9. NADOGRADNJA-NASLIJEĐIVANJE KLASA

Tema ovog poglavlja je način kreiranja novih klasa na način da kao osnovu uzmemo postojeću klasu i dodamo novu funkcionalnost. Taj postupak nazivamo naslijeđivanje ili nadogradnja (extending). Nasljeđivanje je jedna od ključnih osobina objektno orijentiranih programa.

SADRŽAJ

---

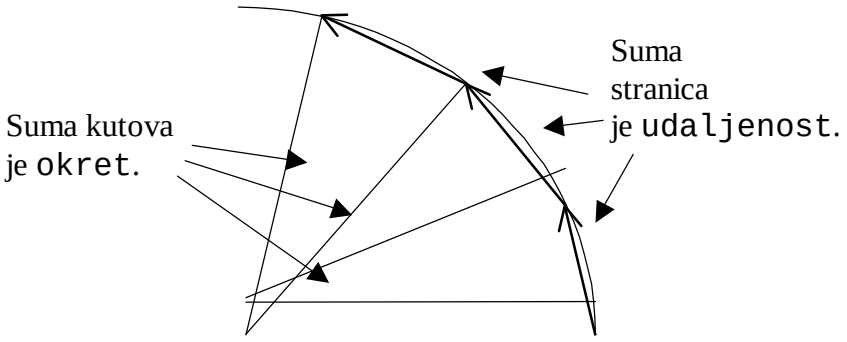
1. Nadogradnja klase (KrivuljaRobot)

Vraćamo se klasi Robot. Želimo joj dodati mogućnost kretanja(crtanja) krivulja. Robot, kako smo ga dosad isprogramirali, može se kretati samo u pravocrtnim koracima. Međutim ako ga isprogramiramo da radi vrlo mali korak i svaki put se malo okrene trebao bi crtati nešto nalik krivuljama.
Želimo nadograditi (*extend*) klasu Robot dodavanjem nove metode instance.

```
r.krivuljaLijevo(udaljenost,okret)
```

Pomakni robota `t` naprijed za udaljenost `udaljenost`, okrećući ka ulijevo kako se kreće za kut okreta `okret` . Robot crta kako se okreće samo ako je olovka spuštena.

To je ono što bi trebao rezultat kretanja. U stvari naredbe u tijelu metode će se robot micati u malim jednakim pravocrtnim koracima svaki put okrenuvši se za mali jednaki kut. Slijedeći dijagram pokazuje kretnju u za samo tri koraka



Napisat ćemo i sličnu metodu, `krivuljaDesno`.

Metoda `krivuljaLijevo` upotrebljavat će konstantu `MAX`. Ona će biti maksimalna udaljenost koju robot može preći prije nego što napravi novi korak.

Kompletan kod koji će pomicati robota po krivulji sadržava još malo geometrijskih proračuna na kojima se nećemo zadržavati. Kod izgleda ovako:

```
int num = (int) Math.ceil(udaljenost/MAX);
double kut = okret/num;
double korak = udaljenost/num;

r.lijevo(kut/2);
for (int i = 0; i < num; i++)
{  r.pomakni(korak);
   r.lijevo(kut);
}
r.desno(kut/2);
```

`Math.ceil(5.7)` = 6.0, `Math.ceil(5.0)` = 5.0. (vraćena vrijednost je opet double !)

Parametar `udaljenost` mora biti veći od nule.

Sad nakon geometrije dolazi ključna stvar.
Definiramo novu klasu nazvanu KrivuljaRobot. Klasa krivulja Robot imat će potpunu funkcionalnost klase Robot plus mogućnost crtanja krivulja. Dodatna funkcionalnost ostvarit će se dodavanje dviju metoda `krivuljaLijevo` i `krivuljaDesno` te statičke varijable MAX.

Sve to nećemo učiniti na način da u klasu Robot dopišemo definicije metoda i varijable. Učinit ćemo to postupkom nasljeđivanja tj. nadograđivanja. U tom slučaju čak nam nije ni potreban izvorni kod klase Robot (što je potrebno ?).

PRIMJER 1 (Definicija klase KrivuljaRobot.)

```
/* KrivuljaRobot je Robot koji se može
   micati po zakrivljenoj putanji. */

public class KrivuljaRobot extends Robot
{

    private final static int MAX = 2;

   /* Pomakni robota za iznos 'udaljenost',
      i okreni za iznos 'okret' nalijevo.
   */
   public void krivuljaLijevo
```

```
         (double udaljenost, double okret)
      {  if (udaljenost == 0)
         {  lijevo(okret);
            return;
         }
         int num;  double kut, korak;
         if (udaljenost > 0)
         {  num = (int) Math.ceil(udaljenost/MAX);
            kut = okret/num;
            korak = udaljenost/num;
         }
         else
         {  num = (int) Math.ceil(-udaljenost/MAX);
            kut = -okret/num;
            korak = udaljenost/num;
         }
         lijevo(0.5*kut);
         for (int i = 0; i < num; i++)
         {  pomakni(korak);
            lijevo(kut);
         }
         desno(0.5*kut);
      }

      /* Pomakni robota za iznos 'udaljenost',
         i okreni za iznos 'okret' nadesno.
      */
      public void krivuljaDesno
         (double udaljenost, double okret)
      {  krivuljaLijevo(udaljenost, -okret);
      }
   }
```

To je kompletna definicija klase KrivuljaRobot . jednostavan izraz u zaglavlju

```
extends Robot
```

kaže Javi da uključi sve definicije koje se pojavljuju u originalnoj definiciji klase Robot. To znači da će objekt tipa KrivuljaRobot imati ista polja `x`, `y`, `smjer`, `jeDolje` kao i objekt Robot ; imat će iste metode `pomakni`, `lijevo`, `desno`, `olovkaGore` i `olovkaDolje` kao objekt Robot , plus što će imati dvije dodatne metode `krivuljaLijevo` i `krivuljaDesno`. Osim toga kad god bude se koristila klasa KrivuljaRobot postojat će statičke varijable `window` i `MAX`.

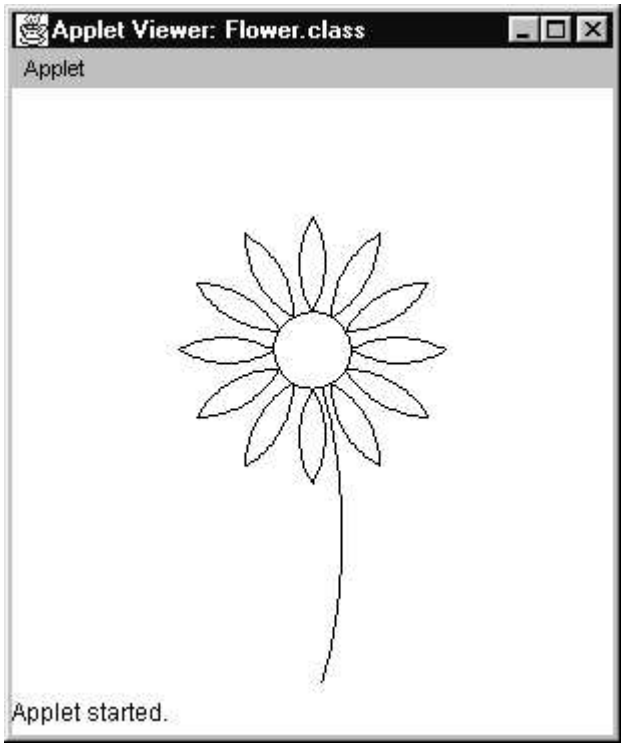
Slijedi applet koji koristi klasu tipa KrivuljaRobot.

PRIMJER 1 - nastavak (Upotreba klase KrivuljaRobot.)

```
public class Flower extends Applet

{  public void init()
   {  Robot.setWindow(this);
   }
```

```
    public void paint(Graphics g)
    {  KrivuljaRobot r = new KrivuljaRobot();
       r.desno(90);
       r.penDown();
       for (int i = 0; i < 12; i++)
       {  r.krivuljaLijevo(10,30);
          r.desno(60);
          r.krivuljaDesno(50,60);
          r.desno(120);
          r.krivuljaDesno(50,60);
          r.desno(60);
       }
       r.krivuljaLijevo(5,15);
       r.desno(90);
       r.krivuljaDesno(150,30);
    }
}
```

Slijedi crtež nastao izvođenjem appleta:



Primijetite da je korišten slijedeći konstruktor:

```
new KrivuljaRobot();
```

Međutim on se ne pojavljuje u definiciji klase KrivuljaRobot !
Ono što se ovdje događa je da nam Java sama osigurava konstruktor. Dakle, ako za neku klasu ne definirate konstruktor Java će vam sama osigurati konstruktor koji nazivamo **default** konstruktor. Taj konstruktor ne radi ništa osim što će u slučaju da se radi o klasi koja

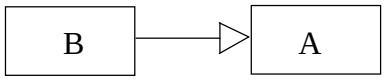
nasljeđuje neku drugu klasu, pozvati konstruktor roditeljske klase. Ne bilo koji konstruktor roditeljske klase, već konstruktor bez argumenata !
U klasi Robot postoji konstruktor bez argumenata i tak konstruktor postavlja robota u centar ekrana s pogledom na istok.

Pojmovi !

Općenito, pretpostavimo da imamo klasu A i pretpostavimo da želimo definirati novu klasu B čiji objekti trebaju imati sve članove objekata klase A te još neke dodatne članove. Tada klasu B definiramo na slijedeći način:

```
public class B extends A

{  Dodatna polja, statičke varijable,
   metode instance, statičke metode,
   i konstruktori.
}
```

Za klasu B kažemo da nadograđuje (**extend)** klasu A. B se naziva **subklasa(subclass)** klase A. A se naziva **superklasa (superclass)** klase B. Ova relacija se ponekad označava slijedećim dijagramom:



Objekti tipa klase B imat će sve članove (varijable, metode i konstruktore) dane u definiciji klase A. Kažemo da klasa B **nasljeđuje (inherit).** Također objekti klase B sadržavat će sve članove dane u definiciji klase B.

Osim što na ovaj način možemo dodavati nove metode u objekt B, možemo dati i potpuno nove definicije metoda koji su već definirani u klasi A. Kada to učinimo kažemo da je nova metoda **pregazila** (**override**) metodu iz superklase.

Isto se može dogoditi i s poljima. Ako u klasi B definiramo polje s istim nazivom kao u klasi A Java će u klasi B vidjeti novodefinirano polje, dok će polje superklase A biti skriveno (hide). Međutim postoji način pristupa skrivenom polju.

U stvari objekt tipa klase B ima dvojnu osobnost. Možete ga koristiti striktno kao objekt tipa B. Međutim kako sadrži potpunu funkcionalnost objekta tipa klase A možete ga koristiti i kao specijalan slučaj objekta klase A.
Java će omogućiti da se kod napisan za objekte tipa klase A koristi na objektima izvedene klase B.

Npr. napišete:

```
Robot r = new KrivuljaRobot();
```

nakon ovog koda nećete moći napisati naredbu `r.krivuljaLijevo(10,30)` jer Java prevodilac neće prihvatiti asociranje metode `krivuljaLijevo` s objektom tipa Robot. Kako objekti B imaju sve atribute objekata A i kako Java omogućava tretiranje objekata klase B kao da su klase A, možemo reći da objekti B stvarno pripadaju klasi A.

---

2. Zaštićeni pristup

Pravilo koje ste dosada uočili je da svaki član klase koji je deklariran kao `private` skriven od drugih klasa. To pravilo vrijedi i za nadgradnju klasa.
Pretpostavimo da klasa B nadograđuje klasu A. U tom slučaju programer koji piše klasu B bit će efektivno korisnik klase A koji treba znati samo javno sučelje (public interface) klase A. Čak ne mora imati izvorni kod klase A.

U tom slučaju subklasa je ograničena u pristupu poljima superklase.

Npr. u prethodnom poglavlju smo klasi Robot dodali novu funkcionalnost dodavanjem metode gledaGA. Pitanje je dali smo to mogli napraviti nadgradnjom klase, odnosno pisanjem nove klase koja bi naslijedila klasu Robot. Odgovor je ne !
Zašto? Razlog je u tome bi tada metoda nove klase `gledaGA` trebala izvoditi proračun smjera gledanja robota koristeći se poljima koordinata `x` i `y` koji su u klasi Robot označena kao `private`.
Dakle metode subklase nemaju pristup članovima superklase koji su označeni kao private !

Kako je ta restrikcija dosta ozbiljna Java dozvoljava kompromis. Postoji mogućnost da se članovima klase da stupanj pristupa između public i private. Mogu biti deklarirani kao **protected.**

To znači da im se može pristupiti *iz bilo koje klase koja nasljeđuje originalnu klasu*, ali ne i iz drugih klasa.

Npr. u originalnoj definiciji klase Robot (bez metode gledaGA) možemo promijeniti slijedeće deklaracije kako je napisano:

```
public class Robot

{  protected double x,y;

   protected double smjer;

   protected boolean jeDolje;

   …
```

(Ostatak klase ostaje nepromijenjen) . Tada su mogućnosti nadogradnje klase mnogo veće. Npr. možemo definirati novu klasu koja nasljeđuje klasu Robot te dodaje metodu gledaGA:

PRIMJER 2

```
/* GledoRobot objekt je Robot s mogućnošću
    gledanja u drugi Robot
```

```
*/

public class GledoRobot extends Robot

{  /* Podesi smjer robota r da gleda ovaj Robot */
   public void gledaGA(Robot r)
   {  double dX = x - r.x;
      double dY = r.y - y;
      double rad = Math.atan2(dY,dX);
      r.dir = 180*rad/Math.PI;
   }
}
```

Primijetite da se u izrazima u tijelu metode `gledaGA` može pristupati poljima koja su u klasi Robot označena kao protected. e

Ovakav način dopuštanja pristupa nije baš u skladu s filozofijom Jave. U Javi je običaj da su sva polja označena s private. Ako postoji potreba pristupa tim poljima iz bilo koje druge klase preporuča se pisanje public metoda kojima će se to omogućiti. Te metode nazivamo metode pristupa (accessor) i metode promjene (mutator).

Tako u klasi Robot možemo napisati dvije metode pristupa:

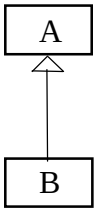
```
public double getX() { return x;}

public double getY() { return y;}
```

To možemo učiniti i za polja `smjer` i `jeDolje`.
U slučaju da smo napisali samo metode pristupa (ne i promjene) omogućili smo svakome tko ovu klasu koristi da dobije vrijednosti polja x,y,smjer i jeDolje, ali ne i promjenu vrijednosti istih.

---

3. Kombiniranje dviju srodnih klasa. (Student i Zaposlenik)

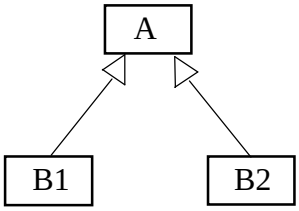
U prijašnjem odjeljku imali smo slijedeću situaciju. Počeli smo s definicijom klase A te zatim odlučili da je nadogradimo s definicijom klase B . To je predstavljeno sa slijedećim dijagramom.



Ovaj odjeljak opisuje drugu situaciju koja se može riješiti upotrebom nadogradnje klasa.

Recimo da imate dvije različite klase objekata B1 i B2 te da možete identificirati određene osobine su zajedničke objektima klase B1 i klase B2. Stoga bi bilo poželjno da se npr. programski kod koji je zajednički za obje klase ne piše za obje klase posebno. Isto vrijedi i za polja koja su zajednička za obje klase.

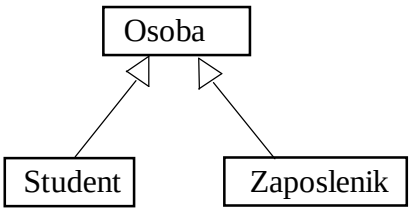
U toj situaciji koristi se mogućnost da se zajednički elementi klasa B1 i B2 izdvoje u superklasu A.



Dakle u klasi A možemo definirati članove koji su zajednički klasama B1 i B2 te ih poslije koristiti u klasama B1 i B2.

Pokušat ćemo definirati klase koje bi se mogle koristiti u jednostavnom programu baze podataka ljudi pripadaju fakultetu. Jedna klasa bi definirala studente koji studiraju na fakultetu, a druga bi definirala zaposlenike fakulteta. Nazvat ćemo ih Student i Zaposlenik.

Također ćemo definirati klasu Osoba u kojoj ćemo definirati zajedničke elemente klasa Student i Zaposlenik. Osoba će biti superklasa klasa Student i Zaposlenik.



Objekt Student ima slijedeća polja:

1. `String ime`
2. `String idBroj`
3. `String programStudija`
4. `int godina`
5. `Zaposlenik tutor`

Primijetite polje `tutor` . To polje sadrži referencu na objekt tipa Zaposlenik .

Polja za objekt `Zaposlenik` su slijedeća:
1. `String ime`

2. `String brojSobe`
3. `String brojTelefona`

Postoji samo jedno polje koje Zaposlenik i Student objekti imaju zajedničko: polje `ime`.

Stoga će objekt Osoba imati samo polje `ime`. To će polje klase Student i Zaposlenik naslijediti iz klase Osoba.

Slijedi lista metoda koja bi trebalo asocirati s objektom tipa Student.

1. `getIme()`
2. `setProgramStudija(d)`
3. `povecajGodinuStudija()`
4. `prikaz()`

Slijedi lista metoda za objekt tipa Zaposlenik.

1. `getIme()`
2. `promjenaUreda(s,t)`
3 `prikaz()`

Oba objekta imaju zajedničke metode `getIme` . Ta metoda bi u oba slučaja trebala imati istu funkcionalnost pa ćemo je definirati u superklasi Osoba.

Oba objekta imaju i zajedničku metodu `prikaz` . Međutim iako imaju sličnu zadaću njihova je funkcionalnost različita jer bi trebale prikazivati različiti skup podataka.

Ovo predstavlja problem jer kako oba objekta imaju zajedničku metodu prikaz bilo bi zgodno da je moguće u kodu pisati slijedeće:

```
Osoba m = neki izraz koji označava ili objekt tipa Zaposlenik ili objekt tipa Student
m.prikaz();
```

i prepustiti Javi da odredi koju metodu prikaz da koristi. Java *interpreter* će to upravo učiniti. Prvo će provjeriti da o kojem tipu objekta se radi. Ako je tip objekta Zaposlenik izvršit će metodu prikaz koja je dio definicije klase Zaposlenik. Ako je tip objekta Student onda će izvršiti metodu prikaz koja je dio definicije klase Student.
(Ova sposobnost intepretera naziva se **dinamičko povezivanje** (**dynamic binding)**)

Na nesreću *prevodilac(compiler)* nije toliko pametan. On će izraz `m.prikaz()` pokušati interpretirati u okviru klase Osoba i tražit će definiciju metode `prikaz` u definiciji klase Osoba. Postoje dva načina kako prevodilac prihvati naš kod.

1. Možemo dodati metodu `prikaz` klasi Osoba . Bit će to metoda koja ne čini ništa (prazno tijelo):

   ```
   public void prikaz() { }
   ```

   Ili će prikazivati ono što za osobu može prikazati, a to je ime osobe:

   ```
   public void prikaz {
   ```

```
        System.out.println(”Ime: ” + ime);
    }
```

2. Možemo klasi Osoba dodati **apstraktnu (abstract)** metodu. Ova metoda služi samo tome da prevodiocu kaže će tek subklasa ove klase definirati metodu `prikaz`.

   To pišemo na slijedeći način:

```
abstract public void prikaz();
```

Ako u definiciji klase postoji jedna ili više apstraktnih metoda, ključnu riječ `abstract` treba dodati i u zaglavlje klase. To indicira da klasa nije potpuno definirana i da neće biti moguće kreirati objekte od te klase. Međutim bit će moguće kreirati objekte koji pripadaju subklasama u slučaju da one kompletiraju definiciju apstraktnih metoda superklase.

U tekućem primjeru koristit ćemo zasad prvi način i u klasu Osoba dodati metodu prikaz koja samo ispisuje ime osobe.

Za svaku od tri klase ćemo napisati i konstruktor koji će kreirati novi objekt i inicijalizirati određena polja.
tako će postojati konstruktor `Osoba(i)` koji će kreirati objekt tipa Osoba i ime postaviti na `i`. Slično će biti definirani konstruktori `Student(i,b,p,g)` i `Zaposlenik(i,s,t)`

Slijedi definicija klase Osoba i njenih dviju subklasa, Student i Zaposlenik.

EXAMPLE 3

```
/* Objekt Osoba s zajedničkim elementima
   za objekte Zaposlenik i Student */

public class Osoba

{  private String ime;
   // ime osobe.

   /* Vrati ime osobe*/
   public String getIme()
   {  return ime;
   }

   /* Prikaži ime osobe*/
   public void prikaz()
   {  System.out.println("Ime: " + ime);
   }

   /* Kreiraj novi objekt Osoba s imenom i*/
   public Osoba(String i)
   {  ime = i;
   }
}
```

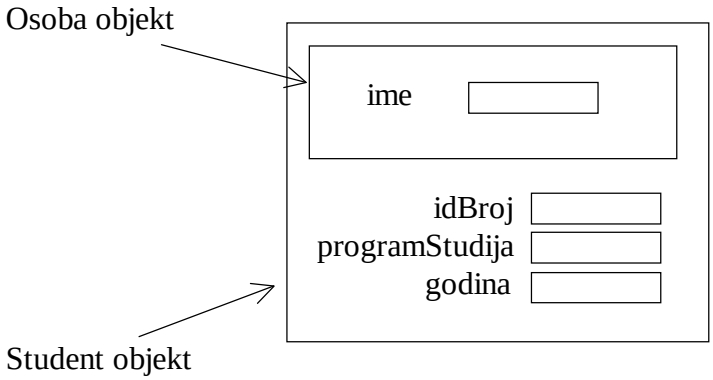
```
/* Student objekt */
```

```
public class Student extends Osoba

{  private String idBroj;
   // Studentov ID broj.

   private String programStudija;
   // Studentov program studija

   private int godina;
   // Studentova godina studija (1, 2 ili 3).

   private Zaposlenik tutor;
   // Studentov tutor.

   /* Promijeni studentov programStudija u p. */
   public void setProgramStudija(String p)
   {  programStudija = p;
   }

   /*  Povećaj godinu studija za 1. */
   public void povecajGodinuStudija()
   {  godina++;
   }

   /* Prikaži podatke o studentu na ekranu*/
   public void prikaz()
   {  super.prikaz();
      System.out.println("ID broj: " + idBroj);
      System.out.println("Program studija: " + programStudija);
      System.out.println("Godina: " + godina);
      System.out.println("Tutor: " + tutor.getIme());
   }

   /* Kreiraj novi objekt Student i inicijaliziraj parametre
   */
   public Student(String i, String b, String p, Zaposlenik t)
   {  super(i);
      idBroj = b;
      programStudija = p;
      tutor = t;
      godina = 1;
   }
}
```

```
/* Zaposlenik */

public class Zaposlenik extends Osoba

{  private String brojSobe;
   //Broj sobe zaposlenika.

   private String brojTelefona;
   //Broj telefona zaposlenika.

   /* Promjena broja sobe i broja telefona*/
   public void promjenaUreda(String s, String t)
   {  brojSobe = s;
      brojTelefona = t;
```

```
    }

    /* Prikaz podataka zaposlenika*/
    public void prikaz()
    {  super.prikaz();
       System.out.println("Broj sobe: " + brojSobe);
       System.out.println("Telefon: " + brojTelefona);
    }

    /* Kreiraj novog Zaposlenika te inicijaliziraj parametre
    */
    public Zaposlenik(String i, String s, String t)
    {  super(i);
       brojSobe = s;
       brojTelefona = t;
    }
}
```

Postoje dvije točke u ovom primjeru koje zaslužuju dodatnu pažnju.
Prva je upotreba konstruktora superklase. zamislite da su objekti Student i Zaposlenik izgrađeni oko objekta Osoba:



Kada pišete konstruktor za subklasu, kao što je klasa Student, Java zahtjeva da počnete s konstrukcijom objekta superklase. U ovome slučaju je objekt superklase objekt Osoba. Postoji određena notacija kako to učiniti. Na početku tijela konstruktora subklase (Student) napisat ćete slijedeći izraz:

```
super(parametri);
```

gdje *parametri* je lista parametara koja se podudara s listom parametra jednog od konstruktora superklase. U gornjem primjeru klasa osoba ima samo jedan konstruktor i njegov jedini parametar je vrijednost koja će biti upisana u polje `ime`.

Tako konstruktori klase Student i Zaposlenik počinju s naredbom:

```
super(i);
```

gdje je i vrijednost koja će biti upisana u polje `ime`.

U nekim slučajevima možete izostaviti ovu naredbu. Tada će prevodilac umetnuti slijedeću naredbu:

```
super();
```

Međutim ako superklasa ne posjeduje konstruktor bez parametara prevodilac će javiti pogrešku.

Postoji i druga upotreba ključne riječi `super` u ovome primjeru. pretpostavimo da ste definirali subklasu neke klase i pregazili (override) metodu npr. nazvanu `m`.
Metodu iz superklase možete i dalje koristiti (naravno ako je public ili protected) ako je pozovete na slijedeći način

```
super.m();
```

To se događa u metodi `prikaz` u klasama Student i Zaposlenik. Obje klase koriste se metodom `prikaz` koja je definirana u klasi Osoba, koju pozivaju naredbom:

```
super.prikaz();
```

Slijedi jednostavan program u kojem ćemo upotrijebiti klase Osoba, Student i Zaposlenik. Program je postavljen u `main` metodu koju možete napisati ili u posebnoj klasi ili u nekoj od prethodno navedenih klasa.

Ova metoda kreira malu listu članova koristeći klasu ArrayList (detaljnije u slijedećem poglavlju) i zatim pretražuje listu tražeći član s nazivom 'Mijo Malek'.
kad ga nađe ispisat će detalje vezane za objekt.

```
Ime:Mijo Malek
ID Broj: 99123456
Program Studija: Elektronika
Godina:1
Tutor: Mr. Bulin
```

Slijedi definicija `main` metode. Unutar metode main koristi se za pretraživanje druga statička metoda `findOsoba`, koja je definirana nakon `main` metode.

PRIMJER 3 nastavak. (`main` metoda)

```
/* Kreiraj malu listu osoba,
   pretraži listu i nađi traženo ime,
   prikaži podatke za pronađenu osobu.
*/
public static void main(String[] a)

{  /*Kreiraj listu osoba. */

   List osobe = new ArrayList();
   Zaposlenik zap1 =
      new Zaposlenik("Mr. Bulin", "201", "5065");
```

```
        osobe.add(zap1);

        Zaposlenik zap2 = new Zaposlenik("Dr. Krpan", "405",
"5055");

        osobe.add(zap2);

        Student student1 = new Student
           ("Mijo Malek", "99123456", "Elektronika", zap1);

        osobe.add(student1);

        Student student2 = new Student

           ("Pero Kavan", "99007964", "Računarstvo", zap2);

        osobe.add(student2);

        /* Pretraži listu i nađi ime "Mijo Malek",
          te ispiši njegove podatke. */

        Osoba m = findOsoba("Mijo Malek", osobe);

        if (m != null)
           m.prikaz();
        else
           System.out.println("Ime nije pronađeno.");
    }

    /* Vrati osobu u osobaList s imenom 'ime'.

       Ako nitko u listi nema to ime vrati null */

    private static Osoba

    findOsoba(String ime, List osobaList)
    {  for (int i = 0; i < osobaList.size(); i++)
       {  Osoba mem = (Osoba) osobaList.get(i);
          if (ime.equals(mem.getIme()))
             return mem;
       }
       return null;
    }
```

Primijetite da `findOsoba` metoda tretira sve elemente liste kao objekte tipa Osoba. Čak se u metodi eksplicitno naglašava da je svaki element u listi tipa Osoba pomoću slijedećeg izraza:

```
Osoba mem = (Osoba) osobaList.get(i);
```

Dakle u listi mogu biti različiti tipovi objekata i sve dok se radi o objektima koji su ili klase Osoba ili su izvedeni iz iste klase program će se uredno izvršavati.

Kada dođemo do izraza

```
m.prikaz();
```

u `main` metodi, gdje je m referenca na objekt tipa Osoba, razvoj događaja je nešto drukčiji. U tijeku izvršavanja programa interpreter će provjeriti da li objekt referenciran s m pripada klasi Osoba ili jednoj od subklasa i prema pripadnosti izvršiti odgovarajuću metodu. (dynamic binding).

---

4. Zajednički okosnica (framework) porodice klasa (GraphApplet primjer)

U prethodnom odjeljku pokazali smo definiranje klase koja je enkapsulirala one dijelove koji su bili zajednički za više klasa. Tako definirana klasa postala je superklasa, a ostale klase su postale njena proširenja (exstensions). Ovaj postupak oslobađa nas ponovnog pisanja zajedničkog koda.

U ovom poglavlju preko primjera appleta koji crta graf funkcije promatrat ćemo hijearhiju klasa odnosno zajedničku okosnicu porodice klasa.

Applet crta graf funkcije y = sin(x)/x. Parametri su vrijednost skale na x osi (xScale) i y osi (yScale).

Aplet možemo realizirati na slijedeći način:

PRIMJER 4

```
public class SinusGrafikonApplet extends Applet

{  private double xScale = 1;
   // Veličina u pikselima jedinice na x-osi.

   private double yScale = 1;
   // Veličina u pikselima jedinice na y-osi

public void paint(Graphics g)
   {  Graphics2D g2 = (Graphics2D) g;
      double dw = getWidth();
      double dh = getHeight();

      /*Crtaj osi. */
      g2.setColor(Color.red);
      g2.draw(new Line2D.Double(0, 0.5*dh, dw, 0.5*dh));
      g2.draw(new Line2D.Double(0.5*dw, 0, 0.5*dw, dh));

      /* Iscrtaj graf. */
      g2.setColor(Color.black);
      for ( int dx = 0; dx < dw; dx++)
      {  double gx = (dx - 0.5*dw) / xScale;
         double gy = function(gx);
         int dy = (int) Math.round(0.5*dh - gy*yScale);

         /* Plot the point (dx,dy). */
         g2.draw(new Rectangle(dx, dy, 1, 1));
      }
```

```
    }

    /* Iscrtavana funkcija */
    public double function(double x)
    {  return Math.sin(x)/x;
    }
}
```

Ako bismo željeli crtati graf neke druge funkcije bilo bi potrebno intervenirati u applet i promijeniti metodu function i eventualno skalu x i y osi.

Sve ostalo trebalo bi biti isto. To nije teško postići korištenjem tehnike 'cut and paste' međutim ako želimo imati klasu koja crta grafove funkcija to nije rješenje.

Ovaj problem može se riješiti tako da se definira klasa GraphApplet kako je to poslije učinjeno. Onda ćete svaki put kad crtate neku drugu funkciju jednostavno nadograditi (extend) GraphApplet klasu i dopisati samo implementaciju metode function.

U GraphApplet klasi metodu function ostavit ćemo nedefiniranom tj. apstraktnom (`abstract`). To automatski znači da će i klasa GraphApplet biti apstraktna.

Još je potrebno riješiti problem inicijalizacije varijabli bazne klase `xScale` i `yScale`.
U subklasi će biti potrebno napisati metodu `init` u kojoj ćemo pozvati `postaviSkalu` metodu bazne klase koja će postaviti vrijednosti `xScale` i `yScale`.

PRIMJER 4 (nova varijanta) (GraphApplet klasa)

```
abstract public class GraphApplet extends Applet

{  private double xScale = 1;
   // Veličina u pikselima jedinice na x-osi.

   private double yScale = 1;
   // Veličina u pikselima jedinice na y-osi

   public void postaviSkalu(double xS, double yS)
   {  xScale = xS;
      yScale = yS;
   }

   public void paint(Graphics g)
   {  Graphics2D g2 = (Graphics2D) g;
      double dw = getWidth();
      double dh = getHeight();

      /*Crtaj osi. */
      g2.setColor(Color.red);
      g2.draw(new Line2D.Double(0, 0.5*dh, dw, 0.5*dh));
      g2.draw(new Line2D.Double(0.5*dw, 0, 0.5*dw, dh));

      /* Iscrtaj graf. */
      g2.setColor(Color.black);
      for ( int dx = 0; dx < dw; dx++)
      {  double gx = (dx - 0.5*dw) / xScale;
```

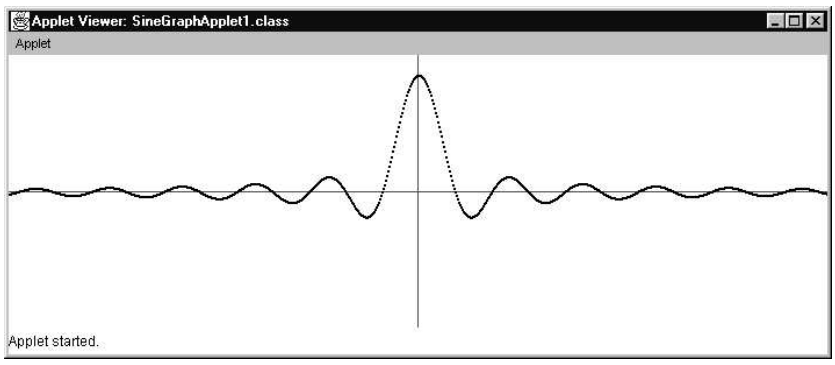
```
            double gy = function(gx);
            int dy = (int) Math.round(0.5*dh - gy*yScale);

            /* Plot the point (dx,dy). */
            g2.draw(new Rectangle(dx, dy, 1, 1));
         }
      }

      /* Funkcija koja će se crtati.
         Potrebno ju je definirati u subklasi od GraphApplet
klase.
      */
      abstract public double function(double x);
   }
```

Slijedi applet koji nadograđuje GraphApplet klasu i crta funkciju y = sin(x)/x.

PRIMJER 4 (nova varijanta)

```
   public class SinusGrafikonApplet extends GraphApplet
   {  public void init()
      {  postaviSkalu(10,100);
      }

      public double function(double x)
      {  return Math.sin(x) / x;
      }
   }
```

U metodi `init` (prva metoda koja se izvršava u appletu) poziva se `setScale` metoda koja je naslijeđena od klase GraphApplet. Graf bi trebao izgledati ovako:



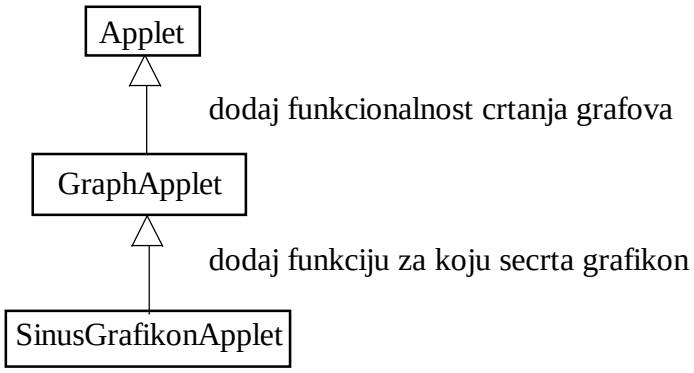
5. Stablo familije klasa

Svaki applet program nadograđuje (extends) klasu Applet. To se može vidjeti iz zaglavlja. Objekt tipa Applet omogućava svu funkcionalnost potrebnu za uspostavu i realizaciju prikaza unutar određenog područja HTML stranice.

Applet nadograđujemo jer time dobivamo specifičniji objekt. Taj objekt će iscrtavati (i tekst se crta) određenu sliku na području appleta npr. smmajlija, robota, grafikon ,...
Uobičajeno se to postiže definicijom nove metode **paint** koja u tom slučaju pregazi originalnu metodu.

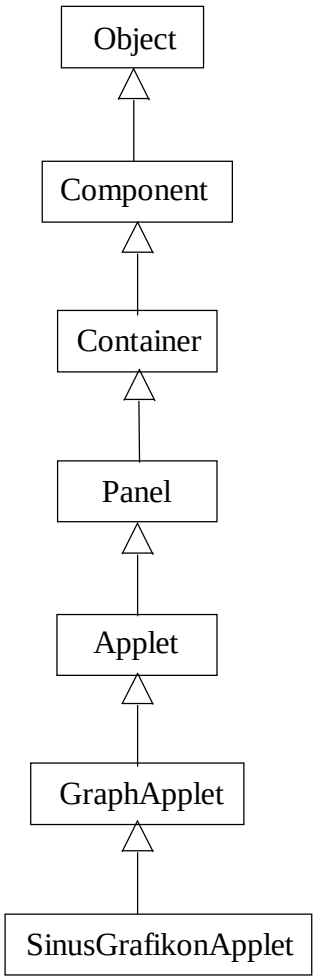
U prethodnom odjeljku išli smo još korak dalje. Prvo smo nadogradili Applet i izveli subklasu GraphApplet. Dodali smo funkcionalnost vezanu za crtanje grafova. Applet SinusGrafikonApplet nadogradio je GraphApplet naslijedivši funkcionalnost i Applet i GraphApplet klase te je još dodao dvije metode.

Slijedeća slika pokazuje što smo učinili:



Trenutne uštede koje postižemo ovakvom organizacijom nisu toliko velike, ali što program postaje kompleksniji ovakvim načinom možemo postići značajne uštede (brži razvoj, manji kod aplikacija,preglednost,...). Ovaj pristup je široko korišten u Java biblioteci.

Prethodna slika nije kompletna. Klasa Applet za svoju superklasu ima klasu Panel koja za svoju subklasu ima klasu Container , ...



Blizu vrha klasa vidite klasu Component . Iz te klase nije izvedena samo klasa Panel nego niz drugih klasa. Isto vrijedi za mnoge druge Java klase. Kako java omugućava samo jednostruko nasljeđivanje onda dijagram svih klasa u Javi izgleda kao obrnuto stablo.

---

6. Zadaci

1. Iskoristi nadogradnju klase GraphApplet za crtanje nekog interesantnog grafa funkcije . (npr. y=5*sin(x / 3)+cos(2 * x)).