JAVA PROGRAMIRANJE

1. Uvod - kako početi programirati u JAVA jeziku

Cilj ovog poglavlja je napisati i pokrenuti jednostavnije Java programe.

SADRŽAJ

1. O predmetu

Sadržaj ovog predmeta je pisanje objektno orijentiranih programa u Java jeziku.

Java je :

- programski jezik
- vrlo velika biblioteka programskih rutina
- izvršna okolina za izvođenje programa.

Java biblioteka programskih rutina sastoji se od tisuća **klasa**. Postoje klase za rad s datotekama, klase za rad s 3D grafikom, klase za pristup bazama podataka, animaciju web stranica, itd. Moglo bi se reći da se u bibliotekama nalazi veći dio onoga što će vam ikada zatrebati u programiranju. Možda ste se dosad susreli s JavaScript jezikom za web stranice. JavaScript nije Java jezik ! Ovaj predmet se zbog ograničenog broja sati ipak mora ograničiti na osnove jezika te na manji dio osnovnih klasa.

Kao dodatnu literaturu možete koristiti knjige:

1. Beginning Java 2, SDK 1.4 Edition by, Ivor Horton ISBN:0764543652 Wrox Press 2003.
2. Java , Čukman, Tihomir , Alfej
3. Java: Programiranje za Internet i World Wide Web , Dario Sušanj, Znak, Zagreb, 1997.

Ako dosad niste programirali bit će potrebno uložiti značajan trud u savladavanje gradiva. Java jezik zajedno s ogromnim bibliotekama namijenjen je profesionalnim programerima. Štoviše Java biblioteke se stalno proširuju i mijenjaju .

Naučiti programirati u bilo kojem jeziku nije moguće bez praktičnog rada na računalu.

Stoga je od vrlo velike važnosti da primjere sa satova pokušate realizirati na računalu tijekom vježbi.

Ako ste dosad programirali u nekom programskom jeziku mogao bi vam prvi uvodni dio biti vrlo jednostavan. Međutim kako lekcije budu išle broj novih sadržaja i znanja će se povećavati.

Potrudite se zadatke odraditi sami. Nemojte kopirati programski kod od kolega. Osim što varate nastavnika varate i sami sebe.

2. Programi i programiranje

U ovome poglavlje prikazat ćemo jednostavni Java program i način kako ga pokrenuti na računalu.
Za početak ćemo ponoviti glavne dijelove računala i njihova svojstva:

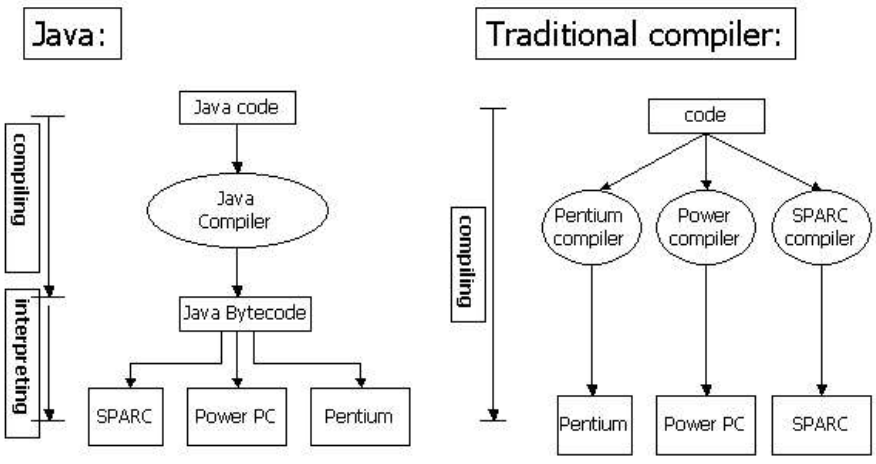
1. **Memorija** je dio računala u koji pohranjujemo informacije. Pohranjene informacije možemo pisati, brisati, obnavljati,... Informacije u memoriji su niz bitova dok na višoj razini predstavljaju brojeve, tekst, slike, glazbu,...
2. **Uređaji za ulaz/izlaz (I/O).** Informacije u računalo unosimo bilo tipkovnicom ili preko medija (disketa, CD-ROM ,mreža...). Informacije iz računala prikazujemo na ekranu ili pak šaljemo na neki od medija.
3. **Procesor** koji djeluje po instrukcijama **programa**. Program se sastoji od niza operacija koje procesor izvršava. To uključuje akcije poput izvršavanja proračuna, čitanja ili pisanja po memoriji, slanja podataka na izlazna sučelja procesora, ...

Program se piše u notaciji koja se naziva **programski jezik**. Svako računalo (procesor) ima svoj programski jezik koji nazivamo **strojni jezik** (machine code). Taj jezik je dizajniran s fokusom na elementarne operacije koje se obavljaju nad hardverom računala. Radi se o jednostavnim operacijama poput pisanja ili čitanja iz memorije, aritmetičkim operacijama nad registrima procesora, itd. Iako je teoretski svaki program moguće napisati koristeći strojni jezik to je vrlo teško čak i za jednostavne programe. U praksi se gotovo sve programiranje izvodi u jezicima koji su prilagođeni programeru. Takvi jezici se nazivaju **jezici visokog nivoa**.

Jezici koji se danas koriste u komercijalnoj upotrebi su C, C++,Java,C#,Perl,Phyton,Pascal(Delphi), Basic,Fortran....
Java jezik je jedan od najmlađih jezika. Prvi put se pojavio 1995. godine. Java 2 specifikacija jezika koju koristimo u ovom predmetu pojavila se 1998.

Rekli smo da programeri pišu programe uglavnom koristeći programske jezike visokog nivoa, a da računala izvršavaju instrukcije strojnog jezika. Pitanje je što računalo radi s programom napisanim u jeziku visokog nivoa. Najčešći način je koristiti računalni program koji nazivamo **prevodilac (compiler)**. Prevodilac prevodi program napisan u jeziku visokog nivoa u program sastavljen od strojnog jezika. Prevedeni program onda možemo pokrenuti na računalu. (U čemu se piše prevodilac ?)

U slučaju Java jezika korišten je malo drukčiji pristup u kojem se u procesu pisanja Java koda do transformacije u računalu razumljiv kod koriste dva programa. Prvo se program koji je programer napisao u Javi, pomoću prevodioca prevodi u **bytecode** program**.** Bytecode je sličan strojnom jeziku, ali je neovisan o bilo kojem računalu. Bytecode program nije više čitljiv od strane programera. Njega čita i izvršava program koji se naziva **Java virtual machine.** Prednosti pristupa u dva koraka jest da se tako proizvode programi koji se ipak izvršavaju zadovoljavajućom brzinom te se Java okolina može brzo realizirati na bilo kojem računalu.
Originalni Java program koji piše programer i kojeg prevodi prevodilac naziva se **izvorni kod**. Bytecode koji proizvodi prevodilac i interpretira Java virtual machine naziva se **objektni kod.**



Slika 1.1 Usporedba izvršavanja Java programa s tradicionalnim postupkom prevođenja

Postoje brojne razvojne okoline u kojima je moguće pisati, prevoditi i izvršavati programe. Za potrebe ovog predmeta zadržat ćemo se na najjednostavnijem sustavu koji je ujedno i podloga i za druge kompleksnije razvojne okoline. Sustav se naziva Software Development Kit (SDK). Može se naći na stranicama

`java.sun.com`

Ove stranice održava kompanija Sun Microsystems koja je odgovorna za razvoj Jave.
Verzija Java okoline u trenutku pisanja ovog teksta je 1.5. U ovom predmetu koristit ćemo prethodnu 1.4 verziju.

Instalacija Java okoline se u najkraćim crtama obavlja na slijedeći način:

- Pokreni Java 2 SDK installer (datoteka j2sdk-1_***-win.exe), odaberi mjesto instalacije i instaliraj.
- Dodaj u PATH varijablu operativnog sustava mjesto instalacije (npr. C:\jdk1.4.2\bin). Način dodavanja je ovisan korištenom operativnom sustavu.
- Provjeri (ukloni) CLASSPATH varijablu (-classpath command-line prekidač je bolji način).

Detaljniji (aktualniji) opis instalacije može se naći na stranicama java.sun.com.

3. Primjeri jednostavnih Java programa

U Javi program iste funkcionalnosti kao C program koji ispisuje "Hello World!" izgleda ovako:

```
public class Hello
{  /* napiši jednostavnu poruku na ekran*/
   public static void main(String[] args)
   {  System.out.println(”Hello, World!”);
   }
}
```

Ako niste vidjeli dosada neki Java program ovaj jednostavni program izgledat će vam konfuzno. Cilj ovih početnih sati predavanja je da vam se objasni struktura ovako jednostavnih programa.

Svaki Java program sadrži **naredbe** (statements). Svaka naredba opisuje neku operaciju koju računalo treba izvršiti. Operacija može biti ispis neke informacije na ekranu, može biti neka računska operacija, provjera položaja miša na ekranu, itd. Računalo jednostavno izvršava naredbu po naredbu.

U programu koji je gore napisan nalazi se samo jedna naredba: `System.out.println("Hello, World!");`

Kad se ta naredba izvrši na ekranu će se pojaviti slijedeći ispis: `Hello, World!`

Java ima različite načine pisanja poruka po ekranu bilo da pišemo po prozoru, na web stranicu, itd. U ovome slučaju koristimo jednu Java **metodu** koja se naziva `System.out.println`. Rezultat izvršavanja bit će ispis poruke u najjednostavnijem obliku prozora kojeg nazivamo **konzola** (ili DOS prozor u Windows OS). Konzola dopušta samo jednostavan ispis teksta , redak po redak.

Slijedi program s dvije naredbe koje su tiskane podebljano (ne koristimo podebljane fontove u Java programima).

```
public class Hello
{  /* napiši jednostavnu poruku na ekran*/

   public static void main(String[] args)
   {  System.out.println("Hello, World!");
      System.out.println("See you later.");
   }
}
```

Kada pokrenemo ovaj program izvršit će se obje naredbe jedna za drugom. Prvo će se na ekranu ispisati u jednoj liniji `Hello, World!` , a nakon toga u drugoj liniji `See you later.` Oba programa imaju formu:

```
public class Hello
{  /* komentar. */
   public static void main(String[] args)
   {
          Naredbe

   }
}
```

Ne postoji ograničenje na broj naredbi u programu. Može ih biti na tisuće. Ostatak ovog jednostavnog programa može se promatrati kao pakiranje. Ovo pakiranje ćemo objasniti poslije. Sad ćemo se zadržati na opisu **objekata** koji su neizostavni dio svakoga pa i najjednostavnijeg Java programa.

4. Objekti i Metode

Razmotrimo naredbu koja ispisuje poruku: `System.out.println(”Hello, World!”);` .

Gdje su tu **objekti** ? Poznavalac Java jezika vidjet će dva objekta. Prvi je objekt `System.Out` , a drugi sami niz znakova `”Hello, World!”`. Java cijelo vrijeme radi s objektima. U Java biblioteci definirano je mnogo vrsta različitih objekata koje možemo koristiti u svojim programima.

Možemo i kreirati objekte prema našim potrebama. Npr. pišemo program koji će pratiti koji su studenti na FESB-u prijavljeni na koji predmet. Tada ćemo napisati takav program koji će pokretanjem:

- za svakog studenta kreirati jedan objekt studenta
- za svaki predmet također jedan objekt predmeta.

Svaki objekt studenta sadržavat će određene podatke poput osobnih podataka studenta i liste upisanih predmeta. Objekt predmeta može sadržavati naziv predmeta i druge podatke vezane za predmet.

U isto vrijeme kad definiramo izgled objekata trebamo i definirati koje će se operacije izvršavati nad tim objektima. Što se tiče objekta student , bit će nam potrebne operacije kreiranja objekta studenta, ažuriranja liste predmeta koje je student upisao, operacije ispisa podataka o studentu na ekran, itd. Te operacije koje se izvršavaju nad objektom nazivaju se **metode**.

Dosad smo već vidjeli primjer metode. Njen puni naziv je: `System.out.println`

Ovaj naziv označava metodu `println` koja pripada objektu `System.out`. `System.out` je objekt čiji je zadatak da primi poruku koju treba prikazati na ekranu.
Zamislimo ga kao osobu kojoj dajemo što treba ispisati na ploči.

`Println` metoda je operacija koja se izvršava nad porukom. tako nam izraz:

```
System.out.println(”Hello, World!”);
```

kaže:

koristi `println` metoda za slanje poruke `”Hello, World!”` objektu `System.out`, koji će je prikazati na ekranu.

Svaki objekt pripada **klasi (class)** koja specificira od kojih podataka se objekt sastoji i koje metode posjeduje. Npr. svi nizovi znakova pripadaju klasi koja se naziva String. Klase String i System.Out definirane su u klasama koje pripadaju Java bibliotekama. Možemo kreirati i svoje klase npr. klasu Student i klasu Predmet.

Koji je odnos klasa-objekt ? Kažemo da je objekt instanca od klase. Jednostavno, klasa je opis objekta napisan u kodu. Možemo je promatrati kao kalup ili skicu prema kojoj se u tijeku izvršavanja programa kreiraju objekti. Program može kreirati više objekata ,instanci iste klase. Java biblioteka je u potpunosti sastavljana od definicija klasa. Ako napišemo bilo koji program u Javi i on će se sastojati od klasa. Većina klasa definira tipove objekata. Postoje samo nekoliko klasa kojima se ne definira objekt već su sastavljene samo od samostalnih metoda.

Ipak nije sve u Javi objekt. Najjednostavniji tipovi podataka poput cjelobrojnih i brojeva u pokretnom zarezu tretiraju se nešto drugačije. takvi podaci nazivaju se **primitivni tipovi podataka**.

5. Kako je konstruiran program iz primjera

Prethodni dio pokazao nam je da se Java programi sastoje od klasa. Programi koji su navedeni kao primjer uklapaju se u tu tvrdnju, ali ipak na vrlo primitivan način. Oba programa sastoje se od jedne klase (Hello) koja se sastoji samo od jedne statičke metode tj. metode koja ne pripada nijednom određenom objektu.

Svaki metoda bilo da pripada objektu ili ne, sadrži određeni broj naredbi koje izvršavaju neku korisnu operaciju. (U drugim računalnim jezicima imamo funkcije ili procedure)

Analizirat ćemo korak po korak kompletnu metodu iz zadnjeg primjera:

```
/* napiši jednostavnu poruku na ekran*/
public static void main(String[] args)
{  System.out.println(″Hello, World! ″);
   System.out.println(″See you later. ″);
}
```

Sastoji se od slijedećih dijelova:

1. `/* napiši jednostavnu poruku na ekran*/`

Ovo je komentar koji opisuje što ni program trebao raditi. to je jednostavno poruka za onoga tko čita izvorni kod programa. Svaki tekst između `/* i */` bit će tretiran kao komentar i Java će ga u potpunosti ignorirati.

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

Ovo predstavlja **zaglavlje** (heading) metode. Svaka metod ima svoj naziv. U ovom slučaju naziv metoda je riječ `main` koja se nalazi neposredno ispred zagrada. Riječi `public`, `static` i `void` pokazuju Java prevodiocu način korištenja metoda main. (Bit će objašnjeno kasnije).

Dio u zagradama, `String[] args`, opisuje informaciju koja će biti proslijeđena metodu svaki put kad bude pozvan. Naziva se lista **parametara.** U navedenom primjeru ta je informacija ignorirana, odnosno nije korištena u programu. (koko se koristi bit će objašnjeno poslije)

3.
```
{  System.out.println(″Hello, World! ″);
   System.out.println(″See you later. ″);
}
```

Ovo je **tijelo** (body) metoda. Uvijek se sastoji od niza naredbi zatvorenih u vitičaste zagrade, {..}. Pozivom ovog metoda izvršava se svaka od naredbi.

Sve metode sastoje se od tri navedena dijela. Strogo rečeno komentar je opcionalan. Međutim preporuča se uvijek početi s komentarom koji ukratko kaže što radi metod koji slijedi. Taj dio nazivamo **specifikacijom**. Gornji primjer je prejednostavan da bi specifikacija bila od veće koristi, ali u većim programima to je najefikasniji način da pomognemo razumijevanju programa. Posebno je to bitno ako na programu radi više programera.

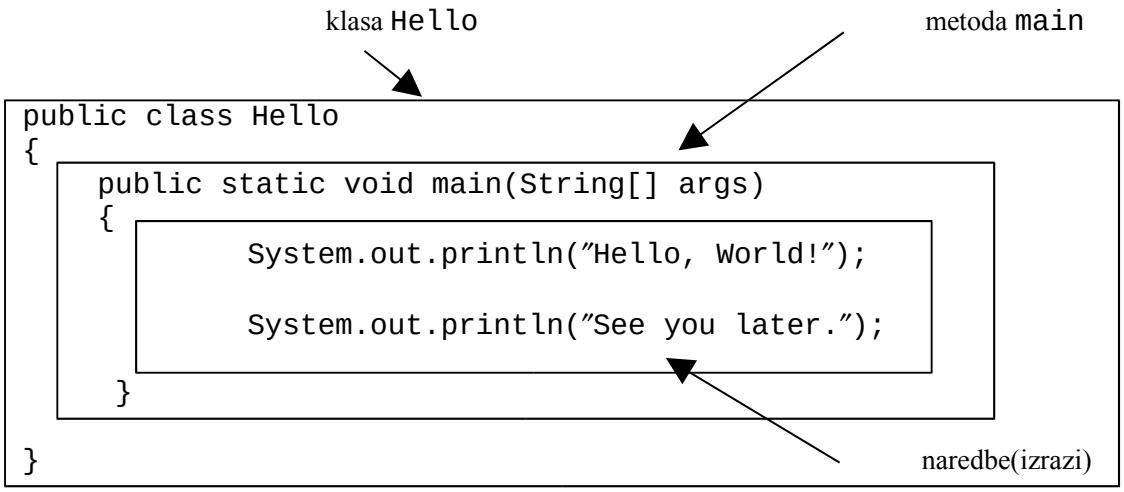
Svaki program sastoji se od određenog broja definicija klasa. U dva gornja primjera u programu je definirana samo jedna jedina klasa nazvana `Hello.`

Definicija klase započinje s zaglavljem: `public class Hello`

Zaglavlje je praćeno elementima koji sačinjavaju klasu, zatvorenim u vitičaste zagrade.
U našim primjerima klasa se sastoji od samo jednog člana, metode nazvane `main`, koju svo već opisali. Općenito program se sastoji od jedne ili više definicija klasa od kojih svaka sadržava jednu ili više metoda.

Java programeri koriste konvenciju po kojoj naziv klase započinje velikim slovom, a naziv metode malim slovom. U ovome predmetu ćemo se nastojati strogo pridržavati navedene konvencije. Općenito bilo koji naziv u Javi (**identifikator**) sastoji se od slova, znamenki i mora početi sa slovom. Za potrebe ove definicije se simboli valuta poput £ i $, i povlaka ('_') računaju kao slova. Duljina naziva nije ograničena.

Na slijedećoj slici prikazan je naš program koji se sastoji od dvije naredbe razdvojen okvirima koji nam pokazuju što je klasa, što je metoda a što su naredbe.



6. Metoda System.out.println

`System.out.println` je prva metoda iz Java biblioteke koji smo upotrijebili. Ova metoda će ispisati niz znakova na konzolu (DOS prozor). Naziv `println` je skraćenica za 'print line'.

Mjesto na ekranu na kojemu će se ispisati slijedeći znak označeno je na ekranu s malom treptajućom linijom nazvanom **kursor.** `System.out.println` metoda ima tu osobinu da ispisom teksta pomiče kursor na početak slijedeće linije. Ako želimo da kursor ostane na prethodnoj liniji koristit ćemo metodu `System.out.print`.

Drugi način određivanja kad želimo ispis u novoj liniji je korištenje para znakova `\n` u nizu znakova svaki put kad je potrebno da ispis krene u novu liniju. Npr. :

```
System.out.print("Hello, World!\nSee you later\n")
```

će ispisati `Hello, World!` , nakon toga pomaknuti kursor u novu liniju gdje će ispisati `See you later.` , i nakon toga pomaknuti kursor na novu liniju. Ta će naredba imati isti rezultat kao i par naredbi:

```
System.out.println("Hello, World! ");
System.out.println("See you later. ")
```

7. Kako prevesti i pokrenuti Java program

Pretpostavimo da želite izvršiti program iz prvog primjera:

```
class Hello
{  /* napiši jednostavnu poruku na ekran*/
   public static void main(String[] args)
   {  System.out.println("Hello, World!");
   }
```

```
}
```

Prvo morate biti pristupiti računalo na koje je instalirana podrška za Javu odnosno Java SDK.

A. **Ukucajte program i pohranite ga u datoteku `Hello.java`.**

Možete koristiti bilo koji tekst editor za unos koda, npr. Notepad. Korisno je imati i neki sofisticiraniji Java editor npr. JedPlus. Takvi editori nam mogu omogućiti korisne funkcije poput sintaksnog naglašavanja ili automatskog uvlačenja teksta. Moguće je iz takvih editora pozvati i operacije prevođenja i izvršavanja.

Ako se vaš program sastoji od samo jedne klase potrebno ga je pohraniti u datoteku koja ima isti naziv kao i klasa uz dodanu ekstenziju '`.java`'. kako se naš jednostavni program sastoji od samo jedne klase nazvane Hello, datoteku **moramo** nazvati `Hello.java`.

B. **Otvori DOS prozor i postavi trenutni direktorij na direktorij gdje je datoteka s programom.**

Sve naredbe koje slijede tipkaju se u ovaj prozor. Poruke prevodioca i sve što će program ispisati odvija se također u ovome prozoru. Postoje i drugi načini koje ćemo obraditi na vježbama, ali ovo je općeniti način koji radi na svim računalima.

U ovome trenutku bit će otvorena dva prozora. Jedan s editorom (npr. Notepad), a drugi s DOS prozorom koji služi za interakciju s Java sustavom. U tom DOS prozoru prevodimo i izvršavamo program.

C. **Korištenje JDK za prevođenje programa u `Hello.java` datoteci.**

Za prevođenje programa treba utipkati:

```
javac Hello.java
```

Ako dobijete poruku da sustav ne može naći javac (Java prevodilac) znači da ili nije pravilno postavljena PATH varijabla sustava ili nije instaliran JDK.

Ako nema grešaka u programu prevodilac će proizvesti bytecode verziju vašeg programa u datoteci nazvanoj `Hello.class`.

Ako prevodilac nađe greške poput tipkanja Class umjesto class ili izostavljanja znaka ; na kraju naredbe, odbit će prevođenje i izvijestiti o pronađenim greškama.
To nazivamo **greškom prevođenja (compiler error)**. Najčešće je poruka o grešci takva da je jednostavno naći mjesto u kodu gdje smo učinili pogrešku. Ponekad iz poruke nije jasno odakle potječe greška i tada je potrebno pažljivo pregledati kod. Početnici često pogriješe tražeći grešku baš u liniji gdje je to prevodilac javio. Međutim , greška se može nalaziti i negdje prije !

Ako postoje greške kod prevođenja potrebno ih je otkloniti u editoru. nakon otklanjanja grešaka ne zaboravite snimiti datoteku.

A. **Korištenje JDK za pokretanje prevedene verzije programa koja se nalazi u `Hello.class` datoteci.**

Nakon što je prevodilac proizveo bytecode verziju programa `Hello.class`, možete ga pokrenuti u Java Virtual machine tipkanjem:

```
java Hello
```

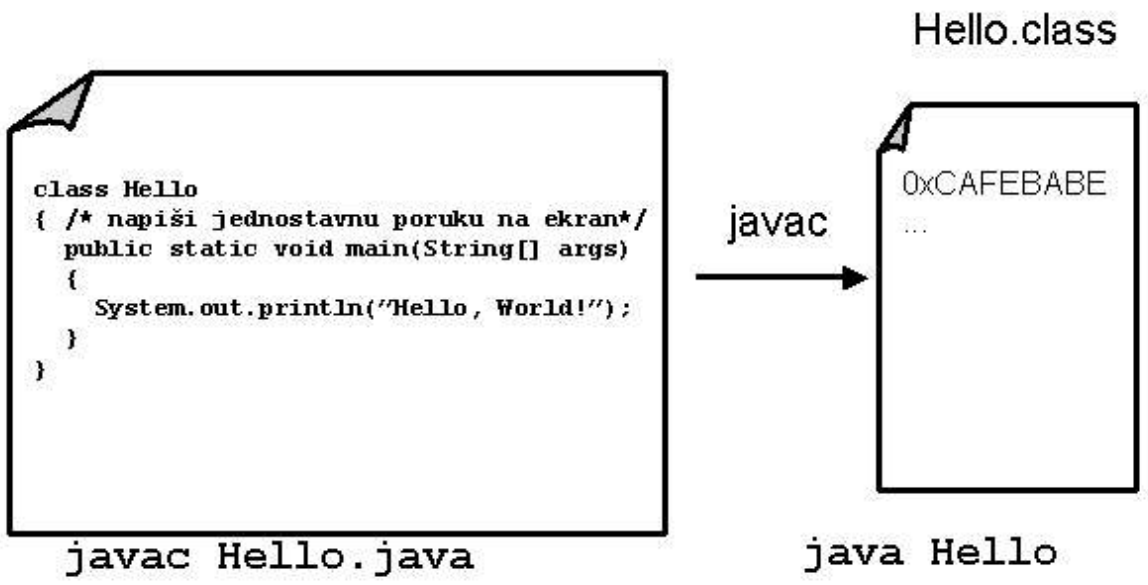
Ne tipkajte `.class` ekstenziju. JDK pretpostavlja da ste mislili na `Hello.class`.

Ovaj se program nakon toga treba izvršiti i ispisati poruku u DOS prozoru. Ako želite modificirati ispis programa ili dodati još koju naredbu opet se trebate vratiti u prozor tekst editora, napraviti modifikacije i nakon toga u DOS prozoru ponoviti postupak prevođenja i izvršavanja.

U tijeku izvođenja programa može nastati greška. Takva greška naziva se **run-time greška** (run-time error) ili **greška u izvršavanju** (execution error).

U nekim slučajevima JVM neće moći izvršiti program do kraja. Tada će ispisati poruku o grešci s podatkom gdje je program došao u izvođenju prije nego što je prekinut.

Proces pisanja, prevođenja i izvođenja programa pokazan je i na slijedećim slikama:



Slika 1.2 Pisanje programa, prevođenje, izvršavanje.

8. Pisanje koda Java programa

Prvo pravilo pisanja bilo kojeg programa je: pišite pažljivije što god možete. Skoro svaka pogreška u pisanju će najvjerojatnije proizvesti grešku prilikom prevođenja ili prilikom izvođenja. Prvo će je trebati detektirati, a nakon toga korigirati. Npr. svaki izraz koji koristi naredbu `System.out.println` mora završiti s ; . Ako izostavite ; dobit ćete poruku o grešci u fazi prevođenja.

U pisanju Java programa morate paziti na korištenje malih i velikih slova. Za razliku od nekih drugih programskih jezika Java razlikuje velika i mala slova (case sensitive). To znači da ne možete utipkati `Class` ili `CLASS` na početku programa. Potrebno je točno pisati `class`.

Riječ `class` ima u Java programima specijalno značenje i jedna je od **ključnih riječi** (keywords) Java jezika. java ima 47 ključnih riječi i sve se pišu malim slovima. kao što je prije napisano držat ćemo se konvencije po kojoj se nazivi klasa pišu s početnim velikim slovom, a nazivi metoda malim slovima.

Ako imamo nazive koji se sastoje od više spojenih riječi možemo početak slijedeće riječi pisati velikim slovom. Npr. klasu možemo nazvati HelloWorld, a metodu npr. ispisTeksta.

Koliko grešaka pisanja možete vidjeti u ovome kodu ? prevodilac će javiti svaku od njih.

```
public Class Hello
{  /* Write a simple message on the screen.
   public static viod main(string[] args)
```

```
      {  System.out,println('Hello, World!");
         System.out.printline("See you latef.");
      }
   )
```

Jedina greška koju prevodilac neće javiti je pogrešno pisanje unutar niza znakova `See you latef`. Ta greška će biti očigledna tek nakon izvršavanja programa.

Zbog svega navedenog nije se praktično oslanjati na prevodilac u otkrivanju grešaka. To je prije svega dugotrajan proces jer je potrebno svaki put nakon ispravljene greške proći kroz faze prevođenja i izvršavanja. Međutim to više vrijedi za iskusne programere nego za početnike kojima će prevodilac često biti neprocjenjiva pomoć u otklanjanju pogrešaka.

Osim što je Java prevodilac razlikovanjem velikih i malih slova zna biti dosta nezgodan u drugim pogledima je vrlo liberalan. Uopće mu nije bitan prostorni raspored vašeg koda tj. upotreba praznina i novih redova.

Slijedeći primjer bi se trebao prevesti bez greške iako je vrlo teško čitljiv:

```
public class Hello{public static void main(String[]
args){System.out.println("Hello, World!");System.
out.println("See you later.");}}
```

Iz ovoga je na prvi pogled teško ustanoviti da se program sastoji od jednog metoda koji sadrži dvije naredbe. Čak će se i slijedeći primjer pravilno prevesti:

```
public                                          class
Hello
public
static
void                          main
(String[] args)
   {  System.    out.       println
(                 "Hello, World!"                  );
                                      System.out.
println(         "See you later."                  );
}                                                      }
```

Sve dok ne spojite neke riječi poput `classHello`, ili ne razdvojite `cla ss`, prevodilac se neće buniti..

Prevodilac prema svemu navedenom je vrlo fleksibilan u korištenju razmaka i novih redova. Međutim bilo tko bude čitao vaš program (i vi sami) poželjet će mnogo uredniji i jasniji kod. Zato je praznine uputno koristiti da bi se dobio jasniji kod.
Ako pogledate na originalni kod gore modificiranih programa, vidjet ćete da postoje neka pravila u pisanju. Npr. metoda `main` je pomaknuta tri mjesta unutar u odnosu na prethodni redak. To nazivamo **uvlačenje koda** (indentation). Ako ima više metoda unutar klase, stavljamo razmak između svake. Primijetite i da vitičaste zagrade imaju konvenciju o smještaju. Vidjet ćemo tijekom ovoga predmeta kako postupati s pojedinim elementima koda. Osim urednosti koja omogućava razumijevanje koda, drugi rezultat je pisanje koda s manje grešaka.

9. Kako rješavati zadatke

Zadaci koji su navedeni na kraju svakog poglavlja prestavljaju zadatke koji će se javiti i na ispitu. Stoga nastojte zadatke samostalno rješavati jer je to najefikasniji način za usvajanje gradiva.Podloga za rješavanje zadataka jeste sadržaj svakog poglavlja. Štoviše, tu ćete naći i dijelove izvornog koda koji će vam koristiti u rješavnju zadataka.

Na računalu na kojemu dobijete korisnički račun kreirajte direktorij `JProg`. Unutar tog direktorija za svako poglavlja otvorite direktorij `Pn` gdje je n broj poglavlja. Dakle za prvo poglavlje otvorite direktorij P1. Kada završite poglavlje 1 unutar direktorija P1 trebali biste imati slijedeće datoteke:

```
Hello.java , Hello.class
ImeUOkviru.java, ImeUOkviru.class
Gresnik.class , Gresnik.java
Inicijali.class , Inicijali.java
```

10. Zadaci za prvo poglavlje

Za ovo poglavlja zadana su tri jednostavna problema tj. pisanje tri jednostavna programa..

1. Ukucaj, prevedi i pokreni slijedeći Hello program koji ispisuje dvije linije na ekran:

```
/* Autor: Ime Prezime */
class Hello
{
   /* jednostavan ispis na ekran. */
   public static void main(String[] args)
   {  System.out.println(″Ciao, Svite!″);
      System.out.println(″Vidimo se kasnije !″);
   }
}
```

2. Napišite program s nazivom klase ImeUOkviru. Program treba ispisati vaše ime u okviru poput slijedećega:

```
+--------+
 |  Ivan  |
+--------+
```

Uputa: nije svejedno kako ćete nazvati datoteku s kodom!

3.Ukucaj i pokreni slijedeći program te analiziraj pogreške koje će javiti prevodilac. ukucajte ga sa svim pogreškama !

```
public Class Gresnik {
   /* Write a simple message on the screen.
   public static viod main(string[] args)
   {  System.out,println('Alo, Svijete!");
      System.out.printline("Vikimo se kasnije.");
   }
}
```

Zatim ispravite pogrešaka koliko možete i pokušajte natjerati program da radi kako bi trebao.

Poglavlje 2. Brojevi i znakovni nizovi (strings)

U prvom poglavlju pisali smo programe koji su samo ispisivali poruke na ekran. U ovome poglavlju uvodimo nove izraze koje ćemo koristiti u metodama. To uključuje upotrebu dvije vrste vrijednosti: brojevi i znakovni nizovi (strings).

SADRŽAJ

Java posjeduje slijedeće tipove primitivnih tipova podataka:

ključna riječ	opis	veličina
cjelobrojni (integers)		
byte	cijeli broj duljine jednog byte-a	8 bit-a bez predznaka
short	cijeli broj duljine dva byte-a	16 bit-a bez predznaka
int	Integer	32 bit-a bez predznaka
long	Long Integer	64 bit-a bez predznaka
brojevi u pokretnom zarezu (real numbers)		
float	jednostruka preciznost	32-bita
double	dvostruka preciznost	64-bita
ostali tipovi		
char	jedan znak	16-bita unicode znak
bolean	boolean(logička vrijednost)	true ili false

1. Cjelobrojne vrijednost (integers)

U prvom poglavlju smo spomenuli da nisu svi podaci u Javi objekti. Najjednostavniji tipovi podataka, poput cijelih brojeva ili brojeva u pokretnom zarezu odstupaju od logike da sve treba biti objekt. Takvi tipovi podataka nazivaju se *primitivni tipovi podataka*.

Ovo poglavlje počinjemo primjerom upotrebe cijelih brojeva (integers).

Pretpostavimo da idete kupovati na tržnici. Kupili ste 2 kg jabuka po 12 kuna, 3kg krumpira po 2kune i 50lipa i 2 kg bresaka po 14 kuna. Koliko ste kuna potrošili.

Nije potrebna snaga Jave na modernom računalu za izračunati navedeni primjer jer i napamet (ili bar uz pomoć papira i olovke) da se izračunati je ukupan račun 85.5 kune.

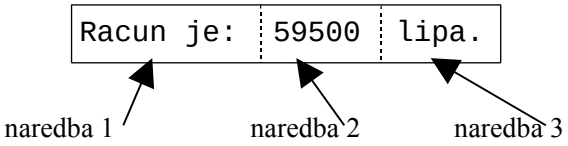
Međutim program će rasvijetliti nekoliko točaka u računanju s cijelim brojevima.

PRIMJER 1

```
public class Racun1

{  /* Izracunaj ukupni racun za 2 kg jabuka po 12 kn
   3kg krumpira po 2kn i 50lipa i 2 kg bresaka po 14 kn */
```

```
    public static void main(String[] args)
    {  System.out.print("Racun je: ");
       System.out.print(2*1200 + 3*250 + 2*1400);
       System.out.println(" lipa.");
    }
}
```

Poput svih primjera za ovo poglavlje program se sastoji od jedne klase koja sadrži samo jednu metodu `main`. U ovome primjeru metoda sadrži tri naredbe. Kad se izvrši tri metode će proizvesti dijelove linije na ekranu kako je to prikazano na slici ispod.
Primijetite da prve naredbe koriste `print` umjesto `println` što znači da ne pomiču kursor u novi red nakon ispisa.



Kada Java izvrši naredbu 2 , izračunat će vrijednost slijedećeg aritmetičkog izraza:

```
2*1200 + 3*250 + 2*1400
```

i prikazati rezultat 59500. Primijetite da možemo ispisivati brojeve koristeći
`System.out.print` na isti način na koji smo koristili kod ispisivanja znakovnih nizova.

U Java programu slijedeći operatori mogu se koristiti u aritmetičkim izrazima:

+	zbrajanje (addition)
–	oduzimanje (subtraction) (koristi se i za minus predznak)
*	množenje (multiplication)
/	dijeljenje (division)
%	ostatak (remainder)

U Javi (i većini drugih programskih jezika višeg nivoa) brojevi su **cjelobrojni** ili **brojevi u pokretnom zarezu** (integers or floating-point numbers). Ove dvije vrste vrijednosti pohranjuju se u memoriji na različite načine. Ako vrijednost napišemo bez decimalne točke , npr. 4 ona će biti tretirana kao cjelobrojna vrijednost. Ako je napišemo s decimalnom točkom npr. 4.0 bit će tretirana kao broj u pokretnom zarezu. Uvijek trebamo koristiti cijele brojeve kad to možemo. Računanja s njima su brža, koriste manje memorije i nisu podložni greškama zaokruživanja poput brojeva s pokretnim zarezom.

Operator dijeljenja , / , u stvari ima dva značenja. Ako ga koristimo s cijelim brojevima tada znači **cjelobrojno dijeljenje** . Npr. 11 / 4 = 2, (–11) / 4 = –2. Ako se koristi s dva broja s pokretnim zarezom onda znači normalno dijeljenje.
Npr. 10.5 / 2 = 5.25. Operator ostatka koristi se samo s cijelim brojevima. Npr. 11%4 = 3 and (–11)%4 = –3.

Ako Javi damo da izračuna izraz m%n gdje je n jednak nuli , Java će prijaviti run-time grešku.

kada procesira složenije izraze Java se drži standardnog prioriteta operatora pa npr. izraz 3+4*5 znači isto kao i 3+(4*5).

Kada se izračunava izraz sa više operatora istog prioriteta u nizu, izraz se računa s lijeva nadesno.
Npr. 3/4/5 znači isto kao i (3/4)/5, na kao 3/(4/5) što bi proizvelo grešku (Zašto?).

2. Varijable

Slijedeći primjer je druga verzija programa prvog primjera. Ovaj primjer koristi **varijable** tj. naziv koji znači vrijednost.
Program izračunava ukupan račun u dva odvojena koraka.

PRIMJER 2

```
public class Racun2

{  /* Izracunaj ukupni racun za 2 kg jabuka po 12 kn
   3kg krumpira po 2kn i 50lipa i 2 kg bresaka po 14 kn */

   public static void main(String[] args)
   {  int total =2*1200 + 3*250 + 2*1400;
      System.out.print("Racun je: ");
      System.out.print(total/100);
      System.out.print(" kn i ");
      System.out.print(total%100);
      System.out.println(" lipa.");
   }
}
```

Izraz: `int total =2*1200 + 3*250 + 2*1400;`

kaže: kreiraj varijablu `total` koja će sadržavati vrijednost izraza `2*1200 + 3*250 + 2*1400` (=59500). Riječ `int` na početku pokazuje da je tip vrijednosti koju varijabla `total` može pohraniti je cjelobrojna vrijednost.

Ovaj izraz se naziva **deklaracija varijable (ili definicija varijable).**

Izraz: `System.out.print(total/100);`

kaže: ispiši vrijednost naznačenu kao `total/100`. Radi se o cjelobrojnom dijeljenju čija je vrijednost 85 i ta će vrijednost biti ispisana.

Izraz: `System.out.print(total%100);`

kaže: ispiši ostatak dijeljenja `total` sa 100. U ovome slučaju je to 50.

Ukupan ispis na ekran bi trebao biti:

Racun je:	59	kn i	50	lipa.

Ovaj primjer smo mogli realizirati i bez varijable s upotrebom slijedeće dvije naredbe:

```
System.out.print((2*1200 + 3*250 + 2*1400)/100);
System.out.print((2*1200 + 3*250 + 2*1400)%100);
```

Upotrebom varijable izbjegli smo dvostruko računanje ukupnog broja lipa. U ovako malom promjeru to je mala ušteda, ali u realnim primjerima mogu se postići značajne uštede.

Drugi razlog upotrebe varijabli je da se jasno označe vrijednosti koje se koriste u računu tako da je jasnije što se u programu radi. Za nazive varijabli možete koristiti bilo koji naziv koji već nije upotrebljen za nešto drugo. U praksi upotrebljavaju se opisni nazivi koji odgovaraju ulozi varijable.

U slijedećoj verziji programa broju kilograma pojedinog artikla dani su očiti nazivi `jabuka, krumpir, breskva.` Usput , prihvaćena je konvencija da nazivi varijabli uvijek počinju s malim slovom.

PRIMJER 3

```
public class Racun3

{  /* Izracunaj ukupni racun za 2 kg jabuka po 12 kn
   3kg krumpira po 2kn i 50lipa i 2 kg bresaka po 14 kn */
```

```
    public static void main(String[] args)
    {  int jabuka = 2;
       int krumpir = 3;
       int breskva = 2;

       int total =
          jabuka*1200 + krumpir*250 +breskva*1400;

       System.out.print("Racun je: ");
       System.out.print(total/100);
       System.out.print(" kn i ");
       System.out.print(total%100);
       System.out.println(" lipa.");
    }
}
```

Ovaj program daje izlaz isto kao i prethodni. Upotreba varijabli omogućava bolje razumijevanje koda bilo vama ili nekome drugome koji će nakon vas modificirati kod.
U programiranju postoji pojam "magični broj". To su razne konstante u programu za koje prije ili kasnije se zaboravi što znače.

Kada je varijabla kreirana za nju se alocira prostor u memoriji gdje se potom može spremiti vrijednost varijable. Količina memorije koja se alocira ovisi o tipu varijable. Npr. za tip varijable `int`, kao što je to u slučaju varijable `total` , bit će alocirana 32 bita memorije.

Format pohrane je rijetko važan programeru. Bitniji je opseg vrijednosti koji se može pohraniti u zadanu varijable. Vrijednost `int` varijable može biti u području od –2147483648 do 2147483647. Ako su potreban veći opseg vrijednosti potrebno je umjesto `int` upotrijebiti `long` tip varijable. Taj tip zauzima 64 bita, a opseg je od -9223372036854775808 do 9223372036854775807. Postoje i kraći tipovi cjelobrojnih varijabli `short` (16 bita) i `byte` (8 bita).

3. **Dodjeljivanje** (Assignments)

Kao što smo vidjeli, svaka varijable ima naziv i odgovarajući dio memorije. Vrijednost koja je sadržana u memoriji je **dodijeljena** varijabli. Ta vrijednost ne mora biti ista u tijeku izvršavanja programa. Može biti zamijenjena drugom vrijednošću u tijeku izvršavanja programa. U slijedećem primjeru promijenit ćemo sadržaj varijable.

PRIMJER 4

```
public class Mjenjacnica1

{  /* Preracunaj eure u kune */

   public static void main(String[] args)

   {  int eura = 110;
      int centa = 55;

      centa = 100*eura + centa;
      double kuna = centa * 0.0751;
      System.out.print("Kuna: ");
      System.out.print(kuna);
   }
}
```

Program započinje kreiranjem dviju varijabli nazvanih `maraka` i `pfeniga` koje su postavljene na određene iznose.
Nakon toga se izvršava slijedeći izraz:

```
centa = 100*eura + centa;
```

Izraz znači: izračunaj vrijednost izraza 100*eura + centa, i pohraniti rezultat (11055) u varijablu centa. Drugim riječima, ovaj izraz će zamijeniti originalnu vrijednost pohranjenu u varijabli centa (=55) s ukupnim iznosom koji imamo izraženim u centima. Nova vrijednost se koristi u slijedećem izrazu:

```
double kuna = centa * 0.0751;
```

i predstavlja naše prvo korištenje brojeva u pokretnom zarezu. Izraz znači: kreiraj varijablu nazvanu `eura` , tipa `double`, i dodijeli joj vrijednost `centa *0.0751`. Varijabla tipa `double` sadržava broj u pokretnom zarezu *duple preciznosti* i alocira 64 bita memorije.
Produkt će se izračunati u aritmetici pokretnog zareza i rezultat će biti 830.2305

Konačan ispis je:

```
Kuna: 830.2305
```

Izraz:

```
pfeniga = 100*maraka + pfeniga;
```

nazivamo **dodjeljivanje** vrijednosti .Ovaj izraz mijenja vrijednost *postojeće* varijable.

Kadgod Java obavlja dodjeljivanje ono se obavlja u dva koraka:

(1) Prvo se računa vrijednost s desne strane od znaka '='.
(2) Zatim se ta vrijednost sprema u varijablu naznačenu s lijeve strana izraza

Moguće je mijenjati vrijednost spremljenu u varijablu, ali nije moguće mijenjati tip (type) varijable.

Moguće je deklarirati varijablu bez specificiranja njene inicijalne vrijednosti. Ako Java izvrši slijedeći izraz:

```
int total;
```

kreirat će varijablu tipa `int` i dati joj naziv `total`, ali neće spremiti nikakvu vrijednost u istu. Vrijednost varijabli će biti potrebno dodijeliti kasnije u programu. U međuvremenu nije moguće znati što sadrži varijabla `total` . U njoj će biti neki bitovi koji su ostali u memoriji otprije. Kažemo da vrijednost varijable 'nije definirana'.

Opći oblik deklaracije varijable je dakle:

deklaracija (s inicijalnom vrijednosti)
TIP NAZIV = IZRAZ ;

Npr. *TIP* može biti `double`, *NAZIV* može biti `eura`, a *IZRAZ* može biti `64.45`.

dakle

```
double eura = 64.45;
```

deklaracija (bez inicijalne vrijednosti)

TIP NAZIV;

dodjeljivanje

NAZIV = IZRAZ;

Java omogućava deklaracije varijabli u kojima se istovremeno deklarira više varijabli odjednom.

Primjer: kreiranje varijabli `w`, `x`, `y` i `z`, i dodjeljivanje inicijalnih vrijednosti varijablama `x` i `z`.

```
int w, x = 1, y, z = 2;
```

Često je nekoj varijabli potrebno samo nadodati neku vrijednost i postoji skraćeno pisanje za takav tip operacije. Npr. za dodati `3*jabuka` varijabli `total` pisat ćemo slijedeći izraz:

```
total += 3*jabuka;
```

Slično možemo koristiti simbol `-=` kada želimo oduzeti vrijednost od varijable , `*=` kada želimo pomnožiti varijablu s izrazom , itd.

Općenito izraz:

`x OP= y;` ekvivalentan je izrazu
`x = x OP y;`
gdje je OP bilo koji binarni operator

Jedan od najčešćih slučajeva promjene vrijednosti varijable je povećavanje za jedan.
Možemo pisati:

`x = x+1;` ili kraće `x += 1;` ili još kraće `x++;`

Slično za oduzimanje 1od `x`, pišemo `x--`.

4. Brojevi s pokretnom zarezom (Floating-point)

Naziv 'broj s pokretnim zarezom' proizlazi iz načina pohrane takvih brojeva u memoriji.

U Javi tip `double` se najčešće koristi za brojeve u pokretnom zarezu. Vrijednost tipa `double` zauzima 64 bita. to omogućava veoma velik opseg vrijednosti, približno $\pm 1.8 \times 10308$, što izraženo preko preciznosti iznosi 15 značajnih znamenki. Naziv `double` je skraćenica za 'double precision floating-point'. Java posjeduje i drugi tip za rad s brojevima s pokretnim zarezom koji se naziva `float`. Taj tip podataka zauzima samo 32 bita tako da predstavlja ekonomičnije korištenje memorije, ali ima samo pola preciznosti tipa `double` te ekvivalentno manji opseg od oko $\pm 3.4 \times 1038$.

Broj s pokretnim zarezom može se pisati s decimalnom točkom, npr. 14.56 ili 14.0. Može biti pisan i s eksponentom, npr. 14.56E-12.

Java sadržava niz metoda koje računaju širok opseg matematičkih funkcija koje koriste `double` vrijednosti. Sve su statičke metode (ne pripadaju nijednom objektu) iz klase Math koja se nalazi u Java biblioteci. Kadgod koristite statičku metodu koja pripada klasi biblioteke potrebno je kao prefiks naziva metode staviti i naziv klase.

`Math.sin(x)`	sinus od x
`Math.cos(x)`	cosinus od x
`Math.tan(x)`	tanges od x
`Math.exp(x)`	e^x
`Math.log(x)`	prirodni logaritam od x

`Math.abs(x)`	apsolutna vrijednost od x
`Math.floor(x)`	najveći cijeli broj ≤ x
`Math.ceil(x)`	najmanji cijeli broj ≥ x

U svakoj od metoda parametar x je tipa `double` , a rezultat je tipa `double`.

Java se neće buniti ako nađe vrijednost tipa `int` na mjestu gdje je predviđen tip `double.`
Kadgod se to dogodi Java će konvertirati cijeli broj u odgovarajući broj s pokretnim zarezom

Razmotrite slijedeće izraze:

```
int m = 3;
int n = 4;
double x = m*n;
```

Kada Java bude izvršavala treći izraz. izračunat će m*n koristeći cjelobrojnu aritmetiku te rezultat privremeno spremiti kao 32-bitnu vrijednost odnosno cijeli broj 6. Zatim će tu vrijednost konvertirati u 64-bitni broj u pokretnom zarezu (6) te ga spremiti u varijablu x.

Što će biti rezultat izvršavanja slijedećeg izraza:

```
double x = m/n;
```

Odgovor je da će x biti postavljen na vrijednost 0. Razlog tome je da će za izračunavanje izraza m/n biti upotrijebljena cjelobrojna aritmetika, jer su oba operanda cjelobrojna

Pretpostavimo da ipak želimo koristiti normalno dijeljenje brojeva s pokretnim zarezom. Potrebno je Javu uvjeriti da vrijednosti m i n tretira kao brojeve s pokretnim zarezom. Možemo to učiniti na slijedeći način:

```
double x = ((double) m) / n;
```

Izraz `(double)` naziva se **cast operator**. Stavljanjem cast operatora ispred m kažemo Javi da konvertira vrijednost od `m` u ekvivalentnu `double` vrijednost. Kad smo to učinili operator se sada odnosi na jednu `double` vrijednost i jednu `int` vrijednost. U tom slučaju Java će koristiti dijeljenje za brojeve s pokretnim zarezom.

Kada želimo konvertirati broj u pokretnom zarezu u cijeli broj postoje dva načina.
Prvi je način da od broja s pokretnim zarezom uzmemo cijeli dio, dakle dio koji ostane kad oduzmemo decimalni dio.

Npr., broj u pokretnom zarezu 4.7 tada postaje cijeli broj 4. Način kako to postižemo je upotreba `int` cast operatora:

```
double x = 4.7;
int i = (int) x;
```

Drugi izraz će konvertirati vrijednost od `x` u cijeli broj 4 i nakon toga pohraniti ga u varijablu `i`. Da je x bio postavljen na –4.7, `i` bi primio vrijednost –4.

Drugi način konvertiranja brojeva u pokretnom zarezu u cijele brojeve je zaokruživanje na najbliži cijeli broj. Npr. najbliži cijeli broj za 4.7 je 5.

Npr. slijedeći izraz obavit će zaokruživanje pozitivnog broja x i pohraniti zaokruženu vrijednost u varijablu i :

```
int i = (int) (x + 0.5);
```

Npr., ako je `x` jednak 4.7, vrijednost od `x + 0.5` bit će 5.2, a vrijednost od `(int)(x + 0.5)` bit će 5.

(Upozorenje. Ne piši: `int i = (int)x + 0.5;`)

Kako bi bilo izvedeno zaokruživanje za negativne brojeve ?

U zaokruživanju broja u pokretnom zarezu(i pozitivnih i negativnih) na najbliži cijeli broj možete koristiti i metodu iz Math klase:

```
Math.round(x)
```

To je zgodno rješenje osim što `Math.round(x)` kao rezultat vraća vrijednost tipa `long`.
Sve što je potrebno da bi se rezultat dodijelio varijabli tipa `int` je primjena `(int)` cast operatora :

```
int i = (int) (Math.round(x));
```

5. Čitanje ulaza korištenjem ConsoleReader objekta

Primjeri koji su dosada obrađivani koriste za svoj račun vrijednosti koje je programer upisao u sam kod programa. Ako u njima želimo neke druge vrijednosti moramo ih mijenjati u kodu programa.

Ljepša alternativa je da ponovo napišemo programe tako da njihove ulazne vrijednosti unosimo koristeći tipkovnicu tijekom izvršavanja programa. Npr. program koji konvertira valute mogao bi prvo ispisati poruku da korisnik upiše iznos u markama, zatim pročita ono što mu se upiše i na kraju ispiše rezultat.

U ovom dijelu napisati ćemo takav program.

Dosada smo u primjerima prikazivali informacije na ekranu koristeći `println i print` metode koje pripadaju objektu `System.out`.

Objekt `System.out` ima pristup ekranu i može proslijediti na njega bilo koje brojeve ili stringove koje mu damo kao parametre. Isto je točno kada čitamo s tipkovnice. U tom slučaju trebamo objekt koji je spojen na tipkovnicu i koji ima metode kojima možemo pristupiti znakovima koje kucamo na tipkovnici.

U Javi postoji takav objekt i naziva se `System.in`. Nažalost posjeduje veoma limitiran broj metoda. Posjeduje jednu metodu `read` koji će pročitati sve što korisnik upiše, ali ga je teško koristiti jer čita samo jedan karakter.

Nije teško konstruirati klasu s boljim setom metoda. U ovome tečaju koristit ćemo jednu klasu nazvanu `ConsoleReader.`

Kad je želimo koristiti u programu na početku programa napišemo:

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

Primijetite da se radi o deklaraciji varijable. Ova deklaracija kreira novi objekt tipa klase `ConsoleReader` koji može pristupiti `System.in` objektu koji je spojen na tipkovnicu.
U isto vrijeme kreira se varijabla `in` tipa `ConsoleReader` koja označava taj objekt.

Kad se izvrši ta naredba objekt koji nam je potreban odsad je prisutan u memoriji i njegove metode možemo pozivati koristeći njegov naziv `in` . (Možemo ga i drukčije nazvati)

`ConsoleReader` objekt posjeduje tri korisne metode:

`in.readInt()` vraća `int` vrijednost.

To će biti cijeli broj koji korisnik utipka. Npr. izraz :

int broj = in.readInt(); //će kreirati varijablu `broj` i pridružiti joj vrijednost koja se ukuca s tastature.

`in.readDouble()`; // vraća `double` vrijednost.

`in.readLine()`; // vraća string. Sastojat će se od znakova koje korisnik ukuca do kraja linije.

Ovo je nova verzija programa za preračun maraka u eure.

Započinje se kreiranjem `ConsoleReader` objekta, koji je označen sa `in`. Tada traži od korisnika da utipka vrijednosti maraka i pfeniga. Naredbe koje čitaju vrijednosti naznačene su podebljano (ne i u stvarnom kodu programa) .

PRIMJER 5

```
public class Mjenjacnica2
{  /* Ucitaj iznos u markama i preracunaj u eure */
public static void main(String[] args)
{

 ConsoleReader in =    new ConsoleReader(System.in);
 System.out.println("Unesi iznose eura i centa");
     System.out.print("Iznos u eurima =");
 int eura = in.readInt();
 System.out.print("Iznos u centima =");
 int centa = in.readInt();

      centa = 100*eura + centa;
      double kuna = centa * 0.0751;
      System.out.print("Kuna: ");
      System.out.print(kuna);

 }
}
```

Na ekranu bi trebalo nakon svega pisati (ono što korisnik ukuca je ovdje naznačeno podebljano)

```
Unesi iznose eura i centa
Iznos u eurima = 110
Iznos u centima = 55
Kuna: 830.2305
```

`ConsoleReader` objekt nije robustan. Npr. ako unesete dva broja u jednu liniju ili ako unesete slova umjesto broja program će javiti run-time grešku.

Prije pokretanja bilo kojeg programa koji koristi klasu `ConsoleReader` potrebno je da istu imate u istome direktoriju kao i program u kojem je koristite.

Pretpostavimo da želite koristiti primjer 5. Prvo ćete primjer ukucati u datoteku s istim nazivom kao i klasa dakle `Mjenjacnica2.java`. U isti direktorij kopirajte i datoteku `ConsoleReader.java`. Tada prevedite i pokrenite `Mjenjacnica2` na uobičajen način. Kada Java prevodilac vidi da java koristi ConsoleReader klasu, automatski će je prevest.

6. Znakovni niz - string.

Nadalje ćemo se koristiti izrazom **string. String** je jedan od najčešće korištenih tipova podataka u Javi.

Niz znakova (**literal** string) možemo napisati kao niz znakova zatvorenih u dvostruke navodnike poput slijedećeg:

```
"Jedan dan"
```

Literal stringovi mogu biti proizvoljne duljine, te uključivati bilo koji znak. Čak možete imati i string bez karaktera. Takav string nazivamo **prazan** string i zadaje se kao "".

String može sadržavati **kontrolne znakove** (control characters). Ti znakovi znače stvari poput novog reda, tabulatora, ... Ako ispisujete string korištenjem `System.out.println` metode i ako ta metoda naiđe na kontrolni karakter npr. na newline kontrolni karakter ona će tada pomaknuti kursor na novi red.

Ako želite uključiti kontrolni karakter u literalni string, potrebno je korištenje \ karaktera nakon kojega slijedi slovo. '\' se naziva **escape** karakter. Koristi se i za uvođenje određenih karaktera u string poput " koji bi inače značio kraj stringa. Slijedi lista najznačajnijih kombinacija karaktera:

\n	novi red (newline)
\t	tabulator – pomak (tab)
\b	nazad (backspace)
\r	return – pomak na početak slijedeće linije
\f	line feed – pomak na slijedeću liniju bez odlaska na početak
\\	\ znak
\′	′ znak
\″	″ znak

Npr. izraz

```
System.out.print(″Jedan\ndan″);
```

napisat će 'Jedan' u jednu liniju te 'dan'u slijedeću liniju.

Ako želite pohraniti string da bismo ga koristili kasnije u programu koristit ćemo varijablu tipa **`String`.** Slijedi primjer izraza koji kreira novu varijablu tipa `String`, nazvanu `automobil` kojoj dodjeljuje vrijednost "BMW".

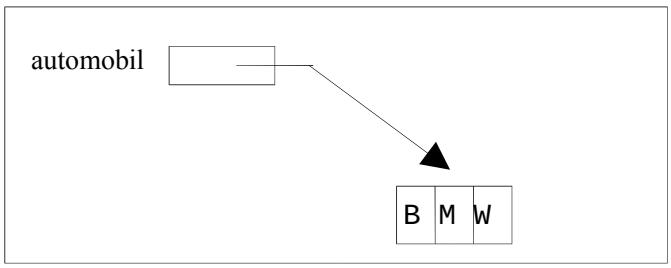
```
String automobil = ″BMW″;
```

Nije ispravno da će ovaj izraz pohraniti string `″BMW″` na memorijsku lokaciju varijable `automobil`. Što se stvarno događa uključuje jednu vrlo bitnu osobinu rada s memorijom Svaka lokacija u memoriji ima svoju adresu (**reference** value or **address**).
To je slično kao što svaka kuća u gradu ima svoju poštansku adresu

Kad se izvrši gornji izraz izvrše se slijedeći koraci:

(1) Kreira se varijabla nazvana `automobil`
(2) String `″BMW″ pohrani se u` ***drugi dio memorije !***
(3) Referenca (adresa*)* gdje je string pohranjen dodijeli se varijabli `automobil`.

Rezultirajuće stanje u memoriji može se prikazati dijagramom na slici 1.



Slika 2.1. Strelica od varijable prema stringu reprezentira činjenicu da varijabla sadrži referencu na lokaciju gdje je string pohranjen.

Ovaj način tretiranja String varijable znači da je sama varijabla uvijek iste veličine – dovoljno velika da sadrži vrijednost reference (adrese)
Npr. ako kasnije izvršimo naredbu:

```
automobil = ″Alfa Romeo″;
```

varijabla `automobil` će nakon toga opet sadržavati vrijednost reference, dakle istu količinu memorije. Razlika je u tome što će to biti referenca na neki drugi dio memorije koji sada sadrži drugi(duži) string `″Alfa Romeo″`.

Postoji jedan vrlo koristan operator koji se može primijeniti na stringove. Naziva se operator spajanja (concatenation) i označava se znakom '+'.

Ako su st1 i st2 stringovi, onda st1+ st2 znači string koji se sastoji od svih karaktera st1 nakon kojih slijede karakteri iz stringa st2.
Npr. ako je varijabli `automobil` dodijeljena referenca na string "BMW" onda će izraz

```
″Novi ″ + automobil + ″ je skup.″
```

znači string:

```
″Novi BMW je skup.″
```

Izrazi poput ovoga mogu se koristiti bilo gdje u programu gdje je potreban neki složeni izraz. Npr. slijedeći izraz :

```
System.out.println(″Novi ″ + automobil + ″ je skup.″);
```

će ispisati na ekranu `″Novi BMW je skup.″`

Kako često trebamo ispisivati poruke koje sadrže i brojeve Java dopušta da se kombiniraju brojevi i stringovi korištenjem + operatora.
Npr. `cijena` je cjelobrojna vrijednost koja sadrži vrijednost 30000. Tada će izraz:

```
″Cijena novog ″ + automobil + ″ je ″ + cijena + ″ eura″.
```

značiti string:

```
″Cijena novog BMW je 30000 eura″.
```

Primijetite da je Java uzela vrijednost iz varijable cijena i pretvorila je u string sa znamenkama pogodnim prikazu cijelih brojeva.

I u primjerima 2 i 3 mogli smo koristiti sličan način :

Izraze:

```
System.out.print("Racun je: ");
System.out.print(total/100);
System.out.print(" kn i ");
System.out.print(total%100);
System.out.println(" lipa.");
```

Mogli smo zamijeniti s jednim izrazom:

```
System.out.println
   ("Racun je: "+ (total/100) + " kn i " +
   (total%100) + " lipa." );
```

Točnije, zagrade oko izraza `total/100` i `total%100` nisu potrebne jer Java primjeni pravilo da operatori / i % imaju veći prioritet od operatora +.

Slijedi program koji obavlja luckastu konverzaciju s korisnikom. Program čita korisnikovo ime i dodjeljuje ga varijabli `ime`. Također program starost korisnika i dodjeljuje ga varijabli `godina`. Nakon toga koristi te dvije vrijednosti da bi konstruirao par rečenica koje će ispisati na ekran.

Program započinje konstrukcijom `ConsoleReader` objekta da bi mogao čitati što korisnik utipka. U ovom slučaju taj objekt je označen nazivom `korisnik`. Metoda `korisnik.readLine()` koristi se za čitanje imena , a metoda `user.readInt()` za čitanje broja godina.

EXAMPLE 6

```
public class Prica

{  /* Pricaj s korisnikom. */

   public static void main(String[] args)

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

      System.out.println
      ("Cao.  kako se zoveš?");
      String ime = korisnik.readLine();
      System.out.println
           ("Koliko imaš godina " + ime + "?");
      int godina = korisnik.readInt();
      System.out.print(godina + " su lijepe godine, ");
      System.out.println
      ("ali " + (godina+1) + " je bolje.");
      System.out.println
      ("Vidimo se kasnije " + ime + "!");
   }
}
```

Ovo bi se trebalo pojaviti na ekranu kad se program pokrene. (Korisnikov unos je podebljan.)

```
Cao.  kako se zoveš?
Ivica.
koliko imaš godina Ivica?
20
20 su lijepe godine , ali 21 je bolje.
Vidimo se kasnije Ivica!
```

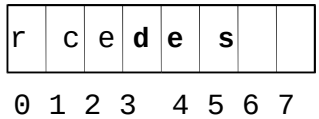
Stringovi imaju mnogo korisnih metode koje su vezane za njih.
Jedna od vrlo korisnih se naziva `substring`. Java koristi konvenciju da su karakteri unutar stringa numerirani s 0, 1, 2, … s lijeva nadesno. Pretpostavimo da st označava string , a da su m i n cjelobrojne vrijednosti. Tada izraz :

```
st.substring(m,n)
```

daje novi string koji se sastoji od karaktera stringa `st`, počevši od pozicije m i završavajući s pozicijom n-1. (n je prva pozicija koja nije uključena !). Npr. ako varijabla `automobil` označava string `"Mercedes"`, onda izraz

```
automobil.substring(5,8)
```

će dati string `"des"`,kako je to dolje pokazano.



Navest ćemo još neke standardne metode koje možemo koristiti u radu sa stringovima. U svakom primjeru st označava string.

`st.length()` Daje broj znakova u stringu ("duljina" stringa st)
Npr. `"Mercedes".length()`
vraća vrijednost 8.

`st.toLowerCase()` Daje string sa svim velikim slovima pretvorenim u mala slova.
Npr. `"Mercedes".toLowerCase()`
vraća string `"mercedes"`

`st.UpperCase()` Daje string sa svim malim slovima pretvorenim u velika slova.
Npr. `"Mercedes".toUpperCase()`
vraća string `"MERCEDES"`

Ako ste izračunali broj i želite ga pretvoriti u odgovarajući string sve što je potrebno je kombinirati prazni string i broj koristeći + operator. Npr. ako varijabla `rezultat` sadrži vrijednost 142, onda izraz `"" + rezultat` će proizvesti string `"142"`.

To radi jer Java koristi pravilo koje kaže da ako kombinirate string i broj pomoću operatora +, broj će biti konvertiran u string, a zatim će oba stringa biti spojena. U ovome slučaju string nastao iz zbroja je spojen s praznim stringom.

U drugom smjeru nije tako jednostavno. Potrebno je koristiti neki od slijedećih metoda.

`Integer.parseInt(st)` Ako string `st` predstavlja cjelobrojnu vrijednost ovo će dati odgovarajuću vrijednost

`Double.parseDouble(st)` Ako string `st` predstavlja broj u pokretnom zarezu ovo će dati odgovarajuću vrijednost.

Na kraju ovog dijela još pokoje pravilo o kombiniranju brojeva i stringova korištenjem + operatora. Razmotrimo slijedeće izraze:

```
"Rezultat je " + 3 + 7

3 + 7 + " je rezultat"
```

na prvi pogled izgleda kao da želimo reći istu stvar. Međutim ako ih ispišemo koristeći metodu `System.out.println` vidjet ćete da ćete dobiti sasvim nešto drugo.

Da biste predvidjeli što će se u ovakvim slučajevima potrebno je koristiti slijedeće pravila:

Ako Java treba odrediti vrijednost izraza x+y gdje su x i y bilo brojevi ili stringovi, tada će se dogoditi slijedeće:

(1) Ako su x i y oboje stringovi onda će ih spojiti.
Ako je jedan string, a drugi broj onda će pretvoriti broj u string i spojiti ga s drugim stringom
Ako su x i y oboje brojevi bit će zbrojeni.

Ako se računa izraz koji sadrži više + operatora izraz se računa slijeva nadesno !

Ponovo razmotrimo izraz:

```
"Rezultat je " + 3 + 7
```

Prvo što će Java obraditi je `"rezultat je " + 3`. Pretvorit će broj 3 u string "3", spojiti ga s prvim stringom i dati `"Rezultat je 3"`. Zatim će obraditi `"Rezultat je 3" + 7`, i konačno dati string:

```
"Rezultat je 37"
```

Sad razmotri:

```
3 + 7 " je rezultat"
```

Ovaj put će Java prvo zbrojiti 3 i 7 i dati broj 10. Zatim će obraditi `10 + " je rezultat "` i konačno dati:

```
"10 je rezultat "
```

7. Zadaci za poglavlje 2

Za ove zadatke kreirajte direktorij P2 unutar direktorija JProg.
Potrebno je unijeti i prevesti i izvršiti 4 zadatka.

1. Zadatak (Koristi naziv klase i datoteke Aritmetika)
   Napiši program koji traži od korisnika da ukuca dva cijela broja i zatim ispiši:

sumu
razliku
umnožak
prosjek (double !)
udaljenost (apsolutna vrijednost razlike)

*1Koristi metodu `Math.abs` za račun apsolutne vrijednosti

2. Zadatak (Koristi naziv klase i datoteke Mjenjacnica)

   Napiši program koji učitava iznos u kunama i preračunava ih u eure (kurs npr. 1 euro = 7.50 kn)

Zadatak. (Koristi naziv klase i datoteke Upoznavanje)
Napiši program koji će ispisati prva dva reda kao što je prikazano, zatim učitati ime i nakon toga ispisati pozdrav. Podebljan je upis od strane korisnika . Ime može biti bilo koje.

```
Bok. Moje ime je Java.
Kako se ti zoveš ?
Ivica
Ivica ! Drago mi je što smo se upoznali.
```

4. Zadatak. (Koristi naziv klase i datoteke Tablica)
   Napiši program koji će nacrtati tablicu na ekranu..

```
+--+--+--+
|  | |  |
+--+--+--+
|  | |  |
+--+--+--+
|  | |  |
+--+--+--+
```

   Učini to na način da kreiraš dvije string varijable za oba uzorka pa onda te varijable naizmjenično ispiši.

3. Objekti i klase

U prva poglavlja koristili smo nekoliko objekata: objekt koji se zove `System.out` za slanje informacija na izlaz, ConsoleReader za čitanje informacija s tipkovnice, i String objekt.
Ovo poglavlje bavi se konstruiranjem novih objekata. Većinom ćemo se baviti konstruiranjem objekta koji će sadržavati informacije o studentu.

Sadržaj:

1. Što je to objekt?

U ranim danima programiranja prvi programski jezici poput Fortrana dozvoljavali su programerima da rukuju s vrlo ograničenim skupom tipova podataka npr. cijeli brojevi, brojevi s pokretnim zarezom i nizovi brojeva.
Nekih 10 godina kasnije programski jezik Simula 67 omogući je programerima definiranje vlastitih tipova podataka nazvanih *objekti*. Objekti su se dokazali kao jedna od najznačajnijih inovacija u programiranju. Danas predstavljaju osnovu programskih jezika poput C++, Java, C# pa čak i Visual Basica.

Što je to objekt ? Postoje dva gledišta na objekt. Jedno sa strane programera koji koristi objekt, a drugo sa strane programera koji dizajner objekt. Prvog programera nazvat ćemo "korisnik", a drugog programera "dizajner" (možda je bolji naziv "konstruktor"). Često se događa da isti programer prvo dizajnira objekt, a zatim ga koristi. Međutim vidjeli smo na primjeru objekata tipa ConsoleReader da je taj objekt dizajnirao neki drugi programer, a mi smo ga vrlo lako koristili.

Razlika u tome što dizajneru predstavlja objekt , a što predstavlja korisniku nije samo ograničena na Javu. Pogledajte na vaš sat(ako ga imate). Ako je dobro dizajniran omogućavat će očitavanje vremena, promjenu vremena, start-stop funkciju štoperice, itd. vama kao korisniku nije potrebno poznavati što se događa unutar sata, koji su to mehanizmi koji stoje iza svih njegovih funkcija. Vi ste "korisnik" sata. S druge strane osoba koja je dizajnirala sat znat će sve što se događa unutra. Dizajneri će potrošiti dosta vremena pokušavajući konstruirati sat da bi vama omogućili korištenje uobičajenih funkcija sata.

Isto je točno za bilo koji uređaj npr. automobil, mobitel, ...

Iako ih ne možete opipati Java objekti su isto tako realni. Programer korisnik je upućen koje su funkcije pridružene pojedinoj vrsti objekata. te funkcije nazivamo **sučelje (interface).** Programer koji je programirao objekt zna koje će se operacije izvršavati kad se te funkcije pozovu. Taj dio nazivamo **implementacija** objekta.

Korisnik će na pitanje "Što je to objekt" odgovoriti s "Ne znam od čega se sastoji već mi je poznato njegovo sučelje i što mogu s njim učiniti". Dizajner će odgovoriti s: "Sastavljen je od varijabli i metoda koje međusobno djeluju da bi realizirale sučelje objekta.

U ovom poglavlju pokušat ćemo objasniti kako objekti izgledaju sa stanovišta dizajnera objekata. U stvari dizajnirat ćemo pojedine objekte.

Svi objekti koji se grade na osnovu istog dizajna pripadaju istoj **klasi**. Dizajn klase je dio programa koji nazivamo **definicija klase.**

2. Klasa Student

Zamislite da ćemo u Javi napisati program koji će u bazi podataka održavati podatke o svim studentima na ovome fakultetu. Program poput toga vjerojatno bi sadržavao mnogo različitih tipova objekata. Jedan o tipova bi sigurno bio onaj koji bi sadržavao podatke o pojedinom studentu. Za očekivat je i postojanje objekta koji bi predstavljao listu svih studenata u bazi podataka.

U ovome poglavlju nećemo pokušavati napisati kompletan program već ćemo se koncentrirati na dizajn objekata i to na objekt koji će sadržavati podatke za pojedinog studenta. klasu koja predstavlja dizajn te vrste objekta nazvat ćemo **Student**.

Ako bismo dizajnirali sat, počeli bismo od ideje kako konstruirati sučelje tj. skup operacija koje sat treba izvoditi. Nakon toga bismo radili na implementaciji sučelja, odnosno kako složiti stavi unutar objekta da se osigura tražena funkcionalnost.

Slično, prvo u definiranju klase Student je odluka koje operacije će biti asocirane s objektom tipa Student.

Da bismo održali jednostavnost primjera uključit ćemo ograničeni skup operacija:

Bit će potrebno omogućiti mijenjanje godine studenta s 1 na 2 ili 2 na 3. Ova akcija će se normalno odvijati na kraju godine.

Bit će potrebno omogućiti promjenu programa studija (stupnja)

Bit će potrebno omogućiti način da objekt vrati ime studenta kako bi program mogao ispitati kojemu studentu pripada odgovarajući objekt. Dobivanje ostalih informacija zasad nećemo realizirati.

Bit će potrebno omogućiti ispisivanje podataka o studentu na ekran. Osigurat ćemo metodu `prikaz` koja će to činiti.

Bit će potrebno kreirati nove objekte tipa Student. Svaki put kad bude kreiran bit će potrebno dati informacije o studentu, npr. ime, program studija (stupanj) , ...

Sad je vrijeme za definiciju klase student. U terminologiji ručnog sata, potrebno je odrediti kako će pojedine komponente biti raspoređene i povezane unutar kućišta da bi proizvele sat koji radi.

Svaki objekt u Javi sastavljen je od nekog broja odvojenih dijelova nazvanih **članovi (members)**. Postoje tri vrste članova.

- **Varijable**. U ove članovi spremaju se informacije koje objekt sadrži. Npr. Student objekt može sadržavati varijablu koje sadrži studentovo ime i drugu koja sadrži naziv programa studija. Skup vrijednosti koje sadrže varijable pojedinog objekta nazivaju se **stanje (state)** objekta.
  Da bismo razlikovali te varijable od varijabli koje smo definirali unutar metoda (u prošlom poglavlju), varijable koje pripadaju objektu nazivamo **polja (fields)** ili varijable instance (**instance variables**).

Metode. Ovi članovi izvršavaju operacije na objektu (ili s objektom). Npr. vidjeli smo da ConsoleReader objekt ima metodu koja se naziva `ReadInt` koja koristi taj objekt da bi se dobila vrijednost koju je korisnik utipkao. Kada napišemo definiciju klase Student imat ćemo metodu `uNovuGodinu` koja će dodavati jedan na tekuću studentovu godinu.

Ova vrsta metoda nazivaju se **metode instance (instance methods)** kako bi ih razlikovali od statičkih metoda koje ne pripadaju nijednom objektu (više o statičkim metodama kasnije

Konstruktori. Ovi članovi se koriste da bismo konstruirali objekte određene klase. U klasu student uključit ćemo jedan konstruktor jer inače ne bi bilo moguće napisati program koji bi koristio objekte te klase. Konstruktori su vrlo slični metodama na mnogo načina.

Sad ćemo specificirati koje će članove posjedovati objekti tipa Student. Prvo ćemo definirati polja (fields) . Te varijable sadržavat će informacije koje su pohranjene u svakom objektu tipa Student. Pretpostavimo za sada 4 informacije (slaže se sa zahtjevima operacija navedenih prije.)

1. Studentov ID.
2. Ime.
3. Naziv programa.
4. Godina studija.

Potrebno je odabrati nazive za ta polja i odlučiti koji će se tip podataka koristiti.
Npr. ID broj može biti cijeli broj ili string. ID broj je u stvarnosti samo niz karaktera. Neće biti nikakve razlike ako se sastoji od niza znakova umjesto od znamenaka. Zbog toga ćemo odabrati tip podatka `String` . Ime studenta i naziv programa studija također će biti tipa `String`. Godina studija bit će cjelobrojna vrijednost tipa `int`.

Slijede definicije polja. Linije koje počinju s `//` su komentari koji objašnjavaju sadržaj svakog polja. Java ignorira sve od `//` do kraja reda.

```
 private String idBroj;
         // ID broj Studenta.

 private String ime;
         //  studentovo ime.

private String programStudija;
    //Program studija.

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

Riječ `private` određuje da se polju ne može direktno pristupiti izvan definicije klase Student. Ako programer "korisnik" objekta napiše kod koji zove takvu varijablu prevodioc će javiti grešku prevodioca. Polja u klasi su obično označena kao `private.` (Kad dođemo do metoda asociranih sa objektom Student, vidjet ćemo da počinju s riječju `private` koja znači da to metodu može pozvati korisnik objekta.

Nakon riječi `private` dolazi tip vrijednosti koje spremamo u polje te naziv polja. Cijela linija predstavlja deklaraciju varijable. Moguće je uključiti i inicijalnu vrijednost.

Kad smo odabrali polja koja će objekt sadržavati, sad trebamo napisati metode koje će izvršavati osnovne operacije pridružene objektu.

Prvo ćemo konstruirati metodu koja će se koristiti za promjenu studentovog programa studija. Nazvat ćemo je **`setProgramStudija`**. Kad je pozovemo bit će potrebno znati koji je novi naziv novog programa studija. Ta dodatna informacija bit će **parametar** metode. Općenito metoda može imati bilo koji broj parametara. Svaki put kad pozovemo metodu svakome od tih parametara bit će dodijeljena vrijednost koju će moći koristiti tijekom izvršavanja metode.

`setProgramStudija` ima samo jedan parametar – naziv studentovog novog programa. U definiciji metoda taj parametar nazvat ćemo jednostavno `p`.

Tijelo svakog parametra sastojat će se od niza izraza koji će izvršavati potrebne akcije. U ovom slučaju potrebna je samo jedna akcija: novi naziv programa studija (dakle parametar p) potrebno je pohraniti u polje `programStudija` . Slijedeće dodjeljivanje će obaviti taj zadatak:

```
programStudija = p;
```

Slijedi kompletna definicija metoda `setProgramStudija`:

```
/* Promijeni programStudija u p. */

public void setProgramStudija(String p)
{  programStudija = p;
}
```

Kao obično počinjemo s komentarom što program radi. Slijedeća linija je zaglavlje metoda .
Svaka od riječi ima značenje:

`public` kaže da se metoda može koristiti bilo gdje u programu, dakle i unutar klase i van definicije klase

`void` kaže da metoda ne vraća nikakvu vrijednost.

`setProgramStudija` je naziv metode.

`(String p)` je lista svih parametara metode. U ovome slučaju radi se o samo jednom parametru koji je tipa `String` , a naziva se `d`.

Na kraju imamo tijelo metoda zatvoreno u vitičaste zagrade{…}. U ovome slučaju tijelo metoda sastoji se samo od jedne naredbe.

Pretpostavimo da programer korisnik piše nekakav program u nekoj drugoj klasi odvojeno od klase student. Pretpostavimo da je kreirao određeni objekt studenta i pridodijelit ga varijabli `stud1` . Tada ako programer želi postaviti za tog student program studija na npr. na strojarstvo može pisati naredbu:

```
stud1.setProgramStudija(”strojarstvo”);
```

Ovaj naredba kaže: izvrši metodu `setProgramStudija` koja je asocirana s objektom `stud1` (tipa `Student`) koristeći vrijednost parametra "strojarstvo".

Rezultat će biti će polje objekta `programStudija` biti postavljeno na vrijednost `”strojarstvo”`.

Slijedeće ćemo definirati metodu koja dodaje jedan na studentovu godinu studija. Ovo je još jednostavnije jer metoda nema nikakvih parametara. Cijelo tijelo je izraz koji dodaje jedinicu na polje `godina.`

```
godina++;
```

Slijedi kompletna metoda.

```
/* Povećaj godinu studija za jedan */

public void povecajGodinu()
{  godina++;
}
```

Što kaže zaglavlje:

`public` znači da se metoda može koristiti bilo gdje u programu.

`void` znači da metoda ne vraća nikakvu vrijednost.

`povecajGodinu` je naziv metode.

`()` pokazuje da nema parametara. To znači da se ne prosljeđuje nikakva informacija metodu koji se poziva.

Van klase student slijedeći izraz može biti korišten za dodavanje jedinice na broj godine studenta. Koristimo opet objekt `stud1` (klase Student)

```
stud1.povecajGodinu();
```

Slijedeća metoda vraća ime koje je pohranjeno u objektu tipa Student. Podsjetite se da se toj varijabli ne može pristupiti direktno izvan klase jer je polje `ime` označeno kao `private`.
Metodu smo nazvali `getIme`. Za razliku od dva prethodna metoda ovaj "vraća vrijednost". Vrijednost se vraća na mjesto poziva metoda. Pretpostavimo da imamo slijedeću naredbu:

```
String st = stud1.getName();
```

Java će kreirati varijablu `st` te nakon toga pozvati metodu

```
stud1.getName()
```

Metoda će biti pozvana za objekt `stud1`. (metoda nema parametara) Metoda će vratiti ime studenta, i Java će ga pohraniti u varijablu `st`. Primijetite da je `stud1.getName()` korišten na isti način kao što je korišteno npr. `Math.sqrt(1.5)`

```
double x = 1 - Math.sqrt(1.5);
```

Ovdje je kompletna metoda:

```
/* Vrati studentovo ime. */

public String getIme()
{  return ime;
}
```

Zaglavlje znači:

`public` kaže da se metoda može koristiti bilo gdje.

`String` kaže da metoda vraća vrijednost tipa `String`.
Ovo se naziva **vraćeni tip podatka (return type)**.

`getIme` je naziv metoda.

`()` pokazuje da metoda nema parametara.

Tijelo metoda je jedna naredba:

```
return ime;
```

Ona kaže: vrati vrijednost polja `ime`. To je tip naredbe koji nismo dosada sreli. Naziva se **naredba vraćanja vrijednosti(return statement).**

Zadnja metoda ispisuje na ekran sve informacije o studentu. Nazvat ćemo je `prikaz`

```
/* Prikaži informacije o studentu na ekran. */

public void prikaz()
```

```
{  System.out.println
      ("Student ID: " + idBroj);
   System.out.println
      ("Ime: " + ime);
   System.out.println
      ("Program studija: " + programStudija);
   System.out.println
      ("Godina: " + godina);
}
```

Zaglavlje pokazuje da metoda ne vraća nikakvu vrijednost i nema parametara pa će poziv metoda izgledati ovako:

```
stud1.prikaz();
```

Naposljetku treba napisati konstruktor koji će programer korisnik moći upotrijebiti za stvaranje novih objekata tipa `Student`. Konstruktor uvijek ima naziv identičan nazivu klase. U ovom slučaju konstruktor se zove `Student`. Konstruktor nema naznaku što vraća jer je očito što bi trebao vratiti, a to je objekt koji se njime kreira. Konstruktor može imati bilo koji broj parametara. Ovaj ovdje ima tri: studentov ID broj, ime i naziv programa studija. Ovdje je pretpostavljeno da će godina studija biti 1 kada je objekt kreiran.

Slijedi konstruktor:

```
/* Kreiraj novog studenta sa zadanim
   ID brojem, imenom i programom studija
   Polje godina bit će postavljeno na 1.
*/

public Student(String id, String im, String p)
{  idBroj = id;
   ime = im;
   programStudija = p;
   godina = 1;
}
```

Sa dana tri parametra konstruktor će:

1. stvoriti novi objekt,
2. pohraniti string `id` u polje `idBroj` novostvorenog objekta,
3. pohraniti string `im` u polje `ime` ,
4. pohraniti string `p` u polje `programStudija` ,
5. pohraniti 1 u polje `godina`.

Primijetite da ne postoji ništa u tijelu konstruktora što indicira prvi korak stvaranja objekta. Java to radi automatski kada izvršava konstruktor.

Slijedi navedeni konstruktor iskorišten u stvaranju objekta tipa Student:

```
Student stud1 =
   new Student(”9912345”, ”Ivo Petrić”, ”strojarstvo”);
```

Ovo znači: prvo kreiraj varijablu nazvanu `st` koja će se odnositi na Student objekt. Zatim kreiraj novi objekt tipa student s podacima naznačenim u listi parametara.
Primijetite upotrebu ključne riječi `new`. Potrebno ju je upotrijebiti svaki put kada s konstruktorom stvaramo novi objekt. Svaka varijabla koja će označavati objekt tipa Student mora biti deklarirana kao varijabla tipa Student. (vrijedi za sve tipove objekata)

S ovim smo kompletirali definiciju klase Student. Još ostaje dodati zaglavlje cijele klase. U klasi su prvo stavljena polja iako redoslijed članova nije bitan.

PRIMJER1

```
/* Student objekt za studenta FESB-a
*/

{  public class Student

     private String idBroj;
         // ID broj Studenta.

     private String ime;
         //  studentovo ime.

    private String programStudija;
        //Program studija.

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


   /* Promijeni programStudija u p. */

   public void setProgramStudija(String p)
   {  programStudija = p;
   }

    /* Povećaj godinu studija za jedan */

    public void povecajGodinu()
    {  godina++;
    }

    /* Vrati studentovo ime. */

    public String getIme()
    {  return ime;
    }

    /* Prikaži informacije o studentu na ekran. */

    public void prikaz()

    {  System.out.println
          ("Student ID: " + idBroj);
       System.out.println
          ("Ime: " + ime);
       System.out.println
          ("Program studija: " + programStudija);
       System.out.println
          ("Godina: " + godina);
    }

    /* Kreiraj novog studenta sa zadanim
       ID brojem, imenom i programom studija
```

```
            Polje godina bit će postavljeno na 1.
         */

         public Student(String id, String im, String p)
         {  idBroj = id;
            ime = im;
            programStudija = p;
            godina = 1;
         }

    }
```

Konačno jedan veoma jednostavan program koji će koristiti objekt tipa Student. Poput programa iz prošlog poglavlja sastoji se od samo jedne statičke metode `main` .On je sam stavljen u klasu koju ćemo nazvati TestStudent. Metoda prvo čita informacije o studentu unesene preko tipkovnice. Nakon toga kreira objekt tipa Student koristeći konstruktor i varijablu `st.` Nakon toga povećava godinu studija za jedan. koristeći izraz:

```
st.povecajGodinu();
```

Kako je konstruktor već postavio tu vrijednost na 1 sad bi trebala biti 2.

Zatim se program studija promijeni s izrazom:

```
st.setProgramStudija(”elektronika”);
```

Na kraju se podaci o studentu ispišu na ekran.

```
st.prikaz();
```

Slijedi `TestStudent` klasa:

PRIMJER 2

```
public class TestStudent

{  /* Učitaj podatke o studentu.
      Kreiraj objekt tipa student.
      Prikaži ne ekranu podatke objekta.
   */
   public static void main(String[] args)
   {  /* Kreiraj ConsoleReader za učitavanje
         što korisnik tipka */
      ConsoleReader in =
          new ConsoleReader(System.in);

      /* Učitaj podatke o studentu i kreiraj
          objekt za njih. */

      System.out.println
         ("Koji je ID broj studenta?");

      String i = in.readLine();

      System.out.println("Koje je ime studenta?");

      String n = in.readLine();
```

```
        System.out.println("Koji je program studija?");

        String d = in.readLine();

        Student st = new Student(i,n,d);

        /* Povećaj broj godine studija
           i prikaži informacije na ekranu. */
        st.povecajGodinu();
        st.setProgramStudija(”elektronika”);
        System.out.println();
        st.prikaz();
    }
}
```

Ovo se može pojaviti na ekranu tijekom izvođenja programa:

```
Koji je ID broj studenta?");
```
9912345
```
Koje je ime studenta?
```
Ivo Petric
```
Koji je program studija?
```
Strojarstvo

```
Student ID: 9912345
Name: Ivo Petric
Program Studija: elektronika
Godina: 2
```

3. Više o klasama

U tijeku izvršavanja vaših Java programa, bit će kreirani različiti objekti i bit će pozivane neke od njihovih metoda. Konstruktori će se koristiti za kreiranje objekata, ali neće se pozvati niti jedan izraz za njihovo uklanjanje. Objekte će ukloniti iz memorije interpreter u trenutku kad mu zatreba više memorije. Intepreter provjerava sve postojeće objekte i briše one objekte na koje ne upućuje niti jedna varijabla ili drugi objekt. Taj proces naziva se "odvoz smeća" (garbage collection).

Proces kreiranja objekata i izvršavanja metoda počinje unutar `main` metode. Kada pozovete intepreter s naredbom:

java klasa

gdje je klasa prevedena klasa (bytecode), intepreter će poći do definicije klase i tražiti metodu sa zaglavljem

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

(Naziv parametra može biti bilo koji naziv.) Ako intepreter nađe takvu metodu počet će izvršavati njene naredbe. Da bi uspješno upravljao s potrebnim objektima i metodama, intepreter mora imati pristup definicijama svih objekata koje treba koristiti. Neke od tih klasa ćete i sami napisati. Neke će biti napisane u Java biblioteci, poput PrintStream klase koja sadrži `System.out.println` metodu. Neke klase će napisati drugi programeri, poput ConsoleReader klase.

Npr. za uspješno pokretanje programa iz primjera 2 osim klasa iz biblioteka potrebne su još tri klase: TestStudent, Student i ConsoleReader. One će se obično nalaziti u tri datoteke nazvane: `TestStudent.java`, `Student.java` i `ConsoleReader.java`.

U procesu prevođenja bit će dovoljno izvršiti naredbu

```
javac TestStudent.java
```

Prevodioc će naći u klasi `TestStudent` reference ne dvije ostale klase i automatski ih prevesti. Napišite slijedeću naredbu:

```
java TestStudent
```

intepreter će doći do `TestStudent` klase i potražiti public, static metodu s nazivom `main`. Ako je uspije pronaći pokušat će je izvršiti. Ako je ne nađe javit će odgovarajuću pogrešku i stati s interpretiranjem.

Da bismo napravili program kompaktnijim moguće je staviti `main` metodu u klasu `Student` i izbjeći stvaranje `TestStudent` klase. Tada ćete ukucati naredbu :

```
java Student
```

To će natjerati intepreter da u klasi `Student` traži `main` metodu.

Kada pogledate na programski kod koji opisuje različite članove klase vidjet ćete da definicija svakog člana počinje s ključnom riječi `public` ili `private`.

Ako je član neke klase označen kao `public` tada mu se pristupiti iz bilo koje metode (ili konstruktora) bilo gdje u programu. Ako je označen kao `private` onda mu se može pristupiti samo iz metoda koje su članovi iste klase. To znači da kada pišete definiciju klase A onda u nju ne možete uključiti izraze koji pozivaju članove klase B koji su označeni kao `private` . Možete pristupiti samo `public` članovima objekta B. Članovi označeni s `public` sačinjavaju sučelje objekta (interface).

Postoji niz razloga zbog kojih bi programer koji dizajnira klasu ograničio pristup drugome programeru svim poljima i metodama unutar klase. Razlog je da se korisniku klase koji ne poznaje u potpunosti rad klase ograniči mogućnost krive upotrebe klase. S druge strane kad je implementirano sučelje klase ista se može isporučiti korisniku, a za kasnije ostaviti poboljšanje unutarnje funkcionalnosti klase poput ubrzanja i slično. Naravno treba paziti da se ne promijeni sučelje.

Java omogućava da se izostavi riječ `public` ili `private`. Tada će polje ili metoda biti tretirana kao da je `public`.

Dosad smo diskusiju o klasama u ovom poglavlju zasnivali na ideji da je definicija klase opis određenog objekta. To je u određenoj mjeri i točno, ali klase su ipak malo kompliciraniji pojam. Svaka klasa ima niz varijabli i metoda koje egzistiraju neovisno u bilo kojem određenom objektu. Takvi članovi klase poznati su kao **statička** polja i metode. Drugi naziv je s engleskog neprevodiv – class fields and methods. U njihovim definicijama pojavljuje se ključna riječ `static`.

Počeli smo poglavlje 1 s programom koji se sastojao od samo jedne klase (`Hello`), i sadržavao je samo jednu metodu `main`. Zaglavlje metode je bilo:

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

Upotreba riječi `static` ovdje indicira da ova metoda nije asocirana s niti jednim objektom. U stvari kada se program pokrene ni jedan objekt tipa klase Hello neće biti kreiran !
Uloga ove klase je samo da omogući prikladno mjesto za `main` metodu.

Druga klasa od koje se ne proizvode objekti, ali je sačinjena od statičkih polja i metoda je Math klasa iz Java biblioteke funkcija. Ona sadrži metode koje računaju uobičajene matematičke funkcije poput sinusa i kosinusa. Slijedi primjer izraza koji koristi navedenu klasu:

```
double x = Math.sin(a);
```

Dosad smo vidjeli da kada pozivate metodu koja pripada instanci neke klase da se prvo piše naziv objekta koji pokazuje na objekt neke klase. U slučaju pozivanja statičkog metoda (koji nije asociran s niti jednim objektom) **piše se ispred naziva metoda naziv klase,** dakle u ovom slučaju `Math`.

4. Reference na objekte (Object References)

U prethodnom poglavlju spomenuli smo da se vrijednosti poput cijelih brojeva i brojeva u pokretnom zarezu tretiraju na drukčiji način nego objekti. Jedna od različitosti je način na koji se pohranjuju u memoriji. Vrijednost varijable primitivnog tipa spremljena je u samoj varijabli. Pretpostavimo da je izvršen slijedeći izraz:

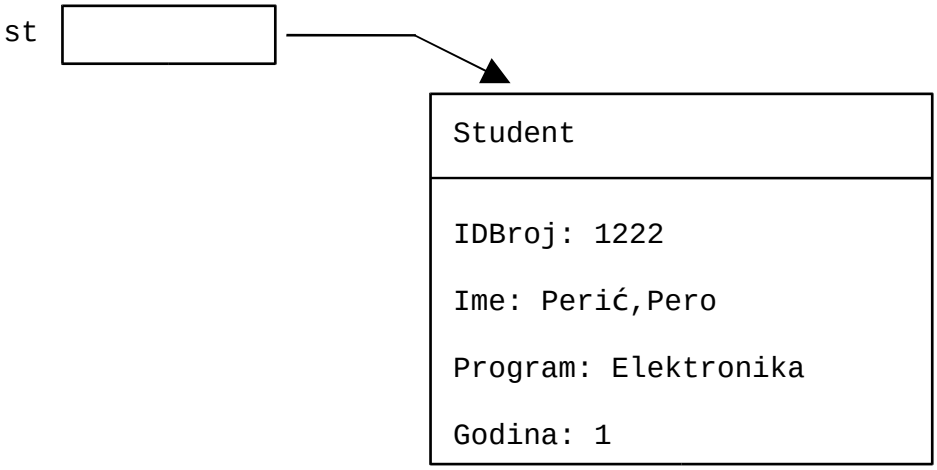
```
int n = 5;
```

Nakon ovoga izraza 32 bita memorije su rezervirana za novu varijablu `n`, i vrijednost 5 je spremljena na to mjesto u memoriji.
Ovaj postupak možemo vizualizirati na slijedeći način: (Pravokutnik pokazuje rezervirano mjesto u memoriji)

Kako smo vidjeli u odjeljku o String klasi, objekt se ne sprema u varijablu. Pretpostavimo da je izvršena slijedeća naredba:

```
Student st =
   new Student(”1222”, ”Perić,Pero”, ”Elektronika ”);
```

Kao i prije određeno mjesto u memoriji bit će alocirano za novu varijablu. U ovom slučaju to je varijabla `st` . Međutim **objekt** koji se kreira u desnom dijelu izraza **neće biti spremljen na mjesto varijable**. Za njega će se rezervirati posebno mjesto u memoriji. Vrijednost koja je pohranjena u `st` je **vrijednost reference. Referenca** je vrijednost koja jednoznačno određuje gdje je u memoriji lociran objekt (u ovom slučaju objekt tipa student).
U slijedećem dijagramu strelica od varijable `st` do objekta tipa Student znači da varijabla sadrži referencu na objekt.



Kada pokrenemo Java program svaka vrijednost bit će ili primitivna vrijednost ili vrijednost reference. To vrijedi za vrijednosti pohranjene u varijablama, vrijednosti vraćene pomoću metoda i vrijednosti proslijeđene kao parametri.

Nikad nećemo moći vidjeti stvarnu vrijednost reference. Nemojte misliti da su to cijeli brojevi. Sve što znamo je da samo varijabla koja je npr. tipa Student može sadržavati referencu na objekt tipa Student.

Postavlja se pitanje da li ima razlike u tome da varijabla sadrži referencu na objekt, a ne sam objekt ? Ima. Razmotrimo slijedeći kod:

```
Student st1 =
   new Student(″1222″, ″Perić,Pero″, ″Elektronika″);

Student st2 = st1;

st1.setProgramStudija(″Strojarstvo″);

st2.prikaz();
```

Što će se prikazati na ekranu. Što će biti ispisano za programStudija. Odgovor je da će to biti "Strojarstvo". Da bi to znali moramo razmotriti liniju po liniju:

1. `Student st1 = new Student(″1222″, ″Perić,Pero″, ″Elektronika″);`

   Ovo će kreirati varijablu `st1`, zatim objekt tipa Student koji će sadržavati zadane podatke te naposljetku u `st1` bit će spremljena referenca na kreirani objekt.

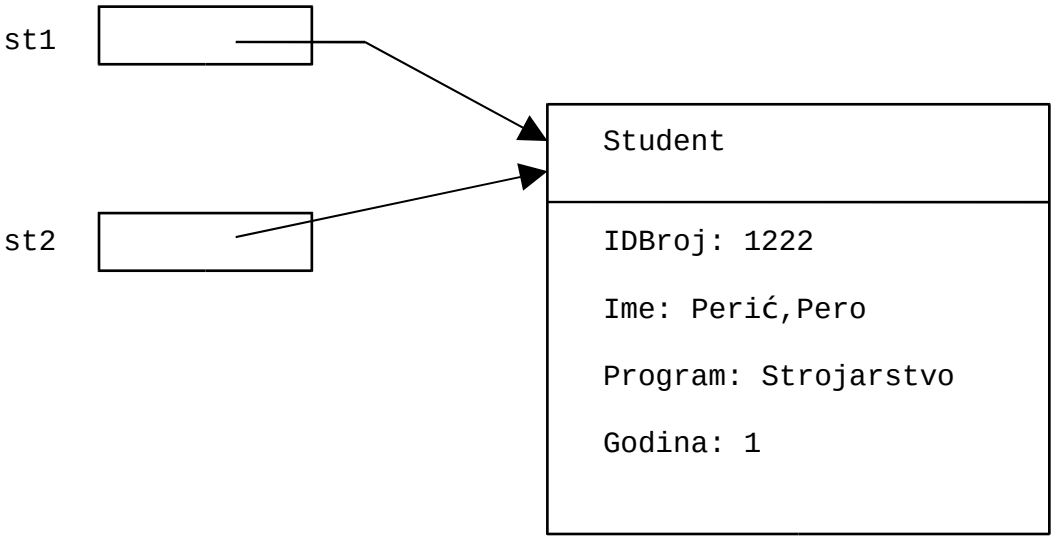
2. `Student st2 = st1;`

   Ova linija će kreirati varijablu `st2` i dodijeliti joj vrijednost koja se nalazi u varijabli `st1`, u ovom slučaju referencu na objekt student koji je kreiran u prvom koraku. **Primijetite da iako postoje dvije varijable samo je jedan Student objekt.** Obje varijable su reference na isti objekt.

3. `st1.setProgramStudija(″Strojarstvo″);`
   Polje `programStudija` objekta na koji referencira varijable `st1` mijenja se u string `″CS″`.

4. `st2.prikaz();`

   Prikazuju se detalji objekta na kojeg referencira `st2` .



Da se objekti spremaju u varijable (**a nisu !**) prva naredba bi spremila objekt Student u st1. Druga bi naredba napravila kopiju istog objekta u st2. Tada bi postojala dva objekta tipa student. Treća bi promijenila polje programStudija u objektu st1 i ne bi promijenila objekt u st2. Četvrta naredba bi na kraju ispisala detalje objekta u st2, uključujući originalni naziv studija "Elektronika".

Primjeri poput ovoga stalno se događaju u Java programiranju. Stoga je vrijeme da se počne misliti u terminima referenci.

5. `null` vrijednost

Ranije smo rekli da varijabli tipa student može biti pridružena samo referenca na objekt tipa Student. To nije sasvim točno. Postoji jedna vrijednost koja može biti pridružena svakoj varijabli koja inače sadržava vrijednost reference. Ta vrijednost se označava sa `null`. Slijedi deklaracija koja kreira varijablu tipa Student i sprema u nju vrijednost `null`:

```
Student st = null;
```

Za primjer gdje se vrijednost null može koristiti uzmimo pretraživanje baze gdje imamo metodu koja pretražuje cijelu bazu podataka o studentima i traži objekt s određenim ID brojem. Ako se objekt s takvim ID brojem nađe vratit će se referenca na taj objekt. Međutim što vratiti ako ne postoji objekt s tim ID brojem. U tom slučaju bit će uobičajeno da ta metoda vrati vrijednost `null`.

Ako varijabla poput prijašnjeg primjera varijable st sadrži vrijednost `null` , potrebno je osigurati da se ne pokuša pristupati metodama koje pripadaju objektu na kojeg se odnosi varijabla st. (varijabla st ne odnosi se u ovom slučaju na nikakav objekt)

Npr. slijedeća naredba neće biti izvršena:

```
st.setProgramStudija(″Strojarstvo″);
```

`st` ne referencira na nikakav kreirani objekt tipa Student te će Java intepreter javiti run-time pogrešku. Greška će biti indicirana kao `NullPointerException`. Naravno, ako varijabli `st` pridružimo objekt ta će se ista naredba bez problema izvršiti.

Još jedno mjesto gdje Java može uvesti vrijednost null je kod kreiranja objekta konstruktorom. U primjerima smo preko argumenata konstruktora prenosili argumente koji su nam poslužili kao inicijalne vrijednosti za polja objekta.

Što se događa kada ne specificirate inicijalnu vrijednost polja u objektu. Java će izabrati sama početnu vrijednost. Ako je polje neki broj onda će biti inicijaliziran na nulu. Ako polje je referenca na objekt početna vrijednost će biti `null.`

Npr. tri polja objekta Student su tipa `String.` **Stringovi su objekti**. Tako ta polja sadrže reference na mjesto gdje su stvarno u memoriji pohranjeni ti karakteri. Ako ne specificiramo inicijalne vrijednosti za njih one će sadržavati `null` vrijednost. To ne znači da se radi o praznom stringu ! Varijabla može referencirati na prazan string , ali to je regularan string kao i svi drugi.

Dakle **poljima** koja nisu inicijalizirana Java pridružuje pretpostavljene početne vrijednosti (default values). To je različito od onoga što se javlja s deklaracijama varijabli unutar tijela metoda. Te varijable se ne inicijaliziraju automatski ! U njima je nepredvidljiva vrijednost ako ih sami nismo inicijalizirali.

6. Identifikatori

Nazivi koje koristimo u Java programima za varijable, metode ili klase nazivamo **identifikatori**. Pravila za pisanje identifikatora u Javi su sljedeća:
mora početi s slovom ili oznakom valute ($,£,...), ili povlakom (_)
ostali znakovi mogu biti slova ili brojevi
dužina nije ograničena
ne mogu se koristiti ključne riječi kao identifikatori (poput `class`, `int`, `private, return,...`)

U Javi je pojam brojki i slova dosta širok. Slovo može biti korejsko, grčko, japansko. To je zbog toga što Java znakove čuva kao 16-bitne *unicode* vrijednosti. To je ogroman skup znakova koji uključuje znakove većine svjetskih jezika. Naše tastature omogućavaju unos limitiranog skupa znakova.

Poput čvrstih pravila za identifikatore, postoje i konvencije. Jedna je da nazivi varijabli, parametara i metoda počinju s malim slovima, a nazivi klasa počinju s velikim slovima. Te konvencije su tako širom prihvaćene da će vas ostali programeri čudno gledati ako ih se ne držite.

Druga konvencija je da ako koristite nazive sastavljene od više riječi da svaku riječ poslije prve počinjete s velikim slovom npr. ConsoleReader , readDouble. Java programeri vole duge nazive pa Java biblioteka sadrži nazive poput GridBagLayoutManager i NoSuchElementException. Korisno je koristiti duge nazive jer se onda može iz toga vidjeti funkcionalnost varijable ili metode. Međutim kod u cjelini postaje nečitljiviji.

7. Zadaci za poglavlje 3

Proširite klasu Student opisanu u odjeljku 3 tako da povećate količinu informacije koja je zapisana u svakom objektu tipa Student. Dodatni stavci bi trebali biti: ime studentovog mentora i broj bodova koje je stekao na osnovu polaganja ispita. Za svaki položeni ispit student dobiva 0.5 bodova. Proširena klasa treba osigurati i slijedeće operacije koje ćemo pozvati iz test programa:

Metoda nazvana `promijeniMentora(m)` koja će promijeniti ime mentora u ime m.

Metoda `dodajBod(n)` koja će dodati broj bodova na ukupan broj bodova koje student ima .

Na kraju trebat će modificirati konstruktor i ubaciti naziv mentora kao parametar. Inicijalna vrijednost broja bodova treba biti 0.

4. Grafika

Rane verzije Jave omogućavale su veoma limitirane grafičke sposobnosti. Java 2 je omogućila mnogo bogatiji skup grafičkih operacija. Od tog skupa koji obuhvaća crtanja počevši od najjednostavnijih linija, preko drugih geometrijskih oblika pa sve do manipulacije s fotografijama u ovome poglavlju preći ćemo maji dio.

Sadržaj:

1. Korištenje grafike

Grafika je tema koju je zgodno uključiti rano unutar predavanja o programiranju jer programeri vole raditi s grafikom, a Java klase koje se odnose na grafiku daju dobar primjer mnogih osobina Jave koje ćemo i dalje susretati na ovim predavanjima. Cijena koju treba platiti je da je potrebno uvest drugačiji način pokretanja programa nego što je to za programe koji nisu koristili grafiku.

Postoje dva načina na koji se mogu pokrenuti Java programi: **aplikacije** i **appleti**. Svi programi koje smo dosada analizirali bili su aplikacije. To znači da su bili pokretani od strane Java SDK interpretera (java) i izvršavanje je započinjalo unutar `main` metode.
Sve aplikacije koje pišemo unutar ovih predavanja ispisivale su u prozor konzole (DOS prozor) koji je prikladan samo za ispisivanje karaktera. Ako bi aplikacija trebala proizvesti neku grafiku bilo bi potrebno da kreira grafički prozor. To je u Javi moguće, ali nećemo se time baviti u ovim predavanjima.

Applet je Java program koji se izvršava unutar Web pretraživača, poput Internet Explorera , ili Netscape pretraživača. Kada Web pretraživač stavlja sadržaj stranice na ekran i ako ta stranica sadrži link na applet, pretraživač će odvojiti dio stranice odnosno područje prikaza (display area) gdje će applet moći prikazati bilo kakvu informaciju, uključujući i grafiku.

Nakon toga će pretraživač započeti s izvršavanjem appleta. To će moći jer sadržava vlastiti Java interpreter.

Kraće rečeno: napišite applet koji može iscrtavati grafiku, napišite Web stranicu koja sadrži link na istu i zatražite od pretraživača da prikaže Web stranicu.

Postoji određeni problem s ovim pristupom. Pojedini pretraživači ne izvršavaju Java 2 programe.
Bilo da slabo prate napredovanje Jave ili namjerno ne podržavaju Javu. Dva su načina da riješimo taj problem:

Rješenje 1. Nabavite *plug-ni* za pretraživač koji će omogućiti pokretanja Java 2 programa.

Rješenje 2. Koristite **appletviewer**. Ovaj program je dio SDK. Pokrenut će bilo koji applet.

Appletviewer ima prvenstvenu namjenu testiranja funkcionalnosti appleta, ali predstavlja najjednostavniji način da vidimo što naš grafički program iscrtava.

Dakle, kada želimo kreirati neki grafički ispis, napisat ćemo applet koji crta grafiku u područje prikaza, zatim napisati malu Web stranicu koja samo sadržava link na applet. Zatim ćemo iskoristiti appletviewer za procesiranje Web stranice.

Jedini nedostatak ovoga pristupa je da appletviewer ne prikazuje ništa osim onoga što proizvede applet. dakle nećemo se baviti kreacijom Web stranica.

Sve što je potrebno znati o HTML za potrebe predavanja je slijedeće.
Web stranica sastoji se od tekst datoteke koja koristi HTML notaciju. U toj datoteci bit će samo link na datoteku Prog.class u kojoj će biti prevedeni Java applet kojega će izvršiti.

```
<APPLET CODE="Prog.class" WIDTH=300 HEIGHT=300>
</APPLET>
```

Osim linka navedena je i veličina područja prikaza koje će biti dodijeljeno appletu na korištenje:

```
WIDTH=300 HEIGHT=300
```

2. Applet koji crta linije

Applet je također jedna Java klasa. U mnogim pogledima je nalik na klase koje smo koristili dosada. Sastavljen je od polja, metoda i konstruktora. Međutim razlikuje se od klasa koje smo dosada koristili u jednoj bitnoj stvari. Posjeduje najmanje jedna metoda koja je namijenjena da je koristi *windows manager* . Windows manager je objekt koji održava prozore na ekranu, omogućavajući njihovo otvaranje, skrivanje(minimiziranje), zatvaranje , itd..

Jedna od metoda koje ćemo priskrbiti za windows manager je metoda *paint*. Windows manager će prvi put pozvati tu metodu kad se područje prikaza pojavi na ekranu. Također kad god bude bilo potrebno osvježiti sadržaj ekrana bit će pozvana metoda paint. Osvježavanje prikaza bit će potrebno npr. kad pomicanjem(skroliranjem) Web stranice područje prikaza izađe izvan vidljivosti pa se novim pomicanjem opet vrati na ekran. U tom slučaju sadržaj područja prikaza mora se ponovo iscrtati odnosno osvježiti. Drugi primjer kad je to potrebno je kad preko područja prikaza npr. otvorimo prozor neke druge aplikacije. Kada zatvorimo taj prozor opet će biti potrebno ponovo osvježiti sadržaj koji je dotada bio pokriven prozorom.

Zadatak `paint` metode je da isporuči (render) sadržaj u područje prikaza (nacrta linije, oboji područja, napiše tekst ,...). Da bi to napravili moramo koristiti jedan objekt koji nazivamo **Graphics2D** objekt. To je nešto slično kao `System.out` objekt koji smo koristili za ispis karaktera u prozoru konzole. Međutim Graphics2D objekt spojen je na područje prikaza appleta (visoka rezolucija), a ne primitivni DOS prozor.

Graphics2D objekt posjeduje niz metoda kao npr. `drawString` kojim možemo ispisivati tekst, metodu `draw` koja može iscrtavati oblike poput kruga, kvadrata, metodu `drawImage`
koju možemo koristiti za ispis kompletnih slika. Također Graphics2D objekt posjeduje metode kojima se može zadati određena boja ili uzorak kojom će se crtati neki lik ili ispisivati tekst. Posjeduje i metode kojima se može zadati neka grafička transformacija lika npr. rotacija.

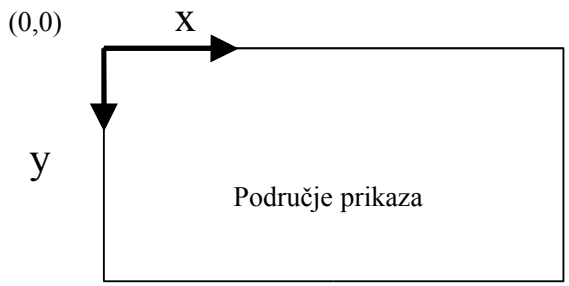
Graphics2D objekt ne moraju biti spojeni samo na područje prikaza na ekranu. Možemo ih spojiti npr. na printer ili na neko područje u memoriji (slika će biti pohranjena u memoriju).

Pretpostavimo da je `g2` objekt tipa Graphics2D spojen na područje prikaza appleta. Pretpostavimo da je `line1` linija između dvije točke. Slijedeća naredba će prikazati liniju na ekranu:

```
g2.draw(line1);
```

Postoje različiti geometrijski oblici koji mogu biti iscrtani ili obojani korištenjem Graphics2D objekta. Koristit ćemo 4 tipa: linije, pravokutnike (rectangles), elipse (uključujući i krugove) te dijelove elipsi tj. lukove (arcs). Svaki od ovih geometrijskih oblika u Javi je reprezentiran kao klasa. Npr. linija je reprezentirana objektom tipa `Line2D.Double`. Riječ `Double` pokazuje da se podaci o liniji pohranjeni ka 64-bitni broj s pokretnim zarezom. (Ako želite uštediti memoriju na raspolaganju je i `Line2D.Float`).

Da bismo kreirali liniju potrebno je specificirati koordinate njenih krajnjih točaka. Pozicije su dane ka x i y koordinate i njihova orijentacija je slijedeća:



Koordinate se mjere u pikselima (pixel = picture element). Npr. područje koje ima 300×300 piksela protezat će se otprilike na trećini ekrana.

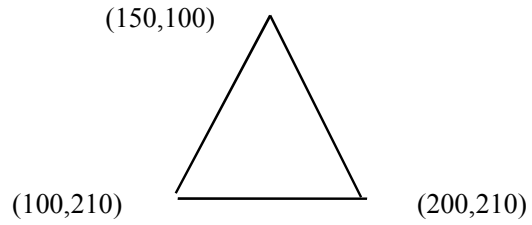
da bismo kreirali objekt linije koristit ćemo `Line2D.Double` konstruktor koji ima koordinate krajnjih točaka linije kao parametre:

```
Line2D.Double(x0,y0,x1,y1)
```

Kreira objekt koji predstavlja liniju od (x0,y0) do (x1,y1).
(Vrijednosti x0, y0, x1 i y1 su `double`. Ako napišete cjelobrojne koordinate Java će ih konvertirati u `double`.)

Primijetite da kreiranje Line2D.Double objekta **nije** isto što i njegovo prikazivanje na ekranu.
Taj objekt bit će prikazan na ekranu tek onda kad ga proslijedimo `draw` metodi Graphics2D objekta koja će ga iscrtati u području prikaza.

Npr. želimo nacrtati slijedeći trokut na ekranu:



Bit će potrebno koristiti slijedeće izraze. Prvo kreiramo objekt koji predstavlja lijevu stranu trokuta tj. objekt tipa Line.Double te ga dodijelimo varijabli `line1`. Zatim ga iscrtamo korištenjem `g2.draw` metode. proces ponovimo za ostale linije.

```
Line2D.Double line1 =
   new Line2D.Double(150,100,200,210);
g2.draw(line1);

Line2D.Double line2 =
   new Line2D.Double(200,210,100,210);
g2.draw(line2);

Line2D.Double line3 =
   new Line2D.Double(100,210,150,100);
g2.draw(line3);
```

Pretpostavimo da pišemo applet koji će samo crtati taj trokut. Napisat ćemo `paint` metodu koja koristi prije navedene naredbe. Postavlja se pitanje kako ćemo kreirati Graphics2D objekt tj. označen s `g2 ?` Odgovor je da ga nećemo mi kreirati. Windows manager će kreirati taj objekt i proslijediti ga `paint` metodi kao parametar.

Postoji jedan mali tehnički problem na koji u svemu moramo paziti. Appleti koriste paint metode još iz najranijih dana Jave, davno prije nego što je Java 2 uvela Graphics2D objekte.
Prije je parametar `paint` metoda imala parametar koji je bio objekt tipa *Graphics* (bez 2D !). Taj objekt je bio sličan Graphics2D objektu, ali daleko manje funkcionalnosti.
Problem je u tome što Java 2 treba pokretati i stare programe pisane za ranije verzije Jave. Zato je ostavljeno da Java kao parametar `paint` metode appleta očekuje Graphics objekt što ćemo morati napisati u zaglavlju paint metode:

```
public void paint(Graphics g)
```

Prevodioc se neće buniti što windows će manager proslijediti Graphics2D objekt kao parametar kad bude pozivao paint metodu ! To je zato što je Graphics2D objekt specijalni slučaj Graphics objekta.

Međutim prevodioc će se pobuniti ako počnemo objekt označen s g tretirati kao Graphics2D objekt. Morat ćemo napisati slijedeći izraz:

```
Graphics2D g2 = (Graphics2D) g;
```

Ovaj izraz kaže: kreiraj Graphics2D varijablu, `g2`, i dodijeli joj objekt g *za kojeg obećavamo da je Graphics2D objekt*. Kastiranje `(Graphics2D)` je naše obećanje prevodiocu da je parametar stvarno Graphics2D objekt. U ostatku metoda g2 tretiramo kao Graphics2D objekt.

Slijedi kompletan program koji će iscrtati trokut. Potrebna je samo jedna klasa. Zaglavlje klase sadrži riječi `extends applet`. Tim pokazujemo prevodiocu da ta klasa definira applet. Prve tri linije pokazuju Java prevodiocu koje tri linije Java biblioteke će biti potrebne.

PRIMJER 1

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

/* Applet koji prikazuje trokut . */

public class Triangle extends Applet

{  public void paint(Graphics g)
   {  Graphics2D g2 = (Graphics2D) g;

      Line2D.Double line1 =
         new Line2D.Double(150,100,200,210);
      g2.draw(line1);

      Line2D.Double line2 =
         new Line2D.Double(200,210,100,210);
      g2.draw(line2);

      Line2D.Double line3 =
         new Line2D.Double(100,210,150,100);
      g2.draw(line3);
   }
}
```

Primijetite da bi se neke naredbe mogle sažetije pisati ako bismo izbjegli uvođenje pomoćnih varijabli `line1, line2 , line3.`

Umjesto pisanja:

```
Line2D.Double line1 =
   new Line2D.Double(150,100,200,210);
g2.draw(line1);
```

Mogli smo pisati

```
g2.draw(new Line2D.Double(150,100,200,210));
```

Obje verzije su ispravne.

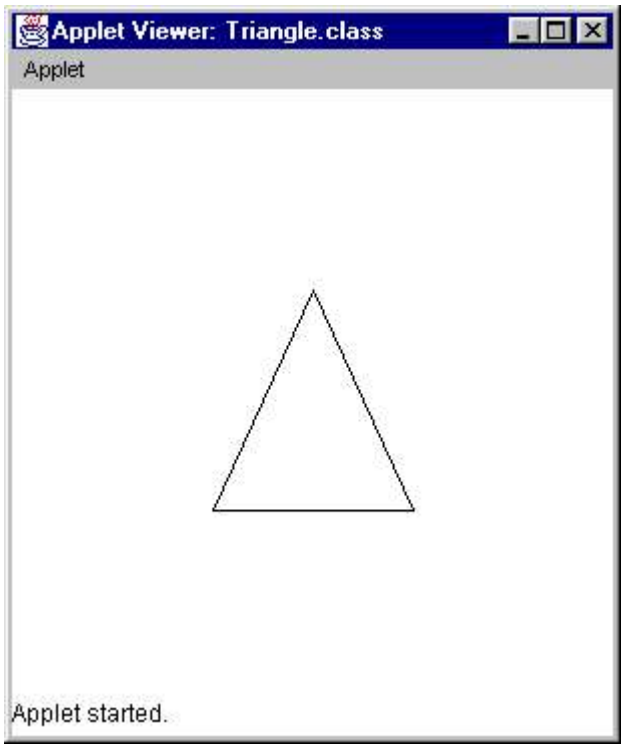
Da bismo pokrenuli ovaj program može biti korištena slijedeća tekstualna datoteka. Ona sadrži ključnu vezu s prevedenom Triangle klasom.

```
<APPLET CODE="Triangle.class"
        WIDTH=300
        HEIGHT=300>
</APPLET>
```

(Smještaj linija nije bitan. Možete otkucati sve u jednoj liniji ako tako želite). Pretpostavimo da je ta datoteka nazvana `Triangle.txt`. Nakon što smo preveli Triangle.java utipkajmo:

```
appletviewer Triangle.txt
```

i slijedeći prozor će se pojaviti na ekranu.



Upamtite. Trokut se iscrtao zato što windows manager poziva `paint` metodu. Pretpostavim da pokušate pokvariti sadržaj prozora tako da npr. pokrenete neki drugi program koji će prekriti applet prozor. Čim učinite nešto što će vratiti vidljivost appleta windows manager će odmah ponovo pozvati `paint` metodu i ponovo iscrtati sadržaj prozora.

Java biblioteka je podijeljena u veći broj dijelova koji se nazivaju **paketi** ili **packages.**
Prva linija primjera:

```
import java.applet.*;
```

kaže prevodiocu da će se koristiti paket koji se naziva `java.applet` . Oznaka `*` pokazuje prevodiocu da očekuje upotrebu *bilo koje* klase iz navedenog paketa. U primjeru se koristi samo jedna klasa iz `java.applet` paketa. Da nismo *importirali* `java.applet` klasu bili bismo prisiljeni koristiti puno ime `java.applet.Applet` umjesto samo `Applet`. To je jedini razlog korištenja `import` naredbe.

U `java.awt` paketu sadržane su mnoge grafičke klase. U prethodnom primjeru koristili smo `Graphics` i `Graphics2D` klase. Npr. paket `java.awt.geom` omogućava upotrebu geometrijskih oblika poput `Line2D.Double` objekta.

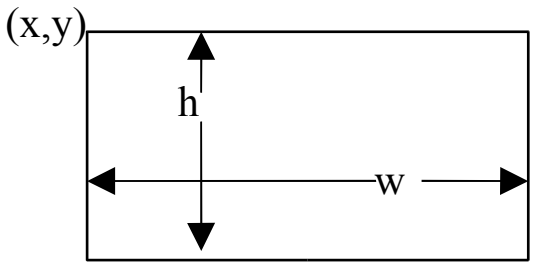
Nije potrebno importirati sve pakete Java biblioteke. Paket `java.lang` uključen je automatski. On uključuje veliki broj često korištenih klasa, poput klasa. `String`, `Math` i `System`.)

3. Drugi oblici

nova grafička biblioteka u javi 2 uključuje različite korisne grafičke oblike. Jedan od njih je i pravokutnik, sadržan u klasi `Rectangle2D.Double`. Pravokutnik možemo konstruirati pomoću slijedećeg konstruktora :

```
Rectangle2D.Double(x,y,w,h)
```

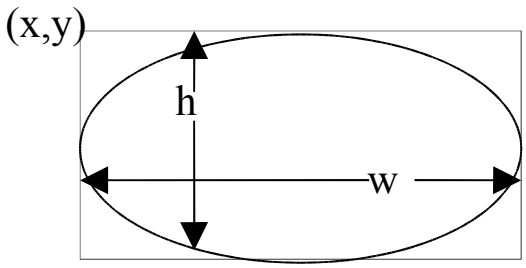
Gornji lijevi kut pravokutnika ima koordinate (x,y). Njegova širina je w, a visina je h.



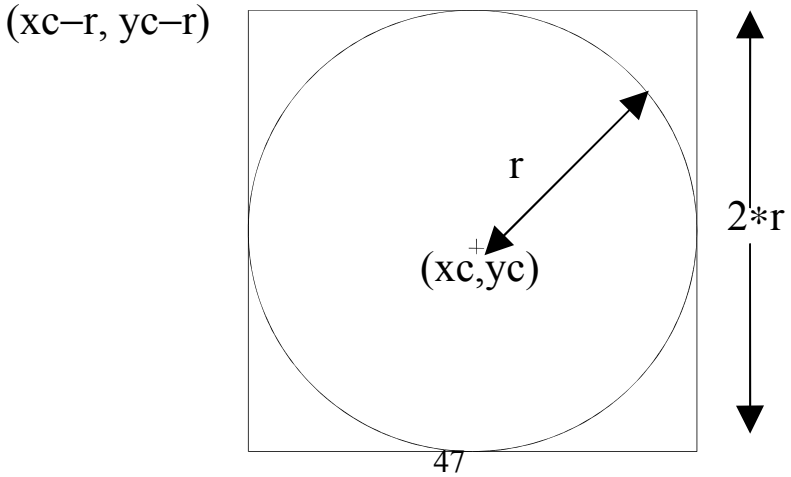
Slično vrijedi i za elipsu koja je definirana klasom Ellipse2D.Double. Može biti kreirana slijedećim konstruktorom

```
Ellipse2D.Double(x,y,w,h)
```

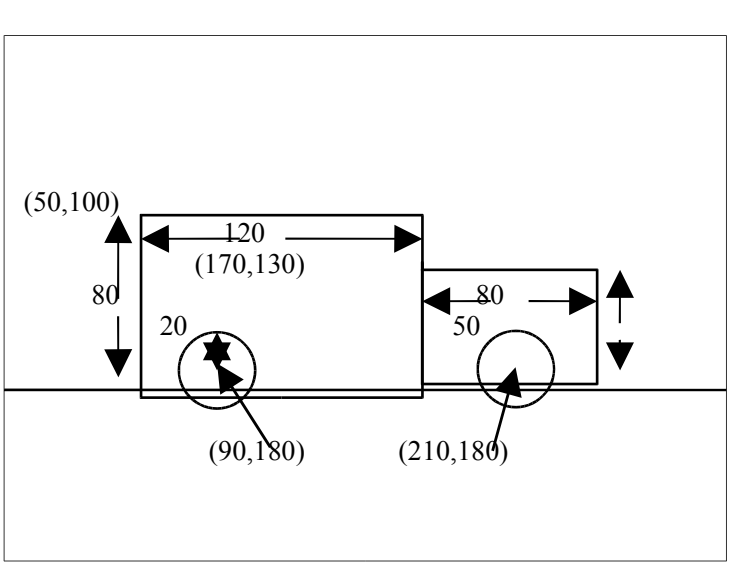
Pozicija i oblik elipse definiran je parametrima x, y, w i h. Ovi parametri određuju pravokutnik koji sadržava elipsu i zadaju se isto kao i za pravokutnik.



Ne postoji objekt koji bi bio rezerviran za definiciju kruga. međutim krug možemo nacrtati kao poseban slučaj elipse čija je širina jednaka visini. Ako je centar kruga u (xc,yc), a njegov radijus je r, tada je to ekvivalentno elipsi koju definiramo preko pravokutnika sa gornjim lijevim kutom u (xc−r, yc−r) te širinom 2∗r i visinom 2∗r.



U slijedećem primjeri koristit ćemo linije, pravokutnike i elipse te nacrtati jedan mali kamion.
Prije nego počnemo pisati kod potrebno je pažljivo definirati položaj pojedinih likova. Slijedi skica kamiona u području prikaza od 300×300.

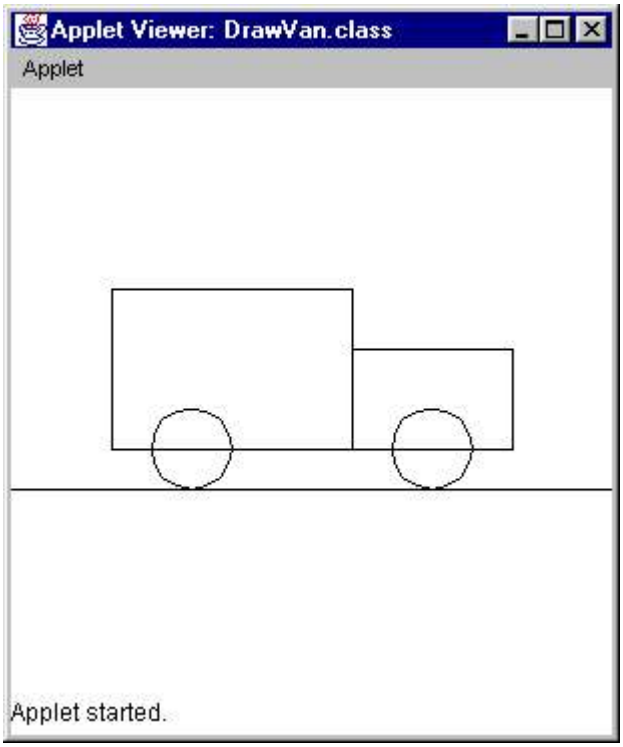


Slijedi applet koji će nacrtati ovu sliku. Za njega je kamion predstavljen kao dva pravokutnika plus dva kruga. Tu je još i linija ceste.

PRIMJER 2

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

public class DrawVan extends Applet
{  public void paint(Graphics g)
   {  Graphics2D g2 = (Graphics2D) g;

      Rectangle2D.Double back =
         new Rectangle2D.Double(50,100,120,80);
      g2.draw(back);

      Rectangle2D.Double back =
         Rectangle2D.Double(170,130,80,50);
      g2.draw(back);

      Ellipse2D.Double backWheel =
         new Ellipse2D.Double(70,160,40,40);
      g2.draw(backWheel);

      Ellipse2D.Double frontWheel =
         new Ellipse2D.Double(190,160,40,40));
      g2.draw(frontWheel);

      Line2D.Double baseLine =
         new Line2D.Double(0,200,300,200);
      g2.draw(baseLine);
   }
}
```

Slijedi prikaz kako bi se trebao vidjeti na ekranu:



4. Boja

Ako drugačije ne navedemo Graphics2D objekt će crtati crnom bojom. Međutim moguće je da promijenite boju kad god to zaželite. Klasa Graphics2D posjeduje `setColor` metodu koja će promijeniti boju u onu koju joj proslijedite kao parametar.

Boje su u Javi predstavljeni kao objekti tipa **Color**. Klasa Color sadrži trinaest konstanti koje predstavljaju standardne boje. (konstanta je varijabla s fiksnom vrijednošću). Npr. me]u njima je konstanta `Colour.red` koja predstavlja crvenu boju. Kompletna lista boja navedena je kasnije.

Stoga, ako želite da objekt g2 (klase Graphics2D) počne pisati crvenom bojom, onda upotrijebite slijedeći izraz prije korištenja `draw` metode:

```
g2.setColor(Color.red);
```

Od momenta kad je izvršena pa sve dok je ponovo ne izvršimo s drugom bojom kao argumentom, sve što `g2` bude prikazivao bit će obojeno crveno.

Objekt Color može predstavljati mnogo više boja nego standardnih 13. Možete definirati svoje boje specificiranjem količine tri osnovne boje : crvene, zelene i plave (red-green-blue = RGB).

Koristi se *aditivni* sustav boja. njegovo osnovno svojstvo je da ako kombiniramo maksimalnu količinu crvene, zelene i plave boje dobivamo bijelu boju. Kombiniranjem nultih vrijednosti crvene, zelene i plave boje dobivamo crnu boju (Ovo je suprotno korištenju boja u tiskanju ili slikarstvu. tu se koriste *substraktivni* sustavi. Npr. miješanjem različitih uljanih boja dobivate sve tamnije boje slične crnoj.

Klasa Color posjeduje konstruktor za kreiranje RGB boje

```
Color(r, g, b)
```

Kreiraj boju kombinirajući r količinu crvene, g količinu zelene i b količinu plave. Ove vrijednosti su brojevi u pokretnom zarezu s vrijednostima od 0 do 1 tipa `float`.

Moramo pažljivo pisati te parametre. Ako npr. napišete 0.5 prevodioc će to shvatiti kao `double` vrijednost i javiti pogrešku. Potrebno je specificirati da se radi o tipu `float`, tj. napisati `0.5f`. Slijedi primjer definiranja žute boje:

```
Color c = new Color(1.0f, 1.0f, 0.0f);
```

(Ne pišite Color(1, 1, 0). Normalno možete koristiti cijele brojeve na mjestu gdje se očekuje float vrijednost, ali ne i ovdje ! Razlog je što Color klasa ima još jedan konstruktor s drukčijim parametrima koji su definirani kao cjelobrojni iznosi crvene, plave i zelene kao *cijeli brojevi u rasponu od 0 do 255 !* Ako dakle napišete Color(1, 1, 0), Java će pretpostaviti da koristite sasvim drugi konstruktor gdje ste specificirali neznatan iznos crvene i zelene i ništa plave te će rezultat biti skoro crna boja.
Slijedi lista 13 standardnih boja i količine crvene, zelene i plave od kojih se sastoje:

```
Color.black      0.0f        0.0f        0.0f
Color.blue       0.0f        0.0f        1.0f
Color.cyan       0.0f        1.0f        1.0f
Color.gray       0.5f        0.5f        0.5f
Color.darkGray   0.25f       0.25f       0.25f
Color.lightGray  0.75f       0.75f       0.75f
Color.green      0.0f        1.0f        0.0f
Color.magenta    1.0f        0.0f        1.0f
Color.orange 1.0f        0.8f        0.0f
Color.pink       1.0f        0.7f        0.7f
Color.yellow 1.0f        1.0f        0.0f
Color.white      1.0f        1.0f        1.0f
```

5. Bojanje likova

Objekt Graphics2D može se koristiti za bojanje unutrašnjosti likova poput pravokutnika ili elipse. Bit će upotrijebljena boja koja je zadnja postavljena s metodom `setColor`. Ako nijedna boja nije postavljena bit će upotrijebljena crna boja.
Metoda koju ćemo koristiti naziva se `fill`. Pozivamo je na slijedeći način:

```
g2.fill(s)
```

Koristi Graphics2D objekt referenciran s `g2` za prikaz oblika `s` ispunjenog s trenutnom bojom pridruženoj `g2`.

Npr. slijedeće naredbe kreirat će mali krug i prikazat ga ispunjenog žutom bojom.

```
Ellipse2D.Double sun =
   new Ellipse2D.Double(140,50,20,20);
g2.setColor(Color.yellow);
g2.fill(sun);
```

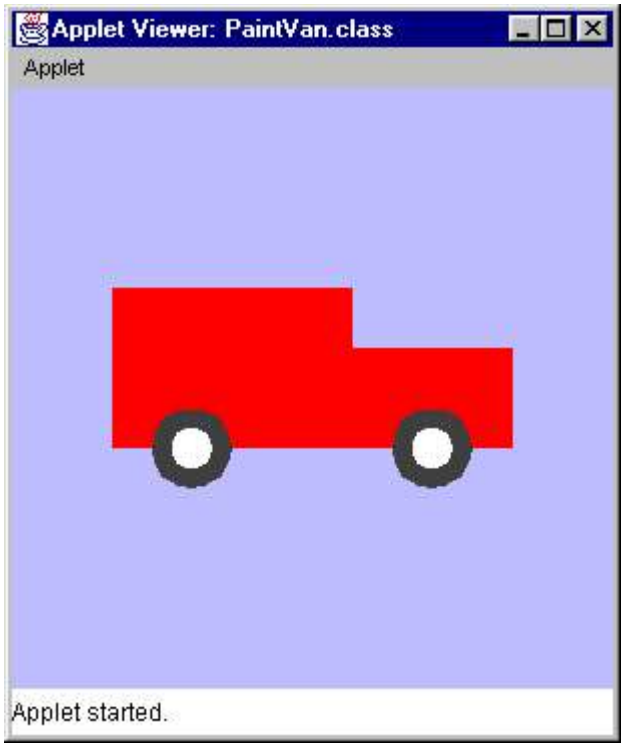
Slijedeći primjer koristi istu tehniku za crtanje kamiona sličnog kao u primjeru 1, ali obojenog u crvenu boju. Točkovi su bijeli, a gume su tamno sive boje. Linija ceste zamijenjena blijedo-plavom bojom. Primijetite da je potrebno iscrtati pozadinu prije kamiona.

PRIMJER 3

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


public class PaintVan extends Applet

{  public void paint(Graphics g)
```

```
    {  Graphics2D g2 = (Graphics2D) g;

       /* Paint the background. */
       Color paleBlue =
          new Color(0.75f, 0.750f, 1.0f);
       g2.setColor(paleBlue);
       g2.fill(new Rectangle2D.Double(0,0,300,300));

       /* Paint the body of the van. */
       g2.setColor(Color.red);
       g2.fill
          (new Rectangle2D.Double(50,100,120,80));
       g2.fill
          (new Rectangle2D.Double(170,130,80,50));

       /* Paint the back wheel. */
       g2.setColor(Color.darkGray);
       g2.fill(new Ellipse2D.Double(70,160,40,40));
       g2.setColor(Color.white);
       g2.fill(new Ellipse2D.Double(80,170,20,20));

       /* Paint the front wheel. */
       g2.setColor(Color.darkGray);
       g2.fill(new Ellipse2D.Double(190,160,40,40));
       g2.setColor(Color.white);
       g2.fill(new Ellipse2D.Double(200,170,20,20));
    }
}
```

Slijedi obojani kamion:



6. Pisanje teksta

Osim za crtanje i bojanje oblika, Graphics2D objekt može se koristiti za ispis teksta na ekranu. Prije nego što to pokušamo učiniti, potrebno je Graphics2D objektu naznačiti koji će *font* koristiti. Font je reprezentiran s objektom tipa `Font` . Koristimo slijedeći konstruktor za kreiranje Font objekta.

`Font(familija, stil, veličina)` (family, style, size)

Parametar `familija` je string koji predstavlja naziv skupa srodnih fontova. Primjeri srodnih familija su Times Roman koji se koristi kao glavni font ovog teksta, i Courier koji se koristi za tekst programa. Ova rečenica je tiskana u Arial fontu. Slijedeća logički nazivi mogu se koristiti za familije fontova:

```
Serif
SansSerif
Monospaced
Dialog
DialogueInput
```

Parametar `stil` ima jednu od slijedećih vrijednosti koje naznačuju da li se koristi *italic (zakošeno) ili* **bold (podebljano)** pisanje:

```
Font.PLAIN
Font.ITALIC
Font.BOLD
Font.ITALIC+Font.BOLD.
```

Parametar `veličina` je *veličina fonta u točkama*. To je visina karaktera mjerena u jedinci od 1/72 inča. Veličina točke od 1 približno odgovara jednom pikselu. Ova predavanja napisana su fontom veličine 12 i karakteri bi trebali biti visoki oko 1/6 inča.

Jednom kada kreirate Font objekt, potrebno je Graphics2D objektu narediti da ga koristi. To činimo upotrebom slijedeće metode:

`g2.setFont(f)`
Postavi Graphics2D objekt referenciran s `g2` tako da koristi `f` za pisanje teksta.

Sve što preostaje je da se ispiše string na određeno mjesto na ekranu:

`g2.drawString(text, x, y)`

Koristi Graphics2D objekt `g2` za ispis stringa `text`, počevši od pozicije (x,y) u području prikaza. x i y su of tipa `int`. (x je pozicija lijeve strane teksta, a y je visina na kojoj se nalazi bazna linija teksta.

Slijedeći primjer ilustrira upotrebu logičkih familija fontova . U svakom koraku Graphics2D objektu pridružujemo jedan logički font te pomoću tog fonta ispisujemo naziv familije. U svakom od slučajeva koristimo stil FONT.PLAIN te je veličina fonta jednaka 30. 0.

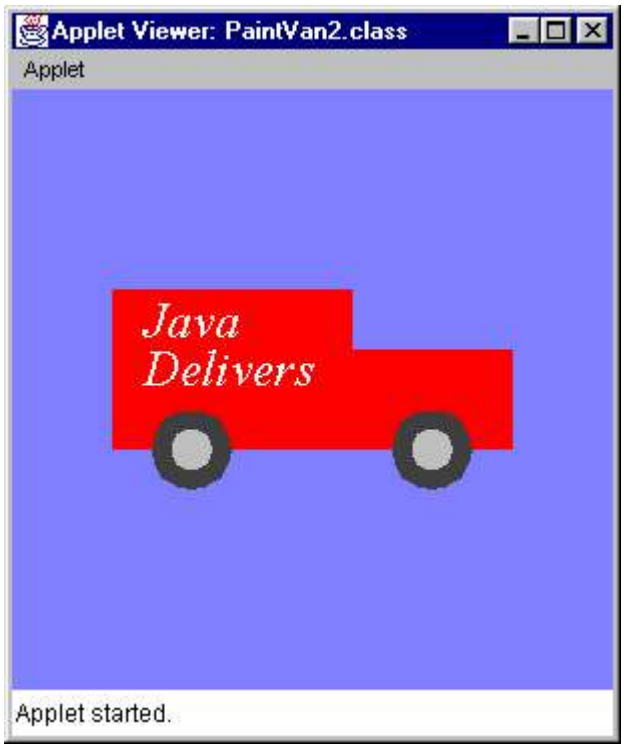
PRIMJER 4

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

public class Fonts extends Applet
{  final int SIZE = 30;

   public void paint(Graphics g)
   {  Graphics2D g2 = (Graphics2D) g;
```

```
        Font font1 =
           new Font("Serif", Font.PLAIN, SIZE);
        g2.setFont(font1);
        g2.drawString("Serif", 50, 50);

        Font font2 =
           new Font("SansSerif", Font.PLAIN, SIZE);
        g2.setFont(font2);
        g2.drawString("SansSerif", 50, 100);

        Font font3 =
           new Font("Monospaced", Font.PLAIN, SIZE);
        g2.setFont(font3);
        g2.drawString("Monospaced", 50, 150);

        Font font4 =
           new Font("Dialog", Font.PLAIN, SIZE);
        g2.setFont(font4);
        g2.drawString("Dialog", 50, 200);

        Font font5 =
           new Font("DialogInput", Font.PLAIN, SIZE);
        g2.setFont(font5);
        g2.drawString("DialogInput", 50, 250);
    }
}
```

Za slijedeći primjer korištenja ispisa teksta u primjeru 3 (koji crta obojani kamion) dodajte slijedećih pet naredbi na kraj `paint` metode.

```
/* nacrtaj logo na stranici kamiona */
Font f = new Font("Serif", Font.ITALIC, 25);
g2.setFont(f);
g2.setColor(Color.white);
g2.drawString("Java",66,124);
g2.drawString("Delivers",66,148);
```



7. Smajli

Grafički objekti koje smo koristili u ovom poglavlju, poput Line2D.Double objekta na neki način posjeduju pridruženi prikaz. Ovu ideju možemo dalje širiti. Ako bismo trebali napraviti grafički program za očekivat bi bilo da imamo niz Java objekata za koje bi trebalo realizirati prikaz na ekranu. U slijedećem primjeru ćemo napraviti jedan takav objekt.

Definirat ćemo tip objekta koji predstavlja osobu. Nazvat ćemo je Person (osoba) klasa. Postoje mnogi atributi koje bismo mogli definirati za objekt tipa Person. U ovom slučaju upotrijebit ćemo samo jedan atribut koji će definirati raspoloženje osobe.
To je vrijednost između 0 i 1 i nazvat ćemo je `raspoloženje`. Osoba s vrijednošću `raspoloženje` ode 1 je veoma raspoložena osoba. 0 znači da je u pitanju potpuno neraspoložena osoba.

Jedna od operacija koje ćemo asocirati s objektom Person je crtanje lika osobe. To će raditi slijedeća metoda:

```
p.crtajLice(double x, double y, Graphics2D g2)
```
Nacrtaj sliku lica osobe reprezentirane s Person objektom referenciranog s varijablom p. Centar lica su koordinate `x i y`. Slika će se iscrtati pomoću Graphics2D objekta `g2`.

Lice će biti jednostavno sa osmjehom proporcionalnim raspoloženju.

Imat ćemo dva metoda koja će uticati na raspoloženje:

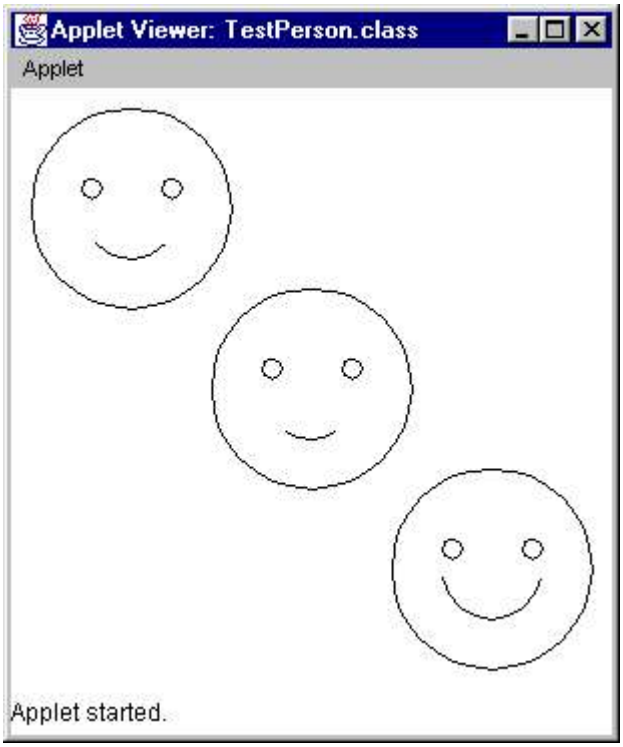
```
p.oraspoloži() // Povećaj raspoloženje osobe za konstantan faktor.

p.oneraspoloži() //    Smanji raspoloženje osobe za konstantan faktor.
```

Konstruktor za kreiranje Person objekta:
`Person(double h) //` Kreiraj Person objekt s 'raspoloženje' postavljenim na `h`.

Prije same implementacije Person klase, slijedi `paint` metoda koji kreira Person objekt i mijenja mu raspoloženje nekoliko puta. Ova metoda crta lice osobe na tri različite lokacije u području prikaza appleta.

PRIMJER 5 `public void paint(Graphics g)`

```
{  Graphics2D g2 = (Graphics2D) g;

   Person kim = new Person(0.5);
   kim.crtajLice(60,60,g2);
   kim.oneraspoloži();
   kim.crtajLice(150,150,g2);
   kim.oraspoloži();
   kim.oraspoloži();
   kim.oraspoloži();
   kim.oraspoloži();
   kim.crtajLice(240,240,g2);
}
```



Sada slijedi implementacija. Kako pretpostavljate , klasa Person sadržava jedno polje nazvano `raspoloženje`. Polje je tipa `double`, i sadrži vrijednost u opsegu od 0 do 1. Konstruktor postavlja to polje na vrijednost parametra.

Metoda `oneraspoloži` množi vrijednost raspoloženja s konstantnim faktorom. Taj faktor se drži u varijabli nazvanoj `factor`. Tijelo metode sadrži samo jednu naredbu:

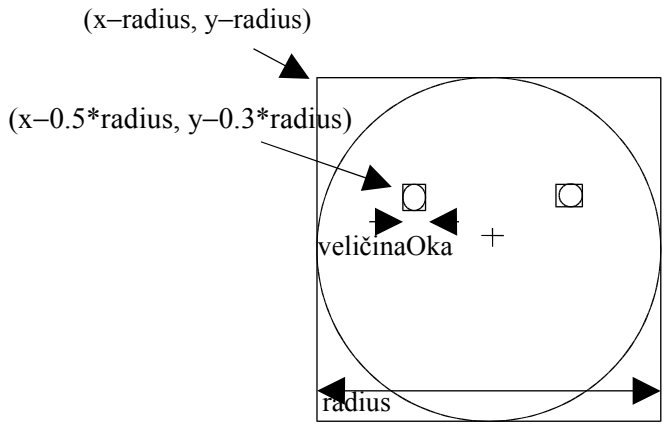
```
raspoloženje = factor*raspoloženje;
```

`factor` je postavljen unaprijed na 0.65. Bilo je jednostavnije pisati `raspoloženje = 0.65*raspoloženje`. Umjesto toga smo ipak smo definirali konstantu i dali joj naziv 'factor'. Poslije ćemo lakše promijeniti njenu vrijednost na jednom mjestu nego na svim mjestima gdje je budemo koristili.

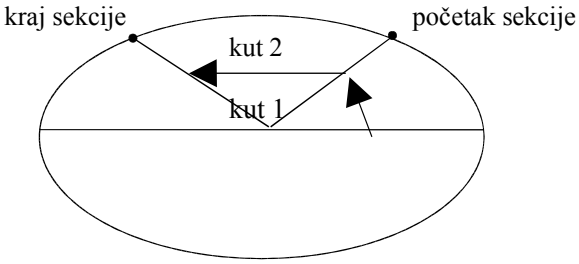
Metoda `oraspoloži` izvodi slijedeći izraz:

```
raspoloženje = 1 - factor*(1 - raspoloženje);
```

Jedini 'teži' dio implementacije klase Person je implementacija metoda `crtajLice`.
Svako lice sastojat će se od konture, lijevog oka desnog oka i osmjeha. Prva tri su Ellipse2D.Double objekti. Parametri korišteni u konstrukciji lica su izračunati koristeći vrijednosti `x` i `y` koji predstavljaju centar lica., `radius` koji je radijus lica, `veličinaOka` koji je dijametar svakog oka.



Konstrukcija osmjeha je kompliciranija jer moramo koristiti novi objekt iz java biblioteke tj. Arc2D.Double objekt. Ovaj objekt predstavlja dio elipse – sekciju. Potrebno je specificirati kut koji predstavlja početak sekcije te kut koji predstavlja kut sekcije.



Postoji više konstruktora koji su mogući za Arc2D Double objekt. Slijedi jedan od njih:

```
Arc2D.Double(double x, double y,
             double w, double h,
             double start, double extent,
             int type)
```

Kreiraj sekciju elipse s gornjim lijevim kutom na (`x`,`y`), širine `w` i visine `h`. Sekcija je definiran početnim kutom `start` i drugim kutom `extent`. Vrsta sekcije definirana je parametrom `type`.

Za kreirati samo dio sekcije koji se sastoji od krivulje (bez spojenih krajeva) parametar `type` treba postaviti na konstantu `Arc2D.Double.OPEN`.

Slijedi naredba za konstrukciju osmjeha:

```
Shape osmjeh =
   new Arc2D.Double
      (x-0.5*radius, y-0.5*radius, radius, radius,
       180+90*(1-raspoloženje), 180*raspoloženje,
       Arc2D.Double.OPEN);
```

Ako pažljivije pogledate vidjet ćete da je osmjeh dio kruga s centrom u centru lica s duplo manjim radijusom od radijusa lica. Ako je raspoloženje jednako 1, sekcija je polukrug. Ako je raspoloženje 0 onda sekcija u potpunosti nestaje.

Varijabla `osmjeh` je deklarirana kao klasa tipa Shape, ne kao Arc2D.Double. U stvari svaki geometrijski oblik koji se može nacrtati pomoću objekta Graphics može biti tipa Shape. 'Shape' je primjer Java sučelja (**interface**). Interface Shape definira koje metode neki objekt treba posjedovati da bi se mogao referirati varijablom tog tipa.

Slijedi kompletna definicija klase Person

PRIMJER 5b

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

/* Person objekt predstavlja osobu
   s različitim nivoom raspoloženja raspoloženje.
*/

public class Person
{
   private double raspoloženje;

   private double factor = 0.65;
      // Faktor korišten za povećanje ili smanjenje
      // raspoloženja osobe

   /* Oraspoloži osobu
   */
   public void oraspoloži()
   {  raspoloženje = 1 - factor*(1 - raspoloženje);
   }

   /* Oneraspoloži osobu
   */

   public void oneraspoloži()
   {  raspoloženje = factor*raspoloženje;
   }

   /* Nacrtaj lice osobe korištenjem Graphics2D objekta g2.
      Centar lica je (x,y), radius 50.
   */
   public void
      crtajLice(double x, double y, Graphics2D g2)
   {  double radius = 50;
      double veličinaOka = 10;

      Shape outline =
         new Ellipse2D.Double
            (x-radius, y-radius, 2*radius, 2*radius);
      Shape leftEye =
         new Ellipse2D.Double
            (x-0.5*radius, y-0.3*radius,
             veličinaOka, veličinaOka);
      Shape rightEye =
         new Ellipse2D.Double
            (x+0.5*radius-veličinaOka, y-0.3*radius,
             veličinaOka, veličinaOka);
      Shape osmjeh =
         new Arc2D.Double
            (x-0.5*radius, y-0.5*radius,
             radius, radius,
             180+90*(1-raspoloženje), 180*raspoloženje,
```

```
                Arc2D.Double.OPEN);

        g2.draw(outline);
        g2.draw(leftEye);
        g2.draw(rightEye);
        g2.draw(osmjeh);
    }

    /* Kreiraj osobu tj. objekt tipa Person s raspoloženjem
     postavljenim na h       */
    public Person(double h)
    {  raspoloženje = h;
    }
}
```

8. Čitanje ulaza

U appletima moramo unošenju podataka s tipkovnice pristupiti na različit način nego što smo to radili u dosadašnjim aplikacijama.

Koristit ćemo se sekundarnim prozorima koje ćemo nazvati *dialog*.
Kreirat ćemo ih korištenjem **JOptionPane** objekta. Ova klasa realizira dialog koji sadrži poruku od korisnika te prostor za upis podataka.

Slijedeća rečenica će uzrokovati pojavu dijaloga s porukom 'Please enter your name.' Dijalog će čekati da korisnik upiše niz znakova i pritisne OK (ili pritisne Enter). Na kraju će uneseni string biti pridružen varijabli reply.

```
String reply =
   JOptionPane.showInputDialog
      ("Please enter your name.");
```

To će izgledati ovako:

Istu tehniku možete koristiti za unos brojeva. Prvo učitajte korisnikov upis kao string, st, i onda ga konvertirajte korištenjem slijedećih metoda:

`Integer.parseInt(st)` ili `Double.parseDouble(st)`.

U primjeru 6 program opet crta smajlija. Ovaj put program preko dijaloga pita korisnika da unese broj koji odgovara raspoloženju osobe. Na osnovu unesenog broja konstruira se Person objekt s unesenim raspoloženjem.

Gdje ubaciti izraz za prikaz dijaloga za unos početnog raspoloženja. Nije dobro mjesto Paint metoda jer se ona izvršava svaki put kada windows manager osvježava prikaz na ekranu.
Potrebno je da se dialog pojavi samo jednom prilikom pokretanja appleta. Pravo mjesto je metoda

```
public void init()
```

Osim `paint` metode windows manager poziva i metodu `init` ,ali samo jednom na početku izvršavanja appleta. Stoga se u tu metodu mogu smjestiti naredbe koje su vezane za inicijalizaciju parametara appleta.

Primijetite da je objekt tipa Person dodijeljen polju `fred` kojemu se kasnije pristupa u metodi `paint`. Nismo mogli deklarirati varijablu tipa Person unutar metode Init !

PRIMJER 6

```
import java.applet.*;
import java.awt.*;
import javax.swing.*;
   // (Potrebno za JOptionPane klasu.)

/* Čitaj 'raspoloženje' vrijednost i
   zatim nacrtaj lice.
*/

public class DrawSmiley extends Applet

{   Person fred;

    public void init()
    {  String input =
          JOptionPane.showInputDialog
             ("Koliko je osoba sretna?");
       double h = Double.parseDouble(input);
       fred = new Person(h);
    }

    public void paint(Graphics g)
    {  Graphics2D g2 = (Graphics2D) g;
       fred.crtajLice(150, 150, g2);
    }
}
```

Ovaj applet je napravljen za dimenzije područja prikaza od 300×300.
Možete applet učiniti pametnijim tako da prvo provjeri dimenzije trenutne dimenzije područja prikaza. Metoda `getWidth()` vratit će trenutnu širinu područja prikaza, dok će metoda `getHeight()` vratiti trenutnu visinu.

Dakle možete pisati slijedeće:

```
double x = getWidth()/2.0;
double y = getHeight()/2.0;
fred.crtajLice(x, y, g2);
```

Probajte rastezati granice appleta i vidjet ćete da će ispis uvijek biti centriran.

Moguće je u init metodi postaviti trajnu boju pozadine s:

```
setBackground(Color.blue);
```

Ako želite vidjeti što applet čini u određenom trenutku možete ubaciti pozive metode `System.out.println`. To će ispisati poruku u DOS prozoru na uobičajen način.

Stavite npr.:

```
System.out.println(″paint called″);
```

na početak `paint` metode i onda ćete u DOS prozoru vidjeti koliko često windows manager poziva paint metodu.

10. Zadaci

Unesite i testirajte primjer 5.

Napišite grafički program koji će ispisati vaše ime u centru ekrana unutar plavog pravokutnika . Applet nazovite ImeApplet.java, a Html datoteku ImeApplet.txt

Napišite grafički program koji će zatražiti unos radijusa i zatim nacrtati krug s tim radijusom. Applet nazovite KrugApplet.java, a Html datoteku KrugApplet.txt

5. ISPITIVANJE UVJETA

Peto poglavlje sastoji se od brojnih primjera korištenja `if` uvjetnog izraza kojima biramo između dvije alternativne akcije.

SADRŽAJ

1. Izrazi za donošenje odluka

U dosadašnjim programima svaki put se prilikom programa izvršavao isti niz izraza u redoslijedu kako su bili napisani. Java poput svih programskih jezika omogućava pisanje programa koji samostalno odlučuju koji će se naredbe izvršiti. Ta vrsta odluke se u Javi najčešće donosi pomoću **if uvjetne naredbe**. Slijedi primjer if uvjetne naredbe:

```
if (odgovor == 6048)
   System.out.println("Točno!");
else
   System.out.println("Netočno!");
```

Ova naredba znači:

> Ako je vrijednost varijable `odgovor` jednaka 6048,
> prikaži poruku "Točno!".
> inače prikaži poruku "Netočno."

Primijetite da je izraz

```
odgovor == 6048
```

upotrijebljen za ispitivanje uvjeta '`odgovor` je jednak 6048'. Simbol `==` koristi se za ispitivanje jednakosti pošto se `=` koristi za dodjeljivanja.

Slijedi program koji koristi `if` naredbu . Program pita koliki je rezultat računske operacije 72 puta 84? Zatim čita korisnikov odgovor i sprema ga u varijablu `odgovor`. Nakon toga slijedi `if` naredba koja provjerava da li korisnik pravilno odgovorio na postavljano pitanje. Ovaj program je poput većine primjera u ovom poglavlju aplikacija, a ne applet.

PRIMJER 1

```
public class Multi1

{  /* Traži od korisnika da unese odgovor na pitanje
      koliko je 72 puta 84, i onda provjeri odgovor. */

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

```
        System.out.println("Koliko je 72 puta 84?");
        int odgovor = user.readInt();
        if (odgovor == 6048)
           System.out.println("Točno!");
        else
           System.out.println("Netočno!");
    }
}
```

Opći oblik `if` naredbe dan je uokviren dolje. Naziv **Logički izraz** zamjenjuje bilo koji izraz čiji je rezultat "točno" ili "netočno" (true or false).

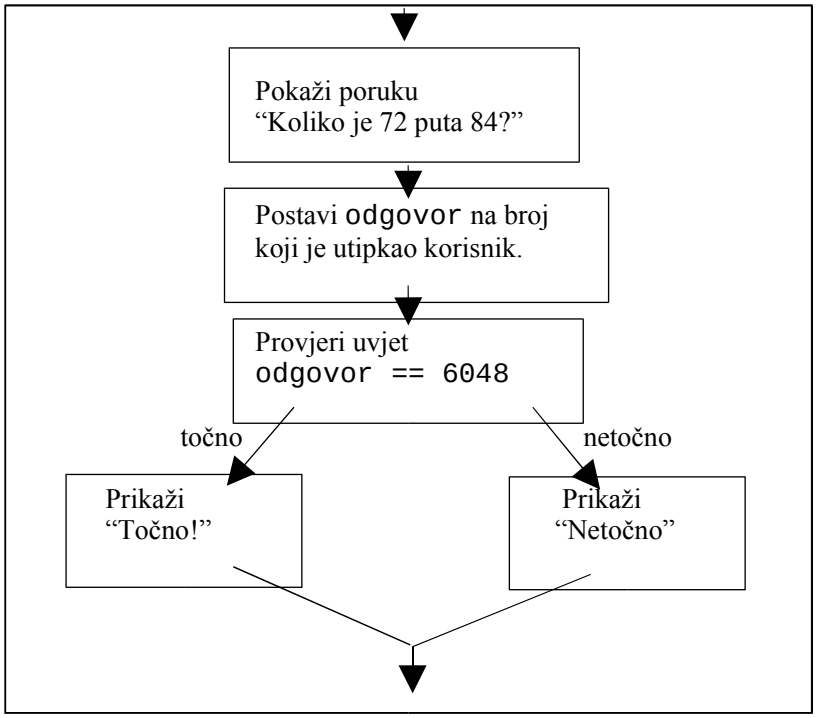
if naredba (s dvije grane)

```
if (LOGIČKI-IZRAZ)
   IZRAZ-1
else
   IZRAZ-2
```

Koda Java izvršava ovu naredbu prvo provjerava da li je rezultat *LOGIČKI-IZRAZ točno ili netočno*. Ako je točno izvršit će IZRAZ-*1*. Ako je netočno izvršit će IZRAZ-*2*.

*IZRAZ-1*se ponekad naziva **prva grana**. *IZRAZ-2*, je **druga grana**. Bilo koji izraz može se upotrijebiti umjesto *IZRAZ-1* i *IZRAZ-2*, čak i druga `if` naredba.

Za vizualizaciju događanja u programu koristit ćemo **dijagram toka.** Dijagram toka prethodnog programa izgleda ovako:



Ponekad je potrebno izvršiti nekoliko naredbi kad je uvjet zadovoljen, a nekoliko kad nije.
U tom slučaju možete zamijeniti *UVJET-1* ili *UVJET-2* (ili oboje) s **blokom naredbi**. **Blok** (naredbi) je niz naredbi koje su zatvorene unutar vitičastih zagrada. Čak možete imati i blok bez i jedne naredbe poput `{}`. Blok se može staviti na bilo koje mjesto u programu na kojem inače dolazi naredba.

Slijedi primjer programa s blokom umjesto grane if naredbe (blok je prikazan podebljano).

PRIMJER 2

```
public class Multi2

{  /* Traži od korisnika da unese odgovor na pitanje
      koliko je 72 puta 84, i onda provjeri odgovor. */

   public static void main(String[] args)

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

      System.out.println("Koliko je 72 puta 84?");
      int ans = user.readInt();
      if (ans == 6048)
         System.out.println("Točno!");
      else
      {  System.out.println
            ("Netočno. Još jednom pokušaj !");
         ans = user.readInt();
         if (ans == 6048)
            System.out.println
               ("Napokon točno");
         else
            System.out.println
               ("Opet netočno. Točan odgovor je 6048");
```

```
        }
    }
}
```

2. `if` naredba s jednom granom

Postoji kraća forma `if` naredbe u kojoj se izostavlja riječ `else` i druga grana. Slijedi primjer:

```
if (a < b) then
   System.out.println(″Manja vrijednost je ″ + a);
```

Ovo znači: ako je a manje od b tada prikaži poruku koja kaže da je manja vrijednost a inače čini ništa. Ovakva naredba ima isti efekt kao slijedeća if naredba.

```
if (a < b) then
   System.out.println(″Manja vrijednost je ″ + a);
else
   {}
```

Opći oblik `if` naredbe s jednom granom je:

if naredba (s jednom granom)

```
if (LOGIČKI-IZRAZ)
   IZRAZ
```

Ako je *`LOGIČKI-IZRAZ`* točan tada će se izvršiti *`IZRAZ`* . Inače je `if` naredba bez ikakvog efekta.

Slijedi još jedan primjer u kojemu tražimo najmanji od četiri broja a,b,c,d i rezultat pohranjujemo u varijablu s.

```
s = a;
if (b < s) s = b;
if (c < s) s = c;
if (d < s) s = d;
```

Za misaonu vježbu pretpostavite slijedeće vrijednosti i odredite tijek izvršavanja:

a = 5, b = 10, c = 3, d = 8.

Slijedi primjer programa s `if` naredbom s jednom granom . Program računa kvadratni korijen unesenog broja. U slučaju da korisnik unese negativan broj program će ispisati poruku o grešci te izvršiti return naredbu.

Ključna riječ `return` znači: završi s izvođenjem metode, u ovom slučaju programa.

PRIMJER 3

```
public class Sqrt

{  /* Pitaj korisnika da unese broj a zatim ispiši
  kvadratni korijen unesenog broja. */

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

      System.out.print("Unesi broj: ");
      double x = in.readDouble();
```

```
        if (x < 0)
        {  System.out.println
              ("Broj treba biti >= 0.");
           return;
        }
        System.out.println
           ("Kvadratni korijen je " + Math.sqrt(x));
     }
  }
```

Primjeri korištenja programa:

```
Unesi broj: 9.2
Kvadratni korijen je 3.03315017762062

Unesi broj: -3
Broj treba biti >=0.
```

Često se s greškama postupa na ovakav način tj. ispitivanjem s `if` naredbom te vraćanjem s naredbom `return.`

Return naredba ima dva osnovna oblika:

`return` naredba
(vraća vrijednost)

`return` *`IZRAZ`*`;`

`return` naredba
(ne vraća vrijednost)

```
return;
```

Prva vrsta return naredbe koristi se unutar metoda koji vraćaju vrijednost. Znači: završi s izvršavanjem metode i vrati vrijednost danu s IZRAZ. Druga se koristi unutar metoda koje ne vraćaju nikakvu vrijednost i znači samo kraj izvršavanja metode

3. Izbor između različitih alternativa

Ponekad je potrebno napisati kod koji bira između 3 ili više alternativa. To se može učiniti kombinacijom nekoliko `if` naredbi. Npr. student ostvari određen broj bodova na ispitu od maksimalnih 100. Na osnovu toga treba izračunati ocjenu.

broj bodova 70 - 100	ocjena = 5
broj bodova 60 - 69	ocjena = 4
broj bodova 50 - 59	ocjena = 3
broj bodova 40 - 49	ocjena = 2
broj bodova 0 - 39	ocjena = 1

Pretpostavimo da imamo varijablu `bodova`, koja sadrži broj bodova, a ocjenu želimo pohraniti u varijablu `ocjena`.

Pseudo kod bi izgledao ovako:

```
if (bodova >= 70) ocjena = ”5”;
else Postupaj s bodovima u području od 0 do 69.
```

Da bismo postupali s bodovima u području od 0 do 69, možemo koristiti drugu if naredbu, poput slijedećeg:

```
if (bodova >= 70) ocjena = ″5″;
else if (bodova >= 60) ocjena = ″4″;
else Postupaj s bodovima u području od 0 do 59.
```

Dalje postupamo na sličan način:

```
if (bodova >= 70) ocjena = ″5″;
else if (bodova >= 60) ocjena = ″4″;
else if (bodova >= 50) ocjena = ″3″;
else Postupaj s bodovima u području od 0 do 49
```

Itd. Na kraju slijedi kompletan izraz:

```
if (bodova >= 70) ocjena = "5";
else if (bodova >= 60) ocjena = "4";
else if (bodova >= 50) ocjena = "3";
else if (bodova >= 40) ocjena = "2";
else ocjena = "1";
```

Primijetite da je ovaj komad koda zapravo `if` naredba !

Slijedi kompletan program za proračun ocjene iz broja bodova:

PRIMJER 4

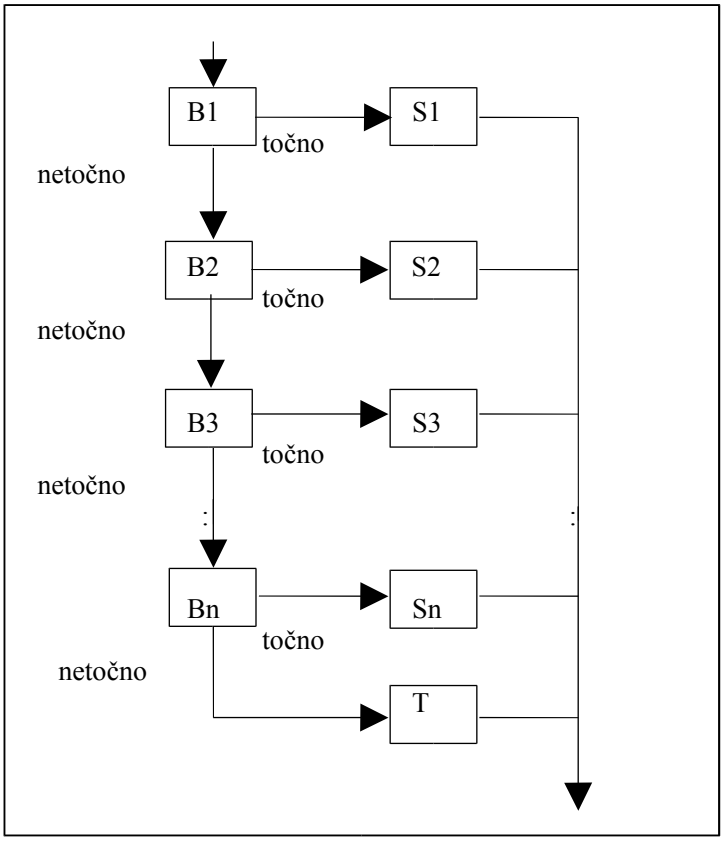
```
public class Ocjena
{  /* Učitaj broj bodova u području od 0 do 100,
      i ispiši odgovarajuću ocjenu.
   */
   public static void main(String[] args)
   {  ConsoleReader in =
         new ConsoleReader(System.in);

      System.out.print("Unesi broj bodova ");
      int bodova = in.readInt()
      String ocjena;
      if (bodova >= 70) ocjena = "5";
      else if (bodova >= 60) ocjena = "4";
      else if (bodova >= 50) ocjena = "3";
      else if (bodova >= 40) ocjena = "2";
      else ocjena = "1";
      System.out.println("Ocjena = " + ocjena);
   }
}
```

Ovom tehnikom ispitujemo niz uvjeta i čim je jedan zadovoljen izvršavamo odgovarajuću akciju.

```
if (B1) S1
else if (B2) S2
else if (B3) S3
  :
  :
else if (Bn) Sn
```

```
    else T
```

Blok dijagram ovakve if naredbe izgleda ovako:



4. Tip podataka `boolean`

Logički (boolean) izraz poput `odgovor == 6048` izračunava se u jednu od slijedećih vrijednosti `true` (točno) and `false`(netočno). Ove dvije vrijednosti nazivaju se **logičke vrijednosti**. Zajedno čine primitivni tip podataka koji u Javi nazivamo `boolean`.

U mnogim pogledima ove vrijednosti se ponašaju poput drugih tipova vrijednosti kao npr. cjelobrojnih vrijednosti. Npr. metode mogu poput vraćanja cjelobrojnih vrijednosti vraćati logičke vrijednosti. Možete kreirati varijablu tipa `boolean` i dodjeljivati im vrijednosti.

Npr.:

```
boolean ok = (x > 0);
```

kao primjer korištenja logički vrijednosti, proširit ćemo Student klasu iz trećeg poglavlja. Dodat ćemo logički polje u koje ćemo bilježiti je li student upisao predmete tekuće godine.
Polje ćemo nazvati `jeUpisan`. Bit će postavljeno na `true` ako je student upisao predmete, a na `false` ako nije. Polje je deklarirano na slijedeći način:

```
private boolean jeUpisan;
```

bit će potrebno učiniti još neke promjene u klasi student. Dodat ćemo slijedeće metode:

`s.upiši()`
Zapiši da je student s upisan.

`s.ispiši()`
Zapiši da Student s nije upisan.

`s.jeliUpisan()`
vrati `true` ako je s upisan, inače vrati `false`.

metode su vrlo jednostavne. Prva postavlja polje `jeUpisan` na `true`, druga postavlja polje `jeUpisan` na `false` i treća vraća vrijednost polja `jeUpisan`.

napravljene su još dvije promjene u klasi Student. U metodi `prikaz` dodano je ispitivanje vrijednosti polja `jeUpisan` na osnovu koje se ispisuje 'Upisan je' ako je njegova vrijednost postavljena na `true`, ili 'Nije upisan' ako je `false`. Konstruktor postavlja vrijednost polja `jeUpisan` na `false`.

Slijedi primjer promijenjene Student klase. Metodi koji nisu mijenjani napisani su u skraćenom obliku.

PRIMJER 5

```
public class Student

{  private String idBroj;
   private String ime;
   private String programStudija;
   private int godina;
   private boolean jeUpisan;

   /* Zabilježi da je student upisan.
   */
   public void upiši()
   {  jeUpisan = true;
   }

   /*  Zabilježi da student nije upisan
   */
   public void ispiši()
   {  jeUpisan = false;
   }

   /* vrati true ako je student upisan
 inače vrati false
   */
   public boolean jeliUpisan()
   {  return jeUpisan;
   }

   /* Prikaži detalje o studentu
   */
   public void prikaz()
   {  System.out.println
         ("Student ID: " + idBroj);
      System.out.println("Ime: " + ime);
      System.out.println
         ("Program studija: " + programStudija);
      System.out.println("Godina: " + godina);
      if (jeUpisan)
         System.out.println(″Nije upisan.″);
      else
         System.out.println(″Upisan je.″);
   }
```

Definicije klasa setProgramStudija, povecajGodinu i getIme su nepromijenjene

```
    /* Kreiraj novog studenta sa zadanim
       ID brojem, imenom i programom studija
       Polje godina bit će postavljeno na 1.
       Student početno nije upisan.
    */

   public Student(String id, String nm, String prog)
   {  idBroj = id;
      ime = nm;
      programStudija = prog;
      godina = 1;
      jeUpisan = false;
   }
}
```

Kada pišemo logičke izraze možemo koristiti slijedeće relacijske operatore:

`==`	jednako
`!=`	nije jednako
`<`	manje od
`<=`	manje ili jednako
`>`	veće
`>=`	veće ili jednako

Ovi operatori se koriste između bilo koja dva izraza koji označavaju cjelobrojnu ili vrijednost u pokretnom zarezu. Kompleksnije logičke izraze formiramo korištenjem logičkih operatora.
(B1 i B2 su logički izrazi)

operator	*značenje*
`B1 && B2`	B1 I B2
`B1 || B2`	B1 ILI B2
`! B1`	NEGACIJA B1

Izraz B1 `&&` B2 daje rezultat `true` ako su i B1 i B2 `true`, inače daje `false`. Kada Java izračunava izraz B1 `&&` B2, prvo računa izraz B1 . Ako je B1 `false` izraz B2 se ne izračunava, jer neovisno o njemu cijeli izraz je `false`.

Izraz B1 `||` B2 bit će `true` ako najmanje jedan od B1 i B2 je `true`. Ako su i B1 i B2 vrijednosti `false`, rezultat je `false`. Java također izračunava B1 `||` B2 na 'lijeni način' . Ako je B1 `true` tada B2 se ne računa ! (Zašto?)

Operatori `&&` i `||` su manjeg prioriteta od drugih operatora koje smo dosada sreli. Java će izraz:

```
n >= 0 && n< 10 || n >= 20 && n < 30
```

interpretirati kao:

```
((n>=0) && (n<20)) || ((n>=20) && (n<30))
```

Operator 'not' tj., `!`, je većeg prioriteta nego ostali operatori. Operator = je najmanjeg prioriteta od dosad spomenutih. Stoga možemo izostaviti zagrade u već spomenutom primjeru:

```
boolean ok = (x > 0);
```

U slijedećem dijelu su primjeri upotrebe logičkih operatora.

5. Usporedba stringova

Jedan od najčešće upotrebljavanih metoda koja vraća `boolean` vrijednost je metoda `equals.` Svaki string (string je klasa) ima `equals` metodu koja se koristi da li string za kojeg smo pozvali metodu ima istu sekvencu karaktera kao i string koji je proslijeđen kao parametar.

Slijedi primjer u kojemu testiramo da li se string referenciran s st odnosi na isti niz znakova koji su proslijeđeni kao parametar:

```
st.equals(″cat″)
```

Slijedi primjer jednostavnog programa koji testira sadržaj stringa na navedeni način. program pokušava prepoznati što je korisnik utipkao te daje odgovarajuće odgovore.

PRIMJER 6

```
public class KakoSi
{  /* Pitaj korisnika “Kako je” i daj odgovarajući
     komentar.

   */

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

      System.out.println
         (″Ćao korisniče. kako ide danas ?");
      String odgovor = in.readLine();
      if (odgovor.equals("Dobro"))
         System.out.println
            ("Lijepo je čuti da nekom ide dobro.");
      else if (odgovor.equals("Loše"))
         System.out.println("Žao mi je, valjda će biti bolje");
   }
}
```

Operator == se normalno ne koristi da bi se ispitalo da li su dva stringa ista. Razlog je što Java izraz:

```
S1 == S2
```

gdje S1 i S2 referenciraju stringove interpretira na način da uspoređuje reference, a ne sadržaj stringova. Znači, testira se da li obje reference pokazuju na isti string u memoriji.

Promotrite slijedeći kod:

```
String S1 = “Isti smo”;
String S2 = “Isti smo”;

if(S1==S2){
   System.out.println("Isti")
}
```

```
else{
    System.out.println("Različiti")
}
```

Rezultat ispisa bit će "Različiti" jer varijable S1 i S2 referenciraju na **različite stringove** s istim sadržajem.

Korisna alternativa `equals` metodi je slijedeća metoda.

```
st1.equalsIgnoreCase(st2)
```

Ova metoda ne razlikuje velika i mala slova pa ako npr. `st1` je `"the dog", a st2` je `"The Dog"`, onda će metoda vratiti `true`.

Postoji još mnogo korisnih metoda vezanih za stringove koje možete pogledati u Java dokumentaciji.

Slijedi primjer appleta koji na osnovu korisnikova unosa željenog oblika i boje iscrtava lik na ekranu. Prvo se traži naziv boje te se na osnovu unosa kreira odgovarajući objekt.

Slijedi dio za unos i kreiranje boje:

```
String odgovor1 =
   JOptionPane.showInputDialog
      ("Naziv željene boje?");
if (odgovor1.equalsIgnoreCase("crvena"))
   color = Color.red;
else if (odgovor1.equalsIgnoreCase("žuta"))
   color = Color.yellow;
else if (odgovor1.equalsIgnoreCase("plava"))
   color = Color.blue;
else
   color = null;
```

Program prepoznaje samo nazive ‘crvena’, ‘žuta’ i ‘plava’. Ako korisnik utipka bilo što drugo npr. ‘zelena’ ili ‘% 7df&#yx!’, program će spremiti `null` u varijablu `color`.

Slijedeći kod pita korisnika za oblik i rezultat sprema u varijablu `shape`. Program prepoznaje samo dva oblika.

```
String odgovor2 =
   JOptionPane.showInputDialog
      ("Koji oblik želite?");
if (odgovor2.equalsIgnoreCase("kvadrat"))
   shape = new Rectangle2D.Double(50,50,200,200);
else if (odgovor2.equalsIgnoreCase("krug"))
   shape = new Ellipse2D.Double(50,50,200,200);
else
   shape = null;
```

Kad smo postavili vrijednosti u varijable color i shape izvršit ćemo dvije slijedeće naredbe:

```
if (color != null)
   g2.setColor(color);
if (shape != null)
   g2.fill(shape);
```

Ako smo unijeli prepoznatljivu boju Graphics2D objekt bit će postavljen na zadanu boju, a ako nismo , bit će ostavljena njegova prethodna boja.
Ako je program prepoznao lik onda će ga nacrtati, inače ne.

Slijedi kompletan program:

PRIMJER 7

```
// Applet koji vas pita za boju i oblik
// te prikazuje unijeti oblik ispunjen sa zadatom bojom

import java.applet.*;
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class Oblici extends Applet

{  private Color boja;
   private Shape oblik;

   public void init()
   {
      String odgovor1 =
         JOptionPane.showInputDialog
            ("Naziv željene boje?");
      if (odgovor1.equalsIgnoreCase("crvena"))
         boja = Color.red;
      else if (odgovor1.equalsIgnoreCase("žuta"))
         boja = Color.yellow;
      else if (odgovor1.equalsIgnoreCase("plava"))
         boja = Color.blue;
      else
         boja = null;


      String odgovor2 =
         JOptionPane.showInputDialog
            ("Koji oblik želite?");
      if (odgovor2.equalsIgnoreCase("kvadrat"))
         oblik = new Rectangle2D.Double(50,50,200,200);
      else if (odgovor2.equalsIgnoreCase("krug"))
         oblik = new Ellipse2D.Double(50,50,200,200);
      else
         oblik = null;
   }

   public void paint(Graphics g)
   {  Graphics2D g2 = (Graphics2D) g;

      /* Prikaži obojani lik. */

      if (boja != null)
         g2.setColor(boja);

      if (oblik != null)
         g2.fill(oblik);
   }
}
```

Analizirajmo formu appleta. Postoje dva osnovna zadatka:

Korak 1. Unesi željenu boju i oblik.

Korak 2. Naslikaj željeni lik u željenoj boji.

Drugi korak će se izvršavati svaki put kad bude potrebno osvježiti sadržaj područja prikaza.
Stoga je uključen u applet `paint` metodu.
S druge strane prvi korak treba biti izvršen samo jednom pa je uključen u `init` metodu.

Za zadnji primjer u ovom poglavlju slijedi dodavanje još jedne metode u klasu Student. Nazvat ćemo je `edit`. Koristit će `equals` metodu za usporedbu stringova i logičke operatore.

```
s.edit(in)
```

Prikazuje podatke o objektu `s` tipa Student, pita korisnika da li želi napraviti kakve promjene. Ako želi `s` se odgovarajuće ažurira. Korisnikov unos učitavamo pomoću ConsoleReader `objekta`.

Metoda `edit` prikazuje redom vrijednost svakog polja, i čeka korisnikov odgovor. Ako je korisnik zadovoljan s trenutnom vrijednošću onda utipka samo Return (Enter). Ako unese neki string metoda će pretpostaviti da se radi o novoj vrijednosti i spremiti je u odgovarajuće polje.

Primjer mogućeg prikaza na ekranu prilikom izvršavanja metode edit:

```
ID (9912345)
Ime (Jure Antić)
Program studija (Računarstvo) CS
Godina (1) 2
Upisan? (Da) no
```

Svaka se linija sastoji od naziva informacije i njene trenutne vrijednosti te podebljano ispisanog korisnikovog unosa. U slučaju prva dva retka korisnik je samo utipkao Enter i odgovarajuće vrijednosti su ostale iste. Slijedeća tri retka korisnik je unio nove podatke i potrebno ih je pohraniti u odgovarajuća polja.

Slijedi kod koji radi s ID brojem.

```
System.out.print("ID (" + idBroj +") " );
String odgovor = input.readLine();
if (! odgovor.equals(""))
   idBroj = odgovor;
```

Za studentovo ime postupa se skoro na isti način.

Za program studija se postupa na malo drukčiji način.
ispitivanje sadržaja stringa obavlja se na slijedeći način:

```
if (!odgovor.equals(""))
   if (odgovor.equals("Elektronika") ||
        odgovor.equals("Računarstvo") ||
        odgovor.equals("Strojarstvo") ||
        odgovor.equals("Brodogradnja") )
      programStudija = odgovor;
   else
      System.out.println("Nepoznat program studija.");
```

Metoda `edit` postupa s poljem `jeUpisan` opet na različit način. Umjesto ispisa stvarnog sadržaja ispisuje se 'da' ili 'ne' za indikaciju upisa studenta. Korisnik mora odgovoriti s 'da' or 'ne' ako želi napraviti promjenu u tom polju

Slijedi potpuna metoda `edit`:

PRIMJER 8

```
/* Editiraj studentove podatke
*/

public void edit(ConsoleReader input)
{
   System.out.print("ID (" + idBroj +") " );
   String odgovor = input.readLine();
   if (! odgovor.equals(""))
      idBroj = odgovor;

   System.out.print("Ime (" + ime + ") ");
   odgovor = input.readLine();
   if (! odgovor.equals(""))
      ime = odgovor;

   System.out.print
      ("Program studija (" + programStudija + ") ");
   odgovor = input.readLine();
   if (! odgovor.equals(""))
  if (odgovor.equals("Elektronika") ||
     odgovor.equals("Računarstvo") ||
         odgovor.equals("Strojarstvo") ||
         odgovor.equals("Brodogradnja") )
        programStudija = odgovor;
      else
         System.out.println("Nepoznat program studija.");

   System.out.print("Godina (" + godina + ") ");
   odgovor = input.readLine();
   if (! odgovor.equals(""))
      godina = Integer.parseInt(odgovor);

   if (jeUpisan)
      System.out.print("Upisan? (da) ");
   else
      System.out.print("Upisan? (ne) ");
   odgovor = input.readLine();
   if (! odgovor.equals(""))
      if (odgovor.equals("da"))
         jeUpisan = true;
      else if (odgovor.equals("ne"))
         jeUpisan = false;
      else
         System.out.println("Odgovor nije razumljiv.");
}

public class TestStudent

{  /* Čita podatke o studentu i kreira odgovarajući
  Student objekt.
```

```
   Zatim editira objekt Student
        te na kraju prikazuje sadržaj.
    */

    public static void main(String[] args)

    {  /* Kreiraj objekt tipa ConsoleReader za učitavanje

          podataka s tastature */

       ConsoleReader in =
          new ConsoleReader(System.in);

       /* Kreiraj objekt Student. */

       System.out.println
          ("Koji je ID broj studenta?");
       String i = in.readLine();
       System.out.println("Koje je ime studenta?");
       String n = in.readLine();
       System.out.println("Koji je program studija?");
       String d = in.readLine();
       Student st = new Student(i,n,d);

       /* Editiraj pa prikaži podatke studenta. */

       System.out.println();
       System.out.println("Editiraj podatke studenta.");
       st.edit(in);
       System.out.println();
       System.out.println("Student podaci:");
       st.prikaz();
    }
}
```

Slijedi što bi program mogao ispisivati (unos korisnika podebljano):

```
Koji je ID broj studenta?
9912345
Koje je ime studenta?
Jure Antić
Koji je program studija?
Računarstvo

Editiraj podatke studenta.
ID (9912345) 9912346
Ime (Jure Antić)
Program studija (Računarstvo) Elektronika
Godina (1) 2
Upisan (ne) da

Student podaci:
Student ID: 9912346
```

```
Ime: Jure Antić
Program studija: Elektronika
Godina: 2
Upisan.
```

U kodu programa postoji izraz slijedećeg oblika(ispitivanje polja programStudija):

```
if (B1) if (B2) S1 else S2
```

Kojemu `if` pripada `else`.
Ovo je primjer tzv. visećeg else.

Java će to interpretirati na slijedeći način:

1. `if (B1) { if (B2) S1 else S2 }`

a ne ovako:

2. `if (B1) { if (B2) S1 } else S2`

Pravilo je da `else` tvori cjelinu s najbližim `if`-om.

U ovim slučajevima preporučljivo je koristiti vitičaste zagrade !

Kod sa vitičastim zagradama:

```
if (! odgovor.equals(""))
{  if (odgovor.equals("da"))
      jeUpisan = true;
   else if (odgovor.equals("ne"))
      jeUpisan = false;
   else
      System.out.println("Odgovor nije razumljiv.");
}
```

6. Blokovi i doseg varijabli

Zapamtite da je blok sekvenca naredbi zatvorena u vitičaste zagrade. Jedna je bitna stvar u radu s blokovima u Javi. Varijabla koju deklaracijom unutar bloka vrijedi samo unutar bloka. čim Java napusti blok varijable iščezava tj. oslobađa se memorija koja je za nju bila korištena.
promotrite slijedeći primjer:

```
if (x == 0)
{  System.out.println(”x is nula.”);
   int y = 1;
} else {
   System.out.println(”x nije nula.”);
   int y = 2;
}
System.out.println(”y je ” + y);
```

Ovaj program će javiti pogrešku u toku prevođenja. varijabla y ne vrijedi izvan bloka u kojem je deklarirana. Što više programer je kreirao dvije posebne varijable y unutar posebnih blokova.

Dio programa u kojemu možemo koristiti određenu varijablu naziva se **doseg** (scope) te varijable.

prethodni program ima više smisla (kompajlirat će se) ako ga preuredimo na slijedeći način:

```
int y;
if (x == 0)
{  System.out.println("x je nula.");
   y = 1;
} else {
   System.out.println("x nije nula.");
   y = 2;
}
System.out.println("y je " + y);
```

varijabla deklarirana unutar bloka naziva se **lokalna varijabla**. Tijelo metoda je također blok pa tako svaka varijabla koja je deklarirana unutar metode je lokalna varijabla.
To znači da bi svaka varijabla deklarirana unutar metode trebala prestati s postojanjem nakon izlaska iz metode.

S druge strane, polje objekta postoji sve dok postoji objekt kome pripada. Objekt traje sve dok Java ne zaključi da se više ne upotrebljava. Taj proces nije nimalo jednostavan kako to pokazuje naziv **garbage collection** (odvoz smeća).

Činjenica da lokalne varijable vrijede samo dok se metod izvršava objašnjava nam zašto smo u primjeru 7 upotrijebili dva nova polja Color i Shape. Pošto smo tim referencama u Init metodi dodijelili objekte boje i oblika, reference su trebale ostati sačuvane da bi objektima mogla pristupiti paint metoda.

7. Zadaci

Napiši program (TriBroja.java) koji čita tri broja u pokretnom zarezu i ispisuje najveći od njih:

Npr:

```
Unesi tri broja:
4  9  2.5
Najveći je broj 9.
```

2. Napiši applet (KrugTest.java, Krugtest.txt (HTML))koji crta krug radijusa 100 s centrom u (110,120). Zatim traži od korisnika da unese koordinate neke točke. Ako točka leži unutar Kruga onda ispisuje poruku "Unutar kruga", a ako leži vani poruku "Van kruga".

3. Godina s 366 dana naziva se prestupna godina. Godina je prestupna ako je djeljiva s 4 (npr. 1980), osim ako nije djeljiva s 100 (npr. 1900). Izuzetak su godine djeljive s 400 (npr. 2000) koje su prestupne u svakom slučaju. Nije bilo prestupnih godina prije uvođenja Gregorijanskog kalendara na dan 15.10.1582 (npr., 1500. ipak nije bila prestupna). napiši program koji će korisnika pitati za godinu , a onda ispisati je li ili nije prestupna.

   Provjera 1996 i 2000 su prestupne. 1900 i 1999 nisu.

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.

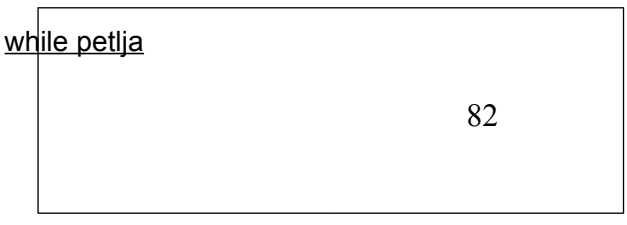
```
System.out.println(″Unesi vrijednosti:″);
double suma = 0;

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:

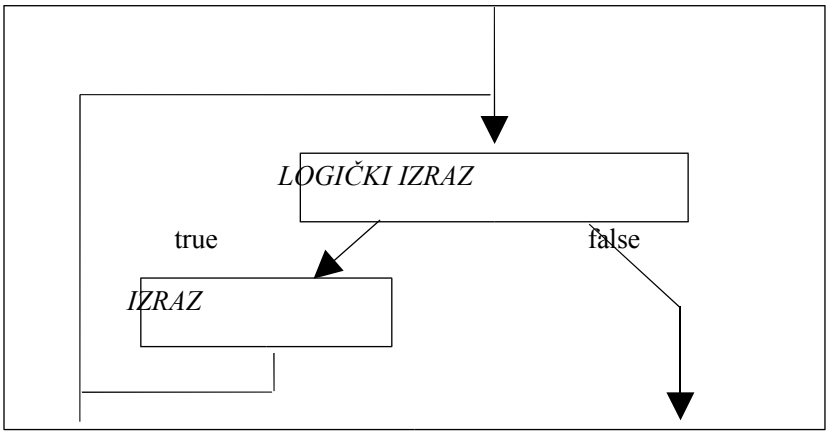


```
while (LOGIČKI IZRAZ)
    IZRAZ
```

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

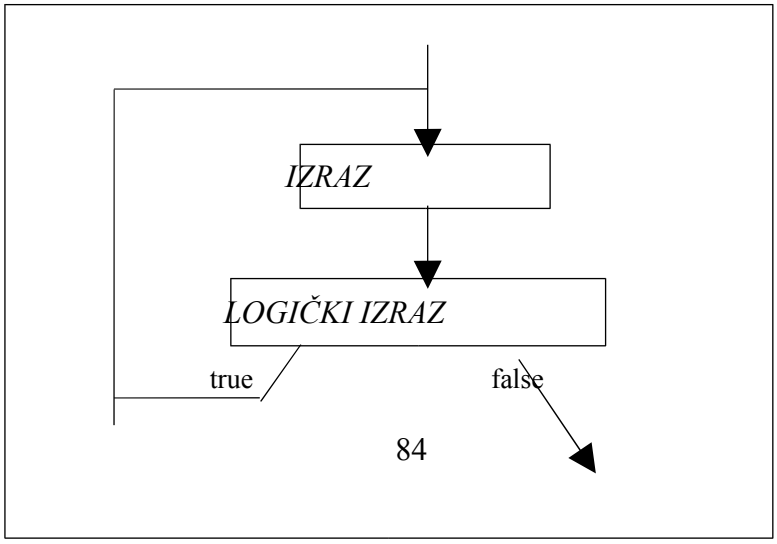
```
do
   IZRAZ
while (LOGIČKI IZRAZ);
```

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 :

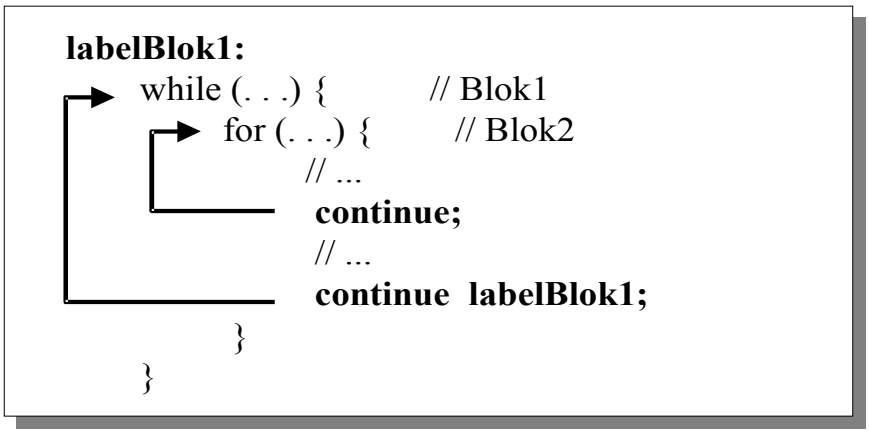
```
break labela;
```

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.

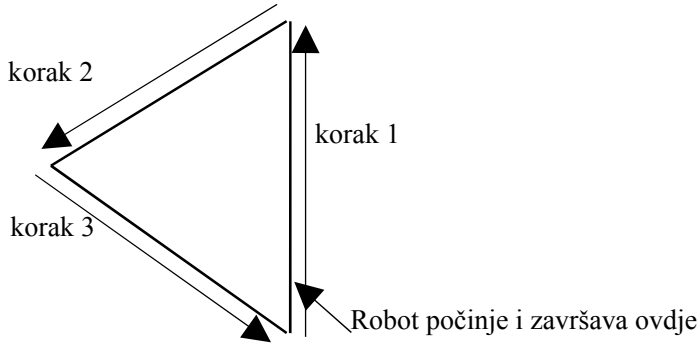
`r.olovkaGore()` Diže olovku robota.

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);
```

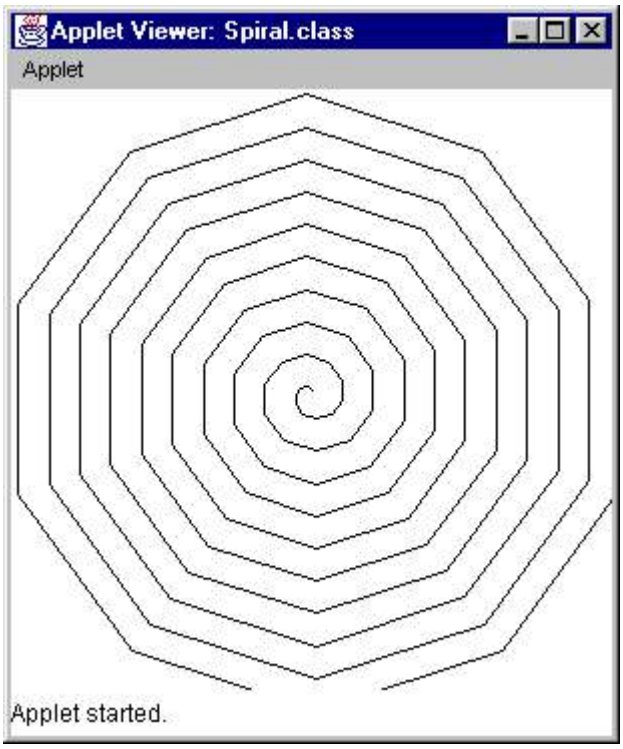
Svaki put kad se robot pokrene ide naprijed za udaljenost od 50(piksela) sa spuštenom olovkom crtajući liniju na ekranu. Svaki put kad stigne do kraja linije okrene se za kut od 120° nakon čega slijedi opet crtanje linije.



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.

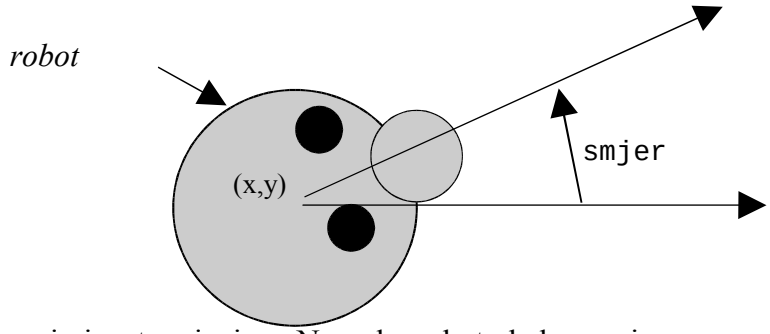
Tako će klasa `Robot` imati slijedeća polja:

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.

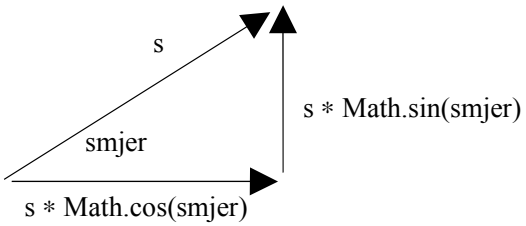
```
/** Podigni olovku gore.
*/
public void olovkaGore()
{  jeDolje = false;
}
```

```
/** 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.

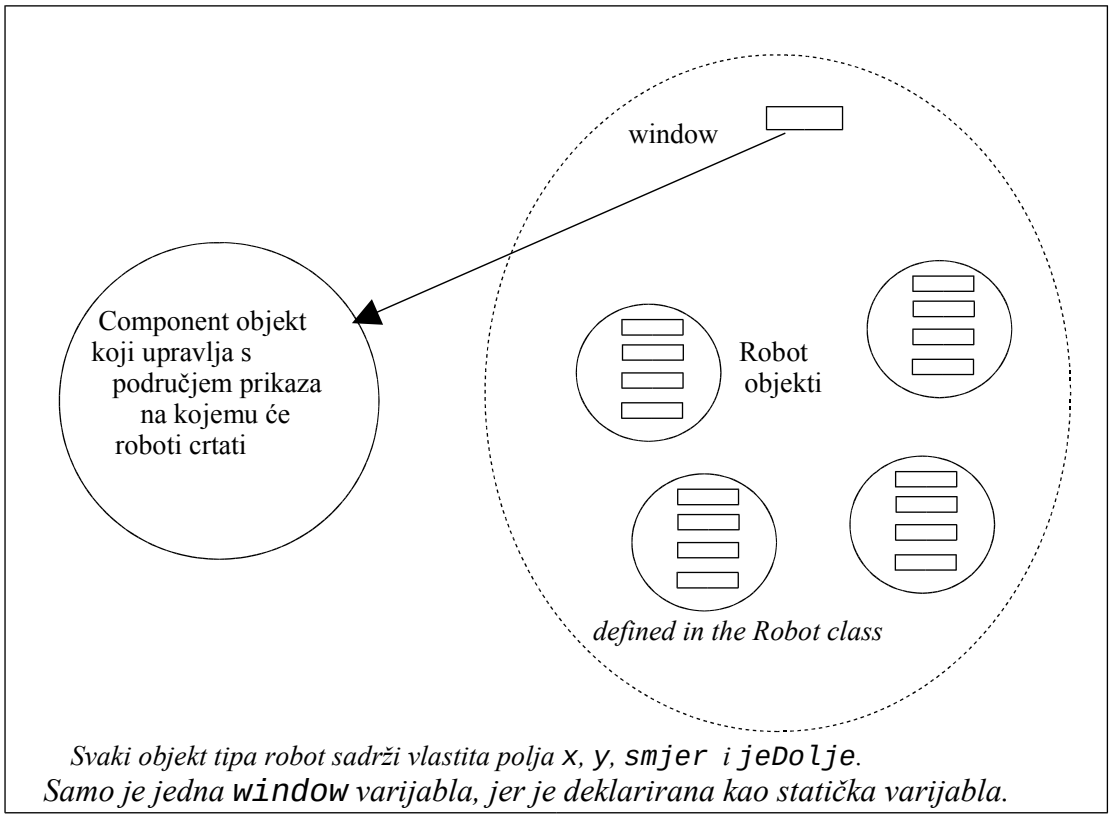
Statička varijabla `window` nije polje objekta Robot.

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.

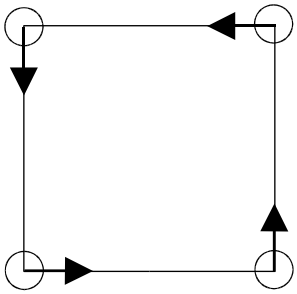
Slijedi applet koji koristi jednog robota za crtanje spirale koju smo već pokazali.

PRIMJER 1

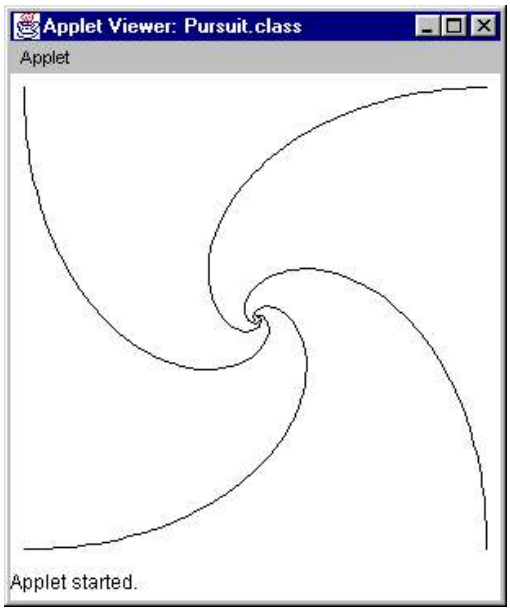
```
import java.applet.Applet;
import java.awr.Graphics;

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:



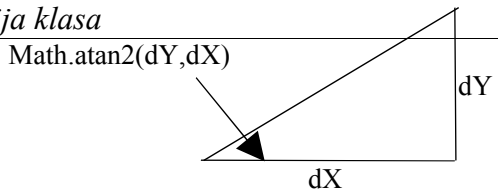
Metode instance koje smo dosada napisali nisu dovoljne za realizaciju ovog programa. Pomoću njih ne možemo ostvariti da jedan robot gleda u drugoga. Bit će potrebno dodati slijedeću metodu:

```
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.

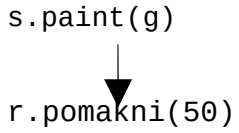
```
public void paint(Graphics g)
{  Robot r = new Robot();
   r.olovkaDolje();
   for (int i = 0; i < 100; i++)
   {  r.pomakni(i);
      r.lijevo(36);
   }
}
```

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:

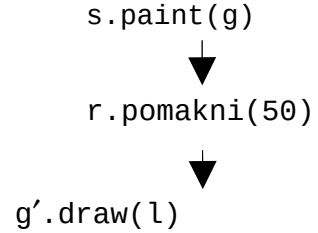


te nam pokazuje dase trenutno izvršava `s.paint(g)` , te je tametoda pozvala `r.pomakni(50)`, koja se također izvršava. Izvršavanje metode `s.paint(g)` će biti zaustavljeno dok metoda `r.pomakni(50)` ne završi.

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

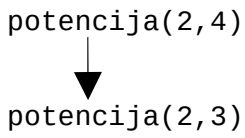
```
public class PotencijaProg

{  /* 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

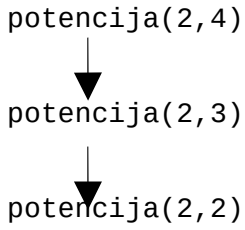
Pretpostavimo da je program pokrenut i da će korisnik unijeti vrijednosti 2 i 4.

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

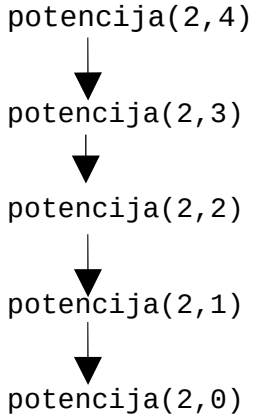
Od ove točke skoncentrirati ćemo se na lanac pozvanih metoda. Za prvi poziv metode `potencija` vrijednosti parametara bit će n = 2 i p =4. U tom slučaju pošto je p>=0 bit će izvršena druga grana tj. Java će vratiti (`return`) vrijednost koja će se dobiti izvršavanjem izraza `potencija(n,p-1)*n`, a to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

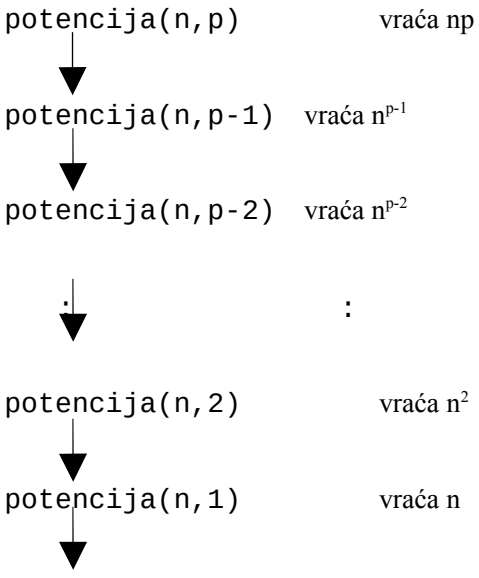


Slično kad se `potencija(2,2)` je izvršava, zvat će `potencija(2,1)`. Kad se `potencija(2,1)` izvršava, zvat će se `potencija(2,0)`.



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
```

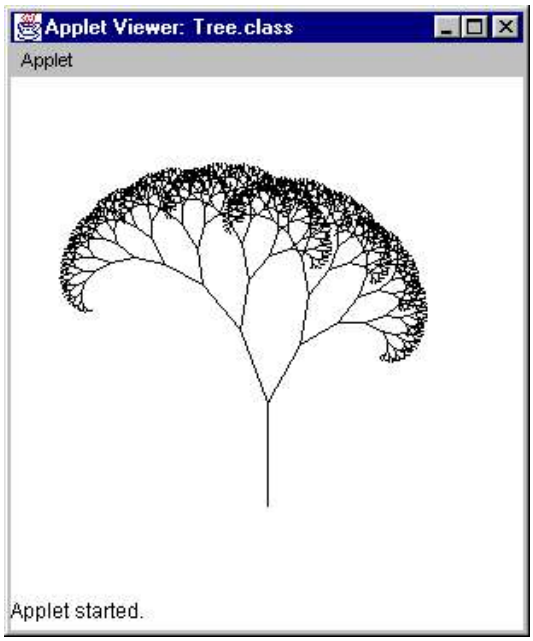
Općenito svaka rekurzivna metoda može biti napisana na nerekurzivan način. Međutim postoje slučajevi kad je rekurzivna verzija značajno jednostavnija.

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)`.

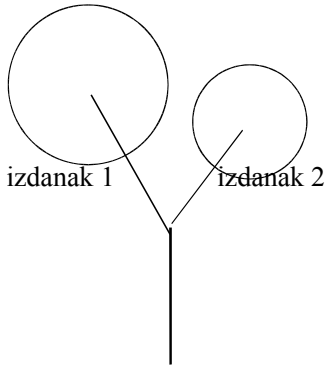


PRIMJER 4

```
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);
    }
}
```

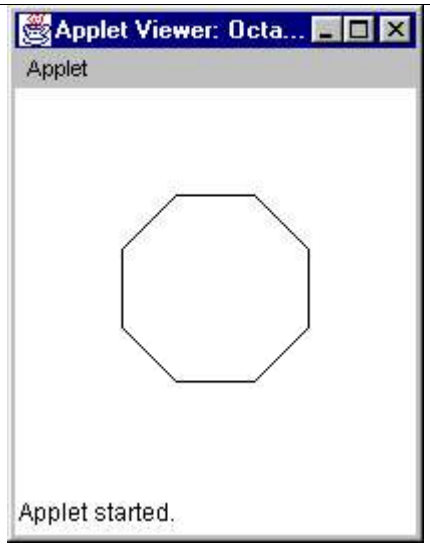
Metod radi na slijedeći način. Ako je veličina grane premalena metod ne radi ništa. Ako nije, nacrtat će granu i dodati dva nova izdanka. Izdanci su razmaknuti za neki kut i oba predstavljaju nova stabla. Izdanci se crtaju pozivom iste metode `crtajStablo` .



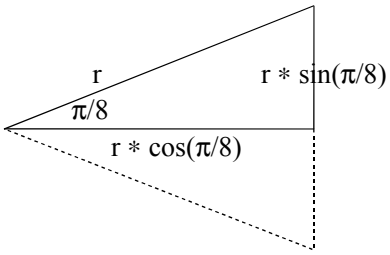
Nije jednostavan odgovor napitanje : Kada koristiti rekurziju ?
Stablo nam daje ključ. Rekurzije se najčešće koriste za programiranje i rad s rekurzivnim strukturama podataka. (npr. direktoriji !)

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.

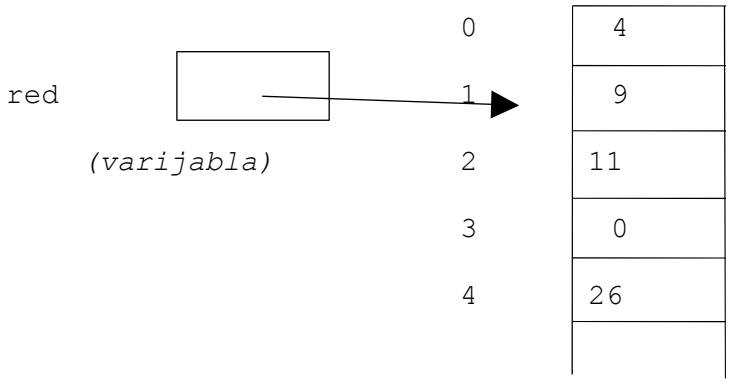
Npr. želite imati niz s elementima tipa `int` . Takav niz se označava s `int[].` Možemo imati i nizove drugih tipova podataka npr. niz `String[]` čiji elementi sadržavaju reference na objekte tipa String, ili niz `Robot[]` koji sadržava reference na objekte tipa `Robot` , itd.

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.:

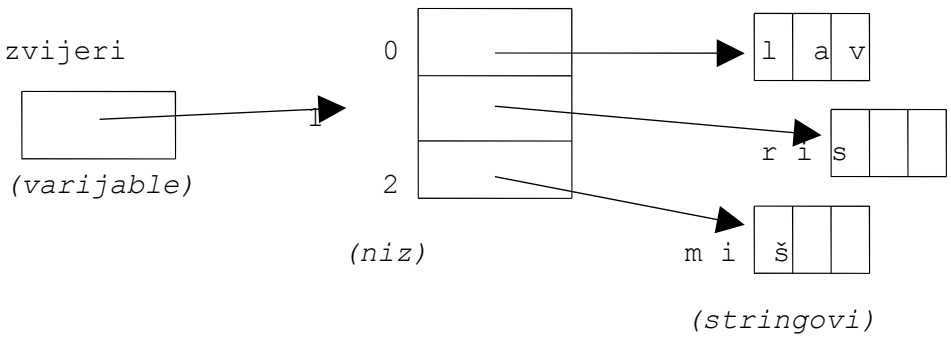
```
int[] red;
```

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:

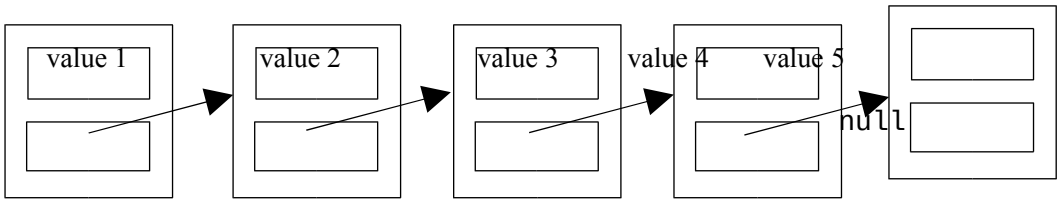
```
miš mačka pas
```

3. Sučelje List (List interface).

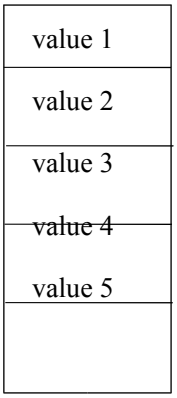
Osim nizova postoje i druge tehnike za pohranu velike količine podataka. Jedna od njih je vezana za pojam Liste (List) i predstavlja jedan od najčešćih i najfleksibilnijih oblika organizacije podataka u programiranju..

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

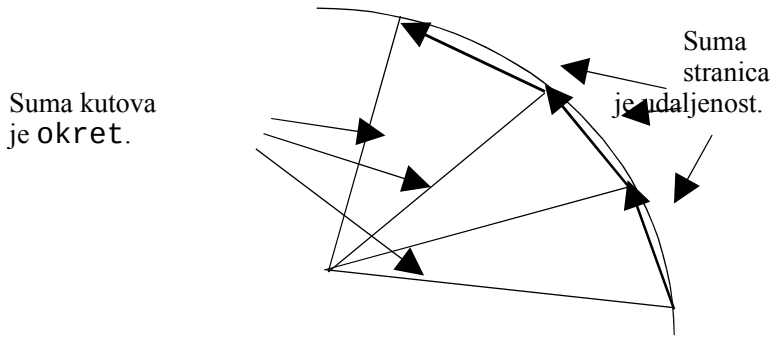
1. Nadogradnja klase (KrivuljaRobot)

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
```

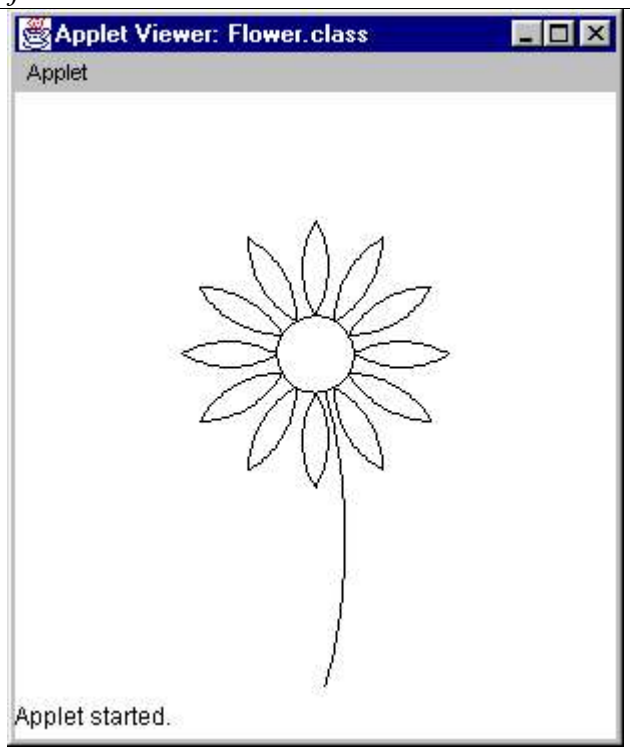
kaže Javi da uključi sve definicije koje se pojavljuju u originalnoj definiciji klase Robot. To znači da će objekt tipa KrivuljaRobot imati ista polja `x`, `y`, `smjer`, `jeDolje` kao i objekt Robot ; imat će iste metode `pomakni`, `lijevo`, `desno`, `olovkaGore` i `olovkaDolje` kao objekt Robot , plus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

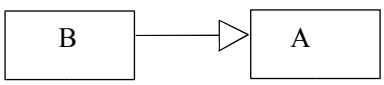
Međutim on se ne pojavljuje u definiciji klase KrivuljaRobot !
Ono što se ovdje događa je da nam Java sama osigurava konstruktor. Dakle, ako za neku klasu ne definirate konstruktor Java će vam sama osigurati konstruktor koji nazivamo **default** konstruktor. Taj konstruktor ne radi ništa osim što će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

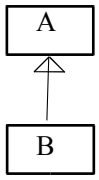
Tako u klasi Robot možemo napisati dvije metode pristupa:

```
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)

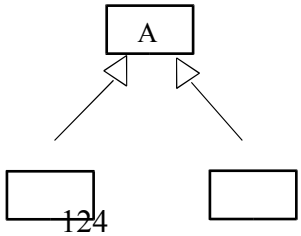
U prijašnjem odjeljku imali smo slijedeću situaciju. Počeli smo s definicijom klase A te zatim odlučili da je nadogradimo s definicijom klase B . To je predstavljeno sa slijedećim dijagramom.



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.

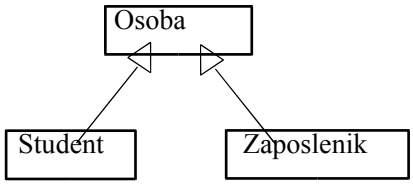


B1 B2

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()
```

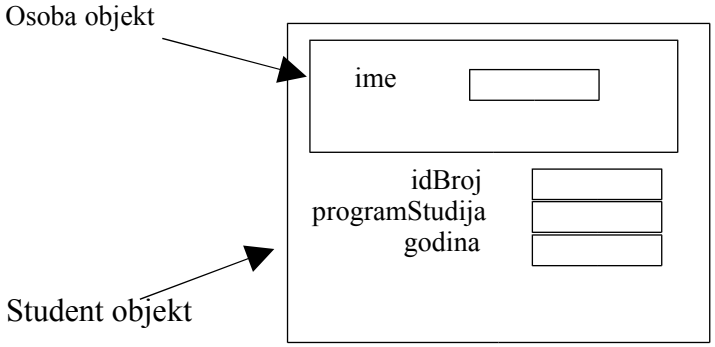
```
    {  return ime;
    }

    /* 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.
```

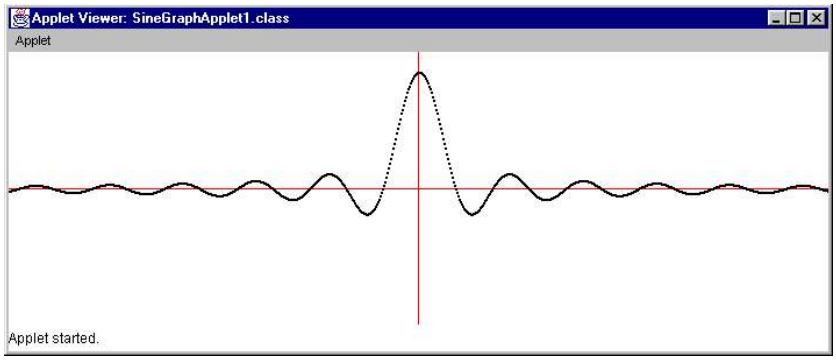
```
        */
        abstract public double function(double x);
    }
```

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:



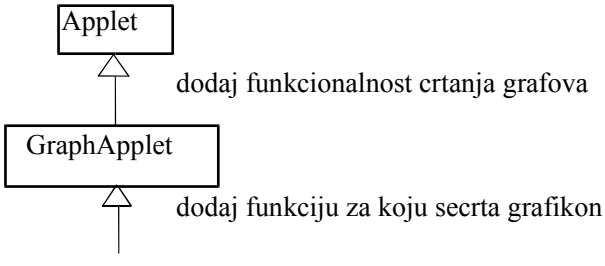
5. Stablo familije klasa

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.

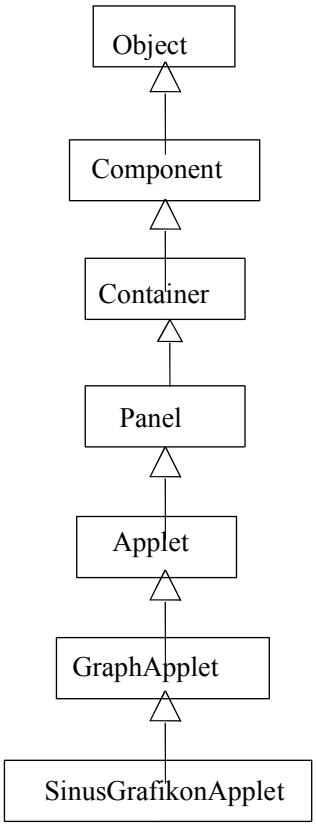
U prethodnom odjeljku išli smo još korak dalje. Prvo smo nadogradili Applet i izveli subklasu GraphApplet. Dodali smo funkcionalnost vezanu za crtanje grafova. Applet SinusGrafikonApplet nadogradio je GraphApplet naslijedivši funkcionalnost i Applet i GraphApplet klase te je još dodao dvije metode.

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:

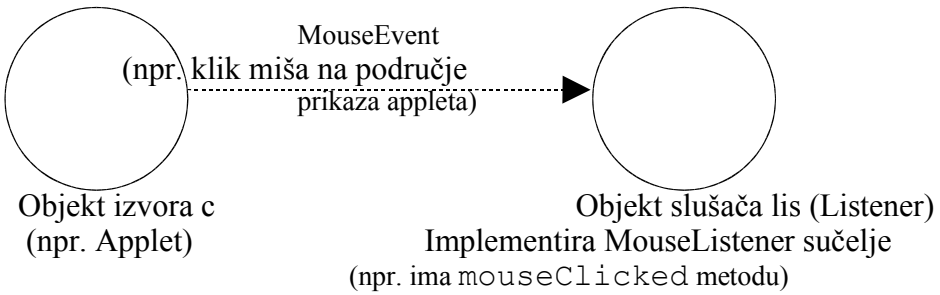
Metoda slušač ne može biti samostalna u memoriji. Ona mora biti pridružena s objektom koji nazivamo objekt slušač (**listener object)**. Možemo odabrati bilo koji implementira *MouseListener* sučelje(interface). To znači da taj objekt mora imati definirano slijedećih pet metoda: `mouseClicked`, `mouseEntered`,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

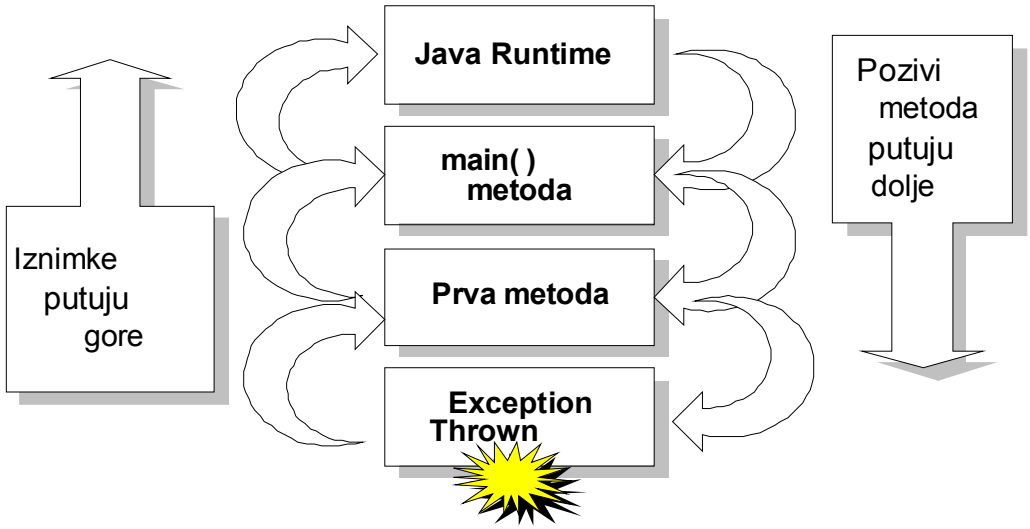
Kada interpreter izvršava `try-catch` izraz, prvo počinje s izvršavanjem naredbi u try bloku. Naredbe se izvršavaju normalnim slijedom i ako se nešto ne baci iznimku neće se izvršiti nijedan od catch blokova. Ako je dodan blok finally bit će izvršen nakon try bloka.

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

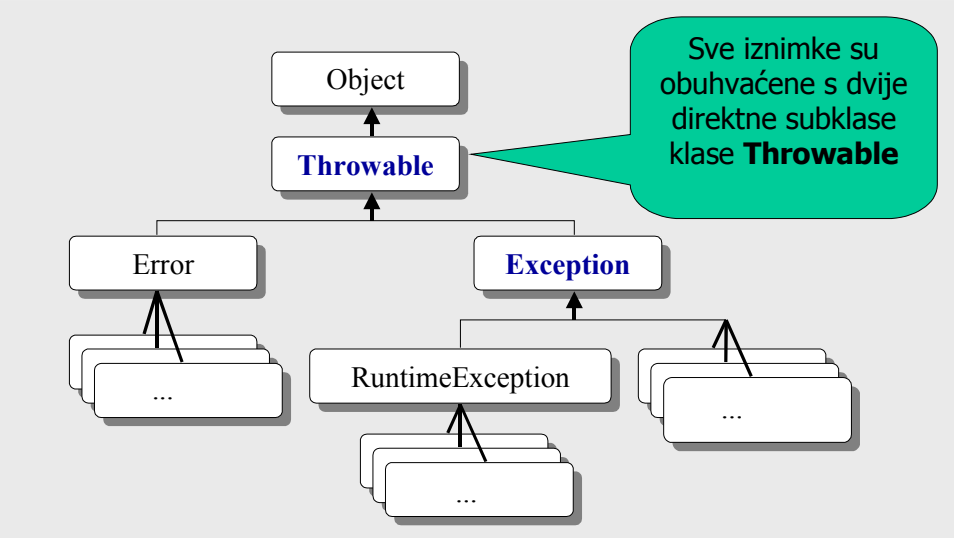
```
public class Prosjek1

{  /* 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

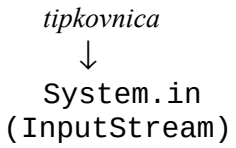
Kada u Java programu čitamo podatke s nekog ulaznog medija onda koristimo objekt koji upravlja s ulazom. Objekt se spaja na izvor podataka, npr. tipkovnicu ili tekstualnu datoteku. Da bismo čitali podatke koristimo metode tog objekta. Objekti tipa ConsoleReader koji je korišten u primjerima za čitanje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

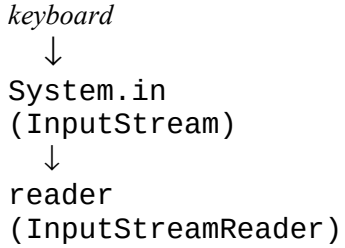


Java posjeduje i klasu **InputStreamReader** koja učitava podatke u Unicode formatu odnosno kao 16-bitne unicode znakove. Postoji konstruktor kojim je moguće pretvoriti InputStream u InputStreamReader. Referenca na objekt tipa InputStream je parametar konstruktora:

```
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:

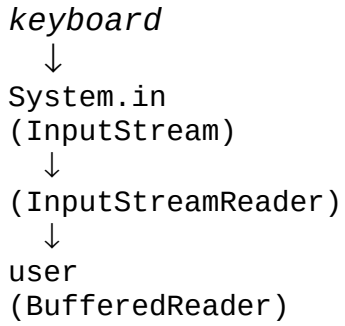
BufferedReader, **LineNumberReader**.

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.