Donošenje odluka u programu

Osnovni element svih programa je donošenje odluka. Morate biti u stanju da izaberete između različitih mogućnosti, na primer, "Ako pada kiša uključiću brisače na svom automobilu, a ako ne pada neću." Programski se donošenje odluka realizuje pomoću relacionih i logičkih operatora i odgovarajućih uslovnih izraza.

Relacioni operatori

Relacioni operatori služe za poređenje vrednosti operanada.

Operator	Upotreba	Vraća `true` (tačno) ako je
`>`	`op1 > op2`	`op1` veće `op2`
`>=`	`op1 >= op2`	`op1` veće ili jednako od `op2`
`<`	`op1 < op2`	`op1` manje `op2`
`<=`	`op1 <= op2`	`op1` manje ili jednako `op2`
`==`	`op1 == op2`	`op1` i `op2` su jednaki
`!=`	`op1 != op2`	`op1` i `op2` su različiti

Ovi operatori vraćaju tačno ili netačno.

U sledećem primeru je pokazano kako bi mogli da se koriste ovi operatori.

```
public class RelacioniOperatori {
    public static void main(String[] args) {
        //Nekoliko brojeva
        int i = 37;
        int j = 42;
        int k = 42;
   //vece od
        System.out.println("i > j = " + (i > j));  //false

   //vece ili jednako
        System.out.println("Vece ili jednako...");
        System.out.println("    i >= j = " + (i >= j));   //false
   //manje od
        System.out.println("Manje od...");
        System.out.println("    i < j = " + (i < j));     //true
   //manje ili jednako
        System.out.println("Manje ili jednako...");
        System.out.println("    i <= j = " + (i <= j));   //true
   //jednako
        System.out.println("Jednako...");
        System.out.println("    i == j = " + (i == j));   //false
   //razlicito
        System.out.println("Razlicito...");
        System.out.println("    i != j = " + (i != j));   //true
    }
}
```

Izlaz iz ovog programa je:

```
Vece od...
```

```
    i > j = false
Vece ili jednako...
    i >= j = false
Manje od...
    i < j = true
Manje ili jednako...
    i <= j = true
Jednako...
    i == j = false
Razlicito...
    i != j = true
```

I pored ovakvog primera, relacioni operatori se uglavnom koriste u okviru uslovnih iskaza, o kojima će kasnije biti više reči.

Relacioni operatori se često koriste zajedno sa logičkim operatorima, čime se dobijaju složeni izrazi.

Logički operatori

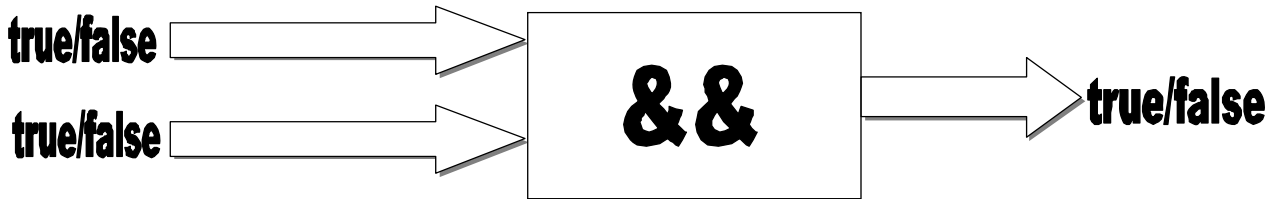
Izraz a > b je relacioni izraz. Relacioni izraz je izraz koji vraća boolean vrednost, a koji za izračunavanje vrednosti koristi relacioni operator. Ovi izrazi se nazivaju i bulovim izrazima. Bulovi izrazi se mogu sastojati od drugih, manjih bulovih izraza. Ovo je slično sa ljudskim jezikom u kome su složene rečenice sastavljene od manjih izraza.

Kombinovanje relacionih izraza u veće relacione izraze se vrši pomoću logičkih operatora.

Logički operatori deluju na operande koji su tipa boolean. Najvažniji logički operatori su logičko I, logičko ILI i logička negacija.

Logičko I (&&)

Ovaj operator se u Javi označava sa &&. Šematski se ovo može predstaviti na sledeći način:



Vrednost izraza sa logičkim I je tačna ako su oba operanda tačna. Ako je makar jedan od njih netačan (ili oba) ceo izraz je netačan. To se može predstaviti sledećom tabelom:

	tačno	netačno
tačno	tačno	netačno
netačno	netačno	netačno

Pretpostavimo da je profesor rešio da svoje studente koji su polagali test oceni na sledeći način:

- Oni koji su osvojili od 50 do 60 poena dobijaju ocenu 6.
- Oni koji su osvojili od 60 do 70 poena dobijaju ocenu 7.
- Oni koji su osvojili od 70 do 80 poena dobijaju ocenu 8.
- Oni koji su osvojili od 80 do 90 poena dobijaju ocenu 9.
- Oni koji su osvojili od 90 do 100 poena dobijaju ocenu 10.

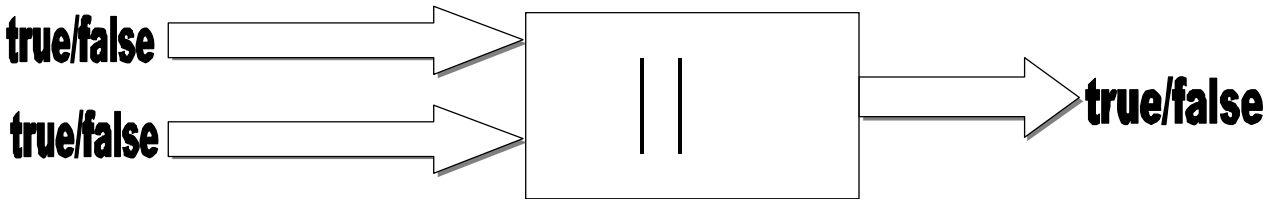
Ovo se mora izraziti preko logičkih operatora. Ako se broj poena smesti u promenljivu ocena, onda bismo mogli da napišemo:

- ocena > 50 && ocena < 60
- ocena > 60 && ocena < 70

- ocena > 70 && ocena < 80
- ocena > 80 && ocena < 90
- ocena > 90 && ocena < 100

Logičko ILI (||)

Ovaj operator se u Javi označava sa ||.



Vrednost izraza sa logičkim ILI je tačna ako je makar jedan od operanada tačan. Izraz je netačan samo ako su oba netačna. To se može predstaviti sledećom tabelom:

	tačno	netačno
tačno	tačno	tačno
netačno	tačno	netačno

Logička negacija (!)

Ovaj operator se u Javi označava znakom uzvika (!). Ovaj operator deluje na jedan operand tipa boolean i menja njegovu vrednost. Ako je vrednost operanda tačno, operator negacije je menja na netačno i obrnuto. Ovo se može predstaviti sledećom tabelom.

x	!x
tačno	netačno
netačno	tačno

Operand na koji deluje operator logičke negacije može biti i složen relacioni ili logički izraz. U tom slučaju je potrebno taj izraz staviti u zagrade.

```
! (a > b && a < c)
```

Kada se ovaj operator koristi kod složenih izraza treba voditi računa o njegovom prioritetu. On ima viši prioritet u odnosu na relacione i aritmetičke operatore, tako da se izvršava pre njih. Pogledajte sledeći izraz u kome se pita da li računar ima brzinu veću od 2200 MHz i memoriju veću od 512 MB. Negacija izraza bi bila:

```
! (  speed > 2000   &&   memory > 512 )
     ------+------      -------+---
           |                   |
! (        T       &&          T     )
           ---------+-----------
                    |
! (                  T               )

                   !T

                    F
```

Ako bi se ovaj izraz napisao bez zagrada dobili bi sledeće:

```
!speed > 2000   &&   memory > 512
--+---
neispravno: na aritmetičku promenljivu se ne može primeniti operator !
```

Pošto ! ima viši prioritet, kompajler će probati da ga primeni na promenljivu brzina, što neće raditi.

Uslovni iskazi

Osnovni element svih programa je donošenje odluka. Morate biti u stanju da izaberete između različitih mogućnosti, na primer, “Ako pada kiša uključiću brisače na autu, a ako ne pada neću". Programski se donošenje odluka realizuje pomoću relacionih i logičkih operatora i odgovarajućih uslovnih iskaza.

Iskaz if

Iskaz if je elementarni uslovni izraz. Njegov najjednostavniji oblik je:

```
if (izraz)
   iskaz;
```

Izraz može biti bilo koji izraz koji daje vrednost true (tačno) ili false (netačno). Ako je vrednost izraza true (tačno), onda se izvršava iskaz koji sledi, u suprotnom ne.

Primer:

```
if(broj % 2 != 0)       //proverava se da li je broj neparan
   broj = broj + 1;     //ako je neparan, neka postane paran
```

Ovaj iskaz se može napisati i na neki drugi način, ali se preporučuje ovakva forma.

if(broj %2 != 0) broj = broj + 1; //ovako ne bi trebalo, mada je moguće

Ako je potrebno da se kao rezultat iskaza if izvrši više iskaza, a ne samo jedan kao u prethodnom primeru, mogu se uporebiti vitičaste zagrade.

```
if(izraz){
   iskaz1;
   iskaz2;
   …
}
```

Primer:

```
if(broj % 2 != 0){      //proverava se da li je broj neparan
   broj = broj + 1;     //ako je neparan, neka postane paran
   System.out.println(“Broj je pretvoren u paran i njegova vrednost je
sada “ + broj);
}
```

Ako je izraz ocenjen kao tačan, izvršavaju se svi iskazi između vitičastih zagrada, a ako nije, ne izvršava se nijedan.

Klauzula else

Osnovni iskaz if može da se proširi klauzulom else.

```
if(izraz)
   iskaz1;
else
   iskaz2;
```

Iskaz ili iskazi koji slede iza klauzule else se izvršavaju samo u slučaju da izraz u klauzuli if nije tačan. Ako je potrebno da se u klauzuli else izvrši više iskaza takođe se mogu uporebiti vitičaste zagrade.

```
if(izraz){
   iskaz1;
   …
```

```
}else{
   iskaz2;
   …
}
```

Klauzula else if

Ukoliko prilikom donošenja odluke postoji više mogućnosti, može se upotrebiti klauzula else if.

```
if(izraz1)
   iskaz1;
else if(izraz2)
   iskaz2;
else if(izraz3)
   iskaz3;
else
   iskaz4;
```

Ako je vrednost izraza izraz1 ocenjena kao tačna, izvršava se iskaz1. Ako izraz nema vrednost true, proverava se vrednost izraza izraz2. Ako je njegova vrednost tačna, izvršava se iskaz iskaz2. Posle izvršenja iskaza iskaz2 program se nastavlja kodom koji sledi iza celog iskaza if. Ako vrednost izraza izraz2 nije true, proverava se vrednost izraza izraz3 itd. Ako nijedan od izraza u klauzulama else if nije ocenjen kao tačan (true), izvršava se iskaz (iskazi) iza klauzule else, ako ona postoji.

Da pogledamo kako ovo izgleda na primeru:

```
public class IfElseDemo {
    public static void main(String{} args) {
        int rezultatTesta = 76;
        char ocena;
        if (rezultatTesta >= 90) {
            ocena = 'A';
        } else if (rezultatTesta >= 80) {
            ocena = 'B';
        } else if (rezultatTesta >= 70) {
            ocena = 'C';
        } else if (rezultatTesta >= 60) {
            ocena = 'D';
        } else {
            ocena = 'F';
        }
        System.out.println("Ocena  = " + ocena);
    }
}
```

Izlaz iz programa je:

```
Ocena = C;
```

Kao što vidite vrednost promenljive rezultatTesta može da zadovolji više od jednog uslova, jer je 76 >= 70, a takođe je i rezultatTesta >= 60. Ocena je ipak C, zato što se izvršava samo prvi blok koda koji pripada izrazu rezultatTesta >= 70. Kada se izvrši taj blok, kontrola programa prelazi na kod iza iskaza if, u ovom slučaju na iskaz System.out… Izrazi koji slede iza izraza koji je bio zadovoljen se ne proveravaju, a samim tim se i ne izvršava pripadajući kod.

Iskazi if mogu biti ugnežđeni. U okviru jednog if iskaza može da se nađe drugi, u okviru tog drugog, treći itd. Evo kako to izgleda:

```
if(izraz1){
   if(izraz1-1){
          if(izraz1-1-1)
```

```
                iskaz1-1-1;
   }else
          iskaz1-1;
}
```

Uslovni operator (?)

Pogledajmo iskaz if koji se koristi za izračunavanje apsolutne vrednosti:

```
if ( vrednost < 0 )
  abs = - vrednost;
else
  abs = vrednost;
```

Sledeći iskaz radi isto, ali sa samo jednim iskazom.

```
abs = (value < 0 ) ? -value : value ;
```

U ovom iskazu se koristi uslovni operator. Sa desne strane operatora = je uslovni izraz. Ovaj izraz se ocenjuje, preko uslovnog operatora i na kraju dodeljuje promenljivoj abs. To se može predstaviti na sledeći način:

```
uslov (tačno ili netačno) ? vrednost-ako-je-tačno : vrednost-ako-je-netačno
```

Ovo radi na sledeći način:

1. Uslovni iskaz vraća jednu vrednost.
2. Rezultat može biti jedna od dve vrednosti.
    - Ako je uslov tačan, onda se koristi izraz između ? i :
    - Ako je uslov netačan, koristi se izraz između : i kraja.

Pogledajmo prethodni primer:

```
double vrednost = -34.569;
double abs;

abs = (vrednost < 0 )   ?   - vrednost: vrednost;
      -------------       ------
      1. uslov         2.  rezultat je
         je tačan               +34.569

----
3.   +34.569 se dodeljuje promenljivoj abs
```

Iskaz switch

Ovaj iskaz se koristi za uslovno izvršavanje iskaza na bazi vrednosti celobrojnog izraza. U primeru koji sledi je upotrebljen iskaz switch koji na osnovu celobrojne vrednosti mesec štampa ime meseca, na koji promenljiva ukazuje.

```
public class SwitchDemo {
    public static void main(String{} args) {

        int mesec = 8;
        switch (mesec) {
            case 1:  System.out.println("Januar"); break;
            case 2:  System.out.println("Februar"); break;
            case 3:  System.out.println("Mart"); break;
            case 4:  System.out.println("April"); break;
            case 5:  System.out.println("Maj"); break;
            case 6:  System.out.println("Jun"); break;
```

```
                case 7:  System.out.println("Jul"); break;
                case 8:  System.out.println("Avgust"); break;
                case 9:  System.out.println("Septembar"); break;
                case 10: System.out.println("Oktobar"); break;
                case 11: System.out.println("Novembar"); break;
                case 12: System.out.println("Decembar"); break;
            }
        }
    }
```

Rezultat ovog programa je Avgust. Isti efekat se mogao postići i pomoću iskaza if:

```
int mesec = 8;
if (mesec == 1) {
    System.out.println("Januar");
} else if (month == 2) {
    System.out.println("Februar");
}
…… itd.
```

Stvar je ličnog izbora koji ćete od ovih metoda koristiti. Prilikom donošenja odluke treba imati u vidu iskaz switch odluku može da donese samo na osnovu celobrojne vrednosti, dok se u iskazu if mogu da koriste različiti uslovi.

Verovatno ste u prethodnom kodu primetili prisustvo iskaza break. Ovaj iskaz se najčešće koristi zajedno sa iskazom switch, jer se bez njega kontrola toka programa, nakon izvršenja pravog iskaza, prenosi na sledeći iskaz u switch bloku. Ako se doda iskaz break, onda se kontrola toka, nakon što se izvrši pravi iskaz, prenosi na kod koji sledi iza iskaza switch.

Način kontrole toka programa u iskazu switch (bez iskaza break) može ponekad i da bude od koristi. Evo primera u kome smo to iskoristili:

```
public class SwitchDemo2 {
    public static void main(String{} args) {

        int mesec = 2;
        int godina = 2000;
        int brojDana = 0;

        switch (mesec) {
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12:
                brojDana = 31;
                break;
            case 4:
            case 6:
            case 9:
            case 11:
                brojDana = 30;
                break;
            case 2:
                if ( godina % 4 == 0)
                    brojDana = 29;
                else
                    brojDana = 28;
```

```
                    break;
            }
            System.out.println("Broj dana je " + brojDana);
        }
    }
```

Izlaz iz programa je:

```
Broj dana je 29
```

Ovaj program izračunava broj dana u mesecu za zadatu godinu i mesec. Kao što vidite svi meseci koji imaju 31 dan su bez iskaza break, koji postoji samo kod poslednjeg u nizu. Isto važi i za mesec koji imaju 30 dana (i oni su grupisani i postoji samo jedan iskaz break).

Ako želite da se neki iskazi izvrše u slučaju da nijedan od case iskaza nije zadovoljen, možete da upotrebite klauzulu default. Iskazi koji slede iza ove klauzule se izvršavaju ako nijedan slučaj (case) nije zadovoljen. Evo kako izgleda prepravljen program iz prvog primera, sada sa dodatom klauzulom default:

```
public class SwitchDemo {
    public static void main(String{} args) {

      int mesec = 8;
      switch (mesec) {
        case 1:  System.out.println("Januar"); break;
        case 2:  System.out.println("Februar"); break;
        case 3:  System.out.println("Mart"); break;
        case 4:  System.out.println("April"); break;
        case 5:  System.out.println("Maj"); break;
        case 6:  System.out.println("Jun"); break;
        case 7:  System.out.println("Jul"); break;
        case 8:  System.out.println("Avgust"); break;
        case 9:  System.out.println("Septembar"); break;
        case 10: System.out.println("Oktobar"); break;
        case 11: System.out.println("Novembar"); break;
        case 12: System.out.println("Decembar"); break;
        default: System.out.println("Niste uneli ispravan broj za
mesec");break;

      }
   }
}
```

Iterativne kontrolne strukture

Puno savremenih mašina rade tako što beskonačno ponavljaju isto kretanje. Motor vašeg automobila ponavlja cikluse kretanja, kojima pretvara energiju sagorevanja benzina u mehaničku. Isto važi i za električne motore. I jedna i druga vrsta mašina su korisne zato što što stalno rade isto i to ponavljaju sve dok nam je potrebno.

I računarski programi u svom radu imaju cikluse. Kada program u sebi ima petlju, neki iskazi se ponavljaju sve dok ne obave posao. Većina programa, svaki put kada se izvrši, upotrebi više miliona iskaza. Obično se isti iskazi izvršavaju više puta.

Petlje se u programskim jezicima realizuju pomoću posebnih stuktura. U Javi, a i većini drugih programskih jezika, postoje tri vrste petlji. To su petlje while, do while i for.

Petlja while

Ova petlja se koristi za ponovljeno izvršavanje bloka iskaza sve dok je uslov tačan. Sintaksa while petlje je:

```
while(uslov)
```

```
iskaz;
```

- Uslov je bulov izraz, odnosno nešto čija vrednost može biti tačno ili netačno.
- Uslov može biti vrlo složen, sa relacionim i logičkimoperatorima.
- Iskaz je jedan iskaz. Tu međutim, može stajati i blok iskaza.
- Iskaz se ponekad naziva telom petlje.

Pošto iskaz (telo petlje) može biti jedan iskaz ili blok iskaza to i while petlja ima dva moguća oblika. To se može prikazati na sledeći način:

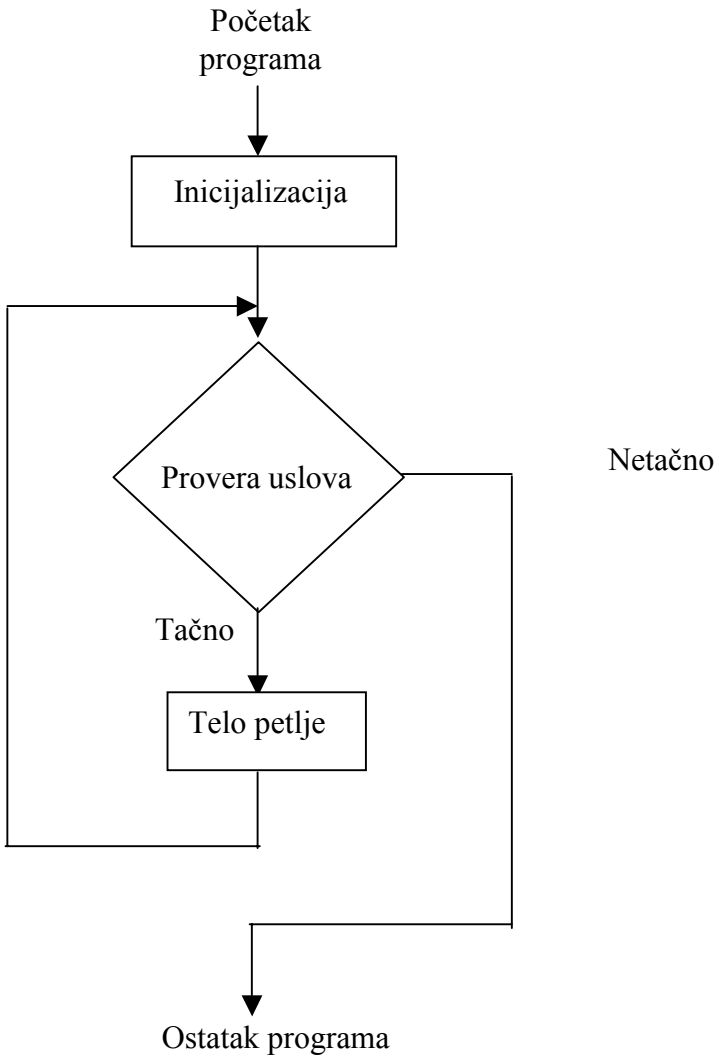
Različiti oblici iskaza while	

`while ( uslov )`
`  iskaz;`
	`while ( iskaz ){`
	`  jedan ili više iskaza`
	`}`

Kako funkcioniše petlja while?

- Sve dok je uslov tačan, izvršava se telo petlje.
- Ako je uslov netačan, telo petlje se preskače i izvršava se iskaz iza petlje, čime se petlja završava
- Ako je kada se prvi put naiđe na petlju uslov netačan, telo petlje se neće nijednom izvršiti.

Ovo se može prikazati sledećim dijagramom:



Pogledajmo sledeći primer:

```
class Petlje{
  public static void main (String[] args )   {
    int brojac = 1;          // brojanje počinje od 1
    while (brojac <= 3 )   // broji se sve dok je brojac <= 3
    {
      System.out.println( "Brojac je:" + brojac);
      brojac = brojac + 1;          // brojac se povecava za 1
    }
    System.out.println( "Iza petlje" );
  }
}
```

Da objasnimo sada kako ovo radi:

1. Promenljivoj brojac se dodeljuje vrednost 1.
2. Izračunava se izraz (brojac <= 3) i njegova vrednost je tačno
3. Pošto je izraz tačan izvršava se blok iza petlje
    - Štampa se trenutna vrednost promenljive brojac
    - brojac se povećava za 1, odnosno dobija vrednost 2.

4. Izračunava se izraz (brojac <= 3) i njegova vrednost je ponovo tačno
5. Ponovo se izvršava blok iza iskaza while.
   - Štampa se vrednost promenlive brojac (2)
   - brojac se povećava za 1, odnosno dobija vrednost 3.
6. Izračunava se izraz (brojac <= 3) i njegova vrednost je ponovo tačno
7. Ponovo se izvršava blok iza iskaza while.
   - Štampa se vrednost promenlive brojac (3)
   - brojac se povećava za 1, odnosno dobija vrednost 4.
8. Izračunava se izraz (brojac <= 3) i njegova vrednost je sada netačno
9. Blok iza iskaza while se više ne izvršava
10. Izvršava se iskaz iza petlje.
    - `System.out.println( " Iza petlje " );`

Kada u svojim programima budete koristili petlje, treba da obratite pažnju na sledeće:

- Inicijalne vrednosti moraju biti podešene na pravi način
- Uslov petlje mora biti tačan
- U telu petlje se moraju menjati vrednosti promenljivih na pravi način. Ovo se odnosi na promenljive koje učestvuju u uslovu petlje. U telu petlje se mora desiti neka akcija koja menja promenljive uslova petlje, tako da taj uslov u nekom trenutku dobije vrednost netačno. Ako se to ne bi desilo, dobili bismo beskonačnu petlju.

Petlja koju smo koristili u prethodnom primeru je koristila promenljivu brojac, koja učestvuje u uslovu petlje i menja se u telu petlje. Ova promenljiva je obično tipa int. Ona ima posebnu ulogu, pa se ponekad naziva kontrolnom promenljivom petlje. Ipak ne moraju sve petlje da imaju kontrolnu promenljivu. Ovakve petlje predstavljaju poseban tip petlji, tzv. petlje sa brojanjem.

Svaki prolaz kroz telo petlje se naziva iteracijom. Kod petlji sa brojanjem u svakoj iteraciji se menja celobrojna kontrolna promenljiva petlje.

Petlja for

Ova vrsta petlje je pogodan način za programiranje petlji sa brojanjem, iako mogu da se koriste i za druge tipove petlji. Sve što se uradi sa ovom petljom može da se uradi i preko petlje while. Iskaz for omogućava da se sve to izrazi na kompaktan, razumljiv način.

Prisetimo se priče o tome da kod petlji treba da postoje incijalne vrednosti, uslov petlje i da se u telu petlje mora menjati promenljiva koja učestvuje u uslovu petlje.

U Javi, kao i u većini savremenih programskih jezika, postoji petlja for, koja kombinuje sva ova tri aspekta petlje. Petlja izgleda ovako:

```
for ( incijalizacija ; uslov_petlje ; izraz_petlje )
  teloPetlje ;
```

Deo označen sa inicijalizacija se izvršava jednom pre pokretanja petlje. Tu se obično inicijalizuje brojač petlje. Izvršavanje petlje se nastavlja sve dok se uslov_petlje izračunava kao tačan. Ovaj izraz se proverava na početku svakog izvršenja petlje. Kada njegova vrednost bude false (netačno), program se nastavlja kodom koji sledi iza petlje. Izraz označen sa izraz_petlje se izvršava na kraju petlje. Obično se tu povećava ili smanjuje brojač petlje za neku zadatu vrednost.

Pogledajmo to na primeru:

```
//   incijalizacija;uslov_petlje; promena (izraz_petlje)
//
for ( count = 0;  count < 10; count++ )
  System.out.print( count + " " );
```

Sve što se može izraziti for petljom može i while petljom i obrnuto. To se može prikazati na sledeći način:

for petlja	<==>	while petlja

`for (incijalizacija;uslov_petlje;izraz_petlje)`
`  teloPetlje ;`
	.	`incijalizacija;`
		`while(uslov_petlje){`
		`  teloPetlje;`
		`  izraz_petlje`
		`}`

Evo uporednog primera, u kome smo isti problem rešili preko for i while petlje.

for petlja	while petlja

`int count, sum;`
 	
`sum = 0;`
`for(count=0;count<=5;count=count+1){`
`  sum = sum + count ;`
`  System.out.print( brojac + " " );`
`}`
`System.out.println( "suma je: " +`
`sum );`
	`int count, sum;`
	`sum   = 0;`
	`count = 0;`
	`while ( count <= 5 ){`
	`  sum = sum + count ;`
	`  System.out.print( brojac + " " );`
	`  count=count+1;`
	`}`
	`System.out.println( "suma je: " +`
	`sum );`

Ova dva programa su ekvivalentna. Na primeru while petlje se može bolje uočiti kako funkcioniše for petlja. Brojač (count) se podešava na početku rada petlje. Inicijalizacija se odvija samo jednom. Uslov petlje se stalno ocenjuje, da bi se videlo da li telo petlje treba da se ponovo izvrši. Izraz petlje se izračunava uvek na kraju tela petlje.

Iz for petlje se mogu izostaviti pojedini delovi. Ako se izostavi deo inicijalizacije, for petlja se ponaša slično petlji while. Izostavljanje inicijalizacije je korisno kada je ta inicijalizacija suviše komplikovana i kada to želite da uradite u nekoliko iskaza pre same petlje. Možda inicijalizacija i zavisi od onog što korisnik unese:

```
// ovde se od korisnika dobija inicijalna vrednost
for (    ; count < 13; count++  ) {
  System.out.println( "Brojac je: " + count );
}
System.out.println( "\nKraj petlje.\nBrojac je sada" + count);
```

Ako želite možete da izostavite i izraz_petlje. Ako Java kompajler naiđe na sledeći kod:

```
for ( count = 0; count < 25;  )
```

on se neće žaliti. U ovom slučaju je na programeru da obezbedi iskaze koji menjaju promenljivu u uslovu petlje. Mogli bismo da napišemo sledeću petlju:

```
for ( brojac = 0; brojac < 25;  ){
  System.out.println("Brojac je: " + brojac);
  brojac = brojac + 1;
}
```

koja bi radila kako treba. (Ipak je bolje ovo staviti u izraz_petlje).

Možete izostaviti čak i uslov petlje, ali u tom slučaju dobijate petlju:

```
for ( incijalizacija ;  tačno  ; izraz_petlje )
  teloPetlje ;
```

što znači da se petlja beskonačno izvršava. Ovo nikad ne bi trebalo koristiti.

Domen kontrolne promenljive petlje

Već smo pomenuli da se for petlje koriste najviše kod petlji sa brojanjem, kod kojih postoji kontrolna promenljiva, na osnovu koje se određuje da li petlja treba da se ponovo izvršava. Domen ove promenljive su granice petlje u kojoj je promenljiva deklarisana. Ovo naravno važi ako je ta promenljiva deklarisana u delu inicijalizacije petlje.

Pogledajmo sledeći primer:

```
int brojac;

. . . . . .

for (brojac = 0;  brojac < 10; brojac ++ )
  System.out.print(brojac + " " );
```

Kontrolna promenljiva petlje brojac je deklarisana negde u programu (možda i daleko ispred petlje). Ovim se narušava ideja o kontroli petlje u samo jednom iskazu. To se i može uraditi na sledeći način:

```
for ( int brojac = 0;  brojac < 10; brojac ++ )
  System.out.print(brojac + " " );
```

Ovaj drugi način je bolji i treba ga koristiti kad god je moguće (skoro uvek).

Ako u delu inicijalizacije for petlje deklarišete promenljivu petlje, onda je sledeće neispravno:

```
for ( int brojac = 0;  brojac < 10; brojac ++ )
  System.out.print(brojac + " " );

// Nije deo tela petlje
System.out.println( "\nPosle petlje brojac je:" + brojac);
```

Pošto iskaz println() nije deo tela petlje u njemu se ne može koristiti promenljiva brojac. Kompajler će javiti grešku tipa "cannot resolve symbol", što znači da brojac nije dostupan.

Realni brojevi kao brojači petlje

Kod petlji sa brojanjem je najbolje da se kao kontrolna promenljiva petlje koristi celobrojna promenljiva. Ipak, ako želite, možete da koristite i realan broj, koji biste u svakom prolazu povećavali opet za neki realan broj. U sledećem primeru smo pokazali primer koji štampa x i ln(x) za vrednosti x od 0.1 do 2.0.

```
class Logaritam{
  public static void main ( String[] args )  {
    System.out.println( "x" + "\t ln(x)" );

    for ( double x = 0.1; x <= 2.0; x = x + 0.1 )
      System.out.println( x + "\t" + Math.log( x ) );
  }
}
```

Ovaj program će se iskompajlirati i radiće svoj posao, ali ne baš na najbolji način. Izlaz iz njega je:

```
x                        ln(x)
0.1                     -2.3025850929940455
0.2                     -1.6094379124341003
0.30000000000000004     -1.203972804325936
0.4                     -0.916290731874155
0.5                     -0.6931471805599453
0.6                     -0.5108256237659907
0.7                     -0.35667494393873245
0.7999999999999999      -0.22314355131420985
0.8999999999999999      -0.1053605156578264
0.9999999999999999      -1.1102230246251565E-16
```

```
1.0999999999999999      0.09531017980432474
1.2                     0.1823215567939546
1.3                     0.26236426446749106
1.4000000000000001      0.336472236621213
1.5000000000000002      0.40546510810816455
1.6000000000000003      0.47000362924573574
1.7000000000000004      0.5306282510621706
1.8000000000000005      0.5877866649021193
1.9000000000000006      0.641853886172395
```

Realni brojevi nisu apsolutno tačni. Konkretno broj 0.1 je u računaru uvek predstavljen sa malim odstupanjem, bez obzira na to koliko bitova koristite. Greške u x će se akumulirati, tako da x sve više odstupa od željene vrednosti.

Obratite pažnju na poslednju odštampanu vrednost za x. Ona nije 2, kao što biste očekivali.

Petlja do while

Ovo je treći način implementacije petlji u Javi. U pitanju je petlja koja se verovatno najmanje upotrebljava.

Opšti oblik ove petlje je:

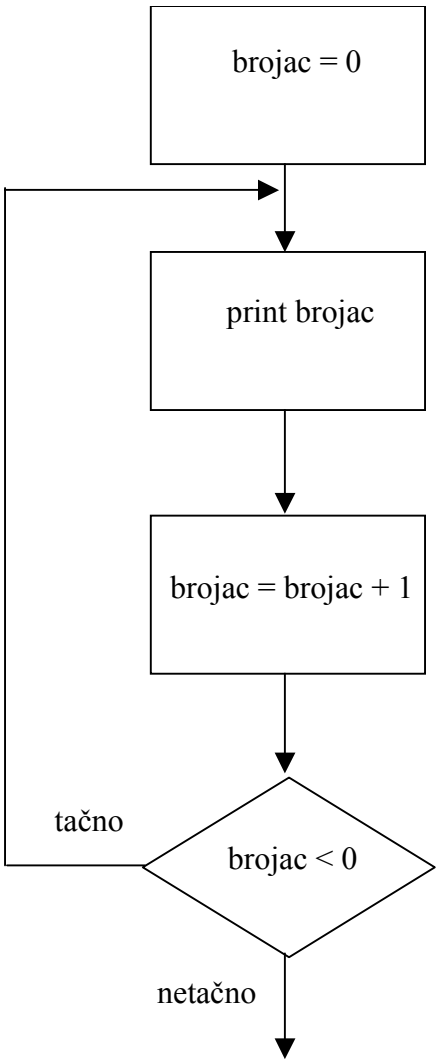
```
do{
   iskazi …
}while(uslov_petlje)
```

Iskaz do while je sličan sa petljom while, ali postoji jedna bitna razlika. U ovoj petlji se provera uslova petlje obavlja posle izvršenja tela petlje. U sledećem primeru smo dali petlju koja štampa cele brojeve od 1 do 9.

```
int brojac = 0;                       // brojac se incijalizuje na 0
do{
  System.out.println(brojac);       // telo petlje: sadrži i kod za
  brojac = brojac + 1  ;                          // promenu brojaca
}while (brojac < 10 );               // proverava se da li telo petlje
                                     // treba ponovo da se izvrši
```

Obratite pažnju na do i while. Uslov se testira posle iskaza while.

Šematski se prethodni program može predstaviti na sledeći način:



Najbitnije što treba imati na umu kod do while petlji jeste da se telo petlje izvršava makar jednom. Tek nakon prvog izvršenja se proverava uslov petlje.

Iskazi continue i break kao delovi petlje

Ako želite da se u nekom prolazu kroz petlju preskoči neki njen deo i da se izvršenje nastavi od početka petlje, možete upotrebiti iskaz continue. Pogledajmo sledeći primer:

```
for(int i = 1; i <= limit; i++){
   if(i % 3 == 0)
          continue;     // preskace se ostatak iteracije
   sum += i;
}
```

Iskaz break se može upotrebiti za trenutni prekid petlje. Program se iskazom break nastavlja kodom koji sledi iza petlje. Sledi primer u kome se pronalaze prosti brojevi. Prost broj je onaj koji nije deljiv bez ostatka ni sa jednim drugim brojem (osim sa 1).

```
public class ProstiBrojevi{
  public static void main (String{} args){
    int nVrednost = 50; // maksimalna vrednost koja se proverava
    boolean prostBroj = true; // tacno ako je prost broj
  // provera svih vrednosti od 2 do nVrednost
    for(int i = 2; i <= nVrednost; i++){
```

```
            prostBroj = true; // pretpostavka da je trenutna vrednost
                              //prost broj
          for(int j = 2; j < i; j++){
            if(i % j == 0){ //tacno ako j deli bez ostatka
              prostBroj = false; // ako smo dospeli do ovde, postojalo je
                    //deljenje bez ostatka, sto znaci da nije prost broj
              break;
            }
          }
// ispisivanje prostih brojeva
          if(prostBroj) // da li je to prost broj
            System.out.println(i);  // u pitanju je prost broj pa se
                                    //ispisuje
      }
    }
}
```

Ispisuju se prosti brojevi od 1 do nVrednost (u ovom slučaju 50). Spoljašnja petlja ide kroz sve brojeve koje treba proveriti. Brojač te petlje je i. Za svako i, unutrašnja petlja proverava da li postoji neki broj sa kojim se to i može da podeli bez ostatka. Ako može, to znači da nije u pitanju prost broj i da treba napustiti unutrašnju petlju (za to služi iskaz break).

Na kraju se svi pronađeni prosti brojevi štampaju.

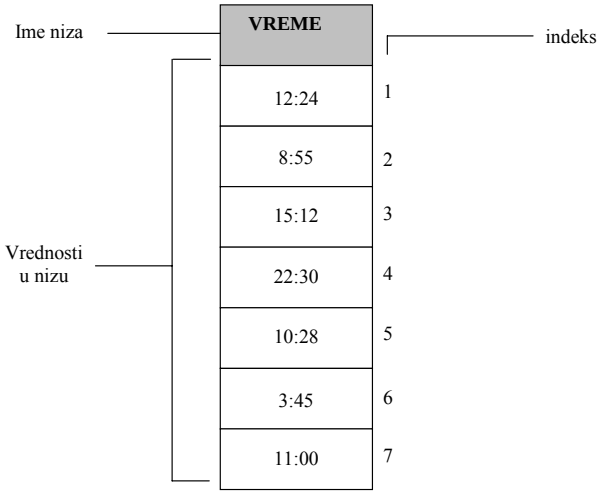
Gneždenje petlji

Petlje mogu da se ugneždavaju. To znači da unutar tela neke petlje može da postoji druga petlja, pa unutar te druge treća itd. Kod ovakvih petlji pravilo je da se uvek prvo izvršava krajnja unutrašnja petlja, pa onda ona u kojoj se ova nalazi i tako redom.

Kod gneždenja nije bitno koji su tipovi petlji. To znači da možete proizvoljno kombinovati petlje while, for i do while.

Nizovi

Niz je istorodan skup elemenata, što znači da su svi elementi iste vrste. Najjednostavniji niz je linearan. Takvi nizovi imaju samo jednu dimenziju i nazivaju se jednodimenzionalnim nizovima. Na sledećoj slici je dat primer jednog takvog niza.



Niz Vreme bi se mogao koristiti za predstavljanje vremena kada se tokom nedelje desi neki značajan događaj. Niz se ponekad naziva indeksiranom promenljivom. Razlozi su što kao i promenljiva niz može da sadrži vrednosti i što se njegovim vrednostima pristupa preko indeksa.

U matematici su se indeksi pisali potpisano, na primer $Vreme_1$, $Vreme_2$ itd.

U većini programskih jezika se indeksi pišu u okviru uglastih zagrada:

Vreme[1], Vreme [2] itd.

Vreme[1], na prethodnoj slici ukazuje na 12:24. Generalno se za indekse mogu koristiti imena koja imaju svoje značenje u kontekstu aplikacije. Na primer, prethodni iskaz bismo mogli zameniti iskazom:

Vreme[ponedeljak]

Na sledećoj slici je prikazan još jedan jednodimenzionalni niz, koji sadrži temperature pacijenta, koje su zapisivane svakog sata tokom dana. Na primer,

Temperatura[3] ukazuje na temperaturu zapisanu trećeg sata.

Sat	Temperatura
1	36
2	37
3	38
23	37
24	37

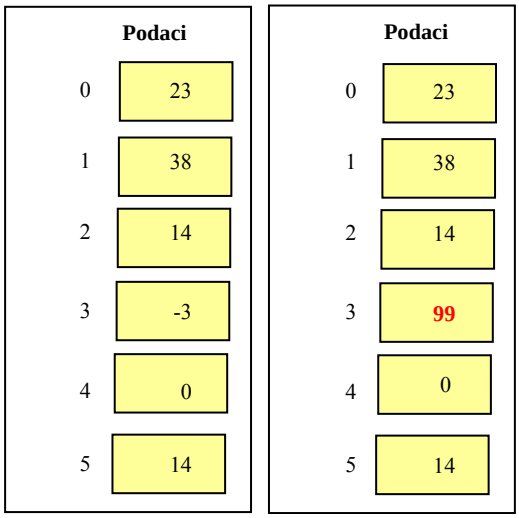
Nizovi u Javi

Sve što je rečeno za nizove uopšte važi za nizove u programskom jeziku Java. Niz je skup elemenata istog tipa. Ti elementi mogu biti primitivnog tipa, kao što je int, double i sl., ali elementi mogu biti i reference na objekte.

Elementima niza se kao što je već pomenuto pristupa preko iskaza:

a[3]=99;

U ovom slučaju je a jedan niz celih brojeva. Prethodnim iskazom smo četvrtom elementu niza dodelili vrednost 30. Obratite pažnju na to da je vrednost indeksa 3, a da se ipak pristupa četvrtom elementu niza. Razlog je što brojanje elemenata u Javi, kao i jezicima C i C++ počinje od 0.



Nakon što se izvrši prethodni iskaz vrednost u četvrtom slotu niza je promenjena i iznosi 99.

Iskaz za pristupanje elementu niza, kao što je a[3], može se koristiti u svim izrazima u kojima se može koristiti promenljiva tipa, koji čine elementi niza. Na primer, ako je vrednost promenljive x 1, onda izraz:

(x + a[3]) / 4

daje vrednost 25.

Nizovi kao objekti

Nizovi se u Javi deklarišu na sledeći način:

```
tip[] imeNiza;
```

Ovim se kompajleru govori da će imeNiza biti upotrebljeno kao ime niza u kome će se nalaziti elementi tipa tip. Ovo, međutim, predstavlja samo deklaraciju niza. Niz još uvek nije napravljen. Ovim se samo deklariše promenljiva, koja će nekad u budućnosti ukazivati na niz.

Vrlo često se dešava da se niz u jednom koraku i deklariše i konstruiše. To se radi na sledeći način:

```
tip[] imeNiza = new tip[duzina];
```

Ovaj iskaz istovremeno radi dve stvari:

1. Govori kompajleru da će imeNiza ukazivati na niz sa elementima određenog tipa (tip)
2. Konstruiše objekat niza, koji će sadržati zadati broj slotova (duzina).

Niz je objekat, kao i bilo koji drugi objekat u Javi. Kao i bilo koji drugi objekat i on, tokom rada programa, zauzima svoj deo glavne memorije. Razlika je samo u sintaksi koja se koristi kod pravljenja novog niza. Konstruktor niza ima sledeću sintaksu:

```
new tip[ duzina ]
```

Ovim se definiše koji tip podataka će se u nizu nalaziti, a takođe i veličina niza. Nakon što se niz jednom napravi, njegova veličina, odnosno maksimalan broj elemenata koje može da primi, se više ne može menjati. Na primer iskaz:

```
int[] dt = new int[10];
```

kreira niz dt, a istovremeno svaki njegov element dobija vrednost 0.

Dužina nekog niza pokazuje koliko elemenata on može da sadrži. To znači da niz dužine N ima N elemenata, kojima se pristupa preko indeksa od 0 do N-1.

Indeksi moraju biti celobrojnog tipa. Sintaktički nije bitno da li između broja i zagrade postoji razmak, tako da je dt[1] i dt[1] potpuno isto. Ne sme se pokušavati pristup elementu koji ne postoji. Ako smo na primer, deklarisali niz:

```
int[] dt = new int[10];
```

Onda važe sledeća pravila:

dt[-1]	uvek neispravno
dt[10]	neispravno (zbog deklaracije)
dt[1.5]	uvek neispravno
dt[0]	uvek u redu
dt[9]	U redu (zbog deklaracije)

Ako u programu postoji izraz koji je uvek neispravan, program se neće iskompajlirati. Sa druge strane, najčešće je veličina niza nepoznata u trenutku kompajliranja. Ona se u tom slučaju određuje u vreme izvršenja programa. Pošto se niz pravi u vreme izvršenja programa, kompajler ne zna njegovu dužinu i ne može da uoči sve greške. Ako u takvim slučajevima, program pristupi elementu koji ne postoji, javlja se greška i program se prekida.

Ako prilikom konstruisanja nema nikakvih drugih informacija o vrednostima elementa niza, onda se svaki element niza inicijalizuje na podrazumevanu vrednost u skladu sa tipom elementa. Ako su u pitanju celobrojne vrednosti, onda se svi elementi inicijalizuju na 0.

U programu, naravno, možete elementima dodeliti druge vrednosti. Pogledajmo sledeći primer:

```
class NizPrimer1{
  public static void main ( String[] args )  {
    int[] st = new int[5];
    st [0] = 23;
    st [1] = 38;
    st [2] = 7*2;
    System.out.println("st[0] je " + st [0] );
    System.out.println("st [1] je " + st [1] );
    System.out.println("st [2] je " + st [2] );
    System.out.println("st [3] je " + st [3] );
    System.out.println("st [4] je " + st [4] );
  }
}
```

Prethodni program bi trebalo da odštampa sledeće;

```
st[0] je 23
st [1] je 38
st [2] je 14
st [3] je 0
st [4] je 0
```

Indeks niza je uvek ceo broj. Ipak to ne mora biti broj, već može biti i promenljiva tog tipa ili izraz čija je vrednost ceo broj. Sledeći iskazi su, na primer, ispravni:

```
int vrednosti[] = new int[7];
int indeks;

indeks = 0;
vrednosti [indeks] = 71;       // prvi element dobija vrednost 71

indeks = 5;
vrednosti [indeks] = 23;       // šesti element dobija vrednost  23

indeks = 3;
vrednosti[2+2]=vrednosti[indeks -3 ];//isto kao
                                     //vrednosti[4]=vrednosti[0];
```

Inicijalizacija niza

U istom iskazu se može izvršiti deklarisanje, konstruisanje i inicijalizacija niza.

```
int[] dt = {23, 38, 14, -3, 0, 14, 9, 103, 0, -56 };
```

Ovim se deklariše niz celobrojnih vrednosti, po imenu dt. Nakon toga se konstruiše niz od 10 elemenata i na kraju se zadate vrednosti dodeljuju tim elementima. Prva vrednost iz liste za inicijalizaciju odgovara indeksu 0, druga indeksu 1 itd. (U ovom primeru dt[0] dobija vrednost 23.)

U ovom iskazu ne morate reći koliko elemenata će niz imati. Kompajler će izbrojati vrednosti u listi i napraviti toliko elemenata. Ovakve liste za inicijalizaciju se obično koriste za male nizove.