmove podklase. Da bi ovo postigli, potrebno je da pristupimo `konstruktoru` naše nadklase i to činimo pomoću specijalnog pokazivača `super`.

Definicija 9.2 (Ključna reč super)**.**

Ključna reč `super` je pokazivač na (direktnu) nadklasu klase odakle se poziva.

Potpuno prirodno, ako pokazivač `this` označava objekat klase o kojoj se radi, tako pokazivač `super` označava objekat nadklase. Pomoću tog pokazivača, moguće je pristupiti svim elementima (atributima ili metodama) nadklase koje su dostupne podklasi (u zavisnosti od modifikatora koji stoje uz te atribute odnosno metode). Pa tako specijalno, možemo pristupiti i konstruktoru nadklase, i u tom slučaju ne navodimo ime klase.

Pošto podklasa *nasledjuje* sve pojmove svoje nadklase, to nema potrebe za ponovnim definisanjem tih pojmova (polja ili metoda) već je dovoljno da u podklasi navedemo samo elemente koji nisu sadržani

Pre nego što nastavimo i spetljamo se sa teoretskim pojmovima, rekreirajmo naše klase `Osoba` i `Ucenik` od početka lekcije, tako da `Ucenik` nasledjuje klasu `Osoba` i rezimirajmo sve što smo iskazali do sada na tim praktičnim klasama.

Osoba

```
1  package nasledjivanje;
2
3  public class Osoba {
4
5
6      private String ime, prezime;
7      private int brojGodina;
8
9      public Osoba(String ime, String prezime, int
   brojGodina) {
10         this.ime = ime;
11         this.prezime = prezime;
12         this.brojGodina = brojGodina;
13     }
14
15     public String getIme() {
16         return ime;
17     }
18
19     public String getPrezime() {
20         return prezime;
21     }
22
23     public int getBrojGodina() {
24         return brojGodina;
25     }
26
27     public void setIme(String ime) {
28         this.ime = ime;
```

```
    }

    public void setPrezime(String prezime) {
        this.prezime = prezime;
    }

    public void setBrojGodina(int brojGodina) {
        this.brojGodina = brojGodina;
    }

    public void kaziStaRadis() {
        System.out.println("Ja bivstvujem");
    }

    @Override
    public String toString() {
        return ime + " " + prezime + " (" + brojGodina + ")";
    }
}
```

Ucenik

```
package nasledjivanje;


import java.util.ArrayList;

public class Ucenik extends Covek{

    //Nema potrebe za ponovnim definisanjem ime, prezime i brojGodina
    private ArrayList<Integer> ocene;

    public Ucenik(String ime, String prezime, int brojGodina, ArrayList<Integer> ocene) {

```

```
12          super(ime, prezime, brojGodina); //Pristupamo
    konstruktoru nadklase
13          this.ocene = new ArrayList<>();
14          this.ocene.addAll(ocene);
15      }
16
17
18      public ArrayList<Integer> getOcene() {
19          return ocene;
20      }
21
22
23      public void setOcene(ArrayList<Integer> ocene) {
24          this.ocene = ocene;
25      }
26
27
28      @Override
29      public String toString() {
30          StringBuilder sb = new StringBuilder();
31          sb.append(super.toString()).append("\n");
32          sb.append(ocene.toString());
33
34          return sb.toString();
35      }
36  }
```

Prodjimo korak po korak šta se gore desilo u klasi `Ucenik`:

1. Sa ključnom rečju `extends` naveli smo da klasa `Ucenik` nasledjuje klasu `Covek`:
   Sada možemo uočiti zašto baš pojmove *nasledjivanja* i *proširivanja* možemo da proizvoljno zamenjujemo. Sa jedne strane, naša klasa `Ucenik` proširuje definiciju klase `Covek` uvodjenjem novog atributa `ArrayList<Integer> ocene`, a sa

druge strane, naša klasa `Ucenik` nasledjuje polja `ime`, `prezime` i `brojGodina` kao i metode klase `Covek` iz definicije klase `Covek`. S toga, nema nikave potrebe za ponovnim definisanjem tih polja i metoda.
Primetimo da metoda `kaziStaRadis` nije metoda koja je strogo vezana za polja klase (nije ni getter ni setter), ali je sa `public` modifikatorom i kao takva, biće dostupna svakom objektu klase `Ucenik`, posto svi objekti klase `Ucenik` su ujedno i objekti klase `Covek`!

2. U konstruktoru sa ključnom rečju `super` pristupamo konstruktoru nadklase `Covek`:
   Već smo rekli da pre nego što kreiramo objekat klase `Ucenik` moramo da kreiramo objekat klase `Covek`, pošto je svaki `Ucenik` ujedno i `Covek`. Ta kreacija objekta nadklase se vrši pozadinski i u našem konstruktoru klase `Ucenik`, pre bilo čega **moramo** da pozovemo konstruktor nadklase! U suprotnom, dobijamo kompajlersku grešku koja nam govori da pozivanje konstruktora nadklase mora biti prva komanda konstruktora podklase! To ima i semantičkog smisla - nema poente kerirati atribute objekta neke klase pre nego što se kreiraju svi neophodni atributi definisani u nadklasi.
3. Nema potrebe ponovno definisanje getter-a i setter-a za polja nadklase:
   Kako su svi getter-i i setter-i u klasi `Covek` javni, tj. `public`, to su oni dostupni i podklasi, pa će biti dostupni i objektima podklase, te ponovno redefinisanje tih metoda bi bilo nepotrebno. Jedino što je potrebno je da definišemo getter-e i setter-e polja koja se tiču samo podklase, što je u ovom slučaju jedno polje `ArrayList<Integer> ocene`.
4. U serijalizaciji pozivamo serijalizaciju nadklase:
   Analogno objasnjenju 3; `toString()` metod nadklase je javan i

dostupan podklasi, te je dovoljno pozvati njega u metodi `toString()` podklase i dodati odgovarajuće stvari (u ovom slučaju samo serijalizaciju polja `ocene`). Ovoj metodi pristupamo preko pokazivača na objekat naše nadklase, tj. preko `super.` sintagme.

Testirajmo naše klase:

Test-Osoba-Ucenik

```
1  package nasledjivanje;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class test {
7
8      public static void main(String[] args) {
9          Osoba o = new Osoba("A", "B", 35);
10         System.out.println(o);
11         o.kaziStaRadis();
12
13         Ucenik u = new Ucenik("X", "Y", 15, new
   ArrayList<>(List.of(4, 5, 5)));
14         System.out.println(u);
15         u.kaziStaRadis();
16         System.out.println("Ucenik u ima "  +
   u.getBrojGodina() + " godina");
17     }
18 }
```

```
1  OUTPUT:
2  A B (35)
3  Ja bivstvujem
4  X Y (15)
```

```
[4, 5, 5]
Ja bivstvujem
Ucenik u ima 15 godina
```

Objekat `o` klase `Osoba` ima pristup metodama:

- `getIme()`
- `getPrezime()`
- `getBrojGodina()`
- `setIme(String ime)`
- `setPrezime(String prezime)`
- `setBrojGodina(int brojGodina)`
- `kaziStaRadis()`
- `toString()` - koja ispisuje ime, prezime i broj godina

dok objekat `u` klase `Ucenik` ima pristup metodama:

- `getIme()` - nasledjeno od nadklase
- `getPrezime()` - nasledjeno od nadklase
- `getBrojGodina()` - nasledjeno od nadklase
- `getOcene()`
- `setIme(String ime)` - nasledjeno od nadklase
- `setPrezime(String prezime)` - nasledjeno od nadklase
- `setBrojGodina(int brojGodina)` - nasledjeno od nadklase
- `setOcene(ArrayList<Integer> ocene)`
- `kaziStaRadis()` - nasledjeno od nadklase
- `toString()` - koja ispisuje ime, prezime, broj godina i ocene

Naravno, objekat `o` nema nikakvu ideju o postajanju metoda `getOcene` i `setOcene` jer `o` nije instanca klase `Ucenik`!

Napomenimo još jednom, da kako je svaki objekat klase `Ucenik` u jedno i objekat klase `Osoba`, to preko objekta `u` možemo pristupiti

javnim metodama definisanim u klasi `Osoba`, pa se linija koda 16 izvršava bez ikakvog problema.

Sada, kada smo uveli pojam nasledjivanja, ima smisla i govoriti o **hijerarhiji kôda**. Pa tako, za klasu `Osoba` kažemo da je viša u hijerarhiji od klase `Ucenik`, tj. da je klasa `Ucenik` niža u hijerarhiji od klase `Osoba`.
Takodje, kažemo da je klasa `Osoba` *roditelj* klase `Ucenik`, tj. da je klasa `Ucenik` *dete* klase `Osoba`.

Završimo ovu sekciju povratkom na malo teoretskije razmatranje nasledjivanja.
Naime, nema nikavkog smisla reci da neka klasa A nasledjuje klasu B i da klasa B nasledjuje klasu A. Time dobijamo neko ciklično definisanje što strogo treba izbegavati! Matematički gledano, u tom slučaju imamo $B \subseteq A \wedge A \subseteq B$, što implicira $A = B$, tj. da su A i B iste klase. Baš zbog ove restrikcije, naše klase možemo da posmatramo kao *aciklične usmerene grafove*, gde svaki čvor grafa predstavlja jednu klasu, a usmerene strelice označavaju nasledjivanje, tako da čvor na koji pokazuje neka strelica označava nadklasu klase koju označava čvor odakle potiče strelica. Alternativno, na naš projekat možemo da posmatramo kao na *(strogo) parcijalno uredjenje*, bilo algebarski ili skupovno.
Pored toga, ako važi $B \subseteq A$ i klasa B nema nikakvih dodatnih polja i metoda (dakle ima samo odgovarajuće konstruktore koji pozivaju nadkonstruktore), tada klasa B ima samo semantičku ulogu u našem kodu.[2] Ukoliko, klasa B ipak ima neka dodatna svojstva, tada kažemo da je klasa B *prava* podklasa klase A, u oznaci $B \subset A$.

Anotacije

Vratimo se na kod koji se ticao `Osoba` i `Ucenika`. Primetimo da klasa `Ucenik` nasledjuje metod `kaziStaRadis()` koja ispisuje poruku "Ja bivstvujem". Iako je ovo tačno za svaku osobu, pa i za svakog učenika, učenik može strožije, odnosno može da kaže "Ja učim". Pre nego što objasnimo kako možemo da **nadjačamo** metode iz nadklase, definišimo prvo šta su **anotacije**.

Definicija 9.3 (Anotacije)**.**

Poda anotacijom podrazumevamo meta-podatke koje bolje opisiju kôd, ali nisu sastavni deo njega.
Anotacije se označavju simbolom `@` pre svog identifikatora.

Anotacije ne utiču dirketno na izvršavanje koda, i njihovo pisanje je potpuno opciono. Uloga anotacija je da:

- olakšaju kompajleru[3] da detektuje greške i upozorenja.
- lakše upravljanje kôda i generisanje automatski generisanog koda.
- upravljanju pokretljivog kôda.

Kao što smo rekli, anotacije se označavaju simbolom `@` i mi nećemo ulaziti u detalje pisanja anotacija ovde. Iznećemo jedan primer i završiti odeljak sa kratkom pričom o `@Override` anotacijom.

Posmatrajmo funkciju:

```
1  public void ispisi(String s) {
2      System.out.println(s);
3  }
```

Ukoliko se ovoj funkciji prosledi `null` vrednost za argument prilikom njego poziva, Java će odštampati `"null"` u konzoli. Radi jednostavnosti, samo fiktivno zamislimo da ova funkcija radi još neke stvari sa prosledjenom niskom i da je jako važno da prosledjena niska ne bude `null`. Tada, možemo anotirati parametar sa anotacijom `@NotNull`:

```
1  public void ispisi(@NotNull String s) {
2      System.out.println(s);
3  }
```

Tada prilikom poziva funkcije `ispisi` sa `null` vrednošću kao argumentom, Java ce izbaciti `IllegalArgumentException` izuzetak i kôd će pasti (exit code 1). Iako je bilo moguće kompajlirati i pokrenuti ovaj kôd, na ovaj način smo nalgasili korisniku naše funkcije `ispisi` da prilikom prosledjivanja niske mora da zagarantuje da ta niska nije nulna.

Nemojte se zavarati da je ovo pravi način a se izbegne pucanje kôda. Izbegavanje pucanja kôda se vrši pomoću obradjivanja grešaka i izuzetaka, što se tiče lekcije o greškama i izuzecima.

Nama od najvećeg značaja će biti anotacija `@Override` koja naglašava da anotirana metoda **nadjačava** neko metodu neke svoje nadklase.

Nadjačavanje metoda

Definicija 9.4 (Nadjačavanje metoda)**.**

Nadjačavanje metoda (eng. 'Method overriding') pretstavlja postupak redefinisanja nasledjenih metoda.

Bitna razlika izmedju *preopterećivanja* metoda i *nadjačavanja* metoda je ta što preopterećivanje predstavlja proces definisanja više različitih implementacija jedne metode (sa različitim potpisima te metode), dok najdačavanje predstavlja ponovno definisanje iste metode (sada sa *identičnim* potpisom te metode).

Iako nije nužno potrebno, nadjačavanje metode anotiramo tako što ponovnu definiciju metoda anotiramo sa `@Override` anotacijom. Ova anotacija prikazuje kompajlersku grešku ukoliko ne postoji metoda koja treba da se nadjača u nekoj roditeljskoj klasi date klase odakle se nadjačava.
Jasno, nema nikakvog smisla nadjačavati metode unutar klase, pa tako:

- Ako jednu metodu ispisujemo više puta u istoj klasi -> radi se o *preotperećivanju* metode.
- Ako jednu metodu ispisujemo ponovo u nekoj podklasi -> radi se o *nadjačavanju* metode.

Nadjačavnje metoda nam omogućava da redefinišemo i promenimo ponašanje jedne metode nadklase u podklasama.

Pogledajmo ovo na praktičnom primeru. Kreirajmo dodatnu klasu `Nastavnik` koja proširuje klasu `Osoba` sa poljem `ArrayList<Ucenik>`. Potrebno je takodje najdačati metodu `kaziStaRadis()` tako da se ispisuje:

- "Ja predajem"; za objekte klase `Nastavnik`
- "Ja ucim"; za objekte klase `Ucenik`

Ucenik-Preopterecivanje

```
1  package nasledjivanje;
2
```

```java

import java.util.ArrayList;

public class Ucenik extends Covek {

    //Nema potrebe za ponovnim definisanjem ime, prezime i brojGodina
    private ArrayList<Integer> ocene;

    public Ucenik(String ime, String prezime, int brojGodina, ArrayList<Integer> ocene) {
        super(ime, prezime, brojGodina); //Pristupamo konstruktoru nadklase
        this.ocene = new ArrayList<>();
        this.ocene.addAll(ocene);
    }


    public ArrayList<Integer> getOcene() {
        return ocene;
    }


    public void setOcene(ArrayList<Integer> ocene) {
        this.ocene = ocene;
    }

    @Override
    public void kaziStaRadis() {
        System.out.println("Ja ucim");
    }


    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
```

```
36          sb.append(super.toString()).append("\n");
37          sb.append(ocene.toString());
38
39          return sb.toString();
40      }
41  }
```

Nastavnik

```
1   package nasledjivanje;
2
3   import java.util.ArrayList;
4
5   public class Nastavnik extends Osoba {
6       private ArrayList<Ucenik> ucenici;
7
8       public Nastavnik(String ime, String prezime, int
    brojGodina, ArrayList<Ucenik> ucenici) {
9           super(ime, prezime, brojGodina);
10          this.ucenici = new ArrayList<>();
11          this.ucenici.addAll(ucenici);
12      }
13
14      public ArrayList<Ucenik> getUcenici() {
15          return ucenici;
16      }
17
18      public void setUcenici(ArrayList<Ucenik> ucenici) {
19          this.ucenici = ucenici;
20      }
21
22      @Override
23      public void kaziStaRadis() {
24          System.out.println("Ja predajem");
25      }
26
```

```
27      @Override
28      public String toString() {
29          StringBuilder sb = new StringBuilder();
30          sb.append(super.toString()).append("\n");
31          sb.append("Ucenici:\n");
32          for (Ucenik u : ucenici)
33              sb.append(u.toString()).append("\n");
34          return sb.toString();
35      }
36  }
```

Testirajmo naše klase:

Test-Osoba-Ucenik-Nastavnik

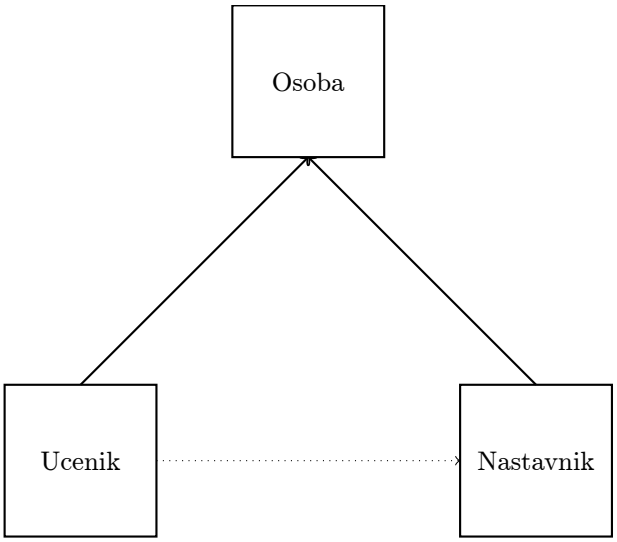
```
1   package nasledjivanje;
2
3   import java.util.ArrayList;
4   import java.util.List;
5
6   public class test {
7
8       public static void main(String[] args) {
9           Osoba o = new Osoba("A", "B", 35);
10          System.out.println(o);
11          o.kaziStaRadis();
12
13          Ucenik u = new Ucenik("X", "Y", 15, new
    ArrayList<>(List.of(4, 5, 5)));
14          System.out.println(u);
15          u.kaziStaRadis();
16
17          Nastavnik n = new Nastavnik("Kosta", "Vujic", 50,
    new ArrayList<>(List.of(
18              new Ucenik("Mihajlo", "Petrovic", 17, new
    ArrayList<>())
```

```
        ));
        System.out.println(n);
        n.kaziStaRadis();
    }
}
```

```
OUTPUT:
A B (35)
Ja bivstvujem
X Y (15)
[4, 5, 5]
Ja ucim
Kosta Vujic (50)
Ucenici:
Mihajlo Petrovic (17)
[]
Ja predajem
```

Sada, prilikom poziva metode `kaziStaRadis`, objekti `u` i `n` pozivaju svoju implementaciju metode `kaziStaRadis`, koja nadjačava implementaciju te metode u roditeljskoj klasi `Osoba`. Ovakvo ponašanje smo već videli, upravo kod pisanja `toString()` metode, koja i jeste bila anotirana od samog početka sa `@Override` anotacijom.

O UML dijagramima[4] ćemo detaljnije pričati u dodatnom odseku nakon lekcije o interfejsima, a za sada prikazujemo grubu sliku hijerarhije našeg koda:



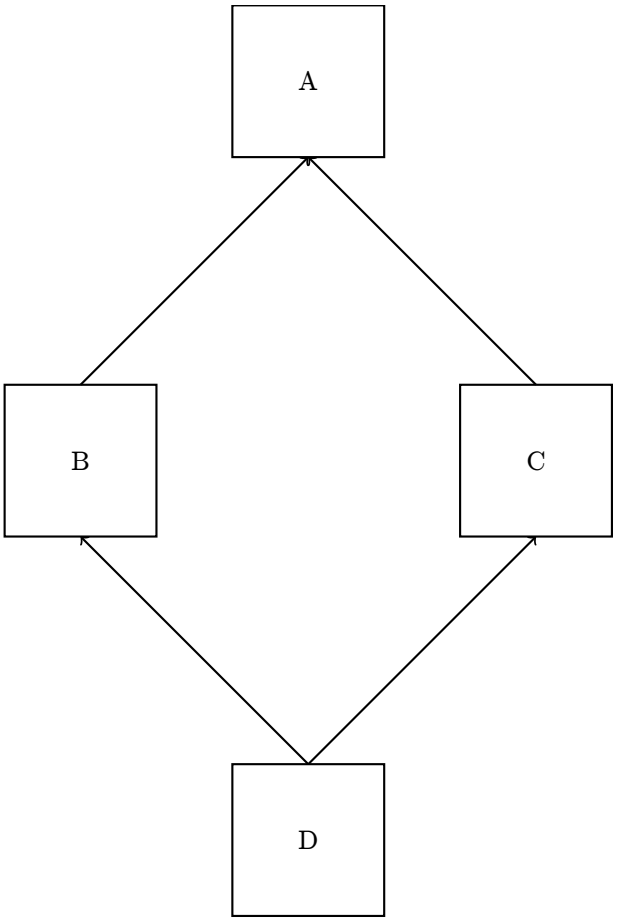
Klase su predstavljene kvadratima, a puna strelica označava roditelj->dete relaciju. Isprekidana strelica označava da jedna klasa koristi u svom kodu objekte druge, kao što klasa `Nastavnik` koristi kalsu `Ucenik` kao elemente liste jednog svog polja. Naravno, kao i u ERD-u, ove stralice mogu da se bolje naznače, ali više o tome u poglavlju koje se tiče UML dijagramima.

Iz ovog dijagrama se jasno vidi da **jednu klasu mogu da nasledjuju više drugih klasa**. Ovakvo ponašanje je i očekivano. Klasu `Osoba` smo proširili na dva različita načina. Ovakvo ponašanje je dostupno u svim objektno orijentisanim programskim jezicima koji baziraju nalsedjivanje na principu klasa. Ono što treba zapamtiti posebno za programski jezik Java[5] je sledeća aksioma:

Aksioma 9.1 (Zabrana dijamantske strukture)**.**

U Javi, svaka klasa može da nasledi **najviše jednu** drugu klasu.

Dakle, nešto poput:



nije dozvoljeno, jer u tom slučaju klasa `D` bi nasledjivala i klasu `B` i klasu `C`, tj. nasledjivala više od jedne klase!
Rezime:

- Jednu klasu mogu više različitih klasa da nasledjuju. (klasu `A` nasledjuju klase `B` i `C`).
- Jedna klasa može najviše jednu drugu klasu da nasledi. (klasa `D` ne može da nasledi i klasu `B` i klasu `C`).

O zaobilasku ove restrikcije kreiranja dijamantske strukture ćemo pričati u poglavlju koji se bavi interfejsima.

Ukoliko malo bolje obratimo pažnju, vidimo da smo i u klasi `Nastavnik`, pored koje ne stoji ključna reč `extends`, naglašavali da metodu `toString()` nadjačavamo koristeći anotaciju `@Override`, a rekli smo da ako ne postoji takva metoda u nadklasi (što i ne postoji ako pretpostavimo da `Nastavnik` ne nasledjuje) da dolazi do kompajlerske greške, sa kojom se mi do sada nismo susretali.

Odgovor na ovu falaciju je prost.

Aksioma 9.2 (Nasledjivanje klase Object)**.**

U Javi, svaka klasa, osim klase `Object`, nasledjuje klasu `Object`, bez eksplicitnog navodjenja, ukoliko se eksplicitno ne navede da nasledjuje neku drugu klasu.
Klasa `Object` ne nasledjuje ni jednu drugu klasu.

Pa tako, klasa `Object` definiše metodu `toString()` kao metodu koja ispisuje `klassName` praćenu simbolom `@` paćen heš kodom memorijske lokacije objekta, koju mi nadjačavamo, prvo u klasi `Osoba`, a zatim tu implementaciju ponovo nadjačavamo u klasama `Ucenik` i `Nastavnik`.

Ova aksioma ima zanimljive posledice:

Posledica 9.1.

Ukoliko neka klasa eksplicitno ne nasledjuje neku drugu klasu, tj. ako ne koristi ključnu reč `extends` u svom potpisu, onda ona implicitno nasledjuje klasu `Object`.

Tako, mi možemo (ali ne moramo) da napišemo potpis naše `Osoba` klase kao:

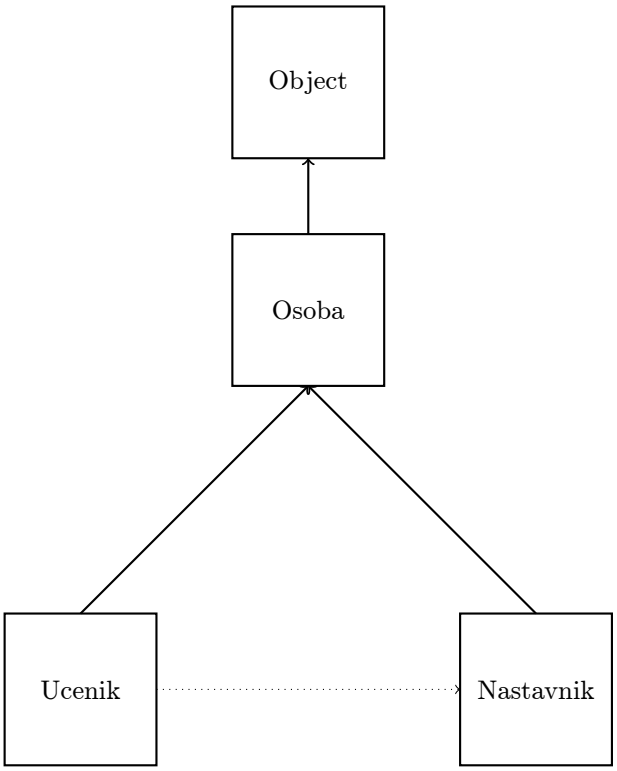
```
1  public class Osoba extends Object {
2      //....
3  }
```

Posledica 9.2.

Klasa `Object` je nadklasa svih drugih klasa.

Dokaz ovoga je prost. Neka klasa može ili da eksplicitno nasledjuje neku klasu ili ne. Ako eksplicitno ne nasledjuje nijednu klasu, to znači da implicitno nasledjuje klasu `Object`. Ako eksplicitno nasledjuje neku durgu klasu, tada možemo da gledamo šta sve ta nadklasa nasledjuje i ponavljamo ovaj postupak. Zbog acikličnosti nasledjivanja, ovaj postupak kad tad mora da se završi, te ćemo ili doći do klase `Object` ili do neke klase koja implicitno nasledjuje klasu `Object` $Q.E.D.$[6]

Dakle, "prava" slika našeg dosadašnjeg koda bi izgledala:



Finalne klase

Ukoliko želimo da onemogućimo korisniku naše klase da kreira klasu koja će je proširiti, to možemo da uradimo tako što ćemo

našu klasu proglasiti finalnom, sa ključnom rečju `final`, po šablonu:

```
<modifikatorKlase> final class <imeKlase> {
    // Telo klase
}
```

Ovo ponašanje ne bi trebalo da nas buni. Naime, proširivanje klasa predstavlja proces dopunjavanja te klase dodatnim pojmovima, pa tako ako je klasa `final`, nema ni smisla pričati u naknadnom dopunjavanju te klase.
Ukoliko nam to konstruktor dozvoljava, mi ćemo idalje moći da kreiramo objekte tih klasa. Pored toga, ta klasa može da nasledi neku drugu (nefinalnu) klasu, jer čak i ako ne navedemo eksplicitno da nasledjuje neku drugu, to samo znači da implicitno nasledjuje `Object` klasu.

Operator instanceof

Ukoliko se prisetimo priče na početku o svim operatorima koje programski jezik Java poseduje, videćemo da smo napomenilu da postoji takozvani `instanceof` operator, čija uloga je da vraća odgovor na pitanje da li je neki objekat instanca neke klase ili ne.

Definicija 9.5 (Operator instanceof)**.**

Operator `instanceof` koji poredi objekat sa klasom i vraća logički literal u zavisnosti od toga da li je taj objekat instance te klase.

Šablon pisanja ovog operatora je sledeći:

```
<referencaObjekat> instanceOf <ImeKlase>;
```

U ovom momentu bitno je da se setimo da prilikom kreacije objekta podklase, mi kreiramo u pozadini prvo objekat nadklase, što za posledicu ima da referenca nekog objekta je instanci svoje klase, kao i **svake** svoje nadklase:

Test-Instanceof

```
1  package nasledjivanje;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class test {
7
8      public static void main(String[] args) {
9          Osoba o = new Osoba("A", "B", 35);
10         Ucenik u = new Ucenik("X", "Y", 15, new
   ArrayList<>(List.of(4, 5, 5)));
11         Nastavnik n = new Nastavnik("Kosta", "Vujic", 50,
   new ArrayList<>(List.of(
12             new Ucenik("Mihajlo", "Petrovic", 17, new
   ArrayList<>())
13         ));
14
15         boolean daLiJeUInstancaUcenika = u instanceof
   Ucenik;
16         System.out.println(daLiJeUInstancaUcenika);
17
18         System.out.println(
19             u instanceof Osoba
20         );
21
22         System.out.println(
23             u instanceof Object
24         );
25
```

```
26          System.out.println(
27              n instanceof Ucenik
28          );
29
30          System.out.println(
31              o instanceof Osoba
32          );
33
34          System.out.println(
35              o instanceof Object
36          );
37
38      }
39  }
```

```
1  OUTPUT:
2  true
3  true
4  true
5  false
6  true
7  true
```

Objašnjenje:

- Prvi `true` se odnosi na upit `u instanceof Ucenik`, tj. na upit da li je objekat `u` instanca klase (objekat klase) `Ucenik`, što očigledno jeste.
- Drugi `true` se odnosi na `u instanceof Osoba`. Kako smo rekli da prilikom kreacije objekta podklase kreiramo objekat nadklase, to je `u` ujedno i `Osoba`, što i ima smisla - svaki učenik **jeste** osoba.
- Treći `true` se odnosi na `u instanceof Object`. Kako `Ucenik` nasledjuje `Osoba`, a `Osoba` nasledjuje `Object`, to je i `u`

instanca klase `Object`. O ovakvim lancima ćemo više pričati u sledećoj glavi. Štaviše, `... instanceof Object` će uvek vratiti `true`, jer je klasa `Object` nadklasa *svih* klasa.

- Prvi `false` se odnosi na upit `n instanceof Ucenik`. Iako objekat `n` klase `Nastavnik` koristi objekte klase `Ucenik`, ona ne nasledjuje klasu `Ucenik` te `n` nije instanca klase `Ucenik` (o ovome više u odeljku nasledjivanje i kompozicija).
- Četvrti `true` ispis se odnosi na `o oinstanceof Osoba`, što trivijalno važi.
- Naravno, o je i instanca `Object` klase, jer `Osoba` implicitno nasledjuje tu klasu, pa `o instanceof Object` vraća `true` vrednost.

Lanac nasledjivanja

Iako ne direktno, već smo nagovestili da možemo kreirati "lanac" nasledjivanja. Zapravo, kreiranjem bilo koje klase, mi kreiramo najkraći lanac nasledjivanja od 2 člana, jer ta klasa sigurno nasledjuje klasu `Object`. Tako na primer realizovanjem klasa za rad sa geometrijskim objektima, kreiramo dugačak lanac nasledjivanja:

Zadatak 9.1 (Geometrija)**.**

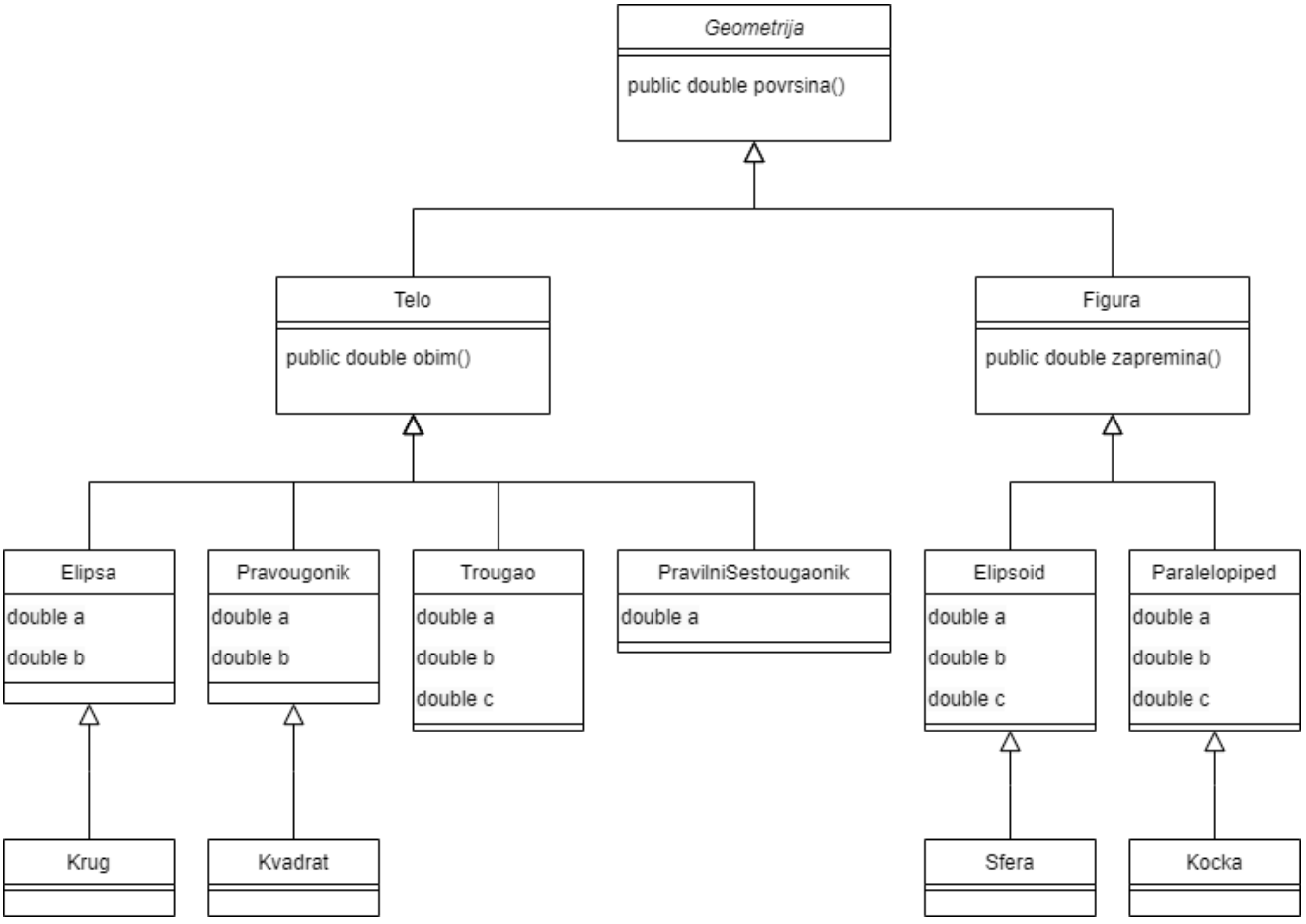
Napisati program za rad sa geometrijskim objektima. Kreirati:

1. Klasu `Geometrija` koja nema atribute i ima jedan `public double povrsina()` metod koji vraća broj `-1`. Kreirati jedan prazan konstruktor.
2. Klasu `Figura` koja nasledjuje klasu `Geometrija` i ima jedan `public double obim()` metod koji vraća broj `-1`. Kreirati jedan prazan konstruktor.

3. Klasu `Telo` koje nasledjuje klasu `Geometrija` i ima jedan `public double zapremina()` metod koji vraća broj `-1`. Kreirati jedan prazan konstruktor.
4. Klasu `Elipsa` koja nasledjuje klasu `Figura` i kao polja ima dva realna broja koja označavaju dva poluprečnika elipse. Kreirati jedan konstruktor koji prima dva realna broja. Nadjačati metodu `povrsina()`.[7]
5. Klasu `Kurg` koja nasledjuje klasu `Elipsa`. Kreirati jedan konstruktor koji prima jedan realan broj. Nadjačati metodu `obim()`.
6. Klasu `Trougao` koja nasledjuje klasu `Figura` i kao polja ima tri realna broja koja označavaju stranice trougla. Kreirati jedan konstruktor koji prima tri realna broja. Nadjačati metode `povrsina()` [8] i `obim()`.
7. Klasu `Pravougaonik` koja nasledjuje klasu `Figura` i kao polja ima dva realna broja koja označavaju dužinu stranica. Kreirati konstruktor koji prima dva realna broja. Nadjačati metode `povrsina()` i `obim()`.
8. Klasu `Kvadrat` koja nasledjuje klasu `Pravougaonik`. Kreirati jedan konstruktor koji prima jedan realan broj.
9. Klasu `PravilniSestougaonik` koja nasledjuje klasu `Figura` i kao polja ima jedan realani broj koji označava dužinu stranice pravilnog šestougaonika. Kreirati jedan konstruktor koji prima jedan realan broj. Nadjačati metode `obim()` i `povrsina()`.
10. Klasu `Elipsoid` koja nasledjuje klasu `Telo` i kao polja ima tri realna broja koja označavaju tri poluprečnika elipsoida. Kreirati jedan konstruktor koji prima tri realna broja. Nadjačati metodu `zapremina()`.[9]
11. Klasu `Sfera` koja nasledjuje klasu `Elipsoid`. Kreirati jedan konstruktor koji prima jedan realan broj. Nadjačati metodu `povrsina()`.

12. Klasu `Paralelopiped` koja nasledjuje klasu `Telo` i kao polja ima tri realna broja koja označavaju tri stranice paralelopipeda. Kreirati jedan konstruktor koji prima tri realna broja. Nadjačati metode `povrsina()` i `zapremina()`. Pretpostaviti da se radi o pravom paralelopipedu.
13. Klasu `Kocka` koja nasledjuje klasu `Paralelopiped`. Kreirati jedan konstruktor koji prima jedan realan broj.

UML dijagram ovih klasa bi zigledao:



Nasledjivanje i Kompozicija

Polimorfizam

---

*Varijanse

1. Ovde se $B \subseteq A$ interpretira ne kao da skup B ima manje elemenata od skupa A (jer će u praksi imati više elemenata), već kao da je pojam B strožiji od pojma A. Npr. možemo reći da je skup sisara podskup skupa životinja. Iako definicija skupa sisara podrazumeva više pojmova od definicije skupa životinja, svaki sisar jeste jedna životnja, ali nije svaka životinja sisar. ↩
2. Ovo se često radi upravo iz tih nekih razloga. Na primer, možemo kreirati klasu `Kvadrat` koja proširuje klasu `Pravougaonik` tako što definiše samo jedan konstruktor koji prima jedan realan broj a u telu konstruktora poziva nadkonstruktor koji postavlja obe stranice na prosledjenu vrednost. Iako nisamo ništa dodali na definiciju `Pravougaonik`-a, ovime smo omogućili korisnicima našeg programa da kreiraju objekte "tipa" `Kvadrat` umesto da su primorani da kreiraju uvek objekte klase `Pravougaonik`. ↩
3. Pogotovo delu kompajlera koji se zove 'linker', čija uloga je upravo korektno praćenje hijerarhije kôda. ↩
4. UML - Unified Modelling Language. Dijagrami koji se koriste za grafički prikaz relacija klasa jednog programa/paketa. Vrlo slični ERD-ovima kod baza podataka. ERD - Entety Relationship Diagram. ↩
5. Što recimo nije slučaj za C# ↩
6. Quod Erat Demonstrandum - ono što je i trebalo da se pokaže. ↩
7. Računanje obima elipse nije analitički izvodljivo. ↩
8. Videti Heronov obrazac. ↩
9. Računanje površine elipsoida nije analitički izvodljivo. ↩