4 Petlje

Sada kada smo videli kako možemo da granamo naš kod, odnosno da izvršavamo delove našeg koda samo kada su zadati uslovi ispunjeni, probajmo da rešimo naredni zadatak

Zadatak 4.1 (Zbir 5 brojeva)**.**

Korisnik unosi 5 celih brojeva sa standardnog ulaza. Ispisati sumu tih 5 brojeva.

Rešenje ovog zadataka je vrlo trivijalno, za njega nam čak ni ne treba grananje! Prosto, imaćemo jednu sumu koji inicijalizujemo na 0 i zatim nakon svakog korisničkog unosa, dodavaćemo taj broj na našu sumu.

Zbir5Brojeva

```
1  package nekiPaket;
2
3  import java.util.Scanner;
4
5  public class ZbirPetBrojeva {
6
7      public static void main(String[] args) {
8          Scanner sc = new Scanner(System.in);
9
10         int suma = 0;
11         int unos = sc.nextInt();
12         suma += unos;
13         unos = sc.nextInt();
14         suma += unos;
```

```
15          unos = sc.nextInt();
16          suma += unos;
17          unos = sc.nextInt();
18          suma += unos;
19          //Mozemo i bez unosa promenjive
20          suma += sc.nextInt();
21
22          System.out.println("Suma je " + suma);
23      }
24  }
```

Modifikujmo formulaciju ovog zadataka tako da korisnik pre početka unosa brojeva bira koliko će brojeva unositi:

Zadatak 4.2 (Zbir n brojeva)**.**

Korisnik unosi pozitivan ceo broj n a zatim i n celih brojeva. Ispisati sumu tih n celih brojeva.

Ovaj zadatak se razlikuje po jednoj, ali itekako bitnoj stvari od prethodnog, a to je da pre korisničkog unosa vrednosti n, mi ne znamo koliko puta treba da izvšimo komandu `sc.nextInt()`!
Ukoliko korisnik na početku unese broj 3, onda ćemo imati samo tri komande `sc.nextInt()`, ali ukoliko unese 100 onda nam treba njih 100 itd.
Naivni pristup bi bio da koristimo `if-else` tako da obradimo svaku mogućnost vrednosti n. U pseudo-kodu to bi izgledalo nešto poput ovog[1]:

```
1  if (n == 1) {
2      suma += sc.nextInt();
3  }
4  else if (n == 2) {
```

```
5       suma += sc.nextInt();
6       suma += sc.nextInt();
7   }
8   else if (n == 3) {
9       suma += sc.nextInt();
10      suma += sc.nextInt();
11      suma += sc.nextInt();
12  }
13   itd
```

Ali ako pogledamo, kako je n tipa `int`, odnosno zauzima 4B memorije, korisnik ima mogućnost da unese bilo koji broj od 0 do $\frac{2^{32}}{2} - 1 = 2,147,483,647$[2]
što je previše!

Čak i da nekako uspemo da rešimo ovaj zadatak, naredni jednostavno ne možemo sa trenutnim znanjem:

Zadatak 4.3 (Zbir brojeva)**.**

Korisnik unosi cele brojeve sve dok ne unese broj 0. Ispisati sumu tih unetih brojeva

Da bi smo ovakav tip zadataka mogli olako da rešimo (zadatke za koje ne znamo koliko puta odredjen skup komandi treba da se izvrši) moramo da uvedemo novi pojam, a to je pojam **petlje**.

Definicija 4.1 (Petlja)**.**

Petlja predstavlja naredbu kontrala toka programa i sačinjena je od blok koda koji se izvršava onoliko puta koliko to nalaže unapred zadat uslov. U Javi razlikujemo 3 vrste petlji:

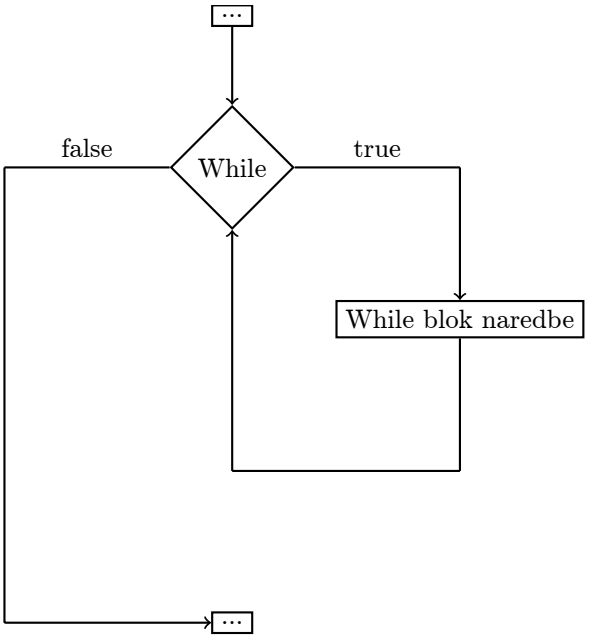
1. While petlja
2. Do while petlja
3. For petlja
   3.1 For-each petlja

Definicija 4.2 (Telo i glava petlje)**.**

Glava petlje predstavlja deo koda koji sadrži uslov petlje.
Telo petlje predstavlja blok koda koji prati ili prethodi uslovu petlje.

While petlja

While, odnosno `sve dok` petlja je naredba kontrola toka programa koja prima uslov i izvršava naredni blok kod sve dok je zadati uslov ispunjen, odnosno sve dok se taj uslov evaluira kao `true` i radi po sledecem principu:
Java će prvo proveriti tačnost uslova i ukoliko je uslov ispunjen izvršiće naredni blok koda liniju-po-liniju. Kada završi izvršavanje tog bloka, vraća se nazad na uslov i ponavlja se isti postupak. Ukoliko uslov nije ispunjen, Java preskače čitav blok i nastavlja sa daljim radom na normalan način. Grafička reprezentacija `while` petlje je:



Šablon pisanja `while` petlje je sledeci:

```
while (<uslov>) {
    //Linija koda 1
    //Linija koda 2
    ...
    //Linija koda N
}
```

Dakle, komanda počinje sa ključnom rečju `while`, prateći sa uslovom čije krajnje rešenje mora biti logički literal upisanim u obične zagrade i zatim otvaranje novog kod bloka. Unutar tog kod bloka, navode se sve komande čije izvršavanje će se desiti ako i samo ako je navedeni uslov ispunjen, tj. ako je vrednost tog logičkog literala `true`. Nakon izvršavanja linije koda N, Java se vraća nazad na uslov i ponovo ga proverava. Taj postupak se ponavlja sve dok je uslov ispunjen. Ukoliko uslov nije ispunjen, preskače se ceo blok koda i java dalje nastavlja svoje izvršavanje na standardan način.
Rešimo prvo zadatak 4.2

Zadatak4.2

```java
package NekiPaket;

import java.util.Scanner;

public class ZbirPrvihNBrojeva {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int suma = 0;
        int trenutniBroj = 1;
        while (trenutniBroj <= n) {
            suma += trenutniBroj;
            trenutniBroj++;
        }
        System.out.println(suma);
    }
}
```

Obratimo pažnju na par stvari, a posebno na naznačenu liniju, liniju 14, koja uvećava `trenutniBroj` za jedan. Da nemamo ovu komandu uslov da je `trenutniBroj` manji ili jednak od `n` bi **uvek** bio zadovoljen i cela naša petlja bi se vrtela u nedogled. Takav vid petlje nazivamo beskonačnom petljom i moramo je uvek izbeći tako što ćemo zagarantovati da u nekom koraku (koji ne mora nužno unapred da bude definisan) izlazimo iz petlje! `trenutniBroj` nazivamo **brojačem petlje** dok komandu u liniji 14 nazivamo ažuriranjem brojača petlje. Primetimo da uopšteno ažuriranje brojača petlje ne mora da bude poslednja linija koda u petlji (mada najčešće jeste):

```
while (<uslov>) {
    Linija koda 1;
```

```
    Linija koda 2;
    ...
    Linija ažuriranja brojača petlje
    ...
    Linija koda N;
}
```

U ovom slučaju iako nakon linije ažuriranja brojača petlje možda azurira brojač tako da uslov više ne bude ispunjen, linije nakonj nje sve do kraja `while` bloka (do linije koda N) biće izvršene svakako. Tek nakon izvršavanja linije koda N, vraćamo se na uslov i ponovo ga proveravamo!

Brojač se zove tako jer broji koliko smo *iteracija* petlje do sada izvršili. Brojači najčešće kreću od 0 i najčešće se uvećavaju samo za jedan, mada to nemora nužno da bude tako. Brojače najčešće označavamo slovima `i`, `j`, `k`, `l`, `m` itd.

Rešimo ostatak zadataka sada koristeći se `while` petljom

Zadatak4.1

```java
package NekiPaket;

import java.util.Scanner;

public class ZbirPetBrojeva {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int suma = 0;
        int i = 0;
        while (i < 5) {
            suma += sc.nextInt();
            i++;
        }
```

```
15          System.out.println(suma);
16      }
17  }
```

Zadatak4.3

```
1   package NekiPaket;
2
3   import java.util.Scanner;
4
5   public class ZbirDoNule {
6
7       public static void main(String[] args) {
8           Scanner sc = new Scanner(System.in);
9           int unetiBroj = sc.nextInt(); //Od korisnika
    zahtevamo da unese barem jedan broj. To moze biti i 0!
10          int suma = prviBroj; //Taj boj postavljamo za
    nasu trenutnu sumu
11          while (unetiBroj != 0) {
12              unetiBroj = sc.nextInt(); //Posto uneti broj
    nije nula, od korisnika zahtevamo da unese novi broj.
13              suma += unetiBroj; //Cak i ako korisnik unese
    nulu, to nece uticati na nasu sumu.
14          }
15          System.out.println(suma);
16      }
17  }
```

U rešenju zadatka 4.1, tačno vidimo da naša promenjiva `i` broji koliko puta smo prošli kroz ceo blok `while` petlje. Dakle u k-toj iteraciji petlje, promenjiva `i` će imati vrednost k. Po konvenciji, počeli smo da brojimo od nule, te idemo do broja 4, odnosno isključujemo broj 5!

U rešenju zadatka 4.3 nemamo eksplicitan brojač, već za uslov koristimo broj koji je korisnik uneo. Sve dok je taj broj različit od

nule, petlja će se vrteti i suma ažurirati adekvatno, u momentu kada korisnik unese broj 0, na sumu će se dodati 0, ali to ne menja njenu vrednost i sledeći put kada se uslov izvrši, preskače se cela `while` petlja i ispisuje se krajnja suma. Ovaj program će raditi i ako korisnik odmah unese broj 0. Tada će suma biti postavljena na 0, uslov će odmah biti neispunje i cela petlja će se preskočiti.

Zadatak 4.3 je specifičan po tome što od korisnika zahtevamo barem jedan unos pre nego što potencijalno udjemo u petlju, drugim rečima kao da želimo da zagarantujemo barem jedanu iteraciju petlje. Ovo možemo da postignemo na vrlo plastičan način koristeći nešto sto nazivamo **zastavice** (eng. 'flags') koje su često tipa `boolean` i koriste se upravo u ovakve svrhe: da razlikujemo prvu iteracije petlje od ostalih, da naznačimo da se neka specifična operacija odradila itd.

Zadatak4.3-Ponovo

```
1  package NekiPaket;
2
3  import java.util.Scanner;
4
5  public class ZbirDoNule {
6
7      public static void main(String[] args) {
8          Scanner sc = new Scanner(System.in);
9          boolean prvaIteracija = true; //postavljamo flag
   za prvu iteracija
10         int unetiBroj = 1; //Kreiramo uneti broj i
   postavljamo ga na proizvoljnu vrednost posto nam ne treba
   za prvu iteraciju i odmah cemo ga azurirati u petlji
11         while (prvaIteracija || unetiBroj != 0) { //Sa
   operatorom || zagarantujemo da ulazimo u petlju barem
   jednom posto je prvaIteracija true
```

```
12              prvaIteracija = false; //odmah postavljamo
    ovaj flag na netačno posto kada se ova iteracija završi,
    više nismo u prvojIteraciji i želimo da se uslov prekine
    samo kada korisnik unese broj 0.
13              unetiBroj = sc.nextInt();
14              suma += unetiBroj;
15          }
16          System.out.println(suma);
17      }
18  }
```

I ako smo uspeli da obrišemo višak komande `sc.nextInt()` to smo uradili po skupoj ceni nepreglednosti koda[3]. Upravo za ovakve svrhe, kao što je kreiranje petlje koja se izvršava barem jedanput koristimo `do-while` petlju.

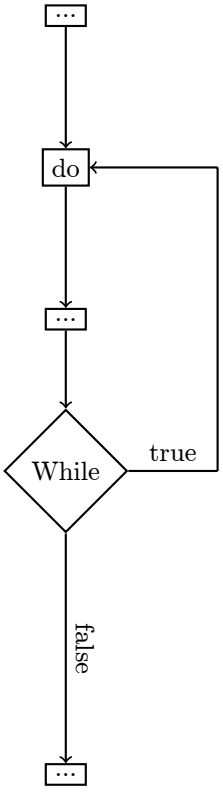
Do while petlja

Do while petlja, odnosno `radi ... sve dok` petlja, je petlja čije telo prethodi uslovu. Ovakva konstrukcija nam zagarantuje barem jednu iteraciju cele petlje, pre poredjenja uslova. Šablon pisanja `do-while` petlje je sledeći:

```
1  do {
2      Linija koda 1;
3      Linija koda 2;
4      ...
5      Linija koda N;
6  } while (<uslov>);
```

Dakle, efektivno zamenjujemo medjusobni polozaj uslova i tela petlje. Iz ovog šablona lako se vidi da će se telo izvšiti **čak i ako uslov inicijalno nije ispunjen!**

Grafički prikaz `do-while` petlje je sledeći:



Rešimo zadatak 4.3 ponovo, ovaj put koristeći `do-while` petlju.

Zadatak4.3-DoWhile

```
1  package NekiPaket;
2
3  import java.util.Scanner;
4
5  public class ZbirDoNule {
6
7      public static void main(String[] args) {
8          Scanner sc = new Scanner(System.in);
9          int suma = 0;
10         int unetiBroj; //Sada ne moramo da ga
   inicijalizujemo ni na sta
11         do {
12             unetiBroj = sc.nextInt();
13             suma += unetiBroj;
14         } while (unetiBroj != 0);
15         System.out.println(suma);
```

```
16        }
17    }
```

Ovde vidimo da smo zagarantovali barem jedan prolazak kroz petlju bez ikakvih dodatnih promenjivih ili uslova.

Važno pitanje koje treba postaviti je "Da li su `while` i `do-while` petlje medjusobno ekvivalentne?" odnosno da li sve što možemo da uradimo jednom, možemo i koristeću drugu petlju i obratno? Treba biti pažljiv prilikom odgovaranja na ovo pitanje; s jedne strane nisu ekvivalentne pošto `do-while` petlja zagarantuje *ulazak u telo petlje* barem jedanput, dok obična `while` petlja to ne garantuje, ali sa druge strane, sve što može da reši `while` petlja, može i `do-while` i obratnu, uz neke prepravke.

Već smo videli kako `while` petlju možemo da 'pretvorimo' u `do-while`, a da bi smo uspeli da `do-while` pretvorimo u `while` potrebne su nam dve naredne konstrukcije - jednu sa kojom smo se već susreli u drugačijem formatu.

Break i Continue naredbe

Razmotrimo naredna dva zadatka

Zadatak 4.4 (Unos 5 imena)**.**

Korisnik unosi 5 imena. Ispisati poruku:

- "<ime_1>, <ime_2>, <ime_3>, <ime_4>, <ime_5>, su prijatelji". Ukoliko se u bilo kom momentu unese ime "Grinc" treba ispisati poruku:
- "Grinc nije niciji prijatelj" i izaci iz programa

Zadatak 4.5 (Zbir nenegativnih)**.**

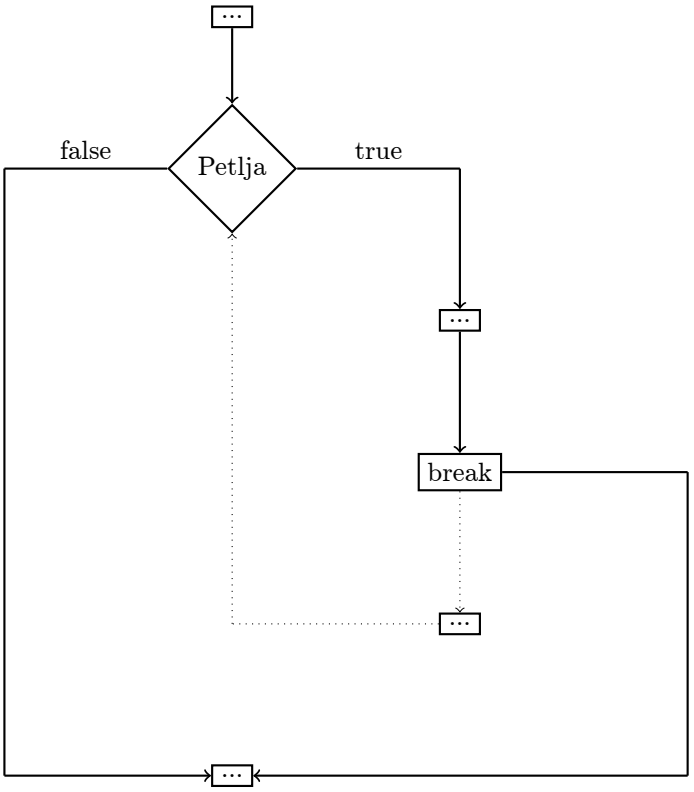
Korisnik unosi broj n a zatim i n celih brojeva. Izračunati sumu tih brojeva. Ukoliko korisnik unese negativan broj, omoguciti, zanemarit ga i od korisnika zahtevati da unese nenegativan broj. Tek kada korisnik unese takav broj, dodati ga na sumu i nastaviti izvršavanje koda.

Zadatak 4.4 je specifičan po tome što od nas zahteva prevremen izlazak iz petlje pod odredjenim uslovom. Ovu funkcionalnost ispoljevamo korišćenjem **break** naredbe.

Definicija 4.3 (Break naredba)**.**

Break naredba je naredba kontrola toka kojom izlazimo iz tela petlje i nastavljamo sa daljim izvršavanjem kod van petlje, bez provere uslova.

Grafički prikaz `break` naredbe je sledeci:



U momentu kada Java naidje na `break` naredbu, preskače sav ostatak kod, ne proverava uslov i nastavlja sa izvršavanjem koda nakon petlje u kojoj se nalazi. Koristeći novostečeno znanje, rešimo zadatak 4.4.

Zadatak4.4

```
1  package nekiPaket;
2
3  import java.util.Scanner;
4
5  public class Imena {
6      public static void main(String[] args) {
7          Scanner sc = new Scanner(System.in);
8
9          int i = 0;
10         String poruka = "";
11         boolean unetaSvaImena = true; //Koristimo
   obelezivac za kasnije
12         while (i < 5) {
13             String trenutnoIme = sc.nextLine(); //Pamtimo
   trenutno uneto ime
14             if (trenutnoIme.equalsIgnoreCase("Grinc")) {
15                 System.out.println("Grinc nije niciji
   prijatelj");
16                 unetaSvaImena = false; //Postavljamo
   obelezivac na false. To ce nam dati do znanja da nismo
   uneli svih 5 validnih imena i da ne treba da ispisemo
   nikakvu poruku.
17                 break; //Izlazimo iz petlje.
18             }
19             // Inace, unosimo ime u poruku
20             poruka += trenutnoIme + ",";
21         }
22         if (unetaSvaImena) { //Ukoliko je ovaj boolean
   true, to znaci da nije unet Grinc
```

```
            System.out.println(poruka + " su
    prijatelji");
        }
    }
}
```

Čitaoci sa oštrijim okom su možda primetili da zadatak 4.3 podseća dosta na zadatak 4.4 po tome što i tamo imamo petlju koja treba da prestane da se vrti pod odredjenom okolnošću. Tako, zadatak 4.3 može da se reši pomoću 'beskonačne' petlje i komande `break`. Beskonačna petlja je oblika `while (true)` jer takav uslov je uvek trivijalno zadovoljen. Kreiranje ovakvih petlji je dozvoljeno dokle god zagarantujemo da će se pod nekim uslovom komanda `break` izvršiti. Tada rešenje zadatka 4.3 izgleda:

Zadatak4.3-Break

```
package NekiPaket;

import java.util.Scanner;

public class ZbirDoNule {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int suma = 0;
        while (true) {
            int unetiBroj = sc.nextInt();
            if (unetiBroj == 0)
                break;
            suma += unetiBroj;
        }
        System.out.println(suma);
    }
}
```

što je na neki način najelegantnije i najviše čitko od svih ponudjenih.

Zadatak 4.5 može da se reši i prostim `if` uslovom na sledeći način

Zadatak4.5

```
1  package NekiPaket;
2
3  import java.util.Scanner;
4
5  public class ZbirDoNule {
6
7      public static void main(String[] args) {
8          Scanner sc = new Scanner(System.in);
9          int n = sc.nextInt();
10         int suma = 0;
11         int i = 0;
12         while (i < n) {
13             int unetiBroj = sc.nextInt();
14             if (unetiBroj > 0) {
15                 suma += unetiBroj; //Azuriramo sumu samo
   sa pozitivnim brojevima
16                 i++; //Azuriramo brojac samo kada je unos
   validan.
17             }
18             // Ukoliko nismo usli u if naredbu, brojac se
   nije azurirao te od korisnika ponovo zahtevamo da unese
   broj
19         }
20         System.out.println(suma);
21     }
22 }
```

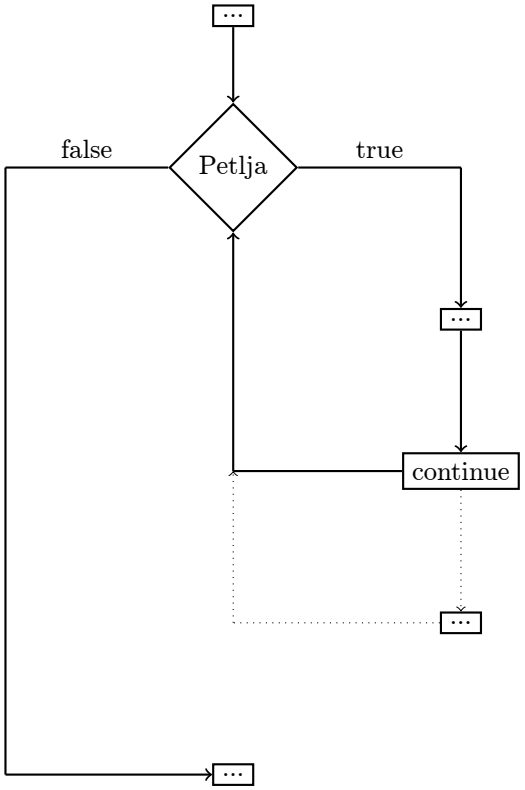
ali radi ilustracije obratimo pažnju malo bolje na ovaj zadatak. Iz formulacije zadatka možemo da vidimo da se od nas zahteva da na

neki način 'zanemarimo' odredjene iteracije petlje - one za negativni unos - i da se vratimo na sam početak petlje, gde ponovo proveravamo uslov i u zavisnosti od njegove vrednosti ponovo izvršavamo telo petlje ili nastavljamo dalje. Ovu funkcionalnost nam omogućava ključna reč **continue**.

Definicija 4.4 (Continue naredba)**.**

Continue naredba je naredba kontrole toka kojom se vraćamo na proveru uslova petlje bez daljnjeg izvršavanja tela petlje.

Grafički prikaz `continue` naredbe je sledeći:



Dakle, `continue` naredba nas vraća na početak petlje, što je u suprotnosti od `break` naredbe koja nas vodi na kraj petlje. Važno je obratiti pažnju na poziciju komande odnosno komandi koje ažuriraju uslov petlje. Ukoliko su one nakon `continue` naredbe, neće se ažurirati u esencijalno imamo ponovljeno iteraciju petlje. Znajući ovo, zadatak 4.5 možemo rešiti na sledeći način:

Zadatak4.5-Continue

```java
package NekiPaket;

import java.util.Scanner;

public class ZbirDoNule {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int suma = 0;
        int i = 0;
        while (i < n) {
            int unetiBroj = sc.nextInt();
            if (unetiBroj < 0)
                continue; // Ukoliko je uneti broj manji od nule vracamo se na pocetak petlje
            //Inace, azuriramo sumu i brojac
            suma += unetiBroj;
            i++;
        }
        System.out.println(suma);
    }
}

```

što je za nijansu čitljivije rešenje od prvog. Ključna reč `continue` se ne koristi toliko često kao `break` i uglavnom se lako može zaobići korišćenjem običnih `if` narebi.

For petlja

Razmotrimo jednostavan zadatak:

Zadatak 4.6 (Suma brojeva od 1 do n)**.**

Korisnik unosi pozitivan ceo broj n, veći od 1. Ispisati sumu brojeva $1 + 2 + \cdots + n$

Rešenje ovog zadatka može da se prikaže sa prostom `while` petljom:

SumPrvihN-while

```java
package nekiPaket;

import java.util.Scanner;

public class SumaPrvihN {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int i = 1; //Inicijalizacija brojaca
        int suma = 0;
        while (i <= n) { //Uslov
            suma += i;
            i++; //Azuriranje brojaca
        }
        System.out.println(suma);
    }
}
```

Kao što možemo da primetimo, ovde nema ničega neočekivanog: imamo inicijalizaciju brojaca, uslov petlje i azuriranje brojaca, no sa druge strane jedino što je bitno u toj petlji, odnosno što je važno za samu logiku zadatka je ova komanda koja ažurira tekuću sumu. U

svrhe lakše preglednosti koda, kao i skraćivanje samog koda pojavljuje se **for petlja**, odnosno `za petlja`.

Definicija 4.5 (For petlja)**.**

For petlja, odnosno `za` petlja, je petlja koja u svom uslovu sadrži inicijalizaciju brojača, uslov i ažuriranje brojača.

Tačna moć ove petlje se vidi u njenom šablonu:

```
for (<komande_inicijalizacije>; <uslov>; <komande_azuriranja>) {
    Linija koda 1;
    Linija koda 2;
    ...
    Linija koda N;
}
```

Pogledajmo na praktičnom primeru kako se realizuje:

SumPrvihN-for

```java
package nekiPaket;

import java.util.Scanner;

public class SumaPrvihN {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int suma = 0;
        for (int i = 1; i <= n; i++) { //Inicijalizacija, uslov i azuriranje sve u jednom

```

```
12                suma += i;
13            }
14            System.out.println(suma);
15        }
16    }
```

Sada, umesto da imamo dve dodatne linije koda koje inicijalizuju i azuriraju brojac, te komande su grupisane zajedno sa uslovom kod `for` petlje, pre početka tela petlja. Na ovaj način, tačno možemo videti odakle krećemo (inicijalizacija), dokle idemo (uslov) i kojim korakom prelazimo iz jedne iteracije u narednu (u našem slučaju korakom od 1).

Važno je napomenuti da se:

- Inicijalizacija izvršava samo jednom, pre prve iteracije petlje.
- Komanda uslova se izvršava pre izvršavanja tela petlje
- Telo petlje se izvršava nakon izvršavanja uslova. Ukoliko uslov u startu nije zadovoljen, celo telo petlje se preskače.
- Komande ažuriranja izvršavaju se nakon izvršavanja svih komandi iz tela petlje; odmah pre ponovonog izvršavanja uslova petlje.
- Nakon izvršavanja komandi ažuriranja, izvršava se komanda uslova i ceo postupak se nastavlja.

Kao i za `while` i `do-while`, treba se zapitati o ekvivalentnosti `for` petlje i recimo `while` petlje. Kao što smo mogli da vidimo za `while` i `do-while` petlje, dovoljno je da pronadjemo algoritam kojim ćemo konvertovati kod da umesto korišćenja jedne od te dve petlje, predjemo na korišćenje druge. Već smo videli kako možemo da konvertujemo `while` petlju u `for` petlju u prethodnom primeru. Mislim da nije teško opisati algoritam kako bi konvertovali `for` u `while` petlju:

1. Iz glave petlje prebaciti prvu komandu i staviti je pre početka petlje.
2. Prebaciti ključnu reč `for` u ključnu reč `while`.
3. Iz glave petlje prebaciti poslednju komandu i staviti je na sam kraj petlje.

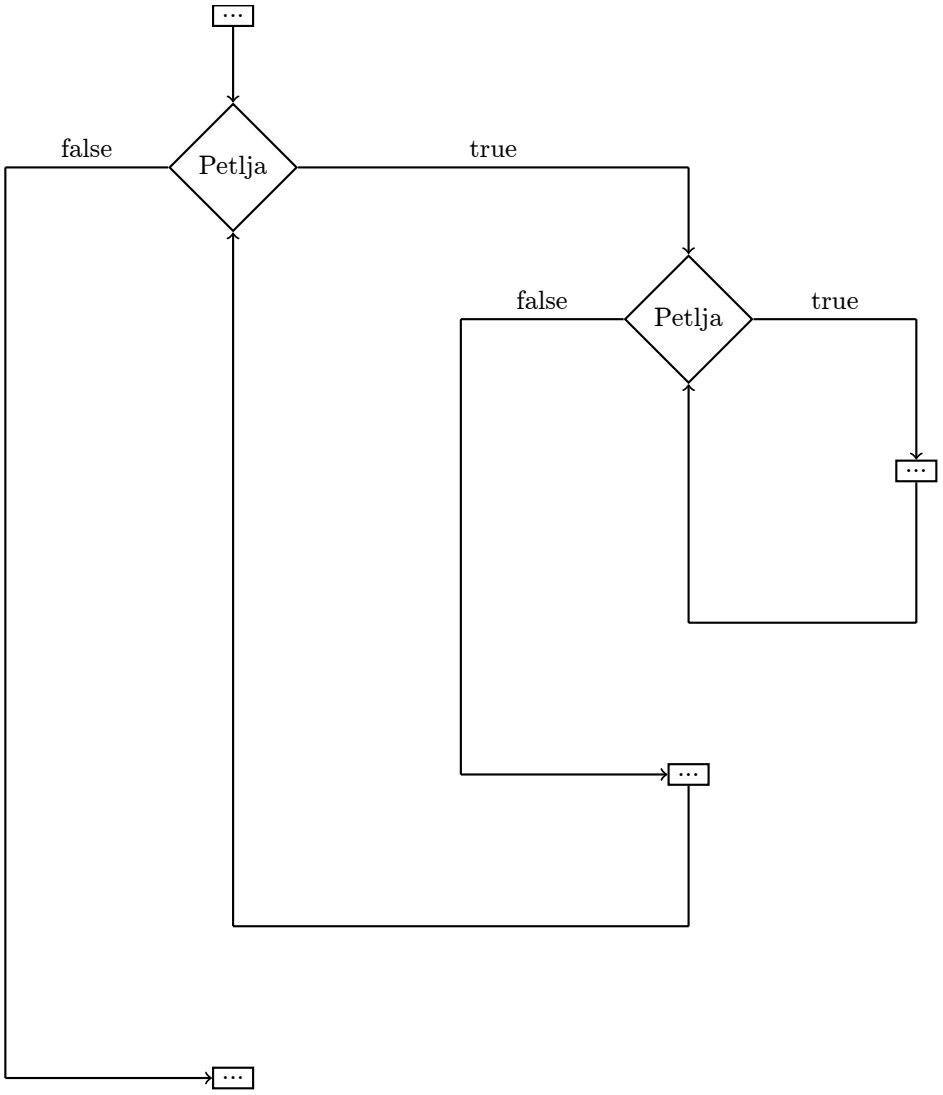
Pažljivi čitaoci su možda primetili da smo koristili množinu prilikom navodjenja komandi inicijalizacije i ažuriranja. To je zato što je moguće inicijalizovati više promenjivih i imati komandu ažuriranja za njih (ne nužno za sve). Primer takvog šablona bi izgledao:

```
for (int i = 0, j = 2*suma, k = 5; (i + j) < n; i++, j += 2) {
    ...
}
```

gde zapetom `,` odvajamo promenjive prilikom njihove inicijalizacije i prilikom navodjenja komandi ažuriranja. Ovi oslobadjamo identifikatore koje koristimo unutar petlje iz bloka koda u kome se nalazi petlja!

Petlje u petlji

Kako telo petlje definiše novi blok koda, to sva pravila koja važe za blokove koda važe i za petlje. Ništa nas ne sprečava da unutar jedne petlje unesemo i novu petlju itd. Tipovi tih *ugnježdenih* petlji takodje ne moraju da se poklapaju.
Grafički prikaz petlje u petlji:



Zadatak 4.6 (Sahovska Tabla)**.**

Korisnik unosi pozitivan ceo broj n. Kreirati $n \times n$ sahovsku tablu. Bela polja predstaviti znakom `o`, a crna znakom `x`. Gornje levo polje smatrati da je uvek belo.

Zadatak4.6

```
1  packge nekiPaket;
2
3  import java.util.Scanner;
4
5  public class SahovskaTabla {
6
7      public static void main(String[] args) {
```

```
8           Scanner sc = new Scanner(System.in);
9           int n = sc.nextInt();
10          boolean jeBelo = true;
11          for (int i = 0; i < n; i++) {
12              for (int j = 0; j < n; j++) {
13                  if (jeBelo) {
14                      System.out.print('o');
15                  else
16                      System.out.print('x');
17                  System.out.print(' ');
18                  jeBelo = !jeBelo;
19              }
20              System.out.println(); //Stampamo novi red, ekvivalentno System.out.println("");
21          }
22      }
23  }
```

---

1. Zapravo, ako se već vodimo ovom suludom idejom, `switch-case` bi bio mnogo poželjni; razmislite zašto! ↩
2. Delimo dvojikom da izaberemo samo pozitivne brojeve, dok -1 stoji zbog toga kako se znak tretira u komplementu dvojike ↩
3. Što se efikasnosti tiče, nismo je ovim narušili, oba rešenja su linearna. ↩