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.

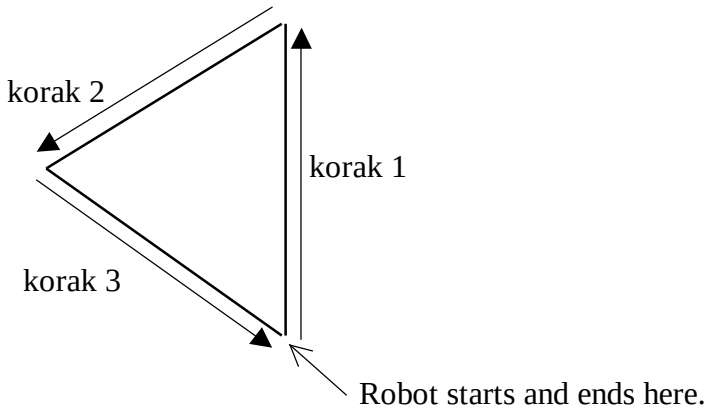
`r.pomakni(s)`	Miče robota naprijed za udaljenost s. s se mjeri u pikselima.
`r.lijevo(deg)`	Okreće robota ulijevo za deg stupnjeva.
`r.desno(deg)`	Okreće robota udesno za deg stupnjeva.
`r.olovkaDolje()`	Spušta olovku robota tako da robot piše svoj trag dok se kreće.
`r.olovkaGore()`	Diže olovku robota.

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

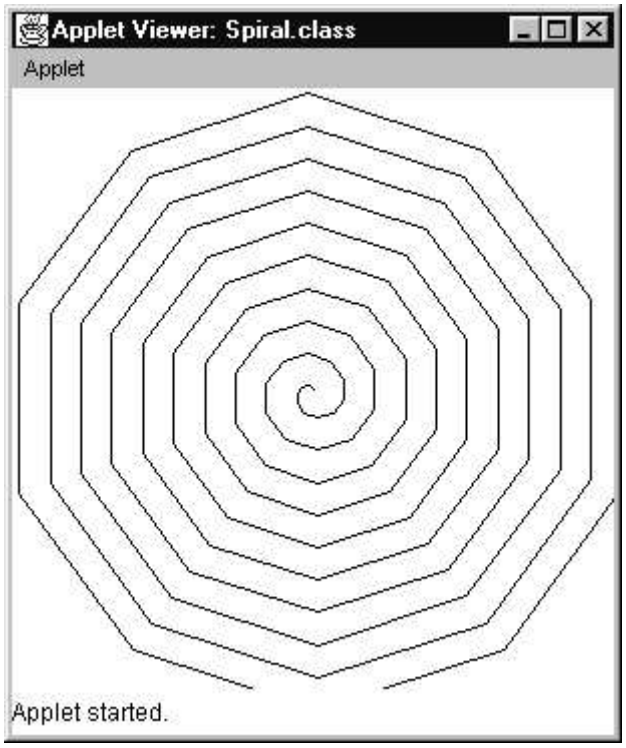
Svaki put kad se robot pokrene ide naprijed za udaljenost od 50(piksela) sa spuštenom olovkom crtajući liniju na ekranu. Svaki put kad stigne do kraja linije okrene se za kut od 120° nakon čega slijedi opet crtanje linije.



Slijedi još jedan robot program.

```
Robot t = 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.

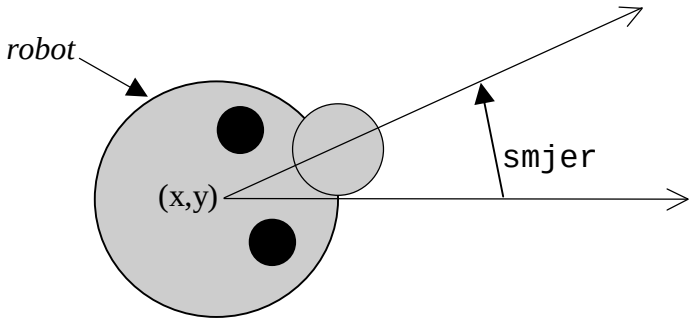
Tako će klasa `Robot` imati slijedeća polja:

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 programet 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.

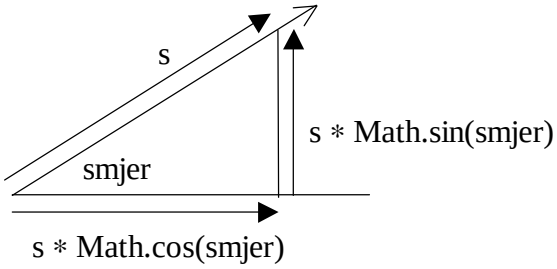
```
/** Podigni olovku gore.
*/
```

```
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.

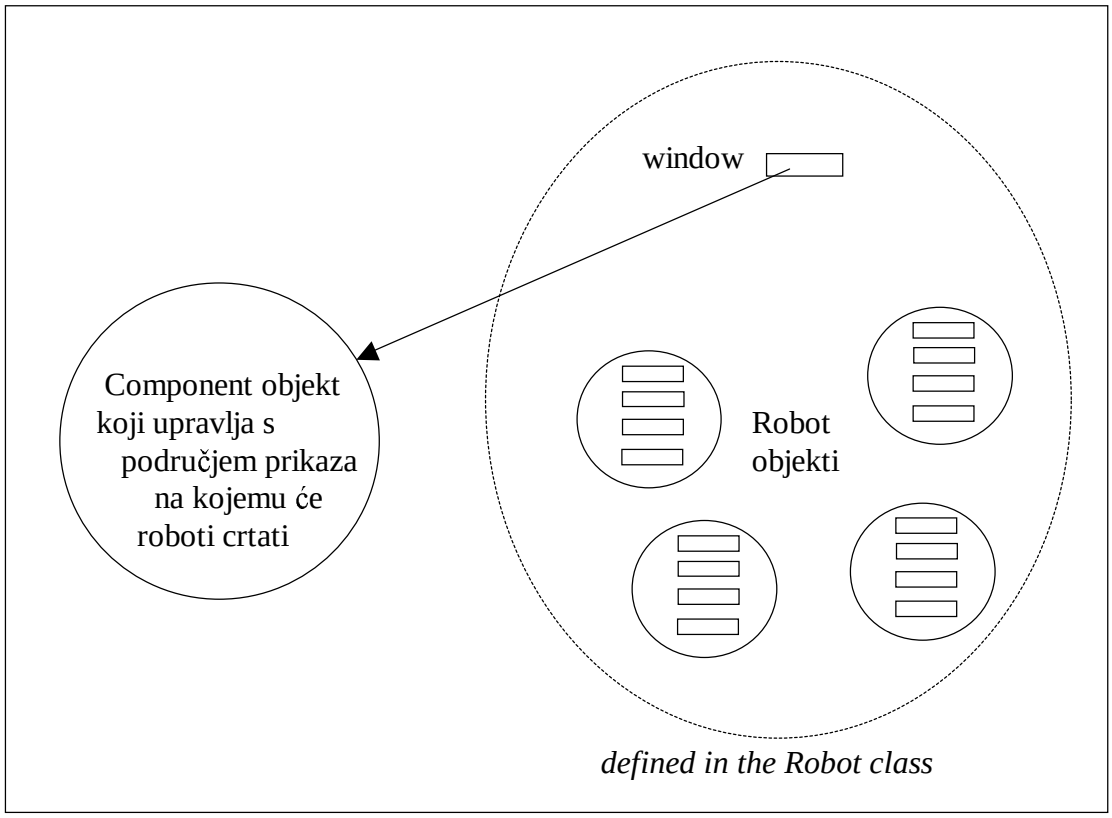
Ono što nam treba za kompletiranje defrinicije klase Robot je da svakom objektu tipa Robot damo pristup do objekta Component koji upravlja s područjem prikaza appleta. To ćemo učiniti tako da definiramo statičku varijablu `window`. Ova varijabla će sadržavati referencu na objekt tipa Component koji upravlja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

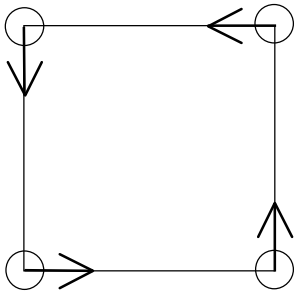
Slijedi applet koji koristi jednog robota za crtanje spirale koju smo već pokazali.

PRIMJER 1

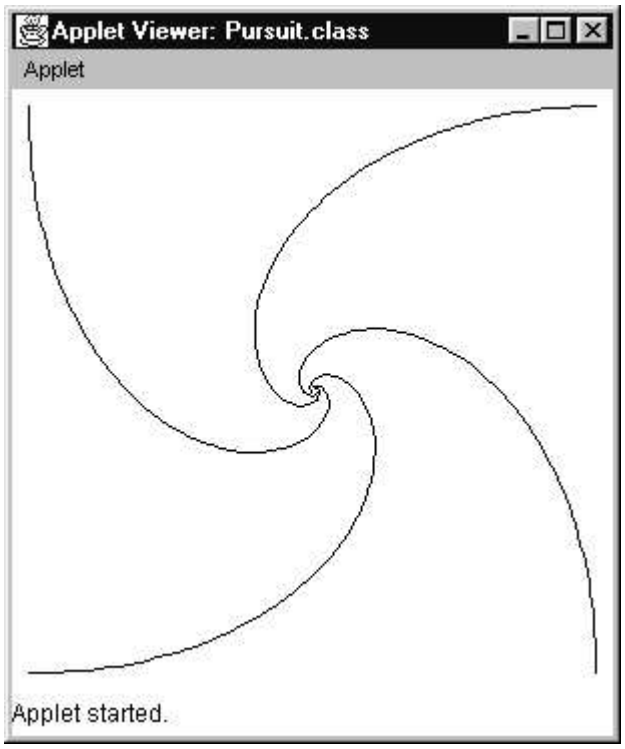
```
import java.applet.Applet;
import java.awr.Graphics;

public class Spiral extends Applet
{  public void init()
   {  Robot.setWindow(this);
   }

   /* Crtaj spiralu. */

   public void paint(Graphics g)
   {  Robot t = 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:



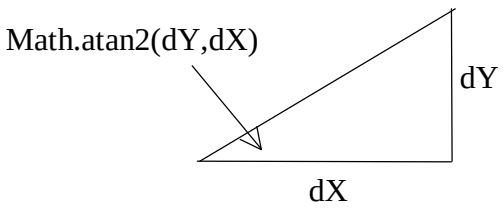
Metode instance koje smo dosada napisali nisu dovoljne za realizaciju ovog programa. Pomoću njih ne možemo ostvariti da jedan robot gleda u drugoga. Bit će potrebno dodati slijedeću metodu:

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

Here is the definition of `gledaGA`.

```
/** Turn Robot t 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 t0 = new Robot();
      t0.lijevo(45);
      t0.pomakni(diag);
      t0.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);
         t0.pomakni(korak);

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

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

         t0.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 t)
   {  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 `t`, olovku objekta `t` 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.

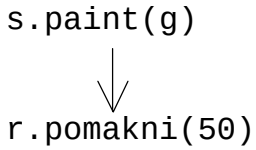
```
public void paint(Graphics g)
{  Robot t = new Robot();
   r.olovkaDolje();
   for (int i = 0; i < 100; i++)
   {  r.pomakni(i);
      r.lijevo(36);
   }
}
```

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:

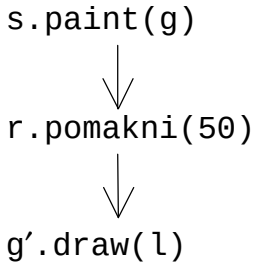


te nam pokazuje dase trenutno izvršava `s.paint(g)` , te je tametoda pozvala `r.pomakni(50)`, koja se također izvršava. Izvršavanje metode `s.paint(g)` će biti zaustavljeno dok metoda `r.pomakni(50)` ne završi.

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 :

C1 ⟶ C2 ⟶ C3 ⟶ …

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

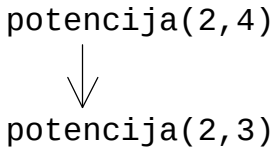
```
public class PotencijaProg

{  /* 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

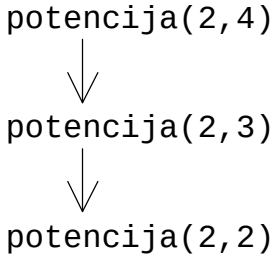
Pretpostavimo da je program pokrenut i da će korisnik unijeti vrijednosti 2 i 4.

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

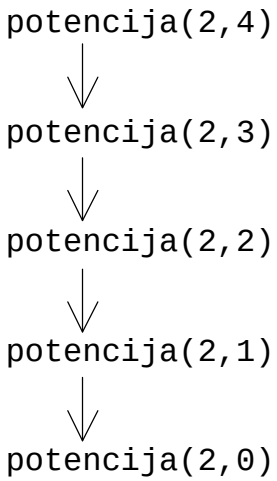
Od ove točke skoncentrirati ćemo se na lanac pozvanih metoda. Za prvi poziv metode `potencija` vrijednosti parametara bit će n = 2 i p =4. U tom slučaju pošto je p>=0 bit će izvršena druga grana tj. Java će vratiti (`return`) vrijednost koja će se dobiti izvršavanjem izraza `potencija(n,p-1)*n`, a to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

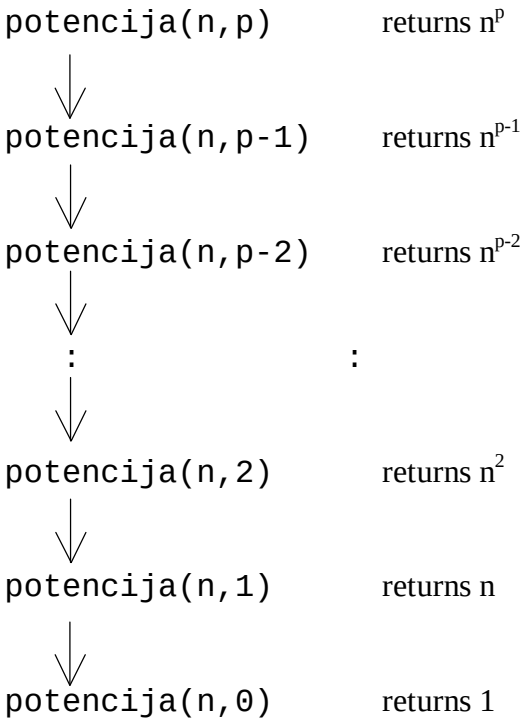


Slično kad se `potencija(2,2)` je izvršava, zvat će `potencija(2,1)`. Kad se `potencija(2,1)` izvršava, zvat će se `potencija(2,0)`.



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 :



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
```

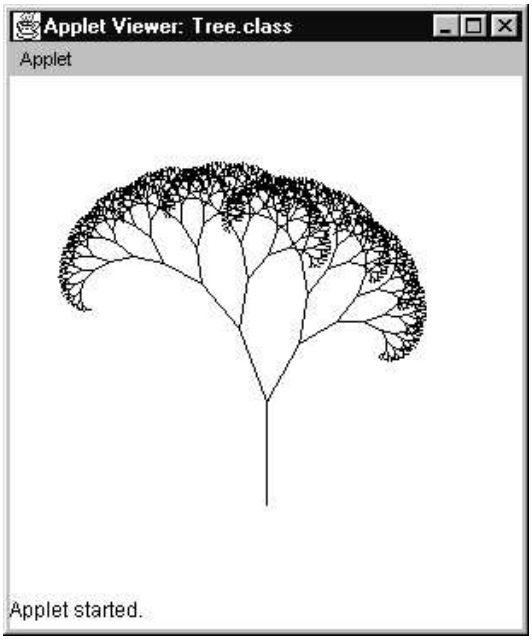
Općenito svaka rekurzivna metoda može biti napisana na nerekurzivan način. Međutim postoje slučajevi kad je rekurzivna verzija značajno jednostavnija.

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 `veli`č`ina`.

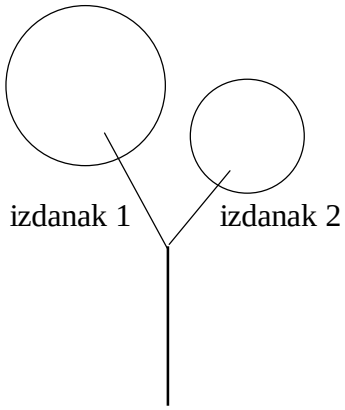
Slijedi applet s nacrtanim stablom pozivom metode `crtajStablo(60)`.



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` .

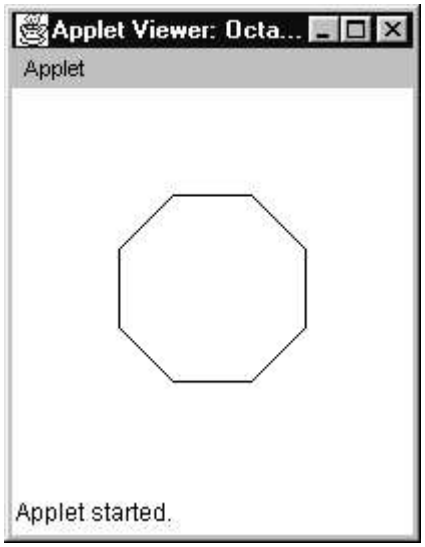


Nije jednostavan odgovor napitanje : Kada koristiti rekurziju ?
Stablo nam daje ključ. Rekurzije se najčešće koriste za programiranje i rad s rekurzivnim strukturama podataka. (npr. direktoriji !)

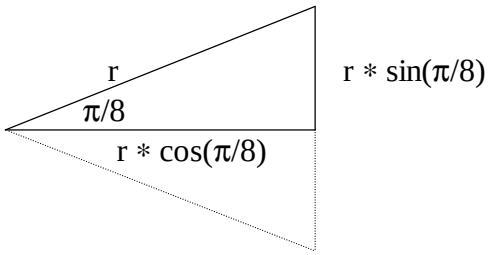
---

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 $2*r*\sin(\pi/8)$. Udaljenost koju robot mora prijeći od centra do sredine stranice je $r*\cos(\pi/8)$.



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

Nazovite applet osmerokutnik.java.