6. SWITCH IZRAZ I PETLJE

SADRŽAJ

1. `switch` izraz (kontrolna struktura)

Osim grananja sa if-else naredbom drugi način grananja je upotreba `switch` kontrolne strukture. Switch izraz se koristi rjeđe od if-else izraza, ali postoje određeni programerski zadaci gdje je korisna njegova upotreba . Switch izraz omogućava nam testiranje vrijednosti određenog izraza i ovisno o vrijednosti izraza, skok na određenu lokaciju unutar switch izraza. Vrijednost ispitivanog izraza treba biti cjelobrojna ili karakter. Ne može biti tipa String ili broj u pokretnom zarezu. Pozicije na koje je moguće skočiti imaju formu "case konstanta:". To je mjesto gdje program se nastaviti izvršavati kada je vrijednost izraza jednaka konstanti. Kao zadnji slučaj u switch izrazu možete opcionalno koristiti oznaku "default:", koja predstavlja mjesto gdje će program nastaviti s izvršavanjem ako nije odabrana nijedan "case konstanta:".

Forma `switch` izraza je:

```
switch (izraz) {
   case konstanta-1:
     izrazi-1
     break;
   case konstanta-2:
     izrazi -2
     break;
     .
     .  // (više case naredbi)
     .
   case konstanta-N:
     izrazi -N
     break;
   default:  // opcionalni default slučaj
     izrazi -(N+1)
} // kraj switch naredbe
```

Naredba `break` je tehnički opcionalna. Njen je efekt skok na kraj switch izraza. Ako bismo je izostavili Java bi nastavila s izvršavanjem slijedeće naredbi vezanih uz slijedeći case izraz. To je rijetko ono što želite, ali je sasvim legalno. Ponekad se može upotrijebiti i return na mjestu gdje bi inače upotrijebili break.

Slijedi primjer switch izraza. Primijetite da konstante u case oznakama ne moraju biti poredane , osim što moraju biti različite.

```
switch (N) { // pretpostavimo da je N cjelobrojna varijabla
   case 1:
      System.out.println("Broj je 1.");
      break;
```

```
            case 2:
            case 4:
            case 8:
               System.out.println("Broj je 2, 4, ili 8.");
               System.out.println("(Broj je potencija broja 2)");
               break;
            case 3:
            case 6:
            case 9:
               System.out.println("Broj je 3, 6, ili 9.");
               System.out.println("(Broj je multiplikant broja 3)");
               break;
            case 5:
               System.out.println("Broj je 5.");
               break;
            default:
               System.out.println("Broj je 7,");
               System.out.println("   ili je izvan područja 1 do 9.");
         }
```

2. `while` petlja

Slijedi primjer koda kojeg bismo lakše realizirali pomoću neke petlje.
Npr. Treba unijeti četiri broja: (Pretpostavimo da je `in` objekt tipa ConsoleReader)

```
System.out.println(″Unesi vrijednosti:″);
double suma = 0;
suma  = suma + in.readDouble();
suma = suma + in.readDouble();
suma = suma + in.readDouble();
suma = suma + in.readDouble();
System.out.println(″Suma je ″ + suma);
```

Za četiri broja mogli bismo i zadržati prethodni kod, ali za slučaj da unosimo više brojeva ili da npr. iz neke datoteke čitamo na tisuće brojeva potrebno je upotrijebiti neku od petlji.

Pretpostavimo da varijabla `n` sadrži broj ulaznih vrijednosti.

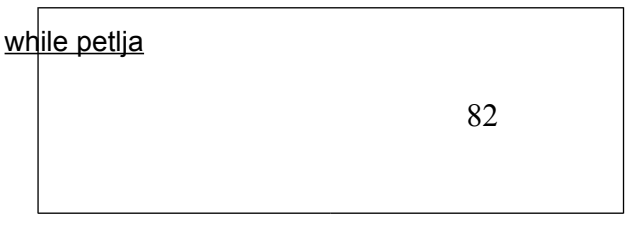
```
System.out.println(″Unesi vrijednosti:″);
double suma = 0;

ponovi n puta  (pseudokod)
   suma = suma + in.readDouble();

System.out.println(″Suma je ″ + suma);
```

Dio koda

```
Ponovi n puta
   suma = suma + in.readDouble();
```

potrebno je konvertirati u Java kod. Upotrijebiti ćemo **petlje (loop statements).**

Prva petlja koju ćemo upotrijebiti je while petlja čiji je opći oblik:

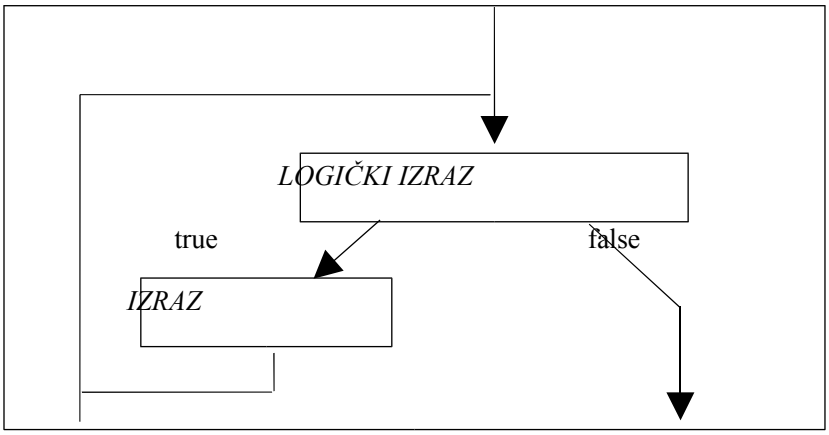


```
while (LOGIČKI IZRAZ)
    IZRAZ
```

Što znači:

Izvršavaj u petlji *IZRAZ*.
Svaki put prije izvršavanja *IZRAZ* provjeri
da li je *LOGIČKI IZRAZ* ima vrijednost true.
Ako nema završi odmah.

Slijedeći dijagram toka pokazuje što se događa tijekom izvršavanja `while` petlje.



Izraz koji se ponavlja je **tijelo** petlje. Tijelo petlje može biti i blok koji se sastoji od niza izraza.

Petlja `while` može se koristiti za obavljanje neke operacije `n` puta.

```
int i = 0;
while (i < n)
{  Operacija
   i++;
}
```

Primijetite da kada *Operacija* bude izvršena `n` puta, `i` će biti jednako `n`, a uvjetni logički izraz `i < n` će biti false te će program izaći iz petlje.

Slijedi kod za određivanje sume `n` ulaznih vrijednosti.

```
System.out.println(”Unesi vrijednosti:”);

double suma = 0;
int i = 0;
while (i < n)
{  suma = suma + in.readDouble();
   i++;
}

System.out.println("Suma je " + suma);
```

Slijedi kompletan program koji zbraja brojeve koje unese korisnik. Program na početku pita koliki je broj ulaznih vrijednosti.

PRIMJER 1

```
public class Zbrajaj1

{  public static void main(String[] args)
   {  ConsoleReader in =
         new ConsoleReader(System.in);

      System.out.print
         ("Koliki je broj ulaznih vrijednosti: ");
      int n = in.readInt();

      System.out.println(”Unesi vrijednosti:”);
      double suma = 0;
      int i = 0;

      while (i < n)
      {  suma = suma + in.readDouble();
         i++;
      }

      System.out.println
         ("Suma je " + suma);
   }
}
```

2. `do-while` petlja

Java ima još jednu petlju koja je vrlo slična `while` petlji. Obično se naziva **`do-while` petlja**. (ili samo `do` petlja.) Jedina razlika između te dvije petlje je da `while` petlja ispituje logički uvjet izlaska iz petlje na početku, prije izvršavanja bilo koje naredbe unutar tijela petlje. Petlja `do-while` izvrši tijelo petlje bar jedanput jer logički uvjet izlaska iz petlje ispituje tak na kraju petlje

Oblik do-while petlje je slijedeći:

do-while petlja

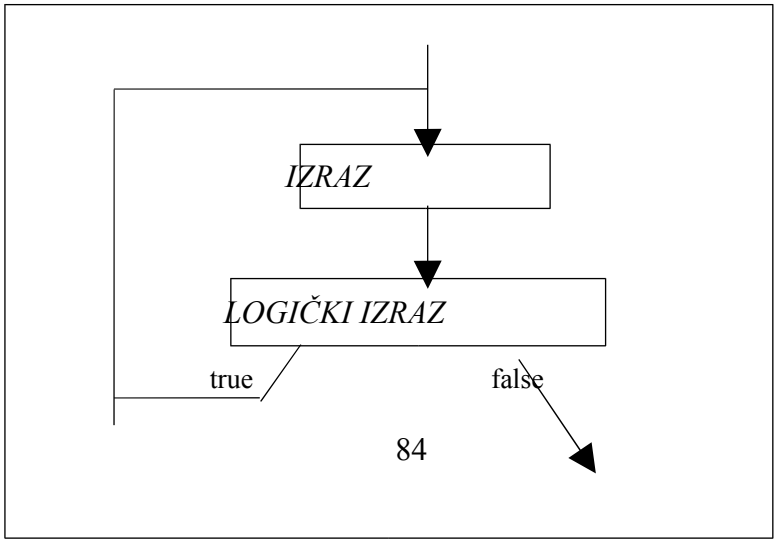
```
do
   IZRAZ
while (LOGIČKI IZRAZ);
```

Što znači:

Izvršavaj u petlji *IZRAZ*.
Svaki put **nakon** što je *IZRAZ* izvršen,

Ako *LOGIČKI IZRAZ* ima vrijednost true nastavi s petljom.
Ako je false, izađi iz petlje.

Blok dijagram do-while petlje je slijedeći:



Jedina razlika je što `while` petlja ima jedan test više odnosno ispitivanje na početku petlje. Zbog toga `do-while` petlju koristimo jedino ako želimo da se tijelo petlje izvrši bar jednom.

Slijedi primjer `do-while` petlje. Program testira korisnika pitajući ga za rezultat operacije množenja dva broja.

Test da li će se test ponoviti ide na kraju i u ovom programu ćemo koristiti `do-while` petlju.

Da bismo generirali brojeve za koje ćemo postaviti upit koristit ćemo generator slučajnih brojeva. Java posjeduje klasu Random koju možemo koristiti za generiranje slučajnih brojeva. Prvo je potrebno generirati objekt tipa random:

```
Random rand = new Random();
```

Ovaj objekt posjeduje metodu `nextInt` koja vraća slučajni cijeli broj. Da bismo dobili slučajni broj u opsegu 0 do n−1, koristimo slijedeći izraz:

```
rand.nextInt(n)
```

Ako koristite `rand.nextInt(n)` više puta , svaki put ćete dobiti različite brojeve. Sekvenca brojeva koju dobijete na ovaj način je naizgled slučajna. Međutim istina je da se za dobivanje slučajnih brojeva koristi relativno jednostavna matematička formula pa po tome brojevi nisu uopće slučajni. Zbog svega ovoga brojevi proizvedeni na ovaj način nazivaju se *pseudo-slučajni* brojevi.

Slijedi primjer generiranja dva slučajna broja u opsegu 1 to 12:

```
int a = rand.nextInt(12) + 1;
int b = rand.nextInt(12) + 1;
```

Nakon ovoga slijedi kompletan program. Klasa Random nalazi se u `java.util` paketu u Java biblioteci. (`util` je skraćenica za 'utility'.)

PRIMJER 2

```
import java.util.*;

public class Multipliciranje

{  /* testiranje tablice množenja. */

   public static void main(String[] args)
   {  ConsoleReader in =
         new ConsoleReader(System.in);
      Random rand = new Random();

      System.out.println
         ("Malo mozganja za korisnika.");
      String reply;
      do
      {  /* generiraj dva pseudo-slučajna broja između 1 i 12*/
         int a = rand.nextInt(12) + 1;
         int b = rand.nextInt(12) + 1;
```

```
            /* Pitaj korisnika koliko je a*b i učitaj odgovor*/

            System.out.println
              ("Koliko je " + a + " puta " + b + "?");
            int odgovor = in.readInt();
            if (odgovor == a*b)
               System.out.println("Točno!");
            else
               System.out.println
                  ("Netočno. Odgovor je ″ + a*b);

            /* Pitaj korisnika da li želi još pitanja*/
            System.out.println
              ("Želite li još pitanja da/ne)?");
            reply = in.readLine();
         }
         while (reply.equals("da"));

         System.out.println("Kraj");
      }
   }
```

Jedno upozorenje oko `do-while` petlje. Razmotrimo slijedeću `do-while` petlju

```
do S while B;
```

Ako je S blok naredbi tada bilo koja varijabla koja je deklarirana unutar bloka S nije dostupna u logičkom izrazu B. Ako želite koristi iste varijable unutar S i B potrebno ih je deklarirati prije `do-while` petlje. (poput varijable `reply` u primjeru 2).

4. `for`-petlja

Java poput svih "C" jezika posjeduje **`for` petlju**. Njen opći oblik je:

`for (`*Uzmi prvu vrijednost*`;` *DokJeTočanIzraz*`;` *OdrediSlijedećuVrijednost*`)`
Operacije

For petlja u javi može koristiti sve cjelobrojne brojčane vrijednosti te brojeve u pokretnom zarezu.

`Primjer petlje:`

```
for (double x = 0; x <= 90; x = x+5)
   System.out.println
      (″\t″ + x + ″\t″ + Math.cos(x));
```

Slijedi jedan od prethodnih primjera napisan korištenjem for petlje:
PRIMJER 5

```
public class Zbrajaj2

{  public static void main(String[] args)

   {  ConsoleReader in =
         new ConsoleReader(System.in);
```

```
         System.out.print
            ("Koliko vrijednosti želite unijeti: ");
         int n = in.readInt();

         System.out.println("Unesi vrijednost");
         double suma = 0;
         for (int i = 0; i < n; i++)
            suma = suma + in.readDouble();
         System.out.println
            ("Suma je " + suma);
      }
   }
```

5. Kontrola izvršavanja petlje - break i continue naredbe

break naredba

Za prisilni izlazak iz petlje koristimo **`break` naredbu**, koju nismo dosada susreli. Naredba `break` kaže Javi: zaustavi trenutačno izvršavanje petlje.

Npr. pišemo program za računanje kvadratnog korijena koji u beskonačnoj petlji učitava brojeve i ispisuje njihov kvadratni korijen. Kada korisnik upiše negativan broj program prestaje s radom.

```
while (true)
{  Čitaj slijedeću ulaznu vrijednost
   i pohrani je u x.
   if (x<0) break;
   Ispiši iznos kvadratnog korijena od x.
}
```

`break` naredba može se koristiti za prekid bilo koje petlje. Također se koristi u formiranju switch izraza.

Slijedi kompletan kod programa:

PRIMJER 3

```
public class KvadratniKorijen
{  /* Čitaj brojeve u pokretnom zarezu i ispisuj njihove
      kvadratne korijene.
      Završi kad korisnik upiše vrijednost <0.
   */
   public static void main(String[] args)
   {  ConsoleReader in =
         new ConsoleReader(System.in);

      System.out.println("Program za račun kv. korijena.\n" +
        "(Unesi vrijednost<0 za kraj.)");

      while (true)
      {  System.out.print("Unesi broj: ");
         double x = in.readDouble();
         if (x<0) break;
         System.out.println
           ("Kv. korijen = " + Math.sqrt(x));
      }
```

```
            System.out.println("Kraj");
        }
    }
```

continue naredba

Da bismo izbjegli trenutnu iteraciju petlje i nastavili sa slijedećom upotrebljavamo `continue` naredbu. Primjer continue naredbe je:

```
for (i=0;  i<=5;  ++i) {
   if (i % 2 == 0)
       continue;
   System.out.println("Ovo je " + i + ". iteracija");
}
```

ili

```
i = 0;
while (i <= 5)  {
      ++i;
      if (i % 2) == 0)
         continue;
      System.out.println("Ovo je neparna iteracija - " + i);
}
```

Ugniježđene petlje

Petlje možemo po volji ugnijezditi jednu u drugu. Ponekad je teško prisilno izaći iz takvih petlji. Java stavlja na raspolaganje varijantu naredbe break :

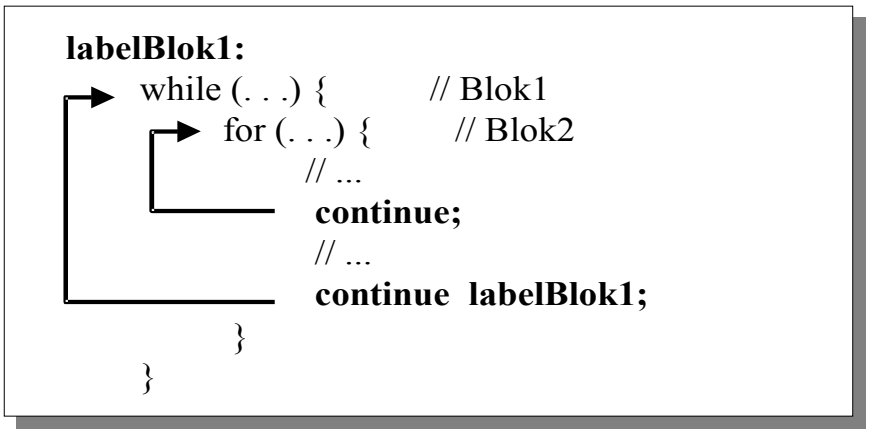
```
break labela;
```

Ova break naredbe izlazi iz petlje koja ispred sebe ima definiranu labelu naziva istog kao naziv labele naveden poslije break naredbe.

```
labelBlok1 :
   for { . . .) // Blok1
   {
     while( . . .) // Blok2
     {
        // . . .
        break;
        // . . .
        break labelBlok1;
     }
        // . . .
   }
```

`continue` naredba ima također varijantu s labelom:

```
continue labela;
```



8. Zadaci za poglavlje 6

1. Napišite program koji će čitati cjelobrojnu vrijednost n koju unese korisnik i onda izračunati faktorijel od n te ga prikazati na ekranu. Koristite bilo koju od petlji. (Faktorijel.java)

2. Napišite program koji će izračunati i ispisati faktorijel za vrijednosti od 1 do 20. Za ovo će vam biti potrebna petlja unutar petlje. (Faktorijel2.java)

3. Napišite program koji će "zamisliti broj" od 1 do 1000. Zatim će korisnika pitati da pogodi koji je broj zamišljen. Kad korisnik unese broj program mu mora odgovoriti da li je broj veći, manji ili jednak zamišljenom broju. Ako je jednak prekida se izvršavanje uz odgovarajuću poruku (čestitka, broj pokušaja). Ako je manji ili veći onda se poslije odgovarajuće poruke korisniku nudi ponovno pogađanje.

7. DEFINICIJA KLASA

U ovom poglavlju detaljnije ćemo razraditi temu koja je načeta u poglavlju 3: pisanje definicija klasa. Ovo je jedna od najvažnijih tema predavanja. Razumijevanje kako se definiraju klase predstavlja osnovu programiranja u Javi.

SADRŽAJ

1. Statičke (static) varijable i metode

U poglavlju 3 definirali smo klasu student. Ta je klasa koristila detalje u definiciji klasa koji su zajednički većini klasa. Klasa student određivala je određeni tip objekta koji je predstavljao skup podataka pojedinog studenta.

Svaki objekt tipa student posjedovao je određeni broj :

varijabli instance – polja: u njima su bile sadržane informacije poput imena, godine studija, upisanog programa itd. pojedinog studenta.

metoda instance: korištene su za pristup i ažuriranje varijabli instance

bar jedan konstruktor: služi za kreiranje objekta tipa Student

Polja su bila označena kao `private`. To je značilo da se poljima ne može pristupiti direktno (programer korisnik klase nije mogao napisati npr. Student.ime)

Metode instance bile su označene kao `public`, što je značilo da ih programer korisnik klase može koristiti.

Konstruktor je također bio `public,` što je značilo da ga programer korisnik može pozvati u slučaju da želi kreirati novi objekt tipa Student.

Programer korisnik vidi samo dijelove klase Student koji su označeni kao **public**. Ostalo mu je nedostupno. Public metode, konstruktori i polja bi trebala biti dovoljna za korištenje određene klase. Ovi poblic članovi nazivaju se ponekad sučelje (interface) klase.

Java klase su mnogo fleksibilnije nego što to pokazuje klasa Student.
Npr. klasa može uključivati **statičke varijable .**
Statička varijabla egzistira u memoriji neovisno o bilo kojem objektu. Kreira se kada program počinje s izvršavanjem i nestaje kada program prestaje s izvršavanjem. Statičkoj varijabli se može pristupiti iz bilo kojeg metoda koji je definiran u klasi. Ako je deklarirana kao `public` može joj se pristupiti i izvan klase.

Kada je program pokrenut postoji samo jedna kopija statičke varijable. To je različito od polja klase koja postoje u svakoj instanci objekta. Npr. možemo napisati takav program koji će kreirati na stotine objekata tipa Student i svaki od njih će imati svoje polje s nazivom `ime`.
U trenutku kad se program pokrene neće postojati nijedan objekt tipa Student. Tek kad program počne kreirati objekte tipa Student pojavit će se u svakom od objekata jedna kopija polja `ime`.

Program može također definirati i **statičke metode**. Takve metode nisu asocirani ni s jednim objektom. Metoda `main` koja kontrolira svaku Java aplikaciju je uvijek statička metoda. Usporedite to s metodom `setProgramStudija` koja je uvijek asocirana uz objekt. To znači da tu metodu nije moguće pozvati ako

nismo kreirali objekt tipa Student. S druge strane statička metoda nije asocirana uz nijedan objekt i moguće ju je pozvati prije nego je ijedan objekt kreiran.

Nije bitno je li metoda statička ili je metoda instance tj. metoda može pristupati svim članovima klase (polja, metode i konstruktori) bez obzira jesu li public ili private.

Statičke metode i polja uvijek sadržavaju ključnu riječ `static`. Npr.

```
public static int brojač;
```

Unutar klase u kojoj su definirani , statičkim metodama uvijek pristupamo preko identifikatora npr. `brojač`.
Van klase naziv varijable je definiran tako da je prefiks naziv klase (varijabla mora biti public).
Npr. `PI` je statička varijabla u klasi Math koja sadržava vrijednost broja π. Ona je `public`, tako da joj možemo pristupati u drugim klasama. Pristupamo joj preko punog imena npr.

```
double konst = 180 / Math.PI;
```

Isto vrijedi i za pozivanje satičkih metoda. Npr. klasa Math sadrži statičku metodu `cos` koja računa kosinus kuta. Ona je public, tj. moguće ju je koristiti. Pozivamo je preko punog imena:

```
stupnjevi = Math.cos(radijani) * konst;
```

Klasa Math je dobar izvor primjera statičkih varijabli i metoda jer je cijela saastavljena od istih. (Pogledajte u dokumentaciji opis klase Math)

2. Klasa Robot

U Javi ćemo definirati klasu nazvanu Robot. Ova klasa predstavljat će robota s olovkom koji će se kretati po površini appleta i ostavljati za sobom trag. Na području prikaza appleta nećemo vidjeti samog robota već samo njegov trag. Koristeći klasu Robot moći ćemo definirati niz objekata tipa Robot.
Kretanje pojedinog robota bit će kontrolirano pozivima metoda klase. Slijedi niz metoda koje su definirane u klasi Robot. Sve metode su metode instance tj. asocirane su s pojedinim objektima tipa Robot. Sve metode su public.

`r.pomakni(s)` Miče robota naprijed za udaljenost s. s se mjeri u pikselima.

`r.lijevo(deg)` Okreće robota ulijevo za deg stupnjeva.

`r.desno(deg)` Okreće robota udesno za deg stupnjeva.

`r.olovkaDolje()` Spušta olovku robota tako da robot piše svoj trag dok se kreće.

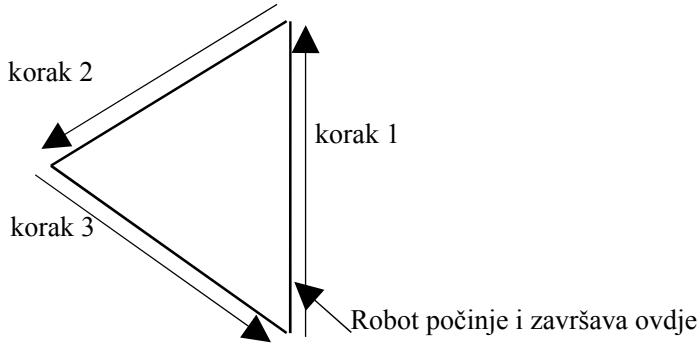
`r.olovkaGore()` Diže olovku robota.

Parametri s i d i deg su svi tipa `double`. Pored ovih metoda imamo još i konstruktor `Robot()`, koji će kreirati novi objekt robota u "početnoj poziciji". To je centar ekrana, s pogledom prema gore i isključenom olovkom.

Slijedi nekoliko naredbi koje bi trebale kreirati objekt tipa Robot i naredit mu crtanje trokuta.

```
Robot r = new Robot();
r.olovkaDolje();
r.pomakni(50);
r.lijevo(120);
r.pomakni(50);
r.lijevo(120);
r.pomakni(50);
```

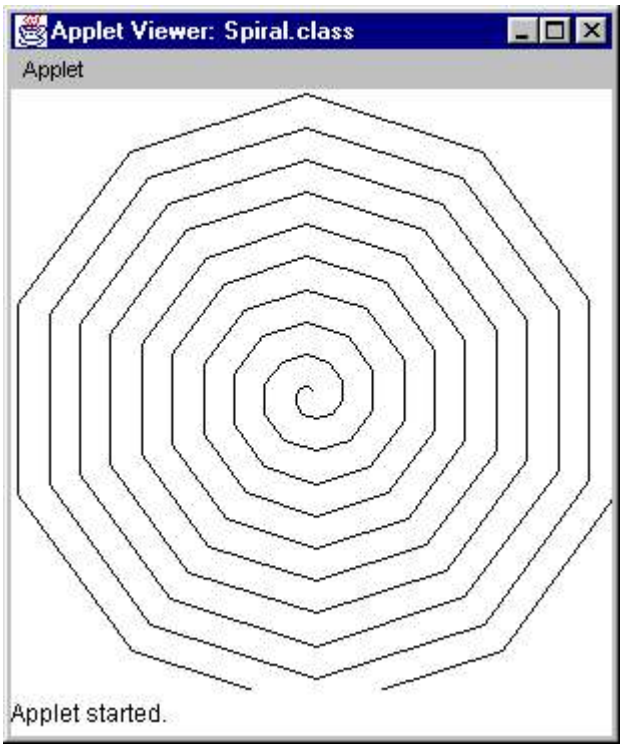
Svaki put kad se robot pokrene ide naprijed za udaljenost od 50(piksela) sa spuštenom olovkom crtajući liniju na ekranu. Svaki put kad stigne do kraja linije okrene se za kut od 120° nakon čega slijedi opet crtanje linije.



Slijedi još jedan robot program.

```
Robot r = new Robot();
r.olovkaDolje();
for (int i = 0; i < 100; i++)
{  r.pomakni(i);
   r.lijevo(36);
}
```

Tijelo petlje izvršava se 1000 puta. Svaki put robot se pokrene naprijed i zakrene ulijevo. Kad bi se pomicao za isti iznos naprije i zakretao za isti kut nakon nekog vremena krivulja bi se zatvorila. Umjesto toga svaki pomak je nešto veći pa na ekranu dobivamo spiralu.



Dosada smo samo gledali kako ćemo koristiti objekte tipa `Robot` odnosno već smo definirali sučelje objekta tipa Robot. Vrijeme je za implementaciju klase Robot.

Polja u objektu tipa Robot moraju sadržavati sve podatke potrebne da bi se predstavilo stanje robota u određenom trenutku. Za ovaj program nisu bitni svi mogući podaci o robotu. Npr. nije potrebno zapisati godinu proizvodnje robota ili koje je boje. Međutim njegova pozicija ne ekranu , smjer u kojem je okrenut i informacija da li ima spoštenu ili podignutu olovku su nam relevantne informacije.

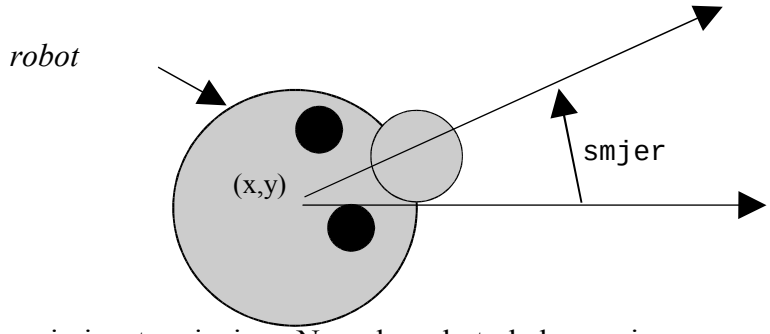
Tako će klasa `Robot` imati slijedeća polja:

1. Dva polja koja će sadržavati x i y-koordinate pozicije robota na ekranu.

```
private double x,y;
```

x i y su koordinate ekrana i mjere se u pikselima s izvorištem u gornjem lijevom kutu ekrana.

2. Jedno polje nazvano `smjer` za predstavljanje smjera u kojem je robot okrenut. To je kut između linije koja pokazuje na istok i linije smjera robota.



Kut se mjeri u stupnjevima. Npr. ako robot gleda u smjeru prema vrhu ekrana kut će biti 90. Ovo polje je deklarirano na slijedeći način:

```
private double smjer;
```

3. Jedno polje za zapis podatka da li je olovka spuštena ili nije. To je polje tipa boolean koje će biti postavljeno na `true` ako je olovka spuštena , a na `false` ako nije. Deklarirano je na slijedeći način:

```
private boolean jeDolje;
```

Neke od odluka prilikom kreiranja ovih polja su uzete sasvim proizvoljno. Izvorište koordinatnog sustava mogli smo staviti u centar ekrana, a osi su mogle biti usmjerene na standardan način.

Ove odluke su stvar *implementacije*. Programer korisnik klase oslanja se samo na sučelje.
Njemu su na raspolaganju već pokazane metode. Npr. njemu nije bitan smjer i izvorište koordinatnog sustava , što onome koji implementira klasu je od bitne važnosti.
Ovo skrivanje detalja implementacije od programera korisnika je poznato pod pojmom *enkapsulacija (encapsulation)*. To predstavlja jednu od najkorisnijih osobina objektno orijentiranog programiranja.

Jedna od osobina enkapsulacije implementacije klase Robot je prema potrebi program koji je implementirao klasu Robot može je potpuno nanovo napisati (ubrzati, učiniti pouzdanijom,...) te ostavljajući isti oblik pet public metoda i konstruktora opet osigurati da programi koji tu klasu koriste za crtanje kao što su već navedeni `Spiral` i `Star` i dalje ispravno rade.

Kada smo odabrali polja slijedeći je korak implementacija metoda i konstruktora.
Metoda `olovkaDolje` mora osigurati da se olovka nalazi u spuštenom položaju. Jednostavno mora postaviti polje `jeDolje` na vrijednost `true`. Slično `olovkaGore` mora postaviti `jeDolje` na `false`. Slijede definicije ovih dviju metoda.

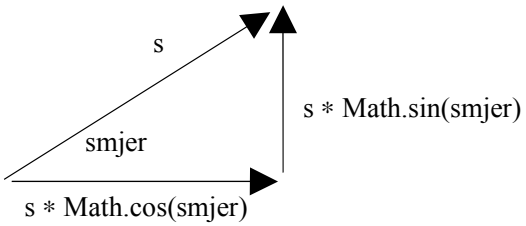
```
/** Podigni olovku gore.
*/
public void olovkaGore()
{  jeDolje = false;
}
```

```
/** Spusti olovku dolje.
*/
public void olovkaDolje()
{  jeDolje = true;
}
```

Metode `lijevo` i `desno` su isto tako jednostavne. Metoda `lijevo(deg)` jednostavno dodaje `deg` na smjer tj. kut gledanja robota. `desno(deg)` oduzima `deg`.

```
/** Okreni robota ulijevo za deg stupnjeva.
*/
public void lijevo(double deg)
{  smjer = smjer + deg;
}

/** Okreni robota udesno za deg stupnjeva.
*/
public void desno(double deg)
{  smjer = smjer - deg;
}
```

Ako robot se pokrene za udaljenost s, njegova x koordinata bit će uvećana za iznos `s*Math.cos(smjer)`. Pogledajte slijedeću sliku:



Koordinata y je *umanjena* za iznos `s*Math.sin(smjer)`. (Umanjena je jer je ordinata y obrnuta u odnosu na matematičku orijentaciju) . Koristeći ova dva izraza možemo izračunati novu poziciju robota. Ako je olovka uključena možemo nacrtati liniju od stare pozicije k novoj. Primjetite da se kut `smjer` mora pretvoriti u radijane prije nego što se upotrijebi kao argument u metodama `Math.cos` i `Math.sin`. Dio koji je podebljan je još uvijek pseudo kod.

```
/** Pomakni robota za udaljenost s.
*/
public void pomakni(double s)
{  double stariX = x;
   double stariY = y;
   double radijani = smjer * Math.PI / 180;
   x = x + s*Math.cos(radijani);
   y = y - s*Math.sin(radijani);
   if (jeDolje)
   Crtaj liniju od (stariX,stariY) do (x,y).
}
```

Kako nacrtati liniju od stare pozicije k novoj ? Kako kreiramo objekt linije već znamo:

```
Line2D.Double =
   new Line2D.Double(stariX, stariY, x, y);
```

Međutim da bismo nacrtali liniju potreban je Graphics2D objekt spojen na područje prikaza appleta. Kako pristupati objektu Graphics2D ? Vratit ćemo se poslije na to pitanje.

Konstruktor treba kreirati objekt lociran u centru područja prikaza. Koordinate te točke su (w/2, h/2) gdje je w širina područja prikaza, a h je visina. Prema tome definicija konstruktora je slijedeća:

```
/** Kreiraj novog robota u centru područja prikaza,
    okrenutog prema sjeveru, s olovkom gore.
*/
public Robot()
{   Postavit w = širina područja prikaza appleta.
    Postavi h = visina područja prikaza appleta.
    x = 0.5*w;
    y = 0.5*h;
    smjer = 90;
    jeDolje = false;
}
```

Pitanje koje se ovdje postavlja je kako dobiti visinu i širinu područja prikaza appleta. Metoda instance `pomakni` i konstruktor trebaju na neki način imati pristup području prikaza na kojemu robot crta. Zbog toga trebamo znati nešto više o tome kako Java radi s prozorima.

Ono što vidite kad se program izvršava predstavlja *grafičko korisničko sučelje* (*graphical user interface)* - GUI. Razni prozori, dugmad, meniji , itd. koji sačinjavaju GUI prestavljaju njegove *komponente (components)*. Dosada su naši Java programi koristili samo neke komonente. Appleti su koristili samo jedno područje prikaza, a aplikacije samo DOS prozor. Čak i najjednostavnije profesionalne aplikacije poput Notepada koriste menije i skroll trake.

U slučaju Java programa, svaka GUI komponenta je asocirana s nekom formom **Component** objekta koji upravlja s područjem ekrana koje je dodijeljeno komponenti. Ako bismo uspjeli izvesti da klasa Robot pristupa Component objektu koji upravlja s područjem prikaza onda bismo mogli izvesti sve potrebne operacije.

Pretpostavimo da je c neki Component objekt i neka je pridružen pravokutnom području prikaza. Vrijednost koju bi vratio poziv metode

```
c.getWidth()
```

bila bi širina područja prikaza u pikselima i

```
c.getHeight()
```

bi bila visina područja. Također bilo bi moguće koristiti izraz

```
c.getGraphics()
```

da bi se dobio Graphics objekt za crtanje po području prikaza komponente.

Ono što nam treba za kompletiranje defrinicije klase Robot je da svakom objektu tipa Robot damo pristup do objekta Component koji upravlja s područjem prikaza appleta. To ćemo učiniti tako da definiramo statičku varijablu `window`. Ova varijabla će sadržavati referencu na objekt tipa Component koji upravlja s područjem prikaza appleta.

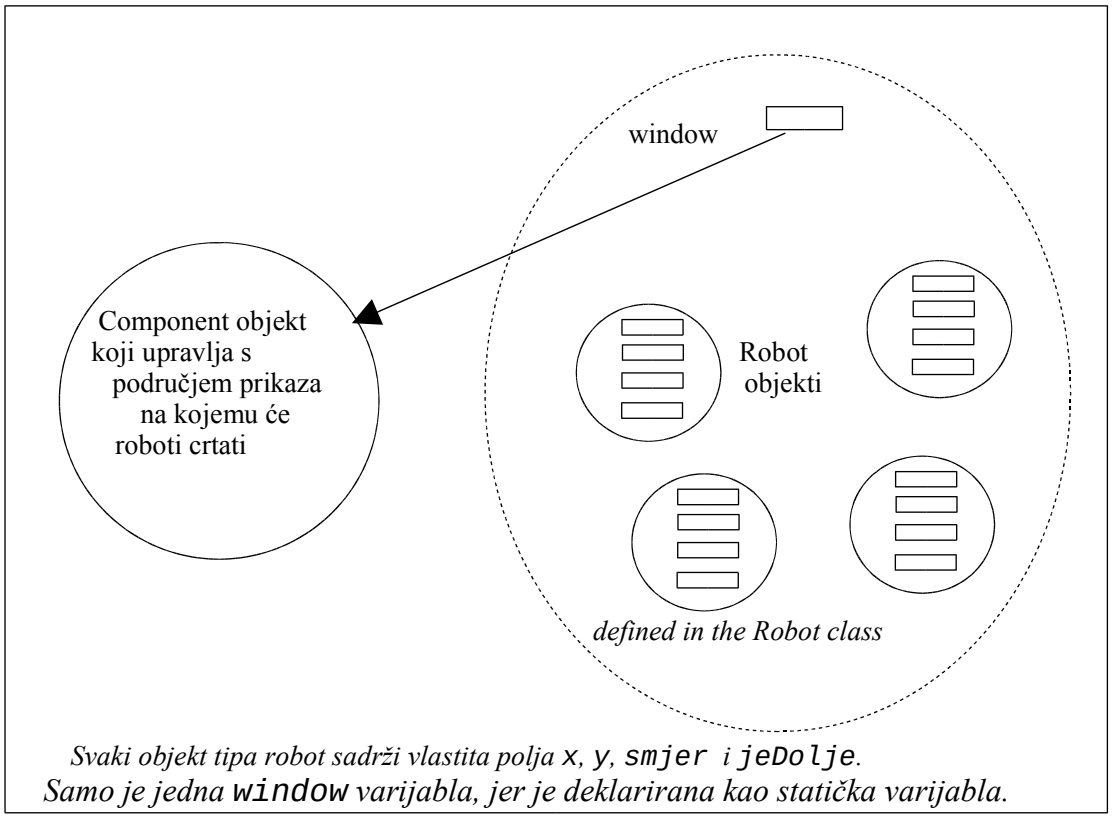
Statička varijabla `window` nije polje objekta Robot.

Ona je po definiciji **jedna** varijabla kojoj mogu pristupiti svi objekti klase Robot. Zbog toga će biti sastavni dio definicije klase Robot.

Onda ćemo moći koristiti izraze:

- `window.getWidth()` za dobiti širinu područja prikaza,
- `window.getHeight()` za dobiti visinu područja prikaza,

`window.getGraphics()` za dobiti Graphics objekt potreban za crtanje po području prikaza.



Svaki objekt tipa robot sadrži vlastita polja `x`, `y`, `smjer` *i* `jeDolje`.
Samo je jedna `window` *varijabla, jer je deklarirana kao statička varijabla.*

Slijedi kompletna definicija konstruktora:

```
public Robot()
{  int w = window.getWidth();
   int h = window.getHeight();
   x = 0.5*w;
   y = 0.5*h;
   smjer = 90;
   jeDolje = false;
}
```

Nakon ovoga slijedi kompletna definicija metode `pomakni` . Graphics objekt proizveden pozivom `window.getGraphics()` može se kastirati u Graphics2D objekt na isti način kako smo to radili u `paint` metodi appleta. Nakon crtanja linije Graphics2D objekt potrebno je odbaciti da ne bi bespotrebno koristio resurse sustava. To je učinjeno u zadnjoj liniji metode.

```
public void pomakni(double s)
{  double stariX = x;
   double stariY = y;
   double radijani = smjer * Math.PI / 180;
   x = x + s*Math.cos(radijani);
   y = y - s*Math.sin(radijani);
   if (jeDolje)
   {  Line2D.Double line =
         new Line2D.Double(stariX, stariY, x, y);
      Graphics2D g2 =
```

```
            (Graphics2D) window.getGraphics();
         g2.draw(line);
         g2.dispose();
      }
   }
```

Za kompletiranje klase Robot još ćemo dodati metodu `setWindow` za postavljanje varijable `window` da referencira na potrebnu komponentu. Ova metoda je neovisna o bilo kojem određenom Robot objektu i predstavlja statičku metodu.

```
/** Postavi 'window' da referencira na komponentu c.
*/
public static void setWindow(Component c)
{  window = c;
}
```

Applet je vrsta Component objekta.

Stoga u pozivu `Robot.setWindow(c)` proslijedit ćemo kao c referencu na sami applet.

Java ima ključnu riječ **`this`** koja se može upotrijebiti u bilo kojoj metodi instance i koja predstavlja referencu na objekt na kojemu se ta metoda trenutno izvodi.

Ako izraz

```
Robot.setWindow(this);
```

izvršimo u jednoj od metoda appleta onda će `window` varijabli biti dodijeljena referenca na taj applet i svi roboti koje budemo kreirali crtat će u području tog appleta.
Kako je ovu naredbu potrebno izvršiti samo jednom prirodno mjesto za njen smještaj je `init` metoda appleta.

(Primjedba. Možda ste primjetili da `init` i `paint` metode u klasi tipa applet nisu statičke metode i zbog toga moraju biti asocirane s nekim objektom).

Nakon što smo postavili varijablu `window` , program može kreirati Robot objekte i koristit ih za crtanje na području prikaza. Prirodno područje za crtanje je `paint` metoda appleta jer će crtanje robota biti svaki put pozvano prilikom obnavljanja sadržaja ekrana.

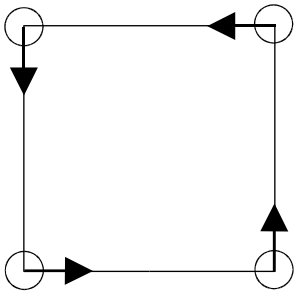
Slijedi applet koji koristi jednog robota za crtanje spirale koju smo već pokazali.

PRIMJER 1

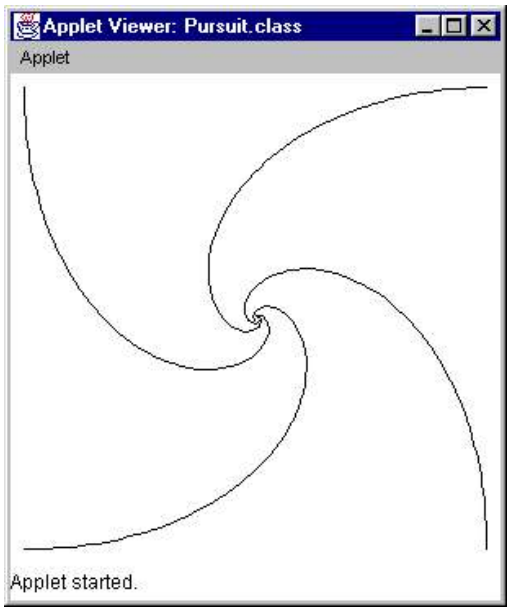
```
import java.applet.Applet;
import java.awr.Graphics;

public class Spiral extends Applet
{  public void init()
   {  Robot.setWindow(this);
   }
   /* Crtaj spiralu. */
   public void paint(Graphics g)
   {  Robot r = new Robot();
      r.olovkaDolje();
      for (int i = 0; i < 100; i++)
      {  r.pomakni(i);
         r.lijevo(36);
      }
   }
}
```

Primjetite da ovaj program ignorira Graphics objekt koji je window manager proslijedio `paint` metodi. Umjesto njega oslanja se na robotovu `pomakni` metodu koja ima pristup Graphics objektu preko `window` reference.

Slijedeći primjer koristi četiri robota. Prvo se pomaknu u četiri kuta kvadrata. Zatim spuste olovke. Nakon toga počnu loviti jedan drugoga. Robot u gornjem lijevom kutu kvadrata okreće se prema robotu u desnom lijevom kutu koji se okreće prema robotu u dojnjem desnom kutu, itd. Jasnije je to predočiti slikom:



Nakon što pređu malu udaljenost , svaki od njih korigira svoj smjer prema robotu kojega prati. Konačna slika bi trebala izgledati ovako:



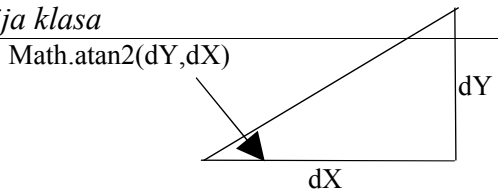
Metode instance koje smo dosada napisali nisu dovoljne za realizaciju ovog programa. Pomoću njih ne možemo ostvariti da jedan robot gleda u drugoga. Bit će potrebno dodati slijedeću metodu:

```
r1.gledaGA(r2);
Okreni robot r2 tako da gleda u robot r1.
```

Here is the definition of `gledaGA`.

```
/** Turn Robot r to face this robot.
*/
public void gledaGA(Robot r)
{  double dX = x - r.x;
   double dY = r.y - y;
   double rad = Math.atan2(dY,dX);
   r.smjer = 180*rad/Math.PI;
}
```

Metoda `Math.atan2(dY,dX)` vraća kut u radijanima torkuta kako je to pokazano na slici:



Zadnji izraz konvertira kut u stupnjeve. Slijedi kompletan applet za crtanje krivulje proganjanja:

PRIMJER 2

```
import java.applet.Applet;
import java.awr.Graphics;

public class Pursuit extends Applet

{  private final int step = 5;
   private final int diag = 200;
   private final int koliko = 58;

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

   public void paint(Graphics g)
   {  /* Pozicioniraj robote. */
      Robot r0 = new Robot();
      r0.lijevo(45);
      r0.pomakni(diag);
      r0.olovkaDolje();

      Robot t1 = new Robot();
      r1.lijevo(135);
      r1.pomakni(diag);
      r1.olovkaDolje();

      Robot r2 = new Robot();
      r2.lijevo(225);
      r2.pomakni(diag);
      r2.olovkaDolje();

      Robot t3 = new Robot();
      r3.lijevo(315);
      r3.pomakni(diag);
      r3.olovkaDolje();

      /*Učini da roboti proganjaju jedan drugoga. */
      for (int i = 0; i < koliko; i++)
      {  r1.gledaGA(t0);
         r0.pomakni(korak);

         r2.gledaGA(t1);
         r1.pomakni(korak);

         r3.gledaGA(r2);
         r2.pomakni(korak);

         r0.gledaGA(t3);
         r3.pomakni(korak);
      }
```

```
      }
   }
```

Slijedi kompletna klasa Robot:

```
import java.awr.*;
import java.awr.geom.*;
import java.applet.*;

/** Robot je na određenoj poziciji ekrana
    i gleda u određenom smjeru. Kada se pomakne
    crta po ekranu ako mu je olovka spuštena
*/

public class Robot

{  private double x,y;
   // x i y-koordinate robota.

   private double smjer;
   // trenutni smjer gledanja robota
   // mjeren u stupnjevima suprotno od kazaljke na satu
    // u odnosu na istok

   private boolean jeDolje;
   // jeDolje = true ako je olovka spuštena,
   // false ako nije.

   private static Component window;
   // komponenta po kojoj robot crta.


   /** Pomakni robota za udaljenost s.
   */
   public void pomakni(double s)
   {  double stariX = x;
      double stariY = y;
      double radijani = smjer * Math.PI / 180;
      x = x + s*Math.cos(radijani);
      y = y - s*Math.sin(radijani);
      if (jeDolje)
      {  Line2D.Double line =
            new Line2D.Double(stariX, stariY, x, y);
         Graphics2D g2 =
            (Graphics2D) window.getGraphics();
         g2.draw(line);
         g2.dispose();
      }
   }

   /** Okreni robota ulijevo za deg stupnjeva.
   */
   public void lijevo(double deg)
   {  smjer = smjer + deg;
   }

   /** Okreni robota udesno za deg stupnjeva.
   */
   public void desno(double deg)
   {  smjer = smjer - deg;
   }
```

```
    /** Podigni olovku gore.
    */
    public void olovkaGore()
    {  jeDolje = false;
    }


    /** Spusti olovku dolje.
    */
    public void olovkaDolje()
    {  jeDolje = true;
    }

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

    /** Kreiraj novog robota u centru područja prikaza,
     okrenutog prema sjeveru, s olovkom gore.
    */
    */
    public Robot()
    {  int w = window.getWidth();
       int h = window.getHeight();
       x = 0.5*w;
       y = 0.5*h;
       smjer = 90;
       jeDolje = false;
    }

   /** Postavi 'window' da referencira na komponentu c.
    */

    public static void setWindow(Component c)
    {  window = c;
    }
}
```

3. Više o varijablama i metodama

Napravit ćemo mali opći pregled tipova varijabli imetoda u Javi.

Postoje četiri različite vrste varijabli:

1. **Lokalne varijable**
   Ovo su varijable koje se deklariraju unutar tijela metoda deklaracijom poput slijedeće:

   ```
   double dX = x – r.x;
   ```

   U tijeku izvršavanja metode varijabla će biti kreirana čim Java izvrši gornju naredbu. Prestat će postojati kada Java napusti blok u kojem je varijabla deklarirana. Svakako će prestati postojati kada Java napusti metodu u kojoj je varijabla deklarirana.

   Kada se lokalna varijabla deklarira, a ne dodijeli joj se inicijalna vrijednost onda će vrijednost te varijable biti *nedefinirana* .

2. **Varijabla u listi parametara.**

Pogledajmo slijedeću definiciju metode:

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

Pretpostavimo da je ova metoda pozvana na slijedeći način:

```
r1.gledaGA(r0)
```

Čim se izvrši tijelo metode kreira se varijabla koja će sadržavati vrijednost parametra. Ta će varijabla biti nazvana r. Na ovaj način ova varijabla postaje extra lokalna varijable koju nazivamo varijabla parametra. Za svaki parametar stvara se posebna varijabla. Ove varijable se u svim pogledima ponašaju kao lokalne varijable tj. postoje do kraja izvršavanja metode. Ovakva varijabla ne može biti neinicijalizirana jer je prilikom pozivanja metode u nju postavljena njena inicijalna vrijednost.

Naziv parametra koji je korišten u definiciji metode ponekad se naziva *formalni parametar* , a parametar koji je upotrebljen u pozivu metode naziva se *stvarni parametar (actual parameter).*

3. **Varijabla instance** (ili **polje**)

Polje uvijek pripada određenom objektu. Polje se kreira u trenutku kada se kreira objekt. Postoji sve dok postoji i objekt.

Polju se može dati inicijalna vrijednost tijekom deklaracije. Ako se to ne učini polje će sadržavati *pretpostavljenu inicijalnu vrijednost (default initial value)*. To je 0 u slučaju brojeva, `false` u slučaju `boolean` varijable, i `null` u slučaju varijable koja je referenca na obekt. (U slučaju `char` polja, koje sadrži znak, to je vrijednost'\u0000'.)

Kada koristimo naziv polja unutar metode potrebno je prije naziva staviti naziv objekta kojemu to polje pripada.

npr. r.x

osim ako polje ne pripada objektu za kojeg je pozvana ta metoda. U tom slučaju pišemo samo naziv polja.

Npr. metoda `gledaGA` u klasi Robot sadrži izraz

```
double dX = x – r.x;
```

`dX` je lokalna varijabla. `x` je polje objekta Robot za kojeg je pozvana metoda `gledaGA` . `r.x` je polje objekta Robot čija je referenca proslijeđena u metodu `gledaGA` kao parametar i označena je sa `r.`

(Unutar statičke metode koja se ne poziva ni za jedan objekt potrebno je uvijek staviti prefiks ispred polja.)

4. **Statičke varijable** (**varijable klase**).

Ova vrsta varijabli ima najduži život. Kreira se kada program započne i traje dok program ne prestane s radom. Poput polja poprima prepostavljenu inicijalnu vrijednost ako nije inicijalizirana.

Kada koristite naziv statičke varijable u metodi potrebno je prije naziva staviti naziv klase kojoj varijabla pripada (npr. Math.PI). Iznimka je kada tu varijablu koristimo unutar metoda klase u kojoj je definirana kada upotrebljavamo naziv bez prefiksa.

Statička varijabla može se koristiti za razmjenu informacija između statičkih metoda. Međutim općenito nije dobro mjesto za razmjenu informacija između metoda instance istoga objekta jer nikad ne znamo da li će neki drugi objekt u međuvremenu promijeniti sadržaj.

Postoje dvije različite vrste metoda:

1. **Metode instance**.
   Kada se metoda instance izvršava uvijek je asocirana za neki objekt. Metoda može referencirati odnosno pozivati sve polja i metode tog objekta koristeći njihov "kratki" naziv. Isto tako može pozivati statičke metode i polja koristeći kratki naziv.

2. **Statičke metode**.
   Statička metoda nikada nije asocirana s nekim objektom. Ako statička metoda treba referencirati polje ili metodu nekog objekta to mora učiniti koristeći referencu na taj objekt odnosno ispred naziva treba biti prefiks reference na objekt. S druge strane može pristupati statičkim metodama i poljima iste klase koristeći kratki naziv.

Često se u Javi susreću klase koje sadrže nekoliko metoda s istim nazivom. Međutim te metode se razlikuju u *signaturi* (signature).

Signatura metode sastoji se od naziva metode i liste tipova parametara (bitan je redoslijed). Dakle dvije metode s istim nazivom imaju različitu signaturu ako imaju različitu listu parametara. Java će po listi parametara u pozivu metode znati koju metodu treba pozvati.

Korištenje istog naziva za više različitih metoda u istoj klasi naziva se **preopterećenje (overloading).**

Slična stvar vrijedi i za konstruktore. Jedna klasa može imati više različitih konstruktora i oni se razlikuju samo po listi parametara. To je vrlo uobičajena stvar u Java biblioteci. Već smo to susreli kod kreiranja objekta tipa Color. Npr. postoji sedam različitih konstruktora za objekt tipa Color. jedan prima tri `float` parametra u opsegu od 0 do 1, jedan prima `int` vrijednosti u opsegu 0 to 255, itd. Ako napišemo:

```
new Color(1f, 1f, 1f)
```

pozvan će biti konstruktor koji smo već koristili u poglavlju 4 koji će dati bijelu boju.
Međutim ako napišemo :

```
new Color(1, 1, 1)
```

pozvat ćemo konstruktor koji očekuje tri `int` vrijednosti u opsegu od 0 to 255. Kako je 1 na toj skali vrlo mala vrijednost kreirat će boju koja je skoro crna.

4. `final` varijable i konstante

U matematici i znanosti , *konstanta* je naziv koji označava određenu vrijednost poput ‘π’ koja je praktično ime za broj 3.14159… . Java ima svoju vrstu konstanti. One su varijable koje pod (a) ne pripadaju nijednom objektu i pod (b) ne mijenjaju svoje vrijednost jednom kad su postavljene.

Riječ `static` u deklaraciji varijable pokazuje da varijabla ne pripada nijednom objektu. Postoji slična ključna riječ koja kaže da će vrijednost varijable biti postavljana samo jednom i dalje se neće mijenjati. To je ključna riječ `final`. Ova ključna riječ nije ograničena na statičke varijable već se može koristiti i s varijablama instance, lokalnim varijablama i čak s varijablama koje su parametri metoda..

Deklariranjem varijable kao `final` pokazujete svima pa i sebi da će varijabla tijekom postojanja imati istu vrijednost. Još bitnije je da će prevodilac detektirati svaki pokušaj da se promijeni vrijednost tako deklarirane varijable i isti prijaviti kao grešku.

Što se tiče Java terminologije varijablu koja je deklarirana kao `static` i `final` nazivamo **konstantom.** Postoji standardna konvencija da se nazivi Java konstanti pišu velikim slovima.Npr. Math.PI je konstanta koja označava vrijednost 3.14159…, a Math.E označava vrijednost 2.71828….

Jedna od uobičajenih upotreba konstanti u javi je kada imamo mali skup vrijednosti s očitim nazivom, ali koje ne pripadaju nijednom tipu podataka u Javi.

Npr. prilikom definicije klase Robot odredili smo da postoji polje koje će zapisivati da li je olovka spuštena ili nije. Bilo je prikladno koristiti boolean varijablu. Međutim postoji i alternativni pristup koji uključuje upotrebu konstanti i koji se vrlo često koristi u Java programiranju.

Mogli smo koristiti cjelobrojno polje i za vrijednost kad je olovka spuštena koristiti 1, a za podignutu olovku 0. Da ne bi koristili brojčane vrijednosti u programu definirat ćemo u programu dvije konstante na slijeeći način:

```
public static final int GORE = 0, DOLJE = 1;
```

Ovu deklaraciju stavit ćemo na početak definicije klase Robot. Polje `jeDolje` koje zapisuje da li je olovka dolje ili nije može biti promijenjeno na slijedeći način:

```
private int pozicijaOlovke;
```

Primjetite da se radi o `int` varijabli. Međutim nemojmo o njoj razmišljati kao o varijabli koja sadrži cjelobrojnu vrijednost. Vrijednost će biti ili `GORE` or `DOLJE`.

Testiranje da li je olovka dolje u metodi `pomakni` , bit će na slijedeći način

```
if (pozicijaOlovke == DOLJE) …
```

Da bismo promijenili položaj olovke koristit ćemo novu metodu umjesto metoda `olovkaGore` i `olovkaDolje`.

```
public void postaviOlovku(int pozicija)
{  pozicijaOlovke = pozicija;
}
```

U klasi koja koristi objekt tipa Robot `r`, olovku objekta `r` postavit ćemo pozivom slijedećeg izraza.

```
r.postaviOlovku(Robot.DOLJE);
```

Primjetite da se izvan klase Robot , konstata mora nazvati punim nazivom `Robot.DOLJE` ili `Robot.GORE`.

5. Rekurzija

Ništa nas ne sprečava da napišemo metodu koja poziva samu sebe. Takve metode nazivaju se **rekurzivne metode**. Postoje određeni tipovi problema koji se rekurzivnim metodama rješavaju mnogo jednostavnije i elegantnije nego nerekurzivnim.

Da bismo vidjeli što se događa kad se izvršava rekurzivna metoda potrebno je razmotriti što se općenito događa prilikom izvršavanja metoda. Kada se Java program izvršava neke metode se izvršavaju. U stvari u jednom trenutku vjerovatno se izvršava *nekoliko* metoda.

Npr. analiziramo primjer 1, tj. robot program koji crta spiralu. Kakvo je stanje u trenutku kad se crta linija, odnosno koje su metode u tom trenutku pozvane.
Crtanje se obavlja ako je pozvana metaoda `paint` . Ta metoda pripada objektu Spiral. Ona ima jedan parametar i to Graphics objekt kojega je kreirao windows manager.

Pretpostavimo da se izvršava slijedeći izraz:

```
s.paint(g)
```

gdje je s Spiral objekt, a g je Graphics objekt.

Pogledajmo `paint` metodu.

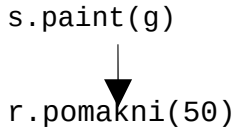
```
public void paint(Graphics g)
{  Robot r = new Robot();
   r.olovkaDolje();
   for (int i = 0; i < 100; i++)
   {  r.pomakni(i);
      r.lijevo(36);
   }
}
```

Pošto robot crta lijiju Java u tom trenutku izvršava naredbu:

```
r.pomakni(i)
```

za neku vrijednost od i. Pretpostavimo da je i jednako 50.

Slijedeća slika pokazuje nam stanje:

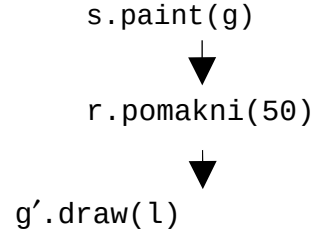


te nam pokazuje dase trenutno izvršava `s.paint(g)` , te je tametoda pozvala `r.pomakni(50)`, koja se također izvršava. Izvršavanje metode `s.paint(g)` će biti zaustavljeno dok metoda `r.pomakni(50)` ne završi.

Ako pogledate definiciju metode `pomakni,` vidjet ćete da jedina naredba koja radi s crtanjem je naredba:

```
g'.draw(l)
```

gdje je g' objekt tipa Graphics2D, a l je objekt tipa Line2D. Tako se lanac metoda produžava na slijedeći način:



Primjetite da se sva tri poziva izvršavaju u isto vrijeme. Prvi poziv (`paint`) pozvao je izvršavanje drugog (`pomakni`) te će čekati dok se metoda `pomakni` izvršava. Međutim i ta metoda je pozvala metodu `draw` . Metoda draw je metoda iz Java biblioteke i ona dalje poziva neke metode iz biblioteke....

Općenito u svakom trenutku dok se izvršava Java program postojat će lanac metoda koje se izvršavaju :

Važna osobina Jave i većine drugih programskih jezika je da svaki put kad se metoda pozove kreira se za nju poseban skup varijabli parametara i lokalnih varijabli.
To je istina i ako neki metod se ponavlja više puta u lancu pozvanih metoda !

Pretpostavimo da se metoda M pojavljuje više puta u lancu pozvanih metoda. Svaki put kad je metoda M pozvana ona radi na novom skupu varijabli. Na taj način se mogu izvršavati više pozvanih metoda M bez međusobne interferencije. Ova vrsta lanca s jednom metodom koja se pojavljuje više puta u lancu događa se kad rekurzivni metod zove samog sebe.

Promotrimo slijedeći program. Program čita dva broja i prikazuje rezultat koji je prvi broj podignut na potenciju drugoga. To se računa o funkciji `potencija(n,p)`. Primjetite da je metoda `potencija` rekurzivna.

PRIMJER 3

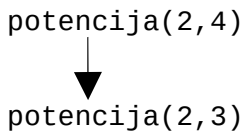
```
public class PotencijaProg

{  /* Čita dva cijela broja, x i y,
      gdje je y>=0. Ispisuje vrijednost x
      na potenciju y. */

   public static void main(String[] args)
   {  ConsoleReader in =
         new ConsoleReader(System.in);

      System.our.print
         ("Unesi bazu(cijeli broj): ");
      int i1 = in.readInt();
      System.our.print
         ("unesi potenciju(cijeli broj): ");

      int i2 = in.readInt();
      System.our.println
         (i1 + " na potenciju " + i2 +
          " = " + potencija(i1,i2));
   }

   /* Ako je p>=0, vrati n na potenciju p. */
   private static int potencija(int n, int p)
   {  if (p == 0)
         return 1;
      else
         return potencija(n,p-1)*n;
   }

}
```

Rekurzivna metoda

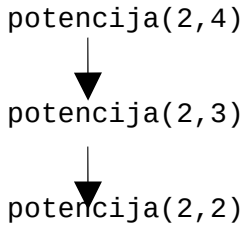
Pretpostavimo da je program pokrenut i da će korisnik unijeti vrijednosti 2 i 4.

Prvo se poziva metoda `main` .Zatim korisnik unosi dvije vrijednosti. Onda Java poziva metodu `potencija (2,4)` da bi se izračunao odgovor.

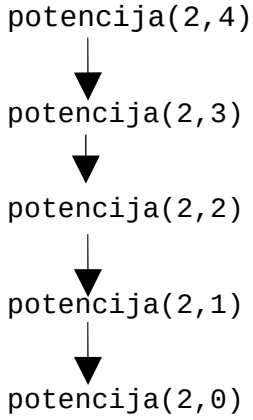
Od ove točke skoncentrirati ćemo se na lanac pozvanih metoda. Za prvi poziv metode `potencija` vrijednosti parametara bit će n = 2 i p =4. U tom slučaju pošto je p>=0 bit će izvršena druga grana tj. Java će vratiti (`return`) vrijednost koja će se dobiti izvršavanjem izraza `potencija(n,p-1)*n`, a to ovdje znači izraza `potencija (2,3).`
Lanac u tom trenutku izgleda ovako:



Sad se u pozivu metode događa isto pa se poziva opet ista metoda s novim parametrima tj. `potencija(2,2)`.

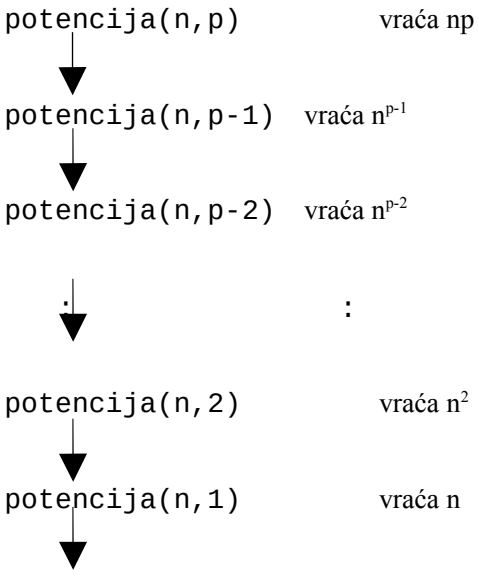


Slično kad se `potencija(2,2)` je izvršava, zvat će `potencija(2,1)`. Kad se `potencija(2,1)` izvršava, zvat će se `potencija(2,0)`.



Ovdje se lanac prekida jer kad se bude izvršavala metoda `potencija(2,0)` , bit će izvršena prva grana, tj. prva `return` naredba. Tako će `potencija(2,0)` vratiti vrijednost 1. To će biti korišteno u izvršavanju metode execution of `potencija(2,1)`, koja će vratiti vrijednost 1∗2, tj. 2. To se koristi u izvršavanju metode `potencija(2,2)`, koja će vratiti vrijednost 4, itd.

Općenito ako je izvršen izraz `pow(n,p)` , gdje je p≥0, kompletan lanac će sadržavati p+1 poziv metode `potencija`. Krajnji poziv vraća 1. Svi ostali vraćaju n puta vrijednost vraćena od slijedećeg. Slijedi da početni poziv `potencija(n,p)` vraća n^p :



`potencija(n,0)` vraća 1

Naravno računanje potencije može se obaviti pozivom odgovarajuće funkcije iz bibliotetke funkcija. Može se napisati i slijedeći nerekurzivni kod:

```
int odgovor = 1;
for (int i = 0; i < p; i++)
   odgovor = odgovor * n;
return odgovor
```

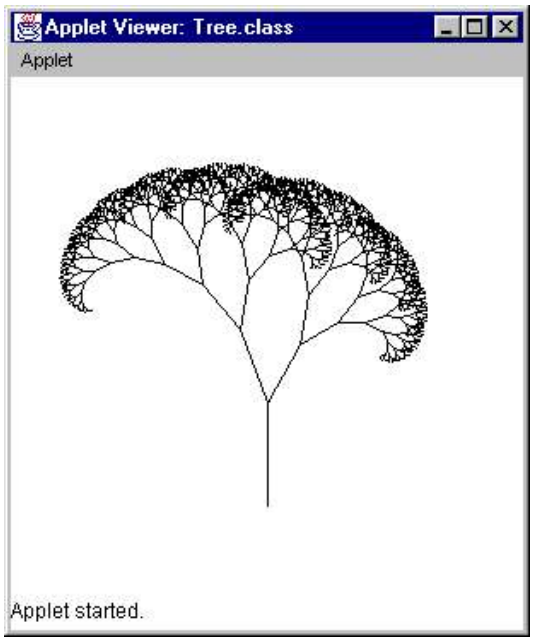
Općenito svaka rekurzivna metoda može biti napisana na nerekurzivan način. Međutim postoje slučajevi kad je rekurzivna verzija značajno jednostavnija.

Slijedi primjer programa koji je mnogo jednostavnije izvesti rekurzivnom metodom. Program koristi Robot objekte za crtanje stabla.

Program sadržava metodu nazvanu `crtajStablo`. Njena je specifikacija:

`crtajStablo(size)`
Nacrtaj stablu. Veličina grane je zadana s parametrom `veličina`.

Slijedi applet s nacrtanim stablom pozivom metode `crtajStablo(60)`.

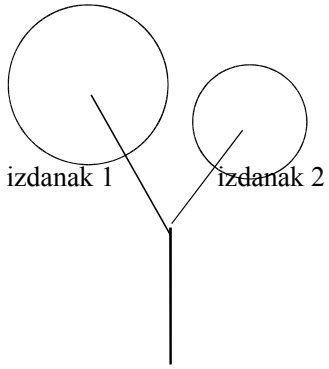


PRIMJER 4

```
import java.applet.Applet;
import java.awr.Graphics;

/* Koristi robota za crtanje fraktalnog stabla. */

public class Tree extends Applet

{  Robot robi;
   //Robot koji crta stablo.

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

```
    public void paint(Graphics g)
    {  robi = new Robot();
       robi.pomakni(-100);
       crtajStablo(60);
    }

    /* Koristi robota robija za crtanje stabla.
       veličina = veličina grane.
    */
    void crtajStablo(double veličina)
    {  if (veličina < 1) return;
       robi.olovkaDolje();
       robi.pomakni(veličina);
       robi.olovkaGore();
       robi.lijevo(20);
       crtajStablo(veličina*0.75);
       robi.desno(50);
       crtajStablo(veličina*0.65);
       robi.lijevo(30);
       robi.pomakni(-veličina);
    }
}
```

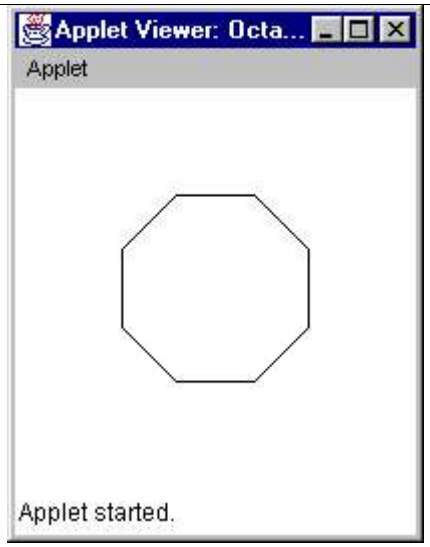
Metod radi na slijedeći način. Ako je veličina grane premalena metod ne radi ništa. Ako nije, nacrtat će granu i dodati dva nova izdanka. Izdanci su razmaknuti za neki kut i oba predstavljaju nova stabla. Izdanci se crtaju pozivom iste metode `crtajStablo` .



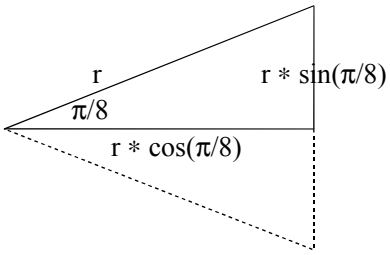
Nije jednostavan odgovor napitanje : Kada koristiti rekurziju ?
Stablo nam daje ključ. Rekurzije se najčešće koriste za programiranje i rad s rekurzivnim strukturama podataka. (npr. direktoriji !)

6. Zadaci

1. Napišite klasu Robot te je iskoristite za crtanje spirale, stabla i slijedećeg lika (opcionalno)



PRIMJEDBA. Ovaj lik traži malo znanja trigonometrije. Pretpostavimo da je r udaljenost od centra osmerokutnika do jedne od stranica. Onda dojnji trokut pokazuje da je duljina jedne od stranica osmerokutnika 2*r*sin(π/8). Udaljenost koju robot mora prijeći od centra do sredine stranice je r*cos(π/8).



Pazite kod upotrebe metoda `Math.sin` i `Math.cos` jer ove metode primaju kao argument radijane a ne stupnjeve

Nazovite applet osmerokutnik.java.

8. NIZOVI I LISTE

Sve do ove točke programi koje smo pisali mogli su pohraniti samo malu količinu podataka. Razlog tome je što varijabla može spremiti samo jedan primjerak neke informacije, a vrlo je zamorno za svaki slijedeći primjerak informacije istog tipa definirati još jednu varijablu.
U realnom svijetu programi rade s tisućama čak milionima primjeraka podataka.
U ovom poglavlju susrećemo se nizovima (*array*) te uopćenjem nizova koje je realizirano sučeljem List *(List interface)*.

SADRŽAJ

1. Nizovi *(arrays)*

Nizovi u Javi su **posebna vrsta objekata**. Sastoje se od nekog broja **elemenata** . Elementi niza su slični poljima objekta, ali nemaju individualne nazive. Umjesto toga oni su numerirani: 0, 1, 2, ... Broj nekog elementa naziva se njegov **indeks**.

Nizovi mogu biti mali pa imati dva ili tri elementa (čak nijedan), ali mogu biti i veliki s tisućama elemenata. Element niza može biti bilo koja varijabla ili objekt, ali svi elementi niza moraju biti istog tipa.

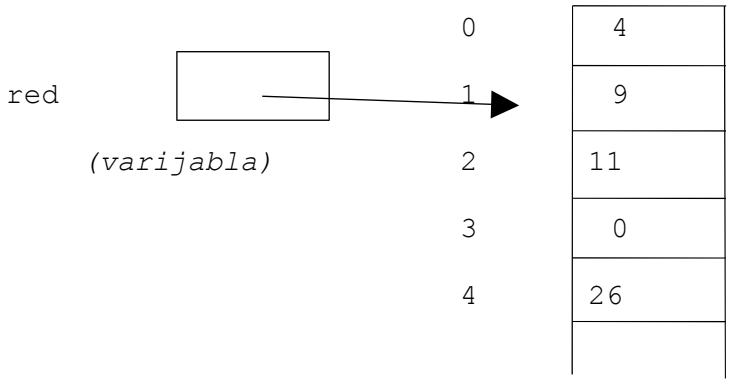
Npr. želite imati niz s elementima tipa `int` . Takav niz se označava s `int[].` Možemo imati i nizove drugih tipova podataka npr. niz `String[]` čiji elementi sadržavaju reference na objekte tipa String, ili niz `Robot[]` koji sadržava reference na objekte tipa `Robot` , itd.

Broj elemenata u nizu naziva se duljina niza *(length)*. Jednom kad je niz kreiran, njegova duljina je fiksna. Iako možete mijenjati vrijednosti pohranjene u elemente niza, nije moguće dodavanje novih elemenata niti uklanjanje već postojećih elemenata.

Ako je s `a` označen neki niz onda **`a.length`** označava njegovu duljinu. Primijetite malu razliku u odnosu na to kako smo dobivali duljinu stringa (`s.length()`).

Da bismo referencirali na bilo koji objekt potrebna nam je referenca na taj objekt. To vrijedi i za nizove. Pretpostavimo da varijabla `red` sadrži referencu na niz tipa `int[]` duljine 5.

Sadržaj memorije bi bio npr.:



Ako `red` označava niz, onda elementu s indeksom i pristupamo kao `red[i]`. Taj element možete tretirati kao bilo koju drugu varijablu.

Dodjeljivanje vrijednosti:

```
red[2] = 4;
```

Ispis vrijednosti:

```
System.out.println(red[4]);
```

Ispis prvih pet elemenata niza `red`.

```
for (int i = 0; i < 5; i++)
   System.out.println(red[i]);
```

Slijedi primjer deklaracije koja kreira niz varijabli tipa `int` zajedno s varijablom `red` koja referira na niz:

```
int[] red = new int[5];
```

Ovo znači: kreiraj varijablu `red` koja će referirati na niz tipa `int`, kreiraj niz tipa `int` duljine 5 i spremi vrijednost reference na taj niz u varijablu `red`.

Svih 5 elemenata niza poprimit će pretpostavljene (default) vrijednosti poput polja objekta. pretpostavljana vrijednost za brojčane elemente je 0, za reference na objekte je `null`.

Slijedi još jedan primjer deklaracije niza (ovaj put s inicijalizacijom);

```
int[] red = {4, 9, 11, 0, 26}
```

Moguće je i kreirati varijablu `red` bez kreiranja niza, npr.:

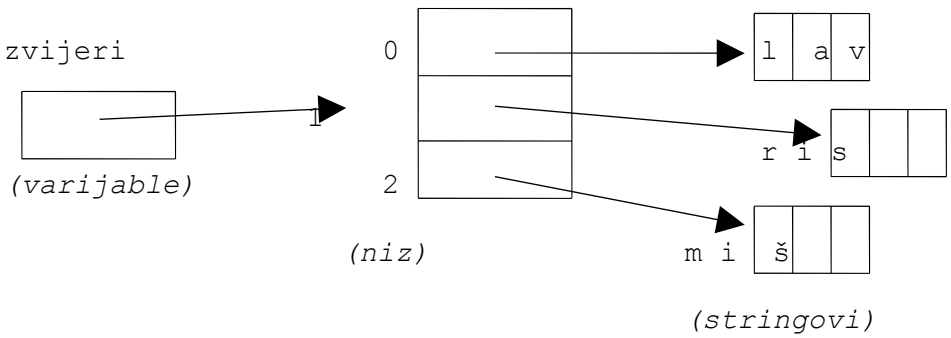
```
int[] red;
```

Ova naredba znači da će `red` biti korišten kao referenca na niz tipa `int`.

Slijedi još jedan primjer deklaracije:

```
String[] zvijeri = {"lav", "ris", "miš"};
```

Slijedeći dijagram pokazuje stanje memorije nakon izvršavanja prethodnog izraza:



U primjeru 1 iskoristit ćemo nizove za pohranu i prikaz podataka o količini padalina. Aplikacija koja slijedi učitava količinu padalina (u milimetrima) za svaki mjesec i onda to prikazuje u obliku histograma. Iako bi bilo ljepše aplikaciju realizirati kao applet (grafika) zbog jednostavnosti upotrebljena je DOS konzola.

Slijedi kompletan izlaz na ekranu realizirane aplikacije. Unos korisnika prikazan je podebljano.
Histogram je realiziran ispisom znaka X za svaka 2 milimetra padalina.

```
Unesi količinu padalina.
Jan: 55
Feb: 41
Mar: 36
Apr: 35
May: 47
Jun: 45
Jul: 60
Aug: 62
Sep: 50
Oct: 57
Nov: 65
Dec: 48

Jan  XXXXXXXXXXXXXXXXXXXXXXXXX
Feb  XXXXXXXXXXXXXXXXXXX
Mar  XXXXXXXXXXXXXXXXX
Apr  XXXXXXXXXXXXXXXX
May  XXXXXXXXXXXXXXXXXXXXXXX
Jun  XXXXXXXXXXXXXXXXXXXXX
Jul  XXXXXXXXXXXXXXXXXXXXXXXXXXXX
Aug  XXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Sep  XXXXXXXXXXXXXXXXXXXXXXXX
Oct  XXXXXXXXXXXXXXXXXXXXXXXXXXXX
Nov  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Dec  XXXXXXXXXXXXXXXXXXXXXXXX
     0   10   20   30   40   50   60   70   80   90
```

U realizaciji programa korištena je klasa ConsoleReader (objekt `input`)

PRIMJER 1

```
public static void main(String[] args)
{  String[] mjesec =
      {"Jan","Feb","Mar","Apr","May","Jun",
       "Jul","Aug","Sep","Oct","Nov","Dec"};
   ConsoleReader input =
      new ConsoleReader(System.in);

   /* Učitaj podatke količine padalina. */
   int[] padalinePodaci = new int[12];
   System.out.println("Unesi količinu padalina.");
   for (int i = 0; i < 12; i++)
   {  System.out.print(mjesec[i] + ": ");
      padalinePodaci[i] = input.readInt();
   }

   /* Prikaži histogram količine padalina. */
   System.out.println();
   for (int i = 0; i < 12; i++)
   {  System.out.print(mjesec[i] + "  ");
      for (int n = 0; n < padalinePodaci[i]/2; n++)
         System.out.print("X");
```

```
            System.out.println();
         }
         /* Iscrtaj skalu. */
         System.out.print("     ");
         for (int p = 0; p < 100; p = p+10)
            System.out.print(p + "   ");
         System.out.println();
      }
```

2. Argument linije naredbe (Command line arguments)

Od početka smo pisali programe koji su sadržavali main metodu s uvijek istom linijom:

```
public static void main(String[] args)
```

Iz novostečenog znanja vidimo da je argument metode main niz objekata tipa String. Što je sadržano u njima?

Kada pokrenemo program tipkanjem slijedećeg izraza:

```
java MyProg
```

Java interpreter će pogledati u datoteku MyProg.class očekujući da će naći klasu koja sadrži main metodu. Osim što traži datoteku s navedenim imenom Java provjerava da li postoji još koji dodatni string kao što je napisano u slijedećoj liniji:

```
java MyProg pas miš mačka lav
```

U prethodnoj liniji osim naziva datoteke prisutna su još 4 stringa. Java kreira niz od 4 stringa čiji sadržaj odgovara sadržaju dodatnih stringova upisanih u liniji naredbe.
Argumenti u liniji naredbe koristan su način prosljeđivanja dodatnih informacija programu u trenutku pozivanja.

Slijedi jednostavni primjer koji učitava bilo koji broj dodatnih argumenata iz linije naredbe i onda ih prikazuje u obrnutom redoslijedu.

PRIMJER 2

```
/*. */
public static void main(String[] args)
{  int howMany = args.length;
   if (howMany == 0)
   {System.out.println("Niste upisali nijedan dodatni argument.");
      return;
   }
   System.out.print(args[howMany-1]);
   for (int i = howMany-2; i >= 0; i--)
      System.out.print(" " + args[i]);
   System.out.println();
}
```

Pretpostavimo da ste pozvali navedenu main metodu u klasi nazvanoj MyProg i da ste ukucali slijedeći izraz u liniju naredbe:

```
java MyProg pas mačka miš
```

Tada će se na ekranu prikazati slijedeće:

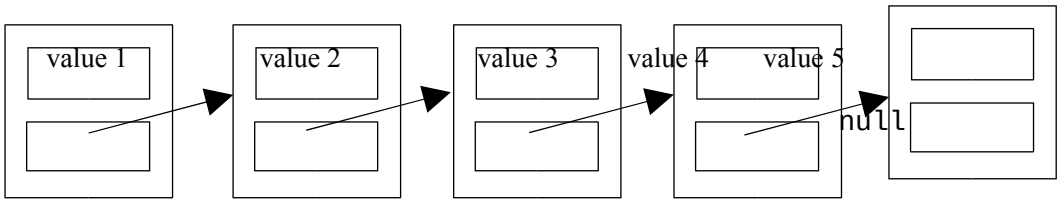
```
miš mačka pas
```

3. Sučelje List (List interface).

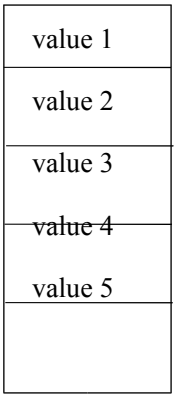
Osim nizova postoje i druge tehnike za pohranu velike količine podataka. Jedna od njih je vezana za pojam Liste (List) i predstavlja jedan od najčešćih i najfleksibilnijih oblika organizacije podataka u programiranju..

Pretpostavimo da je potrebno pohraniti veliki niz podataka. Moguće je definirati klasu DataList koja će se sastojati od jednog dijela u kojem su sadržani podaci i od drugog gdje je sadržana referenca na slijedeći DataList objekt. Taj element klase koji referencira na drugi objekt omogućava povezivanje u po volji duge liste.

Slijedeći dijagram pokazuje pet povezanih DataList objekata.



To može biti alternativa pohranjivanju u niz koje je prikazano na slijedećoj slici:



Sekvenca vezanih objekata naziva se vezana lista (**linked list)**.

Za neke situacije liste su mnogo prikladnije nego nizovi. To je slučaj kad je potrebno dodavanje elemenata između drugih elemenata. Mana lista je sporiji pristup slučajno odabranom elementu nego kod nizova.

postoje različiti načini implementacije lista i ako je u pitanju brzina programa potrebno je pažljivo razmotriti koju od implementacija upotrijebiti.

Java je prepoznala problem i zato je na raspolaganju sučelje (interface) nazvano **List** koji opisuje niz metoda upravljanje nizom vrijednosti koje su zajedničke svim implementacijama lista.

također su dostupne dvije implementacije navedenog sučelja. Jedna je zasnovana na nizovima, a druga na vezanim listama.

Ako koristite sučelje List možete deklarirati sekvencu varijabli kao tip List te na taj način koristiti metode koje to sučelje pruža. Međutim prilikom kreiranja objekta bit će potrebno navesti i tip implementacije.

List sučelje je veoma kompleksno i za kompletan prikaz potrebno je da pogledate u Java dokumentaciju. U ovom poglavlju dat ćemo pregled najčešće korištenih metoda sučelja List (List interface).

`add(v)`

Dodaj vrijednost `v` na kraj liste. `v` može biti referenca na bilo koji tip objekta (ne može biti primitivna vrijednost poput `int`). Novi članovi mogu se dodavati bez ograničenja i bez provjere da li ima dovoljno mjesta za njih.

`get(i)`

Vrati vrijednost elementa s indeksom `i` u listi. poput elemenata niza elementi liste imaju indekse 0, 1, ... Vraćena vrijednost bit će referenca na tip Object. To stvari znači bilo koji tip reference na objekt i bit će ga potrebno kastirati (cast).

`remove(i)`

Ukloni element s indeksom `i` iz liste.

`size()`

Vrati broj pohranjenih elemenata u listi.

`indexOf(v)`

Vrati indeks prve pojave elementa vrijednosti `v` u listi. Ako se vrijednost `v` ne pojavljuje u listi onda vrati −1.

Mogu se koristiti slijedeća dva konstruktora za kreiranje objekata koji će sadržavati metode List sučelja.

`ArrayList()`

Kreiraj novi objekt tipa ArrayList koji ne sadržava nikakve vrijednosti. Kreirani objekt imat će sve metode sučelja List. Interno bit će implementiran korištenjem niza. (Nije vas briga kako će se objekt snalaziti s umetanjem, uklanjanjem elemenata ,...)

`LinkedList()`.

Kreiraj novi objekt tipa LinkedList koji ne sadržava nikakve vrijednosti. Kreirani objekt imat će sve metode sučelja List. Interno bit će implementiran korištenjem vezanih lista.

Tako možemo po potrebi konstruirati ili ArrayList objekt ili LinkedList objekt, ovisno o zahtijevanoj efikasnosti. Oba objekta posjeduju sve metode List sučelja.

Slijedi program koji čita niz riječi koje je korisnik unio i ispisuje ih u obrnutom redoslijedu. Program za pohranu upisanih podataka koristi Listu (ArrayList). Razmotrite što bi bilo komplicirano ako bi se ovaj problem riješio upotrebom niza.

PRIMJER 3

```
/* Čitaj listu riječi koju upiše korisnik (zadnja riječ je
  'kraj') i onda ih ispiši u obrnutom redoslijedu. */

public static void main(String[] a)
{  System.out.println("Unesi riječi! Jedna po liniji!");
   ConsoleReader in = new ConsoleReader(System.in);
   java.util.List wordList = new ArrayList();

   /* Pročitaj i spremi riječi u 'wordList'. */
   while (true)
   {  String word = in.readLine();
      if (word.equalsIgnoreCase("kraj"))
         break;
      wordList.add(word);
   }
```

```
        /*Ispiši riječi iz liste u obrnutom redoslijedu */
        for (int i = wordList.size()-1; i >= 0; i--)
           System.out.println(wordList.get(i));
    }
```

List sučelje definirano je u java.util paketu Java biblioteke. Za prevođenje programa potrebno je importirati slijedeći paket:

```
import java.util.*;
```

na nesreću List sučelje ima isti naziv kao i List klasa definirana u java.awt paketu koja označava objekt za upravljanje listana GUI-a. Stoga ako upotrijebite import naredbu:

```
import java.awt.*;
```

prevodilac će misliti da želite koristiti GUI List klasu, a ne sučelje List. Ako u programu upotrebljavate objekte iz java.awt paketa i želite koristiti List sučelje siguran način je korištenje punog naziva sučelja List:

```
java.util.List
```

Na nesreću u listama možemo pohranjivati samo reference na objekte. Ako u listu želite pohraniti niz primitivnih vrijednosti npr. `double` , ipak postoji zaobilazni put.
Moguće je vrijednost tipa `double` upakirati u klasu koja će od polja sadržavati samo taj podatak. Java biblioteka posjeduje klasu Double koja upravo radi navedenu stvar. ('wrapper' class.- klasa omotač). Za prebaciti `double` vrijednost `d` u Double objekt koji sadrži `d`, koristimo slijedeći izraz:

```
Double x=new Double(d)
```

Objekt stvoren ovim izrazom može se pohraniti u listu. Da bismo dobili natrag vrijednost iz objekta x koristimo metodu Double objekta :

```
x.doubleValue()
```

Slično se može koristi za ostale primitivne vrijednosti (npr. `int` , `boolean` ,...)

Metoda `get` koja se koristi za dobivanje vrijednosti pohranjene u listi vraća referencu na objekt. kako u listi mogu biti pohranjene reference na bilo koji tip objekta vraćena referenca je tipa Object. Kad je želite upotrijebiti, najčešće će biti potrebno izvršiti kastiranje. Ako ste pohranili u listu stringove onda kastirate natrag u String primjenom `(String)` operatora kastiranja.

Međutim ako referencu koristite na mjestu gdje može biti upotrebljena bilo koja referenca onda je kastiranje nepotrebno. Npr. metoda `System.out.println` se koristi za prikaz vrijednosti `wordList.get(i)` jer njen argument može biti bilo koji objekt.

kada su u pitanju reference na objekte Liste se upotrebljavaju češće nego nizovi.

7. Zadaci

Napiši program koji će učitati niz ocjena u rasponu od 0 do 10 sve dok korisnik ne ukuca riječ 'kraj'. Program prvo mora kreirati niz od 11 elemenata za vođenje statistike pojave svake od ocjena. Znači kad se pojavi određena ocjena onda element niza s indeksom istim kao ocjena povećava se za jedan. Na kraju treba ispisati histogram kako je to pokazano na slici koja slijedi.

```
 0  XX
 1  XXXXXX
 2  XXXX
```

```
 3  XXX
 4  XXXXXXX
 5  XXXXXX
 6  XXXXXXXXXXX
 7  XXXXXXXXXXXXX
 8  XXXXXXXX
 9  XXXXX
10  X
```

Nazovite program OcjenaBroj**.**

Napišite program koji koristi Listu za slijedeći problem. Program čita riječi koje korisnik ukuca i ako se ta riječ već prije pojavila napisat će "Ponovljena riječ". Program završava s radom kad se ukuca riječ "kraj".
Program će davati izlaz poput slijedećeg (podebljano je korisnikov unos)

```
ivo
ante
ante
Ponovljena riječ
jes
nou
ivo
Ponovljena riječ
još
kraj
```

Program mora raditi bez obzira na uneseni broj riječi.

Nazovite ovaj program Ponavljalo.

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

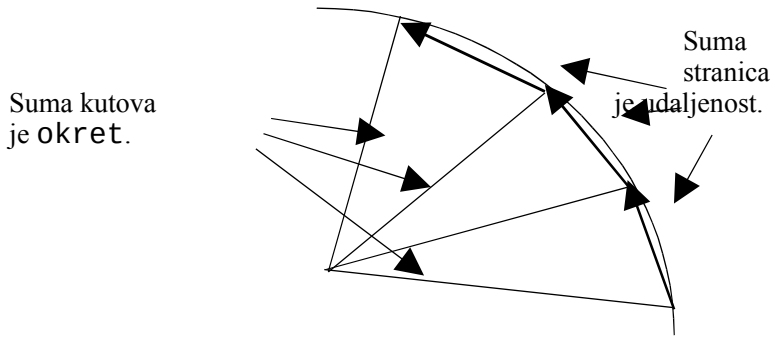
1. Nadogradnja klase (KrivuljaRobot)

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 `r` 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
```

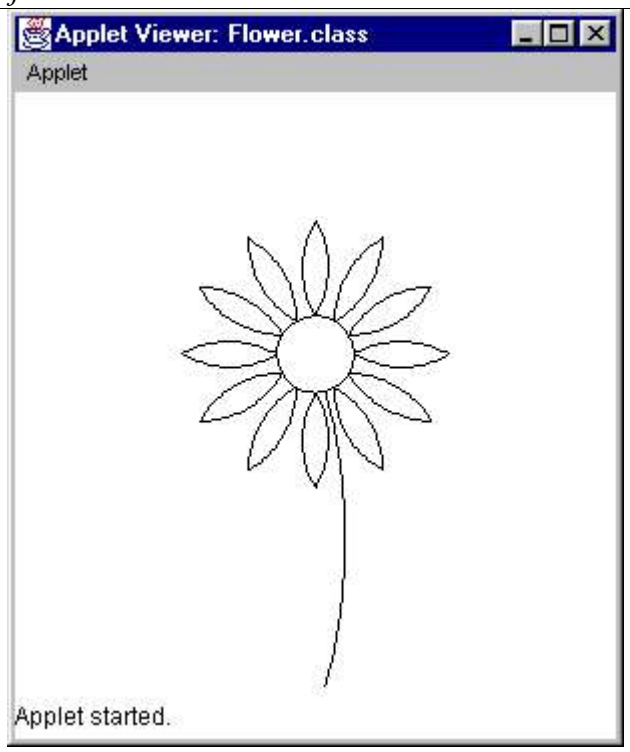
kaže Javi da uključi sve definicije koje se pojavljuju u originalnoj definiciji klase Robot. To znači da će objekt tipa KrivuljaRobot imati ista polja `x`, `y`, `smjer`, `jeDolje` kao i objekt Robot ; imat će iste metode `pomakni`, `lijevo`, `desno`, `olovkaGore` i `olovkaDolje` kao objekt Robot , plus 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();
```

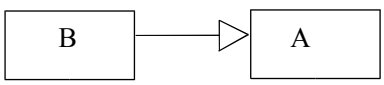
Međutim on se ne pojavljuje u definiciji klase KrivuljaRobot !
Ono što se ovdje događa je da nam Java sama osigurava konstruktor. Dakle, ako za neku klasu ne definirate konstruktor Java će vam sama osigurati konstruktor koji nazivamo **default** konstruktor. Taj konstruktor ne radi ništa osim što će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

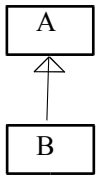
Tako u klasi Robot možemo napisati dvije metode pristupa:

```
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)

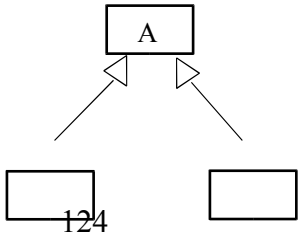
U prijašnjem odjeljku imali smo slijedeću situaciju. Počeli smo s definicijom klase A te zatim odlučili da je nadogradimo s definicijom klase B . To je predstavljeno sa slijedećim dijagramom.



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.

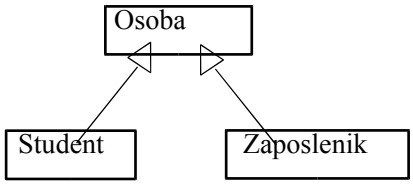


B1 B2

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`

`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);
}
```

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()
```

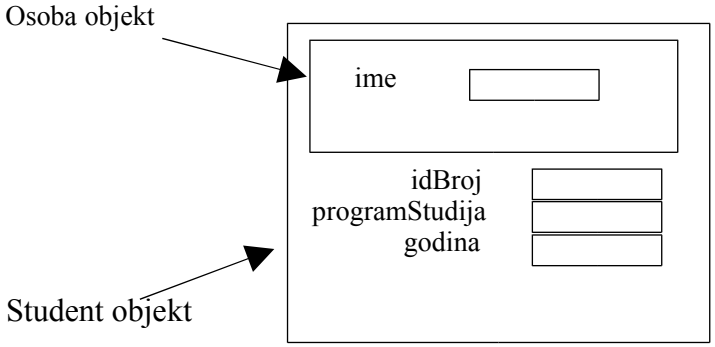
```
    {  return ime;
    }

    /* 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.
```

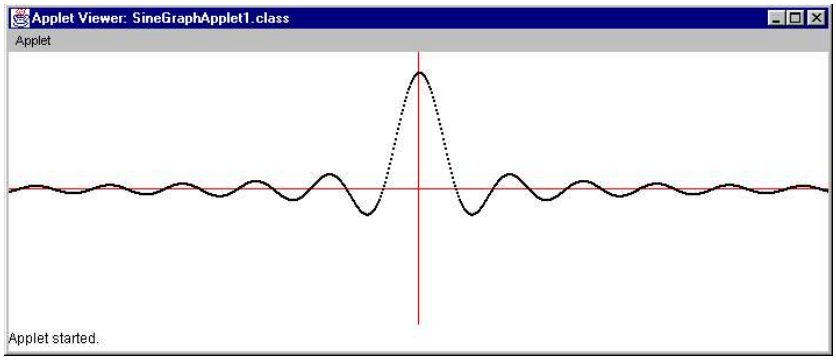
```
        */
        abstract public double function(double x);
    }
```

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:



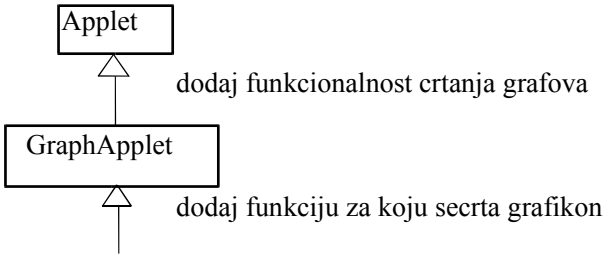
5. Stablo familije klasa

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.

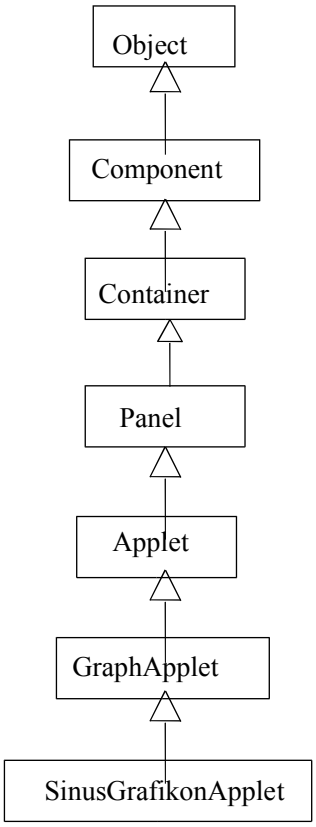
U prethodnom odjeljku išli smo još korak dalje. Prvo smo nadogradili Applet i izveli subklasu GraphApplet. Dodali smo funkcionalnost vezanu za crtanje grafova. Applet SinusGrafikonApplet nadogradio je GraphApplet naslijedivši funkcionalnost i Applet i GraphApplet klase te je još dodao dvije metode.

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 naslijeđivanje 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)).

10. Događaji (events)

Programi koje smo dosada pisali izvršavali su se tako su započinjali svoje izvršavanje od jedne točke i onda predvidljivim tokom prolazili kroz niz naredbi. Eventualno bi korisnik trebao unijeti neku informaciju, a na kraju rezultat su bili neki obrađeni podaci ispisani na neki od izlaza.
Međutim komercijalne aplikacije najčešće se na izvršavaju na opisani način. Pogledajte NotePad ili Word aplikaciju. Oba programa reagiraju na široki opseg korisničkih akcija, poput klikanja mišem po raznim elementima sučelja, pritiskanje neke od tipaka na tipkovnici, ...
U svakom slučaju aplikacija u kratkom vremenu odgovara na svaku od ovih akcija. Odgovor može biti npr. zatvaranje prozora, snimanje datoteke, dodavanje karaktera u tekst,...
Nakon što je izvršio određenu akciju program čeka da korisnik zada slijedeću. Ova vrsta programa izgleda poput sluge koji čeka korisnikove naredbe.

Ovo poglavlje je uvod u pisanje "programa sluge" korištenjem Java mehanizama koji omogućavaju ovakav rad. Posebno ćemo baviti vrijeme kako napisati interaktivni program koji odgovara na akcije miša (kretanje, klikanje) po ekranu. Ova vrsta programiranja zove se **programiranje na osnovu događaja(event-driven).**

Također ćemo vidjeti kako se piše program koji se izvršava po satu (sat driven). Ovaj program kreira objekt nazvan Timer . Timer generira pravilan niz događaja tj. otkucaja. Ostatak programa odgovara na svaki otkucaj nekom akcijom.

SADRŽAJ

1. Događaji

Java programi mogu reagirati na široki niz korisničkih akcija, npr. klikanje miša po ekranu, promjenu veličine prozora, pritisak tipke na tipkovnici i mnoge druge. Svaka od tih akcija naziva se **događaj** (**event**). Kada se dogodi neki od događaja, Java kreira objekt **događaja** (**event object**) koji sadržava reprezentaciju događaja.

Postoje različite vrste događaja. Npr. događaj koji se generira pritiskanjem klikanjem miša po ekranu pripada klasi MouseEvent. Postoji niz drugih akcija koje proizvode objekt klase MouseEvent: pritiskanje (lijevog) dugmeta miša, otpuštanje dugmeta miša, dvostruki klik, itd.

Pritiskanjem tipke na tipkovnici kreira se KeyEvent objekt, pritiskanje dugmeta kreira ActionEvent objekt, itd.

Jednom kad je objekt događaja kreiran , Java će ga proslijediti metodi koja je izabrana da obradi određenu vrstu događaja. Takve metode nazivamo **metoda slušač (listener method).**
Takva metoda mora za određeni događaj izvršiti odgovarajuću akciju. Npr. ako pišete metodu slušač za koja odgovara na klik mišem , **morate** je nazvati `mouseClicked`. Ona koja odgovara na pritisak tipke na tastaturi mora se zvati `keyTyped`, itd. Kompletna lista naziva dana je u odjeljku 3.
Svaka metoda slušač ima jedan parametar koji odgovara objektu događaja za koji metoda treba izvršiti određenu akciju. Metoda slušač može iz tog objekta dobiti detaljnije informacije o određenom događaju.

Tijek izvršavanja interaktivnog programa tj. programa čije je izvršavanje zasnovano na događajima može se podijeliti u dvije faze:

Faza uspostave. Ova faza može sadržavati kreiranje objekata potrebnih za slijedeću fazu, inicijalizaciju varijabli, itd.

Interaktivna faza. Tijekom ove faze program čeka na događaje koje inicira korisnik, sat ili operativni sustav. Čim se događaj dogodi izvršava se odgovarajuća metoda slušač. Često odgovor na događaj uključuje promjene na području prikaza aplikacije tj. ekranu. Ako program ne osigura odgovarajuću metodu slušača, događaj se jednostavno ignorira. Ako nema događaja program jednostavno čeka da se pojavi neki događaj.

Ako metoda slušač ne obavi zadatak dovoljno brzo može se dogoditi da se slijedeći događaj dogodi prije kraja njenog izvršavanja. Da bi se omogućila obrada ovako generiranih događaja svaki generirani objekt događaja sprema se u red. Ovaj red naziva se **red otpreme događaja** (**event dispatching queue**). Čim metoda slušač koja obrađuje prethodni događaj završi s obradom, slijedeći događaj (ako ga ima u redu) šalje se na obradu.

Jedna od posljedica korištenja reda otpreme događaja je da ako određena metoda zaplete u vremenski zahtjevnu obradu, svi ostali događaji moraju čekati. Stoga je potrebno da metode slušači obave svoje zadatke što je brže moguće. Inače će izgledati kao da je program blokirao i ne odgovara na korisnikove akcije.

Pretpostavimo da smo napisali `mouseClicked` metodu za koju želimo da se izvrši svaki put kad se klikne mišem u područje prikaza. Potrebno je učiniti slijedeće stvari:

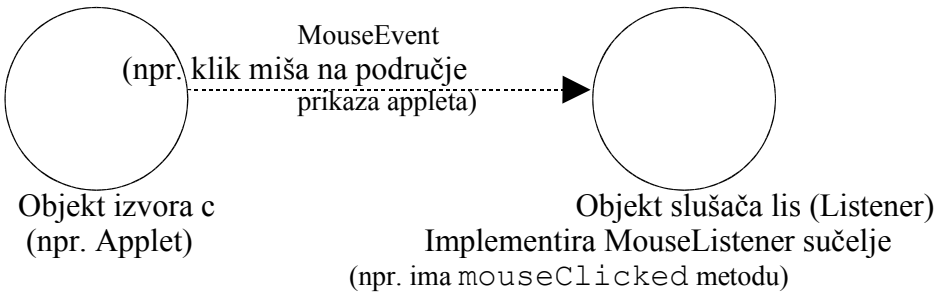
Metoda slušač ne može biti samostalna u memoriji. Ona mora biti pridružena s objektom koji nazivamo objekt slušač (**listener object)**. Možemo odabrati bilo koji implementira *MouseListener* sučelje(interface). To znači da taj objekt mora imati definirano slijedećih pet metoda: `mouseClicked`, `mouseEntered`, `mouseExited`, `mousePressed` i `mouseReleased`.

Potrebno je identificirati *izvor* događaja. To će biti objekt koji upravlja s komponentom sučelja na koju smo kliknuli. Ako smo kliknuli na područje prikaza appleta (kao u primjerima koji slijede) objekt izvor događaja je Applet objekt.

3. U fazi uspostave programa potrebno je uključiti izraz kojim se povezuje objekt slušača s objektom izvora. Taj postupak nazivamo **registracija** objekta slušača. Ako je c izvor događaja (npr. Applet objekt) tada će taj objekt imati niz metoda za registriranje slušača događaja (event listeners). Npr. ako je lis objekt slušača i želimo slušati klikanje miša na c onda se registracija ostvaruje slijedećim izrazom:

```
c.addMouseListener(lis);
```

Nakon izvršavanja prethodnog izraza svaki događaj miša uzrokovan klikanjem mišem po komponenti `c` bit će proslijeđen objektu `lis` , i bit će izvršena njegova `mouseClicked` metoda. Prevodilac će dopustiti da se lis koristi kao parametar `addMouseListener` metode samo ako lis objekt implementira MouseListener sučelje. To znači da lis **mora** imati `mouseClicked` metodu.



Moguće je registrirati više objekata slušača za neki od izvora tj. za svaku vrstu događaja posebno. Također je moguće odregistrirati objekte slušače kad više nisu potrebni.

Kad smo završili s registracijom potrebno je napisati kod koji će izvršiti određene akcije na osnovu generiranog događaja. Kod pišemo unutar metode slušača.

Postoji jedno nezgodno svojstvo predložene sheme. Naime **objekti slušači moraju definirati sve metode slušača** određenog sučelja, iako nam u nekom programu npr. treba samo jedna od njih.
Recimo da želimo definirati MouseListener objekt koji će imati `mouseClicked` metodu koja odgovara na svaki klik miša. Unutar te metode napisat ćemo kod koji izvršava određenu akciju. Iako nam ostale metode sučelja MouseListener nisu potrebne bit će ih potrebno definirati za ovaj objekt. Tijelo metoda ostavit ćemo praznim. Klasa će izgledati ovako:

```
public class MyMouseListener implements MouseListener

{  public void mouseClicked(MouseEvent e)
   {  Odgovori na klik mišem
   }

   public void mouseEntered(MouseEvent e) { }

   public void mouseExited(MouseEvent e) { }

   public void mousePressed(MouseEvent e) { }

   public void mouseReleased(MouseEvent e) { }
}
```

Ako želite izbjeći definiciju metoda koje ništa ne rade postoji jedan način. Java biblioteka posjeduje klasu koja se zove MouseAdapter. Klasa MouseAdapter samo definira prazne metode iz MouseListener sučelja (pet metoda). To ćemo iskoristiti tako da ćemo klasu slušač definirati kao nadogradnju klase MouseAdapter te unutar klase ponovo definirati stvarno potrebne metode (overriding). Ostale metode ne definiramo ponovo. Slijedeći ovaj pristup definicija klase MyMouseListener je slijedeća:

```
public class MyMouseListener extends MouseAdapter

{  public void mouseClicked(MouseEvent e)
   {  Odgovori na klik mišem predstavljen objektom e
   }
}
```

Primijetite da u gornjoj definiciji jedini naziv koji je po volji odabran je naziv klase slušača tj. `MyMouseListener.`

Za svako sučelje slušača koji ima dvije ili više metoda slušača u Java biblioteci nalazi se odgovarajuća adapter klasa.

Prije implementacije MouseListener sučelja ili nadogradnje MouseAdapter klase potrebno je uzeti u obzir jednu već spomenutu osobinu Jave. Java klasa može naslijediti samo jednu klasu. Stoga nije moguće napisati applet koji nasljeđuje MouseAdapter zato što applet uvijek nasljeđuje klasu Applet. S druge strane klasa može implementirati bilo koji broj sučelja.

PRIMJER 1:

Prvi interaktivni program koji ćemo napisati bit će applet. Njegov zadatak je vrlo jednostavan. Korisnik će klikanjem na dva mjesta na području prikaza appleta definirati liniju. Prvo će korisnik kliknuti na jednu točku i program će markirati poziciju crtanjem malog kružića. Onda će korisnik kliknuti na drugu točku i prvi kružić će biti izbrisan te će biti nacrtana linija definirana s dvije kliknute točke. Svaki daljnji klik bit će ignoriran.

Za funkcioniranje ovog appleta potrebno je definirati skup vrijednosti (varijabli) koje definiraju prikaz na ekranu. U ovom primjeru koristimo slijedeće vrijednosti:

1. Cjelobrojnu vrijednost 'faza' koja pokazuje u kojoj smo fazi izvršavanja programa. faza=0 pokazuje da mišem dosada nismo uopće kliknuli. faza=1 znači da je mišem kliknuto samo jednom i da je izabrana prva krajnja točka linije. faza=2 znači da je mišem kliknuto dva puta i da su poznate obje krajnje točke.

2. Koordinate x0, y0 prve točke. (Ove vrijednosti su poznate samo u fazi 1 i fazi 2)

3. Koordinate x1, y1 druge točke. (Ove vrijednosti su poznate samo u fazi 2)

Kako će metoda slušač (`mouseClicked`) odgovoriti na klik mišem ovisit će o trenutnoj fazi. Ako je faza=0, postavit će koordinate x0 i y0 na trenutne koordinate miša i fazu postaviti na 1. Ako je faza = 1, postavit će koordinate klika u x1 i y1 i postaviti fazu u 2. U fazi 2 neće raditi ništa.

Kako metoda `mouseClicked` dolazi do koordinata klika ? Koristi se vrijednosti koje su proslijeđene preko parametra `e`. Svaka metoda slušača ima jedan parametar koji reprezentira događaj koji se dogodio. U slučaju klika mišem, događaj će biti tipa MouseEvent te će imati dvije asocirane metode koje se nazivaju `getX` i `getY` i koje vraćaju vrijednosti koordinata pozicije na koju se kliknulo.

Primijetite u definiciji metode `mouseClicked` da se u dva slučaja kad metoda mijenja vrijednosti varijabli u kojima je definiran prikaz ne ekranu, nakon promjene poziva metoda `repaint()` čiji je zadatak osvježavanje sadržaja ekrana. Ovo osigurava da slika odmah prati klikove miša po ekranu.

Metoda `paint` mora također uzeti u obzir vrijednost varijable `faza`. Ako je ta vrijednost 0, nema ničega za ispisati. Ako je faza 1 metoda treba nacrtati mali kružić na poziciji (x0,y0). Ako je faza 2, treba nacrtati liniju od (x0,y0) do (x1,y1).

Metoda `init` treba učiniti samo jednu stvar. U njoj se registrira metoda slušač za Applet objekt. Drugim riječima tu se registrira Applet objekt (slušač) sa objektom izvorom tj. sa samim sobom. Koristi se slijedeći izraz:

```
addMouseListener(this);
```

Primijetite da applet referira na samog sebe s ključnom riječi `this`.

Koko smo prije spomenuli, klasa Applet ne može nadograditi klasu MouseAdapter jer već nadograđuje klasu Applet. Zbog toga mora implementirati MouseListener sučelje te zbog toga definirati svih pet metoda primijenjenog sučelja. Samo će metoda `mouseClicked` nešto raditi, a ostale definiramo kao prazne metode.

PRIMJER 1

(verzija u kojoj Applet objekt osluškuje klikove mišem.)

```
// Interaktivni applet koji crta jednu liniju.
// Korisnik klikne u jednu točku.
// Zatim u slijedeću točku.
// Nakon toga se crta linija koja spaja točke.

import java.awt.*;
import java.awt.geom.*;
import java.awt.event.*;
import java.applet.Applet;

public class LineApplet1 extends Applet
   implements MouseListener
```

```
{  private int faza = 0;
   // Faza = 0 prije nego što korisnik klikne prvu točku.
   // Faza = 1 nakon prvog klika.
   // Faza = 2 nakon drugog klika.

   private int x0, y0;
   // Koordinate početka linije.

   private int x1, y1;
   // Koordinate kraja linije.

   public void init()
   {  addMouseListener(this);
   }

   public void paint(Graphics g)
   {  if (faza == 0) return;

      Graphics2D g2 = (Graphics2D) g;
      if (faza == 1)
      {  /* Crtaj malu kružnicu s centrom u(x0,y0). */
         double radius = 5;
         Shape circle =
            new Ellipse2D.Double
               (x0-radius, y0-radius, 2*radius, 2*radius);
         g2.draw(circle);
         return;
      }

      /* (Pretpostavi faza == 2)
         Crtaj liniju od (x0,y0) to (x1,y1). */
      Shape line = new Line2D.Double(x0,y0,x1,y1);
      g2.draw(line);
   }

   /* MouseListener metode. */

   public void mouseClicked(MouseEvent e)
   {  if (faza == 0)
      {  x0 = e.getX();
         y0 = e.getY();
         faza = 1;
         repaint();
      }
```

```
            else if (faza == 1)
            {  x1 = e.getX();
               y1 = e.getY();
               faza = 2;
               repaint();
            }
            /* (Za faza == 2 ne čini ništa) */
        }

        public void mouseEntered(MouseEvent e) {}

        public void mouseExited(MouseEvent e) {}

        public void mousePressed(MouseEvent e) {}

        public void mouseReleased(MouseEvent e) {}

    }
```

2. Korištenje unutarnjih klasa (Inner Classes)

Kako smo vidjeli kod pisanja appleta koji osluškuje klikove miša, bilo je potrebno napisati prazne (dummy) definicije nekorištenih metoda MouseListener sučelja.

Bilo bi zgodno definirati odvojene objekte čiji će zadatak biti slušanje klikova mišem te da budu nadogradnja klase MouseAdapter. Tako bi izbjegli definiranje neželjenih metoda sučelja.

Na prvi pogled ovaj pristup ima nedostatak koji se očituje u tome da i applet i slušač (listener) trebaju imati pristup varijablama koje određuju prikaz na ekranu (npr. polja `faza`, `x0`, `y0`, `x1` i `y1` u prethodnom primjeru).

Ako te varijable držimo u appletu i označimo ih kao private što se obično čini onda bi trebali definirati metode kojima bi im mogli pristupiti iz metoda slušača tj. metode `mouseClicked` u prethodnom primjeru (sad u drugoj klasi).

Ako ih držimo u klasi slušača onda trebamo u toj klasi definirati metode s kojima im možemo pristupiti iz metode `paint` koja se nalazi u appletu.

Kako uzeli program postaje kompliciraniji.

Java omogućava rješenje ovog problema. Moguće je definirati dvije klase, recimo A i B, tako da B objekti mogu pristupati poljima objekta A, iako su ta polja označena kao private.

Potrebno je napisati definiciju klase B *unutar* definicije klase A. tako definirana klasa B se naziva **ugniježđena klasa** (**nested class**).

Ako B nije definirana kao statička klasa kažemo da je **unutarnja klasa (inner class)** klase A.

Već prije smo vidjeli da svaka varijabla klase A koja nije statička (polje) pripada objektu klase A. Svaka metoda klase A, ako nije statička, asocirana je s objektom klase A i može pristupiti poljima i metodama objekta.

Isto vrijedi i za objekte koji su tipa unutarnje (inner) klase B. Svaki B objekt je asociran s objektom A, i metode objekta B mogu pristupati privatnim poljima i metodama A objekta.

Ponekad se B objekt naziva pomoćnim objektom (helper) koji asistira asociranom A objektu.

U slučajevima kad je jedan objekt u potpunosti ovisan od objekta drugog tipa to može biti slučaj kad je potrebno da taj objekt definiramo kao unutarnju (inner) klasu.

To vrijedi za objekte slušača (listener) i često ih definiramo na navedeni način. Možemo promatrati objekt slušača kao pomoćni objekt koji asistira objektu koji je izvor događaja.

Postoji još jedna tehnika za dobivanje još kompaktnijeg koda. To je da definiramo klasu slušača unutar metode klase izvora. Tu tehniku u kojoj se kreiraju anonimne klase nećemo obraditi u ovim predavanjima.

Slijedi nova verzija programa iz primjera 1. Radi se o programu s istom funkcijom, ali drukčije strukture. Metode slušača uklonjene su iz appleta i stavljene u unutarnji (inner) objekt asociran s appletom. Kako je unutarnji objekt slušača nadogradnja MouseAdapter klase u njemu nije potrebno definirati sve metode MouseListener sučelja, već samo potrebne.

Izmjene u odnosu na prethodni program su podebljane. Unutarnja klasa nazvana je ClickListener.

PRIMJER 1 B
(verzija gdje je objekt slušača definiran unutarnjom klasom)

```
public class LineApplet2 extends Applet

{  private int faza = 0;
   // Faza = 0 prije nego što korisnik klikne prvu točku.
   // Faza = 1 nakon prvog klika.
   // Faza = 2 nakon drugog klika.

   private int x0, y0;
   // Koordinate početka linije.

   private int x1, y1;
   // Koordinate kraja linije.


   public void init()
   {  addMouseListener(new ClickListener());
   }
```

```
        public void paint(Graphics g)
        {  if (faza == 0) return;

           Graphics2D g2 = (Graphics2D) g;
           if (faza == 1)
           {  /* Crtaj malu kružnicu s centrom u(x0,y0). */
              double radius = 5;
              Shape circle =
                 new Ellipse2D.Double
                    (x0-radius, y0-radius, 2*radius, 2*radius);
              g2.draw(circle);
              return;
           }

           /* (Pretpostavi faza == 2)
              Crtaj liniju od (x0,y0) do (x1,y1). */
           Shape line = new Line2D.Double(x0,y0,x1,y1);
           g2.draw(line);
        }
        /******************Unutarnja (Inner) klasa ***************/

        public class ClickListener extends MouseAdapter

        {  public void mouseClicked(MouseEvent e)
           {  if (faza == 0)
              {  x0 = e.getX();
                 y0 = e.getY();
                 faza = 1;
                 repaint();
              }
              else if (faza == 1)
              {  x1 = e.getX();
                 y1 = e.getY();
                 faza = 2;
                 repaint();
              }
              /* (Ne čini ništa ako je faza == 2.) */
           }
        }
    }
```

Primijetite da se poljima `faza`, `x0`, `y0`, `x1`, `y1` u LineApplet objektu može pristupiti iz metode asociranog listener objekta.

Slijedi još jedan program koji se zasniva na crtanju linija. Program omogućava korisniku da nacrta linija koliko želi. Program predstavlja primitivnu formu programa za crtanja. Svaki put kad dodamo liniju pozvat će se `repaint` metoda što će izazvati poziv `paint` metode. Paint metoda obnavlja cijeli ekran i bit će potrebno imati pohranjene podatke za sve dotada nacrtane linije.

Prema tome bit će potrebno držati zapisano sve dotada nacrtane linije. Prirodan način je da ih pohranimo u ArrayList. Prilikom crtanja proći ćemo po svim elementima liste i nacrtati ih na ekranu.

Novi program će imati slijedeće varijable:

1. Cjelobrojnu vrijednost ‘faza’ koja pokazuje u kojoj smo fazi izvršavanja programa. faza=0 znači da je program spreman za prihvat koordinata prve točke linije. faza=2 znači da je program učitao prvu točku i da je sprema za drugu točku.

2. Koordinate `x0`, `y0` za prvu točku linije. (Ove vrijednosti bit će poznate kada faza bude 1)

3. Lista `lineList sa svim dosada definiranim linijama`.

Ideja programa je u biti slična kao iz prvog primjera. Korisnik klikne mišem. Odgovor je pozivanje metode slušača. Metoda slušača ažurira koordinate linija i poziva metodu `repaint`, koja onda poziva `paint`. `paint` koristi ažurirane sadržaje za obnavljanje sadržaja ekrana. Već prije smo vidjeli da se metoda `paint` poziva i nakon prekrivanja, maksimiziranja, minimiziranja, itd. prozora ekrana..

PRIMJER 2

```
// Interaktivni applet koji crta višestruke linije.
// Korisnik klika na prvi pa onda na drugi kraj.
// Zatim se crta linija koja spaja zadane točke.
// Postupak se ponavlja.

import java.awt.*;
import java.awt.geom.*;
import java.awt.event.*;
import java.util.*;
import java.applet.Applet;

public class ManyLinesApplet extends Applet

{  private int faza = 0;
   // If faza=1, (x0,y0) = početak slijedeće linije.
   // If faza=0, početak slijedeće linije nije kliknut.

   private int x0, y0;
   //Koordinate starta slijedeće linije.

   java.util.List lineList = new ArrayList();
   // Lista svih definiranih linija.

   public void init()
   {  addMouseListener(new ClickListener());
   }

   public void paint(Graphics g)
   {  Graphics2D g2 = (Graphics2D) g;
      if (faza == 1)
      {  /* Crtaj malu kružnicu s centrom u (x0,y0). */
         double radius = 5;
         Shape circle =
            new Ellipse2D.Double
               (x0-radius, y0-radius, 2*radius, 2*radius);
         g2.draw(circle);
      }

      /*Prikaži sve linije pohranjene u listi. */
      for (int i = 0; i < lineList.size(); i++)
      {  Shape nextLine = (Shape) lineList.get(i);
         g2.draw(nextLine);
      }
   }
```

```
    /*****************Unutarnja (Inner) klasa **************/

    public class ClickListener extends MouseAdapter

    {  public void mouseClicked(MouseEvent e)
       {  if (faza == 0)
          {  x0 = e.getX();
             y0 = e.getY();
             faza = 1;
          }
          else
          {  // pretpostavi da je faza == 1.
             int x = e.getX();
             int y = e.getY();
             Shape line = new Line2D.Double(x0,y0,x,y);
             lineList.add(line);
             faza = 0;
          }
          repaint();
       }
    }
}
```

3. Tablica klasa događaja (event classes) i tablica sučelja slušača (listener interfaces)

Prva kolona sadrži nazive različitih tipova događaja i odgovarajuća sučelja slušača. Primijetite da se naziv sučelja slušača i tipa događaja imaju početnu zajedničku osnovu te prvi ima nastavak "Listener" a drugi "Event".

Jedina iznimka je za tip događaja MouseEvents. To je zato što postoje dva sučelja za MouseEvents, nazvana MouseListener i MouseMotionListener.

Druga kolona sadrži nazive metoda sučelja slušača (listener interface). U svakom slučaju kad sučelje sadrži dvije ili više metoda, postoji i odgovarajuća klasa adaptera. Naziv adapter klase ima istu osnovu kao i naziv sučelja. (Listener zamijenjeno s Adapter)

Za registraciju objekta slušača koristi se metoda čiji je naziv "add"+naziv sučelja. Za uklanjanje se koristi "remove"+naziv sučelja

Event Class and Listener Interface	Listener Methods
ActionEvent ActionListener	actionPerformed
AdjustmentEvent AdjustmentListener	adjustmentValueChanged
ComponentEvent ComponentListener	componentHidden componentMoved componentResized componentShown

ContainerEvent	componentAdded
ContainerListener	componentRemoved
FocusEvent	focusGained
FocusListener	focusLost
ItemEvent	itemStateChanged
ItemListener	
KeyEvent	keyPressed
KeyListener	keyRepeased
	keyTyped
MouseEvent	mouseClicked
MouseListener	mouseEntered
	mouseExited
	mousePressed
	mouseReleased
MouseEvent	mouseDragged
MouseMotionListener	mouseMoved
TextEvent	textValueChanged
TextListener	
WindowEvent	windowActivated
WindowListener	windowClosed
	windowClosing
	windowDeactivated
	windowDeiconified
	windowIconified
	windowOpened

4. Korištenje sata (timer)

U prethodnim odjeljcima koristili smo događaje generirane mišem. Ovdje ćemo događaje generirati na drugačiji način.

Dosadašnji programi koje smo dosada vidjeli izvršavali su samo jednu sekvencu akcija. U slučaju aplikacije sekvenca je započinjala i bila određena `main` metodom.
Na njen tijek se moglo djelovati tako da se ovisno o unosu korisnika izvršavanja određeni niz naredbi. Međutim uvijek se radilo o jednoj sekvenci naredbi. Međutim Java može više od toga. Java nam omogućava pisanje programa u kojima se metode izvršavaju neovisno i paralelno svaka sa svojom sekvencom naredbi. Ako jedna sekvenca čeka na unos podataka od strane korisnika druga može neovisno o tome obavljati nekakav posao.

Takve različite sekvence izvršavanja naredbi nazivaju se **niti (threads).** Ovdje nećemo detaljnije objašnjavati niti tj. kako se kreiraju , izvršavaju , sinkroniziraju i zaustavljaju već ćemo u programu koristiti objekt koji će se izvršavati u neovisnoj niti.

Java biblioteka definira klasu objekata nazvanu Timer. Timer objekt posjeduje metodu `start`. Ako pozovemo metodu `start` Timer objekta, ona nastavlja svoje izvršavanje u novoj niti. Sve što ta neovisna nit radi je generiranje objekta događaja klase ActionEvent. U našem primjeri ti će se objekti kreirati u regularnom intervalu koji se zadaje prilikom kreacije Timer objekta.

Timer možemo zaustaviti na određeno vrijeme, nakon toga opet pokrenuti , itd.

Svaki događaj koji objekt Timer kreira stavlja se u red i bit će obrađen kada dođe na red. Događaji će biti proslijeđeni ActionListener objektu koji se registrirao na Timer. ActionListener sučelje ima samo jednu metodu koja odgovara na događaje i naziva se `actionPerformed`.

koristit ćemo tri metode objekta Timer i jedan konstruktor.

`start()`
Pokreni Timer. tj. pokreni nit koja generira ActionEvent događaje.

`stop()`
Zaustavi Timer.

`isRunning()`
Vrati true ako je Timer pokrenut. Inače vrati false.

`Timer(delay,listener)`
Kreiraj novi objekt Timer. Jednom kad je Timer pokrenut, počet će s generiranjem sekvence događaja (ActionEvent). Cjelobrojna vrijednost `delay` je vrijeme između dva sukcesivna događaja tipa ActionEvent. `listener` je ActionListener objekt koji će biti registriran od strane objekta Timer.

Koristit ćemo Timer da bismo proizveli animiranu sliku. Ponovo ćemo definirati Applet s pripadnim poljima koja definiraju prikaz. Osim toga definirat ćemo Timer koji će generirati seriju događaja (ActionEvent). Definirat ćemo ActionListener čija će metoda `actionPerformed` odgovarati na svaki od ActionEvent događaja. ActionListener će stalno u malim izmjenama modificirati sadržaj polja koja definiraju prikaz što će kao rezultat imati sliku koja se stalno mijenja. Ako interval objekta Timer (delay) smanjimo na dovoljno malu vrijednost netko tko gleda prikaz imat će utisak glatke animacije.

Svaka pojedinačna slika naziva se **okvir** (frame). Broj okvira prikazanih u jednoj sekundi naziva se brzina promjene okvira (frame rate). Za animaciju bit će nam dovoljno od 12 do 20 okvira u sekundi.

U primjeru koji slijedi animacija se sastoji od pravokutnika koji se pojavljuje na lijevoj strani, putuje na desnu stranu i na kraju iščezava na desnoj strani područja prikaza.

Metoda `actionPerformed` svaki put dodaje jedan na varijablu i onda poziva `repaint`. Varijabla je inicijalno postavljena na nulu. Zapravo radi se o brojaču koji broji broj okvira. Varijabla je nazvana `vrijeme` jer predstavlja i broj otkucaja Timer objekta.

Metoda `paint` računa položaj kvadrata na slijedeći način:

```
double x = startX + vrijeme*brzinaX;
double y = startY + vrijeme*brzinaY;
```

Kvadrat se kreira slijedećim izrazom.

```
Shape kvadrat =
   new Rectangle2D.Double(x,y,stranica,stranica);
```

Zadnji izraz je registracija MouseListener od strane appleta. Ako se klikne mišem na ekran slijedeća metoda će biti izvršena:

```
public void mouseClicked(MouseEvent e)
{  if (sat.isRunning())
      sat.stop();
   else
      sat.start();
}
```

Naredbe u ovoj metodi zaustavlja Timer ako je pokrenut i time se zaustavlja tok ActionEvent događaja. Kvadrat će stati na ekranu. Ako Timer nije bio pokrenut timer će se ponovo pokrenuti i animacija će se nastaviti. Slijedi kompletan applet.

PRIMJER 3

```
// Animirani applet.  Prikazuje mali kvadrat
// koji se lagano kreće preko područja prikaza.
// Klikni na područje prikaza za zaustavljanje kretanja,
// ili ako je zaustavljeno, klikni ponovo za pokretanje.

import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.applet.Applet;
import javax.swing.*;    // (potrebno za Timer)

public class KvadratFilm extends Applet

{  Timer sat;

   private int vrijeme = 0;
   // Trenutno vrijeme (i.e. trenutni broj okvira).

   private int brzinaOkvira = 20;
   // Broj okvira u sekundi

   private final double brzinaX = 4, brzinaY = 0;
   // Koliko se piksela lik pomiče u svakom okviru

   private final double startX = -20, startY = 100;
   // Pozicija lika za okvir 0.

   private final double stranica = 20;
   // Duljina stranice kvadrata


   public void init()
   {  int delay = 1000/brzinaOkvira;
      // 'delay' =vrijeme između dva uzastopna okvira.

      sat = new Timer(delay, new ClockListener());
      // 'sat'inicijalno nije pokrenut.

      addMouseListener(new ClickListener());
   }

   public void paint(Graphics g)

   {  Graphics2D g2 = (Graphics2D) g;
      double x = startX + vrijeme*brzinaX;
      double y = startY + vrijeme*brzinaY;
      Shape kvadrat =
         new Rectangle2D.Double(x,y,stranica,stranica);
      g2.draw(kvadrat);
   }

   /* Unutarnja klasa (inner class). */

   public class ClockListener implements ActionListener

   {  public void actionPerformed(ActionEvent e)
      {  repaint();
```

```
            vrijeme++;
         }
      }

      public class ClickListener extends MouseAdapter

      {  public void mouseClicked(MouseEvent e)
         {  if (sat.isRunning())
               sat.stop();
            else
               sat.start();
         }
      }
}
```

5. Zadaci

Napiši applet koji će crtati kružnicu na slijedeći način. Korisnik prvo klikne u jednu točku koja sad predstavlja centar kružnice. Zatim klikne u drugu točku koja predstavlja jedno od točaka na kružnici. Nakon toga se nacrta kružnica.

Napiši drugu verziju programa iz primjera 3. Razlika neka bude u tome da klik miša mijenja smjer kretanja kvadrata.

Uputa: u metodi mouseClicked promijenite predznak koraka kretanja.

nazivi applet AmoTamo.

11. Iznimke i tokovi (exceptions and streams)

Ovo poglavlje bavi se iznimkama tj. upravljanjem greškama te kako pisati i čitati s tipkovnice, datoteke, itd. U zadnjem dijelu dan je prikaz tehnike snimanja sadržaja objekata.

SADRŽAJ

1. Iznimke (Exceptions)

Java ima poseban mehanizam za upravljanje run-time pogreškama. Pretpostavite da pišete neki kod koji može uzrokovati pogrešku u tijeku izvršavanja programa. Npr. neka varijabla je trebala referirati na neki objekt, ali je u njoj vrijednost **`null`**. Ako preko takve reference pozovemo metodu objekta javit će se greška. Isto vrijedi i kad npr. pokušamo dijeljenje s nulom (cjelobrojne vrijednosti) ili pokušamo pristupiti elementu van granica niza. Bilo bi prekompliciranо svaki put provjeravati sadržaj varijabli. Stoga se upotrebljava druga tehnika. Program puštamo da se izvršava, a sustav u trenutku pogreške ***baca iznimku* (*throws an exception)*** koju trebamo obraditi. Iznimku možemo i sami generirati. Što se događa kad je iznimka bačena:

Kreira se objekt koji opisuje pogrešku. Takav objekt se obično naziva objekt iznimke exception object). (U stvari ovaj objekt pripada klasi Throwable, i može biti u subklasi Error ako se radi o ozbiljnoj sistemskoj grešci ili u subklasi Exception ako se radi o normalnoj run-time grešci.)

Interpreter zaustavlja izvršavanje tekuće naredbe i počinje tražiti catch blok koji je napisan da odgovori na točno taj tip greške. Ako interpreter ne može naći odgovarajući catch blok, program će se zaustaviti i bit će ispisana poruka o grešci u prozoru (DOS) konzole. U ispisu će biti naziv pogreške i popis svih metoda koji se trenutno izvršavaju. To ima nekog smisla za programera, ali korisnik programa ne zna što će s tim podacima. Program je pao !

`catch` blok je dio `try-catch` izraza koji ima slijedeću formu.

```
try                                 )
{  NAREDBE                  |       try blok
}                                   )
catch (EXCEPTION1 e1)               )
{  NAREDBE                          |
}                                   |
catch (EXCEPTION2 e2)               |   bilo koji broj catch blokova
{  NAREDBE                          |
}                                   |
:                                   )
finally                             )
{  NAREDBE                          |   opcionalni finally blok
}                                   )
```

Zadnji dio tj. `finally` izraz se često izostavlja, a bitno za njega je da se uvijek izvršava i to nakon izvršavanja try-catch blokova.

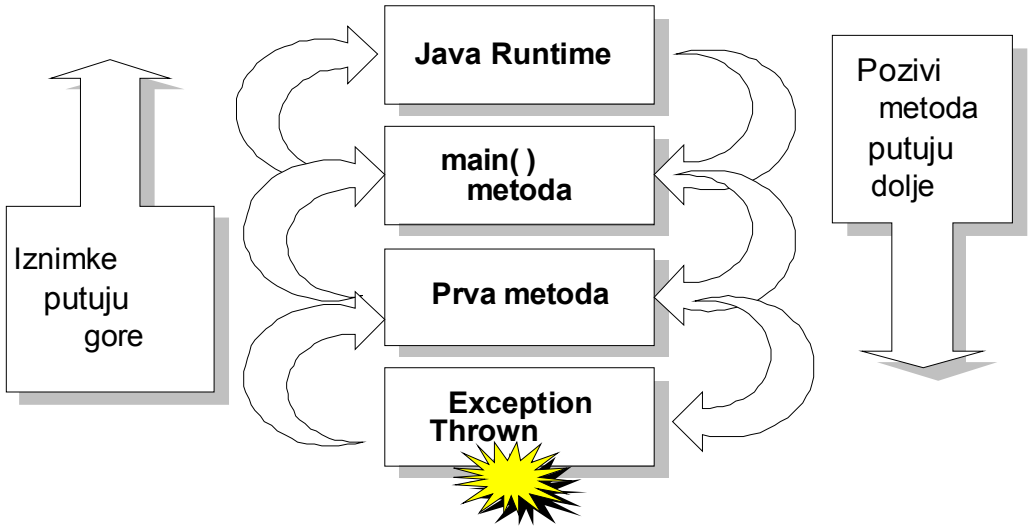
Kada interpreter izvršava `try-catch` izraz, prvo počinje s izvršavanjem naredbi u try bloku. Naredbe se izvršavaju normalnim slijedom i ako se nešto ne baci iznimku neće se izvršiti nijedan od catch blokova. Ako je dodan blok finally bit će izvršen nakon try bloka.

Ako se dogodi bacanje iznimke u try bloku interpreter će potražiti redom po svim catch blokovima da li koji od njih kao argument ima upravo generirani tip objekt iznimke. Ako nađe takav bit će izvršen. Nakon njega opet finally blok. Kako je logika izvršavanja finally bloka jednostavna i jasna odsad ćemo zanemariti mogućnost njegove upotrebe.

U ovome slučaju kažemo da je iznimka ulovljena (has been caught). Ako se iznimka ne ulovi ona se prosljeđuje pozivnoj metodi itd. sve dok se eventualno ne nađemo unutar nekog try bloka.

Slijedeća slika pokazuje smjer prosljeđivanja objekta iznimke:

08OOP13.WMF Figure 8-13 Exception propagation



Slijedi program koji sadržava `try-catch` izraz. To je nova verzija programa koji smo već prije vidjeli. Program učitava niz brojeva u pokretnom zarezu i prestaje s učitavanjem kada naiđe na riječ "kraj". Program zbraja brojeve i u isto vrijeme broji koliko je brojeva uneseno. Na kraju program izračunava prosjek koji se prikazuje na ekranu.

Program provjerava da li se u unesenoj liniji nalazi riječ "kraj". Za čitanje koristi metodu `readLine`. Ako uneseni string nije riječ 'kraj', program pokušava konvertirati string u broj koristeći metodu `Double.parseDouble`. Ako greškom unesete nešto što nije broj (ili riječ "kraj") bit će generirana iznimka NumberFormatException. Ako iznimku ne uhvatimo, program će se zaustaviti i bit će prikazana poruka o grešci.

U ovoj verziji programa umetnut je izraz `try-catch` s ciljem da se iznimka uhvati i izbjegne prekid programa. `try` blok sadržava poziv metode `Double.parseDouble`, i akciju koja slijedi ako je unesen predviđeni string. `catch` blok ispisuje jednostavnu poruku o grešci i program se nastavlja izvršavati.

PRIMJER 1

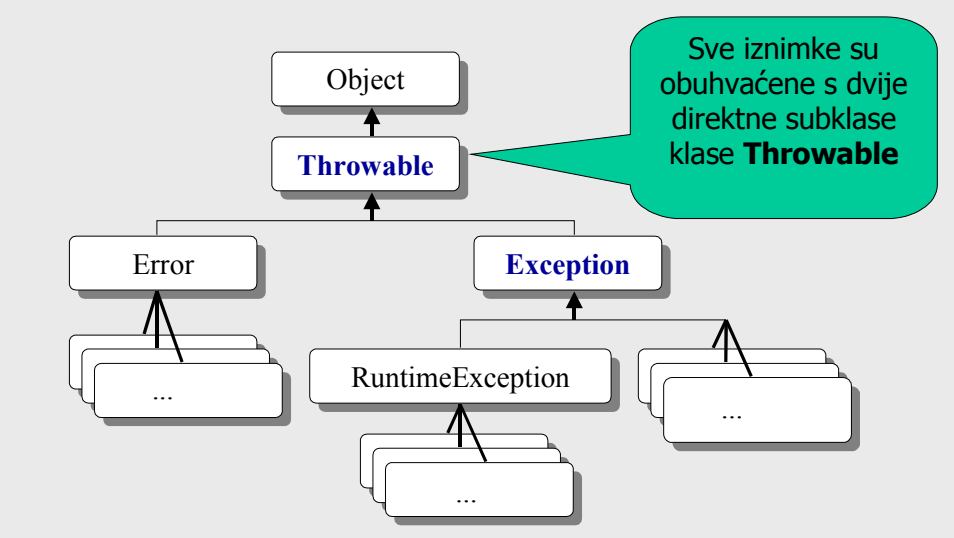
```
public class Prosjek1

{  /* Učitaj brojeve u pokretnom zarezu
      i ispiši njihov prosjek.
      (Verzija koja koristi ConsoleReader.)
   */
   public static void main(String[] args)
   {  ConsoleReader user =
```

```
            new ConsoleReader(System.in);

         /*Čitaj i dodaj vrijednosti,
            te broji ukupan broj vrijednosti. */
         double suma = 0;
         int koliko = 0;
         System.out.println("Unesi podatke.");
         while (true)
         {  String line = user.readLine();
            if (line.equals("kraj")) break;
            try
            {  double next = Double.parseDouble(line);
               suma = suma + next;
               koliko++;
            }
            catch (NumberFormatException e)
            {  System.out.println
                  ("Nerazumljiv ulazni podatak.");
            }
         }

         /* Ispiši prosjek. */
         if (koliko > 0)
            System.out.println
               ("Srednja vrijednost = " + suma/koliko);
         else
            System.out.println("Nema unesenih vrijednosti.");
      }
   }
```

Osim iznimki koje generira Java, možete i sami baciti iznimku. Potrebno je upotrijebiti **throw** naredbu. Opći oblik throw naredbe je:

```
throw new NumberFormatException();
        -- referenca na objekt iznimke ----
```

Primijetite da je za kreiranje iznimke upotrijebljen konstruktor. U Java biblioteci postoje različiti tipovi iznimki.:



Error iznimke :

Predstavljaju iznimke koje nisu predviđene da ih hvata programer. Postoje tri direktne subklase Error iznimke.

ThreadDeath: bačena svaki put kad se namjerno zaustavi nit (thread). Ako se ne uhvati nit završava s izvođenjem(ne i program).

LinkageError : ozbiljna greška unutar klasa programa (nekompatibilnost klasa, pokušaj kreiranja objekta nepostojeće klase.

VirtualMachineError – JVM greška

RunTimeException

Subklase:

ArithmeticException: Greška u aritmetici. Npr. cjelobrojno dijeljenje nulom.

IndexOutOfBoundsException : indeks izvan granica objekta koji koristi indekse npr. array, string, i vector

NegativeArraySizeException : Korištenje negativnog broja za veličinu niza.
NullPointerException : pozivanje metode ili pristup polju objekta preko null reference

ArrayStoreException : pokušaj dodjeljivanja objekta neodgovarajućeg tipa elementu niza (Array)

ClassCastException: pokušaj kastiranja objekta u nepravilan tip

SecurityException : prekršaj sigurnosti (Security manager)

2. Čitanje s tipkovnice

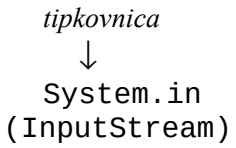
Kada u Java programu čitamo podatke s nekog ulaznog medija onda koristimo objekt koji upravlja s ulazom. Objekt se spaja na izvor podataka, npr. tipkovnicu ili tekstualnu datoteku. Da bismo čitali podatke koristimo metode tog objekta. Objekti tipa ConsoleReader koji je korišten u primjerima za čitanje podataka s tipkovnice je tipičan primjer takvog objekta.

Klase koje upravljaju s ulazom nazivaju se **InputStream** i **Reader**. Razlikuju se u tome što objekti rade kada se čitaju znakovi.

Objekt tipa **InputStream** vraća 8-bitni oktet (byte) svaki put nakon čitanja podatka.

Objekt tipa **Reader** vraća 16-bitne vrijednosti. To je standardan način reprezentacije znakova u Javi, koji se naziva **Unicode.** Unicode skup znakova obuhvaća alfabete većine svjetskih jezika.

Svaki put kad smo dosad kreirali objekt tipa ConsoleReader posredno smo koristili objekt tipa System.in. To je objekt tipa InputStream spojen direktno na tipkovnicu. Klasa InputStream posjeduje više metoda, ali samo jednu za čitanje i to metodu **`read`** koja čita jedan oktet (byte) ili unaprijed zadan niz byte-ova. Slijedeći dijagram pokazuje vezu tipkovnice i pripadne klase za čitanje podataka:

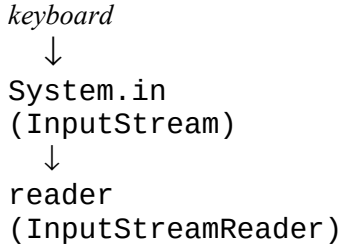


Java posjeduje i klasu **InputStreamReader** koja učitava podatke u Unicode formatu odnosno kao 16-bitne unicode znakove. Postoji konstruktor kojim je moguće pretvoriti InputStream u InputStreamReader. Referenca na objekt tipa InputStream je parametar konstruktora:

```
InputStreamReader reader =
   new InputStreamReader(System.in);
```

U konstruktor se može uključiti i drugi parametar za korištenje "nestandardnog" kodiranja 8-bitnog u Unicode prikaz. Inače će se obaviti standardna "default" konverzija u kodnu stranicu koja je trenutno u upotrebi na računalu.
Ovaj način konverzije jedne vrste ulaza u drugu preko konstruktora je tipična za Javu. Isto vrijedi i za klase izlaza.

Slijedi dijagram koji pokazuje povezanost klasa. Na kraju je rezultat 16-bitni karakter.



klasa InputStreamReader posjeduje metodu `read` koja vraća samo jedan znak. Pomoću ove metode moguće je napisati metode koje će čitati brojeve, riječi ,... Međutim bolja polazna točka bila bi klasa koja posjeduje metode za čitanje niza znakova odjednom. Postoje dvije vrste klasa koje to mogu učiniti:

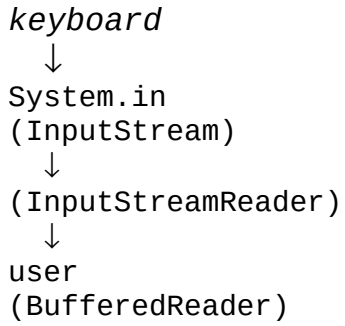
BufferedReader, **LineNumberReader**.

Obje klase posjeduju `readLine` metodu koja vraća uneseni niz znakova.

Moguće je konvertirati InputStreamReader u BufferedReader. Kao i obično to činimo pomoću konstruktora:

```
BufferedReader user =
   new BufferedReader
      (new InputStreamReader(System.in));
```

Ovaj izraz gradi kanal (‘pipeline’) koji izgleda ovako:



Objekt tipa BufferedReader pohranjuje podatke u **buffer.** To ubrzava unos podataka ako izvor podataka dozvoljava učitavanje odjednom cijelog niza znakova. (Konstruktoru se može proslijediti i veličina buffera. Inače će Java kreirati razumno velik buffer).

Slijedi još jedna verzija programa za računanje prosjeka. razlika je u tome da program ne koristi klasu ConsoleReader. Razlike među primjerima su podebljane.

PRIMJER 2

```
import java.io.*;
```

```
public class Prosjek2

{  /* Učitaj brojeve u pokretnom zarezu
      i ispiši njihov prosjek.
      (Verzija koja koristi BufferedReader.)
   */
   public static void main(String[] args)
      throws IOException
   {  BufferedReader user =
         new BufferedReader
           (new InputStreamReader(System.in));

      /*Čitaj i dodaj vrijednosti,
         te broji ukupan broj vrijednosti. */
      double suma = 0;
      int koliko = 0;
      System.out.println("Unesi podatke.");
      while (true)
      {  String line = user.readLine();
         if (line.equals("kraj")) break;
         try
         {  double next = Double.parseDouble(line);
            suma = suma + next;
            koliko++;
         }
         catch (NumberFormatException e)
         {  System.out.println
               ("Nerazumljiv ulazni podatak.");
         }
      }

      /* Ispiši prosjek. */
      if (koliko > 0)
         System.out.println
            ("Srednja vrijednost = " + suma/koliko);
      else
         System.out.println("Nema unesenih vrijednosti.");
   }
}
```

Primijetite promjenu u zaglavlju programa.

```
public static void main(String[] args)
   throws IOException
```

Ovo pokazuje prevodiocu da **`main`** metoda sadržava metodu , u ovom slučaju **`readLine`**, koja može baciti iznimku tipa **IOException** i koja neće biti uhvaćena jer u metodi **`main`** neće biti odgovarajućeg **`try-catch`** izraza da ga ulovi.

Iznimka IOException obuhvaća niz različitih grešaka koje se mogu javiti prilikom čitanja podataka i spada u grupu **checked** iznimki. Znači da je uvijek potrebno ili napisati izraz koji će je uhvatiti ili je potrebno dodati **`throws`** u zaglavlju metode:

```
throws IOException
```

Možemo dodati više tipova iznimki odvojenih zarezom.

Primijetite da kad smo koristili metode **`Double.parseDouble`** ili **`Integer.parseInt`**, nismo uključivali **`throws`** izraz za iznimku **NumberFormatException** koju bacaju navedene metode.

Razlog tome što **NumberFormatException** spada u **unchecked** iznimku. Nema potrebe da se uključuje **`throws`** izraz. Ideja je u tome da postoje iznimke koje se ne bi trebale pojavljivati ako je program dobro napisan.

Kada **`main`** metoda posjeduje **`throws`** izraz to je znak da navedena iznimka može terminirati program ostavljajući korisnika da gleda u ružnu pogrešku. To je normalno ako pišete eksperimentalni program za svoje potrebe, ali ne i za program za krajnjeg korisnika.
U tom slučaju sve iznimke je potrebno uhvatiti i obraditi.

Slijedi popravljeni program iz primjera 2. Dodan je drugi **`catch`**-blok za hvatanje iznimke **IOException** koju može baciti metoda **`readLine`**.

```
try
{  String line = user.readLine();
   if (line.equals("kraj")) break;
   double next = Double.parseDouble(line);
   suma = suma + next;
   koliko++;
}
catch (NumberFormatException e)
{  System.out.println
      ("Input not recognised.");
}
catch (IOException e)
{  System.out.println("Ulazna greška.");
   return;
}
```

Slijedi kompletna definicija **ConsoleReader** klase. U klasi se kreira **BufferedReader** kao što je to učinjeno u primjeru 2. Metoda `readLine` posjeduje kod za hvatanje bilo koje iznimke tipa IOExceptions. U odgovarajućem catch bloku nalazi se kod za izlaz iz programa.

U klasi nema nikakvog pokušaja hvatanje iznimki tipa **NumberFormatExceptions** koje mogu baciti `readInt` ili `readDouble`. One se šalju nazad u pozivnu metodu gdje ih korisnik može uhvatiti ako to želi (tip unchecked).

klasa ConsoleReader

```
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.IOException;

/**
   A class to read strings and numbers
   from an input stream.
   This class is suitable for
   beginning Java programmers.
   It constructs the necessary buffered
   reader, handles I/O exceptions, and
   converts strings to numbers.
*/

public class ConsoleReader
```

```
{  private BufferedReader reader;

   /** Constructs a console reader from
       an input stream such as System.in.
   */
   public ConsoleReader(InputStream inStream)
   {  reader =
       new BufferedReader
        (new InputStreamReader(inStream));
   }

   /** Read a line of input and
       convert it into an integer.
   */
   public int readInt()
   {  String inputString = readLine();
      int n = Integer.parseInt(inputString);
      return n;
   }

   /** Reads a line of input and convert it
       into a floating-point number.
   */
   public double readDouble()
   {  String inputString = readLine();
      double x =
         Double.parseDouble(inputString);
      return x;
   }

   /** Read a line of input.
       In the (unlikely) event of
       an IOException, the program halts.
   */
   public String readLine()
   {  String inputLine = "";
      try
      {  inputLine = reader.readLine();
      }
      catch(IOException e)
      {  System.out.println(e);
         System.exit(1);
      }
      return inputLine;
   }
}
```

3. Čitanje iz tekstualne datoteke

Za čitanje iz datoteke moguće je kreirati posebni tip **Reader** objekta , **`FileReader`** koji je spojen na datoteku. Za kreiranje veze, u **FileReader** konstruktoru navodimo naziv datoteke kao parametar. Npr. ako je potrebno čitati podatke iz datoteke data.txt koristimo slijedeći izraz:

```
FileReader input = new FileReader(″data.txt″);
```

Ako datoteku nije moguće pronaći, Java baca **`FileNotFoundException`**.
Ova iznimka je **checked** iznimka pa je potrebno koristiti **throws** izraz za svaku metodu gdje se ne hvata.

Klasa **FileReader** poput bilo koje klase tipa **Reader,** posjeduje **read** metodu koje vraća slijedeći unicode znak u obliku **int** vrijednosti, ali ne posjeduje **readLine** metodu.
Da bismo dobili **readLine** metodu, konvertiramo **FileReader** u **BufferedReader** korištenjem iste konstrukcije kao u prethodnom poglavlju.

```
BufferedReader data =
   new BufferedReader
      (new FileReader(″data.txt″));
```

Jednom kad smo kreirali objekt tipa **BufferedReader** koristimo njegovu metodu **readLine** na isti način kao što smo to činili kad su podaci dolazili s tipkovnice.
Jedina razlika je što je potrebno provjeriti da li su pročitane sve linije iz datoteke, odnosno da li smo stigli do kraja datoteke. To je jednostavno jer **readLine** vraća **null** kada stigne do kraja datoteke.
Kada je čitanje iz datoteke završeno potrebno je pozvati metodu **close**.

Slijedi finalna verzija programa za računanje prosjeka. Ova verzija čita podatke iz datoteke numbers.dat.

Primjer 3

```
// Program koji čita floating-point vrijednosti
// iz tekstualne datoteke 'numbers.dat',
// i proračunava njihov prosjek.

import java.io.*;

public class Prosjek3

{  public static void main(String[] args)
   {  BufferedReader data;
      try
      {  data = new BufferedReader
                  (new FileReader("numbers.dat"));
      }
      catch (FileNotFoundException e)
      {  System.out.println
            ("Datoteka numbers.dat nije pronađena.");
         return;
      }

  /*Čitaj i dodaj vrijednosti,
         te broji ukupan broj vrijednosti. */

      double suma = 0;
      int koliko = 0;
      try
      {  while (true)
         {  String line = data.readLine();
            if (line == null) break;
            try
            {  double next =
                  Double.parseDouble(line);
               suma = suma + next;
```

```
                koliko++;
             }
             catch (NumberFormatException e)
             {  System.out.println
                   ("Nerazumljiv ulazni podatak: " + line);
             }
          }
          data.close();
       }
       catch (IOException e)
       {  System.out.println(e);
          return;
       }

       /* Ispiši prosjek. */
       if (koliko > 0)
          System.out.println
             ("Srednja vrijednost = " + suma/koliko);
       else
          System.out.println("Nema unesenih vrijednosti.");
    }
}
```

Primjedbe.

1. Ako ne postoji datoteka bit će uhvaćena iznimka FileNotFoundException i program će biti prekinut uz odgovarajući ispis o pogrešci.
2. Vanjski **`try-catch`** blok, koji sadrži **`while`**-petlju, hvata **IOExceptions** koje mogu baciti metode **`readLine`** i **`close`**.
3. Unutarnji **`try-catch`** blok unutar petlje, hvata iznimke tipa **NumberFormatExceptions** koje može baciti metoda `Double.parseDouble`.
4. Program prestaje s čitanjem je **`line`** postavljen na **null**.
5. Ako bilo koja linija sadržava niz znakova koji ne predstavljaju broj bit će bačena iznimka

NumberFormatException. Program će u pripadnom catch bloku ispisati sadržaj te linije.

4. Pisanje u tekstualnu datoteku

Ako želite pisati u tekstualnu datoteku najbolje je koristiti **`PrintWriter`** klasu.
Ova klasa posjeduje **`println`** i **`print`** metode (poput `System.out` klase). **`PrintWriter`** objekt ne može se direktno spojiti na datoteku, već preko `FileWriter` objekta, koji prihvaća unicode karaktere i piše ih u tekstualnu datoteku.

Slijedi izraz koji ilustrira navedeno:

```
PrintWriter out =
   new PrintWriter
     (new FileWriter("data.txt"))
```

U datoteku pišemo korištenjem metoda **`print`** i **`println.`** Nakon što smo završili s ispisom možemo koristiti poziv `out.flush()`. To forsira spremanje sadržaja međuspremnika u datoteku. Na kraju pozivom **`out.close()`** zatvaramo vezu.

Java posjeduje opsežnu biblioteku klasa za čitanje i pisanje. Npr. postoje klase za komprimirano pisanje (zip), klase za rad s XML dokumentima, itd.