4

Petlje

- **`for`** petlja
- **`while`** petlja
- **`do/while`** petlja

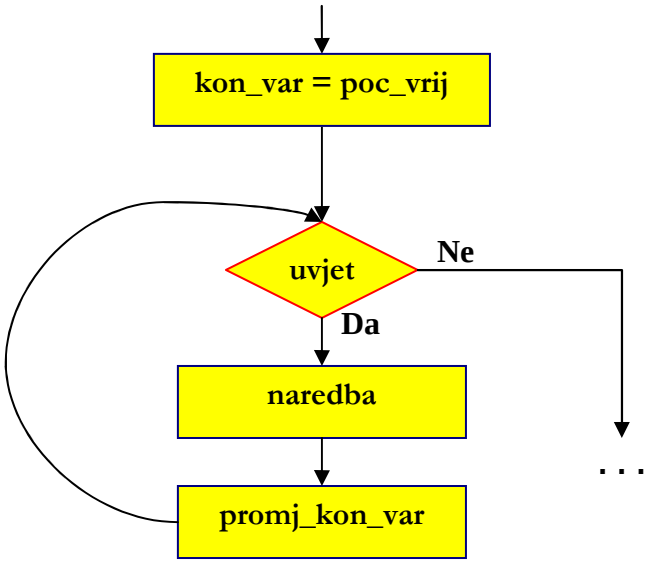
`for` petlja

Petlje ćemo općenito koristiti kada neki dio programa, niz naredbi, trebamo ponoviti više puta. Općenito **`for`** petlju koristimo kada unaprijed znamo koliko puta trebamo izvršiti neki dio programa. U Javi **`for`** petlja ima nešto općenitiji oblik i možemo ju koristiti i u nekim slučajevima kada ne znamo unaprijed koliko puta trebamo izvršiti neki dio programa.

Opći oblik **`for`** petlje je:

```
for (kon_var = poc_vrij; uvjet; promj_kon_var)
     naredba;
```

ili dijagramom toka bi to izgledalo ovako:



Slika 4 – 1 Shematski prikaz **`for`** petlje

pri čemu je:

- **`kon_var`** – kontrolna varijabla;
- **`uvjet`** – najčešće je oblika **`kon_var >(<, >=, <=, ==, !=) vrijednost`** – uvjet izvršavanja **`for`** petlje;
- **`promj_kon_var`** – promjena vrijednosti kontrolne varijable **`kon_var`**. Najčešće je oblika `kon_var++` odnosno `kon_var--`.

Izvršavanje **`for`** petlje:

- na samom početku se kontrolnoj varijabli (`kon_var`) pridruži neka početna vrijednost (`poc_vrij`)
- ako je `uvjet` istinit:
  1. izvršava se `naredba`
  2. mijenja se vrijednost kontrolne varijable (`promj_kon_var`)

3. vraćamo se na početak **for** petlje i provjeravamo uvjet

Napomena:
Ukoliko je **naredba** složena naredba trebamo ju pisati unutar {}.
Trebamo biti jako oprezni prilikom promjene vrijednosti kontrolne varijable (prom_kon_var), trebamo osigurati da nakon konačnog broja koraka kontrolna varijabla (kon_var) poprimi takvu vrijednost koja više neće zadovoljavati uvjet. Ukoliko to ne osiguramo ući ćemo u **beskonačnu petlju** i morat ćemo program prekinuti nasilno.

Primjer 4 – 1:
Koliko će se puta izvršiti sljedeći dio programa:

```
...
int n = 5, t = 0;
for (int i = 0; i <= n; i++)
      t++;
...
```

Rješenje:
- pri ulasku u **for** petlju varijabli i će se pridružiti vrijednost 0 (**i = 0**). Budući da je **i** (0) uistinu **<=** od **n** (5) izvršit će se naredba (**t++**) te će se vrijednost varijable **i** povećati za 1 (**i++**) i iznosit će 1;
- budući da je **i** (2) **<= n** (5) izvršit će se naredba (**t++**) te će se vrijednost varijable **i** povećati za 1 (**i++**);
- **i** (2) je i dalje manje ili jednako od **n** (5) – ponovo se izvršava naredba **t++** te vrijednost varijable **i** postaje 3;
- kako je **i** (3) **<= n** (5) izvršava se **t++**, a **i** postaje 4;
- **i** (4) **<= n** (5), dakle, izvršava se **t++** te **i** poprima vrijednost 5;
- **i** (5) **<= n** (5), izvršava se **t++**, a **i** postaje 6;
- budući da **i** (6) nije **<= n** (5) izlazimo iz **for** petlje i idemo na prvu sljedeću naredbu iza **for** petlje.

Prebrojimo li koliko smo puta izvršili naredbu **t++**, dobit ćemo traženi broj izvršavanja **for** petlje. U ovom slučaju je taj broj 6.

Primjer 4 – 2:
Napišimo metodu čiji će ulazni parametar biti prirodan broj *n*. Metoda treba vraćati ukupni broj djelitelja broja *n*.

Rješenje:

```
public static int pr42 (int n)
{
      int t = 0;
      for (int i = 1; i <= n; i++)
            if (n % i == 0)
                  t++;
      return t;
}
```

U oba prethodna primjera smo vrijednost kontrolne varijable povećavali za 1. U sljedećem ćemo primjeru vrijednost kontrolne varijable smanjivati za 1.

Primjer 4 – 3:

Napišimo metodu čiji će ulazni parametar biti dva prirodna broja *n* i *m*. Metoda treba vraćati najmanji zajednički višekratnik brojeva *n* i *m*.

Rješenje:
Prisjetimo se, najmanji zajednički višekratnik dvaju prirodnih brojeva (*n* i *m*) je najmanji prirodan broj koji se može podijeliti s *n* i s *m*. Sigurno je da je jedan od višekratnika brojeva *n* i *m* njihov umnožak (*n* * *m*), no možda postoji i manji višekratnik od njega. Stoga ćemo pomoću **for** petlje ići od broja *n* * *m* do nekog od brojeva *n* ili *m* (ili do 1), te ćemo vrijednost kontrolne varijable (*i*), kada bude djeljiva s *n* i s *m* zapisati u varijablu *v*. Kako u **for** petlji idemo od većeg broja prema manjem, na kraju će u varijabli *v* pisati najmanji broj koji se može podijeliti s *n* i s *m*.

```
public static int pr43 (int n, int m)
{
      int v = n * m;
      for (int i = n * m; i >= n; i--)
            if (i % n == 0 && i % m == 0)
                  v = i;
      return v;
}
```

Do sada smo u svim primjerima imali po jednu **for** petlju. Često ćemo nailaziti na probleme gdje će unutar jedne petlje biti imati još jednu petlju, pa ćemo za takve petlje reći da su **ugniježđene**.
Ilustrirajmo to na sljedećem primjeru:

Primjer 4 – 4:
Za prirodan broj *n* ćemo reći da je prost ako je djeljiv samo s 1 i sa samim sobom. Prvih nekoliko prostih brojeva su: 2, 3, 5, 7, 11, 13,... (broj 1 nije prost broj). Poznato je da prostih brojeva ima beskonačno mnogo, no ne postoji nikakva egzaktna formula koja će nam reći koji je npr. *n*-ti prosti broj ili koliko ima prostih brojeva koji su manji od prirodnog broja *n*.
Napiši metodu čiji će ulazni parametar biti prirodan broj *m*, a vraćati će broj prostih brojeva koji su manji ili jednaki od *m*.

Rješenje:
Jedan od načina na koji možemo provjeriti je li neki broj prost je da mu prebrojimo sve djelitelje. Ukoliko broj ima samo 2 djelitelja, bit će prost. Ovom ćemo metodom za velike brojeve imati jako puno dijeljenja, što može znatno usporiti izvršavanje programa. Jedan od teorema iz matematike kaže da će broj biti prost ako nema niti jednog djelitelja između dva i korijena iz broja za koji provjeravamo je li prost. Npr. želimo li provjeriti je li broj 113 prost, nije nužno dijeliti ga sa svim brojevima između 1 i 113, već će biti dovoljno provjeriti postoji li djelitelj broja 113 između 2 i $\sqrt{113} \approx 11$.
U ovom primjeru ćemo očito morati proći po svim brojevima koji između 2 i *m*, te ćemo za svaki broj provjeravati je li prost i, u slučaju da je prost povećavati vrijednost neke varijable za 1.
Dakle, očito ćemo imati ugniježđene petlje. Kontrolna varijabla (*i*) prve **for** petlje ići će po brojevima od 2 do *m*. Druga **for** petlja će za svaki takav broj (*i*) provjeriti je li prost i to će raditi tako da prebroji koliko broj *i* ima djelitelja između 2 i $\sqrt{i}$.

```
public static int pr44 (int n)
{
      int p = 0;
      for (int i = 2; i <= n; i++)
      {
            int d = 0;
```

```
            for (int j = 2; j <= Math.sqrt (i); j++)
                  if (i % j == 0)
                        d++;
            if (d == 0)
                  p++;
      }
      return p;
}
```

Rješenje prošlog primjera je ispravno. Međutim kod takvog rješenja je dosta velika mogućnost pogreške: zamijenimo kontrolne varijable (*i*, *j*), zaboravimo inicijalizirati brojač na 0 (*d*),...
Stoga ćemo u nastavku dati još jedno rješenje problema iz primjera 4 – 4. Ovo drugo rješenje temelji se na pisanju još jedne pomoćne metode. Pomoćna metoda, nazovimo ju *prost*, imat će jedan parametar, prirodan broj *n* te će vraćati *true* ako je broj *n* prost, inače će vraćati *false*. Metodu prost ćemo pozivati u svojoj metodi te ćemo na taj način izbjeći ulančane petlje.
Općenito ćemo u metodi pozvati drugu metodu tako da joj napišemo ime i proslijedimo odgovarajuće parametre.

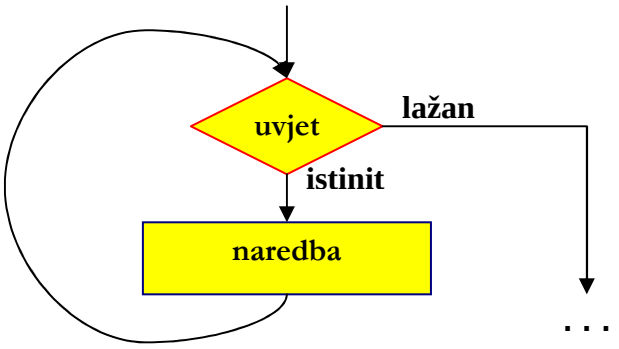
```
public static boolean prost (int n)
{
      int d = 0;
      for (int i = 2; i <= Math.sqrt (n); i++)
            if (n % i == 0)
                  d++;
      if (d == 0)
            return true;
      else
            return false;
}

public static int pr44a (int n)
{
      int p = 0;
      for (int i = 2; i <= n; i++)
      {
            if (prost (i))
                  p++;
      }
      return p;
}
```

Primijetimo da smo u obje metode koristili *i* kao varijablu. Možda nije sasvim jasno kako se ne "pomiješaju" varijable *i* iz tih dviju metoda. Radi se o tome da je varijabla *i* lokalna varijabla i u jednoj i u drugoj metodi. Lokalnu varijablu jedne metode vidi samo ta metoda. Dakle, metoda *pr44a* uopće ne vidi varijablu *i* metode *prost*.
Isto tako bi zbuniti mogao i parametar metode *prost*. Pri definiciji metode *prost* parametar je *n*, a u metodi *pr44a* metodu *prost* pozivamo s parametrom *i*. Pojednostavljeno objašnjenje ovoga je činjenica da je *n* tzv. formalni parametar metode *prost*. Prilikom poziva metode *prost* u varijabli *i* piše konkretan broj i taj broj se pri pozivu metode *prost* prepisuje u formalni parametar *n*, s kojim onda radi metoda *prost*.

`while` petlja

`while` petlju ćemo, slično kao i **`for`** petlju, koristiti u slučajevima kada dio programa trebamo izvršiti više puta.
Za razliku od **`for`** petlje, koju koristimo uglavnom kada znamo koliko puta trebamo ponoviti neki niz naredbi, **`while`** petlju ćemo koristiti kada ne znamo unaprijed koliko puta trebamo izvršiti neki niz naredbi.
Opći oblik **`while`** petlje je:

```
while (uvjet)
      naredba;
```



Slika 4 – 2: Shematski prikaz **`while`** petlje

Dok je **`uvjet`** istinit izvršavat će se **`naredba`**.
Ukoliko je **`naredba`** složena naredba potrebno ju je staviti unutar {}.
Slično kao i kod **`for`** petlje, unutar naredbe koja se nalazi u tijelu petlje trebamo osigurati da nakon konačnog broja koraka **`uvjet`** postane lažan. U suprotnom ćemo uči u beskonačnu petlju.

Primjer 4 – 5:
Što će pisati u varijabli *s* nakon izvršavanja sljedećeg dijela programa?

```
...
int s = 0, i = 1, n = 4;
while (s <= n)
{
      s += i;
      i++;
}
...
```

Rješenje:
Na početku programa očito varijabli **`s`** pridružujemo vrijednost 0, varijabli **`i`** pridružujemo 1, dok varijabli **`n`** pridružujemo vrijednost 4.
- budući da je **`s`** (0) **`<=`** od **`n`** (4) – **`s`** se povećava za 1 te sada iznosi 1, te se **`i`** povećava za 1 i sada iznosi 2;
- s obzirom da je **`s`** (1) **`<=`** **`n`** (4) – **`s`** se povećava za **`i`** (2) te sada poprima vrijednost 3, dok **`i`** postaje 3;
- **`s`** (3) **`<=`** **`n`** (4) – dakle, **`s`** se povećava za **`i`** (3) i postaje 6, a **`i`** postaje 4;

- **s** (6) više nije manje ili jednako od **n** (4), što znači da izlazimo iz **while** petlje
Dakle, nakon petlje će u varijabli s pisati vrijednost **6**.

Primjer 4 – 6:
Napišimo metodu čiji će ulazni parametar biti dva prirodna broja *n* i *m*. Metoda treba računati mjeru brojeva *n* i *m* (mjera brojeva *n* i *m* je najveći prirodan broj koji dijeli broj *n* i broj *m*).

Rješenje:
Primjer ćemo riješiti modificiranom metodom **Euklidovog algoritma** za mjeru dvaju brojeva. Ovakav modificirani Euklidov algoritam glasi ovako:
- ako su **n** i **m** jednaki, onda je njihova mjera jednaka bilo kojem od njih (jednaki su);
- ukoliko brojevi **n** i **m** nisu jednaki, njihova mjera ista je kao mjera brojeva:
    - manji od brojeva **n** i **m**;
    - razlike većeg i manjeg od brojeva **n** i **m**;

Npr. ako je **n = 27** a **m = 18**, Euklidov algoritam bi mogao ići ovako:
- kako *27* i *18* nisu jednaki slijedi da je njihova mjera jednaka mjeri brojeva *18* (manji od brojeva *27* i *18*) i *9* (razlika brojeva *27* i *18*);
- budući da *18* i *9* nisu jednaki, prema Euklidovom algoritmu je njihova mjera jednaka mjeri brojeva *9* (manji od brojeva *18* i *9*) i *9* (razlika brojeva *18* i *9*);
- kako su brojevi *9* i *9* jednaki, dakle njihova mjera je *9*, pa je onda i mjera brojeva *27* i *18* isto tako *9*.

Na osnovu izrečenog lako možemo zaključiti da dok brojevi ne postanu jednaki uspoređujemo ih te na mjesto većeg zapisujemo razliku većeg i manjeg broja, dok manji broj ostaje nepromijenjen:

```
public static int pr46 (int n, int m)
{
      while (n != m)
            if (n > m)
                  n = n - m;
            else
                  m = m - n;
      return n;
}
```

Slično kao što možemo imati ulančane **for** petlje, možemo imati i ugniježđene **while** petlje, pa čak i više, možemo ulančavati **for** petlju s **while** petljom i obrnuto.

Primjer 4 – 7:
Za prirodan broj *n* ćemo reći da je **simetričan** ako se jednako čita s obje strane (npr. 121, 56765,...). Napiši metodu čiji će ulazni parametar biti prirodan broj *n*. Metoda treba vraćati *n*-ti po redu simetrični broj.

Rješenje:
Prvih nekoliko simetričnih brojeva su: 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 22,...
Da bismo za neki broj provjerili je li simetričan trebamo ga okrenuti i provjeriti je li tako okrenut isti kao polazni broj.
Broj *k* ćemo okrenuti tako da mu "skidamo" jednu po jednu znamenku i dodajemo na kraj drugog broja (*m*). Pod "skidanjem" znamenke podrazumijevamo da posljednju znamenku spremimo u privremenu varijablu *z* (`z = k % 10`), te broju *k* "prekrižimo" posljednju znamenku (`k = k / 10`). Znamenku *z* ćemo dodati na kraj broja *m* tako da broj *m* pomnožimo s 10 i dodamo *z* (`m = m * 10 + z`). Znamenke broju *k* ćemo "skidati" sve dok *k* ne postane 0 (primijetimo da smo na taj način zauvijek izgubili početnu vrijednost broja *k*, stoga ju na samom početku trebamo pospremiti u neku pomoćnu varijablu *t*).

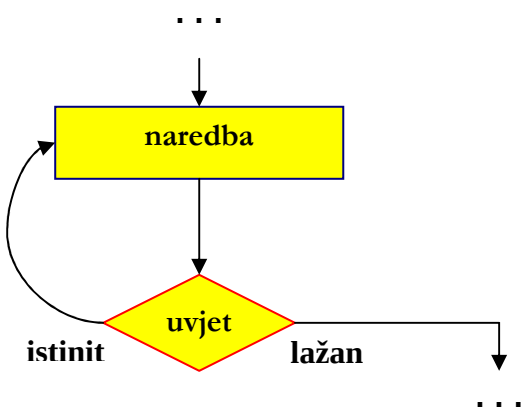
Budući da u zadatku trebamo pronaći *n*-ti po redu simetrični broj, varijabli *k* ćemo pridruživati vrijednosti 1, 2, 3,... i svaki puta kada se u varijabli *k* bude nalazio broj koji je simetričan vrijednost brojača (*b*) ćemo povećati za 1. Stat ćemo kada vrijednost brojača *b* bude bila jednaka *n*, te ćemo vratiti broj koji u tom trenutku piše u *k*.

```
public static int pr47 (int n)
{
	int k = 0, b = 0, z, t, m;
	while (b < n)
	{
		k++;
		t = k;
		m = 0;
		//okrećemo broj t
		while (t > 0)
		{
			z = t % 10;
			t /= 10;
			m = m * 10 + z;
		}
		if (m == k)
			b++;
	}
	return k;
}
```

`do/while` petlja

`do/while` petlja gotovo je identična **`while`** petlji o kojoj smo govorili u prethodnom poglavlju. Jedina razlika je što se kod **`while`** petlje uvjet provjerava na početku pa se `naredba` unutar **`while`** petlje ne mora izvršiti niti jednom (ako je uvjet odmah lažan), dok se kod **`do/while`** petlje uvjet provjerava na kraju pa se petlja, tj. `naredba` unutar petlje uvijek izvrši najmanje jednom.
Opći oblik **`do/while`** petlje je:

```
do
      naredba;
while (uvjet);
```



Slika 4 – 3: Shematski prikaz **`do/while`** petlje

Primjer 4 – 8:
Što će pisati u varijabli *t* nakon izvršavanja sljedećeg dijela programa:

```
...
int t = 0, i = 5;
do
{
      t++;
      i += 5;
}
while (i < 12);
...
```

Rješenje:
Na početku varijabli *t* pridružujemo vrijednost 0, dok varijabli *i* pridružujemo vrijednost 5.
- ulaskom u **`do/while`** petlju vrijednost varijable *t* povećavamo za 1 (`t++`) i sada ona ima vrijednost 1, dok vrijednost varijable *i* povećavamo za 5 i sada ona iznosi 10. Budući da je *i* (10) < 12 vraćamo se na početak **`do/while`** petlje;
- vrijednost varijable *t* povećavamo za 1 (`t++`) te ona sada iznosi 2, dok vrijednost varijable *i* povećavamo za 5 i ona sada iznosi 15. Budući da je *i* (15) veće od 12 izlazimo iz **`do/while`** petlje, dakle u varijabli *t* će pisati vrijednost 2.

Primijetimo da bi u slučaju **`while`** petlje:

```
...
int t = 0, i = 5;
while (i < 12)
{
      t++;
      i += 5;
}
...
```

vrijednost varijable *t* nakon izvršavanja petlje bila **1**.

Na samom kraju dajmo još jedan primjer u kojem ćemo imati ugniježđene **do/while** i **for** petlju.

Primjer 4 – 9:
Štediša je tijekom posljednjih nekoliko godina u banku stavljao određene iznose. Prve godine je stavio iznos *x*. Svake sljedeće godine je stavio *y* kuna više nego prethodne godine. Na ulagane iznose nije dobivao nikakve kamate, a nakon *n* godina je raspolagao sa ukupno *k* kuna. Danas, dok o tome ponosno govori svojim prijateljima zanima ga koji je iznos u banku stavio prve godine i tu za njega započinju problemi.
Napišimo metodu čiji će ulazni parametri biti:
- prirodan broj *y* – razlika uloga između svake dvije godine;
- prirodan broj *n* – broj godina koje je štediša stavljao novac u banku;
- prirodan broj *k* – iznos u kunama s kojim štediša raspolaže.

Metoda treba vraćati prirodan broj, a koji će predstavljati iznos *x* – koji je štediša uplatio prve godine.

Rješenje:
Prije nego započnemo rješavati ovaj problem, simulirajmo ovaj problem na jednom konkretnom primjeru.
Pretpostavimo da je štediša prve godine u banku uložio 100 kuna (*x*), svake sljedeće je ulagao 100 kuna više (*y*), dakle, nakon 5 (*n*) godina u banci je imao 100 + 200 + 300 + 400 + 500 = 1500 (*k*) kuna.
Uz elementarno znanje matematike mogli bismo doći do formule prema kojoj bismo iz poznatih podataka (*y*, *n* i *k*) lako izračunali nepoznati podatak (*x*). Iako bi to rješenje bilo nešto elegantnije, mi zadatak nećemo riješiti na taj način, već ćemo ga riješiti "pogađanjem" broja *x*.
Budući da je traženi broj (*x*) prirodan broj, krenut ćemo od $x = 1$ i simulirati ulaganja. Ukoliko za dane *y* i *n* simulacijom dobijemo *n*, to će značiti da je 1 traženi broj. Inače ćemo povećavati broj *x* za 1 sve dok simulacijom iz brojeva *x*, *y* i *n* ne dobijemo broj *k*.

```
public static int pr49 (int y, int n, int k)
{
      int x = 0, s, p;
      do
      {
            x++;
            s = 0;
            p = x;
            //simuliramo rješenje problema, tj.
            //računamo koliko bi štediša imao nakon n godina
            //ako je prve godine u banku stavio x kuna a svake
            //sljedeće godine je iznos povećavao za y
            for (int i = 1; i <= n; i++)
            {
                  s += p;
```

```
                    p += y;
                }
            }
            while (s != k);
            return x;
}
```