Uvod u programiranje u programskom jeziku C#

Dragan Mašulović

u saradnji sa

Nebojšom Vasiljevićem

i

Milanom Vugdelijom

Novi Sad, Beograd, 2020.

Predgovor

Računarski program, na prvi pogled, predstavlja sredstvo komunikacije čoveka i mašine. Zapravo, radi se o komunikaciji *među ljudima*, i to višeslojnoj i komplikovanoj. Na jednom nivou računarski program predstavlja oblik komunikacije između dizajnera programa i korisnika programa: dizajneri u program ugrađuju svoj pogled na svet ili jedan njegov segment, i to prenose korisniku programa. Računarski program, međutim, predstavlja i oblik komunikacije među programerima. Zato je važno da programi budu ne samo dobro osmišljeni, već i dobro napisani, kako bi mogli da se upotrebe na oba nivoa komunikacije. To, dalje, znači da programski jezik u kome je program napisan treba da ima veliku izražajnu moć kako bi izlaganje ideja koje su u program ugrađene bilo što prirodnije. Ovo je važno i za budućeg korisnika programa, ali i za drugog programera koji će imati potrebu da pročita i razume kôd.

Tekst pred vama predstavlja *uvod u programiranje* u programskom jeziku C#. Programski jezik C# je programski jezik 21. veka – kompanija Microsoft je prvu verziju ovog jezika (C# 1.0) objavila 2002. godine. Osnovna namena programskog jezika C# je industrijski razvoj softvera. Međutim, niz dizajnerskih odluka kojima je u poslednjih dvadesetak godina C# uobličen u moćan alat koga danas imamo doveli do toga da on može da se koristi i kao dobra osnova za uvod u programiranje i algoritamsko mišljenje. Jedna od ključnih karakteristika ovog jezika je da se on neprestano razvija i redovno održava, i slobodan je za upotrebu.

Programski jezik C# ima ogroman broj mogućnosti i većina njih *neće* biti opisana u ovom tekstu – Detaljan opis svih njegovih mogućnosti bi zahtevao hiljade strana. Zato smo u prethodnom pasusu naglasak stavili na reč *uvod*: ovaj tekst je *uvod u programiranje* u jednom savremenom programskom jeziku. Naša iskrena nada je da će nakon savladavanja ovog uvoda čitalac moći sam da usvoji i ostale elemente programskog jezika C# ako proceni da je to jezik u koji vredi investirati svoje vreme.

Osnovna svojstva programskog jezika C# ćemo demonstrirati na primerima. To su u nekim slučajevima kratki delovi C# kôda, ali ćemo često pokazivati i kompletne C# primere. Kompletan C# program će u tekstu biti naveden ovako:

⟨C# fajl⟩

```
using System;

class Pozdrav {
  static void Main() {
    string ime;
    Console.WriteLine("Kako se zoves?");
    ime = Console.ReadLine();
    Console.WriteLine("Zdravo, {0}!", ime);
  }
}
```

Klikom na link ⟨C# fajl⟩ koji se javlja na margini pored kôda čitalac dobija pristup C# datoteci sa izvornim kôdom koga može da iskompajlira i izvrši.

Pored primera, svaka celina je propraćena nizom zadataka za samostalan rad. Neki od zadataka su "obični" zadaci, dok su drugi preuzeti iz *Zbirke algoritamskih zadataka* sa portala www.petlja.org. Ovi drugi su "interaktivni" – za njih postoji podrška na portalu www.petlja.org koja čitaocu nudi mogućnost da svoje rešenje zadatka pošalje Petljinim test-serverima na proveru. Zadaci će u tekstu biti navedeni ovako:

Zadaci.

1.1. Napisati po analogiji C# program koji računa površinu kruga.

1.2. Putovanje ⟨LINK⟩

Zadaci čiji redni broj je crn su "obični" zadaci, a zadaci čiji redni broj je zelen su "interaktvini" zadaci iz *Zbirke algoritamskih zadataka* sa portala www.petlja.org. Za zadatke sa portala www.petlja.org je najčešće navedeno samo ime zadatka iza čega sledi link na odgovarajuću stranicu na portalu.

Želimo vam mnogo zabave prilikom prolaska kroz ovaj tekst i uspeha u savladavanju materijala. Iščekujemo vaše komentare!

U Novom Sadu i Beogradu,
februar 2020.

Vaša,
Petlja

Sadržaj

Glava 1

Prvi program u C#

U ovoj glavi počinjemo sa predstavljanjem osnovnih osobina programskog jezika C#. Počećemo analizom jednostavnog primera, a onda ćemo pokušati da po analogiji napravimo nekoliko sličnih programa.

1.1 Prvi C# program

Pogledajmo C# program koji od korisnika učita njegovo ime i onda ga pozdravi:

⟨C# fajl⟩

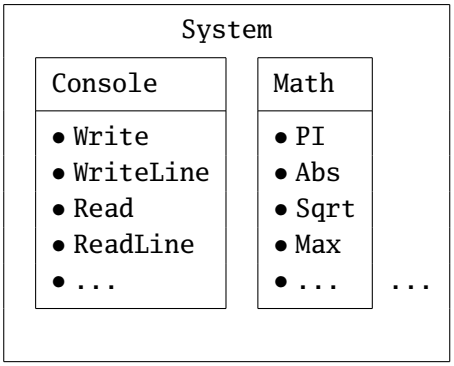
```
using System;

class Pozdrav {
  static void Main() {
    string ime;
    Console.WriteLine("Kako se zoves?");
    ime = Console.ReadLine();
    Console.WriteLine("Zdravo, {0}!", ime);
  }
}
```

U modernim jezicima ulazno-izlazne naredbe, matematičke funkcije i slično nisu deo jezika, već se nalaze u bibliotekama. Zato većina C# programa počinje pozivanjem biblioteka koje su nam potrebne za rad, recimo ovako:

```
using System;
```

Biblioteka `System` sadrži razne klase kao što su klasa `Console` (u kojoj se nalaze metodi za ispis na konzolu i učitavanje sa konzole) i klasa `Math` (u kojoj su

definisane osnovne matematičke konstante i implementirane osnovne matematičke funkcije).



Ključna konstrukcija svakog modernog programskog jezika je *klasa*. Klase služe kao kontejneri za podatke i metode za obradu podataka. Svaki C# program mora da ima bar jednu klasu. Zato naš primer izgleda ovako:

```
using System;

class Pozdrav {
    ...
}
```

Primetimo da se klasa `Pozdrav` nalazi u datoteci `PrviPrimer.cs` odakle lako zaključujemo da ime klase koja sadrži program ne mora da bude u vezi sa imenom datoteke u koju smo smestili klasu.

Da bi program mogao da se izvrši, klasa u koju je smešten mora da sadrži specijalni metod koji se zove `Main` (tačno tako, sa velikim M na početku). To je metod od koga program kreće sa izvršavanjem. Klasa može da sadrži više metoda, i takve primere ćemo videti uskoro.

U primerima koje ćemo pisati metod `Main` neće imati argumente i zato pišemo `Main()` što označava metod bez argumenata. Takođe, metod `Main` neće vraćati operativnom sistemu nikakvu vrednost, pa ga opisujemo kao `void` (engl. void = prazno):

```
using System;

class Pozdrav {
  static void Main() {
    ...
  }
}
```

Osim toga, metod `Main` je deklarisan i kao statički. Metodi mogu da budu *statički* i *dinamički*. Dinamički metodi se vezuju za objekte i njih koristimo za objektno-orijentisano programiranje, o čemu ćemo pričati znatno kasnije. Za sada će svi naši metodi biti statički. (Međutim, čak i kada pišemo objektno-orijentisani kôd, metod `Main` mora biti deklarisan kao statički. No, kao što smo rekli, o tome kasnije.) Dakle, metod `Main` predstavlja *glavni metod* od koga počinje izvršavanje programa. Odatle ime: engl. *main* = glavni.

Telo metoda `Main` počinje *deklaracijom promenljive* tipa *string*. *Promenljiva* je parče memorije računara u koje može da se smesti neka vrednost. Promenljiva ima svoje *ime* i svoj *tip*. U svim jezicima koji vuku korene od programskog jezika C promenljiva se deklariše ovako:

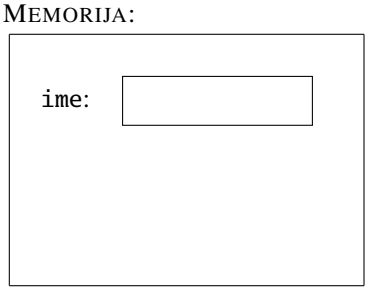
$$\langle tip \rangle \ \ \langle spisak\ imena \rangle;$$

tako da deklaracija

```
string ime;
```

znači da smo kompajleru najavili da ćemo imati promenljivu tipa `string` koja će se zvati `ime`.

Nakon ove deklaracije računar će u memoriji rezervisati prostor za promenljivu koja se zove `ime` i u koju može da se smesti neki niz slova.



Tip promenljive objašnjava računaru koliko memorije da zauzme za tu promenljivu, kao i to koje operacije se mogu izvoditi sa vrednostima koje su smeštene u promenljivoj. (Ako ovo nije baš sasvim jasno, razjasniće se uskoro!)

Tipom `string` označavamo nizove simbola kao što su ‘`Zdravo! Kako si?`’.

Sledi naredba

```
Console.WriteLine("Kako se zoves?");
```

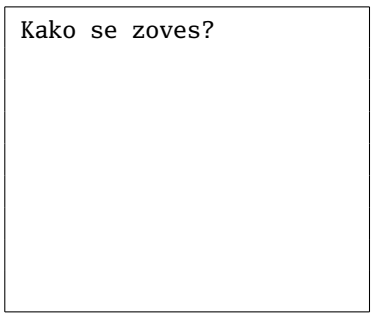
koja ispisuje poruku korisniku tako što iz klase `Console` poziva metod `WriteLine` (a klasu `Console` smo uvezli iz biblioteke `System` na samom početku).

Tekst koji je naveden između navodnika se ispiše u jednom redu na monitoru. Naredba

```
ime = Console.ReadLine();
```

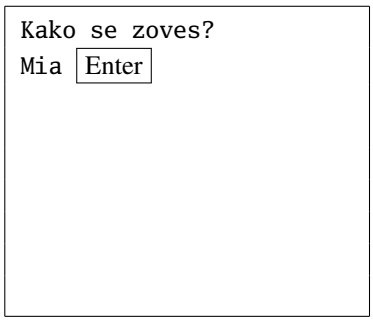
od korisnika učitava neki niz simbola. Naredba `ReadLine` je kovanica i čita se "read line" (*ρi:d lajn*), a znači "čitaj red (do kraja)".

MONITOR:



Kada naiđe na ovu naredbu računar prekine sa radom i sačeka da korisnik otkuca neki niz simbola i pritisne taster Enter.

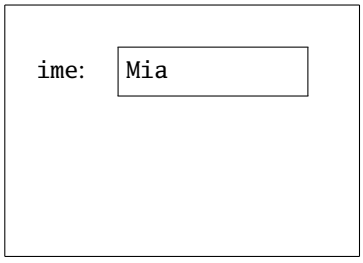
MONITOR:



U ovom primeru korisnik je uneo niz simbola `Mia`. Obratite pažnju na to da se nakon pritiska na taster Enter na monitoru *neće* pojaviti uokvirena reč "Enter"; ovo je samo da se podsetimo da računar neće nastaviti sa radom dok mu pritiskom na Enter ne stavite do znanja da ste završili sa unosom.

Nakon toga se uneti niz simbola smešta u promenljivu `ime`.

MEMORIJA:



Konačno, pozdrav ispisujemo naredbom

```
Console.WriteLine("Zdravo, {0}!", ime);
```

Jedina razlika između naredbi `Write` i `WriteLine` je u tome što `WriteLine` nakon ispisanog teksta pređe u novi red na konzoli, dok `Write` to ne čini. I jedna i druga naredba imaju sledeći oblik:

`Console.WriteLine(`⟨*format*⟩`,`⟨*promenljive*⟩`);`

Pri tome je ⟨*format*⟩ string koji sadrži "običan tekst" i "plejs-holdere" (engl. *place holders*). Plejs-holder ima oblik `{0}`, `{1}`, `{2}`..., i označava prvu, drugu, treću... promenljivu sa spiska (indeksi svih nizova u jeziku C# počinju od nule, pa zato `{0}` označava *prvu* promenljivu na spisku).

Tako, kada se ovakav string pojavi kao format:

```
"Zdravo, {0}!"
```

na monitoru će biti biti ispisano "Zdravo, Mia!"

MONITOR:

```
Kako se zoves?
Mia [Enter]
Zdravo, Mia!
```

☞ *U ovom primeru smo iz klase* `Console` *pozivali metode* `WriteLine` *i* `ReadLine` *i primetili smo da koristimo oblik*

⟨*klasa*⟩.⟨*metod*⟩(...);

kadgod želimo da iz neke klase pozovemo neki metod.

1.2 Još tri primera

Primer: ***Introducing James Bond.*** Ko je gledao filmove o Džejmsu Bondu zna da se on uvek predstavlja sa

"My name is Bond. James Bond."

Evo kako to izgleda u obliku C# programa:

⟨C# fajl⟩

```
using System;

class IntroducingJamesBond {
  static void Main() {
    string ime, prezime;
    Console.WriteLine("Who are you?");
    ime = "James";
    prezime = "Bond";
    Console.WriteLine("My name is {1}. {0} {1}.", ime, prezime);
  }
}
```

Kada se pokrene, program ispisuje:

```
Who are you?
My name is Bond. James Bond.
```

Primer: Obim kruga. Ispod je dat C# program kojim se računa obim kruga. Program radi ovako:

(1) učita od korisnika poluprečnik kruga,

(2) izračuna obim, i

(3) ispiše dobijenu vrednost obima.

⟨C# fajl⟩

```
using System;

class ObimKruga {
  static void Main() {
    double r, Obim;

    Console.WriteLine("Unesi poluprecnik kruga");
    r = double.Parse(Console.ReadLine());
    Obim = 2 * r * Math.PI;
    Console.WriteLine("Obim kruga je {0}", Obim);
  }
}
```

Telo metoda `Main` počinje deklaracijom dve promenljive tipa `double`. To su promenljive u koje možemo da smestimo neki realan broj (što je broj koji može da ima decimale, kao što je $-2,49$ ili $0,27758$ ili 11) i to "dvostruke preciznosti" — odatle ime. (Realne brojeve "jednostruke preciznosti" nećemo koristiti.) Deklaracijom

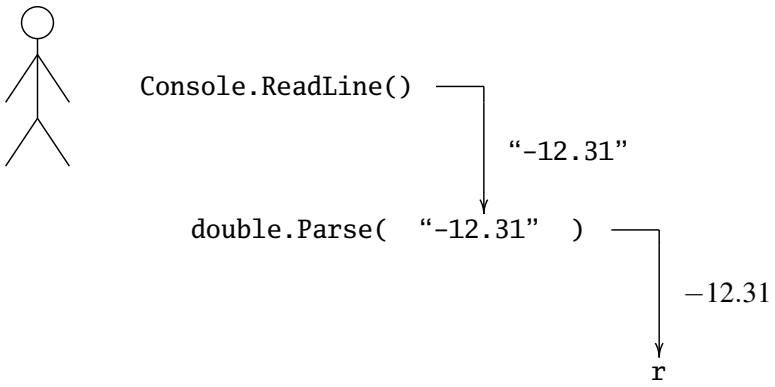
```
double r, Obim;
```

kompajleru najavljujemo da ćemo imati dve promenljive tipa `double` koje će se zvati `r` i `Obim`.

Učitavanje podataka u modernim programskim jezicima je relativno zakukuljeno. U našem primeru smo jedan realan broj učitali ovako:

```
r = double.Parse(Console.ReadLine());
```

To znači da smo prvo pozvali metod `Console.ReadLine` koji je od korisnika učitao niz simbola, nakon čega konstrukcija `double.Parse` poziva metod `Parse` klase `double`. (O, da... `double` je klasa. Zapravo, u modernim programskim jezicima sve je klasa, i klasa je sve. O ovome kasnije.) Metod `double.Parse` onda preuzme taj niz simbola i pokuša od njega da napravi mašinsku reprezentaciju realnog broja dvostruke preciznosti. Ako je korisnik umesto razumne vrednosti uneo neku besmislicu ovaj metod će se pobuniti i generisati *izuzetak (engl. exception)*. (O izuzecima kasnije.)



☞ *Učitavanje u jeziku C# uvek radi u dve faze:*

- *učitavanje niza simbola, i*
- *parsiranje učitanog niza.*

Parsiranje je proces kojim se proverava da li je dati niz simbola formiran prema odgovarajućim pravilima. Ako je niz simbola formiran ispravno, metod `double.Parse` potom određuje mašinsku reprezentaciju tog niza simbola.

Sledi naredba dodele:

```
Obim = 2 * r * Math.PI;
```

koja promenljivoj `Obim` dodeljuje vrednost izraza `2 * r * Math.PI` (primetimo da smo koristili unapred definisanu konstantu `PI` iz klase `Math`.) Konačno, rezultat ovih komplikovanih izračunavanja ispisujemo naredbom

```
Console.WriteLine("Obim kruga je {0}", Obim);
```

Primer: PDV. Kao poslednji primer u ovom odeljku napisaćemo C# program koji učitava cenu nekog proizvoda u koju nije uračunat PDV, i ispisuje cenu istog proizvoda sa uračunatim PDV. Uzećemo da PDV iznosi 20%.

⟨C# fajl⟩

```
using System;

class CenaSaPorezom {
  static void Main() {
    const double PDV = 20;
    Console.WriteLine("Unesi cenu bez PDV");
    double cena = double.Parse(Console.ReadLine());
    double porez = cena * PDV / 100;
    double cenaSaPDV = cena + porez;
    Console.WriteLine("Cena sa PDV je {0}", cenaSaPDV);
    Console.WriteLine("PDV od {0}% iznosi {1}", PDV, porez);
  }
}
```

Naredbom

```
const double PDV = 20;
```

deklarišemo konstantu koja je tipa `double`, koja se zove `PDV` i čija vrednost je `20` (stopa PDV). Razlika između promenljive i konstante je u tome što konstantama ne možemo da menjamo vrednost. Naredba

```
double cena = double.Parse(Console.ReadLine());
```

u isto vreme deklariše promenljivu i dodeljuje joj vrednost. To je skraćeni oblik od

```
double cena;
cena = double.Parse(Console.ReadLine());
```

Zadaci.

1.1. Napisati C# program koji od korisnika učitava njegovo ime u nominativu i vokativu, i onda ga pozdravlja, recimo ovako:

```
Kako se zoves -> Milan
Kako tvoje ime glasi u vokativu -> Milane
Znao sam! Ti se zoves Milan.
Zdravo, Milane!
```

1.2. Napisati po analogiji C# program koji računa površinu kruga.

1.3. Napisati po analogiji C# program koji računa obim i površinu pravougaonika. (Uputstvo: Sabiranje se označava znakom + kao što smo i navikli, množenje ima veći prioritet od sabiranja, a mogu se koristiti i obične zagrade tačno onako kako smo navikli u matematici.)

1.4. Putovanje ⟨LINK⟩

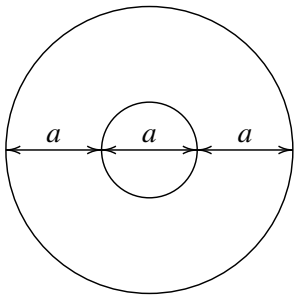
1.5. Napisati C# program koji učitava temperaturu iskazanu u Celzijusima (t_C) i ispisuje odgovarajuću vrednost temperature u Farenhajtima (t_F). Odnos je sledeći:

$$t_F = \gamma \cdot t_C + \delta$$

gde je $\gamma = 1.8$, a $\delta = 32$. Pri tome koristiti konstante i obratiti pažnju da na raspolaganju nemamo grčka slova kao ni indekse.

1.6. Napisati C# program koji učitava temperaturu iskazanu u Farenhajtima i ispisuje odgovarajuću vrednost temperature u Celzijusima.

1.7. Beli kružni stolnjak treba porubiti i ukrasiti crvenom trakom tako da rub stolnjaka bude opšiven trakom i da na stolnjak bude našivena još jedna traka, ali tako da rastojanja među trakama budu a centimetara, ovako:



Napisati C# program koji od korisnika učitava realan broj a i potom određuje i štampa koliko centimetara crvene trake je za to potrebno.

1.8. Stolnjak ⟨LINK⟩

1.9. Fudbalski teren ⟨LINK⟩

1.10. Atletičar se sprema za takmičenje tako što trči oko fudbalskog terena dimenzija $d \times s$. Prve sedmice trči svaki dan po 10 krugova oko terena. Druge sedmice trči svaki dan po 20 krugova oko terena. Treće sedmice trči svaki dan po 30 krugova oko terena.

Napisati C# program koji učitava d i s (koji su dati u metrima) i za svaku sedmicu ispisuje koliko metara atletičar pretrči dnevno te sedmice, i koliko će ukupno kilometara pretrčati na kraju sedmice.

1.11. Napisati C# program koji učitava cenu nekog proizvoda sa uračunatim PDV, i ispisuje cenu istog proizvoda bez PDV. Uzeti da PDV iznosi 20%.

1.12. Jedna vrsta pantalona je na sezonskom sniženju od 25%, ali ako se pantalone plate u gotovini ovako snižena cena se umanjuje na kasi još za 5%. Milena je odlučila da kupi baš tu vrstu pantalona i da plati gotovinom. Napisati C# program koji od korisnika učitava cenu pantalona pre sniženja i računa koliko Milena treba da plati za njih.

1.13. Prilikom uvoza robe u Republiku Srbiju se na osnovnu cenu proizvoda prvo zaračuna carina od 10%, pa se na tako dobijeni iznos zaračuna PDV od 20%. Napisati C# program koji od korisnika učitava cenu (u evrima) nekog proizvoda koji se uvozi u Srbiju, i utvrđuje za koliko je njegova cena u Srbiji uvećana ovom procedurom.

1.14. Napisati C# program koji od korisnika učitava realan broj r i potom računa i štampa zapreminu lopte čiji poluprečnik je r koristeći formulu $V = \frac{4}{3} r^3 \pi$.

1.15. Napisati C# program koji od korisnika učitava realne brojeve r i H, nakon čega računa i štampa površinu i zapreminu pravilnog uspravnog valjka čiji poluprečnik je r, a visina H koristeći formule $P = 2r\pi(r+H)$ i $V = r^2\pi H$.

1.16. Nivo bazena ⟨LINK⟩

1.3 Komentari

Komentar je tekst koji je namenjen čoveku i zato ga C# prevodilac ignoriše. Komentare ostavlja programer ili zato da bi sebi objasnio šta je hteo da postigne nekim delom programa, ili da pomogne onom ko posle njega bude čitao program u želji da shvati šta se tu dešava. Prava uloga komentara se ne može dobro videti

na malim primerima. Kasnije, kada budemo pisali veće programe, videćemo da su dobri komentari ključni za razumevanje programa.

U jeziku C# postoje dve vrste komentara – komentari u jednom redu koji izgledaju ovako:

```
// ovo je komentar koji traje do kraja reda
```

i komentari u više redova koji izgledaju ovako:

```
/*
   ovo je komentar koji je poceo u prethodnom redu
   a zavrsice se u narednom redu
 */
```

Ovi poslednji se često pišu i ovako da bi se lakse uočili:

```
/*
 * ovo je komentar koji je poceo u prethodnom redu
 * a zavrsice se u narednom redu
 */
```

Program sa komentarima izgleda ovako:

```
using System; // ovde su klase Console, Math i jos stosta

class ObimSaKomentarima {
  static void Main() {
    double r, Obim;

    /*
     * formulu za obim kruga
     * svako dete zna!
     */

    Console.WriteLine("Unesi poluprecnik kruga");
    r = double.Parse(Console.ReadLine());
    Obim = 2 * r * Math.PI;
    Console.WriteLine("Obim kruga je {0}", Obim);
  }
}
```

⟨C# fajl⟩

1.4 O razmacima i stilu

C# prevodilac (kompajler) ne vodi računa o broju razmaka koji se pojavljuju u programu (osim, naravno, onih koji se pojavljuju u porukama korisniku). Tačnije,

ne pravi razliku između jednog i trideset sedam razmaka. Bitno je samo da na nekim mestima postoji bar jedan razmak. Na primer, važno je da se između `static`, `void` i `Main` nalazi bar po jedan razmak zato što je nemoguće utvrditi šta bi tačno značilo `staticvoidMain`. Prevodilac ne bi imao druge, nego da pretpostavi da je `staticvoidMain` jedna reč.

Ova lepa osobina (da broj razmaka nije bitan) se koristi da bi se naznačila struktura programa. Ostavljanjem nekoliko razmaka na početku reda, ali na sistematski i lukav način, program dobija na čitkosti.

Program `ObimKruga` koga smo analizirali u prethodnom odeljku se može napisati i ovako→ ali teško da bi neko pri zdravom razumu ovako zapisan program smatrao privlačnim.

```
using System; class ObimKruga
{ static void Main() { double
r, Obim; Console.WriteLine(
"Unesi poluprecnik kruga"
); r = double.Parse(
Console.ReadLine()); Obim = 2
*r*Math.PI; Console.WriteLine
("Obim kruga je {0}", Obim); }}
```

Postoje preporuke o tome kako se nazubljuje program da bi bio "lep" i "čitak". Osnovna ideja je da se naredbe koje se nalaze unutar zagrada { i } uvuku za nekoliko razmaka (najčešće 2, 3 ili 4). Mi nećemo navoditi ova pravila, nego ćemo ih "pokupiti uz put".

1.5 Vrste reči — leksika programskog jezika

Mada je C# veštački jezik, i on, po ugledu na prirodne jezike, ima vrste reči, pravila interpunkcije i gramatiku. Spisak propisa koji regulišu vrste reči, pravila interpunkcije (i još neke poslove) zove se *leksika* jezika. Spisak propisa koji regulišu gramatiku jezika zove se *sintaksa* jezika.

Rezervisane reči opisuju neke specijalne programske konstrukcije. Do sada smo videli sledeće rezervisane reči programskog jezika C#:

`using` `class` `static` `void`

Rezervisanih reči ima još, naravno.

Imena. Rezervisane reči se tako zovu zato što su rezervisane za konstrukcije programskog jezika i one se ne mogu u programu pojaviti kao imena promenljivih. Druga vrsta reči koje se mogu javiti u programu su imena kao što je ime klase ili ime promenljive.

☞ *Ime je niz slova i brojeva koji mora početi slovom. U imenu se može pojaviti i specijalni simbol _ (donja crta).*

Primeri ispravnih imena: `obimKruga`, `program_broj_14`, `string2real`, `a1b2c3`. Primeri *neispravnih* imena:

`2obim`	ime mora da počinje slovom,
`Obim@Kruga`	pojavljuje se znak koji nije ni slovo ni broj,
`Obim Kruga`	u imenu ne sme da se pojavi razmak

☞ *Velika i mala slova se razlikuju, naravno, tako da su* `Pi`*,* `pI` *i* `PI` *tri različita imena.*

Znaci interpunkcije razdvajaju različite delove programa, ili manje delove unutar većih delova. *Vitičaste zagrade* ograničavaju nizove naredbi koji čine *blokove kôda*. *Obične (oble) zagrade* obuhvataju argumente metoda koji su razdvojeni *zarezima*. Zarezi se javljaju i u deklaraciji promenljivih. *Tačka-zarez* se javlja kao *terminator* — označava kraj naredbe. *Navodnici* obuhvataju tekst koji se javlja kao poruka korisniku. *Tačka* se koristi u pozivu metoda iz neke klase.

```
using System;

class ObimKruga {
  static void Main() {
    double r, Obim;
    Console.Write("Unesi poluprecnik kruga -> ");
    r = double.Parse(Console.ReadLine());
    Obim = 2 * r * Math.PI;
    Console.WriteLine("Obim kruga je {0}", Obim);
  }
}
```

Kviz.

1. U sledećoj tabeli označiti rezervisane reči

☐ `uses`	☐ `classic`	☐ `System`
☐ `Class`	☐ `static`	☐ `class`
☐ `ReservedWord`	☐ `void`	☐ `empty`

2. U sledećoj tabeli označiti one nizove simbola koji se mogu koristiti kao imena (promenljivih, klasa itd):

☐ `Vojvodina`	☐ `1234A`	☐ `Torta`
☐ `A*b`	☐ `NeSme`	☐ `Ne Sme`
☐ `A1234`	☐ `Program`	☐ `$xyz`
☐ `555-2313`	☐ `END`	☐ `1end`
☐ `enda`	☐ `writeln`	☐ `real`
☐ `Don't`	☐ `Dont`	☐ `ime+prezime`
☐ `AAA`	☐ `A1B2C3`	☐ `1a2b3c`

3. Ako promenljiva `x` ima vrednost 3, a promenljiva `y` vrednost 5, koju vrednost imaju sledeći izrazi:

`x /* + y */`	=	
`x /* + */ - y`	=	
`/* x */ y`	=	
`2 /* x */ + 3 /* y */`	=	
`x * /* x */ x`	=	
`3 /* x */ /* + */ /* 4y */`	=	
`/* - */ x * y`	=	
`5 * /* y * */ x`	=	

Glava 2

Numerički tipovi podataka

Računar je mašina koja je nastala kako bi nama, ljudima, olakšala *računanje*. Zato ne čudi što računari imaju čitav spektar mogućnosti da predstave razne vrste brojeva i da manipulišu njima. Programski jezik C#, kao i većina savremenih programskih jezika, razlikuje dve vrste brojeva:

- *realne brojeve*, što su brojevi koji mogu da imaju decimale i sa kojima smo se već sreli, i
- *cele brojeve*, što su brojevi koji nemaju decimale i sa kojima ćemo se sresti u ovom poglavlju.

Celi brojevi zaslužuju izdvojen status zato što zauzimaju manje mesta u memoriji i zato sto se sa njima lakše i brže računa.

U ovoj glavi ćemo se, između ostalog, upoznati i sa načinom na koji se u programskom jeziku C# formiraju algebarski izrazi. Problem sa formiranjem algebarskih izraza nastaje zato što C# ne poseduje specijalne simbole kao što je $\sqrt{\ }$ i zato što algebarski izraz mora biti "spakovan u liniju". Zato moramo biti pažljivi kada neki algebarski izraz zapisujemo u većini programskih jezika uključujući i C#: ponekad je potrebno dodati zagrade koje se u izrazu podrazumevaju!

2.1 Realni brojevi

U programskom jeziku C# postoji nekoliko tipova podataka koji se koriste za računanje sa realnim brojevima (`float`, `double` i `decimal`), ali ćemo mi koristiti isključivo tip `double`. Ova tri tipa se razlikuju samo po tome koji skup realnih brojeva mogu da predstave. Ime tipa `float` potiče od načina za reprezentaciju realnih brojeva koji se na engleskom zove *floating point* što znači brojevi sa pokretnim zarezom. Ime je motivisano činjenicom da je $125,3 = 12,53 \cdot 10^1 = 1,253 \cdot 10^2$ (da li

vidite kako se zarez slobodno kreće desno-levo, uz odgovarajuću korekciju eksponenta?). Ime tipa `double` potiče od *double precision floating point*, što znači da se i dalje radi o brojevima sa pokretnim zarezom, ali sa udvostručenom preciznošću u odnosu na `float`.

Promenljiva tipa `double` može da zapamti jedan realan broj kao što su:

```
1.4142    -0.618    2341.0    -26
```

☞ *Primetite još jednom da se brojevi koji imaju decimale pišu koristeći* ***decimalnu tačku,*** *prema engleskom pravopisu, a ne decimalni zarez kako nalaže naš pravopis!*

U fizici i tehnici se veoma često javljaju brojevi u tzv. *eksponencijalnoj* notaciji kao što su, recimo, brojevi $-0.61 \cdot 10^7$ i $0.2 \cdot 10^{-13}$. U programskom jeziku C# se "... puta deset na ..." piše pomoću jednog `e` (od *eksponent*). Primeri su dati pored.

Fizika i tehnika	$\rightarrow$	C#
$1,45 \cdot 10^{12}$	$\rightarrow$	`1.45e12`
$-0,61 \cdot 10^{7}$	$\rightarrow$	`-0.61e7`
$0,2 \cdot 10^{-13}$	$\rightarrow$	`0.2e-13`
$-12,65 \cdot 10^{-6}$	$\rightarrow$	`-12.65e-6`

2.2 Operacije sa realnim brojevima

Programski jezik C# poznaje sledeće algebarske operacije nad realnim brojevima:

Operacija	Objašnjenje	Primer
`+`	sabiranje	`x + 1.14`
`-`	oduzimanje	`x - 1.14`
	promena znaka	`-x + 2`
`*`	množenje	`x * y`
`/`	deljenje	`x / 1.036`

dok se u biblioteci `Math` nalaze razne matematičke funkcije od kojih navodimo nekoliko:

Operacija	Objašnjenje	Primer
`Math.Sqrt`	kvadratni koren	`Math.Sqrt(x)`
`Math.Abs`	apsolutna vrednost	`Math.Abs(x - 5.0)`
`Math.Min`	manji od dva broja	`Math.Min(x, y)`
`Math.Max`	veći od dva broja	`Math.Max(x, y)`

Ime funkcije `Sqrt` potiče od engleskog *square root* = kvadratni koren; ime funkcije `Abs` potiče od engleskog *absolute value* = apsolutna vrednost; za imena poslednje dve funkcije je jasno.

Kao i u matematici, operacije `*` i `/` imaju veći prioritet od operacija `+` i `-`. To znači da će se prvo izvršiti operacije množenja i deljenja, pa tek onda operacije sabiranja i oduzimanja. Evo primera:

C# izraz	Odgovara izrazu
`1 - x / y`	$1-\frac{x}{y}$
`x * y - 4`	$xy-4$
`1.12 + x / y - 3 * z`	$1,12+\frac{x}{y}-3z$

Operacije sabiranja i oduzimanja imaju isti prioritet, što je i očekivano i ovo ne dovodi do čudnih situacija. Operacije množenja i deljenja imaju takođe isti prioritet, što je i očekivano, *ali ovo može da dovede do čudnih situacija.* Na primer:

C# izraz	Tačno	**Pogrešno**
`x / y * z`	$\frac{x}{y}\cdot z$	$\frac{x}{y\cdot z}$
`x / y / z`	$\frac{\frac{x}{y}}{z}=\frac{x}{yz}$	$\frac{x}{\frac{y}{z}}$

Zagrade se koriste kao što smo i navikli. Ako nismo sigurni kako će C# prevodilac shvatiti neki izraz postavljanjem zagrada možemo obezbediti da se izraz računa tačno onako kako želimo. Na primer:

C# izraz	Odgovara izrazu
`(1.12 + x) / (y - 3 * z)`	$\frac{1,12+x}{y-3z}$
`x / (y * z)`	$\frac{x}{yz}$

Problem sa svim programskim jezicima je u tome što algebarski izraz mora biti "spakovan u jednu liniju". Zato moramo biti pažljivi kada neki algebarski izraz zapisujemo u programskom jeziku: *ponekad je potrebno dodati zagrade koje se u izrazu podrazumevaju!* Primeri:

Izraz	C# tačno	C# **pogrešno**
$\frac{x-1}{2-x}$	`(x - 1) / (2 - x)`	`x - 1 / 2 - x`
$\frac{x}{3z}$	`x / (3 * z)`	`x / 3 * z`
$\frac{1-\frac{x}{y}}{3z-2}$	`(1 - x/y) / (3*z - 2)`	`1 - x/y / (3*z - 2)`

Pogledajmo još nekoliko primera upotrebe funkcija `Abs` i `Sqrt`:

Izraz	C#
$\sqrt{1-\frac{x}{y}+z}$	`Math.Sqrt(1 - x/y) + z`
$\sqrt{\frac{\lvert x-1\rvert}{x+1}}$	`Math.Sqrt(Math.Abs(x - 1) / (x + 1))`
$\frac{2-x}{\lvert y-1\rvert}$	`(2 - x) / Math.Abs(y - 1)`
$\frac{\sqrt{2-x}}{\lvert y-1\rvert}$	`Math.Sqrt(2 - x) / Math.Abs(y - 1)`

Programski jezik C# poznaje još neke standardne matematičke funkcije koje ćete učiti kasnije. Zato sada navodimo samo spisak bez detaljnih objašnjenja.

C#	Matematika	Ime
`Math.Sin(x)`	$\sin x$	sinus
`Math.Cos(x)`	$\cos x$	kosinus
`Math.Tan(x)`	$\mathrm{tg}\, x$	tangens
`Math.Atan(x)`	$\mathrm{arctg}\, x$	arkus tangens
`Math.Exp(x)`	e^x	$(e = 2.71\ldots)$
`Math.Log(x)`	$\ln x$	prirodni logaritam
`Math.Log10(x)`	$\log_{10} x$	logaritam sa osnovom 10

Primer. Napisati C# program koji računa vrednost izraza

$$\frac{x+y}{x-y},$$

gde su x i y realni brojevi.

⟨C# fajl⟩

```
using System;

class Izraz {
  static void Main() {
    double x, y, rez;
    Console.WriteLine("Unesi x");
    x = double.Parse(Console.ReadLine());
    Console.WriteLine("Unesi y");
    y = double.Parse(Console.ReadLine());
    rez = (x + y) / (x - y);
    Console.WriteLine("Rezultat je {0}", rez);
  }
}
```

Ponekad se u većem izrazu neki manji izraz javi nekoliko puta. Rad programa se ubrzava ako se taj podizraz izračuna posebno i smesti u neku pomoćnu promenljivu, koju potom pozivamo u većem izrazu.

Primer. Napisati C# program koji računa vrednost izraza

$$\frac{-(x^2+xy+y^2)+2z}{x^2+xy+y^2-z}$$

gde su x, y i z realni brojevi.

⟨C# fajl⟩

```
using System;

class Izraz {
  static void Main() {
    double x, y, z, d, rez;
    Console.WriteLine("Unesi x");
    x = double.Parse(Console.ReadLine());
    Console.WriteLine("Unesi y");
    y = double.Parse(Console.ReadLine());
    Console.WriteLine("Unesi z");
    z = double.Parse(Console.ReadLine());
    d = x * x + x * y + y * y;
    rez = (-d + 2 * z)/(d - z);
    Console.WriteLine("Rezultat je {0}", rez);
  }
}
```

Kviz.

1. Koju vrednost će imati promenljive `x`, `m`, `n` nakon izvršavanja sledećih naredbi:

(a) `x = 5.5; x = x + 3.5;`

(b) `m = 35; m = m + 1; m = m - 32;`

(c) `m = 5; n = 6; m = m*n; n = n*m;`

2. Koju vrednost će imati promenljive `a`, `b`, `c` nakon izvršavanja sledećih naredbi:

```
a = 1; b = 2; c = 3;
a = b; b = c; c = a;
```

3. Koju vrednost će imati promenljive `x`, `y` nakon izvršavanja sledećih naredbi:

```
x = 5; y = 7;
t = x; x = y; y = t;
```

4. Šta će biti vrednosti promenljivih `x` i `y` nakon što se izvrši niz naredbi koji je naveden pored, kada je
(a) `x = 3, y = 7;`
(b) `x = 9, y = 2`?
Šta radi taj fragment?

```
x = x + y;
y = x - y;
x = x - y;
```

Zadaci.

2.1. (Rastojanje tačaka) Napisati C# program koji računa rastojanje dve tačke u ravni. Ako tačka A ima koordinate (x_1, y_1), a tačka B koordinate (x_2, y_2), onda je rastojanje tačaka A i B dato sa

$$d(A,B) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

⟨LINK⟩

2.2. Napisati C# program koji od korisnika učitava tri tačke u ravni date svojim koordinatama i potom računa i štampa obim tog trougla.

2.3. Površina trougla datih temena ⟨LINK⟩

2.4. Napisati program koji računa intenzitet sile privlačenja dva tela čije mase su m_1 i m_2, a koja se nalaze na rastojanju r. Dakle, od korisnika treba učitati realne brojeve m_1, m_2 i r i izračunati F koristeći se poznatim Njutnovim obrascem

$$F = \gamma \frac{m_1 m_2}{r^2},$$

gde je $\gamma = 6.67 \cdot 10^{-11}$ (nekih jedinica).

2.5. U tački $A(x_1, y_1)$ se nalazi telo mase m_1, a u tački $B(x_2, y_2)$ se nalazi telo mase m_2. Napisati C# program koji od korisnika učitava brojeve x_1, y_1, m_1, x_2, y_2 i m_2 i potom računa i ispisuje intenzitet sile provlačenja ova dva tela.

2.6. Napisati C# program koji računa vrednost sledećih funkcija:

$$(a)\quad \frac{\min(x,y) + 0.5}{1 + \min(x,y)^2}; \qquad (b)\quad \frac{\min(x,y) + 0.5}{1 + \max(x,y)^2}.$$

2.7. Napisati C# program koji od korisnika učitava tri broja i određuje i štampa najmanji od njih.

2.8. Napisati C# program koji od korisnika učitava pet brojeva i određuje i štampa najveći od njih.

2.9. Napisati C# program koji računa vrednost sledećeg izraza:

$$\frac{1}{\dfrac{-x}{x^2+y^2+1} + \dfrac{y}{x^2+y^2+1}} \cdot \sqrt{x^2+y^2+1}.$$

2.10. Trougao određen pravom i koordinatnim početkom ⟨LINK⟩

2.11. Biciklista ⟨LINK⟩

2.12. Sustizanje automobila ⟨LINK⟩

2.13. Rastojanje kuća ⟨LINK⟩

2.14. Kolona ⟨LINK⟩

2.15. Braća i pas ⟨LINK⟩

2.16. Prosek na takmičenju ⟨LINK⟩

2.3 Tip int

U programskom jeziku C# postoji nekoliko celobrojnih tipova (`byte`, `short`, `int`, `long`, ...), ali ćemo mi koristiti samo tip `int` (od engleskog *integer* što znači "ceo broj"). Promenljiva tipa `int` može da zapamti jedan ceo broj iz skupa

$$\{\texttt{-2147483648}, \ldots, \texttt{-1}, \texttt{0}, \texttt{1}, \ldots, \texttt{2147483647}\}.$$

Programski jezik C# poznaje sledeće operacije nad celim brojevima:

Operacija	Objašnjenje	Primer
`+`	sabiranje	`x + 14`
`-`	oduzimanje	`x - 27`
`-`	promena znaka	`-x + 2`
`*`	množenje	`x * y`
`/`	celobrojno deljenje	`x / 7`
`%`	ostatak pri deljenju	`x % 7`
`Math.Abs`	apsolutna vrednost	`Math.Abs(x - 2)`

Kada se operacija `/` primeni na realne brojeve rezultat deljenja je realan broj (dakle broj koji može da ima decimale), ali kada se ista operacija primeni na cele brojeve rezultat deljenja je *celobrojni količnik* ta dva broja. Na primer:

C# izraz	Rezultat	Tip
`9.0 / 4.0`	`2.25`	`double`
`9 / 4`	`2`	`int`
`9.0 / 4`	`2.25`	`double`
`9 / 4.0`	`2.25`	`double`

Poslednja dva primera iz gornje tabele ukazuju na jednu važnu osobinu aritmetičkih izraza u C#: ako izraz sadrži i realne i celobrojne vrednosti rezultat će biti realna vrednost. Na primer:

C# izraz	Rezultat	Tip
`9.0 + 4.0`	`13.0`	`double`
`9 + 4`	`13`	`int`
`9.0 + 4`	`13.0`	`double`
`9 + 4.0`	`13.0`	`double`

Primer. Otac ima d dece, $d \geqslant 2$. Jednog dana je kupio kesicu u kojoj je bilo $k \geqslant 2$ bombona i podelio ih je deci tako da sva deca dobiju najviše što može, ali svi isti broj bombona. Bombone koje su mu preostale je pojeo on. Napisati C# program koji učitava d i k i određuje koliko bombona je dobilo svako dete, a koliko bombona je pojeo otac.

⟨C# fajl⟩

```
using System;

class TataIBombone {
  static void Main() {
    int d, k, b_deca, b_otac;
    Console.WriteLine("Koliko ima dece?");
    d = int.Parse(Console.ReadLine());
    Console.WriteLine("Koliko ima bombona?");
    k = int.Parse(Console.ReadLine());
    b_deca = k / d;
    b_otac = k % d;
    Console.WriteLine("Svako dete je dobilo po {0} bombona", b_deca);
    Console.WriteLine("Ocu je ostalo {0} bombona", b_otac);
  }
}
```

Primetimo, prvo, da se celobrojne promenljive deklarišu na uobičajeni način, s tim da se kao tip navede `int`:

```
int d, k, b_deca, b_otac;
```

Dalje, celi brojevi se učitavaju isto kao realni brojevi:

```
k = int.Parse(Consple.ReadLine());
```

Drugim rečima, prvo naredbom `Console.ReadLine()` učitamo niz simbola koga posle toga parsiramo kao ceo broj komandom `int.Parse`

Kviz.

1. Pronađi bar jednu kombinaciju parametara za koju ocu ne ostane nijedna bombona.

2. Pronađi bar jednu kombinaciju parametara za koju otac dobije više bombona nego svako od njegove dece.

Primer. Napisati program koji od korisnika učitava pozitivan ceo broj n i potom računa i štampa zbir poslednje tri cifre broja n.

⟨C# fajl⟩

```
using System;

class Poslednje3Cifre {
  static void Main() {
    Console.WriteLine("Unesi n");
    int n = int.Parse(Console.ReadLine());
    int c0 = n % 10; n = n / 10;
    int c1 = n % 10; n = n / 10;
    int c2 = n % 10;
    int z = c0 + c1 + c2;
    Console.WriteLine("Zbir poslednje tri cifre je {0}", z);
  }
}
```

Ključni deo rešenja se sastoji u tome da uočimo da je

$$\boxed{\text{poslednja cifra broja } n} = n \ \% \ 10$$

i da je

$$\boxed{\text{broj } n \text{ bez poslednje cifre}} = n \ / \ 10$$

Dakle, `n % 10` je izraz koji "pročita" poslednju cifru broja `n`, dok je `n / 10` izraz koji "odgrize" poslednju cifru broja `n`. Sledeća tabela ilustruje rad programa kada je $n = 15384$:

n	$c2$	$c1$	$c0$
15384	–	–	–
1538	–	–	4
153	–	8	4
15	3	8	4

Primer. Na raspolaganju imamo kovanice od 1, 5 i 10 dinara. Program od korisnika učitava neki pozitivan ceo broj koji predstavlja količinu novca, i "isplaćuje" mu taj iznos koristeći najmanji mogući broj kovanica.

⟨C# fajl⟩

```
using System;

class RazmenaNovca {
  static void Main() {
    int n, k;
    Console.WriteLine("Unesi iznos");
    n = int.Parse(Console.ReadLine());

    // prvo apoeni od 10 din
    k = n / 10; n = n % 10;
    Console.WriteLine("{0} po 10 din", k);

    // potom apoeni od 5 din
    k = n / 5; n = n % 5;
    Console.WriteLine("{0} po 5 din", k);

    // ako je nesto ostalo isplacuje se novcicima od 1 din
    Console.WriteLine("{0} po 1 din", n);
  }
}
```

Kviz.

1. Kako program radi za sledeće iznose: 43, 96, 158, 189?
2. Koliko najviše kovanica od 1 din može biti "isplaćeno"? Navesti primer.
3. Koliko najviše kovanica od 5 din može biti "isplaćeno"? Navesti primer.

2.4 Konverzija numeričkih tipova

Videli smo da C# poznaje realne brojeve (tip `double`) i cele brojeve (tip `int`). Njihov odnos je isti kao odnos celih i realnih brojeva u matematici: kao što u matematici imamo da je $\mathbb{Z} \subset \mathbb{R}$ tako i u mnogim programskim jezicima među koje spada i C# imamo da je

$$\texttt{int} \subset \texttt{double}.$$

Zato unutar izraza možemo mešati celobrojne i realne vrednosti.

Prilikom rada sa celim brojevima zagrade se koriste na isti način kao pri radu sa realnim brojevima. Operatori `*`, `/` i `%` imaju isti prioritet, i on je viši od prioriteta koga imaju `+` i `-` (oduzimanje).

Ukoliko se u nekom algebarskom izrazu javljaju brojevi i realnog i celobrojnog tipa, svi brojevi se automatski konvertuju u brojeve realnog tipa i rezultat je realnog tipa. Primeri:

Izraz	Tip rezultata	Izraz	Tip rezultata
`10 * 2 + 9`	`int`	`11 + 0`	`int`
`10 * 2 + 9.12`	`double`	`11 + 0.0`	`double`
`12 * 3.71`	`double`	`Math.Sqrt(9)`	`double`
`12 / 4`	`int`	`Math.Abs(-9)`	`int`
`360 / 10 + 5`	`int`	`Math.Abs(-9.0)`	`double`

Promenljivoj tipa `double` možemo dodeliti i realne i celobrojne vrednosti. Pri tome će celobrojne vrednosti automatski biti konvertovane u ekvivalentan realan broj, na primer:

$$-5 \mapsto -5.0$$

Međutim, celobrojnoj promenljivoj *ne možemo* dodeliti realan broj. Evo jednog *primera gde kompajler prijavljuje grešku:*

```
int k = 3.75;
```

Razlozi za prijavljivanje greške su jasni: prevodilac ne zna šta da radi sa decimalama. Da li da ih odseče, ili da zaokruži broj? Kako kompajler ne sme da dopusti da dođe do gubitka informacija bez prethodne saglasnosti programera, od programera se očekuje da navede koji oblik konverzije želi.

Celobrojne i realne vrednosti i promenljive možemo eksplicitno konvertovati iz jednog tipa u drugi koristeći konstrukciju koja se zove *type cast* (što znači "upodobljavanje tipova"; "upodobljavanje" je samo jedno od mnogih značenja engleske reči *cast*). *Type cast* ima sledeći oblik:

$$\texttt{(}\langle tip \rangle\texttt{)}\ \langle izraz \rangle$$

na primer ovako:

```
(int)Math.Sqrt(a*a + b*b)
```

i znači da će se vrednost izraza konvertovati u navedeni tip. Šta se tu tačno dešava zavisi od slučaja do slučaja, a mi ćemo sada pokazati kako se C# ponaša prilikom konvertovanja u tip `int` i u tip `double`. Neka je:

```
int m = 15;
double y = -3.21;
```

Tada su sledeće dve naredbe korektne:

```
double x = (double)m;
int n = (int)y;
```

i vrednosti promenljivih su `x = 15.0` i `n = -3` zato što:

- prilikom konvertovanja celobrojne vrednosti u tip `double` ne treba donositi nikakve posebne odluke; mi želimo da bude $15 = 15.0$, pa se posao kompajlera sastoji samo u tome da se broj 15 iz mašinske reprezentacije celog broja konvertuje u mašinsku reprezentaciju realnog broja;
- prilikom konvertovanja realne vrednosti u tip `int` kompajler odseče decimale ($-3.21 \mapsto -3.0$) i onda se broj -3.0 konvertuje iz mašinske reprezentacije realnog broja u mašinsku reprezentaciju celog broja.

Pored toga, biblioteka `Math` ima četiri funkcije koje su namenjene tome da na razne načine petljaju sa decimalama realnih brojeva.

- Funkcija `Math.Round` kao argument prima realan broj, zaokružuje ga na najbliži ceo broj i vraća taj broj *kao realan broj tipa* `double`. U slučaju da je argument funkcije oblika $\langle$*ceo-broj*$\rangle + 0.5$ funkcija *zaokružuje na najbliži paran ceo broj*. Na primer,

x	2.49	2.50	3.50	2.51	-2.49	-2.50	-2.51
`Math.Round(`x`)`	2.00	2.00	4.00	3.00	-2.00	-2.00	-3.00

  Ovo je staro pravilo koje potiče iz računovodstva: zaokruživanjem na najbliži paran broj se neki brojevi zaokruže na malo više (recimo, 3.50), neki na malo manje (recimo, 2.50) i tako se u konačnom zbiru umanjuje ukupna greška zaokruživanja.

  Postoje, međutim, situacije u kojima želimo da broj oblika $\langle$*ceo-broj*$\rangle + 0.5$ uvek zaokružimo *naviše*, tako da se 2.50 zaokruži na 3 i da se 3.50 zaokruži na 4. U tom slučaju je potrebno pozvati funkciju `Math.Round` sa dodatnim argumentom:

```
Math.Round(x, MidpointRounding.AwayFromZero)
```

- Funkcija `Math.Truncate` kao argument prima realan broj, odseca njegove decimale i vraća tako dobijeni broj *kao realan broj tipa* `double`. Na primer,

x	2.49	2.50	2.51	-2.49	-2.50	-2.51
`Math.Truncate(`x`)`	2.00	2.00	2.00	-2.00	-2.00	-2.00

- Funkcija `Math.Floor` računa funkciju $\lfloor x \rfloor$, što je najveći ceo broj koji nije veći od x. Dakle, $\lfloor x \rfloor \leqslant x$. I ova funkcija vraća odgovarajuću vrednost tipa `double`. Na primer,

x	2.49	2.50	2.51	-2.49	-2.50	-2.51
`Math.Floor(`x`)`	2.00	2.00	2.00	-3.00	-3.00	-3.00

- Funkcija `Math.Ceiling` računa funkciju $\lceil x \rceil$, što je najmanji ceo broj koji nije manji od x. Dakle, $\lceil x \rceil \geqslant x$. I ova funkcija vraća odgovarajuću vrednost tipa `double`. Na primer,

x	2.49	2.50	2.51	-2.49	-2.50	-2.51
`Math.Ceiling(`x`)`	3.00	3.00	3.00	-2.00	-2.00	-2.00

Uočimo sledeće:

- Sve četiri funkcije imaju jedan argument tipa `double` i vraćaju vrednost tipa `double`, i
- svaka od njih preslikava celobrojne vrednosti (u realnom zapisu) u iste te vrednosti. Na primer,

  `Math.Round(2.0) = 2.0`, `Math.Truncate(2.0) = 2.0`,

  `Math.Floor(2.0) = 2.0`, `Math.Ceiling(2.0) = 2.0`.

Prema tome, ako želimo da zaokružimo realan broj i da dobijemo odgovarajuću vrednost tipa `int` moramo koristiti *type cast* jer `Math.Round` uvek vraća vrednost tipa `double`:

```
double x = 2.50;
int n = (int)Math.Round(x, MidpointRounding.AwayFromZero);
// sada je n = 3
```

Postoji još jedna situacija kod koje moramo biti veoma pažljivi. Šta mislite, šta ispisuje sledeći programski fragment:

```
double x = 1/1 + 1/2 + 1/3 + 1/4;
Console.WriteLine(x);
```

Odgovor je: 1. O čemu se radi? Svi brojevi koji se javljaju u izrazu

```
1/1 + 1/2 + 1/3 + 1/4
```

su celi. Prema tome, C# kompajler smatra da je to celobrojni izraz i onda svaku operaciju deljenja shvata kao *celobrojno deljenje*. Zato je $1/2 = 1/3 = 1/4 = 0$ jer kada delimo celobrojno (kao kod učiteljice) znamo da je

"jedan podeljeno sa dva jednako nula, ostatak 2".

Ako želimo da dobijemo odgovarajući realan broj potrebno je da bar jedan broj u svakom sabirku napišemo kao realan broj (dakle, da eksplicitno navedemo decimale), recimo ovako:

```
double x = 1.0/1 + 1.0/2 + 1.0/3 + 1.0/4;
Console.WriteLine(x);
```

ili da primenimo *type cast*:

```
double x = (double)1/1 + (double)1/2 + (double)1/3 + (double)1/4;
Console.WriteLine(x);
```

Primer. U severnoj Americi se visina osoba izražava u stopama i inčima. Jedna stopa iznosi 30,48 cm dok jedan inč iznosi 2,54 cm. Napisati C# program koji od korisnika učitava visinu u centimetrima kao decimalan broj i potom je preračunava u stope i inče. Visinu u stopama i inčima prikazati kao cele brojeve. (Jasno je da konverzija neće uvek biti tačna.)

⟨C# fajl⟩

```
using System;

class Visina {
  static void Main() {
    const double foot = 30.48; // cm
    const double inch =  2.54; // cm

    // u kodu koji sledi obratiti paznju na konstrukciju
    // deklaracija + dodela vrednosti!

    Console.WriteLine("Unesi visinu u cm");
    double visina_cm = double.Parse(Console.ReadLine());

    int visina_ft = (int)(visina_cm / foot);
    double ostalo_cm = visina_cm - visina_ft * foot;
    int visina_in = (int)(ostalo_cm / inch);

    Console.WriteLine("{0} cm = {1} ft {2} in",
                      visina_cm, visina_ft, visina_in);
  }
}
```

Ovaj primer zaslužuje pažnju jer smo u njemu iskoristili jednu važnu konstrukciju: moguće je u isto vreme deklarisati promenljivu i dodeliti joj vrednost! Do sada smo ovakve akcije pisali u dva poteza, prvo deklaracija, pa onda dodela vrednosti:

```
int n;
n = int.Parse(Console.ReadLine());
```

Kada nam to odgovara možemo postići isti efekat u jednom potezu:

```
int n = int.Parse(Console.ReadLine());
```

Na ovaj način dobijamo kôd koji se brže piše. Naravno, treba voditi računa da promenljivu smemo deklarisati samo jednom! Na primer,

```
int n = int.Parse(Console.ReadLine());
n = n / 10; // ovde ne treba ponovo pisati int!
Console.WriteLine("Bez poslednje cifre -> {0}", n);
```

Zadaci.

2.17. Trening ⟨LINK⟩

2.18. Razlomak u mešovit broj ⟨LINK⟩

2.19. Pravougaonik dat naspramnim temenima ⟨LINK⟩

2.20. Podela intervala ⟨LINK⟩

2.21. Kurs ⟨LINK⟩

2.22. Generator slučajnih brojeva ⟨LINK⟩

2.23. Grupa radnika ⟨LINK⟩

2.24. Taksi ⟨LINK⟩

2.25. GPS ⟨LINK⟩

2.26. Napisati C# program koji od korisnika učitava neki realan broj i potom određuje i štampa prve dve decimale tog broja.

2.27. Napisati C# program koji računa vrednost

(a) funkcije koja je data na Sl. 2.1 (a);

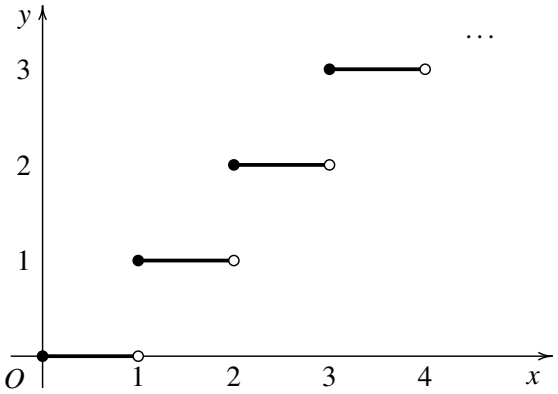
(b) funkcije koja je data na Sl. 2.1 (b);

Obe funkcije su definisane samo za $x \geqslant 0$. Program zato treba da učita neki realan broj $x \geqslant 0$, izračuna vrednost funkcije za to x i potom ispiše dobijenu vrednost. Zna se da će korisnik uneti nenegativan realan broj.

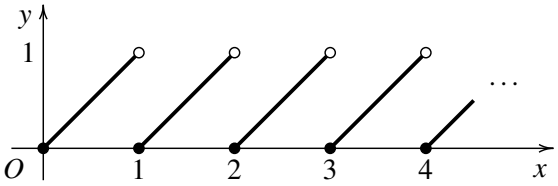
2.28. Šahovska tabla broj crnih polja ⟨LINK⟩

2.29. Podela jabuka ⟨LINK⟩

2.30. Pokloni ⟨LINK⟩



(*a*)



(*b*)

Slika 2.1: Funkcije iz Zadatka 2.27

2.31. Lift ⟨LINK⟩

2.32. Kruške ⟨LINK⟩

2.33. Sredina intervala ⟨LINK⟩

2.34. Zbir cifara ⟨LINK⟩

2.35. *Sudbinski broj* osobe je jednocifreni broj koji se računa na osnovu godine njenog rođenja na sledeći način: saberu se sve cifre u godini rođenja; ako se dobije broj koji ima više od jedne cifre, ponovo se saberu cifre dobijenog broja, i tako dalje, dok se ne dobije jednocifren broj. Na primer,

$$1990 \xrightarrow{1+9+9+0} 19 \xrightarrow{1+9} 10 \xrightarrow{1+0} 1,$$
$$1984 \xrightarrow{1+9+8+4} 22 \xrightarrow{2+2} 4,$$
$$2000 \xrightarrow{2+0+0+0} 2.$$

Napisati C# program koji od korisnika učitava ceo broj g, $1000 \leqslant g \leqslant$

9999, koji predstavlja godinu rođenja neke osobe i potom računa i ispisuje sudbinski broj te osobe.

2.36. Izbaci cifru stotina ⟨LINK⟩

2.37. Razmeni cifre ⟨LINK⟩

2.38. Ograda terase ⟨LINK⟩

2.39. Operacije po modulu ⟨LINK⟩

2.40. Ugao meren stepenima može se predstaviti pomoću tri celobrojne promenljive `d`, `m`, `s`, gde `d` sadrži broj stepeni, `m` broj minuta, a `s` broj sekundi. Ovakve tri promenljive predstavljaju korektan zapis mere ugla ako je $0 \leqslant d \leqslant 359$, $0 \leqslant m \leqslant 59$ i $0 \leqslant s \leqslant 59$.

Napisati C# program koji od korisnika učitava dva ugla α i β merena stepenima i određuje $\alpha + \beta$. Svi podaci su uneti u korektnom formatu i to ne treba proveravati.

2.41. Vreme završetka filma ⟨LINK⟩

2.42. Ugao satne kazaljke ⟨LINK⟩

2.43. Kuglice ⟨LINK⟩

2.44. Sportisti ⟨LINK⟩

2.45. Sijalice ⟨LINK⟩

2.46. Pozitivan deo intervala ⟨LINK⟩

2.47. Deo pravougaonika u prvom kvadrantu ⟨LINK⟩

2.48. Proja ⟨LINK⟩

2.49. Kraljevo rastojanje ⟨LINK⟩

2.50. (H2S04) Jedan molekul sumporne kiseline (H_2SO_4) se sastoji od dva atoma vodonika, jednog atoma sumpora i četiri atoma kiseonika. Napisati C# program koji od korisnika učitava cele brojeve `NH`, `NS` i `NO`, koji predstavljaju broj atoma vodonika, sumpora i kiseonika, tim redom, koji nam stoje na raspolaganju, i potom računa i štampa maksimalan broj molekula sumporne kiseline koji se mogu formirati od datih atoma. ⟨LINK⟩

2.51. Molekul etanola ima formulu C_2H_5OH. Napisati C# program koji od korisnika učitava cele brojeve `NC`, `NH` i `NO`, koji predstavljaju broj atoma ugljenika, vodonika i kiseonika, tim redom, koji nam stoje na raspolaganju, i potom računa i štampa maksimalan broj molekula etanola koji se mogu formirati od datih atoma.

2.52. Ponoć ⟨LINK⟩

2.53. Planinari ⟨LINK⟩

2.54. Deljivi oko broja ⟨LINK⟩

2.55. Sečenje pločica ⟨LINK⟩

2.56. Brojanje ovaca ⟨LINK⟩

2.57. Razbrajalica ⟨LINK⟩

2.58. Pertla ⟨LINK⟩

2.59. Točak ⟨LINK⟩

2.60. Kružna meta ⟨LINK⟩

2.61. Skaliranje ⟨LINK⟩

2.62. Dinari i pare ⟨LINK⟩

2.63. Trkači ⟨LINK⟩

2.64. Radijani ⟨LINK⟩

2.65. N-ti dan treninga ⟨LINK⟩

2.66. Sportske pripreme ⟨LINK⟩

2.67. Proizvodnja malina ⟨LINK⟩

2.68. Suma niza brojeva ⟨LINK⟩

2.69. Aritmetički trougao ⟨LINK⟩

2.70. Aritmetički kvadrat ⟨LINK⟩

2.71. Rastvor ⟨LINK⟩

2.72. Kredit ⟨LINK⟩

Glava 3

Grananje i logički tip

Algebarske operacije nisu definisane za sve vrednosti koje se mogu pojaviti kao njihovi argumenti. Na primer, ne možemo deliti nulom, niti u skupu realnih brojeva možemo odrediti koren iz negativnog broja. Potrebno je, dakle, kontrolisati tok programa i tako razdvojiti slučajeve u kojima je vrednost algebarskog izraza definisana od slučajeva u kojima algebarski izraz nije određen. Tome služi "if" kontrolna struktura. Upoznaćemo se i sa blokom kao sa posebnim načinom da se niz naredbi programskog jezika C# grupiše u jednu celinu.

3.1 "If" kontrolna struktura

Podsetimo se C# programa koji pomaže ocu da podeli bombone svojoj deci. Zadatak je glasio ovako:

> Otac ima d dece, $d \geqslant 2$. Jednog dana je kupio kesicu u kojoj je bilo $k \geqslant 2$ bombona i podelio ih je deci tako da sva deca dobiju najviše što može, ali svi isti broj bombona. Bombone koje su mu preostale je pojeo on. Napisati C# program koji učitava d i k i određuje koliko bombona je dobilo svako dete, a koliko bombona je pojeo otac.

Rešenje koje smo pokazali je izgledalo ovako:

⟨C# fajl⟩

```
using System;

class TataIBombone {
  static void Main() {
    int d, k, b_deca, b_otac;
    Console.WriteLine("Koliko ima dece?");
    d = int.Parse(Console.ReadLine());
```

```
    Console.WriteLine("Koliko ima bombona?");
    k = int.Parse(Console.ReadLine());
    b_deca = k / d;
    b_otac = k % d;
    Console.WriteLine("Svako dete je dobilo po {0} bombona", b_deca);
    Console.WriteLine("Ocu je ostalo {0} bombona", b_otac);
  }
}
```

Ako se korisnik lepo ponaša i unese razumne brojeve za broj dece i broj bombona u kesici program radi lepo. Ali šta se dešava ako zločesti korisnik unese da otac ima 0 dece? Tada prilikom računanja vrednosti promenljive `b_deca` dolazi do deljenja nulom i "program pukne":

```
Koliko ima dece?
0
Koliko ima bombona?
12
Unhandled Exception: System.DivideByZeroException:
   Attempted to divide by zero.
   at TataIBombone.Main()
```

Pošto nijedan pravi programer ne sme da dozvoli da mu program pukne, pre računanja vrednosti promenljive `b_deca` treba proveriti da li je korisnik uneo nulu kada ga je program upitao za broj dece. Ako jeste, programer treba da se pobuni, a ako nije onda se može preći na računanje. Zato nam treba način da proverimo da li su neka dva broja jednaka, odnosno, opštije, da li neke veličine stoje u nekim odnosima.

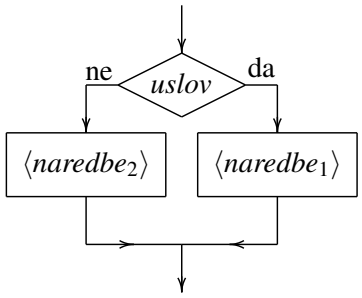
U programskom jeziku C# postoji poseban način da se utvrde odnosi među veličinama (da li su dva broja jednaka, različita, da li je neki od njih manji ili veći od onog drugog, manji ili jednak, veći ili jednak). Tabelica je data pored.

Matematika	C#	Primer
$=$	`==`	`x - y == 0`
$\neq$	`!=`	`x != y`
$<$	`<`	`x < 5`
$>$	`>`	`y > x + 2`
$\leqslant$	`<=`	`x <= 5`
$\geqslant$	`>=`	`y >= x + 2`

Svaki programski jezik ima način da se proveri da li je ispunjen uslov za izvršenje neke operacije. Taj mehanizam se zove *"if" kontrolna struktura*.

"If" kontrolna struktura se može javiti u raznim oblicima, a mi ćemo krenuti od najjednostavnijeg. Osnovni oblik "if" kontrolne strukture izgleda ovako:

```
if ( ⟨uslov⟩ ) {
     ⟨naredbe1⟩
}
else {
     ⟨naredbe2⟩
}
```



a tok programa koji sadrži "if" predstavljen je slikovito pored. Kada naiđe na "if" kontrolnu strukturu:

- program prvo proveri uslov;
- ako je uslov ispunjen, izvrši $\langle naredbe_1 \rangle$, preskoči $\langle naredbe_2 \rangle$ i nastavi sa radom;
- ako uslov nije ispunjen, preskoči $\langle naredbe_1 \rangle$, izvrši $\langle naredbe_2 \rangle$ i nastavi sa radom.

☞ *Ovo se zove kontrolna struktura zato što kontroliše tok programa: ako je ispunjen uslov, program radi na jedan način, a ako uslov nije ispunjen onda program radi na drugi način.*

Primer. U jednoj prodavnici odeće se za dva kupljena artikla dobija popust od 50% na jeftiniji. Milica je u toj radnji kupila pantalone i džemper. Napisati C# program koji učitava cenu pantalona i džempera koje je Milica kupila i potom računa koliko će Milica platiti.

⟨C# fajl⟩

```
using System;

class DvaArtikla {
  static void Main() {
    double pantalone, dzemper, platiti;
    Console.WriteLine("Unesi cenu pantalona");
    pantalone = double.Parse(Console.ReadLine());
    Console.WriteLine("Unesi cenu dzempera");
    dzemper = double.Parse(Console.ReadLine());
```

```
    if (pantalone < dzemper) {
      platiti = 0.5 * pantalone + dzemper;
    }
    else {
      platiti = pantalone + 0.5 * dzemper;
    }
    Console.WriteLine("Milica ce platiti {0} din", platiti);
  }
}
```

Primer. Ceo broj n je *potpun kvadrat* ako postoji ceo broj k takav da je $n = k^2$. Napisati C# program koji od korisnika učitava ceo broj n i utvrđuje da li je n potpun kvadrat.

⟨C# fajl⟩

```
using System;

class PotpunKvadrat {
  static void Main() {
    int n, k;
    Console.WriteLine("Unesi ceo broj");
    n = int.Parse(Console.ReadLine());
    k = (int)Math.Sqrt(n);
    if (k * k == n) {
      Console.WriteLine("Broj {0} je kvadrat: {1}*{1} = {0}", n, k);
    }
    else {
      Console.WriteLine("Broj {0} nije kvadrat", n);
    }
  }
}
```

Primer. Za neke vodoinstalaterske poslove idealan prečnik cevi je 60 mm sa tolerancijom od $\pm 1,25$ mm. Napisati C# program koji od korisnika učitava prečnik cevi koje su isporučene građevinskoj kompaniji i proverava da li se on uklapa u date parametre.

⟨C# fajl⟩

```
using System;

class IdealanPrecnik {
  static void Main() {
    double d;

    Console.Write("Unesi precnik cevi -> ");
    d = double.Parse(Console.ReadLine());
```

```
        if(Math.Abs(d - 60.0) <= 1.25) {
          Console.WriteLine("Cev odgovara standardu");
        }
        else {
          Console.WriteLine("Cev NE ODGOVARA standardu");
        }
      }
    }
```

Za kraj ćemo ponovo napisati program o tati koji deli bombone deci. U novoj varijanti programa delimo samo ako broj dece nije nula. U ostalim slučajevima *program sam* prijavi grešku. Kada učitamo podatke o broju dece i broju bombona prvo ćemo proveriti da li je korisnik uneo nulu za broj dece. Ako jeste prijavićemo grešku, a ako nije izračunaćemo koliko ko dobija bombona.

⟨C# fajl⟩

```
using System;

class TataIBombone2 {
  static void Main() {
    int d, k, b_deca, b_otac;
    Console.WriteLine("Koliko ima dece?");
    d = int.Parse(Console.ReadLine());
    Console.WriteLine("Koliko ima bombona?");
    k = int.Parse(Console.ReadLine());
    if(d == 0) {
      Console.WriteLine("Greska: ne moze se deliti nulom!");
    }
    else {
      b_deca = k / d;
      b_otac = k % d;
      Console.WriteLine("Svako dete je dobilo po {0} bombona", b_deca);
      Console.WriteLine("Ocu je ostalo {0} bombona", b_otac);
    }
  }
}
```

Niz naredbi ograđen vitičastim zagradama se zove *blok*:

$$
\begin{array}{l}
\{ \\
\quad \langle naredba_1 \rangle; \\
\quad \langle naredba_2 \rangle; \\
\qquad \vdots \\
\quad \langle naredba_k \rangle; \\
\}
\end{array}
$$

Naredbe koje se javljaju u bloku mogu, naravno, biti proizvoljne naredbe, pa i druge "if" kontrolne strukture. Na taj način se može postići finija kontrola programa što ćemo koristiti za računanje komplikovanijih izraza.

Zadaci.

3.1. Pera i Mika su brali jabuke. Pera je ubrao p, a Mika m jabuka. Napisati C# program koji proverava da li je Pera uspeo da nabere više jabuka nego Mika.

3.2. Napisati C# program koji proverava da li tačka $A(x_1, y_1)$ pripada krugu sa centrom u tački $C(x_0, y_0)$ čiji poluprečnik je $r > 0$.

3.3. Kvadratna jednačina je jednačina oblika $ax^2 + bx + c = 0$, gde su $a \neq 0$, b i c proizvoljni realni brojevi. Svaka kvadratna jednačina ima dva rešenja koja se računaju ovako:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{i} \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}.$$

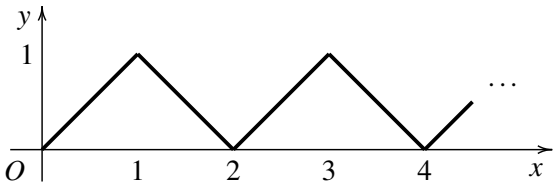
Napisati C# program koji od korisnika učitava relane brojeve a, b, c i računa rešenja kvadratne jednačine.

3.4. Za neke vodoinstalaterske poslove idealan prečnik cevi i tolerancija su propisni građevinskim standardima. Napisati C# program koji od korisnika učitava idealan prečnik cevi, dozvoljenu toleranciju, kao i prečnik cevi koje su isporučene građevinskoj kompaniji i proverava da li se isporučeni prečnik uklapa u standarde.

3.5. U jednoj kompaniji koja prodaje automobile agentima prodaje je dozvoljeno da preporučenu cenu automobila promene (spuste ili podignu, kako procene kupca) za najviše 5%. Napisati C# program koji od korisnika učitava preporučenu cenu automobila i cenu po kojoj je automobil prodat, i onda utvrđuje da li je agent prodaje poštovao politiku kuće o promeni cene za najviše 5%.

3.6. Milan je krenuo da kupuje nove patike i roditelji su mu dali d dinara. Kada je ušao u radnju video je da patike koje mu se sviđaju koštaju c dinara. Napisati C# program koji od korisnika učitava brojeve d i c i određuje da li Milan ima dovoljno novca da kupi patike koje su mu se svidele. Ako nema dovoljno novca da kupi patike koje su mu se svidele, proveriti da li će moći da ih kupi ako sačeka sezonu sniženja kada će cena tih patika biti umanjena za 10%.

3.7. Napisati C# program koji računa vrednost funkcije koja je data na Sl. 3.1. Funkcija je definisana samo za $x \geqslant 0$. Program zato treba da učita neki



Slika 3.1: Funkcija iz Zadatka 3.7

realan broj $x \geqslant 0$, izračuna vrednost funkcije za to x i potom ispiše dobijenu vrednost. U slučaju da je $x < 0$ program treba da ispiše poruku o greški.

3.8. Zbir godina braće i sestre ⟨LINK⟩

3.9. Jednakostranični trougao datog obima ⟨LINK⟩

3.10. Monopol ⟨LINK⟩

3.11. Trajanje vožnje ⟨LINK⟩

3.12. Čekanje ⟨LINK⟩

3.13. Ugao između kazaljki ⟨LINK⟩

3.14. Poklapanje kazaljki ⟨LINK⟩

3.15. Da li se dame napadaju ⟨LINK⟩

3.16. Ko je pre stigao ⟨LINK⟩

3.17. Punoletstvo ⟨LINK⟩

3.18. Sutrašnji datum ⟨LINK⟩

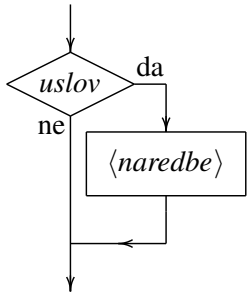
3.19. Jučerašnji datum ⟨LINK⟩

3.2 Razni oblici "if" kontrolne strukture

Osim osnovnog oblika, "if" kontrolna struktura se može javiti i u drugim oblicima koji nastaju ili skraćivanjem osnovnog oblika, ili nadovezivanjem nekoliko "if" struktura u osnovnom obliku.

Uslovno izvršavanje predstavlja skraćeni oblik "if" kontrolne strukture i izgleda ovako:

```
if ( ⟨uslov⟩ ) {
    ⟨naredbe⟩
}
```

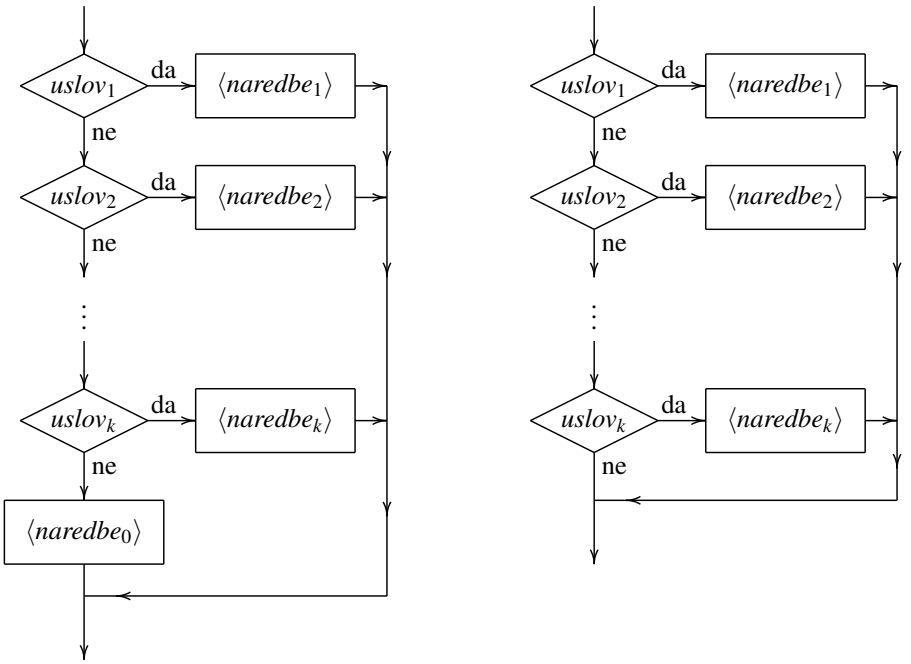


Kada naiđe na "skraćeni if", program se ponaša na sledeći način:

- proveri uslov;
- ako je uslov ispunjen izvrši ⟨*naredbe*⟩ i nastavi sa radom;
- ako uslov nije ispunjen, preskoči ⟨*naredbe*⟩ i nastavi sa radom.

Višestruki "if" u osnovnom obliku je prikazan pored. Računar proveri ⟨$uslov_1$⟩. Ako je on ispunjen, izvrši ⟨$naredbe_1$⟩, preskoči sve ostale spiskove naredbi i nastavi sa onim što sledi iza "if"-a. Ako ⟨$uslov_1$⟩ nije ispunjen, računar proveri ⟨$uslov_2$⟩. Ako je on ispunjen, izvrši ⟨$naredbe_2$⟩, preskoči sve ostale spiskove naredbi i nastavi sa onim što sledi iza "if"-a. I tako dalje. Ako nijedan uslov nije ispunjen, računar izvrši ⟨$naredbe_0$⟩.

```
if ( ⟨uslov_1⟩ ) {
    ⟨naredbe_1⟩
}
else if ( ⟨uslov_2⟩ ) {
    ⟨naredbe_2⟩
}
  ⋮
else if ( ⟨uslov_k⟩ ) {
    ⟨naredbe_k⟩
}
else {
    ⟨naredbe_0⟩
}
```



Slika 3.2: Višestruki "if" u osnovnom i skraćenom obliku

Višestruki "if" u skraćenom obliku je prikazan pored. Prilikom izvršavanja ove komande računar se ponaša na isti način kao u prethodnom slučaju. Jedina razlika je u tome da ukoliko nijedan uslov nije ispunjen, ne izvrši se ništa, već se nastavi sa radom od naredbe koja je iza "if"-a. Tok programa prilikom izvršavanja ova dva oblika "if" kontrolne strukture prikazan je na Sl. 3.2.

```
if ( ⟨uslov1⟩ ) {
    ⟨naredbe1⟩
}
else if ( ⟨uslov2⟩ ) {
    ⟨naredbe2⟩
}
  ⋮
else if ( ⟨uslovk⟩ ) {
    ⟨naredbek⟩
}
```

Primer. Milan i Dejan su rešili da unaprede moći svojih likova u njihovoj omiljenoj računarskoj igri. Milan je, zato, kupio *k spell*ova po ceni od *x* zlatnika za *spell*, a Dejan je kupio *n health*ova po ceni od *y* zlatnika za *health*. Napisati C# program koji od korisnika učitava brojeve *k*, *x*, *n* i *y*, i utvrđuje ko je potrošio više zlatnika i za koliko. Ukoliko su Milan i Dejan potrošili istu količinu zlata na unapređivanje svojih likova, program treba da ispiše "Isto".

⟨C# fajl⟩

```
using System;

class Zlatnici {
  static void Main() {
    double k, x, n, y, Milan, Dejan, razl;

    Console.Write("Koliko je Milan kupio spellova? ");
    k = double.Parse(Console.ReadLine());
    Console.Write("Koliko kosta jedan spell? ");
    x = double.Parse(Console.ReadLine());
    Console.Write("Koliko je Dejan kupio healthova? ");
    n = double.Parse(Console.ReadLine());
    Console.Write("Koliko kosta jedan health? ");
    y = double.Parse(Console.ReadLine());

    Milan = k * x;
    Dejan = n * y;

    if(Milan > Dejan){
      razl = Milan - Dejan;
      Console.WriteLine("Milan za {0}", razl);
    }
    else if(Milan < Dejan){
      razl = Dejan - Milan;
      Console.WriteLine("Dejan za {0}", razl);
    }
    else {
      Console.WriteLine("Isto");
    }
  }
}
```

Zadaci.

3.20. Sledeći programski fragment napisati pomoću samo jedne `if` naredbe:

```
if (a > b)  { c = 1; }
if (a > b)  { d = 2; }
if (a <= b) { c = 3; }
if (a <= b) { d = 4; }
```

3.21. Napisati ponovo program koji računa rešenja kvadratne jednačine (Zadatak 3.3). Ako je $b^2 - 4ac < 0$ ispisati poruku "Rešenja nisu realna". Ako je $b^2 - 4ac = 0$ tada su oba rešenja jednaka. Ispisati poruku "Rešenja su jednaka" i potom ispisati vrednost rešenja. Ako je $b^2 - 4ac > 0$ ispisati oba rešenja. U svim slučajevima rešenja zaokružiti na dve decimale.

3.22. (*Body Mass Index*) Indeks telesne težine za neku osobu se računa ovako:

$$\text{BMI} = \frac{\text{masa u kilogramima}}{(\text{visina u metrima})^2}.$$

Na osnovu ove vrednosti se težina osobe kategoriše prema sledećoj tabeli:

Kategorija	BMI
Neuhranjenost	$\text{BMI} < 18.5$
Idealna težina	$18.5 \leqslant \text{BMI} < 25.0$
Uvećana težina	$25.0 \leqslant \text{BMI} < 30.0$
Gojaznost	$\text{BMI} \geqslant 30.0$

Napisati C# program koji na osnovu težine (`t`) i visine (`v`) osobe određuje kategoriju kojoj pripada.

3.23. Dnevnice za službeni put se isplaćuju po sledećim pravilima:

- za svaka 24 časa provedena na putu se isplaćuje dnevnica u punom iznosu;
- ukoliko je nakon primene prvog pravila ostalo 12 ili više sati, za to vreme se takođe isplaćuje dnevnica u punom iznosu;
- ukoliko je nakon primene prvog pravila ostalo 8 ili više sati, ali manje od 12 sati, za to vreme se isplaćuje pola dnevnice;
- ukoliko je nakon primene prvog pravila ostalo manje od 8 sati, za to vreme se ne isplaćuje dnevnica.

Napisati C# program koji od korisnika učitava ceo broj h koji predstavlja broj sati provedenih na službenom putu, kao i decimalan broj d koji predstavlja visinu dnevnice, i potom računa i štampa količinu novca koju treba isplatiti na ime dnevnica.

3.24. Ugao meren stepenima može se predstaviti pomoću tri celobrojne promenljive `d`, `m`, `s`, gde `d` sadrži broj stepeni, `m` broj minuta, a `s` broj sekundi. Ovakve tri promenljive predstavljaju korektan zapis mere ugla ako je $0 \leqslant d \leqslant 359$, $0 \leqslant m \leqslant 59$ i $0 \leqslant s \leqslant 59$.

(a) Napisati C# program koji od korisnika učitava dva ugla α i β merena stepenima i utvrđuje da li je $\alpha > \beta$, $\alpha = \beta$ ili $\alpha < \beta$. Svi podaci su uneti u korektnom formatu i to ne treba proveravati.

(b) Napisati C# program koji od korisnika učitava dva ugla α i β merena stepenima i određuje $\alpha - \beta$. Svi podaci su uneti u korektnom formatu, zna se da je $\alpha > \beta$ i to ne treba proveravati.

3.25. (Progresivno oporezivanje) U jednoj državi se porez na zarade obračunava na sledeći način: na godišnje zarade koje su manje ili jednake od donje granice (`dgr`) se ne plaća nikakav porez; na godišnje zarade koje su strogo veće od donje granice (`dgr`) i manje su ili jednake od gornje granice (`ggr`) obračunava se porez po nižoj kamatnoj stopi (`nks`), ali samo na deo koji je iznad `dgr`; a na godišnje zarade koje su veće od gornje granice (`ggr`) porez se obračunava tako što se na deo zarade `ggr` − `dgr` porez obračunava po nižoj kamatnoj stopi (`nks`), dok se na deo zarade koji je iznad `ggr` porez obračunava po višoj kamatnoj stopi (`vks`).

Preciznije, ako je `z` godišnja zarada, onda se porez `p` obračunava ovako:

- ako je $\texttt{z} \leqslant \texttt{dgr}$ onda je $\texttt{p} = 0$;
- ako je $\texttt{dgr} < \texttt{z} \leqslant \texttt{ggr}$ onda je $\texttt{p} = \dfrac{(\texttt{z} - \texttt{dgr}) * \texttt{nks}}{100}$;
- ako je $\texttt{z} > \texttt{ggr}$ onda je $\texttt{p} = \dfrac{(\texttt{ggr} - \texttt{dgr}) * \texttt{nks} + (\texttt{z} - \texttt{ggr}) * \texttt{vks}}{100}$.

Napisati C# program koji od korisnika učitava parametre poreskog sistema `dgr`, `ggr`, `nks` i `vks`, potom zaradu na godišnjem nivou `z` nekog radnika, i onda računa i štampa iznos poreza `p`.

3.26. Agregatno stanje ⟨LINK⟩

3.27. Školarina ⟨LINK⟩

3.28. Uspeh učenika ⟨LINK⟩

3.29. Ocena na ispitu ⟨LINK⟩

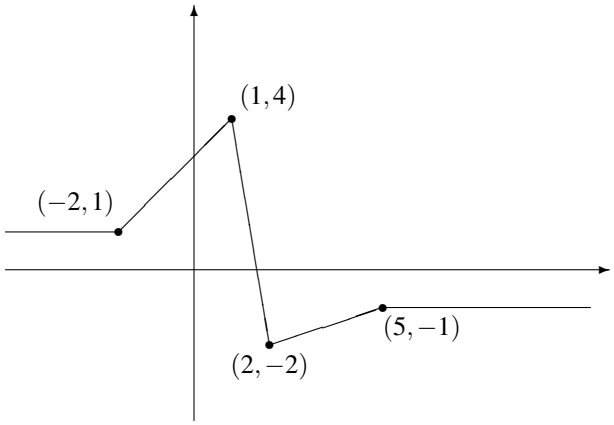
3.30. Solidarni porez ⟨LINK⟩

3.31. Odmor na pola puta ⟨LINK⟩

3.32. Rastojanje tačka–duž ⟨LINK⟩

3.33. Pravougaoni prsten ⟨LINK⟩

3.34. Napisati C# program koji od korisnika učitava realan broj x i računa vrednost funkcije $f(x)$ koja je data grafikom:



3.3 Logički tip i logički izrazi

Logički tip podataka se zove *boolean* i označava sa `bool`, a promenljiva logičkog tipa može da ima jednu od sledeće dve vrednosti: `true` (tačno), ili `false` (netačno). Tip `bool` programskog jezika C# je dobio ime po engleskom matematičaru Džordžu Bulu (George Boole), koji je živeo u drugoj polovini 19. veka.

Promenljive logičkog tipa koristimo kada treba da registrujemo da li se pojavio neki fenomen ili ne. One se ponekad zovu i *zastavice* (engl. *flags*) zato što možemo da "podignemo zastavicu" kada se desi neki događaj, ili da je "spustimo" ako se događaj nije desio.

Evo primera koji pokazuje kako se deklariše konstanta i promenljiva logičkog tipa. Vidimo da se promenljivoj logičkog tipa vrednost dodeljuje kao i svakoj drugoj promenljivoj.

```
const bool OK = true;
bool p, q;
p = OK;
q = false;
```

Ukoliko pokušate da logičkoj promenljivoj dodelite neki broj, ili bilo šta što nije logička vrednost, C# prevodilac će prijaviti grešku. Takođe, ukoliko pokušate promenljivoj koja nije deklarisana kao logička promenljiva da dodelite logičku vrednost, C# prevodilac će prijaviti grešku!

Dakle, ovo ne sme→

```
bool p;
p = 13.5; // GRESKA!
...
double x;
x = false; // GRESKA!
...
```

Često je potrebno ispitati da li važe dva uslova istovremeno pre nego što se pređe na računanje vrednosti nekog izraza. Da se ne bismo petljali sa ugnježdenim "if"-ovima, postoji način da se od dva ili više jednostavnih uslova napravi jedan složeni uslov: jednostavni uslovi se povežu se logičkim veznicima *konjunkcije* `&&`, *disjunkcije* `||` i *negacije* `!`.

Hajde da se prvo podsetimo matematičke logike. Sećate li se priče o iskaznim formulama i tautologijama? Imali smo iskazna slova, recimo p, q, r, ..., i logičke operacije $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$, pa smo od svega toga pravili iskazne formule (ili izraze), kao što je:

$$(\neg p \vee q) \wedge r \Rightarrow (q \wedge r).$$

Vrednosti logičkih promenljivih mogu da budu $\top$ ili $\bot$, pa kada nam neko zada konkretne vrednosti za p, q i r, možemo lako da izračunamo vrednost bilo kog logičkog izraza. Sličnu situaciju imamo i u programskom jeziku C#. Iskazna slova su logičke promenljive, dok složene logičke izraze možemo da pravimo koristeći logičke promenljive, proste uslove i logičke veznike. Evo nekoliko primera:

Matematika	$\mapsto$	C#
p	$\mapsto$	`p`
$\top$	$\mapsto$	`true`
$\bot$	$\mapsto$	`false`
$\wedge$	$\mapsto$	`&&`
$\vee$	$\mapsto$	`||`
$\neg$	$\mapsto$	`!`

Matematika	C#
$x < 5 \wedge y \geqslant 0$	`x < 5 && y >= 0`
$x < 5 \vee x > 9$	`x < 5 || x > 9`
$\neg(x > 3) \wedge y = 5$	`!(x > 3) && y == 5`
$(x < 2 \wedge y \neq 4) \vee (x > 3 \wedge y > 3)$	`x < 2 && y != 4 || x > 3 && y > 3`
$(x < 2 \vee y \neq 4) \wedge (x > 3 \vee y > 3)$	`(x < 2 || y != 4) && (x > 3 || y > 3)`

Kao što znamo još od priče sa operacijama na brojevima, nemaju sve operacije isti prioritet (na primer, množenje brojeva ima viši prioritet od sabiranja, tako da se

u izrazu $x+yz$ prvo računa yz, pa se na to doda x). Logički veznici u programskom jeziku C# takođe imaju različite prioritete. Veznik `!` ima najviši prioritet zato što je on unarni operator (napada samo jedan izraz). Osim toga, veznik `&&` ima viši prioritet od veznika `||` (`&&` se ponaša kao neka vrsta množenja, a `||` se ponaša kao neka vrsta sabiranja). Ako želimo da promenimo redosled dejstva logičkih veznika, slobodno možemo da koristimo zagrade, kako je to pokazano u poslednjim primeru iz prethodne tabele.

Posebno treba obratiti pažnju na sledeće situacije, kod kojih se (matematički korektan) produženi zapis mora predstaviti u obliku konjunkcije dva prosta izraza:

Matematika	C# ispravno	C# **neispravno**
$x = y = z$	`x == y && y == z`	`x == y == z`
$1 < x < 5$	`1 < x && x < 5`	`1 < x < 5`

Operatori poređenja `<`, `>`, `==`, `!=`, `>=` i `<=` programskog jezika C# se ponašaju kao logičke operacije. Svaki od njih uporedi date izraze i kao rezultat vrati `true` ako dati izrazi stoje u datom odnosu, odnosno, `false` ako ne stoje. Dakle,

```
(3 < 5) == true
(2 >= 9) == false
(2 + 2 == 5) == false
```

☞ *Ovde je potreban oprez! Ne smemo nikako pobrkati operator poređenja == i operator dodele = jer to može dovesti do grešaka koje kompajler neće primetiti i koje se teško ispravljaju!*

3.4 Složeni uslovi

Uslov koji se navodi u "if" kontrolnoj strukturi može biti "prost", kao u situacijama koje smo videli, ali može biti i "složen". Pravi oblik "if" kontrolne strukture, zato, izgleda kako je pokazano pored (i analogno za ostale varijante).

```
if (⟨logički izraz⟩) {
  ⟨naredbe1⟩
}
else {
  ⟨naredbe2⟩
}
```

☞ *Dakle, uslov u "if" strukturi može biti* proizvoljan logički izraz!

To može biti i samo jedna usamljena logička promenljiva, kao u primeru pored.

```
bool OK;
...
if (OK) { ... }
else { ... }
```

Primer. Jedna škola svake godine dodeljuje povelju primernog učenika. Učenik je primeran ako je završio školsku godinu kao odličan učenik, ako nema neopravdanih izostanaka i ako mu je ocena iz vladanja 5. Napisati C# program koji od korisnika učitava prosek učenika, broj neopravdanih izostanaka i ocenu iz vladanja i utvrđuje da li će taj učenik dobiti povelju.

⟨C# fajl⟩

```
using System;

class Povelja {
  static void Main() {
    double prosek, neopravdani_izost, vladanje;

    Console.Write("Prosek -> ");
    prosek = double.Parse(Console.ReadLine());
    Console.Write("Neopravdani izostanci -> ");
    neopravdani_izost = double.Parse(Console.ReadLine());
    Console.Write("Ocena iz vladanja -> ");
    vladanje = double.Parse(Console.ReadLine());

    if(prosek >= 4.5 && neopravdani_izost == 0 && vladanje == 5){
      Console.WriteLine("Dobija povelju");
    }
    else {
      Console.WriteLine("Ne dobija povelju");
    }
  }
}
```

Primer. Napisati C# program koji računa vrednost izraza $\sqrt{x-3}+\sqrt{|y|-z}$, gde su x, y i z realni brojevi.

⟨C# fajl⟩

```
using System;

class Izraz {
  static void Main() {
    double x, y, z, rez;
    Console.WriteLine("Unesi x");
    x = double.Parse(Console.ReadLine());
    Console.WriteLine("Unesi y");
```

```
    y = double.Parse(Console.ReadLine());
    Console.WriteLine("Unesi z");
    z = double.Parse(Console.ReadLine());

    if (x - 3 >= 0 && Math.Abs(y) - z >= 0) {
      rez = Math.Sqrt(x-3) + Math.Sqrt(Math.Abs(y)-z);
      Console.WriteLine("Rezultat je {0}", rez);
    }
    else {
      Console.WriteLine("Izraz nije definisan");
    }
  }
}
```

Primer. Da li su sledeća dva programska fragmenta ekvivalentna, odnosno, da li se za iste ulazne vrednosti ponašaju na isti način?

```
if (y != 0) {
  if (x/y >= 0) {
    rez = Math.Sqrt(x/y);
  }
}
```

```
if (y != 0 && x/y >= 0) {
  rez = Math.Sqrt(x/y);
}
```

Odgovor: DA!

☞ *Logički veznik && će izračunati vrednost desnog uslova samo ako je levi uslov tačan! Ako je levi uslov netačan, desni se neće računati jer je tada ceo izraz sigurno netačan.*

Ako je `y == 0` u drugom primeru, računar neće ni pokušati da proveri da li je `x/y >= 0`. Po analogiji,

☞ *logički veznik || će izračunati vrednost desnog uslova samo ako je levi uslov netačan! Ako je levi uslov tačan, desni se neće računati jer je tada ceo izraz sigurno tačan.*

Prema tome, ako u primeru

```
if (x == 5 || x / (x - 5) > 0) { ... }
```

znamo da je `x == 5`, desni argument operatora `||` se neće računati i zato program neće pući. Ovo je veoma važna konvencja koja štedi programerima mnogo vremena, kako ćemo videti kasnije.

Primer. Godine 1582. papa Grgur XIII je odlučio da modifikuje kalendar jer je primećeno da tadašnji kalendar, koga je bio ustanovio još Julije Cezar, nije više tačan. Tada je dogovoreno da se zadrži osnovni princip julijanskog kalendara, a da se samo promeni način računanja prestupnih godina. Naime, po julijanskom kalendaru svaka četvrta godina je bila prestupna. Ispostavilo se da je to previše, jer je julijanski kalendar za nekih 1500 godina koliko je bio na snazi kasnio oko 10 dana. Tako je dekretom kalendar pomeren 10 dana unapred i ustanovljeno je novo pravilo za računanje prestupnih godina:

- ako je godina deljiva sa 400 prestupna je;
- ako nije deljiva sa 400, ali je deljiva sa 100, onda nije prestupna;
- ako nije deljiva sa 100, ali je deljiva sa 4, onda je prestupna;
- a ako nije deljiva sa 4, onda nije prestupna.

Na primer: 2000. i 1968. godina su prestupne, dok 1900. i 1969. nisu.

Napisati C# program koji od korisnika učitava ceo broj iz skupa $\{1582, \ldots, 9999\}$ i utvrđuje da li je odgovarajuća godina prestupna prema pravilu za računanje prestupnih godina u gregorijanskom kalendaru.

Rešnje. Da bismo napisali program potrebno je proveriti da li je neki ceo broj deljiv nekim drugim celim brojem. Srećom, to se lako može proveriti:

$$\boxed{x \text{ je deljivo sa } y \text{ ako i samo ako je } x\%y = 0.}$$

Program izgleda ovako:

⟨C# fajl⟩

```
using System;

class PrestupneGodine {
  static void Main() {
    Console.WriteLine("Unesi godinu");
    int g = int.Parse(Console.ReadLine());
    if (g < 1582) { Console.WriteLine("Greska: Duboka proslost!"); }
    else if (g % 400 == 0) { Console.WriteLine("Prestupna"); }
    else if (g % 100 == 0) { Console.WriteLine("Nije prestupna"); }
    else if (g % 4 == 0) { Console.WriteLine("Prestupna"); }
    else { Console.WriteLine("Nije prestupna"); }
  }
}
```

Drugo rešenje:

⟨C# fajl⟩

```
using System;

class PrestupneGodine2 {
  static void Main() {
    Console.WriteLine("Unesi godinu");
    int g = int.Parse(Console.ReadLine());
    if (g < 1582) { Console.WriteLine("Greska: Duboka proslost!"); }
    else if (g % 400 == 0 || g % 100 != 0 && g % 4 == 0) {
      Console.WriteLine("Prestupna");
    }
    else { Console.WriteLine("Nije prestupna"); }
  }
}
```

Zadaci.

3.35. Napisati C# program kojim se proverava da li su dva učitana broja istog znaka.

3.36. Napisati C# program kojim se proverava da li dva učitane tačke $A(x_1, y_1)$ i $B(x_2, y_2)$ pripadaju istom kvadrantu.

3.37. Napisati C# program koji od korisnika učitava tri realna broja, a, b i c, i utvrđuje da li postoji trougao kome su to merni brojevi dužina stranica. Ukoliko takav trougao postoji, izračunati njegovu površinu koristeći Heronov obrazac $P = \sqrt{s(s-a)(s-b)(s-c)}$ gde je s poluobim trougla.

3.38. Napisati C# program koji od korisnika učitava tri realna broja, a, b i c, za koje znamo da predstavljaju merne brojeve dužina stranica nekog trougla (i to ne treba proveravati!), i određuje da li je taj trougao oštrougli, pravougli ili tupougli.

3.39. Napisati C# program koji od korisnika učitava koordinate (x, y) donjeg levog temena kvadrata čije stranice su paralelne koordinatnim osama, dužinu a stranice kvadrata kao i koordinate (px, py) tačke P, i utvrđuje da li se tačka P nalazi u unutrašnjosti kvadrata.

3.40. Napisati C# program koji od korisnika učitava koordinate (x, y) donjeg levog temena kvadrata čije stranice su paralelne koordinatnim osama, dužinu a stranice kvadrata kao i koordinate (px, py) tačke P za koju se zna da se nalazi izvan kvadrata, i određuje rastojanje tačke P od kvadrata. (Rastojanje tačke od kvadrata se računa tako što se nađe tačka Q na rubu kvadrata koja je najbliža tački P, pa se izračuna rastojanje tačaka P i Q.)

3.41. Tri atletičara su se takmičila u skoku u dalj. Svaki od njih je skakao tri puta, a kao konačan rezultat svakom takmičaru se računa najbolji skok. Napisati C# program koji za svakog od ova tri atletičara učitava ime i dužinu sva tri skoka, a onda određuje i štampa ime pobednika, kao i dužinu pobedničkog skoka.

3.42. U jednoj prodavnici se prodaju tri vrste čokolade. Za svaku vrstu čokolade znamo koliko grama ima svako pakovanje i koliko košta. Napisati C# program koji za svaku od ove tri vrste čokolade od korisnika učitava ime čokolade, masu i cenu pojedinačnog pakovanja, i onda određuje koja čokolada je najisplativija. Najisplativija je ona čokolada kod koje se dobija najviše čokolade za najmanje novca. Na primer, za sledeće podatke

Ime	Organska čok.	Crna čok.	Čok. sa lešnikom
Masa	80 g	100 g	90 g
Cena	319,99 din	389,99 din	495,99 din

najisplativija čokolada je crna čokolada.

3.43. Jedna transportna kompanija ima flotu od 25 kamiona istog proizvođača i istih karakteristika. Snabdevač gorivom ovu kompaniju snabdeva sa tri vrste goriva. Analizirajući podatke kompanija je došla do zaključka da nisu sve tri vrste goriva istog kvaliteta, što znači da se potrošnja goriva razlikuje od jedne do druge vrste. Napisati C# program koji od korisnika za svaku od tri vrste goriva učitava ime goriva, prosečnu potrošnju u litrima na 100 km, i cenu jednog litra goriva, i onda određuje najekonomičnije gorivo. Najekonomičnije gorivo je ono koje postiže najmanju cenu na pređenih 100 km. Na primer, za sledeće podatke

Ime	DieselX	TurboD	DeXtra
Potrošnja (100 km)	45 ℓ	46 ℓ	47 ℓ
Cena za 1 ℓ	167,99 din	171,99 din	155,99 din

najekonomičnije gorivo je DeXtra.

3.44. Za neke limarske poslove koriste se pravougaone ploče lima čije idealne dimenzije su propisane građevinskim standardima. Za izvođenje radova prihvatljive su one čije dimenzije se od onih propisanih standardima ne razlikuju za više od 1%. Napisati C# program koji od korisnika učitava idealne dimenzije ploče, potom dimenzije ploče koja je isporučena građevinskoj kompaniji, i proverava da li se isporučena ploča uklapa u standarde uz toleranciju od 1%.

3.45. Napisati C# program koji od korisnika učitava tri cela broja, `d`, `m`, `g`, i pro-

verava da li oni predstavljaju korektan datum, pri čemu je `d` redni broj dana, `m` redni broj meseca, a `g` redni broj godine. (Uzeti da je $1582 \leqslant g \leqslant 9999$.)

Na primer, za `d = 26`, `m = 12`, `g = 1735` računar treba da prijavi da se radi o korektnom datumu, dok za `d = 31`, `m = 4`, `g = 2001` i `d = 30`, `m = 2`, `g = 1848` treba da prijavi da se radi o nekorektnom datumu.

3.46. Napisati C# program koji od korisnika učitava tri cela broja, `d`, `m`, `g`, za koje se zna da predstavljaju *korektan* datum, i utvrditi redni broj tog dana u godini.

Na primer, za `d = 1`, `m = 1`, `g = 1875` program treba da ispiše 1, dok za `d = 7`, `m = 10`, `g = 2000` program treba da ispiše 281.

3.47. Napisati C# program koji od korisnika učitava dva cela broja, `n` i `g` i onda određuje datum koji odgovara danu sa rednim brojem `n` u godini `g`. Znamo da je $1 \leqslant g \leqslant 9999$, kao i da je $1 \leqslant n \leqslant 365$ ako `g` nije prestupna godina, odnosno, $1 \leqslant n \leqslant 366$ ako je `g` prestupna godina; navedeni uslovi će sigurno biti ispunjeni i ne treba ih proveravati.

3.48. Milutin Milanković je 1923. godine predložio pravilo za određivanje prestupnih godina koje bi garantovalo da kalendar ostane tačan narednih 43000 godina. Milankovićevo pravilo glasi ovako:

- ako godina *nije sekularna* (tj. nije deljiva sa 100), ona je prestupna ako je deljiva sa 4, kao i kod julijanskog i gregorijanskog kalendara; na primer, 2004. godina bi bila prestupna, a 2005. ne bi bila prestupna;
- *sekularna godina* je prestupna ako *broj vekova* u toj godini pri deljenju sa 9 daje ostatak 2 ili 6; na primer

Godina	Broj vekova	Ostatak pri deljenju sa 9	Prestupna?
2000	20	2	da
2100	21	3	ne
2400	24	6	da
2700	27	0	ne

Napisati C# program koji od korisnika učitava ceo broj g, $g \geqslant 1923$, i proverava da li je ta godina prestupna prema Milankovićevom pravilu.

3.49. Pobednik u dve discipline ⟨LINK⟩

3.50. Kvadranti i ose ⟨LINK⟩

3.51. Položaj dve prave ⟨LINK⟩

3.52. Rastojanje tačka–pravougaonik ⟨LINK⟩

3.53. Vrsta trougla na osnovu uglova ⟨LINK⟩

Glava 4

Ciklusi

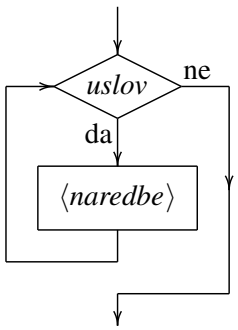
Pred nama je glava koja otvara vrata ka pravom programiranju jer se po prvi put srećemo sa cikličkim kontrolnim strukturama. Uvođenje ciklusa nam omogućuje da se uhvatimo u koštac sa ozbiljnijim programerskim problemima: videćemo kako se koriste brojači, kako se računaju jednostavnije sume i proizvodi i kako se računaju elementi rekurzivno zadatih nizova. Glavu zaključujemo diskusijom o ugnežđenim ciklusima.

4.1 "While" ciklus

Često se tokom programiranja javlja potreba da se jedan isti deo programa izvrši više puta (20 puta, ili n puta, gde je n vrednost neke promenljive, ili sve dok je neki uslov ispunjen). Kontrolne strukture koje omogućuju da se program zavrti zovu se *ciklusi* ili *petlje*. Programski jezik C# poznaje nekoliko vrsta ciklusa i sada ćemo se upoznati sa osnovnom vrstom ciklusa – "while" ciklusom.

"While" ciklus ima sledeći oblik:

```
while (⟨uslov⟩) {
    ⟨naredbe⟩
}
```



gde je ⟨*uslov*⟩ proizvoljan logički izraz. Spisak naredbi u "while" ciklusu zovemo *telo ciklusa*.

"While" ciklus radi na sledeći način:

sve dok je ispunjen navedeni uslov izvršavaj naredbe sa spiska.

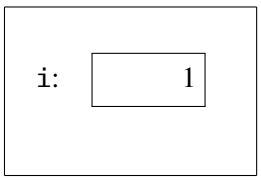
Primer. Program koji 10 puta ispisuje "Zdravo!" može se napisati ovako:

⟨C# fajl⟩

```
using System;

class ZdravoX10 {
  static void Main() {
    int i = 1;
    while(i <= 10) {
      Console.WriteLine("{0}. Zdravo!", i);
      i++;
    }
  }
}
```

Program radi ovako. Prvo promenljiva `i` dobije vrednost 1:

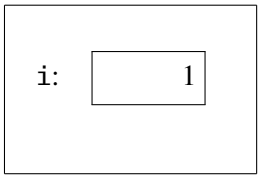
MEMORIJA:




```
int i = 1;
while(i <= 10) {
  Console.WriteLine("{0}. Zdravo!", i);
  i++;
}
```

nakon čega računar kreće sa izvršavanjem "while" ciklusa. "While" ciklus se izvršava tako što se prvo proveri uslov koji je naveden u zagradi:

MEMORIJA:




```
int i = 1;
while(i <= 10) {
  Console.WriteLine("{0}. Zdravo!", i);
  i++;
}
```

Pošto je uslov ispunjen izvršava se telo ciklusa (a to su naredbe oivičene vitičastim zagradama). Prvo se na monitor ispiše poruka:

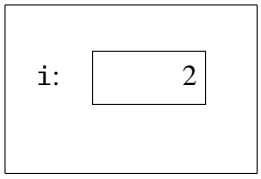
MONITOR:

```
1. Zdravo!
```

```
int i = 1;
while(i <= 10) {
  Console.WriteLine("{0}. Zdravo!", i);
  i++;
}
```

pa se vrednosti promenljive `i` uveća za 1:

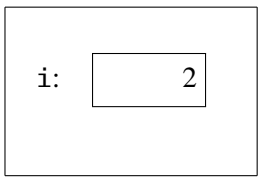
MEMORIJA:




```
int i = 1;
while(i <= 10) {
  Console.WriteLine("{0}. Zdravo!", i);
  i++;
}
```

Kada završi sa izvršavanjem naredbi iz tela "while" ciklusa računar se vraća na početak i ponovo proverava uslov koji je naveden u zagradi:

MEMORIJA:




```
int i = 1;
while(i <= 10) {
  Console.WriteLine("{0}. Zdravo!", i);
  i++;
}
```

Pošto je uslov ispunjen izvršava se telo ciklusa:

MONITOR:

```
1. Zdravo!
2. Zdravo!
```

```
int i = 1;
while(i <= 10) {
  Console.WriteLine("{0}. Zdravo!", i);
  i++;
}
```

Računar se opet vraća na početak i proverava uslov koji je naveden u zagradi:

MEMORIJA:

i: 3

```
int i = 1;
while(i <= 10) {
  Console.WriteLine("{0}. Zdravo!", i);
  i++;
}
```

Pošto je uslov ispunjen izvršava se telo ciklusa:

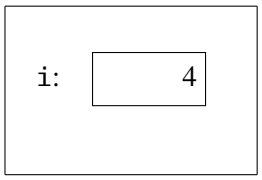
MONITOR:

```
1. Zdravo!
2. Zdravo!
3. Zdravo!
```

```
int i = 1;
while(i <= 10) {
  Console.WriteLine("{0}. Zdravo!", i);
  i++;
}
```

pa se vrednosti promenljive `i` uveća za 1:

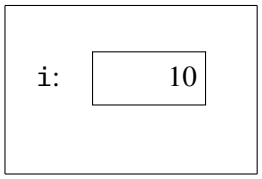
MEMORIJA:




```
int i = 1;
while(i <= 10) {
  Console.WriteLine("{0}. Zdravo!", i);
  i++;
}
```

i tako redom za `i = 4, 5, 6, 7, 8, 9, 10`. Pogledajmo šta se dešava kada promenljiva `i` dostigne vrednost 10. Računar proverava uslov "while" ciklusa:

MEMORIJA:




```
int i = 1;
while(i <= 10) {
  Console.WriteLine("{0}. Zdravo!", i);
  i++;
}
```

Pošto je uslov ispunjen izvršava se telo ciklusa:

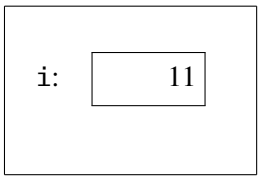
MONITOR:

```
1. Zdravo!
2. Zdravo!
3. Zdravo!
4. Zdravo!
5. Zdravo!
6. Zdravo!
7. Zdravo!
8. Zdravo!
9. Zdravo!
10. Zdravo!
```

```
int i = 1;
while(i <= 10) {
  Console.WriteLine("{0}. Zdravo!", i);
  i++;
}
```

pa se vrednosti promenljive `i` uveća za 1:

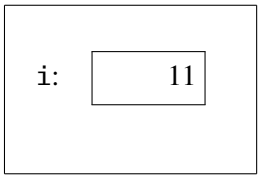
MEMORIJA:




```
int i = 1;
while(i <= 10) {
  Console.WriteLine("{0}. Zdravo!", i);
  i++;
}
```

Ovaj put kada se vratimo na početak ciklusa uslov *neće* biti ispunjen:

MEMORIJA:




```
int i = 1;
while(i <= 10) {
  Console.WriteLine("{0}. Zdravo!", i);
  i++;
}
```

Time se ciklus završava i stanje na monitoru je

MONITOR:

```
1. Zdravo!
2. Zdravo!
3. Zdravo!
4. Zdravo!
5. Zdravo!
6. Zdravo!
7. Zdravo!
8. Zdravo!
9. Zdravo!
10. Zdravo!
```

```
int i = 1;
while(i <= 10) {
  Console.WriteLine("{0}. Zdravo!", i);
  i++;
}
```

Primer. Program koji 10 puta ispisuje "Zdravo!", ali pri tome odbrojava od 10 do 1 izgleda ovako:

⟨C# fajl⟩

```
using System;

class ZdravoNanize {
  static void Main() {
    int i = 10;
    while(i >= 1) {
      Console.WriteLine("{0}. Zdravo!", i);
      i--;
    }
  }
}
```

Primer. Ispisati dečiju pesmicu "Ten Little Monkeys":

10 little monkeys were jumping on a bed
one fell off and bumped its head.
Mommy called the doctor and the doctor said:
'No more monkeys jumping on the bed!'

9 little monkeys were jumping on a bed
one fell off and bumped its head.
Mommy called the doctor and the doctor said:
'No more monkeys jumping on the bed!'

⋮

1 little monkey was jumping on a bed
it fell off and bumped its head.
Mommy called the doctor and the doctor said:
'Put those monkeys back to bed!'

⟨C# fajl⟩

```
using System;

class TenLittleMonkeys {
  static void Main() {
    int i = 10;
    while(i >= 2) {
      Console.WriteLine("{0} little monkeys were jumping on a bed", i);
      Console.WriteLine("one fell off and bumped its head.");
      Console.WriteLine("Mommy called the doctor and the doctor said:");
      Console.WriteLine("'No more monkeys jumping on the bed!'");
      Console.WriteLine();
      i--;
    }
    Console.WriteLine("1 little monkey was jumping on a bed");
    Console.WriteLine("it fell off and bumped its head.");
    Console.WriteLine("Mommy called the doctor and the doctor said:");
    Console.WriteLine("'Put those monkeys back to bed!'");
  }
}
```

4.2 "For" ciklus

"For" ciklus u programskom jeziku C# predstavlja kompaktan zapis "while" ciklusa. On je zgodan zato što se sve o čemu programer treba da brine navede odmah na početku i onda nema potrebe razmišljati da li smo zaboravili `i++` na kraju.

"For" ciklus izgleda ovako →

```
for(⟨inicijalizacija⟩;⟨uslov⟩;⟨inkrement⟩) {
    ⟨spisak naredbi⟩
}
```

i izvršava se ovako →

```
⟨inicijalizacija⟩;
while(⟨uslov⟩) {
    ⟨spisak naredbi⟩
    ⟨inkrement⟩
}
```

Na primer, ciklus levo predstavlja samo kompaktan zapis ciklusa desno:

```
for(i = 1; i <= n; i++) {
  Console.Write("Zdravo");
}
```

```
i = 1;
while(i <= n) {
  Console.Write("Zdravo");
  i++;
}
```

Takođe, ciklus levo predstavlja samo kompaktan zapis ciklusa desno:

```
for(i = n; i >= 1; i--) {
  Console.Write("Zdravo");
}
```

```
i = n;
while(i >= 1) {
  Console.Write("Zdravo");
  i--;
}
```

☞ *Uobičajeno je da se "for"-ciklus koristi kada treba da prođemo redom kroz sve elemente nekog niza vrednosti, recimo, kroz sve brojeve od 1 do n, ili kroz sve borojeve od n do 1. Upravo smo videli primere i za prvi i za drugi slučaj.*

Primer. Napisati C# program koji od korisnika učitava prirodan broj n i potom ispisuje prvih n kvadrata.

⟨C# fajl⟩

```
using System;

class PrvihNKvadrata {
  static void Main() {
    int i, n;
    Console.WriteLine("Unesi n");
    n = int.Parse(Console.ReadLine());
    Console.WriteLine("Prvih {0} kvadrata su:", n);
    for(i = 1; i <= n; i++) {
      Console.Write("{0} ", i * i);
    }
    Console.WriteLine();
  }
}
```

Primer. Napisati C# program koji odbrojava od 10 do 0 prilikom lansiranja rakete.

⟨C# fajl⟩

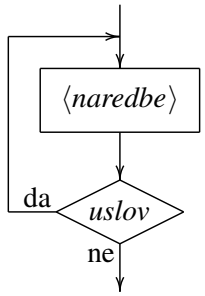
```
using System;

class Lansiranje {
  static void Main() {
    int i;
    Console.WriteLine("Odbrojavanje pocinje!");
    for(i = 10; i >= 0; i--) {
      Console.WriteLine(i);
    }
    Console.WriteLine("Poletanje!");
  }
}
```

4.3 "Do-while" ciklus

"Do-while" ciklus je ciklus sličan "while' ciklusu, s tim da se uslov proverava nakon izvršavanja bloka naredbi.

"Do-while" ciklus ima sledeći oblik:

```
do {
    ⟨naredbe⟩
} while(⟨uslov⟩);
```



gde je ⟨*uslov*⟩ proizvoljan logički izraz. "Do-while" ciklus radi na sledeći način:

izvršava se navedeni spisak naredbi sve dok je navedeni uslov tačan.

Osnovna razlika između "while" i "do-while" ciklusa je u sledećem:

- kod "while" ciklusa uslov se proverava pre nego što izvršimo blok; zato se može desiti da se blok ne izvrši nijednom u slučaju da je uslov odmah na početku netačan;
- kod "do-while" ciklusa se prvo izvrši blok, pa se tek onda proverava uslov; dakle, navedene naredbe se izvrše bar jednom!

C# poznaje obe vrste ciklusa zato što je nekad zgodnije koristiti jedan, a nekad drugi. Ne postoji spisak pravila koji reguliše kada se koja vrsta ciklusa koristi. Štaviše, postoje problemi koji se mogu rešiti i na jedan i na drugi način. Ukus i navike programera odlučuju kada će upotrebiti koju vrstu ciklusa.

Primer. Napisati C# program koji Njutnovim (I. Newton) iterativnim postupkom računa približnu vrednost korena pozitivnog realnog broja.

Njutnov iterativni postupak je postupak kojim se izračunava niz brojeva

$$x_0, x_1, x_2, \ldots$$

na sledeći način:

$$x_0 = \frac{a}{2}, \;\; x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right).$$

Niz brojeva $x_0, x_1, x_2, \ldots$ se polako približava korenu broja a, ali nikada ne dostiže tačnu vrednost. Brojevi $x_0, x_1, x_2, \ldots$ se zovu *iteracije* algoritma. Na primer, računanje broja $\sqrt{100}$ ovim postupkom daje sledeći niz brojeva:

$$\begin{aligned}
x_0 &= \tfrac{100}{2} = 50 \\
x_1 &= \tfrac{1}{2}(x_0 + \tfrac{100}{x_0}) = \tfrac{1}{2}(50 + \tfrac{100}{50}) = 26 \\
x_2 &= \tfrac{1}{2}(x_1 + \tfrac{100}{x_1}) = 14.923076923076923076923076923 \\
x_3 &= \tfrac{1}{2}(x_2 + \tfrac{100}{x_2}) = 10.812053925455987311657414750 \\
x_4 &= \tfrac{1}{2}(x_3 + \tfrac{100}{x_3}) = 10.030495203889795034734384359 \\
x_5 &= \tfrac{1}{2}(x_4 + \tfrac{100}{x_4}) = 10.000046356507898011634982428 \\
x_6 &= \tfrac{1}{2}(x_5 + \tfrac{100}{x_5}) = 10.000000000107445793143744907 \\
x_7 &= \tfrac{1}{2}(x_6 + \tfrac{100}{x_6}) = 10.000000000000000000000577230
\end{aligned}$$

i tako dalje. Kako ne očekujemo da postupak dođe do broja 10 kao tačne vrednosti za $\sqrt{100}$ postavlja se pitanje kada da stanemo sa postupkom. Za potrebe ovog primera ćemo se zadovoljiti situacijom u kojoj se dve uzastopne iteracije razlikuju za manje od 10^{-4} što nam u ovom primeru garantuje da će postupak dati vrednost korena na četiri decimale.

⟨C# fajl⟩

```
using System;

class KorenNa4Decimale {
  static void Main() {
    double x1, x2, a;
    Console.WriteLine("Unesi a");
    a = double.Parse(Console.ReadLine());
    // pretpostavljamo da je a > 0 i to ne proveravamo
    x2 = a / 2;
    do {
      x1 = x2;
      x2 = 0.5*(x1 + a/x1);
    } while(Math.Abs(x2 - x1) >= 0.0001);
    Console.WriteLine("Koren je {0}", x2);
  }
}
```

☞ *"While" i "do-while" ciklusi ne moraju da se završe! Na primer, sledeća dva ciklusa*

```
int k = 1;
while (k > 0) {
  k++;
}
```

```
int k = 1;
do {
  k++;
} while(k != 0);
```

će se završiti tek kada nestane struje.

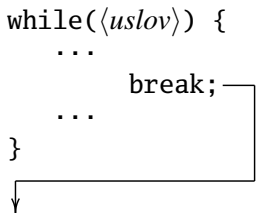
Zato uvek moramo imati nešto što raste (ili opada) i tako se približava uslovu za izlazak iz ciklusa. To je najčešće neka promenljiva čija vrednost raste (ili se smanjuje) u svakom prolazu kroz ciklus. Kadgod programer napiše ciklus obavezno mora da proveri da li će se ciklus završiti.

Na primer, u ciklusu pored vrednost promenljive k raste u svakom prolazu kroz ciklus. Kako smo krenuli od nule, ciklus će se *sigurno* završiti.

```
int k = 0;
do {
  k++;
  Console.WriteLine(k)
while (k != 10);
```

4.4 Naredbe "break" i "continue"

Postoje situacije u kojima želimo da naprasno prekinemo rad ciklusa ("while", "for" ili "do-while"), iako je uslov ciklusa ispunjen. Tome služi naredba "break" (engl. prekini [sa radom]). Kada naiđe na ovu naredbu unutar ciklusa računar prekida sa izvršavanjem ciklusa i nastavlja rad od prve naredbe koja je navedena *nakon* ciklusa:



Naredba "break" se na isti način ponaša u sve tri vrste ciklusa koje smo opisali.

Primer. Napisati C# program koji od korisnika učitava cele brojeve i za svaki od njih ispisuje da li je paran ili neparan. Program učitava brojeve sve dok korisnik ne unese reč KRAJ.

⟨C# fajl⟩

```
using System;

class ParNepar {
  static void Main() {
    while(true) {
      Console.WriteLine("Unesi broj");
      string s = Console.ReadLine();
      if(s == "KRAJ") { break; }
      int n = int.Parse(s);
      if(n % 2 == 0) { Console.WriteLine("Paran"); }
      else { Console.WriteLine("Neparan"); }
    }
  }
}
```

Ovaj zadatak zaslužuje mali komentar. Konstrukcija "while(true)" predstavlja *beskonačnu petlju* pošto je uslov ciklusa stalno ispunjen. Iz ovog ciklusa ne izlazimo tako što će uslov u nekom trenutku postati netačan, već pozivom naredbe "break" u telu ciklusa kada korisnik unese reč KRAJ.

Primer. Napisati C# program koji od korisnika učitava pozitivan ceo broj n i potom n puta ispisuje reč "ZDRAVO", ali ne više od 10 puta. Na primer, za $n = 7$ program će sedam puta ispisati reč "ZDRAVO", a za $n = 12$ program će deset puta ispisati reč "ZDRAVO".

⟨C# fajl⟩

```
using System;

class ZdravoMax10Puta {
  static void Main() {
    Console.WriteLine("Unesi n");
    int n = int.Parse(Console.ReadLine());
    for(int i = 1; i <= n; i++) {
      Console.WriteLine("Zdravo");
      if(i == 10) { break; }
    }
  }
}
```

Postoje situacije kada želimo da prekinemo redovno izvršavanje tela ciklusa, ali umesto da iskočimo iz ciklusa želimo da krenemo sa narednom iteracijom. To se postiže narednom "continue" (engl. nastavi [sa radom]). Kod "while" i "for" ciklusa naredba "continue" skoči na početak ciklusa i rad ciklusa se nastavlja na uobičajeni način ponovnom proverom uslova (ako je uslov ispunjen ponovo se izvršava telo ciklusa, a ako nije ciklus se završava):

```
while(⟨uslov⟩) {
    ...
    continue;
    ...
}
```

```
for(⟨init⟩; ⟨uslov⟩; ⟨inkrement⟩) {
    ...
    continue;
    ...
}
```

U slučaju "do-while" ciklusa naredba "continue" skoči na *kraj* ciklusa i rad ciklusa se nastavlja na uobičajeni način ponovnom proverom uslova (ako je uslov ispunjen ponovo se izvršava telo ciklusa, a ako nije ciklus se završava):

```
do {
    ...
    continue;
    ...
} while(⟨uslov⟩);
```

Primer. Na jednom turniru karatisti se takmiče u tri kategorije: juniori (takmičari od 7 do 12 godina), kadeti (takmičari od 13 godina do 20 godina) i seniori (takmičari od 21 godine na dalje). Napisati C# program koji od korisnika učitava cele brojeve (koji predstavljaju starost prijavljenih takmičara) sve dok korisnik ne unese reč KRAJ, i za svakog takmičara ispisuje kategoriju kojoj pripada. Ukoliko takmičar ima manje od 7 godina program treba da prijavi da takmičar nema pravo da se takmiči.

⟨C# fajl⟩

```
using System;

class JuniorKadetSenior {
  static void Main() {
    while(true) {
      Console.WriteLine("Unesi starost takmicara");
      string s = Console.ReadLine();
      if(s == "kraj") { break; }
      int g = int.Parse(s);
      if(g >= 21) {
        Console.WriteLine("Senior"); continue;
      }
      if(g >= 13) {
        Console.WriteLine("Kadet"); continue;
      }
      if(g >= 7) {
        Console.WriteLine("Junior"); continue;
      }
      Console.WriteLine("Nema pravo da se takmici");
    }
  }
}
```

Zadaci.

4.1. Brojevi od a do b ⟨LINK⟩

4.2. Brojanje u igri žmurke ⟨LINK⟩

4.3. Brojevi trocifreni parni ⟨LINK⟩

4.4. Brojevi deljivi sa 3 ⟨LINK⟩

4.5. Napisati C# program koji od korisnika učitava pozitivan ceo broj n i ispisuje sve njegove pozitivne delioce.

4.6. Podela intervala na jednake delove ⟨LINK⟩

4.7. Tabeliranje funkcije ⟨LINK⟩

4.8. Milje u kilometre ⟨LINK⟩

4.9. Najave emisije u pravilnim vremenskim intervalima ⟨LINK⟩

4.10. Ima li cifru ⟨LINK⟩

4.11. Geometrijska serija ⟨LINK⟩

4.12. Napisati C# program koji od korisnika učitava cele brojeve $d \geqslant 0$ i $n \geqslant 10$, i potom određuje i štampa sve brojeve iz skupa $\{10, 11, \ldots, n\}$ čiji zbir poslednje dve cifre je jednak sa d.

4.13. Napisati C# program koji od korisnika učitava pozitivan ceo broj p i potom određuje pozitivne cele brojeve a i b tako da to budu dužine stranica pravougaonika najmanjeg obima čija površina je p.

4.14. Deljiv brojevima od 1 do n ⟨LINK⟩

4.15. Napisati C# program koji za pozitivan ceo broj n određuje najveći broj k takav da je n deljiv sa 2^k.

4.16. Decimale broja 1/n ⟨LINK⟩

4.17. Napisati C# program koji učitava pozitivne cele brojeve m, n i k, i potom ispisuje količnik brojeva m i n na k decimala.

4.18. Napisati C# program koji štampa sve četvorocifrene brojeve $\overline{abcd}$ za koje je $\overline{abcd} = (a+b+c+d)^2$.

4.19. Napisati C# program koji od korisnika učitava prirodan broj k i potom određuje i štampa najmanji broj $n > k$ sa osobinom da je n deljivo sa 2, $n+1$ deljivo sa 3, $n+2$ deljivo sa 5 i $n+3$ deljivo sa 7.

4.20. Napisati C# program koji od korisnika učitava cele brojeve sve dok korisnik ne unese reč `KRAJ` i potom štampa najveći od njih. Na primer, za brojeve 2, 7, 2, 9, 7 program ispisuje 9.

4.21. Napisati C# program koji od korisnika učitava visine planinskih vrhova i onda određuje i štampa razliku u visini najvišeg i najnižeg unetog planinskog vrha. Visina planinskog vrha je pozitivan realan broj. Brojevi se unose sve dok korisnik ne unese reč `KRAJ`.

4.22. Napisati C# program koji od korisnika učitava nekoliko zatvorenih intervala $[a_1, b_1]$, $[a_2, b_2]$, …, $[a_n, b_n]$ na brojevnoj osi i potom određuje i štampa najmanji zatvoreni interval $[a^*, b^*]$ koji u sebi sadrži sve učitane intervale. Intervali se unose dok korisnik ne unese reč `KRAJ`.

4.23. *Gornje desno teme* skupa tačaka $\{A_1(x_1, y_1), A_2(x_2, y_2), \ldots, A_n(x_n, y_n)\}$

koje su date svojim koordinatama u ravni je tačka $A_i(x_i, y_i)$ tog skupa sa sledećom osobinom: kada kroz tačku A_i povučemo vertikalu, u otvorenoj desnoj poluravni u odnosu na tu vertikalu nema drugih tačaka skupa, niti na uočenoj vertikali iznad tačke A_i ima tačaka tog skupa.

Napisati C# program koji od korisnika učitava n, potom n tačaka u ravni koje su date svojim koordinatama (što su realni brojevi) i potom računa i štampa koordinate gornjeg desnog temena tog skupa tačaka.

4.24. Napisati C# program koji učitava stanje na računu neke kompanije, onda učitava niz realnih brojeva koji predstavljaju uplate na račun, odnosno, isplate sa računa (uplata je predstavljena pozivitnim brojem, a isplata negativnim) i na kraju štampa novo stanje računa. Brojevi se unose dok korisnik ne unese reč `KRAJ`. Pri tome stanje na računu ni u jednom trenutku ne sme da bude negativno, pa ako korisnik unese izmenu koja će račun dovesti u negativno stanje program treba odmah da prekine sa radom i upozori korisnika da se radi o nedozvoljenoj transakciji.

4.25. Parno neparni ⟨LINK⟩

4.26. Rastuće cifre ⟨LINK⟩

4.27. Maksimalna razlika susednih ⟨LINK⟩

4.28. Provera monotonosti ⟨LINK⟩

4.29. Testerast niz ⟨LINK⟩

4.30. Napisati C# program koji traži sve moguće načine da se dešifruje jednakost

$$(***)^2 = *00**,$$

gde zvezdice označavaju proizvoljne cifre. Pri tome, prva cifra u broju nije nula.

4.31. Napisati C# program koji ispisuje sve moguće načine da se dešifruje sabiranje

$$\begin{array}{cccc} & & Z & A \\ + & & D & A \\ \hline & T & A & K \end{array}$$

gde različitim slovima odgovaraju različite cifre, i pri tome $Z \neq 0$, $D \neq 0$ i $T \neq 0$.

4.32. Napisati C# program koji ispisuje sve moguće načine da se dešifruje sabiranje

$$\begin{array}{cccccc} & & B & O & R \\ + & & B & O & R \\ \hline & Š & U & M & A \end{array}$$

gde različitim slovima odgovaraju različite cifre, i pri tome $Š \neq 0$ i $B \neq 0$.

4.33. Napisati C# program koji ispisuje sve moguće načine da se dešifruje sabiranje

$$\begin{array}{cccccc} & U & Č & I & T & I \\ & U & Č & I & T & I \\ + & U & Č & I & T & I \\ \hline & L & E & Nj & I & N \end{array}$$

gde različitim slovima odgovaraju različite cifre, i pri tome $U \neq 0$ i $L \neq 0$.

4.34. U redu u samoposluzi se nalazi n kupaca. Kasirka potroši t_k vremenskih jedinica da bi opslužila kupca k, $k \in \{1, \ldots, n\}$. Kupac k na raspolaganju ima f_k vremenskih jedinica za čekanje u redu i plaćanje. Ako do završetka plaćanja ukupno protekne više od f_k jedinica, smatraćemo da je zakasnio. Napisati C# program koji od korisnika učitava prirodan broj n, potom n parova brojeva $(t_1, f_1), \ldots, (t_n, f_n)$ i za svakog kupca računa da li će zakasniti ili ne.

4.35. (Hi-Lo Game) Hi-Lo Game je igra u kojoj računar pogađa broj koga je zamislio korisnik. Korisnik zamisli broj između 1 i 10000, a računar pokušava da pogodi o kom se boju radi postavljanjem pitanja korisniku. Pitanja su oblika: "Da li je zamišljeni broj veći od ...?" Na svako pitanje korisnik odgovara sa "da" ili "ne". Napisati C# program koji igra Hi-Lo igru, i pogađa broj koga je zamislio korisnik sa najviše 14 pitanja. Na primer,

```
Zamislite neki broj izmedju 1 i 10000.
Da li je broj veci od 5000? ne
Da li je broj veci od 2500? ne
Da li je broj veci od 1250? da
Da li je broj veci od 1875? da
Da li je broj veci od 2187? ne
Da li je broj veci od 2031? ne
Da li je broj veci od 1953? da
Da li je broj veci od 1992? da
Da li je broj veci od 2011? ne
Da li je broj veci od 2001? ne
```

```
Da li je broj veci od 1996? da
Da li je broj veci od 1998? da
Da li je broj veci od 1999? da
Da li je broj veci od 2000? ne
Sada znam! Zamislili ste broj 2000.
```

4.5 Brojač

Brojač je celobrojna promenljiva koju u programu koristimo da prebrojimo nešto, na primer, delioce nekog broja, ili parne brojeve među brojevima koje je uneo korisnik. Brojač se deklariše kao i svaka druga celobrojna promenljiva. Pre svake upotrebe brojač mora da se *inicijalizuje* – postavi na početnu vrednost, najčešće 0. Vrednost brojača se potom može uvećati prema potrebi. Ako želimo da brojač k uvećamo za 1, recimo, to možemo učiniti naredbom k++.

Primer. Napisati C# program koji od korisnika učitava pozitivan ceo broj n i potom određuje i ispisuje broj njegovih delilaca.

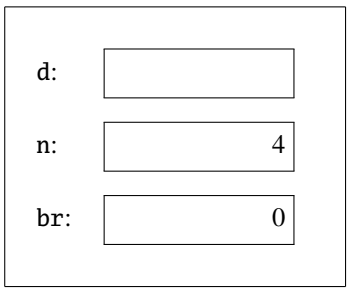
⟨C# fajl⟩

```
using System;

class BrojDelilaca {
  static void Main() {
    int n, br, d;
    Console.WriteLine("Unesi n");
    n = int.Parse(Console.ReadLine());

    br = 0;
    for(d = 1; d <= n; d++) {
      if (n % d == 0) { br++; }
    }
    Console.WriteLine("Broj delilaca -> {0}", br);
  }
}
```

U ovom programu promenljiva br broji delioce broja n i ona predstavlja *brojač*.

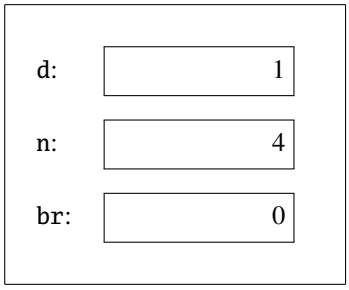
Pogledajmo sada detaljnije kako program radi. Prvo korisnik unese vrednost promenljive `n`, neka je to 4 za potrebe ovog primera, a onda se vrednost promenljive `br` postavi na 0.

MEMORIJA:



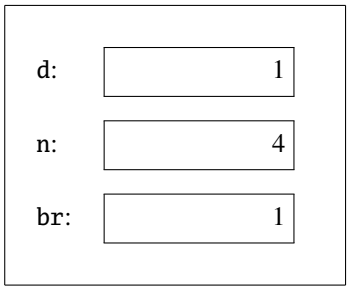
Sada krećemo sa izvršavanjem "for" ciklusa. Vrednost promenljive `d` se postavi na `1` zato što je to početna vrednost ciklusa:

MEMORIJA:



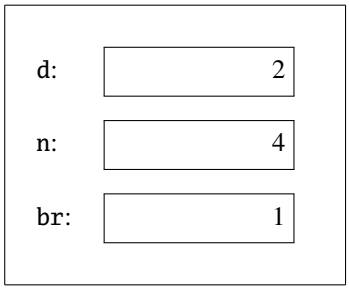
Potom proverimo da li je `d` delilac broja `n`. Pošto to jeste slučaj, vrednost promenljive `br` se uveća za `1`.

MEMORIJA:



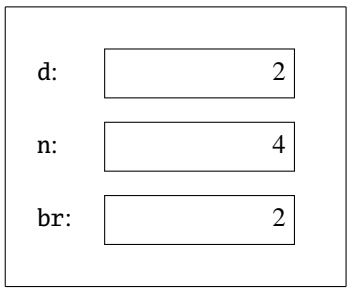
Na kraju ciklusa se vrednost promenljive `d` uveća za 1.

MEMORIJA:



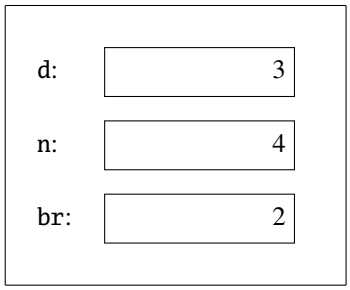
Ponovo proverimo da li je `d` delilac broja `n`. Pošto to i ovaj put jeste slučaj, vrednost promenljive `br` se uveća za `1`.

MEMORIJA:



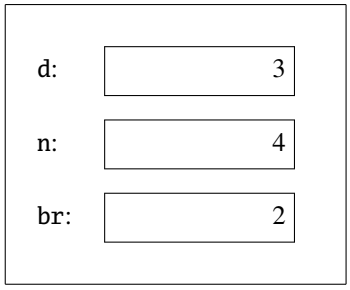
Na kraju ciklusa se vrednost promenljive `d` uveća za 1.

MEMORIJA:



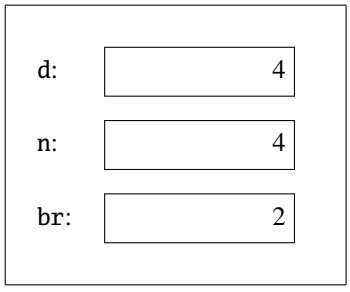
Ponovo proverimo da li je `d` delilac broja `n`. Pošto to sada nije slučaj, vrednost promenljive `br` se ne menja.

MEMORIJA:



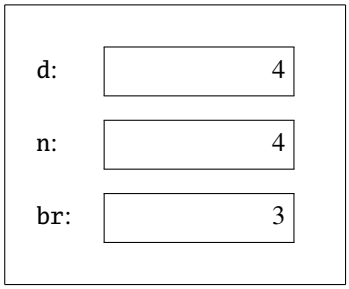
Na kraju ciklusa se vrednost promenljive `d` uveća za 1.

MEMORIJA:



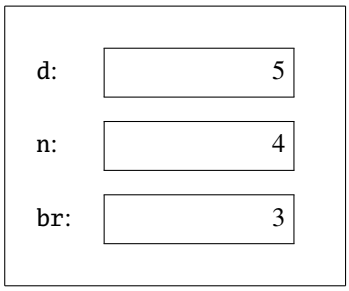
Ponovo proverimo da li je d delilac broja n. Pošto to jeste slučaj, vrednost promenljive br se uveća za 1.

MEMORIJA:



Na kraju ciklusa se vrednost promenljive d uveća za 1. Kako je sada d veće od n ciklus se završava i program ispisuje 3.

MEMORIJA:



Primer. Napisati program koji od korisnika učitava ceo broj n, potom n celih brojeva i utvrđuje koliko je među njima parnih.

⟨C# fajl⟩

```
using System;

class BrojDelilaca {
  static void Main() {
    int i, n, parnih, a;
    Console.WriteLine("Unesi n");
    n = int.Parse(Console.ReadLine());

    parnih = 0;
    for(i = 1; i <= n; i++) {
      Console.Write("{0}. broj -> ", i);
      a = int.Parse(Console.ReadLine());
      if (a % 2 == 0) { parnih++; }
    }
    Console.WriteLine("Parnih je bilo {0}", parnih);
  }
}
```

Primer. Napisati C# program koji utvrđuje koliko cifara ima dati prirodan broj.

⟨C# fajl⟩

```
using System;

class BrojCifara {
  static void Main() {
    int n, brCif;
    Console.WriteLine("Unesi n");
    n = int.Parse(Console.ReadLine());

    brCif = 0;
    while (n > 0) {
      n = n / 10;
      brCif++;
    }
    Console.WriteLine("Broj cifara -> {0}", brCif);
  }
}
```

Evo kako program radi: podeliti broj celobrojno sa 10 je isto što i skratiti ga za poslednju cifru. Zato, sve dok je broj pozitivan (tj. sve dok "ima cifara") radimo sledeće: delimo ga sa 10 i uvećavamo brojač cifara za jedan. Kada `n` padne na nulu to znači da smo mu "otkinuli" sve cifre jednu po jednu, i zato brojač `brCif` sadrži broj cifara broja `n`. Program radi korektno samo za pozitivne cele brojeve `n`. (Šta će program ispisati ako korisnik unese 0 ili negativan broj?)

Zadaci.

4.36. Napisati C# program koji od korisnika učitava pozitivan ceo broj n, a potom još n celih brojeva za koje proveri da li su svi pozitivni.

4.37. Napisati C# program koji od korisnika učitava ceo broj n, a potom još n celih brojeva za koje proveri da li je bar jedan veći ili jednak sa 5.

4.38. Napisati C# program koji od korisnika učitava ceo broj n, potom n celih brojeva i utvrđuje koliko je među njima neparnih.

4.39. Kategorije džudista ⟨LINK⟩

4.40. Prva negativna temperatura ⟨LINK⟩

4.41. Poslednja negativna temperatura ⟨LINK⟩

4.42. Napisati C# program koji od korisnika učitava ceo broj $n \geqslant 1$, potom n celih brojeva i određuje najveći od njih, kao i koliko puta se on pojavio u nizu. Na primer, za $n = 6$ i za brojeve 2, 9, 2, 9, 7, 9 program ispisuje 9 i 3.

4.43. Napisati C# program koji od korisnika učitava ceo broj $n \geqslant 1$, potom n celih brojeva i određuje najveći od njih, kao i

(a) mesto na kome se on prvi put pojavio u nizu (na primer, za $n = 6$ i za brojeve 2, 9, 2, 9, 7, 9 program ispisuje 9 i 2);

(b) mesto na kome se on poslednji put pojavio u nizu (na primer, za $n = 6$ i za brojeve 2, 9, 2, 9, 7, 9 program ispisuje 9 i 6).

4.44. Napisati C# program koji od korisnika učitava ceo broj $n \geqslant 1$, potom n celih brojeva i određuje najmanji i najveći od njih, kao i koliko puta su se najmanji i najveći broj pojavili u nizu. Na primer, za $n = 6$ i za brojeve 2, 9, 2, 9, 7, 9 program ispisuje 2, 2, 9, 3 zato što je 2 najmanji broj u nizu i pojavljuje se 2 puta, dok je 9 najveći broj u nizu i pojavljuje se 3 puta.

4.45. Napisati program koji od korisnika učitava n celih brojeva i utvrđuje koliko ima brojeva iza poslednjeg negativnog broja u nizu. Ako su svi brojevi u nizu nenegativni, program ispisuje n.

4.46. Napisati C# program koji učitava pozitivan ceo broj n, potom učitava n realnih brojeva i nalazi najduži strogo rastući segment u tom nizu. (Napomena: segment čine uzastopni elementi nekog niza.) Program i ispisuje dužinu segmenta i redni broj elementa od koga segment počinje. Na primer, za niz brojeva `1 1 2 1 2 3 1 2 3 4 1 2 3 4 5 1 2 3 4 5 6 0 1 2 3 4 5 6 7 0 1` program ispisuje `8 22` zato što je `0 1 2 3 4 5 6 7` najduži strogo rastući segment u tom nizu, njegova dužina je 8 i počinje na 22. mestu. Zna se da će niz imati bar tri elementa.

4.47. *Fiksna tačka* niza $a_1, a_2, \ldots, a_n$ je svaki broj i sa osobinom $a_i = i$. Napisati C# program koji od korisnika učitava ceo broj n, potom n celih brojeva i utvrđuje koliko elemenata učitanog niza su fiksne tačke tog niza.

4.48. (*Kolacova hipoteza*) Za proizvoljan prirodan broj n uradimo sledeće:

- ako je paran, podelimo ga sa 2;
- ako je neparan, pomnožimo ga sa 3 i dodamo 1.

Hipoteza teorije brojeva poznata pod imenom *Kolacova hipoteza* (L. Collatz) tvrdi da ćemo nakon konačno mnogo koraka doći do jedinice, ma od kojeg prirodnog broja da smo krenuli. Na primer:

$$\begin{aligned} 7 &\to 22 \to 11 \to 34 \to 17 \to 52 \to 26 \to 13 \to 40 \to 20 \to 10 \to \\ &\to 5 \to 16 \to 8 \to 4 \to 2 \to 1; \\ 113 &\to 340 \to 170 \to 85 \to 256 \to 128 \to 64 \to 32 \to 16 \to 8 \to \\ &\to 4 \to 2 \to 1. \end{aligned}$$

(a) Napisati C# program koji od korisnika učitava ceo broj $n \geqslant 2$ i potom utvrđuje u koliko koraka se dolazi do broja 1 počev od učitanog broja.

(b) Naći Kolac-rekordera do 100 000.

4.49. Dresirani žabac se nalazi na livadi. Kada dobije komandu "1" on skoči $1m$ na sever; kada dobije komandu "2" skoči $1m$ na istok; kada dobije komandu "3" skoči $1m$ na jug; a kada dobije komandu "4" skoči $1m$ na zapad.

Napisati C# program koji od korisnika učitava ceo broj n, potom n komandi (dakle, brojeve iz skupa $\{1,2,3,4\}$) i utvrđuje da li će se nakon tog niza komandi žabac naći u polaznoj tački.

4.50. Na beskonačnoj, idealno ravnoj platformi nalazi se robot koji razume sledeće tri komande:

- 1 — okreni se levo za 90° u mestu,
- 2 — okreni se desno za 90° u mestu,
- 3 — idi napred $1m$.

Napisati C# program koji od korisnika učitava ceo broj n, potom n komandi i utvrđuje da li se robot posle izvršenih n komandi našao na mestu na kome je stajao pre početka izvršavanja komandi.

4.51. Papir je paralelnim linijama izdeljen na jedinične kvadratiće. Nacrtan je krug sa centrom u preseku jednog para normalnih linija i poluprečnikom r. Napisati C# program koji od korisnika učitava pozitivan realan broj r i ispituje koliko jediničnih kvadratića potpuno leži u unutrašnjosti kruga.

4.52. Papir je paralelnim linijama izdeljen na jedinične kvadratiće. Nacrtan je kružni prsten sa centrom u preseku jednog para normalnih linija i poluprečnicima r i R. Napisati C# program koji od korisnika učitava pozitivne realne brojeve r i R, i ispituje koliko jediničnih kvadratića potpuno leži u unutrašnjosti prstena.

4.6 Sume i proizvodi

Veoma često se u programiranju javlja problem da se izračuna suma

$$a_1 + a_2 + \ldots + a_n$$

ili proizvod

$$a_1 \cdot a_2 \cdot \ldots \cdot a_n$$

gde su a_i neki brojevi. Na primerima ćemo pokazati kako se to radi. Prvo ćemo videti najjednostavniju ideju, a potom i jedan mali trik kojim se sume i proizvodi ponekad mogu inteligentnije izračunati.

Primer. Napisati C# program koji računa sumu

$$1^2 + 2^2 + 3^2 + \ldots + n^2.$$

⟨C# fajl⟩

```
using System;

class PrvaSuma {
  static void Main() {
    int n, i, sum;
    Console.WriteLine("Unesi n");
    n = int.Parse(Console.ReadLine());
    sum = 0;
    for(i = 1; i <= n; i++) {
      sum += i*i;
    }
    Console.WriteLine("Zbir je -> {0}", sum);
  }
}
```

U ovom programu se javlja jedna nova interesantna konstrukcija:

```
sum += i*i;
```

Ova naredba znači: “Na promenljivu `sum` dodaj vrednost izraza `i*i`”. Prema tome, tokom rada programa na pomoćnu promenljivu `sum` polako dodajemo sabirke jedan po jedan. Na kraju, promenljiva `sum` sadrži vrednost cele sume. Evo kako se menja stanje promenljive `sum` u raznim fazama rada programa:

Pre ulaska u petlju:	$\texttt{sum} = 0$
1. prolaz koz petlju:	$\texttt{sum} = 1^2$
2. prolaz koz petlju:	$\texttt{sum} = 1^2 + 2^2$
3. prolaz koz petlju:	$\texttt{sum} = 1^2 + 2^2 + 3^2$
$\vdots$	
n-ti prolaz koz petlju:	$\texttt{sum} = 1^2 + 2^2 + 3^2 + \ldots + n^2$

Kao što smo već rekli, naredba `sum += i*i` predstavlja kraći zapis naredbe `sum = sum + i*i` koja znači: "Nova vrednost promenljive `sum` se dobija tako što se na staru vrednost promenljive `sum` dodaje vrednost izraza `i*i`". Slično tome, naredba `k++;` koju smo ranije viđali je kraći zapis naredbe

```
k = k + 1;
```

U Tabeli 4.1 je dat pregled niza sličnih naredbi.

Primer. Napisati C# program koji za dati prirodan broj n računa faktorijel tog broja, što je proizvod svih prirodnih brojeva od 1 do n:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n.$$

⟨C# fajl⟩

```
using System;

class Faktorijel {
  static void Main() {
    int n, i, prod;
    Console.WriteLine("Unesi n");
    n = int.Parse(Console.ReadLine());
    prod = 1;
    for(i = 2; i <= n; i++) {
      prod *= i;
    }
    Console.WriteLine("n! = {0}", prod);
  }
}
```

Naredba	Isto kao	Značenje
`k = k + a;`		nova vrednost promenljive k se dobija tako što se na staru vrednost promenljive k doda a
`k += a;`	`k = k + a;`	uvećaj k za a
`k++;`	`k = k + 1;`	uvećaj k za 1
`k = k - a;`		nova vrednost promenljive k se dobija tako što se od stare vrednosti promenljive k oduzme a
`k -= a;`	`k = k - a;`	smanji k za a
`k--;`	`k = k - 1;`	smanji k za 1
`k = k * a;`		nova vrednost promenljive k se dobija tako što se stara vrednost promenljive k pomnoži sa a
`k *= a;`	`k = k * a;`	uvećaj k a puta
`k = k / a;`		nova vrednost promenljive k se dobija tako što se stara vrednost promenljive k podeli sa a
`k /= a;`	`k = k / a;`	smanji k a puta
`k = k % a;`		nova vrednost promenljive k je ostatak pri deljenju k sa a (samo kada su k i a celobrojne promenljive)
`k %= a;`	`k = k % a;`	(ne znam kako ovo lepo da opišem rečima, pa pustimo neka formule govore)

Tabela 4.1: Specijalne naredbe dodele

Primer. Napisati C# program koji za dati realan broj x i dati prirodan broj n računa x^n.

⟨C# fajl⟩

```
using System;

class StepenXN {
  static void Main() {
    int n, i;
    double x, stepen;
    Console.WriteLine("Unesi x");
    x = double.Parse(Console.ReadLine());
    Console.WriteLine("Unesi n");
    n = int.Parse(Console.ReadLine());
    stepen = 1.0;
    for(i = 1; i <= n; i++) {
      stepen *= x;
    }
    Console.WriteLine("{0}^{1} = {2}", x, n, stepen);
  }
}
```

Prethodni program je veoma jednostavan i lako se pamti, ali nije baš efikasan: ako neki realan broj x dižemo na stepen n, tada se "for" ciklus ovog programa izvršava n puta. Pokazaćemo sada i jedan komplikovaniji ali *neuporedivo efikasniji* način da realan broj x dignemo na stepen n. Odgovarajuća petlja novog programa će se izvršavati onoliko puta koliko cifara ima binarni zapis broja n. Na primer, ako pokušamo da odredimo x^{32000}, prvi, naivan algoritam će se vrteti u petlji 32000 puta, dok će se novi, efikasniji algoritam vrteti u petlji samo 15 puta!

Recimo da želimo da izračunamo x^{154}. Osnovna ideja novog algoritma se sastoji u tome da se izračunaju svi stepeni broja x oblika x^{2^k}:

$$x,\quad x^2,\quad x^4,\quad x^8,\quad x^{16},\quad x^{32},\quad x^{64},\quad x^{128},\ldots$$

i da se potom pomnože samo oni stepeni koji su nam potrebni. Kako je

$$154 = 10011010_{(2)} = 128+16+8+2$$

dobijamo da je $x^{154} = x^{128}\cdot x^{16}\cdot x^8\cdot x^2$. Zato algoritam uzastopnim kvadriranjem računa stepene broja x oblika x^{2^k} i u isto vreme konvertuje broj n iz osnove 10 u osnovu 2. Pri tome, kada naiđe na cifru 1 u binarnom zapisu broja n, neće je ispisati, nego će odgovarajući stepen broja x domnožiti[1] na promenljivu u kojoj se polako akumulira rezultat.

[1] ovo *nije* reč književnog srpskog jezika, ali nam je u ovom trenutku jako zgodna!

i	rez	x	n	b_i
0	1	x	154	0
1	$1 \cdot x^2$	x^2	77	1
2	x^2	x^4	38	0
3	$x^2 \cdot x^8$	x^8	19	1
4	$x^{10} \cdot x^{16}$	x^{16}	9	1
5	x^{26}	x^{32}	4	0
6	x^{26}	x^{64}	2	0
7	$x^{26} \cdot x^{128}$	x^{128}	1	1
8	x^{154}	x^{256}	0	

Tabela pored pokazuje kako donji algoritam radi za $n = 154$ i proizvoljan realan broj x.

```
// efikasno stepenovanje
// radi samo za n > 0

double x, rez;
int n;
x = double.Parse(Console.ReadLine());
n = int.Parse(Console.ReadLine());
rez = 1.0;
while(n > 0) {
  if(n % 2 == 1) { rez *= x; }
  x *= x;
  n /= 2;
}
Console.WriteLine(rez);
```

Pogledajmo, za kraj ovog odeljka, kako možemo da računamo zbirove kod kojih sabirci imaju komplikovanu, ali pravilnu strukturu, kao što je, recimo, ova:

$$1! + 2! + 3! + \ldots + n!$$

Program koji računa vrednost gornjeg izraza se može napisati veoma efikasno uz malo lukavstvo koje se sastoji u tome da sledeći sabirak računamo na osnovu prethodnog. U ovom slučaju je to moguće zato što je

$$\begin{aligned} k! &= k \cdot (k-1) \cdot (k-2) \cdot \ldots \cdot 1 \\ &= k \cdot (k-1)!. \end{aligned}$$

Program LukavaSuma računa sumu faktorijela na lukav način.

⟨C# fajl⟩

```
using System;

class LukavaSuma {
  static void Main() {
    int n, k, i, sum;
    Console.WriteLine("Unesi n");
    n = int.Parse(Console.ReadLine());
```

```
      sum = 1;
      k = 1;
      for(i = 2; i <= n; i++) {
        k *= i;
        sum += k;
      }
      Console.WriteLine("Suma -> {0}", sum);
    }
  }
```

Evo kako lukavstvo radi:

Pre ulaska u petlju:	$\mathtt{k} = 1!, \quad \mathtt{sum} = 1!$
1. prolaz koz petlju za $\mathtt{i} = 2$:	$\mathtt{k} = 2!, \quad \mathtt{sum} = 1+2!$
2. prolaz koz petlju za $\mathtt{i} = 3$:	$\mathtt{k} = 3!, \quad \mathtt{sum} = 1+2!+3!$
3. prolaz koz petlju za $\mathtt{i} = 4$:	$\mathtt{k} = 4!, \quad \mathtt{sum} = 1+2!+3!+4!$
$\vdots$	
poslednji prolaz koz petlju za $\mathtt{i} = n$:	$\mathtt{k} = n!, \quad \mathtt{sum} = 1+2!+3!+\ldots+n!$

☞ ***Opšta preporuka:*** *Kada treba izračunati sumu $a_1+a_2+\ldots+a_n$ pogledati prvo da li se a_k može jeftino dobiti od a_{k-1}. U tu svrhu proveriti da li je neki od ova dva izraza $a_k - a_{k-1}$ ili $\dfrac{a_k}{a_{k-1}}$ lep i jednostavan. Ako jeste, onda se može koristiti metod lukavog sumiranja. Ako nije, onda nam nema spasa ...*

Zadaci.

4.53. Napisati C# program koji od korisnika učitava prirodan broj n, potom n realnih brojeva $a_1, \ldots, a_n$ i računa

(a) njihovu aritmetičku sredinu $A = \dfrac{a_1+a_2+\ldots+a_n}{n}$;

(b) njihovu kvadratnu sredinu $K = \sqrt{\dfrac{a_1^2+a_2^2+\ldots+a_n^2}{n}}$;

(c) njihovu harmonijsku sredinu $H = \dfrac{n}{\dfrac{1}{a_1}+\dfrac{1}{a_2}+\ldots+\dfrac{1}{a_n}}$.

4.54. *Težište* konačnog skupa tačaka $A_1(x_1,y_1)$, $A_2(x_2,y_2)$, ..., $A_n(x_n,y_n)$ koje su date svojim koordinatama u ravni je tačka $T(x^*,y^*)$, gde je

$$x^* = \frac{x_1+x_2+\ldots+x_n}{n} \quad \text{i} \quad y^* = \frac{y_1+y_2+\ldots+y_n}{n}.$$

Napisati C# program koji od korisnika učitava n, potom n tačaka u ravni koje su date svojim koordinatama (što su realni brojevi) i potom računa i štampa koordinate težišta tog skupa tačaka.

4.55. *Pitagorin n-ugao* je n-ugao kod koga je kvadrat jedne stranice jednak zbiru kvadrata ostalih. Napisati C# program koji od korisnika učitava ceo broj $n \geqslant 3$ koji predstavlja broj strana n-ugla, potom n realnih brojeva koji predstavljaju dužine strana tog n-ugla, i onda određuje da li se radi o Pitagorinom n-uglu.

4.56. Napisati C# program koji od korisnika učitava prirodne brojeve n i k i potom ispisuje sumu poslednjih k cifara broja n.

4.57. Napiši C# program koji učitava realne brojeve sve dok korisnik ne unese reč `KRAJ` i potom računa i ispisuje njihov prosek.

4.58. Milan u dnevniku ima tri ocene iz informatike. Napiši C# program koji od korisnika učitava tri ocene koje Milan ima upisane u dnevnik, a onda određuje koliko petica Milan još treba da dobije iz informatike da bi nastavnik morao da mu zaključi pet za polugodište. (Nastavnik učniku mora da zaključi pet ako je prosek ocena upisanih u dnevnik 4,50 ili više.)

4.59. Učenici tokom školske godine dobijaju više ocena. Nastavnik je obećao da će prilikom zaključivanja ocena na kraju godine svakom učeniku zanemariti najslabiju ocenu (ako postoji više takvih, zanemariće samo jednu). Napiši C# program koji učitava ocene dok korisnik ne unese reč `KRAJ` i potom ispisuje prosečnu ocenu kada se zanemari najslabija.

4.60. U jednoj prodavnici je u toku akcija u kojoj se kupcima koji kupe dva ili više proizvoda nudi da najjeftiniji od njih dobiju za jedan dinar. Napiši C# program koji učitava cene proizvoda koje je kupila Nevena sve dok korisnik ne unese reč `KRAJ` i potom računa i ispisuje koliko će to sve koštati pod uslovima ove akcijske prodaje.

4.61. Prosečan rast cena ⟨LINK⟩

4.62. Jednakost rastojanja ⟨LINK⟩

4.63. Prosek skokova ⟨LINK⟩

4.64. Sredine ⟨LINK⟩

4.65. Čitanje do minus 1 ili do n-tog ⟨LINK⟩

4.66. Redni broj maksimuma ⟨LINK⟩

4.67. Najbliži datom celom broju ⟨LINK⟩

4.68. Broj maksimalnih ⟨LINK⟩

4.69. Drugi na rang listi ⟨LINK⟩

4.70. Druga vrednost po veličini ⟨LINK⟩

4.71. Pobednik u tri discipline ⟨LINK⟩

4.72. Najmanji krug ⟨LINK⟩

4.73. Razlika suma do max i od max ⟨LINK⟩

4.74. Prosek odličnih ⟨LINK⟩

4.75. Broj i zbir cifara broja ⟨LINK⟩

4.76. Transformacija broja u proizvod cifara ⟨LINK⟩

4.77. Najmanji broj sa najvećim zbirom parnih cifara ⟨LINK⟩

4.78. Broj formiran od datih cifra s leva na desno ⟨LINK⟩

4.79. Zameni cifre 0 sa 5 ⟨LINK⟩

4.80. Kombinacija dva broja minimumom i maksimumom odgovarajućih cifara ⟨LINK⟩

4.81. Armstrongov broj ⟨LINK⟩

4.82. Harmonijski pi ⟨LINK⟩

4.83. Za prirodan broj n sa $\sigma(n)$ označavamo sumu svih brojeva iz skupa $\{1,2,\ldots,n-1\}$ koji su delioci broja n. Napisati C# program koji od korisnika učitava ceo broj n i ako je $n \geqslant 3$ računa $\sigma(n)$.

4.84. Prirodan broj n je *savršen* ako je $n = \sigma(n)$ (videti Zadatak 4.83).

(a) Napisati C# program koji od korisnika učitava ceo broj $n \geqslant 3$ i utvrđuje da li je on savršen.

(b) Napisati C# program koji od korisnika učitava ceo broj $n \geqslant 3$ i ispisuje sve savršene brojeve iz skupa $\{3,4,\ldots,n\}$.

4.85. Prirodni brojevi m i n su *prijateljski brojevi* ako je $n = \sigma(m)$ i $m = \sigma(n)$ (videti Zadatak 4.83).

(a) Napisati C# program koji od korisnika učitava cele brojeve $m,n \geqslant 3$ i utvrđuje da li su oni prijateljski brojevi.

(b) Napisati C# program koji od korisnika učitava ceo broj $n \geqslant 3$ i ispisuje sve parove prijateljskih brojeva (a,b) takve da je $3 \leqslant a \leqslant b \leqslant n$.

4.86. Napisati C# program koji računa x^n za sve realne brojeve x i sve cele bro-

jeve n. Imati u vidu da 0^0 nije definisano, da je $x^0 = 1$ za $x \neq 0$ i da za negativne n imamo $x^n = \frac{1}{x^{|n|}}$.

4.87. Za prirodan broj n, broj $n!!$ se definiše ovako: ako je n paran, onda je $n!!$ proizvod svih parnih brojeva od 2 do n; ako je n neparan, onda je $n!!$ proizvod svih neparnih brojeva od 3 do n. Na primer, $8!! = 2 \cdot 4 \cdot 6 \cdot 8$, a $11!! = 3 \cdot 5 \cdot 7 \cdot 9 \cdot 11$.

Napisati C# program koji učitava ceo broj n i ako je $n > 0$ štampa $n!!$. Ako je $n \leqslant 0$ program ispisuje poruku o greški.

4.88. Verižni razlomak je razlomak koji ima oblik koji je naznačen pored, gde su a_k neki realni brojevi. Napisati C# program koji od korisnika učitava n, i onda računa verižni razlomak za niz realnih brojeva $a_0, \ldots, a_n$ koji se tim redom učitavaju od korisnika.

$$a_n + \cfrac{1}{a_{n-1} + \cfrac{1}{a_{n-2} + \cfrac{1}{\ddots \cfrac{}{a_1 + \cfrac{1}{a_0}}}}}$$

(Napomena: korisnik vodi računa da svi razlomci budu definisani.)

4.89. U kombinatorici se veoma često javlja sledeći broj:

$$\binom{n}{k} = \frac{n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n-k+1)}{k!}$$

koji predstavlja broj načina da se iz skupa od n objekata odabere k-elementni podskup. Napisati C# program koji od korisnika učitava cele brojeve n i k, i ako je $n \geqslant k \geqslant 1$ računa $\binom{n}{k}$, dok u ostalim slučajevima prijavljuje grešku.

4.90. Napisati C# program koji za prirodan broj n i prost broj p određuje broj K kao najveći broj k takav da je $n!$ deljiv sa p^k. Uputstvo: koristiti formulu

$$K = \left\lfloor \frac{n}{p} \right\rfloor + \left\lfloor \frac{n}{p^2} \right\rfloor + \left\lfloor \frac{n}{p^3} \right\rfloor + \left\lfloor \frac{n}{p^4} \right\rfloor + \ldots$$

4.91. Napisati C# program koji od korisnika učitava pozitivan ceo broj n i određuje sa koliko nula se završava broj $n!$. (Napomena: *nipošto ne računati* $n!$ *već koristiti Zadatak 4.90.)*

4.92. Napisati C# program koji nalazi najmanji prirodan broj veći od 1 koji je jednak zbiru kubova svojih cifara.

4.93. Napisati C# program koji od korisnika učitava pozitivne cele brojeve n i k i potom računa i štampa vrednost sledećeg izraza:

$$1^k + 2^k + 3^k + \ldots + n^k.$$

4.94. Suma reda ⟨LINK⟩

4.95. Napisati C# program koji računa proizvod:

$$\frac{1}{\sqrt{2}} \cdot \frac{1}{\sqrt{2+\sqrt{2}}} \cdot \frac{1}{\sqrt{2+\sqrt{2+\sqrt{2}}}} \cdot \ldots \cdot \frac{1}{\sqrt{2+\ldots\sqrt{2+\sqrt{2}}}}$$

gde se u poslednjem razlomku znak za koren javlja n puta.

4.96. Izračunati:

(a) $n^1 - n^2 + n^3 - \ldots + (-1)^{k+1} n^k$, ($n$ i k su neki prirodni brojevi)

(b) $a_1 + a_2 + \ldots + a_n$, gde je $a_k = (k+3)^2 - k$

(c) $\frac{1!}{n^1} + \frac{2!}{n^2} + \frac{3!}{n^3} + \ldots + \frac{m!}{n^m}$, gde su n i m neki prirodni brojevi

(d) $a_1 - a_2 + \ldots + (-1)^{n-1} a_n$, gde je $a_k = 1^2 + 3^2 + \ldots + (2k-1)^2$

(e) $a_1 + a_2 + \ldots + a_n$, gde je $a_k = \frac{1}{k} + \frac{1}{k+1} + \ldots + \frac{1}{2k}$

4.97. Napisati C# program koji od korisnika učitava ceo broj $n \geqslant 0$, potom učitava realan broj x i realne brojeve $a_0, a_1, \ldots, a_n$ i nakon toga računa i štampa vrednost izraza

$$a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n.$$

4.98. Funkcija $\sin x$ se može približno izračunati ovako:

$$\sin x \approx \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots + (-1)^n \frac{x^{2n+1}}{(2n+1)!}.$$

Pri tome, što je n veće, to je i vrednost izraza na desnoj strani tačnija (bliža stvarnoj vrednosti). Napisati C# program koji od korisnika učitava realan broj x i na opisani način računa približnu vrednost broja $\sin x$. Sumu računati dok ne dođemo do sabirka čija apsolutna vrednost je manja od 10^{-5}.

4.7 Rekurzivno zadati nizovi

Za niz brojeva $a_1, a_2, a_3, \ldots, a_n, \ldots$ kažemo da je zadat *rekurzivno* ako je, osim prvih nekoliko članova, svaki član niza definisan preko nekih prethodnih. Tako, na primer, za niz brojeva

$$a_1 = 1, \quad a_n = 3a_{n-1} - 1$$

imamo: $a_1 = 1$, $a_2 = 3a_1 - 1 = 2$, $a_3 = 3a_2 - 1 = 5$, $a_4 = 3a_3 - 1 = 14$, $a_5 = 3a_4 - 1 = 41$, itd.

Primer. Napisati C# program koji računa n-ti element niza

$$a_1 = 1, \quad a_n = 3a_{n-1} - 1.$$

⟨C# fajl⟩

```
using System;

class RekurzNiz1 {
  static void Main() {
    int n, i, a;
    Console.WriteLine("Unesi n");
    n = int.Parse(Console.ReadLine());
    if(n <= 0) { Console.WriteLine("Nije definisan"); }
    else if (n == 1) { Console.WriteLine("a_n = 1"); }
    else {
      a = 1;
      for(i = 2; i <= n; i++) { a = 3*a - 1; }
      Console.WriteLine("a_n = {0}", a);
    }
  }
}
```

Ovu tehniku možemo da koristimo kadgod imamo niz kod koga je opšti član definisan samo preko njegovog prethodnika. Međutim, ako imamo niz kod koga je opšti član definisan preko više prethodnih članova, odgovarajući program se malo komplikuje. Razmotrimo sledeći primer. *Fibonačijevi (Fibonacci)* brojevi se definišu na sledeći način:

$$F_1 = 1, \quad F_2 = 1,$$
$$F_n = F_{n-1} + F_{n-2}.$$

Na primer,

$$F_3 = F_2 + F_1 = 2,$$
$$F_4 = F_3 + F_2 = 3,$$
$$F_5 = F_4 + F_3 = 5,$$
$$F_6 = F_5 + F_4 = 8,$$
$$F_7 = F_6 + F_5 = 13, \text{ i tako dalje.}$$

Evo programa koji računa n-ti Fibonačijev broj:

⟨C# fajl⟩

```
using System;

class Fibonacci {
  static void Main() {
    int n, i, Fn, Fn_1, Fn_2;
    Console.WriteLine("Unesi n");
    n = int.Parse(Console.ReadLine());
    if(n <= 0) { Console.WriteLine("Nije definisan"); }
    else if (n == 1 || n == 2) { Console.WriteLine("F_n = 1"); }
    else {
      Fn_1 = 1;
      Fn = 1;
      for(i = 3; i <= n; i++) {
        Fn_2 = Fn_1;
        Fn_1 = Fn;
        Fn = Fn_1 + Fn_2;
      }
      Console.WriteLine("F_n = {0}", Fn);
    }
  }
}
```

Zadaci.

4.99. Napisati C# program koji računa n-ti član sledećeg niza:

(a) $a_1 = -1,\ a_2 = 5,\ a_n = a_{n-1}^2 - 4a_{n-2}$

(b) $b_1 = 1,\ b_2 = 2,\ b_3 = 3,\ b_n = 2b_{n-1}b_{n-2} - b_{n-3}^2$

(c) $c_1 = 1,\ c_2 = -2,\ c_3 = 3,\ c_4 = -2,\ c_n = c_{n-2} - c_{n-4}$

(d) $d_1 = 1,\ d_2 = -1,\ d_n = d_{n-1}^2 - (n-2)! \cdot d_{n-2}$
(Pažnja: faktorijel računati lukavo.)

4.100. Napisati C# program koji od korisnika učitava realne brojeve p, q, r, s i ceo broj n, i računa n-ti član niza koji je definisan ovako:

$$a_1 = p,\ a_2 = q,\ a_n = r \cdot a_{n-1} + s \cdot a_{n-2}.$$

4.101. Niz brojeva a_n je definisan ovako:

$$a_1 = 1, a_2 = 1,\ a_n = a_{n-2} + a_{n-3} + \ldots + a_1.$$

Napisati C# program koji od korisnika učitava prirodan broj n i računa n-ti element ovog niza.

4.102. *Paskalov trougao* je trougaona šema brojeva koja izgleda ovako:

```
1
1  1
1  2   1
1  3   3   1
1  4   6   4   1
1  5  10  10   5   1
```

Primetimo da n-ti red ovog trougla ima oblik $q_0\ q_1\ q_2 \ldots q_n$ gde je $q_{k+1} = (n-k)q_k$ div $(k+1)$. Napisati C# program koji ispisuje elemente n-tog reda Paskalovog trougla.

4.103. Niz brojeva a_n je definisan ovako:

$$a_1 = 1, a_2 = 1,\ \ a_n = \begin{cases} a_{n-1} + a_{n-2}, & n \text{ parno} \\ 3a_{n-1} - 2a_{n-2}, & n \text{ neparno.} \end{cases}$$

Napisati C# program koji od korisnika učitava prirodan broj n i računa n-ti element ovog niza.

4.104. Niz brojeva a_n je definisan ovako:

$$a_1 = 3, a_2 = -1,\ \ a_n = \begin{cases} na_{n-1} + a_{n-2}, & n \text{ parno} \\ a_{n-1} - na_{n-2}, & n \text{ neparno.} \end{cases}$$

Napisati C# program koji od korisnika učitava prirodan broj n i računa i štampa sumu prvih n elemenata ovog niza.

4.105. Neka je $a_1 = 5, a_2 = 7$, a svaki sledeći element niza se dobija kao poslednja cifra zbira prethodna dva elementa niza. Napisati C# program koji nalazi broj $n > 2$ sa osobinom $a_n = 5$ i $a_{n+1} = 7$.

4.106. Da se podsetimo: Fibonačijev niz je niz brojeva definisan na sledeći način: $F_1 = 1, F_2 = 1$ i $F_{n+2} = F_{n+1} + F_n$. Napisati C# program koji određuje kada će se u nizu poslednjih cifara Fibonačijevog niza brojeva ponoviti 1, 1.

4.107. Dimenzije papira ⟨LINK⟩

4.108. Pčele i trutovi ⟨LINK⟩

4.109. Procena kvadratnog korena Njutnovom metodom ⟨LINK⟩

4.110. Poslednje dve cifre Fibonačijevog broja ⟨LINK⟩

4.8 Najveći zajednički delilac dva broja

Sada ćemo pokazati čuveni Euklidov algoritam za nalaženje *NZD* dva prirodna broja. Iako jednostavan, Euklidov algoritam je izuzetno važan i predstavlja osnovu za mnoge druge algoritme teorije brojeva. On je značajan i istorijski, čak toliko da se ponekad može sresti i ovakva izreka:

Euklidov algoritam je pradeda svih algoritama!

Najveći zajednički delilac (*NZD*) brojeva a i b je najveći prirodan broj d takav da se d sadrži i u a i u b. Brojevi a i b mogu biti i negativni, *ali je NZD dva broja uvek pozitivan broj!*

Euklidov algoritam za računanje *NZD* dva pozitivna broja a i b se zasniva na uzastopnom deljenju:

(1) podelimo celobrojno a sa b, količnik odbacimo, a ostatak nazovemo r;

(2) vrednost iz b prepišemo u a, vrednost iz r prepišemo u b;

(3) postupak ponavljamo sve dok je $b \neq 0$; kada b postane 0, vrednost promenljive a predstavlja *NZD* brojeva a i b.

Primer. Izračunati $NZD(3289, 2415)$ Euklidovim algoritmom.

Stavimo $a = 3289$, $b = 2415$. Rad Euklidovog algoritma je prikazan tabelom pored, gde kolona q označava celobrojni količnik brojeva a i b, a kolona r označava ostatak pri celibrojnom deljenju. Poslednja vrednost u koloni a je 23 i zato je $NZD(3289, 2415) = 23$.

a	b	q	r
3289	2415	1	874
2415	874	2	667
874	667	1	207
667	207	3	46
207	46	4	23
46	23	2	0
23	0	STOP	

Pored je dat C# programski fragment koji računa *NZD* za pozitivne cele brojeve `a` i `b`.

```
int a, b, r;
a = int.Parse(Console.ReadLine());
b = int.Parse(Console.ReadLine());
do {
  r = a % b; a = b; b = r;
} while(b > 0);
Console.WriteLine(a);
```

Zadaci.

4.111. Izračunati "peške" *NZD* sledećih brojeva:

(a)	89 i 55	(c)	437 i 529
(b)	125 i 75	(d)	330 i 121

4.112. Napisati C# program koji računa *NZD* za dva *proizvoljna* cela broja. Voditi računa o tome da $NZD(0,0)$ nije definisano, da je $NZD(x,y) = NZD(|x|,|y|)$ ako x i y nisu oba pozitivni, kao i da je $NZD(0,x) = NZD(x,0) = x$ za $x > 0$.

4.113. Za prirodne brojeve a i b kažemo da su uzajamno prosti ako je $NZD(a,b) = 1$. Napisati C# program koji učitava prirodne brojeve a, p i q, i štampa sve prirodne brojeve iz intervala $[p,q]$ koji su uzajamno prosti sa a.

4.114. Napisati C# program koji računa *NZS* prirodnih brojeva a i b. Koristiti činjenicu da je

$$NZS(a,b) = \frac{ab}{NZD(a,b)}.$$

4.115. (*MRKA* – Mali Racionalni KAlkulator) Napisati C# program koji učitava cele brojeve p_1, q_1, p_2 i q_2 i računa zbir, razliku, proizvod i količnik razlomaka $\frac{p_1}{q_1}$ i $\frac{p_2}{q_2}$. Razlomak $\frac{u}{v}$ koji se dobija kao rezultat treba da bude skraćen i ispisan u obliku u/v. (Na primer, ako je rezultat neke operacije $\frac{4}{6}$, računar treba da ispiše $2/3$.)

4.116. Za prirodan broj n, Ojlerova funkcija $\varphi(n)$ se definiše ovako: $\varphi(n)$ je jednako broju elemenata skupa $\{1,2,3,\ldots,n\}$ koji su uzajamno prosti sa n. Napisati C# program koji učitava prirodan broj n i računa i štampa $\varphi(n)$.

4.117. Euklid ⟨LINK⟩

4.118. Kineska teorema ⟨LINK⟩

4.119. Učenici na istim sedištima ⟨LINK⟩

4.9 Formatirani ispis

Ima nogo situacija kada je potrebno podatke ispisati u obliku tabele. U takvim slučajevima prilikom ispisa podatka možemo koristiti "proširene plejs-holdere" koji osim rednog broja argumenta mogu da sadrže širinu polja za ispis i broj decimala za ispis realnog broja. Na primer, naredba

```
Console.WriteLine("Pi = {0}", Math.PI);
```

će na monitor ispisati

```
Pi = 3.14159265358979
```

dok će naredba

```
Console.WriteLine("Pi = {0:0.00}", Math.PI);
```

na monitor ispisati

```
Pi = 3.14
```

Broj π na četiri decimale bismo ispisali ovako:

```
Console.WriteLine("Pi = {0:0.0000}", Math.PI);
```

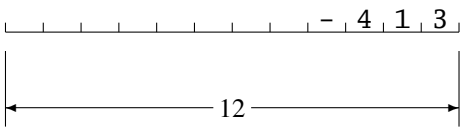
Dakle, broj nula iza decimalne tačke nam kaže na koliko decimala želimo da ispišemo broj.

Druga mogućnost je da koristimo "plejs-holder" u obliku {i,w} što znači da će odgovarajuća promenljiva sa spiska promenljivih biti ispisana u polju širine w. Ako je širina polja veća od potrebne, ispis će biti poravnat po desnoj ivici.

Na primer, ako je $n = -413$ celobrojna vrednost onda naredba

```
Console.WriteLine("{0,12}", n)
```

ispisuje:



(Naravno, sve prateće strelice i tarabe se neće videti!)

Primer. Otplata duga primenom konformne kamatne stope se realizuje tako što svakog meseca dužnik uplaćuje neki iznos (rata), a kamata se obračunava na ostatak duga. Kada ostatak duga postanje manji od rate, dug se likvidira odjednom. Pri tome se iznos rate i kamatna stopa regulišu ugovorom između banke i dužnika. Na primer, ako otplaćujemo dug u visini od 150.000 din primenom konformne kamatne stope od 1,83% na mesečnom nivou i sa visinom rate od 15.000 din, plan otplate kredita izgleda ovako:

Mesec	*Staro stanje*	*Kamata*	*Rata*	*Novo Stanje*
1	150000.00	2745.00	15000.00	137745.00
2	137745.00	2520.73	15000.00	125265.73
3	125265.73	2292.36	15000.00	112558.10
4	112558.10	2059.81	15000.00	99617.91
5	99617.91	1823.01	15000.00	86440.92
6	86440.92	1581.87	15000.00	73022.79
7	73022.79	1336.32	15000.00	59359.10
8	59359.10	1086.27	15000.00	45445.37
9	45445.37	831.65	15000.00	31277.03
10	31277.03	572.37	15000.00	16849.39
11	16849.39	308.34	15000.00	2157.74
12	2157.74	39.49	2197.23	0.00

Vidimo da će dug biti otplaćen za 12 meseci i da će banka zaraditi 17197,23 din (zarada banke se dobija kao zbir brojeva u koloni *Kamata*).

Napisati C# program koji od korisnika učitava visinu duga, kamatnu stopu na mesečnom nivou i visinu rate, a onda štampa plan otplate duga i računa za koliko meseci će dug biti otplaćen, kao i kolika je zarada banke.

⟨C# fajl⟩

```
using System;

class OtplataDuga {
  static void Main() {
    double stanje, mesKamStopa, kamata, rata, zaradaBanke;
    int n;

    Console.Write("Visina duga: ");
    stanje = double.Parse(Console.ReadLine());
    Console.Write("Kamatna stopa na mesecnom nivou (%): ");
    mesKamStopa = double.Parse(Console.ReadLine());
    Console.Write("Rata: ");
    rata = double.Parse(Console.ReadLine());
    zaradaBanke = 0;
    n = 0;
```

```
    Console.Write("Mesec");
    Console.Write("{0,15}", "Staro stanje");
    Console.Write("{0,15}", "Kamata");
    Console.Write("{0,15}", "Rata");
    Console.WriteLine("{0,15}", "Novo stanje");
    while (stanje > 0.0) {
      n++;
      kamata = stanje * mesKamStopa / 100.0;
      zaradaBanke = zaradaBanke + kamata;
      Console.Write("{0,5}", n);
      Console.Write("{0,15:0.00}", stanje);
      Console.Write("{0,15:0.00}", kamata);
      Console.Write("{0,15:0.00}", rata);
      if(stanje + kamata > rata) {
        stanje = stanje + kamata - rata;
      }
      else {
        rata = stanje + kamata;
        stanje = 0.0;
      }
      Console.WriteLine("{0,15:0.00}", stanje);
    }
    Console.WriteLine("Dug ce biti otplacen za {0} meseci", n);
    Console.WriteLine("Zarada banke iznosi: {0:0.00}", zaradaBanke);
  }
}
```

U ovom primeru smo naišli na dve nove konstrukcije:

- komandu `Console.Write`, i
- ispis realnog broja u polje date širine.

Komanda `Console.Write` je veoma slična komandi `Console.WriteLine`, s tim što komanda `Console.WriteLine` ispiše tekst i pređe u novi red, dok komanda `Console.Write` ispiše tekst i ostane u istom redu. Na taj način možemo jedan red ispisati u nekoliko koraka, što nam je u ovom primeru bilo veoma pogodno. Nizom komandi

```
Console.Write(...);
Console.Write(...);
Console.Write(...);
Console.WriteLine(...);
```

se jedan red tabele formira u više koraka. Naredbe `Console.Write` ispisuju podatke jedan za drugim u istom redu, nakon čega naredba `Console.WriteLine` dopiše još i svoje podatke i onda pređe u novi red.

U ovom primeru se od nas očekivalo i da prilikom ispisa formiramo preglednu tabelu. Da bismo to postigli morali smo da kontrolišemo širinu ispisa u svakoj `Write`, odnosno, `WriteLine` naredbi, što smo postigli plejs-holderima sa složenijom strukturom. Oznaka `{0,15:0.00}` u formatu naredbe za ispis (`Write`, odnosno, `WriteLine`) znači da će prva promenljiva sa spiska promenljivih biti ispisana u polju širine 15, i to na dve decimale.

4.10 Ugnežđeni ciklusi

U telu ciklusa se mogu nalaziti bilo kakve C# naredbe, pa i drugi ciklusi. Za ovako raspoređene cikluse kažemo da su *ugnežđeni*.

```
while (n > 0) {
  ...
  for(k = 1; k <= n; k++) {
    ...
  }  ...
}
```

Primer. Napisati C# program koji od korisnika učitava ceo broj n i potom ispisuje tablicu množenja do n. Na primer, za $n = 3$ program ispisuje niz redova koji je dat pored.

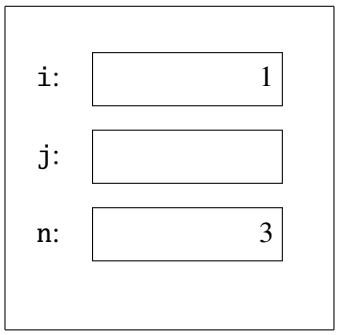
```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
```

⟨C# fajl⟩

```
using System;

class Tabelica {
  static void Main() {
    int n, i, j;
    Console.WriteLine("Unesi n");
    n = int.Parse(Console.ReadLine());
    for(i = 1; i <= n; i++) {
      for(j = 1; j <= n; j++) {
        Console.WriteLine("{0} * {1} = {2}", i, j, i * j);
      }
    }
  }
}
```

Pogledaćemo sada detaljno kako program radi za $n = 3$. U prvom prolazu kroz spoljašnji "for" ciklus promenljiva `i` dobije vrednost 1:

MEMORIJA:




```
for(i = 1; i <= n; i++) {
  for(j = 1; j <= n; j++) {
    Console.WriteLine(
      "{0} * {1} = {2}", i, j, i * j);
  }
}
```

nakon čega se izvrši unutrašnji "for" ciklus koji ispiše prva tri reda tabele:

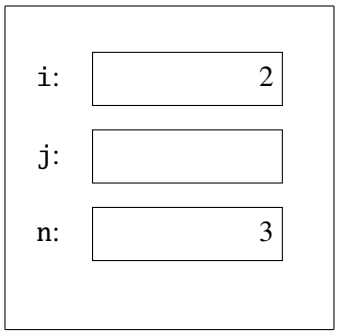
MONITOR:

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
```

```
for(i = 1; i <= n; i++) {
  for(j = 1; j <= n; j++) {
    Console.WriteLine(
      "{0} * {1} = {2}", i, j, i * j);
  }
}
```

U drugom prolazu kroz spoljašnji "for" ciklus promenljiva `i` dobije vrednost 2:

MEMORIJA:




```
for(i = 1; i <= n; i++) {
  for(j = 1; j <= n; j++) {
    Console.WriteLine(
      "{0} * {1} = {2}", i, j, i * j);
  }
}
```

nakon čega se izvrši unutrašnji "for" ciklus koji ispiše naredna tri reda tabele:

MONITOR:

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
```

```
for(i = 1; i <= n; i++) {
  for(j = 1; j <= n; j++) {
    Console.WriteLine(
      "{0} * {1} = {2}", i, j, i * j);
  }
}
```

U sledećem prolazu kroz spoljašnji "for" ciklus promenljiva `i` dobije vrednost 3 nakon čega se izvrši unutrašnji "for" ciklus koji ispiše poslednja tri reda tabele:

MONITOR:

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
```

```
for(i = 1; i <= n; i++) {
  for(j = 1; j <= n; j++) {
    Console.WriteLine(
      "{0} * {1} = {2}", i, j, i * j);
  }
}
```

Primer. Napisati C# program koji ispisuje tablicu množenja.

⟨C# fajl⟩

```
using System;

class TablicaMnozenja {
  static void Main() {
    int i, j;
    Console.WriteLine("* |   1   2   3   4   5   6   7   8   9  10");
    Console.WriteLine("--+----------------------------------------");
    for(i = 1; i <= 10; i++) {
      Console.Write("{0,2}|", i);
      for(j = 1; j <= 10; j++) {
        Console.Write("{0,4}", i * j);
      }
      Console.WriteLine();
```

```
    }
  }
}
```

Primer. Napisati C# program koji od korisnika učitava ceo broj n i potom ispisuje prvih n redova trougla datog pored:

```
1
1  2
1  2  3
1  2  3  4
1  2  3  4  5
.....................
```

⟨C# fajl⟩

```
using System;

class Trougao {
  static void Main() {
    int n, i, j;
    Console.WriteLine("Unesi n");
    n = int.Parse(Console.ReadLine());
    for(i = 1; i <= n; i++) {
      for(j = 1; j <= i; j++) {
        Console.Write("{0,3}", j);
      }
      Console.WriteLine();
    }
  }
}
```

Zadaci.

4.120. Kockice za jamb ⟨LINK⟩

4.121. Za uređenu trojku (k,l,m) prirodnih brojeva kažemo da je *Pitagorina trojka* ako je $k^2+l^2=m^2$. Napisati C# program koji od korisnika učitava ceo broj n i ako je $n>0$ štampa sve Pitagorine trojke (k,l,m) za koje je $k,l,m \in \{1,2,\ldots,n\}$. Ako je $n \leqslant 0$ ispisati neku poruku o greški.

4.122. Svi sufiksi niza brojeva od 1 do n ⟨LINK⟩

4.123. Serije 123... ⟨LINK⟩

4.124. Trouglovi celobrojnih stranica, zadatog obima ⟨LINK⟩

4.125. Kombinacije poena ⟨LINK⟩

4.126. Varijacije trojki ⟨LINK⟩

4.127. Napisati C# program koji ispisuje sve moguće načine da se dešifruje sabiranje

$$\begin{array}{cccc} & & A & V \\ & & \vdots & \\ + & & A & V \\ \hline & P & A & S \end{array}$$

gde različitim slovima odgovaraju različite cifre, i pri tome $A \neq 0$ i $P \neq 0$. Pas može da zalaje i više od dva puta, a program treba da pronađe sve mogućnosti.

4.128. Napisati C# program koji ispisuje sve moguće načine da se dešifruje sabiranje

$$\begin{array}{ccccc} & & B & O & R \\ & & & \vdots & \\ + & & B & O & R \\ \hline & Š & U & M & A \end{array}$$

gde različitim slovima odgovaraju različite cifre, i pri tome $Š \neq 0$ i $B \neq 0$. U šumi može da bude i više od dva bora, a program treba da pronađe sve mogućnosti.

4.129. Kvadrat od zvezdica ⟨LINK⟩

4.130. Napisati C# program koji utvrđuje da li je sledeća formula tautologija:

$$\neg p \vee (q \wedge (r \vee \neg s))$$

ali pri tome iscrtava i tabelu sa istinitosnim vrednostima za sve moguće vrednosti iskaznih slova.

```
p q r s | F
--------+--
F F F F |
F F F T |
F F T F |
F F T T |
    ⋮
T T T T |
```

4.131. Napisati C# program koji od korisnika učitava ceo broj n i potom ispisuje prvih n redova sledećeg trougla:

```
1
1 1
1 1 1
1 1 1 1
1 1 1 1 1
. . . . . .
```

4.132. Napisati C# program koji od korisnika učitava ceo broj n i potom ispisuje prvih n redova sledećeg trougla:

```
1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
. . . . . .
```

4.133. Napisati C# program koji od korisnika učitava ceo broj n i potom štampa prvih n redova sledećeg trougla (koji je prikazan za $n = 5$):

```
1 2 3 4 5
  2 3 4 5
    3 4 5
      4 5
        5
```

4.134. Napisati C# program koji od korisnika učitava ceo broj n i potom štampa prvih n redova sledećeg trougla (koji je prikazan za $n = 5$):

```
        1
      1 2
    1 2 3
  1 2 3 4
1 2 3 4 5
```

4.135. Napisati C# program koji od korisnika učitava paran broj n i potom iscrtava sledeći kvadrat $n \times n$ (prvih $n/2$ vrsta je popunjeno nulama, a poslednjih $n/2$ vrsta jedinicama):

$$\begin{array}{ccccc}
0 & 0 & 0 & \ldots & 0 \\
0 & 0 & 0 & \ldots & 0 \\
\ldots & \ldots & \ldots & \ldots & \ldots \\
0 & 0 & 0 & \ldots & 0 \\
1 & 1 & 1 & \ldots & 1 \\
1 & 1 & 1 & \ldots & 1 \\
\ldots & \ldots & \ldots & \ldots & \ldots \\
1 & 1 & 1 & \ldots & 1
\end{array}$$

4.136. Napisati C# program koji od korisnika učitava paran broj n i potom iscrtava sledeći kvadrat $n \times n$ (prvih $n/2$ kolona je popunjeno nulama, a poslednjih $n/2$ kolona jedinicama):

$$\begin{array}{cccccccc}
0 & 0 & \ldots & 0 & 1 & 1 & \ldots & 1 \\
0 & 0 & \ldots & 0 & 1 & 1 & \ldots & 1 \\
0 & 0 & \ldots & 0 & 1 & 1 & \ldots & 1 \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
0 & 0 & \ldots & 0 & 1 & 1 & \ldots & 1
\end{array}$$

4.137. Romb od zvezdica ⟨LINK⟩

4.138. Trougao od zvezdica ⟨LINK⟩

4.139. Napisati C# program koji od korisnika učitava ceo broj $n \geqslant 1$ i potom ispisuje kvadrat brojeva formata $2n \times 2n$ koji se sastoji od četiri manja kvadrata, kako je to pokazano u primerima:

$n = 1$

$$\begin{array}{cc}
0 & 1 \\
1 & 0
\end{array}$$

$n = 2$

$$\begin{array}{cccc}
0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 \\
1 & 1 & 0 & 0
\end{array}$$

$n = 3$

$$\begin{array}{cccccc}
0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0
\end{array}$$

4.140. Napisati C# program koji od korisnika učitava prirodan broj n i potom štampa šahovsku tablu stranice $2n$. Polja šahovske table su sastavljena od simbola `X` i `.` (tačka) i svako polje je veličine 2×2. Na primer, za $n = 4$ program štampa:

```
XX..XX..
XX..XX..
..XX..XX
..XX..XX
XX..XX..
XX..XX..
..XX..XX
..XX..XX
```

4.141. Napisati C# program koji ispisuje prvih n redova sledećeg trougla:

```
 1
 2  3
 4  5  6
 7  8  9 10
11 12 13 14 15
. . . . . .
```

4.142. Napisati C# program koji ispisuje prvih n redova sledećeg trougla:

```
1
2 3 2
3 4 5 4 3
4 5 6 7 6 5 4
5 6 7 8 9 8 7 6 5
. . . . . .
```

4.143. Napisati C# program koji od korisnika učitava ceo broj n i potom štampa prvih n redova sledećeg trougla (koji je prikazan za $n = 5$):

```
1  2  3  4  5  6  7  8  9
  10 11 12 13 14 15 16
     17 18 19 20 21
        22 23 24
           25
```

4.144. (Ciklične permutacije) Napisati C# program koji od korisnika učitava pozitivan ceo broj n i potom ispisuje kvadrat brojeva kako sledi:

Opšti slučaj

1	2	3	...	$n-1$	n
2	3	4	...	n	1
3	4	5	...	1	2
⋮	⋮	⋮		⋮	⋮
n	1	2	...	$n-2$	$n-1$

Posebno za $n = 5$

1	2	3	4	5
2	3	4	5	1
3	4	5	1	2
4	5	1	2	3
5	1	2	3	4

⟨LINK⟩

4.145. Napisati C# program koji od korisnika učitava pozitivan ceo broj n i potom brojeve $1, 2, \ldots, n^2$ ispisuje u obliku kvadrata dimenzija $n \times n$ kako je to pokazano u primeru pored za $n = 5$.

$n = 5$

1	6	11	16	21
2	7	12	17	22
3	8	13	18	23
4	9	14	19	24
5	10	15	20	25

4.146. Napisati C# program koji od korisnika učitava ceo broj $n \geqslant 2$ i potom ispisuje kvadrat formata $n \times n$ u čija polja su upisani brojevi 1, 2, ..., n^2 kako je to pokazano u primerima:

$n = 2$

1	2
4	3

$n = 3$

1	2	3
6	5	4
7	8	9

$n = 4$

1	2	3	4
8	7	6	5
9	10	11	12
16	15	14	13

4.147. Brojevi u datoj osnovi ⟨LINK⟩

4.148. Mali loto ⟨LINK⟩

4.149. Ne sadrže cifru 5 ⟨LINK⟩

4.150. Serije neparni parni ⟨LINK⟩

4.151. Nedelje sa negativnim temperaturama ⟨LINK⟩

4.152. Suma segmenata niza između nula ⟨LINK⟩

4.153. Parno neparne serije ⟨LINK⟩

4.154. Najduža serija pobeda ⟨LINK⟩

4.155. Najduža rastuća serija ⟨LINK⟩

4.156. Serije kutija ⟨LINK⟩

4.157. Raspored paketa na policama ⟨LINK⟩

4.158. Najstabilniji temperaturni period ⟨LINK⟩

4.159. Utovar transportnog broda ⟨LINK⟩

4.160. Najduži segment uzastopnih brojeva ⟨LINK⟩

4.161. Planiranje izgradnje prodavnica ⟨LINK⟩

4.162. Broj dana između dva datuma ⟨LINK⟩

4.163. Broj tacaka u trouglu ⟨LINK⟩

4.164. Konverzija baza ⟨LINK⟩

Glava 5

Statički metodi

Programski jezik C# ima "ugrađenu" operaciju `*` koja množi dva broja, ali nema funkciju koja ume da računa *NZD* dva broja. S druge strane, mi *znamo* da napišemo C# program koji računa *NZD* dva broja. Tako se prirodno postavlja pitanje: Da li je moguće C# na neki način "proširiti" novom funkcijom *NZD*?

U programskom jeziku C# se može u jednom redu proveriti da li je dati ceo broj neparan, ali nemamo "ugrađen" test kojim se može proveriti da li je neki ceo broj prost. Ponovo, mi *znamo* da napišemo C# program koji utvrđuje da li je dati broj prost. Da li je moguće C# na neki način "proširiti" novim testom `Prost`?

Programski jezik C# ima bibliotečki metod `WriteLine` koji ume da ispiše neki ceo broj u decimalnom zapisu, ali nema metod kojom bi se dati broj mogao ispisati rečima. Da li je moguće C# na neki način "proširiti" novom komandom `SayTheNumber` koja to radi?

Ova tri retorička pitanja, naravno, imaju očekivani odgovor: DA. U programskom jeziku C# postoji način da korisnik doda svoje metode u kojima implementira procese koje često koristi. Time program postaje kraći, čitkiji i jednostavniji. Postoji mogućnost da se deo programa koji radi jedan nezavisan i smislen posao izdvoji u posebnu celinu, *potprogram*, koji se pozove kadgod nam trebaju njegove usluge. Potprogram u C# može da vraća rezultat, a ne mora. Potprogrami koji vraćaju rezultat se uglavnom pozivaju kao deo nekh složenijih izraza, dok se oni koji ne vraćaju rezultat koriste se kao nove "komande" koje smo uveli u programski jezik. U terminologiji jezika C# potprogrami se zovu *statički metodi*.

5.1 Prvi primer

Primer. Napisati C# program koji za dati broj $n \geqslant 1$ određuje broj elemenata skupa $\{1,2,3,\ldots,n\}$ koji su uzajamno prosti sa n.

Pored je dato rešenje koje se zasniva na onome što do sada znamo iz programiranja. Zato što je $NZD(1,n) = 1$ i zato što je $NZD(n,n) = n > 1$ za $n > 1$, postavili smo odmah `br` na 1, a u "for"-ciklusu proveravamo samo brojeve od 2 do $n-1$.

Rešenje bi bilo veoma jednostavno da nema jedne male poteškoće: Euklidov algoritam *uništava polazne vrednosti svojih promenljivih*. Zato je pre primene Euklidovog algoritma potrebno iskopirati vrednosti promenljivih `n` i `k` u pomoćne promenljive `a` i `b`, i onda pustiti algoritam da izračuna *NZD* brojeva `a` i `b`. (Ako bismo pustili Euklidov algoritam da radi direktno sa promenljivim `n` i `i`, vrednosti ovih promenljivih bi bile uništene i program ne bi radio korektno.)

```
using System;
class BrUzProstih {
  static void Main() {
    int a, b, r, n, i, br;
    Console.Write("n -> ");
    n = int.Parse(
          Console.ReadLine());
    br = 1;
    for(i = 2; i < n ; i++) {
      a = n; b = i;
      do {
        r = a % b;
        a = b; b = r;
      } while(b > 0);
      if(a == 1) { br++; }
    }
    Console.WriteLine(br);
  }
}
```

Evo sada elegantnijeg rešenja gde ćemo uvesti novi metod `NZD`. Naime, kada bi C# imao metod koji računa *NZD*, rešenje bi bilo veoma jednostavno i prirodno. Mi ćemo se, zato, prvo napisati program pod pretpostavkom da takav metod postoji, a onda ćemo ga i napisati.

Vidimo da je na ovaj način program postao razumljiviji i jednostavniji. Jednom kada smo uveli novi metod on se poziva kao i funkcije koje je C# poznavao od ranije.

```
using System;
class BrUzProstih2 {
  static void Main() {
    int n, br, i;
    Console.Write("n -> ");
    n = int.Parse(
          Console.ReadLine());
    br = 1;
    for(i = 2; i < n ; i++) {
      if(NZD(n, i) == 1) {
        br++;
      }
    }
    Console.WriteLine(br);
  }
```

Metod NZD ima dva argumenta, a i b, i njihove deklaracije unutar liste argumenata su razdvojene zarezom. Naravno, kada pozivamo metod NZD moramo da navedemo dva argumenta (što mogu biti promenljive ili izrazi) koji treba da budu razdvojeni zarezom. Poslednji red u telu metoda sadrži posebnu naredbu `return` koja kaže koji izraz treba vratiti kao rezultat rada metoda.

```
  static int NZD(int a, int b) {
    int r;
    do {
      r = a % b;
      a = b; b = r;
    } while(b > 0);
    return a;
  }
}
```

Primer. Napisati C# program koji ispisuje brojeve od 1 do *n* tim redom, ali svaki unatraške. Na primer, broj 1275 će biti ispisan kao 5721.

U rešenju koje je dato pored imamo metod `Unazad` koji ne vraća nikakav rezultat već samo ispisuje dati broj unazad. Metodi koji ne vraćaju rezultat predstavljaju nove "komande" kojima obogaćujemo programski jezik.

Pisanje metoda koji ne vraćaju rezultat je slično pisanju metoda koji vraćaju rezultat, s tim da:

- umesto tipa rezultata se navodi `void` (što na engleskom znači *prazno*);
- ovakvi metodi se pozivaju kao nezavisne komande (a ne unutar drugih izraza);
- metod ne mora da sadrži `return` naredbu; takav metod se završava kada se izvrše sve naredbe navedene u metodu;
- ako se u metodu javlja `return` naredba, iza nje se *ne* navodi izraz (jer metod ne vraća rezultat).

```
using System;
class IspisiUnazad {
  static void Main() {
    int n, i;
    Console.Write(n -> ");"
    n = int.Parse(
          Console.ReadLine());
    for(i = 1; i <= n; i++) {
      Unazad(i);
    }
  }
  static void Unazad(int n) {
    while(n > 0) {
      Console.Write(n % 10);
      n /= 10;
    }
    Console.WriteLine();
  }
}
```

Primer. Napisaćemo ponovo C# program koji za dati prirodan broj n utvrđuje broj elemenata skupa $\{1,2,\ldots,n\}$ koji su uzajamno prosti sa n, s tim da ćemo osim metoda NZD i utvrđivanje broja uzajamno prostih elemenata gornjeg skupa izdvojiti u poseban metod.

Metod BUP koristi metod NZD, a glavni deo programa je postao krajnje jednostavan: nakon učitavanja broja n ispišemo vrednost BUP(n).

Uvođenjem metoda programi postaju kraći, jasniji, prirodniji i čitkiji. Čitkost je jedna od bitnih osobina programa. Program koji liči na špagete (bolonjeze) se najlakše piše, ali najteže shvata i kasnije popravlja.

Doista, u računarskom žargonu i postoji termin "špageti programiranje" (*spaghetti programming*) koji označava zbrda-zdola skarabudžene programe. Postoje standardne preporuke čijim poštovanjem dolazimo do lepih, čitkih i prirodnih programa i o njima ćemo govoriti u nastavku.

U jednom programu može biti više metoda (na kraju krajeva, i metod Main je jedan metod u programu). Važno je znati sledeće:

☞ *Ako u programu ima više metoda koji pozivaju jedan drugi, redosled kojim su navedeni nije bitan!*

```
using System;
class BrUzProstih3 {
  static void Main() {
    int n;
    Console.Write("n -> ");
    n = int.Parse(
          Console.ReadLine());
    Console.WriteLine(BUP(n));
  } // Main

  static int BUP(int n) {
    int br = 1;
    for(int i = 2; i < n ; i++) {
      if(NZD(n, i) == 1) {
        br++;
      }
    }
    return br;
  } // BUP

  static int NZD(int a, int b) {
    int r;
    do {
      r = a % b;
      a = b; b = r;
    } while(b > 0);
    return a;
  } // NZD
}
```

Metod se poziva (aktivira) tako što se na odgovarajućem mestu u programu navede njegovo ime, a u zagradi se navede vrednost za koju treba proveriti da li je prost broj. Prilikom pozivanja metoda argument ne mora biti samo promenljiva, već to može biti i čitav izraz. Računar će prvo izračunati vrednost tog izraza, pa će primeniti metod na dobijenu vrednost.

5.2 Drugi primer

Za prirodan broj kažemo da je *prost* ako ima tačno dva delioca. Jedan ne baš inteligentan algoritam kojim se proverava da li je n prost broj sastoji se u tome da prosto prebrojimo delioce broja n i proverimo da li smo dobili 2. Sada ćemo pokazati znatno bolju ideju.

Primetimo, prvo, da ako je $n \geqslant 2$ onda n ima bar dva delioca: 1 i n. Njih zovemo *trivijalni delioci* broja n. Zato se provera da li je n prost svodi na to da se utvrdi da li n ima *netrivijalnog delioca*. Lako se vidi da je sledeće tačno:

> *Ako broj n ima netrivijalni delilac koji je* $\geqslant \lfloor\sqrt{n}\rfloor$ *onda ima i netrivijalni delilac koji je* $\leqslant \lfloor\sqrt{n}\rfloor$.

Na osnovu toga zaključujemo da je prilikom provere da li je $n \geqslant 2$ prost broj dovoljno proveriti da li on ima netrivijalnog delioca iz skupa $\{2,3,\ldots,\lfloor\sqrt{n}\rfloor\}$. Pri tome je dovoljno proveriti samo neparne brojeve iz tog skupa, zato što parni brojevi veći od 2 nisu prosti, a parnost broja lako proveravamo. Sve ove ideje možemo za zapakujemo u metod `Prost` koji proverava da li je dati ceo broj prost:

```
static bool Prost(int n) {
  if(n <= 1) { return false; }
  if(n == 2) { return true; }
  if(n % 2 == 0) { return false; }
  int L = (int)Math.Sqrt(n);
  for(int d = 3; d <= L; d += 2) {
    if(n % d == 0) { return false; }
  }
  return true;
}
```

Primetimo, pre svega, da metod može da vrati i logičku vrednost. Jedan način da pozovemo takav metod je unutar uslova "if" kontrolne strukture, recimo ovako:

```
if(Prost(n)) {
  Console.WriteLine("Broj je prost");
}
else {
  Console.WriteLine("Broj nije prost");
}
```

Osim toga, vidimo i da naredba `return` može da se pojavi bilo gde u metodu. Čim naiđe na naredbu `return` računar prekida sa izvršavanjem metoda i vraća se u deo programa odakle je metod bio pozvan.

Primer. Za neparan prirodan broj p kažemo da je je *lep prost*, ako su i p i $\frac{p-1}{2}$ prosti brojevi. Napisati C# program koji ispituje da li je broj lep prost.

⟨C# fajl⟩

```
using System;
class LepProst {
  static void Main() {
    int n;
    Console.Write("n -> ");
    n = int.Parse(Console.ReadLine());
    if(n % 2 == 0) { Console.WriteLine("Broj nije neparan!"); }
    else if(n < 3) { Console.WriteLine("Broj < 3!"); }
    else if(Prost(n) && Prost((n - 1)/2)) {
      Console.WriteLine("Lep prost");
    }
    else {
      Console.WriteLine("Nije lep prost");
    }
  }

  static bool Prost(int n) {
    if(n <= 1) { return false; }
    if(n == 2) { return true; }
    if(n % 2 == 0) { return false; }
    int L = (int)Math.Sqrt(n);
    for(int d = 3; d <= L; d += 2) {
      if(n % d == 0) { return false; }
    }
    return true;
  }
}
```

Primer. Napisati metod `static bool Prestupna(int g)` koji proverava da li je godina prestupna po gregorijanskom kalendaru.

```
static bool Prestupna(int g) {
  return g % 400 == 0 || g % 100 != 0 && g % 4 == 0;
}
```

Ovaj metod ima samo jednu naredbu – "return". On računa vrednost komplikovanog logičkog izraza i vraća njegov rezultat kao rezultat rada metoda. Ovakve metode pišemo da bi program bio jasniji, prirodniji i čitkiji.

Uporedite ova dva parčeta kôda koja rade isti posao:

```
// verzija 1
if(Prestupna(god)) { ... }
else { ... }

// verzija 2
if(g % 400 == 0 || g % 100 != 0 && g % 4 == 0) { ... }
else { ... }
```

5.3 Treći primer

Primer: Večiti kalendar. Napisati C# program koji za proizvoljan mesec i godinu (koja treba da bude posle 1583. godine) ispisuje kalendar za taj mesec.

Ključni problem je da se za dati datum predstavljen pomoću tri celobrojne promenljive d, m, g utvrdi koji je to dan u nedelji. Njega rešava sledeća formula u kojoj / označava celobrojno deljenje, a % ostatak pri celobrojnom deljenju:

$$p = \big(d + k + g + g/4 - 2 \cdot (v\%4) - f\big)\%7,$$

gde je

- $v = g/100$ broj vekova u godini;
- $f = 1$ ako je g prestupna godina i $m \in \{1,2\}$, u ostalim slučajevima je $f = 0$;
- k je broj koji se dobija od broja m prema sledećoj tabeli:

m	8	2, 3, 11	6	9, 12	4, 7	1, 10	5
k	0	1	2	3	4	5	6

Broj p tumačimo ovako: ako je $p = 0$ taj dan je nedelja, ako je $p = 1$ taj dan je ponedeljak, ..., ako je $p = 6$ taj dan je subota.

⟨C# fajl⟩

```
using System;

class VecitiKalendar {

  // Da li je godina g prestupna?
  static bool Prestupna(int g) {
    return g % 400 == 0 || g % 100 != 0 && g % 4 == 0;
  }
```

```
// Broj dana u mesecu m godine g
static int BrojDana(int m, int g) {
  if(m == 2) {
    if(Prestupna(g)) { return 29; } else { return 28; }
  }
  else if(m == 4 || m == 6 || m == 9 || m == 11) { return 30; }
  else { return 31; }
}

// Za 1. dan meseca m godine g odredjujemo koji je to dan
// u nedelji: 0 = nedelja, 1 = ponedeljak, ..., 6 = subota
static int PrviDan(int m, int g) {
  int v = g / 100;

  int f = 0;
  if(Prestupna(g) && (m == 1 || m == 2)) { f = 1; }

  int k;
  if(m == 8) { k = 0; }
  else if(m == 5) { k = 6; }
  else if(m == 6) { k = 2; }
  else if(m == 9 || m == 12) { k = 3; }
  else if(m == 4 || m == 7) { k = 4; }
  else if(m == 1 || m == 10) { k = 5; }
  else { k = 1; }

  return (1 + k + g + g/4 - 2 * (v % 4) - f) % 7;
}

// Metod ispisuje tabelicu koja predstavlja kalendar za mesec
// koji ima n dana i ciji prvi dan je nedelja ako je d = 0,
// ponedeljak ako je d = 1, ..., subota ako je d = 6
static void IspisiMesec(int d, int n) {
  int i;
  Console.WriteLine(" ne po ut sr ce pe su");
  for(i = 0; i < d; i++) { Console.Write("   "); }
  for(i = 1; i <= n; i++) {
    if(i <= 9) { Console.Write("  "); }
    else { Console.Write(" "); }
    Console.Write(i);
    if((i + d) % 7 == 0) { Console.WriteLine(); }
  }
  Console.WriteLine();
}
```

```
  // Glavni program samo ucita podatke, proveri da li
  // su smisleni i onda poziva metod IspisiMesec
  static void Main() {
    Console.Write("mesec: ");
    int m = int.Parse(Console.ReadLine());
    Console.Write("godina: ");
    int g = int.Parse(Console.ReadLine());
    if(!(1 <= m && m <= 12 && g >= 1583)) {
      Console.WriteLine("Mesec i godina nisu korektni");
    }
    else {
      int d = PrviDan(m, g);
      int n = BrojDana(m, g);
      IspisiMesec(d, n);
    }
  }
}
```

5.4 Deklaracija metoda

Pogledajmo sada korak po korak kako se deklariše metod. Kao primer će nam poslužiti metod `NZD` koga smo upravo videli.

Prvi red deklaracije kaže da je metod *statički* (štagod to značilo, za sada samo takve metode koristimo), vraća rezultat tipa `int`, zove se `NZD` i ima dva argumenta koji se zovu `a` i `b` i oba su tipa `int`. Iz prvog reda deklaracije metoda odmah je jasno da metod `NZD` nešto radi sa dva cela broja i onda vrati ceo broj kao rezultat svog rada.

```
static int NZD(int a, int b) {
  int r;
  do {
    r = a % b;
    a = b; b = r;
  } while(b > 0);
  return a;
}
```

☞ *Jedna od prvih preporuka je da se imena metoda biraju tako da podsećaju na ono čemu metod služi. Mi smo mogli metod* `NZD` *nazvati* `Glb` *i program bi radio jednako dobro, ali bi čitaocu trebalo mnogo više vremena da shvati šta on zapravo radi.*

Posle uvodnih ljubaznosti sledi *telo metoda* koje objašnjava šta i kako metod računa. Telo metoda je najobičniji spisak naredbi ograđenih sa { i }. Poslednja izvršna naredba u telu metoda koji vraća neki rezultat je `return` naredba koja ima oblik:

`return` ⟨*izraz*⟩`;`

```
static int NZD(int a, int b) {
  int r;
  do {
    r = a % b;
    a = b; b = r;
  } while(b > 0);
  return a;
}
```

Promenljive koje se javljaju u metodu se zovu *lokalne promenljive* metoda. Promenljive `a`, `b` i `r` postoje samo u ovom metodu i nigde više u programu. Metod `NZD` ih koristi slobodno, a izvan njega su nedostupne.

```
static int NZD(int a, int b) {
  int r;
  do {
    r = a % b;
    a = b; b = r;
  } while(b > 0);
  return a;
}
```

Metod može da ima više argumenata različitog tipa, ili se može desiti da uopšte nema argumente. Metod može da vrati rezultat, ali postoje i metodi koji ne vraćaju rezultat. Evo nekoliko primera deklaracija metoda:

```
static void Main() { ... }
static double Stepen(double x, int n) { ... }
static int OdseciDecimale(double x) { ... }
static string Formatiraj(string s, int w) { ... }
static string UcitajRed() { ... }
```

☞ Na osnovu ovih primera možemo da zaključimo sledeće:

- metod može imati više argumenata koji mogu, a ne moraju biti istog tipa;
- tip rezultata metoda može biti svaki standardni tip (`double`, `int`, `bool`, `char`, `string` itd), kao i neki drugi tipovi o kojima ćemo govoriti kasnije;
- metod ne mora da vraća rezultat i tada umesto tipa rezultata pišemo `void`;
- metod može imati svoje privatne promenljive;
- metod koji ne vraća rezultat ne mora da sadrži `return` naredbu; takav metod se završava kada se izvrše sve naredbe navedene u metodu;
- čim naiđe na naredbu `return` računar prekida sa izvršavanjem metoda i vraća se u deo programa odakle je metod bio pozvan;
- poslednja izvršna naredba metoda koji vraća rezultat mora da bude naredba `return` iza koje je navedena vrednost koja se vraća kao rezultat rada metoda.

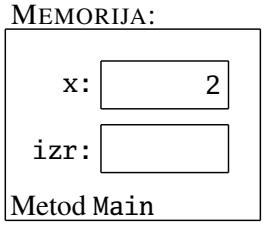
5.5 Aktivacija metoda

Pogledajmo na jednom primeru šta sve računar preživljava kada korisnik pozove metod. Niz postupaka koje računar preduzme tim povodom zove se *aktivacija metoda*.

Pored je dat C# program koji računa vrednost izraza

$$25 + x^{10} - 0.5x^{13}.$$

Kada startujemo program, u memoriji će biti napravljene dve kućice: jedna će se zvati x, a druga izr. Potom će program učitati od korisnika neki realan broj i upisati ga u fioku x. Recimo da je korisnik uneo broj 2.



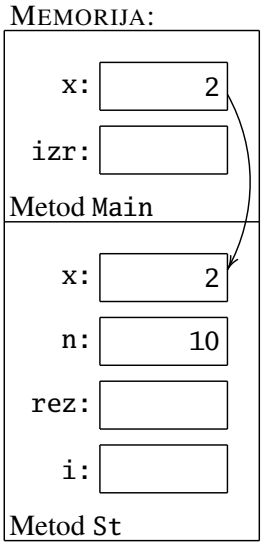

```
using System;
class Izraz {
  static void Main() {
    double x, izr;
    Console.Write("x -> ");
    x = double.Parse(
      Console.ReadLine());
    izr = 25 + St(x,10) - 0.5*St(x,13);
    Console.WriteLine(izr);
  }

  static double St(double x, int n) {
    double rez;
    int i;
    rez = 1.0;
    for(i = 1; i <= n; i++) {
      rez *= x;
    }
    return rez;
  }
}
```

Sledeća naredba je naredba dodele. Njeno izvršavanje je prilično komplikovano zbog dva poziva funkcije St. Sada ćemo korak po korak objasniti kako se izračunava vrednost izraza na desnoj strani.

Prvo se računa St(x,10). Nakon poziva, metod St otvori svoje četiri fioke u memoriji: dve za argumente x i n, i dve za svoje promenljive rez i i.

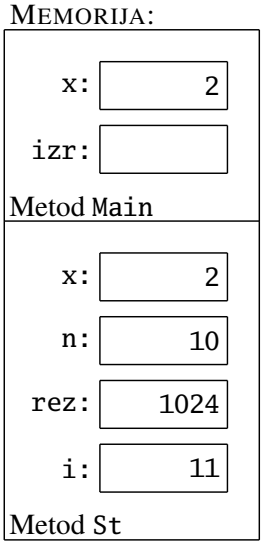
```
using System;
class Izraz {
  static void Main() {
    double x, izr;
    Console.Write("x -> ");
    x = double.Parse(
      Console.ReadLine());
    izr = 25 + St(x,10) - 0.5*St(x,13);
    Console.WriteLine(izr);
  }

  static double St(double x, int n) {
    ...
  }
}
```

Nakon toga se vrednost promenljive x u pozivu metoda prepiše u fioku x koja pripada metodu St, a broj 10 se upiše u fioku n metoda St.



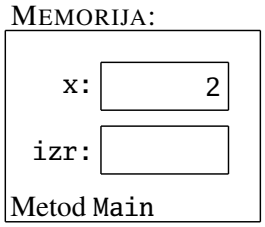

```
using System;
class Izraz {
  static void Main() {
    ...
    izr = 25 + St(x,10) - 0.5*St(x,13);
    ...
  }

  static double St(double x, int n) {
    double rez;
    int i;
    rez = 1.0;
    for(i = 1; i <= n; i++) {
      rez *= x;
    }
    return rez;
  }
}
```

Kada je prenos parametara gotov izvrši se metod St. Time se izračuna x^n, u našem primeru 2^{10}, i vrednost upiše u promenljivu rez.



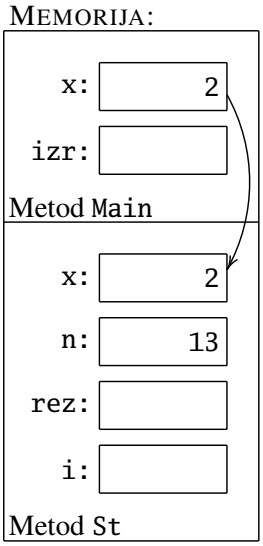

```
using System;
class Izraz {
  static void Main() {
    ...
    izr = 25 + St(x,10) - 0.5*St(x,13);
    ...
  }

  static double St(double x, int n) {
    double rez;
    int i;
    rez = 1.0;
    for(i = 1; i <= n; i++) {
      rez *= x;
    }
    return rez;
  }
}
```

Poslednjom naredbom u telu metoda se rezultat rada metoda vrati u izraz. Možemo da zamislimo da sada u izrazu umesto poziva metoda `St` stoji vrednost 1024. Kada se metod završi, kontrola se vrati glavnom programu, a iz memorije se uklone kućice koje je koristio metod `St`.



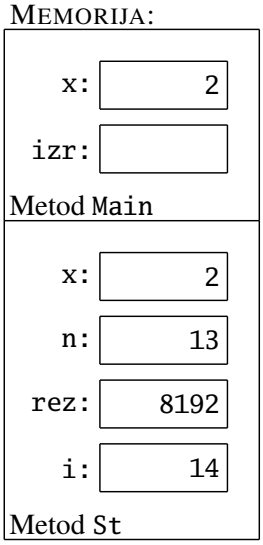

```
using System;
class Izraz {
  static void Main() {
    ...
    izr = 25 + 1024 - 0.5*St(x,13);
    ...
  }

  static double St(double x, int n) {
    double rez;
    int i;
    rez = 1.0;
    for(i = 1; i <= n; i++) {
      rez *= x;
    }
    return rez;
  }
}
```

Na isti način se izračuna vrednost `St(x,13)` u izrazu. Ponovi se ceo proces: metod `St` ponovo otvori svoje četiri fioke i prepiše odgovarajuće vrednosti.



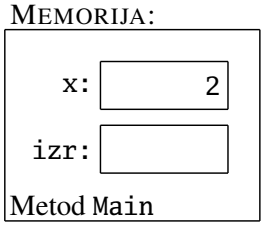

```
using System;
class Izraz {
  static void Main() {
    ...
    izr = 25 + 1024 - 0.5*St(x,13);
    ...
  }

  static double St(double x, int n) {
    double rez;
    int i;
    rez = 1.0;
    for(i = 1; i <= n; i++) {
      rez *= x;
    }
    return rez;
  }
}
```

Kada je prenos parametara gotov izvrši se metod `St`. Time se izračuna x^n, u našem primeru 2^{13}, i vrednost upiše u promenljivu `rez`.




```
using System;
class Izraz {
  static void Main() {
    ...
    izr = 25 + 1024 - 0.5* St(x,13) ;
    ...
  }

  static double St(double x, int n) {
    double rez;
    int i;
    rez = 1.0;
    for(i = 1; i <= n; i++) {
      rez *= x;
    }
    return rez;
  }
}
```

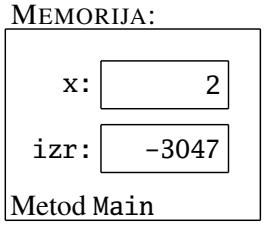
Poslednjom naredbom u telu metoda se rezultat rada metoda vrati u izraz. Možemo da zamislimo da sada u izrazu umesto poziva metoda `St` stoji vrednost 8192. Kada se metod završi, kontrola se vrati glavnom programu, a iz memorije se uklone kućice koje je koristio metod `St`.




```
using System;
class Izraz {
  static void Main() {
    ...
    izr = 25 + 1024 - 0.5* 8192 ;
    ...
  }

  static double St(double x, int n) {
    double rez;
    int i;
    rez = 1.0;
    for(i = 1; i <= n; i++) {
      rez *= x;
    }
    return rez;
  }
}
```

Sada su u izrazu na desnoj strani sve vrednosti poznate, računar izvrši naznačene operacije i dobijenu vrednost upiše u fioku `izr`.

MEMORIJA:




```
using System;
class Izraz {
  static void Main() {
    double x, izr;
    Console.Write("x -> ");
    x = double.Parse(
      Console.ReadLine());
    izr = 25 + 1024 - 0.5*8192;
    Console.WriteLine(izr);
  }

  static double St(double x, int n) {
    ...
  }
}
```

☞ Da rezimiramo, aktivacija (poziv) metoda se izvršava u nekoliko koraka:

- Metod rezerviše memorijski prostor za svoje argumente i lokalne promenljive. Argument ili lokalna promenljiva jednog metoda sme da se zove isto kao i promenljiva nekog drugog metoda jer metod gleda svoj memorijski prostor.
- Prenesu se argumenti: vrednosti izraza koji se javljaju u pozivu metoda se upišu u fioke dodeljene argumentima metoda.
- Izvrši se telo metoda. Ako se radi o metodu koji vraća vrednost, naredba `return` kaže šta će biti vraćeno kao rezultat rada metoda.
- Ako se radi o metodu koji vraća vrednost i poziv se javlja kao deo nekog izraza, poziv metoda se "zameni" rezultatom računanja funkcije. Važno je napomenuti da do fizičke zamene poziva vrednošću zapravo ne dolazi, ali nam je za sada lakše da ovako razmišljamo.
- Oslobodi se prostor rezervisan za argumente i privatne promenljive metoda.

Zadaci.

5.1. Napisati metod koji računa funkciju `sgn` (*signum* – znak, lat.) koja se definiše ovako:

$$\mathrm{sgn}(x) = \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ 1, & x > 0. \end{cases}$$

5.2. Napisati funkciju `BinKo(n, k)` koja računa vrednost binomnog koeficijenta

$$\binom{n}{k} = \frac{n \cdot (n-1) \cdot \ldots \cdot (n-k+1)}{k!}.$$

5.3. Kvadrati i trouglovi ⟨LINK⟩

5.4. Napisati C# program koji ispisuje sve *proste* delioce datog prirodnog broja n.

5.5. Napisati C# program koji od korisnika učitava prirodan broj n, a potom još n prirodnih brojeva za koje proveri da li su svi prosti.

5.6. Za prirodan broj kažemo da je palindrom ako se i sleva i zdesna čita na isti način (recimo, 1, 22, 121, 1331, 12521 su palindromi). Napisati C# program koji od korisnika učitava prirodan broj n i štampa sve brojeve k iz skupa $\{1, 2, \ldots, n\}$ takve da su i k i k^2 palindromi.

5.7. Napisati program koji ispisuje prvih n prirodnih brojeva koji su deljivi sa tačno dva od brojeva 2, 3, 5, 7.

5.8. Napisati program koji utvrđuje koliko dana je prošlo između dva datuma:

(a) ako su datumi u istoj godini;

(b) ako datumi ne moraju biti u istoj godini.

5.9. Napisati C# program koji od korisnika učitava cele brojeve g_1, g_2 i ako je $1582 \leqslant g_1 \leqslant g_2 \leqslant 9999$ ispisuje sve prestupne godine u intervalu $[g_1, g_2]$. U ostalim slučajevima ispisuje poruku o greški.

5.10. Napisati C# program koji od korisnika učitava dva cela broja g_1, g_2 tako da je $2001 \leqslant g_1 \leqslant g_2 \leqslant 9999$ i potom štampa sve godine iz intervala $[g_1, g_2]$ kojima je 1. januar ponedeljak. Zna se da je 2001. godine 1. januar bio ponedeljak.

5.11. Vreme zapisano u obliku *HH:MM:SS* možemo da shvatimo kao šestocifreni broj $\overline{HHMMSS}$. Na primer, 23:15:46 možemo da shvatimo kao broj 231 546. Napisati C# program koji određuje kada ima više prostih brojeva ovog oblika: pre podne (tj. od 00:00:01 do 11:59:59) ili posle podne (tj. od 12:00:01 do 23:59:59).

5.12. Napisati C# program koji za dati prirodan broj n određuje njegov najmanji i najveći prost faktor. Na primer, za $n = 14245$ program štampa brojeve 5 i 37 zato što je $14245 = 5 \cdot 7 \cdot 11 \cdot 37$.

5.13. Prosta baza prirodnog broja n je prirodan broj s koji ima iste proste faktore

kao i n, s tim da se svaki prost faktor broja n javlja kao faktor broja s tačno jednom. Na primer, za $n = 56 = 2^3 \cdot 7$ je $s = 2 \cdot 7 = 14$, a za $n = 42471 = 3^3 \cdot 11^2 \cdot 13$ je $s = 3 \cdot 11 \cdot 13 = 429$.

Napisati C# program koji od korisnika učitava prirodan broj $n \geqslant 2$ i računa i ispisuje njegovu prostu bazu. Za brojeve manje od 2 se prosta baza ne definiše.

5.14. Napisati C# program koji od korisnika učitava ceo broj $n \geqslant 2$ i utvrđuje da li je on proizvod uzastopnih prostih brojeva. Na primer, sledeći brojevi jesu proizvod uzastopnih prostih brojeva: $6 = 2 \cdot 3$, $2341 = 11 \cdot 13 \cdot 17$, $23 = 23$, a sledeći nisu: $12 = 2^2 \cdot 3$, $21 = 3 \cdot 7$, $286 = 2 \cdot 11 \cdot 13$.

5.15. Najbliži prost broj ⟨LINK⟩

5.16. Dopuna do punog kvadrata ⟨LINK⟩

5.17. Rastavljanje na proste činioce ⟨LINK⟩

5.18. Faktorizacija ⟨LINK⟩

5.19. Faktorizacija 2 ⟨LINK⟩

5.20. Fin raspored ⟨LINK⟩

5.21. Fin raspored 2 ⟨LINK⟩

5.22. Za prirodan broj n kažemo da je *super-složen* ako je on rekorder po broju delitelja, tj. ima više delitelja nego bilo koji prirodan broj manji od njega. Na primer, 60 je super-složen broj, zato što je to najmanji prirodan broj sa 12 delitelja. Prvih nekoliko super-složenih brojeva su: 2, 4, 6, 12, 24, 36, 48, 60, 120, 180.

Napisati C# program koji od korisnika učitava prirodan broj $n \geqslant 2$ i štampa sve super-složene brojeve iz skupa $\{1, 2, \ldots, n\}$.

5.23. Za broj q kažemo da je *pseudoprost* ako nije prost, ali $q | 2^q - 2$. Napisati C# program koji od korisnika učitava pozitivan ceo broj n i određuje njemu najbliži pseudoprost broj.

5.24. Krug je određen trojkom realnih brojeva (x, y, r), gde je (x, y) centar kruga, a r je poluprečnik kruga. Napisati C# program koji od korisnika učitava ceo broj n, potom n krugova (x_i, y_i, r_i) i određuje dužinu najdužeg mogućeg niza krugova koji ima sledeće osobine: na prvom mestu u nizu je prvi krug; na drugom mestu u nizu je prvi naredni krug koji je sadržan u prvom krugu; na trećem mestu u nizu je prvi naredni krug koji je sadržan u drugom krugu, itd.

5.25. Goldbahova hipoteza tvrdi da se svaki paran broj $\geqslant 4$ može predstaviti kao zbir dva prosta broja. Napisati program koji proverava Goldbahovu hipotezu za sve parne brojeve koji ne prelaze dati broj n tako što za svaki od njih ispisuje jednu reprezentaciju u obliku zbira dva prosta broja.

5.26. Napisati program koji učitava broj iz skupa

$$\{-2\,000\,000\,000, \ldots, -1, 0, 1, \ldots, 2\,000\,000\,000\}$$

i ispisuje taj broj rečima na engleskom jeziku. Na primer, za $-1\,328\,451\,820$ program treba da ispiše:

```
minus one billion three hundred twenty eight million
four hundred fifty one thousand eight hundred twenty
```

5.27. Parovi prostih brojeva ⟨LINK⟩

5.28. Da li je potpun kvadrat ⟨LINK⟩

5.29. A na N mod M ⟨LINK⟩

5.30. Igra sa brojevima ⟨LINK⟩

5.31. Glista ⟨LINK⟩

5.32. NZD dva proizvoda ⟨LINK⟩

5.33. Loptica u pravougaoniku ⟨LINK⟩

5.6 Prenos argumenata po referenci

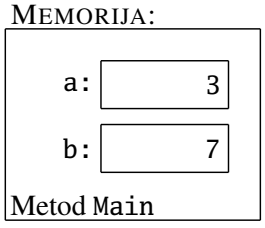
Pretpostavimo da želimo da napišemo metod `void Razmeni(int x, int y)` koji treba da razmeni vrednosti svojih argumenata. Ako to napišemo ovako:

```
// OVO NIJE DOBRO RESENJE!
static void Razmeni(int x, int y) {
  int pom = x;
  x = y;
  y = pom;
}
```

dobićemo metod koji *ne radi svoj posao*, kako se vidi prilikom detaljne analize sledećeg primera:

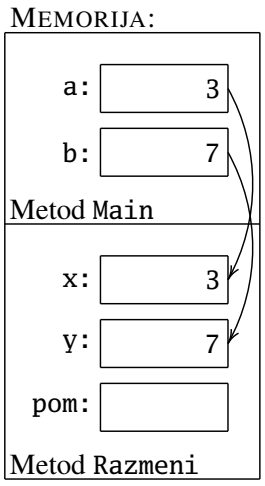
```
static void Main() {
  int a = int.Parse(Console.ReadLine());
  int b = int.Parse(Console.ReadLine());
  Razmeni(a, b);
  Console.WriteLine("{0}, {1}", a, b);
}

static void Razmeni(int x, int y) {
  int pom = x;
  x = y;
  y = pom;
}
```

Kada startujemo program, metod `Main` alocira prostor za promenljive `a` i `b` i učita njihove vrednosti. Recimo da je korisnik uneo 3 i 7. Tada je stanje u memoriji ovakvo:



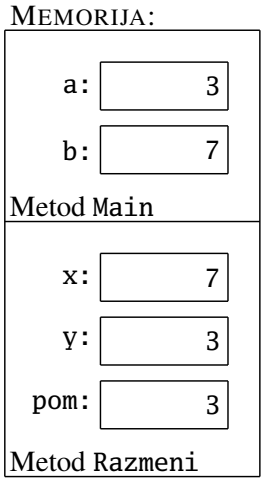

```
static void Main() {
  int a = int.Parse(
     Console.ReadLine());
  int b = int.Parse(
    Console.ReadLine());
  Razmeni(a, b);
  Console.WriteLine("{0}, {1}", a, b);
}

static void Razmeni(int x, int y) {
  int pom = x;
  x = y;
  y = pom;
}
```

Nakon toga se iz metoda `Main` pozove metod `Razmeni` i prenesu se parametri:



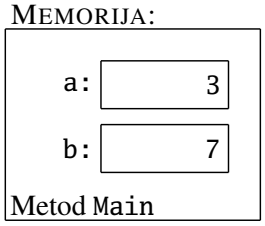

```
static void Main() {
  int a = int.Parse(
     Console.ReadLine());
  int b = int.Parse(
    Console.ReadLine());
  Razmeni(a, b);
  Console.WriteLine("{0}, {1}", a, b);
}

static void Razmeni(int x, int y) {
  int pom = x;
  x = y;
  y = pom;
}
```

Kada se izvrši telo metoda `Razmeni` promenljive `x` i `y` metoda `Razmeni` razmene vrednosti, *ali se nikakva promena nije desila na promenljivim* `a` *i* `b` *metoda* `Main`*:*




```
static void Main() {
  int a = int.Parse(
     Console.ReadLine());
  int b = int.Parse(
    Console.ReadLine());
  Razmeni(a, b);
  Console.WriteLine("{0}, {1}", a, b);
}

static void Razmeni(int x, int y) {
  int pom = x;
  x = y;
  y = pom;
}
```

Kada se metod Razmeni završi, iz memorije se uklanjaju njegove promenljive, a stanje promenljivih a i b je nepromenjeno:

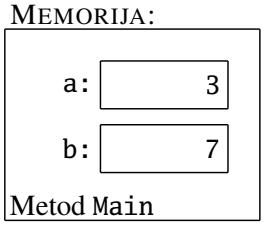



```
static void Main() {
  int a = int.Parse(
     Console.ReadLine());
  int b = int.Parse(
    Console.ReadLine());
  Razmeni(a, b);
  Console.WriteLine("{0}, {1}", a, b);
}

static void Razmeni(int x, int y) {
  int pom = x;
  x = y;
  y = pom;
}
```

Programski jezik C#, za razliku od sličnih programskih jezika kao što je Java, ima način da se argumenti prenesu u metod *po referenci*, tako da izmene koje se vrše unutar metoda budu globalno vidljive. Kada se argument prenosi po referenci potrebno je staviti reč ref i ispred argumenta u pozivu metoda, i ispred argumenta u deklaraciji metoda, ovako:

```
static void Main() {
  int a = int.Parse(Console.ReadLine());
  int b = int.Parse(Console.ReadLine());
  Razmeni(ref a, ref b);
  Console.WriteLine(0, 1", a, b);"
}

static void Razmeni(ref int x, ref int y) {
  int pom = x;
  x = y;
  y = pom;
}
```

Kada startujemo program, metod `Main` alocira prostor za promenljive `a` i `b` i učita njihove vrednosti. Recimo da je korisnik uneo 3 i 7. Tada je stanje u memoriji ovakvo:

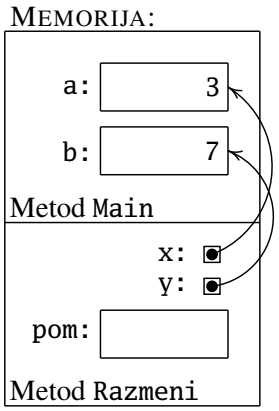



```
static void Main() {
  int a = int.Parse(
     Console.ReadLine());
  int b = int.Parse(
    Console.ReadLine());
  Razmeni(ref a, ref b);
  Console.WriteLine("{0}, {1}", a, b);
}

static void Razmeni(ref int x,
                    ref int y) {
  int pom = x;
  x = y;
  y = pom;
}
```

Ovaj put prenos parametara radi potpuno drugačije: metod `Razmeni` alocira prostor za promenljivu `pom`, dok argumenti `x` i `y` metoda `Razmeni` "preklope" promenljive `a` i `b` metoda `Main`. Naime, umesto da se u metod `Razmeni` prenesu vrednosti promenljivih `a` i `b`, prenose se *reference* na fijoke koje su im u memoriji dodeljene.

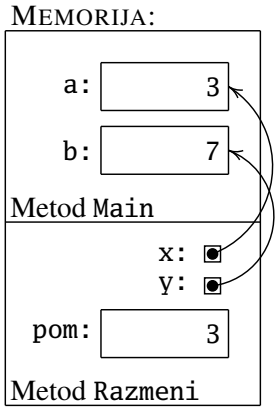
☞ *Referenca nije prava promenljiva sa svojom fijokom za vrednosti, već je to samo jedna strelica koja pokazuje na fijoku neke druge promenljive.*

Tako rad sa promenljivim `x` i `y` u metodu `Razmeni` zapravo upravlja memorijskim prostorom metoda `Main`.



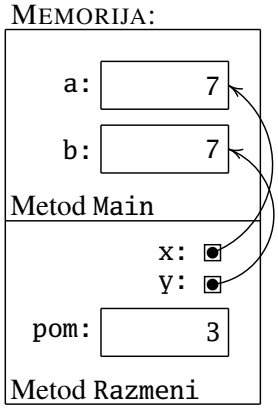

```
static void Main() {
  int a = int.Parse(
     Console.ReadLine());
  int b = int.Parse(
    Console.ReadLine());
  Razmeni(ref a, ref b);
  Console.WriteLine("{0}, {1}", a, b);
}

static void Razmeni(ref int x,
                    ref int y) {
  int pom = x;
  x = y;
  y = pom;
}
```

Pogledajmo sada kako se izvršava telo metoda `Razmeni`. Kada u metodu `Razmeni` napišemo `pom = x` to znači da će u fijoku koja odgovara promenljivoj `pom` biti upisana vrednost iz fijoke na koju se *referencira* `x`:



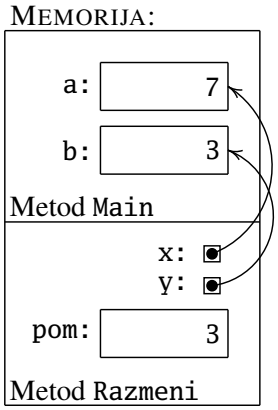

```
static void Main() {
  int a = int.Parse(
     Console.ReadLine());
  int b = int.Parse(
    Console.ReadLine());
  Razmeni(ref a, ref b);
  Console.WriteLine("{0}, {1}", a, b);
}

static void Razmeni(ref int x,
                    ref int y) {
  int pom = x;
  x = y;
  y = pom;
}
```

Naredba `x = y` znači da će vrednost iz fijoke na koju referencira `y` biti iskopirana u fijoku na koju referencira `y`:



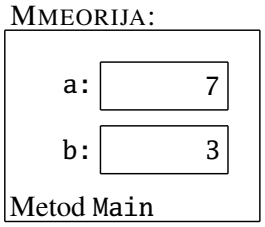

```
static void Main() {
  int a = int.Parse(
     Console.ReadLine());
  int b = int.Parse(
    Console.ReadLine());
  Razmeni(ref a, ref b);
  Console.WriteLine("{0}, {1}", a, b);
}

static void Razmeni(ref int x,
                    ref int y) {
  int pom = x;
  x = y;
  y = pom;
}
```

Konačno, naredba `y = pom` znači da će vrednost iz fijoke `pom` biti upisana u fijoku na koju referencira `y`:




```
static void Main() {
  int a = int.Parse(
     Console.ReadLine());
  int b = int.Parse(
    Console.ReadLine());
  Razmeni(ref a, ref b);
  Console.WriteLine("{0}, {1}", a, b);
}

static void Razmeni(ref int x,
                    ref int y) {
  int pom = x;
  x = y;
  y = pom;
}
```

Kada se metod `Razmeni` završi, iz memorije se uklanjaju njegove promenljive, a promenljive `a` i `b` su razmenile vrednosti kao što smo i želeli:




```
static void Main() {
  int a = int.Parse(
     Console.ReadLine());
  int b = int.Parse(
    Console.ReadLine());
  Razmeni(ref a, ref b);
  Console.WriteLine("{0}, {1}", a, b);
}

static void Razmeni(ref int x,
                    ref int y) {
  int pom = x;
  x = y;
  y = pom;
}
```

Vidimo da u programskom jeziku C# argumenti metoda mogu biti

- ulazni, i
- ulazno-izlazni.

Ulazni argumenti se deklarišu isto kao promenljive:

$$\langle tip \rangle \ \langle ime \rangle ;$$

Ulazno-izlazni argumenti se deklarišu navođenjem reči `ref` pre deklaracije:

$$\texttt{ref} \ \langle tip \rangle \ \langle ime \rangle ;$$

Zadaci.

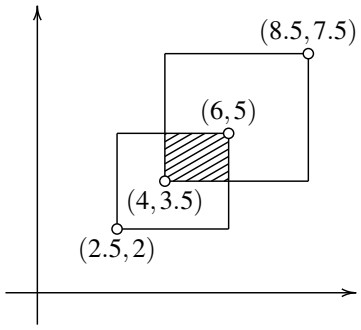
5.34. Napisati metod `void Uredi(ref int a, ref int b, ref int c);` koja preuredi svoje argumente tako da oni budu u neopadajućem redosledu: $a \leqslant b \leqslant c$.

5.35. Napisati C# program koji od korisnika učitava broj $n \geqslant 3$, a potom učitava n razlomaka i štampa njihov zbir. Razlomak se unosi u jednom redu kao par celih brojeva razdvojenih bar jednim razmakom. Razlomak koji se dobija kao rezultat treba da bude skraćen.

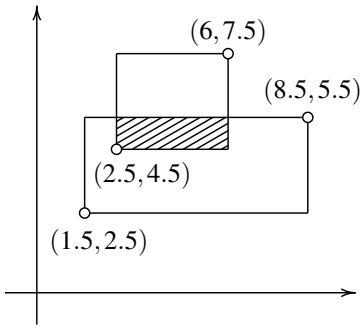
5.36. Kutija je određena trojkom realnih brojeva (x,y,z) koji predstavljaju dimenzije kutije. Napisati C# program koji od korisnika učitava ceo broj n, potom n kutija (x_i,y_i,z_i) i određuje dužinu najdužeg mogućeg niza kutija koji ima sledeće osobine: na prvom mestu u nizu je prva kutija; na drugom mestu u nizu je prva naredna kutija koja može da stane u prvu kutiju; na trećem mestu u nizu je prva naredna kutija koja može da stane u drugu kutiju, itd. Kutije se smeštaju jedna u drugu tako da im strane budu paralelne.

5.37. Duž na x-osi određena je svojim krajnjim tačkama koje su predstavljene kao brojevi, na primer ovako: $[7,12]$. Napisati C# program koji od korisnika učitava dve ovako zadate duži i određuje dužinu duži koja im se nalazi u preseku. Na primer, $[7,12] \cap [9,15] = [9,12]$ pa za ove dve duži program ispisuje 3, $[7,12] \cap [9,11] = [9,11]$ pa za ove dve duži program ispisuje 2, $[7,12] \cap [12,15] = [12,12]$ pa za ove dve duži program ispisuje 0, $[7,12] \cap [15,17] = \emptyset$ pa i ovde program ispisuje 0.

5.38. Pravougaonik čije stranice su paralelne koordinatnim osama određen je koordinatama donjeg levog ugla i koordinatama gornjeg desnog ugla. Napisati program koji od korisnika učitava dva ovako određena pravougaonika i određuje površinu njihovog preseka (ako se pravougaonici ne seku, odgovarajuća površina je 0).



Površina je 3



Površina je 3.5

Glava 6

Nizovi

Nizovi u programskom jeziku C# predstavljaju način da se efektivno radi sa velikim brojem podataka istog tipa. Na primer, problemi koji zahtevaju da se istovremeno obrađuje 1.000.000 brojeva bi se veoma teško mogli realizovati tako što bi se deklarisalo 1.000.000 različitih promenljivih i onda radilo sa njima. U ovoj glavi upoznajemo se sa nizovima kao strukturom podataka programskog jezika C#.

6.1 Deklaracija niza

Promenljive sa kojima smo do sada radili su imale osobinu da o njima brine računar: u deklaraciji promenljive se navede njen tip, a program sam kreira kućicu u odgovarajućem delu memorije kada se aktivira potprogram koji je deklarisao promenljivu, i sam oslobodi prostor u tom delu memorije kada se potprogram koji je deklarisao promenljivu završi. Takve promenljive se zovu *statičke* promenljive i prostor za njih se alocira u *statičkoj memoriji*.

Moderni programski jezici poznaju i drugu vrstu promenljivih, *dinamičke promenljive*, kod kojih o rezervaciji prostora brine programer, dok o oslobađanju prostora kada nam promenljiva više ne treba brine poseban podsistem koji se zove *garbage collector* (tj. đubretarac).

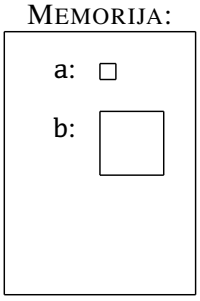
Nizovi u programskom jeziku C# se deklarišu ovako:

```
int[] a;
```

Ova deklaracija znači *da će* `a` *jednog dana biti niz celih brojeva*.

Obratite pažnju na to da postoji velika razlika između ove dve deklaracije:

```
int[] a;
int b;
```



Za promenljivu b je rezervisan prostor u memoriji u koji odmah može da se upiše neka vrednost, dok je za niz a rezervisan prostor u koji može da se upiše samo *referenca* na budući niz →

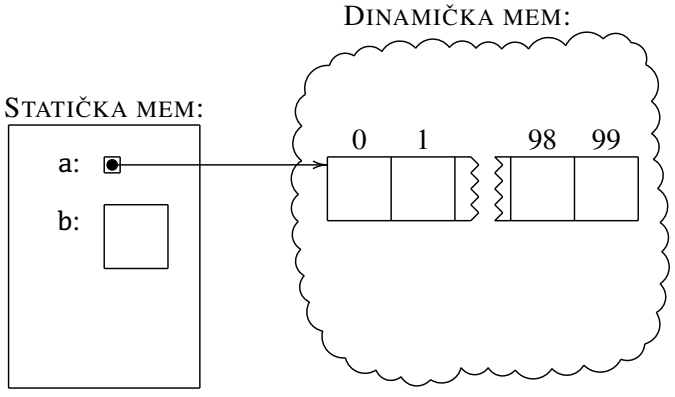
Postoje dva oblika organizacije memorije koje programski jezici koriste:

- *statička memorija* (koju smo do sada zvali prosto "memorija") je deo memorije u kojoj kompajler rezerviše prostor za promenljive, i
- *dinamička memorija* (ili *heap*, što na engleskom znači gomila, hrpa) koja predstavlja deo memorije u kojoj programer posebnom naredbom rezerviše prostor, kako ćemo sada objasniti.

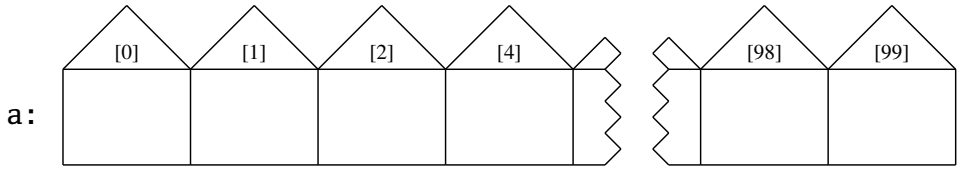
Nizovi i druge *strukture podataka* kojima se pristupa preko reference *ne postoje dok programer ručno ne rezerviše prostor za njih!* Kada naiđe na deklaraciju niza (objekta ili druge "referencirane promenljive") računar u *statičkoj memoriji* rezerviše samo prostor za *strelicu* koja će jednog dana pokazivati na sadržaj promenljive, a prostor za promenljivu mora da se rezerviše ručno. Naredba koja rezerviše prostor za nizove izgleda, na primer, ovako:

```
a = new int[100];
```

Naredba new u *dinamičkoj memoriji* rezerviše odgovarajući prostor, pa u promenljivu a stavi samo referencu na taj prostor. Tako, nakon gornje naredbe stanje memorije izgleda ovako:



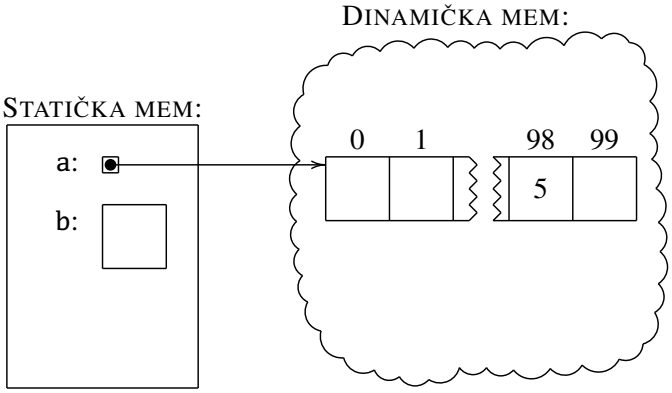
Promenljiva a sada označava (referencu na) složenu promenljivu koja se sastoji od 100 kućica ("malih" promenljivih), a svaka od tih kućica može da primi jedan ceo broj:



Promenljiva nizovnog tipa liči na ulicu: a je ime ulice, dok su [0], [1], itd. kućni brojevi. Pojedinačni elementi niza (tj. pojedinačne kućice) se ponašaju kao obične promeniljive tipa `int`, a zovu se ovako: `a[0]`, `a[1]`, ..., `a[99]`. Primetimo da "kućni brojevi" uvek počinju od 0 i idu do $n-1$ (ako je rezervisan prostor za n kućica). Na primer, ako želimo da u pretposlednju kućicu smestimo broj 5, to radimo komandom

```
a[98] = 5;
```

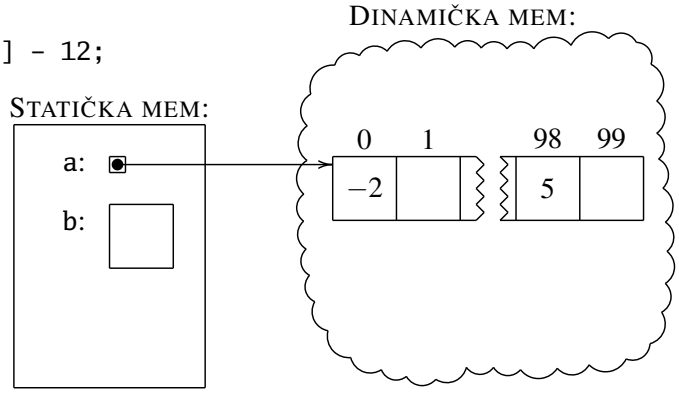
i efekt je sledeći:



dok nakon naredbe

```
a[0] = 2 * a[98] - 12;
```

imamo da je `a[0] = -2`, tj:



Možemo praviti nizove promenljivih bilo kog tipa: `int`, `double`, `bool`, kao bilo kog tipa koga ćemo učiti kasnije! Evo još nekoliko primera deklaracija nizova:

```
double[] x;
x = new double[200];

bool[] testiran;
testiran = new bool[150];
```

Kada smo alocirali prostor za niz, elemente niza možemo učitati, recimo, ovako:

```
double[] a; int i, n;
n = int.Parse(Console.ReadLine());
a = new double[n];
for(i = 0; i < n; i++) {
  a[i] = double.Parse(Console.ReadLine());
}
```

Kao što vidimo dužina niza ne mora biti poznata unapred već to može biti i vrednost neke promenljive zato što se prostor za niz rezerviše tokom izvršavanja programa.

☞ *Pošto u nizu dužine n kućice imaju "kućne brojeve" 0, 1, ..., $n-1$, najzgodniji način da se u* `for`*-ciklusu prođe kroz niz izgleda ovako:*

```
for(i = 0; i < n; i++) {
  ... a[i] ...
}
```

Ovo je veoma važan idiom koga treba zapamtiti!

Najveći element niza `a` možemo da nađemo ovako:

```
max = a[0];
for(i = 1; i < n; i++) {
  if(a[i] > max) { max = a[i]; }
}
```

a zbir elemenata niza ovako:

```
sum = 0;
for(i = 0; i < n; i++) {
  sum += a[i];
}
```

Primer. Napisati C# program koji od korisnika učitava ceo broj $n \geqslant 3$, potom niz od n realnih brojeva za koje se zna da su svi različiti (i to ne treba proveravati) i onda ispisuje učitani niz, ali uz preskakanje najvećeg i najmanjeg elementa niza.

⟨C# fajl⟩

```
using System;

class NizBezMinMax {
  static void Main() {
    double[] a; int i;
    Console.WriteLine("Unesi n");
    int n = int.Parse(Console.ReadLine());
    a = new double[n];

    Console.WriteLine("Unesi elemente niza (svi razliciti!)");
    for(i = 0; i < n; i++) {
      a[i] = double.Parse(Console.ReadLine());
    }

    double min = a[0];
    double max = a[0];
    for(i = 1; i < n; i++) {
      if(a[i] > max) { max = a[i]; }
      if(a[i] < min) { min = a[i]; }
    }

    Console.WriteLine("Niz bez najveceg i najmanjeg elementa:");
    for(i = 0; i < n; i++) {
      if(a[i] != min && a[i] != max) {
        Console.WriteLine(a[i]);
      }
    }
  }
}
```

Primer. Nizovi se često koriste kada je potrebno računati sa brojevima koji imaju mnogo cifara ili mnogo decimala. Navodimo primer programa koji računa decimale broja π koristeći Spigot algoritam[1]. Nije lako objasniti zašto ovaj algoritam radi bez dubljeg poznavanja tajni matematike, tako da ga nećemo detaljno komentarisati.

[1] S. Rabinowitz and S. Wagon, A spigot algorithm for the digits of π, Amer. Math. Monthly 102 (1995) 195–203.

⟨C# fajl⟩

```
using System;

class MnogoDecimalaBrojaPi {
  static void Main() {
    Console.Write("Broj decimala (bar 2): ");
    int n = int.Parse(Console.ReadLine());
    if(n < 2) { Console.WriteLine("Bar 2 decimale!"); }
    else {
      n++;
      int duzina = 10 * n / 3;
      int[] a = new int[duzina];
      for(int i = 0; i < duzina; i++) { a[i] = 2; }
      int devetke = 0;
      int cifra = 0;
      for(int j = 0; j < n; j++) {
        int q = 0;
        for(int i = duzina - 1; i >= 0; i--) {
          int x = 10*a[i] + q*(i + 1);
          a[i] = x % (2*i + 1);
          q = x / (2*i + 1);
        }
        a[0] = q % 10; q = q / 10;
        if (q == 9) { devetke++; }
        else if (q == 10) {
          Console.Write(cifra+1);
          for(int k = 0; k < devetke; k++) { Console.Write(0); }
          cifra = 0; devetke = 0;
        }
        else {
          if (j == 1) { Console.Write("{0}.", cifra); }
          else if (j >= 2) { Console.Write(cifra); }
          cifra = q;
          if (devetke != 0) {
            for(int k = 0; k < devetke; k++) { Console.Write(9); }
            devetke = 0;
          }
        }
      }
      Console.WriteLine(cifra);
    }
  }
}
```

Kao što smo već rekli, indeksi niza (“kućni brojevi kućica”) u C# uvek kreću od 0. To može ponekad biti nezgodno i iziskivati posebnu pažnju, kako je pokazano u sledećem primeru.

Primer. Učenici jedne srednje škole su anketirani kako bi se utvrdilo koliko su zadovoljni ukupnim stanjem u školi. Svaki učenik je dobio jedan anketni listić

Anketa
Ovo je *anonimna* anketa o tvom stavu o ukupnom stanju u školi! Na sledećoj skali zaokruži broj koji najbolje opisuje tvoj stav (-5 imam veoma negativno mišljenje, ..., 0 nemam stav, ..., 5 imam veoma pozitivno mišljenje)
-5 -4 -3 -2 -1 0 1 2 3 4 5

na kome treba da se zaokruži jedan od navedenih brojeva. Napisati C# program koji od korisnika učitava ceo broj n (broj anketiranih učenika), potom n brojeva iz skupa $\{-5,\ldots,-1,0,1,\ldots,5\}$ i ispisuje koliko se puta koji broj pojavio kao ocena.

⟨C# fajl⟩

```
using System;

class Anketa {
  static void Main() {
    int brUcenika, i, k;
    int[] frekv;

    // ocene na skali su -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5
    // i zato nam treba niz duzine 11;
    // frekv[0] broji ocene -5, frekv[1] broji ocene -4, itd.
    frekv = new int[11];
    for(i = 0; i < 11; i++) { frekv[i] = 0; }

    Console.Write("Broj anketiranih ucenika -> ");
    brUcenika = int.Parse(Console.ReadLine());

    Console.WriteLine("Unesite sada rezultate ankete:");
    for(i = 1; i <= brUcenika; i++) {
      Console.Write("Ucenik {0} -> ", i);
      k = int.Parse(Console.ReadLine());
      if(-5 <= k && k <= 5) {
        // frekv[0] broji ocene -5, frekv[1] broji ocene -4, itd.
        // dakle, frekv[k + 5] broji ocene k
```

```
        frekv[k + 5]++;
      }
      else {
        Console.WriteLine("Nekorektan podatak -- odbacuje se!");
      }
    }

    Console.WriteLine("Frekvencije ocena:");
    for(i = 0; i < 11; i++) {
      Console.WriteLine("Ocena {0} -> {1}", i - 5, frekv[i]);
    }
  }
}
```

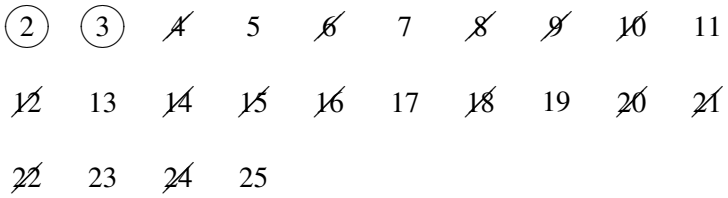
Primer: Eratostenovo sito. Posebno interesantan način za pronalaženje svih prostih brojeva iz skupa $\{2,3,\ldots,n\}$ pronašao je starogrčki naučnik Eratosten. Svi brojevi od 2 do n se ispišu u niz. Potom neke od njih zaokružujemo, a neke precrtavamo prema jednom veoma jednostavnom pravilu. Na kraju brojevi koji ostanu neprecrtani su prosti. Precrtavanje se radi ovako. Uočimo prvi broj koji nije zaokružen. Neka je to k. Zaokružimo k, a svaki k-ti broj počev od njega precrtamo. Evo primera za $n = 25$. Na početku ispišemo brojeve od 2 do 25:

2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21
22	23	24	25						

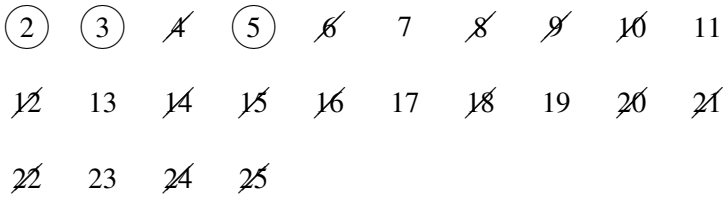
Prvi neprecrtani broj, a to je 2, zaokružimo i precrtamo svaki drugi broj počev od njega:

(2)	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~	11
~~12~~	13	~~14~~	15	~~16~~	17	~~18~~	19	~~20~~	21
~~22~~	23	~~24~~	25						

Prvi sledeći neprecrtani broj, a to je 3, zaokružimo i precrtamo svaki treći broj počev od njega:



Sledeći broj je 5, itd. Kao i ranije, dovoljno je raditi samo sa brojevima koji su manji ili jednaki sa $\lfloor\sqrt{n}\rfloor$. Na kraju dobijamo:



Brojevi koji nisu precrtani su prosti, nezavisno od toga da li su zaokruženi ili ne. Kako ceo postupak liči na prosejavanje, ovaj algoritam se zove *Eratostenovo sito*.

⟨C# fajl⟩

```
using System;

class EratostenovoSito {
  static void Main() {
    int N, i, k;
    bool[] Precrtan;
    Console.Write("Unesite broj (najmanje 5) -> ");
    N = int.Parse(Console.ReadLine());
    if(N <= 5) {
      Console.WriteLine("Broj nije >= 5");
    }
    else {
      // trebaju nam kucice sa indeksima 2, 3, ..., N
      // zato zauzimamo N + 1 mesta;
      // kucice sa indeksima 0 i 1 necemo koristiti
      Precrtan = new bool[N + 1];
      for(k = 2; k <= N; k++) { Precrtan[k] = false; }
      for(k = 2; k <= (int)Math.Sqrt(N); k++) {
        if(!Precrtan[k]) {
            // precrtavanje umnozaka
            for(i = 2 * k; i <= N; i += k) { Precrtan[i] = true; }
        }
      }
```

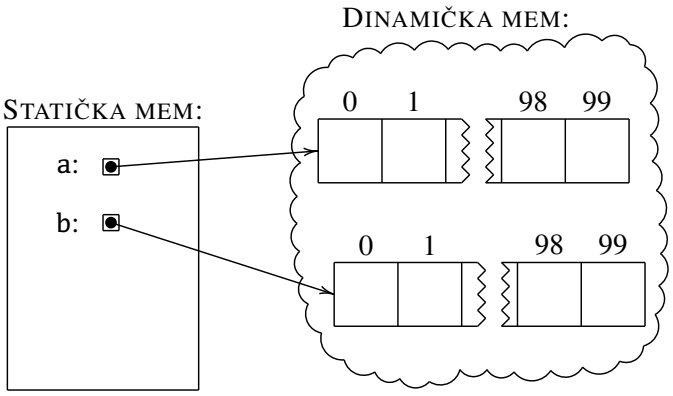
```
      Console.WriteLine("Prosti brojevi su:");
      for(k = 2; k <= N; k++) {
        if(!Precrtan[k]) { Console.Write("{0}  ", k); }
      }
      Console.WriteLine();
    }
  }
}
```

6.2 Dodeljivanje i jednakost nizova

Neka su a i b dva niza deklarisana na sledeći način:

```
int[] a, b;
a = new int[100];
b = new int[100];
```

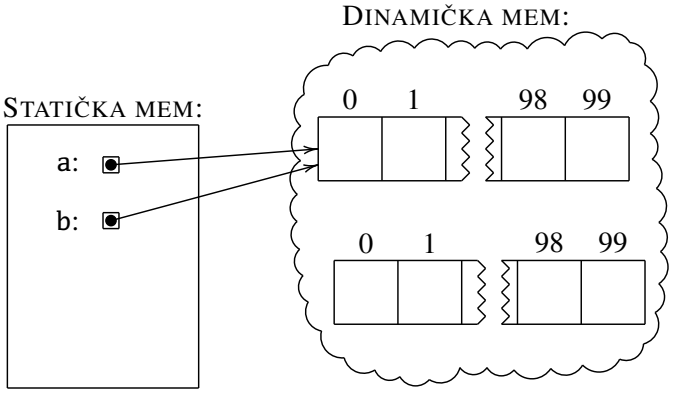
Tada stanje u memoriji izgleda ovako:



Činjenica da su a i b reference (dakle, strelice) ima važne posledice! Pre svega, naredba

```
b = a;
```

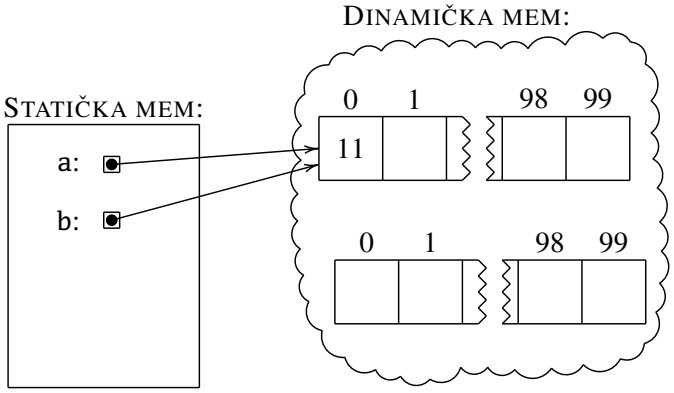
znači da referenca b dobija vrednost reference a, pa nakon ove naredbe *strelica b pokazuje na onaj deo memorije na koji pokazuje strelica a.*



Prema tome, niz naredbi

```
a[0] = 11;
Console.WriteLine(b[0]);
```

će ispisati 11, jer je sada stanje u memoriji ovakvo:



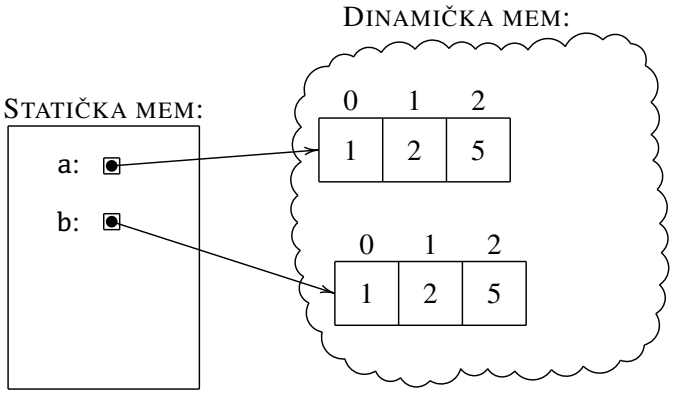
Slično tome, ako su a i b nizovi, tada

```
a == b
```

proverava da li reference a i b pokazuju na isti deo memorije. Na primer, nakon niza naredbi

```
int[] a, b;
a = new int[3];
b = new int[3];
a[0] = 1; a[1] = 2; a[2] = 5;
b[0] = 1; b[1] = 2; b[2] = 5;
```

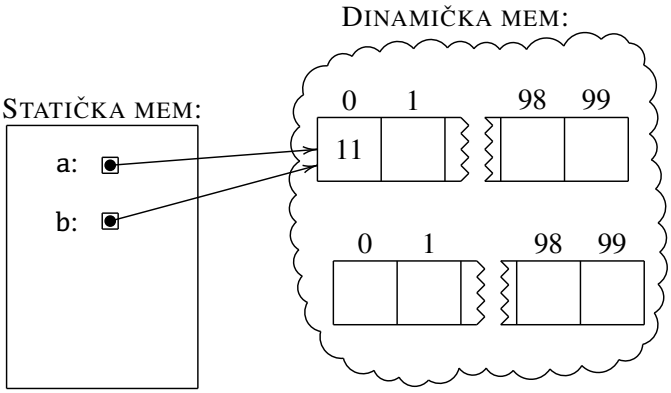
stanje u memoriji izgleda ovako:



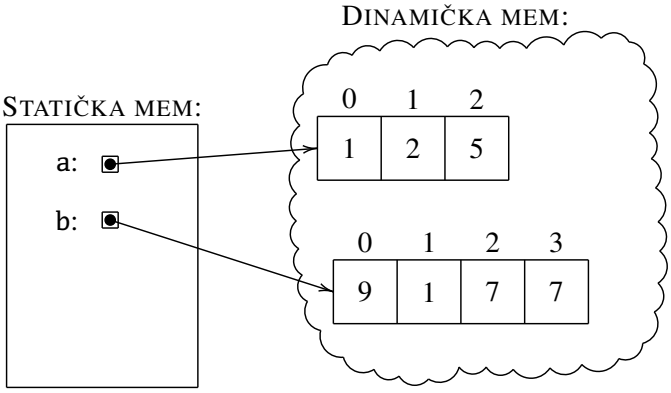
U ovoj situaciji je `a != b` iako su `a` i `b` "isti nizovi"! Razlog je u tome što strelice `a` i `b` pokazuju na različite delove memorije!

☞ *Dakle, za nizove* `a` *i* `b` *test* `a == b` *proverava da li su* `a` ***i*** `b` ***reference na isti deo meorije!***

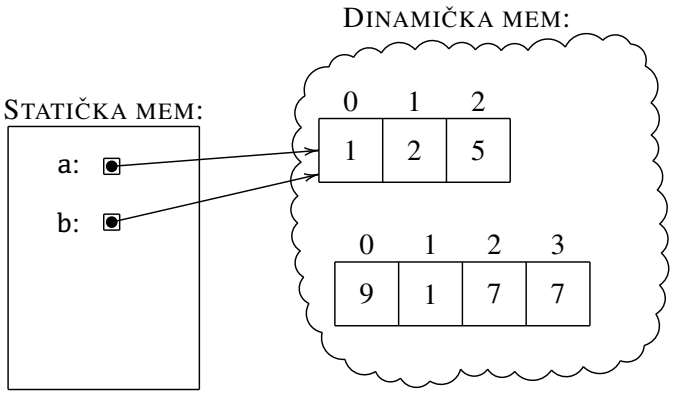
Da bi važilo `a == b` situacija mora da izgleda ovako:



Ovakav pristup nizovima (da promenljiva `a` sadrži referencu na niz) dovodi do situacija u kojima može da se desi da posle nekih naredbi dodele u dinamičkoj memoriji može da ostane đubre, odnosno, podaci koji zauzimaju prostor, a do kojih nikako ne možemo da dobacimo. O ovakvim situacijama automatski brine podsistem koji se zove *garbage collector* ili "đubretarac". Na primer, neka je situacija u memoriji ovakva:



Nakon naredbe `b = a;` situacija postaje ovakva:



Ako nijedna druga strelica ne pokazuje na niz $9, 1, 7, 7$ on predstavlja đubre koje će u nekom trenutku pokupiti đubretarac i osloboditi deo dinamičke memorije koju je ovaj niz zauzimao. Zato se ova memorija i zove *dinamička memorija*: tu ima mnogo dinamike oko rezervisanja i oslobađanja prostora.

U prethodnom primeru smo videli i da može da se desi da je `b` bio niz dužine 4, a da nakon naredbe dodele to postane niz neke druge dužine (recimo, 3). To programeru otežava evidenciju o dužini niza, pa da bi se izbegle nesuglasice i smanjila mogućnost greške u C# svaki niz pamti svoju dužinu. Do dužine niza `a` se dolazi ovako:

```
a.Length
```

Na primer programski fragment

```
int[] a, b;
a = new int[3];
b = new int[4];
b = a;
Console.WriteLine(b.Length);
```

ispisuje 3: iako je `b` u početku bio niz dužine 4, nakon naredbe `b = a;` on postaje niz dužine 3.

Sada možemo da pokažemo kako se proverava da li nizovi `a` i `b` imaju *isti sadržaj*:

```
if(a.Length != b.Length) {
  Console.WriteLine("Nemaju isti sadrzaj");
}
else {
```

```
  int i; bool ok;
  ok = true;
  for(i = 0; i < a.Length; i++) {
    if(a[i] != b[i]) { ok = false; break; }
  }
  if(ok) { Console.WriteLine("Imaju isti sadrzaj"); }
  else { Console.WriteLine("Nemaju isti sadrzaj"); }
}
```

☞ *U ovom primeru vidimo da promenljiva može biti deklarisana unutar bloka određenog vitičastim zagradama* `{ ... }`. *Ukoliko to uradimo važno je da znamo da tako deklarisana promenljiva "živi" samo unutar navedenog bloka: izvan bloka kompajler ne zna da ona postoji.*

Ova konvencija se često koristi kod "for"-ciklusa jer se upravo tu često dešava da nam kontrolna promenljiva treba samo za potrebe ciklusa. Prethodni primer se, shodno tome, može napisati i ovako:

```
if(a.Length != b.Length) {
  Console.WriteLine("Nemaju isti sadrzaj");
}
else {
  bool ok = true;
  for(int i = 0; i < a.Length; i++) {
    if(a[i] != b[i]) { ok = false; break; }
  }
  if(ok) { Console.WriteLine("Imaju isti sadrzaj"); }
  else { Console.WriteLine("Nemaju isti sadrzaj"); }
}
```

Ovde je važno napomenuti da promenljiva `i` ne postoji izvan tela "for"-ciklusa u kome je deklarisana. Takođe vidimo i mogućnost da se promenljivoj dodeli vrednosti prilikom deklaracije. Tako je

```
bool ok = true;
```

isto što i

```
bool ok;
ok = true;
```

6.3 Inicijalizacija malih nizova i "foreach" ciklus

Videli smo da je moguće promenljivoj dodeliti vrednost prilikom deklaracije tipa. To je moguće i u slučaju deklaracije nizova, i tada se vrednost niza promenljivoj dodeljuje ovako:

```
int[] ProstiDo100 = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
                     31, 37, 41, 43, 47, 53, 59, 61, 67,
                     71, 73, 79, 83, 87, 89, 93, 97};
```

Prethodna naredba deklariše niz celih brojeva koji se zove `ProstiDo100` i odmah mu postavlja dužinu na 27 i elementima dodeljuje vrednosti prema spisku. Ovo je isto kao da smo napisali sledeći kôd (samo kraće i elegantnije):

```
int[] ProstiDo100 = new int[27];
ProstiDo100[0] = 2;
ProstiDo100[1] = 3;
// ... itd ...
ProstiDo100[26] = 97;
```

Ukoliko želimo da ispišemo elemente niza `ProstiDo100`, možemo to učiniti kako smo već pokazali:

```
for(int i = 0; i < ProstiDo100.Length; i++) {
  Console.WriteLine(ProstiDo100[i]);
}
```

Među tim, postoji i elegantniji način — "foreach" naredba:

```
foreach(int p in ProstiDo100) {
  Console.WriteLine(p);
}
```

Značenje je sledeće:

> *"za svaki element* `p` *sa spiska* `ProstiDo100` *uradi sledeće: . . . ".*

Tako dobijamo visok nivo apstrakcije, skoro kao da pišemo matematičke izjave:

C#	Matematika
`int[] ProstiDo100 = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 87, 89, 93, 97};`	$ProstiDo100 = \{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 87, 89, 93, 97\}$
`foreach(int p in ProstiDo100) { Console.WriteLine(p); }`	Za svaki $p \in ProstiDo100$ ispiši p u posebnom redu

Prethodni "foreach" ciklus je ekvivalentan sledećem "običnom for" ciklusu:

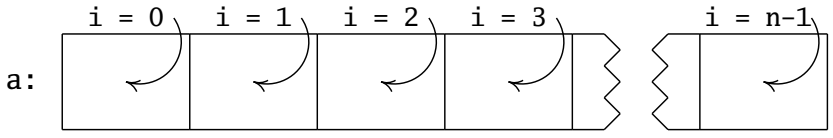
```
for(int i = 0; i < ProstiDo100.Length; i++) {
  int p = ProstDo100[i];
  Console.WriteLine(p);
}
```

☞ *"For"-ciklus iterira po **indeksima** niza dok "foreach"-ciklus iterira po **elementima** niza. Zato "foreach"-ciklus ne možemo da koristimo za učitavanje elemenata niza!*

Elemente niza moramo da učitamo na uobičajeni način:

```
double[] a = new double[n];
for(int i = 0; i < n; i++) {
  a[i] = double.Parse(Console.ReadLine());
}
```

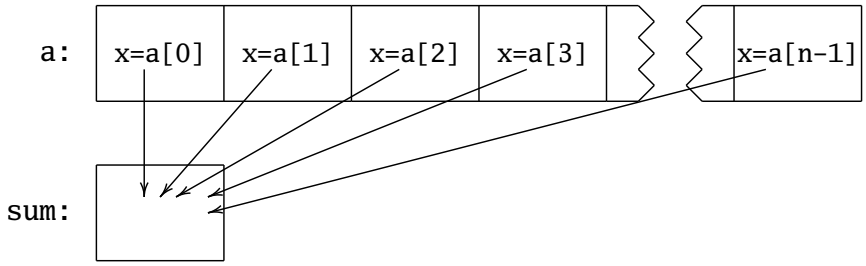
Gornji ciklus iterira po *indeksima* niza i u odgovarajuće kućice upisuje brojeve koje je uneo korisnik:



Kada su brojevi učitani u niz možemo da koristimo "foreach"-ciklus da bismo obradili te podatke. Na primer, zbir elemenata niza možemo da izračunamo ovako:

```
double sum = 0;
foreach(double x in a) {
  sum += x;
}
```

Ovaj ciklus iterira po *elementima* niza koje, jednog po jednog, dodaje na promenljivu `sum`:



"Foreach" je relativno ćudljiva naredba, tako da kod nje *mora* da se navede deklaracija tipa promenljive koja redom uzima vrednosti elemenata niza, i pri tome promenljva sa istim imenom ne sme da bude deklarisana ni u jednom bloku koji okružuje "foreach" ciklus.

Primer. Napisati C# program koji utvrđuje da li je sledeća formula tautologija:

$$\neg(p \wedge q \wedge r) \Longrightarrow (\neg p \vee \neg q \vee \neg r)$$

⟨C# fajl⟩

```
using System;

class Tautologija {
  static void Main() {
    // proveravamo da li je ova formula tautologija:
    //     !(p && q && r) => (!p || !q || !r)
    // kako C# nema implikaciju proveravamo njoj ekviv. formulu:
    //     (p && q && r) || (!p || !q || !r)
    bool[] LogVrednosti = {false, true};
    bool tautologija = true;
    foreach(bool p in LogVrednosti) {
      foreach(bool q in LogVrednosti) {
        foreach(bool r in LogVrednosti) {
          tautologija &= (p && q && r) || (!p || !q || !r);
        }
      }
    }
    if(tautologija) {
      Console.WriteLine("Formula jeste tautologija");
    }
    else {
      Console.WriteLine("Formula nije tautologija");
```

```
        }
    }
}
```

Zadaci.

6.1. Ispis u obratnom redosledu ⟨LINK⟩

6.2. Parni i neparni elementi ⟨LINK⟩

6.3. Izbacivanje elemenata ⟨LINK⟩

6.4. Napisati C# program koji od korisnika učitava prirodan broj n, potom niz od n realnih brojeva i utvrđuje koliko njih je strogo iznad proseka svih učitanih brojeva.

6.5. Napisati C# program koji od korisnika učitava prirodan broj n, potom niz od n realnih brojeva i utvrđuje indeks i vrednost prvog člana u nizu koji je najbliži proseku elemenata niza.

6.6. Translacija tačaka ⟨LINK⟩

6.7. Prosečno odstupanje od minimalnog ⟨LINK⟩

6.8. Minimalno odstupanje od proseka ⟨LINK⟩

6.9. Napisati C# program koji od korisnika učitava cele brojeve iz skupa $\{1, 2, 3, \ldots, 1000\}$ sve dok korisnik ne unese reč `kraj` i potom štampa koliko se puta koji broj pojavio.

6.10. Napisati C# program koji od korisnika učitava brojeve iz skupa $\{-10, \ldots, -1, 0, 1, \ldots, 10\}$ i broji koliko se puta svaki od njih pojavio. Kraj unosa označen je brojem 99 koji se, naravno, ne računa.

6.11. Broj pojavljivanja ⟨LINK⟩

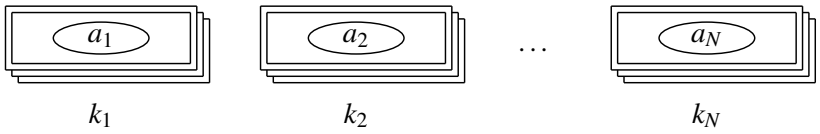
6.12. Napisati C# program koji od korisnika učitava prirodan broj n i utvrđuje koliko se puta koja cifra javlja u njegovom zapisu, tj. koliko ima nula, koliko jedinica, koliko dvojki, itd, koliko devetki.

6.13. Napisati C# program koji traži sve moguće načine da se dešifruje jednakost

$$***+***=***,$$

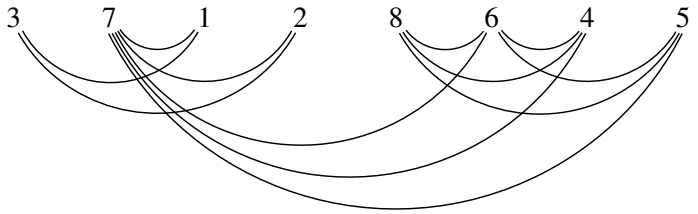
gde zvezdice označavaju proizvoljne cifre, i pri tome se svaka od cifara 1, 2, 3, 4, 5, 6, 7, 8, 9 javlja tačno jednom.

6.14. Na raspolaganju imamo k_1 novčanica u vrednosti od a_1 dinara, k_2 novčanica u vrednosti od a_2 dinara, $\ldots$, k_N novčanica u vrednosti od a_N dinara, i podaci su tako organizovani da je $a_1 > a_2 > \ldots > a_N$.



Napisati C# program koji od korisnika učitava N, potom parove (k_1, a_1), $\ldots$, (k_N, a_N) i na kraju pozitivan ceo broj m koji predstavlja količinu novca, a onda "isplaćuje" korisniku iznos od m dinara ovako: prvo isplati najviše što može novčanica od a_1 dinara, onda isplati najviše što može novčanica od a_2 dinara, i tako redom. Ukoliko je nakon isplate ostao iznos koji ne može da bude isplaćen ovim algoritmom, program obaveštava korisnika o iznosu koji preostaje.

6.15. *Inverzija* u nizu $a_1, a_2, \ldots, a_n$ je svaki par indeksa (i, j) takav da je $i < j$ i $a_i > a_j$. Na primer, na sledećoj slici su označene sve inverzije niza brojeva $3, 7, 1, 2, 8, 6, 4, 5$:



Napisati C# program koji od korisnika učitava ceo broj n, potom n brojeva i onda utvrđuje i štampa broj inverzija u tom nizu.

6.16. Dato je n tačaka u ravni $A_1(x_1, y_1)$, $\ldots$, $A_n(x_n, y_n)$ svojim koordinatama. Napisati C# program koji određuje dužinu najduže duži čiji krajevi su neke od ovih tačaka, kao i redne brojeve tačaka koje postižu ovu dužinu. Rastojanje tačaka $A(x_1, y_1)$ i $B(x_2, y_2)$ dato je formulom

$$d(A, B) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

6.17. Dato je n tačaka u ravni $A_1(x_1, y_1)$, $\ldots$, $A_n(x_n, y_n)$ svojim koordinatama. Napisati C# program koji određuje površinu najvećeg trougla sa temenima u ovim tačkama i štampa površinu, kao i redne brojeve temena koji čine taj trougao. Ako su $A(x_1, y_1)$, $B(x_2, y_2)$, $C(x_3, y_3)$ temena trougla onda je

$$P(\triangle ABC) = \frac{1}{2}|(x_2 y_3 - x_3 y_2) - (x_1 y_3 - x_3 y_1) + (x_1 y_2 - x_2 y_1)|.$$

6.18. Za prirodan broj p kažemo da je *period niza* $(a_0,\ldots,a_{n-1})$ ako je $a_{i+p}=a_i$ za sve $i\in\{0,\ldots,n-1-p\}$. Napisati C# program koji od korisnika učitava niz realnih brojeva, potom ceo broj p i proverava da li je p period učitanog niza.

6.19. Za niz $(a_0,\ldots,a_{n-1})$ kažemo da je *periodičan* ako postoji broj p koji je period tog niza (vidi Zadatak 6.18). Napisati C# program koji od korisnika učitava niz realnih brojeva i proverava da li je on periodičan. Ako je niz periodičan program ispisuje dužinu perioda, a ako nije ispisuje poruku `NIJE`.

6.20. *Palindromičnost niza* $(a_0,a_1,\ldots,a_{n-1})$ je najveći nenegativan ceo broj $k\leqslant\frac{n}{2}$ sa osobinom

$$a_0=a_{n-1},\quad a_1=a_{n-2},\quad a_2=a_{n-3},\quad \ldots,\quad a_{k-1}=a_{n-k}.$$

Napisati C# program koji od korisnika učitava pozitivan ceo broj n, potom niz $(a_0,a_1,\ldots,a_{n-1})$ celih brojeva dužine n i određuje i štampa njegovu palindromičnost.

6.21. Napisati C# program koji od korisnika učitava prirodan broj n i potom ispisuje prvih n prostih brojeva od najvećeg ka najmanjem. Na primer, za $n=10$ program ispisuje

29 23 19 17 13 11 7 5 3 2

6.22. Eratostenovo sito ⟨LINK⟩

6.23. Ciklično pomeranje za jedno mesto ⟨LINK⟩

6.24. Različiti elementi niza ⟨LINK⟩

6.25. Neobrisani brojevi ⟨LINK⟩

6.26. Vrednost polinoma ⟨LINK⟩

6.27. Različite cifre ⟨LINK⟩

6.28. Svega sa različitim ciframa ⟨LINK⟩

6.29. Dvojke i trojke deljive sa 3 ⟨LINK⟩

6.30. Histogram ⟨LINK⟩

6.31. Suma prethodna tri ⟨LINK⟩

6.32. Ruter ⟨LINK⟩

6.33. Popunjavanje praznina ⟨LINK⟩

6.34. Dvobojka ⟨LINK⟩

6.35. Trobojka ⟨LINK⟩

6.36. Broj parova datog zbira ⟨LINK⟩

6.37. Trojke datog zbira (3sum) ⟨LINK⟩

6.38. Razlika visina ⟨LINK⟩

6.39. Segment datog zbira u nizu prirodnih brojeva ⟨LINK⟩

6.4 Metodi i nizovi

Argument nekog metoda može da bude niz, ali tada moramo da vodimo računa da se u statičkoj memoriji čuva samo referenca na niz, što ima posledice na ponašanje metoda. Metod može i da vrati niz kao rezultat svog rada. U ovom odeljku ćemo objasniti šta se dešava kada se niz prosledi kao argument metoda, ili kada metod vraća niz kao rezultat svog rada. Posmatrajmo sledeća tri metoda:

```
static double Prosek(double[] A, int p, int L) {
  double suma = 0.0;
  for(int i = 0; i < L; i++) {
    suma += A[p + i];
  }
  return suma / L;
}

static double[] Segment(double[] A, int p, int L) {
  double[] B = new double[L];
  for(int i = 0; i < L; i++) {
    B[i] = A[p + i];
  }
  return B;
}

static void Napuni(int[] A, int n) {
  for(int i = 0; i < A.Length; i++) {
    A[i] = n;
  }
}
```

Metod `Prosek` prihvata niz `A` realnih brojeva i dva cela broja, `p` i `L`, i računa prosek segmenta niza `A` koji počinje na mestu `p` i ima dužinu `L`. Drugim rečima, metod

Prosek vraća *vrednost izraza*

$$\frac{A[p]+A[p+1]+\ldots+A[p+(L-1)]}{L}.$$

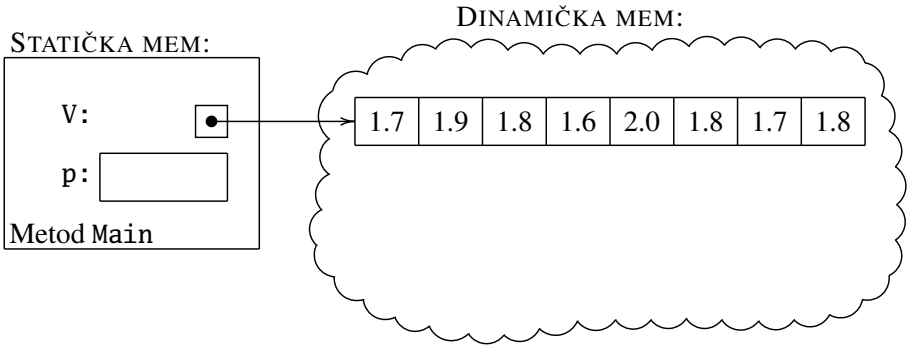
Metod Segment prihvata niz A realnih brojeva i dva cela broja, p i L, i kao rezultat svog rada vraća segment niza A koji počinje na mestu p i ima dužinu L kao novi niz realnih vrednosti. Drugim rečima, metod Segment vraća *niz vrednosti*

$$\{A[p],A[p+1],\ldots,A[p+(L-1)]\}.$$

Metod Napuni popuni sve elemente niza A brojem koji je prosleđen kao argument n.

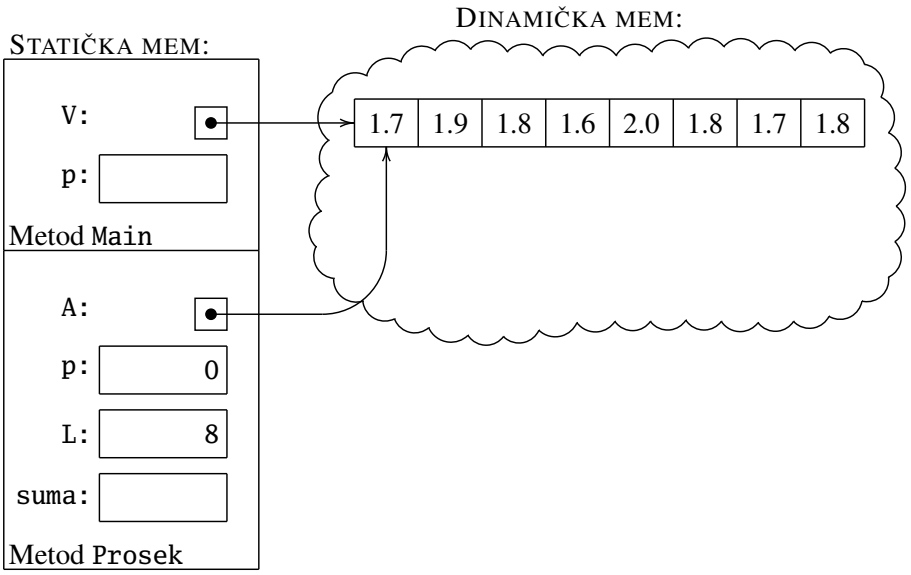
Metod Prosek. Pogledajmo na jednom primeru kako radi metod Prosek. Program, kao i uvek, krene od izvršavanja metoda Main u kome su deklarisane dve promenljive: niz realnih vrednosti V i realna promenljiva p.

```
static void Main() {
  double[] V = { 1.7, 1.9, 1.8, 1.6, 2.0, 1.8, 1.7, 1.8 };
  double p = Prosek(V, 0, V.Length);
  Console.WriteLine(p);
}

static double Prosek(double[] A, int p, int L) {
  double suma = 0.0;
  for(int i = 0; i < L; i++) {
    suma += A[p + i];
  }
  return suma / L;
}
```



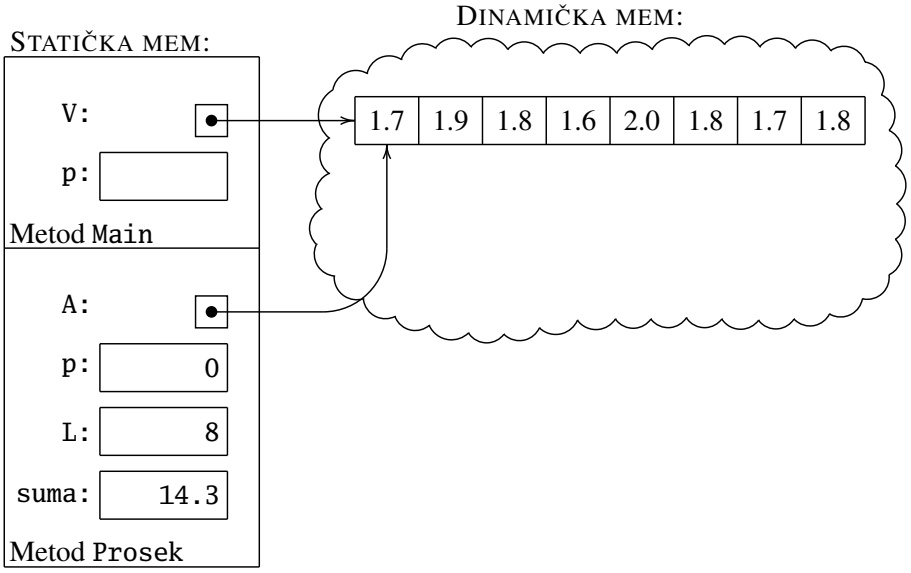
Poziv metoda `Prosek` prvo otvori nove fioke u memoriji za promenljive `A`, `p`, `L` i `suma` i iskopira vrednosti argumenata. (Promenljiva `i` je deklarisana unutar "for"-ciklusa i za nju će u memoriji biti alociran prostor tek kada krene da se izvršava ciklus.) Argument `p` metoda dobija vrednost 0, argument `L` vrednost 8 (jer je to dužina niza u primeru), dok se u argument `A` upiše *kopija reference na niz*. Dakle, kada kopiramo vrednost promenljive `V` u promenljivu `A` *kopira se referenca na niz, a ne sadržaj niza!*

```
static void Main() {
  double[] V = { 1.7, 1.9, 1.8, 1.6, 2.0, 1.8, 1.7, 1.8 };
  double p = Prosek(V, 0, V.Length);
  Console.WriteLine(p);
}

static double Prosek(double[] A, int p, int L) {
  double suma = 0.0;
  for(int i = 0; i < L; i++) {
    suma += A[p + i];
  }
  return suma / L;
}
```



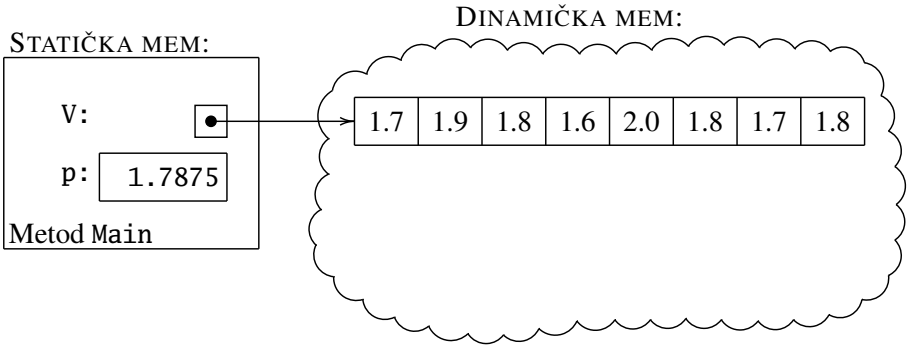
Nakon izvršenja "for"-ciklusa stanje u memoriji je prikazano na slici ispod. Treba obratiti pažnju na to da i metod `Main` i metod `Prosek` imaju promenljivu `p` ali svako vidi samo svoju promenljivu `p`.

```
static void Main() {
  double[] V = { 1.7, 1.9, 1.8, 1.6, 2.0, 1.8, 1.7, 1.8 };
  double p = Prosek(V, 0, V.Length);
  Console.WriteLine(p);
}

static double Prosek(double[] A, int p, int L) {
  double suma = 0.0;
  for(int i = 0; i < L; i++) {
    suma += A[p + i];
  }
  return suma / L;
}
```



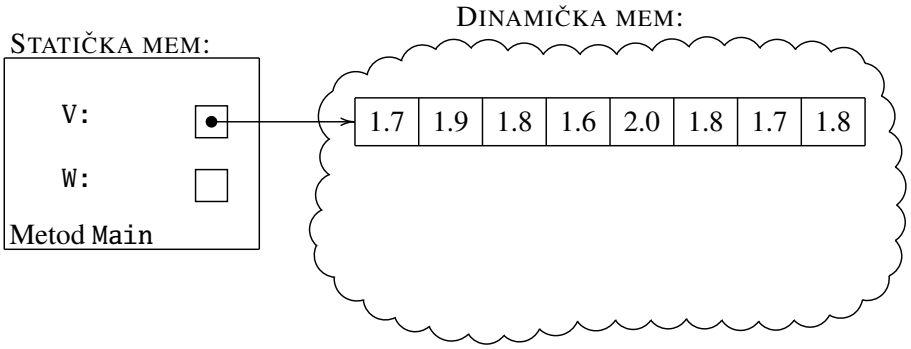
Naredbom `return` se glavnom programu vraća rezultat rada metoda. Metod `Prosek` završava sa radom, njegove promenljive se uklanjaju iz memorije i metod `Main` nastavlja sa radom da ispiše vrednost promenljive `p`.

```
static void Main() {
  double[] V = { 1.7, 1.9, 1.8, 1.6, 2.0, 1.8, 1.7, 1.8 };
  double p = Prosek(V, 0, V.Length);
  Console.WriteLine(p);
}

static double Prosek(double[] A, int p, int L) {
  double suma = 0.0;
  for(int i = 0; i < L; i++) {
    suma += A[p + i];
  }
  return suma / L;
}
```



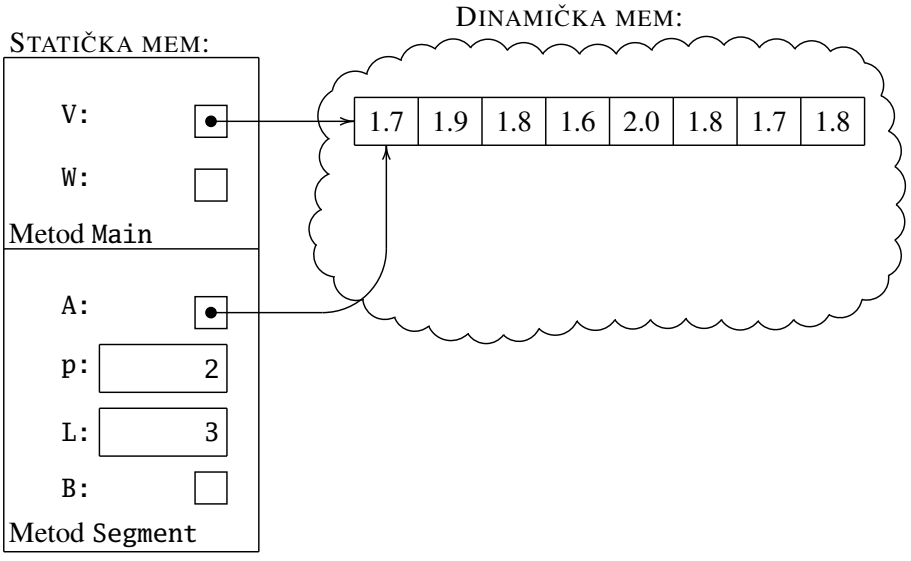
Metod Segment. Pogledajmo na jednom primeru kako radi metod `Segment`. Program krene od izvršavanja metoda `Main` u kome su deklarisana dva niza realnih vrednosti, `V` i `W`. (Promenljiva `i` je deklarisana unutar "for"-ciklusa i za nju će u memoriji biti alociran prostor tek kada krene da se izvršava ciklus.)

```
static void Main() {
  double[] V = { 1.7, 1.9, 1.8, 1.6, 2.0, 1.8, 1.7, 1.8 };
  double[] W = Segment(V, 2, 3);
  for(int i = 0; i < 3; i++) {
    Console.WriteLine(W[i]);
  }
}

static double[] Segment(double[] A, int p, int L) {
  double[] B = new double[L];
  for(int i = 0; i < L; i++) {
    B[i] = A[p + i];
  }
  return B;
}
```



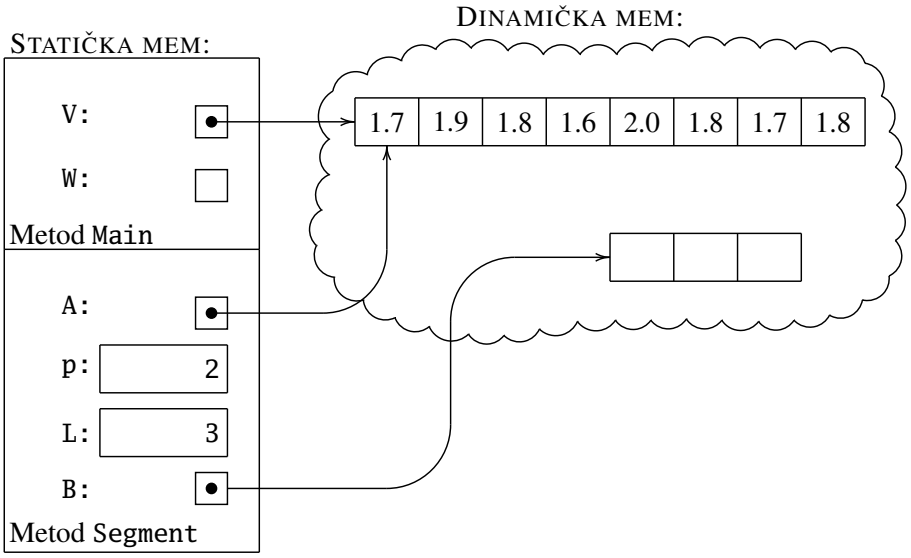
Poziv metoda `Segment` otvori nove fioke u memoriji za promenljive `A`, `p`, `L` i `B` i iskopira vrednosti argumenata. (Promenljiva `i` je deklarisana unutar "for"-ciklusa i za nju će u memoriji biti alociran prostor tek kada krene da se izvršava ciklus.) Argument `p` metoda dobija vrednost 2, argument `L` vrednost 3, dok se u argument `A` upiše kopija reference na niz.

```
static void Main() {
  double[] V = { 1.7, 1.9, 1.8, 1.6, 2.0, 1.8, 1.7, 1.8 };
  double[] W = Segment(V, 2, 3);
  for(int i = 0; i < 3; i++) {
    Console.WriteLine(W[i]);
  }
}

static double[] Segment(double[] A, int p, int L) {
  double[] B = new double[L];
  for(int i = 0; i < L; i++) {
    B[i] = A[p + i];
  }
  return B;
}
```



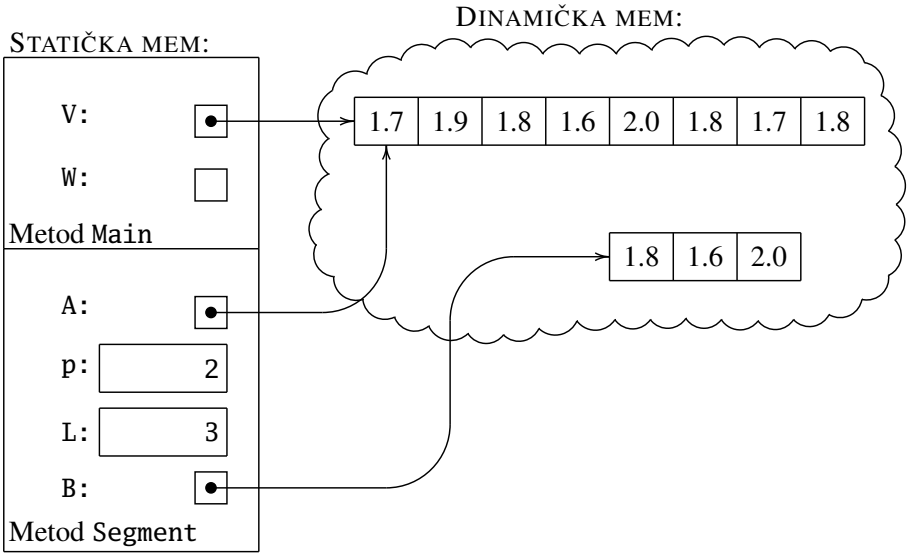
Sledeća naredba alocira niz dužine L i u B upiše referencu na njega.

```
static void Main() {
  double[] V = { 1.7, 1.9, 1.8, 1.6, 2.0, 1.8, 1.7, 1.8 };
  double[] W = Segment(V, 2, 3);
  for(int i = 0; i < 3; i++) {
    Console.WriteLine(W[i]);
  }
}

static double[] Segment(double[] A, int p, int L) {
  double[] B = new double[L];
  for(int i = 0; i < L; i++) {
    B[i] = A[p + i];
  }
  return B;
}
```



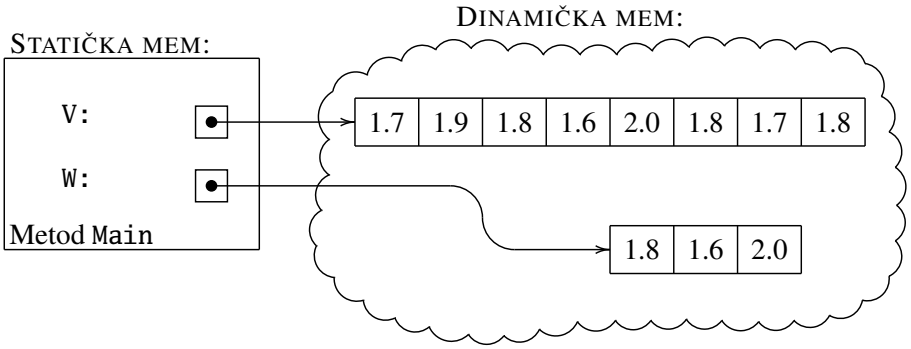
"For"-ciklus koji sledi u niz B prepiše deo niza A koji počinje od indeksa p i ima dužinu L. (Voditi računa o tome da prvi element niza ima indeks 0.)

```
static void Main() {
  double[] V = { 1.7, 1.9, 1.8, 1.6, 2.0, 1.8, 1.7, 1.8 };
  double[] W = Segment(V, 2, 3);
  for(int i = 0; i < 3; i++) {
    Console.WriteLine(W[i]);
  }
}

static double[] Segment(double[] A, int p, int L) {
  double[] B = new double[L];
  for(int i = 0; i < L; i++) {
    B[i] = A[p + i];
  }
  return B;
}
```



Naredbom `return` se glavnom programu vraća rezultat rada metoda. Metod `Segment` završava sa radom, njegove promenljive se uklanjaju iz memorije i metod `Main` nastavlja sa radom da ispiše elemente niza `W`.

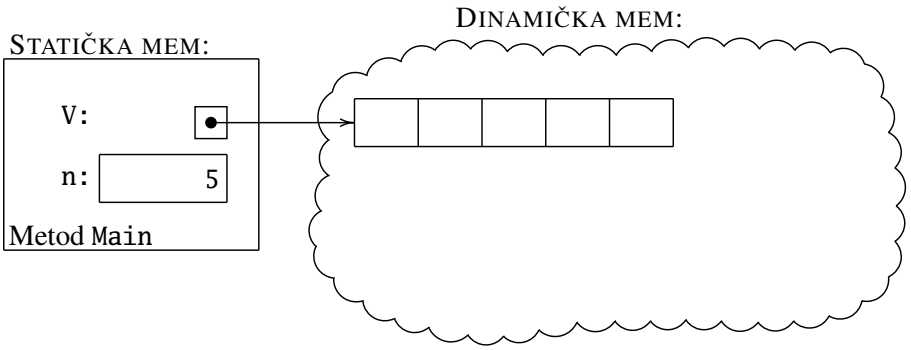
```
static void Main() {
  double[] V = { 1.7, 1.9, 1.8, 1.6, 2.0, 1.8, 1.7, 1.8 };
  double[] W = Segment(V, 2, 3);
  for(int i = 0; i < 3; i++) {
    Console.WriteLine(W[i]);
  }
}

static double[] Segment(double[] A, int p, int L) {
  double[] B = new double[L];
  for(int i = 0; i < L; i++) {
    B[i] = A[p + i];
  }
  return B;
}
```



Metod Napuni. Sada ćemo pokazati još jednu posledicu činjenice da se u statičkoj memoriji čuva samo referenca na niz, dok se niz nalazi u dinamičkoj memoriji: time se omogućeno da elemente niza izmenimo u nekom metodu i da izmena ostane vidljiva i nakon završetka rada metoda.

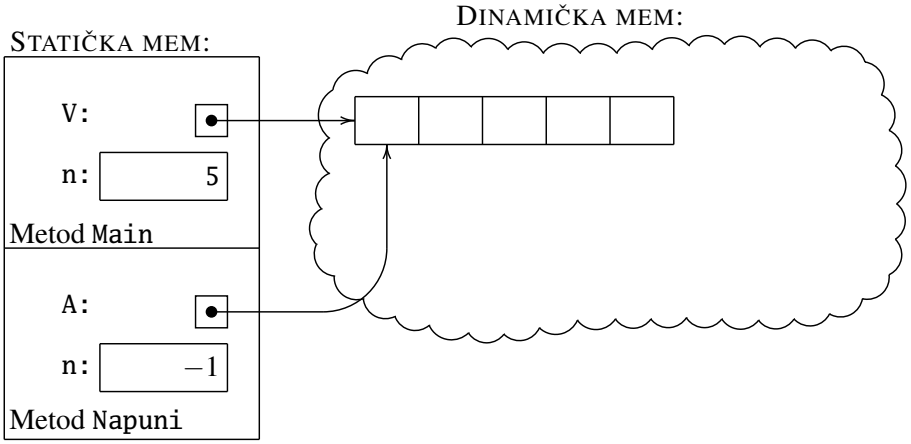
Pogledajmo na jednom primeru kako radi metod `Napuni`. Program krene od izvršavanja metoda `Main` u kome su deklarisani celobrojna promenljiva `n` i niz realnih vrednosti `V`. (Promenljiva `i` je deklarisana unutar "for"-ciklusa i za nju će u memoriji biti alociran prostor tek kada krene da se izvršava ciklus.) Program učita vrednost promenljive `n` i u dinamičkoj memoriji alocira odgovarajući broj kućica. Recimo da je korisnik uneo 5 kao vrednost promenljive `n`. Tada je stanje memorije prikazano na slici ispod.

```
static void Main() {
  int n = int.Parse(Console.ReadLine());
  int[] V = new int[n];
  Napuni(V, -1);
  for(int i = 0; i < n; i++) {
    Console.WriteLine(V[i]);
  }
}

static void Napuni(int[] A, int n) {
  for(int i = 0; i < A.Length; i++) {
    A[i] = n;
  }
}
```



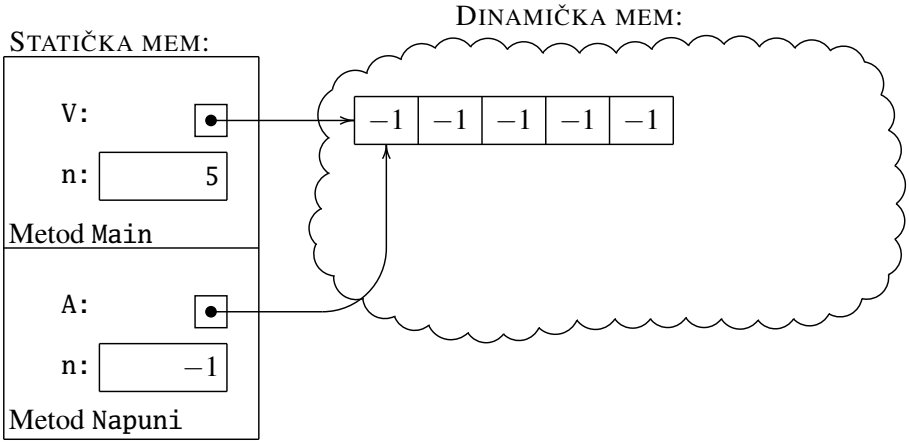
Poziv metoda `Napuni` otvori nove fioke u memoriji za promenljive `A` i `n` i iskopira vrednosti argumenata. (Promenljiva `i` je deklarisana unutar "for"-ciklusa i za nju će u memoriji biti alociran prostor tek kada krene da se izvršava ciklus.) Argument `n` metoda `Napuni` dobija vrednost -1, dok se u argument `A` upiše kopija reference na niz.

```
static void Main() {
  int n = int.Parse(Console.ReadLine());
  int[] V = new int[n];
  Napuni(V, -1);
  for(int i = 0; i < n; i++) {
    Console.WriteLine(V[i]);
  }
}

static void Napuni(int[] A, int n) {
  for(int i = 0; i < A.Length; i++) {
    A[i] = n;
  }
}
```



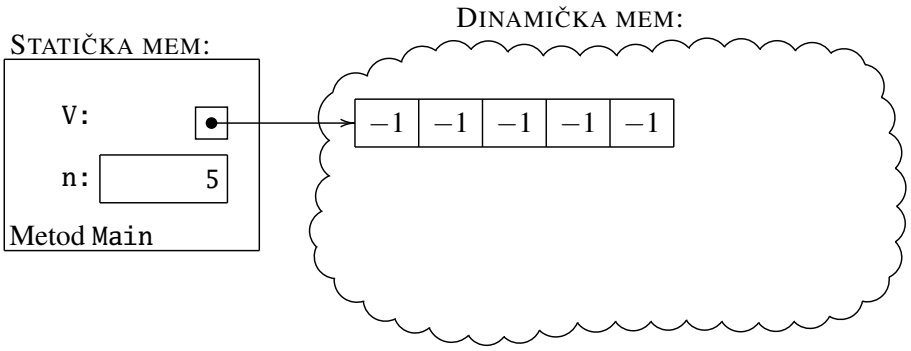
"For"-ciklus koji sledi elemente niza A postavi na vrednost promenljive n metoda Napuni.

```
static void Main() {
  int n = int.Parse(Console.ReadLine());
  int[] V = new int[n];
  Napuni(V, -1);
  for(int i = 0; i < n; i++) {
    Console.WriteLine(V[i]);
  }
}

static void Napuni(int[] A, int n) {
  for(int i = 0; i < A.Length; i++) {
    A[i] = n;
  }
}
```



Kada se metod Napuni završi njegove promenljive se uklanjaju iz memorije, kontrola se vraća glavnom programu i metod Main nastavlja sa radom da ispiše elemente niza V.

```
static void Main() {
  int n = int.Parse(Console.ReadLine());
  int[] V = new int[n];
  Napuni(V, -1);
  for(int i = 0; i < n; i++) {
    Console.WriteLine(V[i]);
  }
}

static void Napuni(int[] A, int n) {
  for(int i = 0; i < A.Length; i++) {
    A[i] = n;
  }
}
```



6.5 Varijante metoda

U prethodnom odeljku smo videli metod Prosek koji računa prosek segmenta niza, i to onog koji počinje od elementa niza sa indeksom p i ima dužinu L:

```
static double Prosek(double[] A, int p, int L) {
  double suma = 0.0;
  for(int i = 0; i < L; i++) {
    suma += A[p + i];
  }
  return suma / L;
}
```

Ako želimo da izračunamo prosek celog niza, moramo pozvati metod `Prosek` recimo ovako:

```
Prosek(X, 0, X.Length);
```

Druga mogućnost je da napišemo novi metod. Takođe, ako želimo da imamo metod koji računa prosek tri broja, treba da napišemo novi metod. Svi ovi metodi rade isto (računaju prosek nekih brojeva), a razlikuju se samo po tome kako im se prosleđuju brojevi čiji prosek želimo da izračunamo. U nekim (starijim) jezicima svi ovi metodi bi morali da imaju različita imena (recimo, `ProsekSegmenta`, `ProsekNiza`, `Prosek3`, itd). Međutim, u modernim programskim jezicima kao što je C# *kompajler razlikuje metode ne samo po imenu nego i po strukturi argumenata metoda*. Struktura argumenata metoda se zove *signatura* (*potpis*) metoda. Tako, sledeći programski fragment je potpuno korektan:

```
static double Prosek(double[] A, int p, int L) {
  double suma = 0.0;
  for(int i = 0; i < L; i++) {
    suma += A[p + i];
  }
  return suma / L;
}

static double Prosek(double[] A) {
  double suma = 0.0;
  for(int i = 0; i < A.Length; i++) {
    suma += A[i];
  }
  return suma / A.Length;
}

static double Prosek(double x, double y, double z) {
  return (x + y + z) / 3.0;
}

static double Prosek(double x, double y) {
  return (x + y) / 2.0;
}
```

Ovo su četiri varijante metoda `Prosek` i kompajler tačno zna kad koju da pozove. Na primer, uz deklaracije `double a, b, c; int n; double[] X;` imamo sledeću situaciju:

Poziv	Signatura	Biće pozvana...
`Prosek(a, b, c);`	`double, double, double`	treća varijanta
`Prosek(X);`	`double[]`	druga varijanta
`Prosek(a, c);`	`double, double`	četvrta varijanta
`Prosek(X, 5, n);`	`double[], int, int`	prva varijanta

Ovo svojstvo programskih jezika se na engleskom jeziku zove *overloading*.

Zadaci.

6.40. Napisati C# metod

```
static double MinNiza(double[] x)
```

koji određuje najmanji element niza `x`. Zna se da niz `x` ima bar jedan element i to ne treba proveravati.

6.41. Napisati C# metod

```
static double DrugiPoVelicini(double[] x)
```

koji određuje drugi najveći element niza `x`. Zna se da niz `x` ima bar dva elementa i to ne treba proveravati.

6.42. Ako je $A(x_1,x_2)$ tačka u ravni, dužina vektora $\overrightarrow{OA}$ se računa po formuli $|\overrightarrow{OA}| = \sqrt{x_1^2+x_2^2}$. Ako je $A(x_1,x_2,x_3)$ tačka u prostoru, dužina vektora $\overrightarrow{OA}$ se računa po formuli $|\overrightarrow{OA}| = \sqrt{x_1^2+x_2^2+x_3^2}$. Napisati C# metode

```
static double Duzina(double x, double y)
static double Duzina(double x, double y, double z)
```

koji računaju dužinu vektora.

6.43. *Skalarni proizvod* nizova $(a_0,a_1,\ldots,a_{n-1})$ i $(b_0,b_1,\ldots,b_{n-1})$ je broj koji se računa ovako:

$$a_0b_0 + a_1b_1 + \ldots + a_{n-1}b_{n-1}.$$

Napisati C# metode

```
static double SkalPr(double a0, double a1,
                     double b0, double b1)
static double SkalPr(double a0, double a1, double a2,
                     double b0, double b1, double b2)
static double SkalPr(double[] a, double[] b)
```

koji računaju skalarni proizvod datih nizova. Pretpostavlja se da su nizovi iste dužine i to ne treba proveravati.

6.44. Ako su A i B tačke u ravni ili prostoru, kosinus ugla koga zaklapaju vektori $\overrightarrow{OA}$ i $\overrightarrow{OB}$ se računa tako što se skalarni proizvod vektora $\overrightarrow{OA}$ i $\overrightarrow{OB}$ podeli proizvodom njihovih dužina. Napisati C# metode

```
static double CosUgla(double a0, double a1,
                      double b0, double b1)
static double CosUgla(double a0, double a1, double a2,
                      double b0, double b1, double b2)
```

koji računaju kosinus ugla koga zaklapaju vektori $\overrightarrow{OA}$ i $\overrightarrow{OB}$ gde su koordinate tačaka A i B date odgovarajućim nizovima.

6.45. Knjižara na zalihama ima više naslova. Napisati C# metod

```
static double CenaZaliha(double[] c, double[] k)
```

koji računa ukupnu vrednost robe u magacinu, gde je `c[i]` cena i-tog naslova u dinarima, a `k[i]` broj primeraka i-tog naslova u magacinu. (Napomena: Uporediti sa Zadatkom 6.43.)

6.46. *Konvolucija* nizova $(a_0, a_1, \ldots, a_{n-1})$ i $(b_0, b_1, \ldots, b_{n-1})$ je broj koji se računa ovako:

$$a_0 b_{n-1} + a_1 b_{n-2} + \ldots + a_{n-1} b_0.$$

Napisati C# metod

```
static double Konvolucija(double[] a, double[] b)
```

koji računa konvoluciju datih nizova. Pretpostavlja se da su nizovi iste dužine i to ne treba proveravati u metodu.

6.47. Napisati C# metod

```
static int[] IzbaciMin(int[] a)
```

koji formira i vraća niz celih brojeva koji se dobija od niza `a` izbacivanjem najmanjeg elementa tog niza i svih njemu jednakih elemenata. Metod napisati pod pretpostavkom da nisu svi elementi niza `a` međusobno jednaki.

6.48. Neka je dat niz brojeva a_1, a_2, …, a_n. Niz brojeva definisan sa

$$s_k = a_1 + \ldots + a_k$$

zove se *niz parcijalnih suma* polaznog niza. Napisati C# metod

```
static int[] ParcijalneSume(int[] a)
```

koji za dati niz `a` određuje niz parcijalnih suma i to vraća kao rezultat svoj rada.

6.49. Sve varijacije ⟨LINK⟩

6.50. Svi podskupovi ⟨LINK⟩

6.51. Sledeći podskup ⟨LINK⟩

6.52. Svi binarni nizovi bez susednih jedinica ⟨LINK⟩

6.53. Brojevi koji u binarnom zapisu nemaju dve susedne nule ⟨LINK⟩

6.54. Sve kombinacije ⟨LINK⟩

6.55. Sve permutacije ⟨LINK⟩

6.56. Sledeća particija ⟨LINK⟩

6.57. Sve particije ⟨LINK⟩

6.58. Sve jednocifrene particije ⟨LINK⟩

6.59. Svi n-tocifreni brojevi sa datim zbirom cifara ⟨LINK⟩

Glava 7

Simboli i nizovi simbola

Iako su nastali kao mašine za obradu numeričkih podataka, računari su doživeli vrhunac svog uspeha kada se shvatilo da oni mogu da obrađuju i *simboličke* podatke. Obrada simboličkih podataka danas predstavlja dominantan način upotrebe računara. Prvo ćemo predstaviti najjednostavniji simbolički tip podataka – tip `char` koji omogućuje rad sa pojedinačnim simbolima. Nakon toga ćemo upoznati `switch` naredbu koja se koristi kada treba proveriti mnogo jednostavnih uslova oblika ⟨*izraz*⟩=⟨*konstantna vrednost*⟩ jedan za drugim. U tom slučaju je `switch` naredba znatno efikasnija od višestruke `if` naredbe. Zatim prelazimo na prezentaciju različitih načina za obradu nizova simbola. Prvo ćemo se vratiti stringovima (= niskama) koje poznajemo još od samog početka teksta i detaljno ćemo prodiskutovati njihove najvažnije osobine, kao i operacije koje se nad njima mogu primeniti. Opisaćemo osnovne bibliotečke funkcije za rad sa stringovima, a na kraju odeljka ćemo pokazati nekoliko problema za čije rešavanje je potrebno koristiti nizove stringova. Nakon toga ćemo pokazati kako se u programskom jeziku C# može upravljati tekstualnim datotekama, što su tekstovi pohranjeni na disku računara čija veličina je ograničena samo veličinom diska. Rad sa tekstualnim datotekama predstavlja veoma važnu veštinu jer se dominantni protokoli za prenos informacija u današnjem svetu zasnivaju na tekstualnim datotekama. Na primer, HTML fajl je obična tekstualna datoteka, struktuirana na odgovarajući način. Isto važi za XML, JSON i mnoge druge standarde kodiranja informacija.

7.1 Tip char

Tip podataka `char` opisuje promenljive u koje može da se smesti tačno jedno slovo. Ime potiče od engleske reči *character* koja znači *karakter* (skup mentalnih i moralnih osobina neke osobe), ali i *simbol* (npr. *Chinese characters*, što znači

kineski simboli, a ne *karakteri Kineza*).

Na primer, kao vrednost promenljive tipa `char` može da se pojavi bilo koji od simbola `'a' 'b' 'c'` ... `'A' 'B' 'C'` ... `'0' '1' '2'` ... `'#' '$' '%' '+' '-' '*' '(' ']' '{' '!' '?' '_' ';' ':' ','` i slično. Primetimo da odgovarajući simbol mora biti "ograđen" apostrofima.

Konstante i promenljive tipa `char` se deklarišu na uobičajeni način, kako se vidi u primeru pored. Prilikom dodele vrednosti promenljivoj takođe nema iznenađenja, tako da fragment pored ispisuje `3$` zato što promenljiva `d` nakon dodele `d = c` ima vrednost `'3'`, a promenljiva `money` ima vrednost `'$'`.

```
const char DollarSign = '$';
char c, d, money;

c = '3';
d = '@';
money = DollarSign;
d = c;
Console.Write("{0}{1}", d, money);
```

Apostrofi oko simbola koji predstavljaju vrednosti tipa `char` se pišu samo u programskom kôdu, a prilikom ispisa se ne pojavljuju.

☞ *Prilikom ispisa se apostrofi ne pojavljuju. Ispisuju se samo simboli!*

Smisao apostrofa oko simbola je u tome da naprave razliku između imena promenljive i vrednosti tipa `char` kako to pokazuje programski fragment pored. Tu se promenljivoj `d` dodeljuje *simbol* `c`, a promenljivoj `e` vrednost *promenljive* `c`.

```
char  c, d, e;
c = '@';
d = 'c';
e = c;
Console.Write("{0}{1}", d, e)
```

U tabeli pored prikazan je rad ovog fragmenta naredbu po naredbu. Na početku vrednosti promenljivih nisu određene. Posle naredbi `c = '@'` i `d = 'c'` promenljiva `c` ima vrednost `'@'`, a promenljiva `d` vrednost `'c'`. Naredba `e = c` kopira vrednost *promenljive* `c`, tako da posle ove naredbe promenljiva `e` ima vrednost `'@'`. Dakle, fragment na kraju ispisuje `c@` (bez apostrofa!).

Naredba	Promenljiva		
	`c`	`d`	`e`
	?	?	?
`c = '@';`	`'@'`	?	?
`d = 'c';`	`'@'`	`'c'`	?
`e = c;`	`'@'`	`'c'`	`'@'`

Ako želimo da predstavimo apostrof kao simbol, to možemo da uradimo ovako:

```
'\''
```

Prilikom zapisivanja simbola (i stringova koje ćemo detaljno upoznati kasnije) simbol \ se koristi kao *escape character*, što znači da se on koristi kao upozorenje kompajleru da treba prikazati simbol koji je naveden iza njega. Na taj način se mogu prikazati simboli kao što je ' (apostrof) i \ (obrnuta kosa crta), ali i neki *neprintabilni simboli*, tj. simboli koji nemaju grafičku reprezentaciju, ali ih koristimo za kontrolu teksta, kao što su *newline* (instrrukcija sistemu da prilikom ispisa pređe u novi red) i *tab* (tabulator – prelazak u sledeću unapred definisanu kolonu).

C# zapis	Značenje	Akcija
`'a'`	simbol `a`	prikaže se simbol `a`
`'\''`	simbol '	prikaže se simbol '
`'\"'`	simbol "	prikaže se simbol "
`'\\'`	simbol \	prikaže se simbol \
`'\n'`	*newline*	prilikom ispisa pređe se u novi red
`'\t'`	*tab*	prilikom ispisa pređe se u sledeću unapred definisanu kolonu

☞ *Naravno, moramo voditi računa da se promenljivoj tipa* `char` *ne može dodeliti ni broj ni logička vrednost! Ukoliko pokušate da promenljivoj tipa* `char` *dodelite nešto što nije slovo ili slovna promenljiva prevodilac će prijaviti grešku. Slično, ukoliko pokušamo numeričkoj ili logičkoj promenljivoj da dodelimo vrednost koja je tipa* `char` *prevodilac će prijaviti grešku.*

Svakom simbolu je dodeljen neki broj, njegov *kôd*. Računar u suštini ne pamti simbole kao simbole, već pamti i radi sa odgovarajućim brojevima – kôdovima simbola. Kodiranje simbola je prvi put standardizovano još sedamdesetih godina ovog veka sistemom propisa koji se zove ASCII (*American Standard Code for Information Interchange*) koji je još uvek u upotrebi. Kako je ASCII kôd *too American, even for the British*,[1] 1990-ih je uveden novi sistem kodiranja simbola koji se zove UNICODE (od engl. *universal code*) koji podržava većinu svetskih jezika i sve simbole koji su za njih karakteristični. Da bi se od simbola dobio odgovarajući kôd, i da bi se od celog broja dobio simbol čiji je to kôd u C# se koristi *type cast*:

```
(int)'A'  →  65
(char)65  →  A
```

[1] na primer, ASCII kôd predviđa kôd za simbol $, ali *ne* predviđa kôd za simbol £

Na skupu svih simbola je uveden poredak ovako:

$$c_1 < c_2 \text{ ako i samo ako je } \texttt{(int)}c_1 < \texttt{(int)}c_2.$$

Tako je `'a' < 'b' < 'z'`, `'A' < 'Z' < 'a'` i `'0' < '1' < '9'`. Ako je `c` promenljiva tipa `char`, sledeći simbol u UNICODE tabeli možemo da dobijemo ovako:

```
c++;
```

a prethodni ovako:

```
c--;
```

Primer. Napisati C# program koji od korisnika učitava karaktere `c1` i `c2` i štampa sve karaktere između njih.

⟨C# fajl⟩

```
using System;

class SvaSlovaIzmedju {
  static void Main() {
    char c1, c2, c;
    Console.Write("c1 -> ");
    c1 = (char)Console.Read(); Console.ReadLine();
    Console.Write("c2 -> ");
    c2 = (char)Console.Read(); Console.ReadLine();
    for(c = c1; c <= c2; c++) {
      Console.WriteLine(c);
    }
  }
}
```

U ovom primeru vidimo kako se učitava jedan karakter (komplikovano). Naredba `Read` klase `Console` učitava jedan simbol sa tastature i vraća *kôd* tog simbola, dakle podatak tipa `int`. Zato je potrebno dobijeni `int` konvertovati u `char` koristeći *type cast*. Nakon unosa karaktera u ulaznom baferu su ostali kontrolni karakteri koji označavaju kraj jednog unosa (oni se generišu kada korisnik pritisne taster `Enter`) i zato nakon učitavanja karaktera pozivamo naredbu `ReadLine` da počisti smeće koje je ostalo u ulaznom baferu.

Klasa `char` ima metode `ToLower` i `ToUpper` koji konvertuju slovo u odgovarajuće malo, odnosno, veliko slovo. Pri tome se karakteri koji nisu slova ne menjaju. Jedan veoma jednostavan (i ne mnogo smislen) primer je dat ispod. Ovaj program

dva puta ispisuje True:

⟨C# fajl⟩
```
using System;

class VelikoMaloSlovo {
  static void Main() {
    char c, d;
    c = 'A';
    Console.WriteLine(char.ToLower(c) == 'a');
    d = '!';
    Console.WriteLine(char.ToUpper(d) == '!');
  }
}
```

7.2 "Switch" kontrolna struktura

Pogledajmo program koji od korisnika učitava dva cela broja koji predstavljaju mesec i godinu, i potom određuje i štampa broj dana u tom mesecu:

⟨C# fajl⟩
```
using System;

class BrojDanaUMesecu {
  static void Main() {
      Console.WriteLine("Mesec: ");
      int m = int.Parse(Console.ReadLine());
      Console.WriteLine("Godina: ");
      int g = int.Parse(Console.ReadLine());
      // ispitamo da li je g prestupna godina
      int p = 0;
      if (g % 400 == 0 || g % 100 != 0 && g % 4 == 0) { p = 1; }
      // ispisujemo broj dana u mesecu m
      if     (m ==  1) { Console.WriteLine(31); }
      else if(m ==  2) { Console.WriteLine(28 + p); }
      else if(m ==  3) { Console.WriteLine(31); }
      else if(m ==  4) { Console.WriteLine(30); }
      else if(m ==  5) { Console.WriteLine(31); }
      else if(m ==  6) { Console.WriteLine(30); }
      else if(m ==  7) { Console.WriteLine(31); }
      else if(m ==  8) { Console.WriteLine(31); }
      else if(m ==  9) { Console.WriteLine(30); }
      else if(m == 10) { Console.WriteLine(31); }
      else if(m == 11) { Console.WriteLine(30); }
      else if(m == 12) { Console.WriteLine(31); }
      else { Console.WriteLine("Pogresan mesec"); }
```

```
  }
}
```

Kada treba proveriti mnogo jednostavnih uslova oblika

```
if(⟨izraz1⟩=⟨konstantna vrednost1⟩) { ... }
else if(⟨izraz2⟩=⟨konstantna vrednost2⟩) { ... }
  ⋮
else if(⟨izrazk⟩=⟨konstantna vrednostk⟩) { ... }
```

jedan za drugim kao što je ovde slučaj, višestruki "if" može biti neefikasno i nepregledno rešenje. Zato C# nudi jedan drugi vid grananja: "switch" kontrolnu strukturu.

"Switch" kontrolna struktura ima oblik koji je naveden pored. Ona radi tako što računar prvo izračuna vrednost izraza ⟨*izraz*⟩ i traži "case" koji predstavlja tu vrednost. Ukoliko nađe odgovarajući "case", izvrše se naredbe koje su navedene. Ukoliko nijedan "case" ne navodi vrednost koja se traži, izvršava se spisak naredbi iza oznake `default:`.

```
switch(⟨izraz⟩) {
  case ⟨vrednost1⟩:
    ⟨naredbe1⟩
    break;
  case ⟨vrednost2⟩:
    ⟨naredbe2⟩
    break;
    ⋮
  case ⟨vrednostk⟩:
    ⟨naredbek⟩
    break;
  default:
    ⟨naredbe0⟩
    break;
}
```

Koristeći "switch" kontrolnu strukturu program koji ispisuje broj dana u mesecu može da se napiše ovako:

⟨C# fajl⟩

```
using System;

class BrojDanaUMesecu {
  static void Main() {
      Console.WriteLine("Mesec: ");
      int m = int.Parse(Console.ReadLine());
      Console.WriteLine("Godina: ");
      int g = int.Parse(Console.ReadLine());
      // ispitamo da li je g prestupna godina
      int p = 0;
      if (g % 400 == 0 || g % 100 != 0 && g % 4 == 0) { p = 1; }
```

```
      // ispisujemo broj dana u mesecu m
      switch(m) {
        case 1:
          Console.WriteLine(31);
          break;
        case 2:
          Console.WriteLine(28 + p);
          break;
        case 3:
          Console.WriteLine(31);
          break;
        case 4:
          Console.WriteLine(30);
          break;
        case 5:
          Console.WriteLine(31);
          break;
        case 6:
          Console.WriteLine(30);
          break;
        case 7:
          Console.WriteLine(31);
          break;
        case 8:
          Console.WriteLine(31);
          break;
        case 9:
          Console.WriteLine(30);
          break;
        case 10:
          Console.WriteLine(31);
          break;
        case 11:
          Console.WriteLine(30);
          break;
        case 12:
          Console.WriteLine(31);
          break;
        default:
          Console.WriteLine("Pogresan mesec");
          break;
    }
  }
}
```

ili još kraće, ovako:

⟨C# fajl⟩

```
using System;

class BrojDanaUMesecu {
  static void Main() {
      Console.WriteLine("Mesec: ");
      int m = int.Parse(Console.ReadLine());
      Console.WriteLine("Godina: ");
      int g = int.Parse(Console.ReadLine());
      // ispitamo da li je g prestupna godina
      int p = 0;
      if (g % 400 == 0 || g % 100 != 0 && g % 4 == 0) { p = 1; }
      // ispisujemo broj dana u mesecu m
      switch(m) {
        case 1: case 3: case 5: case 7: case 8: case 10: case 12:
          Console.WriteLine(31);
          break;
        case 4: case 6: case 9: case 11:
          Console.WriteLine(30);
          break;
        case 2:
          Console.WriteLine(28 + p);
          break;
        default:
          Console.WriteLine("Pogresan mesec");
          break;
    }
  }
}
```

Poslednji primer je važan: on pokazuje da je moguće nanizati nekoliko “case” deklaracija pre istog koda, i to je jedno od svojstava koje čini “switch” kontrolnu strukturu korisnom!

Primer. Napisati C# program koji učitava redni broj meseca (broj između 1 i 12) i štampa godišnje doba kojem pripada taj mesec. Jednostavnosti radi, uzimamo da su decembar, januar i februar zimski meseci, mart, april i maj prolećni, jun, jul i avgust letnji, a septembar, oktobar i novembar jesenji meseci.

⟨C# fajl⟩

```
using System;

class GodisnjaDoba {
  static void Main() {
```

```
    Console.Write("Mesec: ");
    int m = int.Parse(Console.ReadLine());
    switch(m) {
      case 12: case 1: case 2:
        Console.WriteLine("Zima");
        break;
      case 3: case 4: case 5:
        Console.WriteLine("Prolece");
        break;
      case 6: case 7: case 8:
        Console.WriteLine("Leto");
        break;
      case 9: case 10: case 11:
        Console.WriteLine("Jesen");
        break;
      default:
        Console.WriteLine("Greska!");
        break;
    }
  }
}
```

Primer. *Encyclopaedia Hysterica* ima 6 tomova: prvi tom obuhvata slova od A do F i slovo X, drugi slovo G, treći slova od H do N i slovo Y, četvrti slova od O do S, peti slova T i U, a šesti slova V, W i Z (pri čemu se iz nekih nikada razjašnjenih razloga koristi engleska abeceda). Napisati program koji od korisnika učitava slovo i ispisuje redni broj toma koji sadrži to slovo.

⟨C# fajl⟩

```
using System;

class EncyclopaediaHysterica {
  static void Main() {
    char c;
    Console.Write("Slovo -> ");
    c = (char)Console.Read();
    Console.ReadLine();


    switch(char.ToUpper(c)) {
      case 'A': case 'B': case 'C': case 'D':
      case 'E': case 'F': case 'X':
        Console.WriteLine("Tomus I");
        break;
      case 'G':
```

```
        Console.WriteLine("Tomus II");
        break;
      case 'H': case 'I': case 'J': case 'K':
      case 'L': case 'M': case 'N': case 'Y':
        Console.WriteLine("Tomus III");
        break;
      case 'O': case 'P': case 'Q': case 'R': case 'S':
        Console.WriteLine("Tomus IV");
        break;
      case 'T': case 'U':
        Console.WriteLine("Tomus V");
        break;
      case 'V': case 'W': case 'Z':
        Console.WriteLine("Tomus VI");
        break;
      default:
        Console.WriteLine("Errare humanum est...");
        break;
    }
  }
}
```

Zadaci.

7.1. Sa deklaracijom `char c;` na snazi označiti korektne naredbe:

☐ `c = 'a';`	☐ `c = ''';`	☐ `c = true;`
☐ `c = 'c';`	☐ `c = '\'';`	☐ `c++;`
☐ `c = "a";`	☐ `c = 65;`	☐ `c--;`
☐ `c = '';`	☐ `c = (int)65;`	☐ `c = 12.1;`
☐ `c = '"';`	☐ `c = (char)65;`	

7.2. Napisati C# program koji od korisnika učitava brojeve p i q takve da je $32 \leqslant p \leqslant q \leqslant 126$ i štampa simbole čiji kôd pripada intervalu $[p,q]$, kao i odgovarajuće kodove. Na primer, za brojeve 65 i 68 izlaz je dat pored.

```
65  A
66  B
67  C
68  D
```

7.3. Šahovsko polje ⟨LINK⟩

7.4. Šta radi sledeći fragment?

```
int n = int.Parse(Console.ReadLine());
switch(n % 3) {
  case 0:
    Console.WriteLine("jeste");
    break;
  default:
    Console.WriteLine("nije");
    break;
}
```

7.5. Sledeći programski fragment napisati koristeći "switch" naredbu:

```
if(k == 0) {
  r++;
  Console.WriteLine(r);
}
else if(k == 1) {
  s++;
  Console.WriteLine(s);
}
else if(k == 2 || k == 3 || k == 4) {
  t++;
  Console.WriteLine(t);
}
```

7.6. Utvrditi vrednosti promenljivih `p` i `d` nakon izvršenja sledećeg programskog fragmenta ako je korisnik za vrednost promenljive `k` uneo

(a) 6; (b) 235; (c) 71; (d) 100.

```
int k = int.Parse(Console.ReadLine());
int p = 1; int d = 1;
switch(k % 10) {
  case 2: case 3: case 5: case 7: d = k; break;
  case 1: break;
  case 4: case 8: p = 0; d = 2; break;
  case 9: case 6: p = 0; d = 3; break;
}
```

7.7. Napisati program koji učitava tri cela broja koji predstavljaju neki datum, a potom ispisuje taj datum tako da mesec bude ispisan rečima. Na primer za 3. 11. 2000. program treba da ispiše `3. novembar 2000`.

7.8. Napisati program koji učitava tri cela broja koji predstavljaju neki datum, a potom ispisuje taj datum prema engleskom standardu, ali tako da mesec bude ispisan rečima. Na primer za 3. 11. 2000. program treba da ispiše `November 3rd, 2000`.

7.9. Napisati program koji konvertuje dati ceo broj iz skupa $\{1, 2, \ldots, 3999\}$ u rimski zapis. Na primer:

```
Unesite broj : 1937
Rimski zapis broja: MCMXXXVII
```

7.3 Tip string

String je reč književnog engleskog jezika čije osnovno značenje je "objekti iste vrste koji su nanizani na nit". U programskom jeziku C# *string je niz karaktera.* Zbog toga se stringovi u literaturi na srpskom jeziku često zovu i *niske*. Na primer, ovo su stringovi (niske):

```
"Zdravo! Kako si?"
"Osnovna skola \"Jovan Jovanovic-Zmaj\""
":-)"
""
```

Vidimo da se stringovi pišu tako što se između navodnika navede proizvoljan niz karaktera. Poslednji primer nam pokazuje kako izgleda *prazan string*, tj. string koji nema nijedno slovo, dok nam drugi primer pokazuje kako da zapišemo string u kome se javljaju znaci navoda: moramo da koristimo *escape* sekvencu `\"` zato što se navodnici koriste kao graničnici za sadržaj stringa.

Broj karaktera u stringu se zove *dužina stringa*. Dužine stringova iz prethodnog primera su, redom, 16, 36, 3 i 0.

Promenljiva tipa string se može deklarisati ovako:

```
string s;
```

Informaciju o dužini stringa `s` daje nam ugrađena funkcija `Length` koja vraća rezultat tipa `int`. Dužina stringa `s` se dobija ovako:

```
s.Length
```

Programski fragment pored učitava string od korisnika i ispisuje njegovu dužinu. Na primer, ako kao vrednost promenljive `s` korisnik unese `Programiranje`, program će ispisati 13.

```
string s;
s = Console.ReadLine();
Console.WriteLine(s.Length);
```

Stringovi se učitavaju naredbom `Console.ReadLine` koja čita niz karaktera do kraja reda, a ispisuju se naredbama `Console.Write` i `Console.WriteLine`. Stringu se može dodeliti neka vrednost naredbom dodele. Sve ovo je pokazano u programskom fragmentu pored.

```
string ime, pozdrav;
Console.Write("Tvoje ime ->")
ime = Console.ReadLine();
pozdrav = "Zdravo, ";
Console.Write(pozdrav);
Console.WriteLine(ime);
```

☞ *Pošto je string zapravo niz karaktera, moguće je pristupiti jednom elementu stringa (što je onda vrednost tipa* `char`*), ali* elemente stringa nije moguće menjati!

U programskom jeziku C# kao i u većini modernih jezika *stringovi su nepromenjlivi (strings are immutable!)*. **Zato programski fragment pored dovodi do greške prilikom kompajliranja!**

```
string s;
s = Console.ReadLine();
if(s.Length > 3) {
  s[3] = 'x'; // GRESKA
}
Console.WriteLine(s);
```

Primer. Napisati C# program koji od korisnika učitava neki string, i onda ga štampa u obrnutom redosledu. Na primer, ako korisnik unese

```
Napisati C# program
```

program će ispisati

```
margorp #C itasipaN
```

```
using System;

class Obrnuto {
  static void Main() {
    Console.Write("Tekst   -> ");
    string s = Console.ReadLine();
    Console.Write("Obrnuto -> ");
    for(int i = s.Length - 1; i >= 0; i--) {
      Console.Write(s[i]);
    }
    Console.WriteLine();
  }
}
```

⟨C# fajl⟩

Primer. Napisati C# program koji utvrđuje koliko ima razmaka u učitanom stringu.

⟨C# fajl⟩

```
using System;

class BrojRazmaka {
  static void Main() {
    Console.Write("Tekst -> ");
    string s = Console.ReadLine();
    int br = 0;
    foreach(char c in s) {
      if(c == ' ') { br++; }
    }
    Console.WriteLine(br);
  }
}
```

Primer. Napisati C# program koji utvrđuje da li je u učitanom stringu broj otvorenih zagrada `'('` jednak broju zatvorenih zagrada `')'`.

⟨C# fajl⟩

```
using System;

class Zagrade {
  static void Main() {
    Console.Write("Tekst -> ");
    string s = Console.ReadLine();
    int br = 0;
    foreach(char c in s) {
      if(c == '(') { br++; }
      else if(c == ')') { br--; }
    }
    if(br == 0) {
      Console.WriteLine("Isti je broj ( i )");
    }
    else {
      Console.WriteLine("Nije isti broj ( i )");
    }
  }
}
```

Primer. Kažemo da su zagrade `(` i `)` *balansirane* u nekom stringu ako i samo ako imamo isti broj `(` i `)` zagrada, i u svakom početnom segmentu stringa imamo da je broj `(` zagrada $\geqslant$ od broja `)` zagrada. Na primer:

`a((bc)()+)n`	jesu
`[)`	nisu
`(())())`	nisu
`ab)(`	nisu

Napisati C# program koji proverava da li su zagrade `'('` i `')'` u unetom stringu balansirane.

⟨C# fajl⟩

```
using System;

class BalansiraneZagrade {
  static void Main() {
    Console.Write("Tekst -> ");
    string s = Console.ReadLine();
    int br = 0;
    foreach(char c in s) {
      if(c == '(') { br++; }
      else if(c == ')') { br--; }
      if(br < 0) { break; }
    }
    if(br == 0) {
      Console.WriteLine("Zagrade su balansirane");
    }
    else {
      Console.WriteLine("Zagrade nisu balansirane");
    }
  }
}
```

Ideja ovog rešenja je veoma jednostavna: kao i prethodnom primeru idemo redom kroz string i brojimo zagrade, s tim što sada *ne smemo da dozvolimo da brojač* `br` *postane negativan!* Čim `br` postane negativan, znači da smo detektovali početni segment stringa u kome ima više `)` zagrada nego `(` zagrada, i tada treba prekinuti sa radom ciklusa i prijaviti da zagrade u stringu nisu balansirane.

Stringovi se mogu porediti po jednakosti upotrebom operatora `==` (i `!=` kada želimo da utvrdimo da nisu jednaki). S obzirom na to da su stringovi specijalni nizovi, i operator poređenja se ponaša drugačije nego kod ostalih nizova:

☞ *Kada se stringovi porede po jednakosti ne utvrđuje se jednakost referenci, već se proverava* ***da li imaju isti sadržaj!***

⟨C# fajl⟩

```
using System;

class JednakiStringovi {
  static void Main() {
    Console.Write("Ime -> ");
    string ime = Console.ReadLine();
    if(ime == "Pera" || ime == "pera" || ime == "PERA") {
      Console.WriteLine("Gde si, Pero, druze stari!");
    }
    else {
      Console.WriteLine("Ja tebe ne poznajem, a deca ne");
      Console.WriteLine("treba da se druze sa nepoznatima!");
    }
  }
}
```

Primer. Napisati program koji od korisnika učitava jedan red teksta i utvrđuje koliko se puta u tom redu javlja svako od slova $a, \ldots, z$, nezavisno od toga da li se u redu pojavljuje kao veliko ili kao malo slovo. Na primer, u redu *Ana voli Milovana* se slovo *a* javlja 4 puta, a slovo *m* samo jednom.

⟨C# fajl⟩

```
using System;

class SlovaURedu {
  static void Main() {
    Console.Write("Tekst -> ");
    string s = Console.ReadLine();
    int[] br = new int[26];
    for(int i = 0; i < 26; i++) { br[i] = 0; }
    int a = (int)'a';
    foreach(char c in s) {
      char d = char.ToLower(c);
      if('a' <= d && d <= 'z') {
        br[(int)d - a]++;
      }
    }
    for(int i = 0; i < 26; i++) {
      if(br[i] > 0) {
        Console.WriteLine("Slovo {0} -> {1} puta",
                          (char)(i + a), br[i]);
      }
    }
  }
}
```

U ovom primeru promenljiva `a` sadrži kôd karaktera `'a'` koji će nam trebati za preračun indeksa. Kako se broj pojavljivanja slova `'a'` pamti u kućici `br[0]`, broj pojavljivanja slova `'b'` pamti u kućici `br[1]`, i tako dalje, lako se vidi da se broj pojavljivanja slova *d* pamti u kućici `br[(int)d - a]`. Slično, u poslednjem "for" ciklusu indeks se kreće od 0 do 25, a indeksu `i` odgovara slovo `(char)(i + a)`.

7.4 Operacije sa stringovima

Tip `string` poseduje sledeće operatore koji su "ugrađeni" u jezik, gde su `s` i `t` promenljive tipa `string`, a `c` promenljiva tipa `char`:

Operacija	Značenje (rezultat je uvek tipa string)
`s + t`	nadovezivanje (konkatenacija)
`s + c`	dodavanje karaktera `c` na kraj stringa `s`
`c + s`	dodavanje karaktera `c` na početak stringa `s`

Na primer, neka su `s` i `t` stringovi čija vrednost je `s = "bar"` i `t = "bara"`, a `c` promenljiva tipa `char` čija vrednost je `c = 'i'`. Tada:

Izraz	Vrednost	Tip
`s + t`	`'barbara'`	`string`
`t + s`	`'barabar'`	`string`
`s + c`	`'bari'`	`string`
`c + s`	`'ibar'`	`string`

Naravno, umesto

```
s = s + c;
```

uvek možemo da pišemo

```
s += c;
```

Na ovaj način možemo da gradimo stringove tako što pođemo od praznog stringa i na njega nadovezujemo karaktere ili druge stringove. Kako su stringovi u C# nepromenljivi, svaka izmena stringa se svodi na to da napravimo novi string u koga ćemo prepisati delove starog stringa koji nam odgovaraju, a delove koji nam ne odgovaraju da zamenimo. Ovo je prihvatljivo ako radimo sa kratkim stringovima, ali može osetno da uspori rad programa ako radimo sa dugačkim stringovima. Zato pored tipa `string` u programskom jeziku C# postoje i drugi načini organizacije karaktera u nizove, kao što je `StringBuilder`, koji nam omogućuju da efikasno modifikujemo njihov sadržaj. Ovaj, efikasniji način rada sa stringovima detaljnije diskutujemo u Odeljku 7.6.

Primer. Napisati C# program koji učitava string `s` i onda formira i ispisuje novi string u kome su sva mala slova stringa `s` konvertovana u velika. Na primer, string `"Govori tiho!"` će biti konvertovan u string `"GOVORI TIHO!"`

⟨C# fajl⟩

```
using System;

class SveVelikaSlova {
  static void Main() {
    Console.Write("Tekst -> ");
    string s = Console.ReadLine();
    string t = "";
    foreach(char c in s) { t += char.ToUpper(c); }
    Console.WriteLine("TEKST -> {0}", t);
  }
}
```

Primer. Napisati C# program koji učitava string `s`, iz njega izbacuje sva pojavljivanja karaktera `' '` (razmak) i potom ispisuje tako skraćeni string. Na primer, ako je je program učitao string `"Idi  mi      dodji mi"` ispisaće `"Idimidodjimi"`.

⟨C# fajl⟩

```
using System;

class BezPraznina {
  static void Main() {
    Console.Write("Tekst -> ");
    string s = Console.ReadLine();
    string t = "";
    foreach(char c in s) {
      if(c != ' ') { t += c; }
    }
    Console.WriteLine("Bez praznina -> {0}", t);
  }
}
```

Primer. Napisati C# metod `string Obrni(string s)` koji od stringa `s` formira novi tako što obrne njegova slova.

⟨C# fajl⟩

```
using System;

class ObrniString {
  static void Main() {
    Console.Write("Unesi string: ");
    string s = Console.ReadLine();
    Console.WriteLine(Obrni(s));
```

```
  }

  static string Obrni(string s) {
    string t = "";
    foreach(char c in s) { t = c + t; }
    return t;
  }
}
```

Veoma je bitno voditi računa o tome da *nadovezivanje stringova nije komutativna operacija!* U ovom primeru smo dodavali `c` na početak stringa `t` da bismo na kraju dobili obrnuti poredak.

☞ *U ovom primeru* ***se ne sme*** *naredba* `t = c + t;` *zameniti sa* `t += c;` *jer ova poslednja dodaje* `c` ***na kraj***.

Primer. Napisati C# metod `bool Palindrom(string s)` koji proverava da li dati niz slova predstavlja *palindrom* nakon izbacivanja razmaka, znakova interpunkcije i zanemarivanja velikih i malih slova. Palindrom je reč koja se isto čita i sleva i zdesna. Na primer, za sledeće rečenice program treba da prijavi da su palindromi:

```
Sir ima miris.
Sava zidar radi za vas!
Ana voli Milovana!
Maja sa Nedom ode na sajam.
```

⟨C# fajl⟩

```
using System;

class DaLiJePalindrom {
  static void Main() {
    Console.Write("Unesi string: ");
    string s = Console.ReadLine();
    Console.WriteLine(Palindrom(s));
  }

  static bool Palindrom(string s) {
    int i = 0;
    int j = s.Length - 1;
    while(i < j) {
      if(NijeSlovo(s[i])) { i++; continue; }
      if(NijeSlovo(s[j])) { j--; continue; }
```

```
        if(NijeIstoSlovo(s[i], s[j])) { return false; }
        i++; j--;
      }
      return true;
    }

    static bool NijeSlovo(char c) {
      return !char.IsLetter(c);
    }

    static bool NijeIstoSlovo(char a, char b) {
      return char.ToLower(a) != char.ToLower(b);
    }
  }
```

7.5 Bibliotečke funkcije za rad sa stringovima

Stringovi su veoma važni u programiranju i zato postoji veliki broj funkcija koje su namenjene raznim transformacijama stringova. Ovde ćemo prikazati samo nekoliko.

Za poređenje stringova u programskom jeziku C# koristi se metod `CompareTo` koji poredi stringove u odnosu na *leksikografski poredak*. To je poredak koji se može videti u rečnicima (odatle ime), a definisan je na sledeći način. Neka su `s` i `t` različiti stringovi.

- Ako je string `s` prefiks stringa `t` onda je `s` manje od `t`.
- Ako string `s` nije prefiks stringa `t`, onda krenemo sleva na desno i nađemo prvo mesto na kome se stringovi razlikuju. Neka je to pozicija `i`. Sada, ako je `s[i]` $<$ `t[i]` onda je `s` manje od `t`. U suprotnom je `s` veće od `t`.

Na primer,

`"bar"` < `"bara"` zato što je string `"bar"` prefiks stringa `"bara"`
`"bara"` < `"bas"` zato što se stringovi razlikuju na 3. mestu i `'r'` < `'s'`
`"Avala"` < `"ala"` zato što se stringovi razlikuju na 1. mestu i `'A'` < `'a'`

Stringovi `s` i `t` se porede metodom `CompareTo` ovako:

```
s.CompareTo(t)
```

i ovaj metod vraća -1 ako je `s` manje od `t`, vraća 0 ako je `s == t`, i vraća 1 ako je `s` veće od `t` u leksikografskom poretku.

Operacija	Značenje
`s == t`	stringovi su jednaki
`s != t`	stringovi su različiti
`s.CompareTo(t) == -1`	`s` je ispred `t` u leksikografskom poretku
`s.CompareTo(t) == 0`	`s == t`
`s.CompareTo(t) == 1`	`s` je iza `t` u leksikografskom poretku
`s.CompareTo(t) <= 0`	`s` je jednako sa `t` ili ispred `t` u leks. poretku
`s.CompareTo(t) >= 0`	`s` je jednako sa `t` ili je iza `t` u leks. poretku

Sve velika i sve mala slova. Funkcija `ToUpper` konvertuje sva mala slova stringa u velika slova, dok funkcija `ToLower` konvertuje sva velika slova stringa u mala slova.

Na primer, ako u programskom fragmentu pred korisnik unese string `Pera` program će ispisati `peraPERA`.

```
string s, t, u;
s = Console.ReadLine();
t = s.ToLower();
u = s.ToUpper();
Console.WriteLine(t + u);
```

IndexOf i Substring. Metod `IndexOf` se poziva ovako:

```
s.IndexOf(x)
```

traži prvo pojavljivanje karaktera ili stringa `x` stringu `s`. Ako se karakter ili string `x` ne pojavljuje u stringu `s` metod vraća -1. Na primer, ako je `s = "barbara"` i `t = "rhubarb"` onda je

```
s.IndexOf("bara") == 3
t.IndexOf("bara") == -1
s.IndexOf('h') == -1
t.IndexOf('h') == 1
```

Metod `Substring` se poziva ovako:

```
s.Substring(n, L)
```

i vraća podstring (što je niz uzastopnih karaktera) stringa `s` koji počinje od indeksa `n` i ima dužinu `L`. Na primer, ako je `s = "barbara"` onda je

```
s.Substring(3, 4) == "bara"
```

Primer. Napisati program koji konvertuje temperaturu iskazanu u Farenhajtima, t_F, u Celzijuse, t_C, i obrnuto, imajući u vidu da je:

$$t_F = 1.8 \cdot t_C + 32.$$

Korisnik unosi niz simbola koji predstavlja decimalni zapis broja odmah iza koga se nalazi slovo `C` ili `F`. Ako se niz simbola završava sa `C` znači da je temperatura iskazana u Celzijusima i da je treba konvertovati u Farenhajte, a ako se niz simbola završava sa `F` znači da je temperatura iskazana u Farenhajtima i da je treba konvertovati u Celzijuse. Na primer:

```
Temperatura -> 92F      ili     Temperatura -> 12C
92F = 33.33333333C              12C = 53.6F
```

⟨C# fajl⟩

```
using System;

class CelFar {
  static void Main() {
    Console.Write("Temperatura -> ");
    string s = Console.ReadLine();
    int L = s.Length;
    if(L == 0) {
      Console.WriteLine("Greska");
    }
    else if(char.ToUpper(s[L - 1]) == 'C') {
      double tC = double.Parse(s.Substring(0, L - 1));
      double tF = 1.8 * tC + 32.0;
      Console.WriteLine("{0}C = {1}F", tC, tF);
    }
    else if(char.ToUpper(s[L - 1]) == 'F') {
      double tF = double.Parse(s.Substring(0, L - 1));
      double tC = (tF - 32.0) / 1.8;
      Console.WriteLine("{0}F = {1}C", tF, tC);
    }
    else {
      Console.WriteLine("Nepoznata jedinica");
    }
  }
}
```

U ovom primeru vidimo kako se metod `double.Parse` koristi da se string za koga očekujemo da sadrži decimalni zapis realnog broja konvertuje u odgovarajuću vrednost.

Primer. Napisati C# program koji od korisnika učitava algebarski izraz oblika

$$\langle broj \rangle + \langle broj \rangle$$

i računa njegovu vrednost. Ceo izraz je niz karaktera bez razmaka koji se nalazi u jednom redu. Brojevi u izrazu su realni brojevi.

⟨C# fajl⟩

```
using System;

class RacunajZbir {
  static void Main() {
    Console.Write("Izraz -> ");
    string s = Console.ReadLine();
    int L = s.Length;
    int n = s.IndexOf('+');
    if(L == 0) {
      Console.WriteLine("Greska: prazan string");
    }
    else if(n == 0) {
      Console.WriteLine("Greska: nema zanaka +");
    }
    else {
      double a = double.Parse(s.Substring(0, n));
      double b = double.Parse(s.Substring(n + 1, L - 1 - n));
      Console.WriteLine("{0}+{1}={2}", a, b, a + b);
    }
  }
}
```

7.6 Stringovi i nizovi karaktera

Kao što smo već rekli, stringovi u C# su nepromenljivi pa se svaka izmena stringa svodi na to da napravimo novi string u koga ćemo prepisati delove starog stringa koji nam odgovaraju. Ovo neprestano pravljenje novih i novih stringova od starih je prihvatljivo ako radimo sa kratkim stringovima, ali može osetno da uspori rad programa ako radimo sa dugačkim stringovima. Zato se pored tipa `string` u programskom jeziku C# korsite i drugi načini organizacije karaktera u nizove koji nam omogućuju da efikasno modifikujemo njihov sadržaj. U ovom odeljku ćemo razmotriti kako da efikasno upravljamo sadržajem stringa koristeći nizove karaktera, odnosno, strukturu podataka `StringBuilder`.

Posmatrajmo sledeće dve deklaracije:

```
string s;
char[] t;
```

U obe promenljive možemo da smestimo niz karaktera. Razlika je u tome što elemente stringa u programskom jeziku C# nije moguće menjati, dok se elementi dodeljeni nizu karaktera kao što je `t` mogu slobodno menjati.

Primer. Napisati program za igranje igre pogađanja reči. Prvi igrač unese reč koju drugi igrač treba da pogodi. Računar prikaže na ekranu niz od onoliko zvezdica koliko slova sadrži reč. Drugi igrač onda pogađa slova. Ako uneta reč ne sadrži slovo, računar uveća brojač negativnih poena za jedan. Ako uneta reč sadrži slovo, računar prikaže reč u kojoj su sva pogođena slova otkrivena. Tok igre bi mogao da izgleda, recimo, ovako:

```
1. Pokusaj -> **********     Negativnih poena: 0
-> a
2. Pokusaj -> *****a*a***     Negativnih poena: 0
-> e
3. Pokusaj -> *****a*a**e     Negativnih poena: 0
-> z
4. Pokusaj -> *****a*a**e     Negativnih poena: 1
-> k
5. Pokusaj -> k*k**aka**e     Negativnih poena: 1
-> m
6. Pokusaj -> k*k**aka**e     Negativnih poena: 2
-> o
7. Pokusaj -> koko*aka**e     Negativnih poena: 2
-> d
8. Pokusaj -> kokodaka**e     Negativnih poena: 2
-> n
9. Pokusaj -> kokodakan*e     Negativnih poena: 2
-> j
BRAVO! Zamisljena rec je -> kokodakanje
```

⟨C# fajl⟩

```
using System;

class IgraPogadjanjaReci {
  static void Main() {
    Console.Write("Unesi rec -> ");
    string rec = Console.ReadLine();
    Console.Clear(); // brise ekran

    int L = rec.Length;
    char[] skrivenaRec = new char[L];
    for(int i = 0; i < L; i++) { skrivenaRec[i] = '*'; }
    int brZvezdica = L;
    int negPoena = 0;
    int pokusaj = 0;

    while(brZvezdica > 0) {
      pokusaj++;
      Console.Write("{0}. Pokusaj -> ", pokusaj);
      foreach(char c in skrivenaRec) { Console.Write(c); }
      Console.WriteLine("    Negativnih poena: {0}", negPoena);

      Console.Write("-> ");
      char slovo = (char)Console.Read();
      Console.ReadLine();

      if(rec.IndexOf(slovo) == -1) { negPoena++; }
      else {
        for(int i = 0; i < L; i++) {
          if(rec[i] == slovo) {
            skrivenaRec[i] = slovo;
            brZvezdica--;
          }
        }
      }
    }
    Console.WriteLine("BRAVO! Zamisljena rec je -> {0}", rec);
  }
}
```

Programski jezik C# obezbeđuje mehanizme konverzije između tipova `string` i `char[]`. Neka su i dalje `s` i `t` promenljive deklarisane na sledeći način:

```
string s;
char[] t;
```

Sadržaj promenljive tipa `string` možemo da konvertujemo u `char[]` pozivom metoda `ToCharArray()` ovako:

```
t = s.ToCharArray();
```

dok sadržaj promenljive tipa `char[]` možemo da konvertujemo u `string` ovako:

```
s = new string(t);
```

Primer (ponovo). Napisati C# program koji učitava string `s` i onda formira i ispisuje novi string u kome su sva mala slova stringa `s` konvertovana u velika. Na primer, string `"Govori tiho!"` će biti konvertovan u string `"GOVORI TIHO!"`

⟨C# fajl⟩

```
using System;

class SveVelikaSlova2 {
  static void Main() {
    Console.Write("Tekst -> ");
    string s = Console.ReadLine();
    char[] t = s.ToCharArray();
    for(int i = 0; i < t.Length; i++) { t[i] = char.ToUpper(t[i]); }
    s = new string(t);
    Console.WriteLine("TEKST -> {0}", s);
  }
}
```

Primer (ponovo). Napisati C# metod `string Obrni(string s)` koji od stringa `s` formira novi tako što obrne njegova slova.

⟨C# fajl⟩

```
using System;

class ObrniString2 {
  static void Main() {
    Console.Write("Unesi string: ");
    string s = Console.ReadLine();
    Console.WriteLine(Obrni(s));
  }
```

```
  static string Obrni(string s) {
    char[] t = s.ToCharArray();
    int n = t.Length;
    for(int i = 0; i < n/2; i++) {
      char c = t[i];
      t[i] = t[n - 1 - i];
      t[n - 1 - i] = c;
    }
    return new string(t);
  }
}
```

Primer. Napisati C# metod

```
static string Ispremestaj(string s, int[] a, int[] b)
```

koji od stringa `s` pravi novi string tako što ispremešta njegova slova na sledeći način:

- slovo na poziciji `a[0]` zameni mesto sa slovom na poziciji `b[0]`;
- u tako dobijenom stringu slovo na poziciji `a[1]` zameni mesto sa slovom na poziciji `b[1]`;
- u tako dobijenom stringu slovo na poziciji `a[2]` zameni mesto sa slovom na poziciji `b[2]`;

i tako dalje. Pretpostavlja se (i to ne treba proveravati) da su nizovi `a` i `b` iste dužine i da su elementi oba niza celi brojevi između 0 i `s.Length - 1`.

⟨C# fajl⟩

```
using System;

class IspremestajSlovaStringa {
  static void Main() {
    string s = "ANA VOLI MILOVANA";
    int[] a = new int[] {0,  5, 11, 13, 6, 10, 3};
    int[] b = new int[] {1, 16,  7,  4, 8,  2, 9};
    Console.WriteLine(Ispremestaj(s, a, b));
  }

  static string Ispremestaj(string s, int[] a, int[] b) {
    char[] x = s.ToCharArray();
    for(int i = 0; i < a.Length; i++) {
      char pom = x[a[i]];
      x[a[i]] = x[b[i]];
```

```
      x[b[i]] = pom;
    }
    string t = new string(x);
    return t;
  }
}
```

Kao što smo videli, kada radimo sa promenljivim tipa `char[]` lako je menjati karaktere koji učestvuju u stringu. Međutim, kada radimo sa promenljivim tipa `char[]` nije lako dodavati nove karaktere na kraj niza, ili između dva elementa niza. Zato je u programski jezik C# uvedena struktura podataka `StringBuilder` koja, pored mogućnosti da menjamo pojedinačne karaktere u nizu, implementira i efikasne operacije za nadovezivanje i umetanje stringova.

Klasa `StringBuilder` se nalazi u biblioteci `System.Text` tako da svaki C# program koji želi da koristi ovu klasu mora prvo da uveze biblioteku `System.Text`. Programski jezik C# obezbeđuje mehanizme konverzije između tipova `string` i `StringBuilder`. Neka su `s` i `t` promenljive deklarisane na sledeći način:

```
string s;
StringBuilder t;
```

Sadržaj promenljive tipa `string` možemo da konvertujemo u `StringBuilder` ovako:

```
t = new StringBuilder(s);
```

dok sadržaj promenljive tipa `StringBuilder` možemo da konvertujemo nazad u `string` ovako:

```
s = t.ToString();
```

Ponekad je zgodno poći od prazne `StringBuilder` strukture kako bismo tokom rada algoritma na nju dodavali karaktere ili stringove. Prazan `StringBuilder` se dobija ovako:

```
t = new StringBuilder();
```

Struktura tipa `StringBuilder` predstavlja (unapređenu) vrstu nizova karaktera, pa se kao i kod nizova karaktera jednom elementu niza pristupa na uobičajeni način:

```
t[i]
```

Osim toga, i dužinu niza možemo dobiti na uobičajeni način:

```
t.Length
```

Metod `Append(s)` na kraj strukture tipa `StringBuilder` dodaje ceo string `s`, dok metod `Insert(i, s)` umeće string `s` od pozicije `i` (karakteri koji se već nalaze u strukturi se pomeraju udesno). Metod `Replace(stari, novi)` će svako pojavljivanje stringa `stari` zameniti stringom `novi`. Konačno, metod `Remove(i, n)` će ukloniti `n` karaktera počev od pozivije `i`.

Primer (ponovo). Napisati C# program koji učitava string `s`, iz njega izbacuje sva pojavljivanja karaktera `' '` (razmak) i potom ispisuje tako skraćeni string. Na primer, ako je je program učitao string `"Idi  mi      dodji mi"` ispisaće `"Idimidodjimi"`.

```
using System;
using System.Text;

class BezPraznina2 {
  static void Main() {
    Console.Write("Tekst -> ");
    string s = Console.ReadLine();
    StringBuilder t = new StringBuilder(s);
    t.Replace(" ", "");
    Console.WriteLine("Bez praznina -> {0}", t);
  }
}
```

⟨C# fajl⟩

Primer. Napisati C# program koji učitava string `s`, iz njega izbacuje sve suvišne praznine i potom ispisuje tako skraćeni string. Suvišne praznine su praznine na početku i na kraju stringa, kao i svaka praznina u stringu ispred koje se nalazi praznina. Na primer, ako je je program učitao string `"  Idi  mi      dodji mi  "` ispisaće `"Idi mi dodji mi"`.

```
using System;
using System.Text;

class BezNepotrebnihPraznina {
  static void Main() {
    Console.Write("Tekst -> ");
    string s = Console.ReadLine();
    StringBuilder t = new StringBuilder();
    // eliminacija vodecih praznina
    int i = 0;
    while(i < s.Length && s[i] == ' ') { i++; }
    // eliminacija visestrukih praznina u tekstu
    bool praznina = false;
```

⟨C# fajl⟩

```
    while(i < s.Length) {
      if(s[i] != ' ') {
        t.Append(s[i]);
        praznina = false;
      }
      else if(!praznina) {
        t.Append(s[i]);
        praznina = true;
      }
      i++;
    }
    // moze da se desi da je ostala tacno jedna praznina na kraju
    if(t.Length > 0 && t[t.Length - 1] == ' ') {
      t.Remove(t.Length - 1, 1);
    }
    Console.WriteLine("Bez nepotrebnih praznina -> {0}", t);
  }
}
```

Zadaci.

7.10. Napisati C# program koji od korisnika učitava string i potom formira i ispisuje novi string koji se dobija dodavanjem jednog razmaka iza svake tačke u stringu `s`.

7.11. Napisati C# program koji od korisnika učitava string i potom formira i ispisuje novi string koji se dobija dodavanjem jednog razmaka iza svakog znaka interpunkcije, ako je to potrebno. Na primer,

$$\texttt{"Halo?Da? Ne.Ko je to?"} \mapsto \texttt{"Halo? Da? Ne. Ko je to?"}$$

7.12. Napisati C# program koji od korisnika učitava string i potom ispisuje broj reči u tom stringu. (Napomena: reč je neprekidan niz slova.)

7.13. Napisati program koji ispisuje reči unetog teksta u obrnutom poretku. Na primer, ukoliko se unese prethodna rečenica, program treba da ispiše:

`poretku obrnutom u teksta unetog reci ispisuje koji program Napisati`

7.14. Napisati metod `string Shorten(string s, int w)` koji formatira string `s` na dužinu `w` na sledeći način: ako je dužina stringa `s` manja ili jednaka sa `w`, ne treba ništa menjati. Ukoliko je dužina stringa `s` strogo veća od `w`, onda treba iz sredine stringa izbaciti odgovarajući broj karaktera i zameniti taj deo stringa stringom `...` (tri tačke), tako da dužina dobijenog stringa bude tačno `w`. Na primer, poziv

```
Shorten("Ajaoj sto je skola zgodna", 10)
```

treba da vrati `"Aja...odna"`. Pretpostavlja se da je `w` > 5.

7.15. Napisati C# program koji od korisnika učitava string i potom ispisuje poslednju reč u tom stringu. Na primer, ako je korisnik uneo string `'Biti il' ne biti...'`, program ispisuje reč `biti`

7.16. Frekvencija znaka ⟨LINK⟩

7.17. Napisati C# program koji proverava da li su u unetom stringu balansirane sledeće zagrade: `'('` i `')'`, `'['` i `']'`, `'<'` i `'>'`.

7.18. Napisati C# program koji utvrđuje da li je dati *veliki* broj deljiv sa 30. Veliki broj je broj sa proizvoljno mnogo cifara koje se unose u jednoj liniji kao niz karaktera.

Primer 1:
```
Unesite broj: 458273048572304587230924
Nije deljiv sa 30
```

Primer 2:
```
Unesite broj: 303030300000000030000000030
Deljiv je sa 30
```

7.19. Isto kao u prethodnom zadatku, samo što se proverava deljivost sa 4.

7.20. Isto kao u prethodnom zadatku, samo što se proverava deljivost sa 11.

Kriterijum deljivosti sa 11. Neka je s_p suma cifara broja na parnim mestima, a s_n suma cifara tog broja na neparnim mestima. Broj je deljiv sa 11 ako i samo ako je $s_n - s_p$ deljivo sa 11. Na primer, za broj 190949 je $s_n = 5$, $s_p = 27$, $s_n - s_p = -22$, što je deljivo sa 11, pa je i taj broj deljiv sa 11 (zapravo, $190949 = 11 \cdot 17359$).

7.21. Isto kao u prethodnom zadatku, samo što se proverava deljivost sa 33.

7.22. Napisati C# program koji utvrđuje da li je zbir dva data *velika* broja deljiv sa 3. Veliki broj je broj sa proizvoljno mnogo cifara koje se unose u jednoj liniji kao niz karaktera.

7.23. Napisati program koji od korisnika učitava pozitivan ceo broj n, potom n redova teksta, i utvrđuje koliko se puta u tim redovima javlja svako od slova $a, \ldots, z$, nezavisno od toga da li se u redu pojavljuje kao veliko ili kao malo slovo. Na primer, u redu *Ana voli Milovana* se slovo a javlja 4 puta, a slovo m samo jednom.

7.24. U sledećoj tabeli data je atomska masa nekih hemijskih elemenata:

H	1.0079	O	15.9990	F	18.9984
He	4.0026	Na	22.9898	Au	196.9665
Be	9.0122	S	32.0600	Hg	200.5000
C	12.0110	Cl	35.4530	Ra	226.0254

Napisati program koji od korisnika učitava hemijsku formulu nekog molekula i oređuje i štampa njegovu molekularnu masu. Formula molekula se unosi kao što smo na to i navikli, s tim da se brojevi ne pišu kao indeksi. Na primer: `C2H5OH`, `NaCl`, `H2O`, `H2SO4`.

7.25. Napisati C# program koji konvertuje iznos iskazan u evrima, u dolare i obrnuto, imajući u vidu da u trenutku pisanja ovog teksta 1 dolar vredi 0.9 evra. Korisnik unosi niz simbola koji predstavlja zapis valute tako što prvo unese oznaku valute, "$" za dolare ili "E" za evre, a odmah iza toga i iznos. Na primer:

```
Unesite iznos:        ili     Unesite iznos:
$210                          E136
$210.00 = E189.00             E136.00 = $151.11
```

7.26. Napisati program koji iz dva uneta stringa izdvaja njihov najduži zajednički podstring.

7.27. Napisati program koji iz unetog stringa uklanja sve podstringove koji se nalaze između zagrada `(` i `)`, kao i zagrade. Zagrade mogu biti ugnežđene i zna se da, ako ih ima, onda su izbalansirane.

7.28. Za potrebe ovog zadatka, broj decimala nekog realnog broja koje su navedene u njegovom decimalnom zapisu zvaćemo *preciznost* tog broja. Tako, broj 12.3430 ima preciznost 4, broj -175.00 ima preciznost 2, dok broj 75 ima preciznost 0. *Preciznost niza brojeva* je najmanja preciznost broja u tom nizu. Na primer:

Niz	*Preciznost*
12.5660 -1.726656 9.300	3
13.9483 55 43.00 1496.3432	0

Napisati C# program koji od korisnika učitava string koji sadrži niz realnih brojeva gde su svaka dva razdvojena tačno jednim razmakom, i potom računa i štampa preciznost tog niza.

7.29. (MICIKA - MalI CelobrojnI KAlkulator) Napisati C# program koji od korisnika učitava algebarski izraz oblika

$$\langle broj \rangle \langle operacija \rangle \langle broj \rangle$$

i računa njegovu vrednost. Ceo izraz je niz karaktera bez razmaka koji se nalazi u jednom redu. Brojevi koji su navedeni su pozitivni celi brojevi koji mogu da stanu u `longint`, a ⟨*operacija*⟩ je jedan od ovih karaktera: `+` (sabiranje), `-` (oduzimanje), `*` (množenje), `/` (celobrojni količnik), `%` (ostatak pri celobrojnom deljenju).

Primer 1:
```
Izraz: 123*45
5535
```

Primer 2:
```
Izraz: 1247%13
12
```

Primer 3:
```
Izraz: 1247/13
95
```

7.30. Sve podreči ⟨LINK⟩

7.31. Sve podreči po opadajućoj dužini ⟨LINK⟩

7.32. Trougao od reči ⟨LINK⟩

7.33. Izbacivanje podniski ⟨LINK⟩

7.34. Podniska ⟨LINK⟩

7.35. Cezarov kod ⟨LINK⟩

7.36. Brojevne osnove ⟨LINK⟩

7.37. Spreadsheet kolone ⟨LINK⟩

7.38. Izomorfne niske ⟨LINK⟩

7.39. Najkraća podniska koja sadrži sve date karaktere ⟨LINK⟩

7.7 Nizovi stringova

Programski jezik C# podržava i nizove stringova, naravno. Deklaracija niza stringova, kao i deklaracija svakog drugog niza, izgleda ovako:

```
string[] s;
```

Primer. Napisati program koji učitava broj iz skupa $\{1, 2, \ldots, 9999\}$ i ispisuje taj broj rečima na engleskom jeziku. Na primer, za 328 program treba da ispiše:

```
three hundred twenty eight
```

⟨C# fajl⟩

```
using System;

class SayTheNumber {
  static void Main() {
    // niz pocinje praznim stringom zato sto zelimo da
    // rec "one" ima indeks 1, itd.
    string[] jedinice = { "", "one", "two", "three",
      "four", "five", "six", "seven", "eight", "nine", "ten",
      "eleven", "twelve", "thirteen", "fourteen", "fifteen",
      "sixteen", "seventeen", "eighteen", "nineteen" };

    // na pocetku niza su dva prazna stringa zato sto zelimo da
    // rec "twenty" ima indeks 2, itd.
    string[] desetice = { "", "",  "twenty", "thirty", "fourty",
      "fifty", "sixty", "seventy", "eighty", "ninety" };

    Console.Write("Broj od 1 do 9999 -> ");
    int n = int.Parse(Console.ReadLine());
    if (n <= 1 || n >= 9999) {
      Console.WriteLine("Broj izvan opsega");
    }
    else {
      int k = n / 1000;
      if(k > 0) { Console.Write(jedinice[k] + " thousand "); }
      k = (n / 100) % 10;
      if(k > 0) { Console.Write(jedinice[k] + " hunderd "); }
      k = n % 100;
      if(k >= 20) {
        Console.Write(desetice[k / 10] + " ");
        k = k % 10;
      }
      if(k > 0) { Console.Write(jedinice[k]); }
      Console.WriteLine();
    }
  }
}
```

Ako program treba da obradi puno podataka, onda nije isplativo navoditi podatke svaki u zasebnom redu. Tada se podaci u program često unose tako što se više podataka navede u jednom redu, pri čemu su podaci u istom redu razdvojeni prazninama, zarezima ili nekim drugim pogodnim *separatorom*, na primer ovako:

```
23 4 -1 0 5 9 6 11
```

ili ovako:

```
23, 4, -1, 0, 5, 9, 6, 11
```

Ukoliko string `s` sadrži niz brojeva razdvojenih, recimo, zarezom, on se lako može "razbiti" na pojedinačne elemente pozivom metoda `Split` ovako:

```
s.Split(',');
```

Argument metoda `Split` je karakter koji predstavlja separator (simbol koji se koristi da razdvoji vrednosti), a metod onda vraća niz stringova. Ukoliko se ne navede argument iza `Split`, recimo ovako:

```
s.Split();
```

onda se podrazumeva da su vrednosti u nizu razdvojene prazninama. Dakle u metodu `Split` podrazumevani separator je `' '` (praznina).

Primer. Napisati C# program koji od korisnika učitava nekoliko celih brojeva koji se svi nalaze u jednom redu i razdvojeni su zarezima, pa računa i štampa njihov maksimum.

⟨C# fajl⟩

```
using System;

class NizUJednomRedu {
  static void Main() {
    Console.Write("Niz brojeva razdvojenih zarezima -> ");
    string red = Console.ReadLine();
    string[] brojevi = red.Split(',');

    int max = int.Parse(brojevi[0]);
    foreach(string s in brojevi) {
      int k = int.Parse(s);
      if(k > max) { max = k; }
    }

    Console.WriteLine("Njihov max -> {0}", max);
  }
}
```

Treba biti posebno pažljiv ako je niz podataka razdvojen prazninama. Direktna primena prethodne ideje radi samo kada su podaci razdvojeni *tačno jednom prazninom i kada nema praznina ni na početku ni na kraju reda.* Svaka praznina viška će napraviti prazan string u nizu i sa tim se moramo izboriti na poseban način.

Primer. Napisati C# program koji od korisnika učitava nekoliko celih brojeva koji se svi nalaze u jednom redu i razdvojeni su razmacima, pa računa i štampa njihov maksimum.

⟨C# fajl⟩

```
using System;

class NizUJednomRedu2 {
  static void Main() {
    Console.Write("Niz brojeva razdvojenih praznинama -> ");
    string red = Console.ReadLine();
    string[] brojevi = red.Split();

    // trazimo prvi neparazan string da inicijalizujemo max;
    // moramo da postavimo max na neku vrednost jer se
    // u suprotnom kompajler nervira
    int max = 0;
    bool nasaoPrvi = false;
    foreach(string s in brojevi) {
      if(s != "") {
        int k = int.Parse(s);
        if(!nasaoPrvi) {
          // ovo je prvi neprazan element niza
          max = k;
          nasaoPrvi = true;
        }
        else if(k > max) { max = k; }
      }
    }

    if(nasaoPrvi) {
      Console.WriteLine("Njihov max -> {0}", max);
    }
    else {
      Console.WriteLine("Ceo niz je prazan!");
    }
  }
}
```

Priču o nizovima stringova ćemo završiti komentarom o argumentima glavnog metoda `Main`. Glavni metod `Main` može da ima argumente, a ako su navedeni onda to može biti samo jedan niz stringova. Dakle, metod `Main` se deklariše ili ovako:

```
static void Main() {
  ...
}
```

ili ovako:

```
static void Main(string[] args) {
  ...
}
```

Ako metod `Main` ima argument, niz stringova koji mu se prosleđuju može da se popuni vrednostima samo ako se odgovarajući program pozove kao komanda iz komandne linije, i tada se popunjava vrednostima koje su navedene iza imena programa. Recimo, u sledećem primeru je navedena "komanda" koja samo ispiše šta je navedeno iza poziva.

⟨C# fajl⟩

```
using System;

class CommandLineArgs {
  static void Main(string[] args) {
    Console.WriteLine("Komanda ima {0} argumenata", args.Length);
    if(args.Length > 0) {
      Console.WriteLine("Argumenti su:");
      foreach(string s in args) {
        Console.WriteLine(s);
      }
    }
  }
}
```

Iako se klasa zove `ComandLineArgs`, izvršni program se zove `komanda.exe` zato što se fajl u kome se nalazi klasa zove `komanda.cs`. Ako pozovemo ovu komandu iz komandne linije dobićemo sledeće:

```
C:\Users\DM\Primeri>komanda
Komanda ima 0 argumenata


C:\Users\DM\Primeri>komanda -h -t=1 -m
Komanda ima 3 argumenata
Argumenti su:
-h
-t=1
-m
```

Operativni sistem šalje komandi niz stringova koji sledi iza imena komande. Argumenti moraju biti razdvojeni razmakom, a tokom izdvajanja argumenata operativni sistem uklanja sve suvišne razmake. *Zato se u ovom slučaju nikako ne može desiti da se kao argument prosledi prazan string.*

Primer. Napisati komandu operativnog sistema `saberi` koja sabira sve brojeve koji su navedeni iza nje. Brojevi koji su navedeni iza imena komande su realni.

⟨C# fajl⟩

```
using System;

class saberi {
  static void Main(string[] args) {
    if(args.Length > 0) {
      double sum = 0.0;
      foreach(string s in args) {
        sum += double.Parse(s);
      }
      Console.WriteLine(sum);
    }
    else {
      Console.WriteLine("Nije naveden nijedan broj");
    }
  }
}
```

Zadaci.

7.40. List papira ⟨LINK⟩

7.41. Napisati C# program koji računa aritmetičku sredinu svih brojeva koji su navedeni u ulaznoj liniji. Ulazna linija sadrži niz realnih brojeva gde su svaka dva razdvojena tačno jednim razmakom.

7.42. Napisati C# program koji nalazi najveći i najmanji od brojeva koji su navedeni u ulaznoj liniji. Ulazna linija sadrži niz realnih brojeva gde su svaka dva razdvojena tačno jednim razmakom.

7.43. Kvadratna sredina niza brojeva $(a_1, a_2, \ldots, a_n)$ se definiše ovako:

$$K_n = \sqrt{\frac{a_1^2 + a_2^2 + \ldots + a_n^2}{n}}.$$

Napisati C# program koji od korisnika učitava nekoliko brojeva i određuje i štampa njihovu kvadratnu sredinu. Brojevi se unose u jednom redu razdvojeni razmacima, a njihov broj nije unapred poznat.

7.44. U ulaznoj liniji se nalazi nekoliko celih brojeva razdvojenih razmacima. Količina nije unapred poznata. Napisati C# program koji učitava te brojeve i utvrđuje koliko njih je veće od odgovarajućeg Fibonačijevog broja. Dakle, ako korisnik unese brojeve a_1, a_2, …, a_n, program utvrdi koliko brojeva a_k ima osobinu $a_k > F_k$.

7.45. Napisati C# program koji za svaki od pozitivnih brojeva iz ulazne linije utvrđuje da li je prost. Ulazna linija sadrži niz celih brojeva gde su svaka dva razdvojena tačno jednim razmakom. Primer je dat pored.

```
Unesite brojeve
5 -3 10 0 7 9
5 je prost
(-3 se odbacuje)
10 nije prost
(0 se odbacuje)
7 je prost
9 nije prost
```

7.46. Morzeov kôd izgleda ovako:

A	·–	H	····	O	–––	U	··–
B	–···	I	··	P	·––·	V	···–
C	–·–·	J	·–––	Q	––·–	W	·––
D	–··	K	–·–	R	·–·	X	–··–
E	·	L	·–··	S	···	Y	–·––
F	··–·	M	––	T	–	Z	––··
G	––·	N	–·				

Napisati program koji od korisnika učitava string koji se sastoji samo od velikih slova i praznina i kodira ga Morzeovim kodom tako da se između kodova dva slova nalazi jedna praznina, a između reči dve praznine.

7.47. Napisati program koji učitava broj iz skupa $\{1, 2, \ldots, 2\,000\,000\,000\}$ i ispisuje taj broj rečima na srpskom jeziku. Na primer, za 328 program treba da ispiše:

```
tri stotine dvadeset osam
```

7.48. Napisati C# program koji od korisnika učitava dva razlomka i potom određuje i štampa njihovu razliku. Razlomci se učitavaju kao stringovi u obliku "a/b" gde je a neki ceo broj, a b je prirodan broj, i u istom obliku treba da bude odštampana i njihova razlika. (Primer je dat ispod.) Pretpostavlja se da će korisnik poštovati konvenciju o zapisu razlomaka i zato ne treba proveravati korektnost ulaznih podataka.

```
Unesi razlomak -> -5/6
Unesi razlomak -> 7/10
Razlika je -23/15
```

7.49. (*Hronometar*) Napisati program koji od korisnika učitava string u obliku "h:m:s" gde su h, m i s tri cela broja i proverava da li oni predstavljaju korektno zapisano vreme (sati, minuti, sekunde), odnosno, da li je $\mathtt{h} \geqslant 0$ i

$0 \leqslant$ `m`, `s` $\leqslant 59$. (Pažnja: ovo nije časovnik, već hornometar, dakle uređaj koji meri koliko je vremena proteklo!) Ukoliko se radi o korektno zapisanom vremenu, učitati pozitivan ceo broj `q` koji predstavlja neki broj sekundi, i potom ispisati novo vreme u obliku "*h*:*m*:*s*" koje se dobija tako što se na učitano vreme dodaju učitane sekunde.

Na primer, ako je korisnik uneo `31:58:30` i `q = 1251`, računar treba da ispiše `32:19:21`.

7.50. Prvi i poslednji pristup ⟨LINK⟩

7.51. Poređani datumi ⟨LINK⟩

7.52. Reč Frankenštajn ⟨LINK⟩

7.53. Prezime pa ime ⟨LINK⟩

7.54. Leksikografski minimum ⟨LINK⟩

7.55. Frekvencije reči ⟨LINK⟩

7.8 Tekstualne datoteke

Datoteka (ili fajl) je proizvoljno dugačak niz podataka istog tipa koji se nalazi u spoljašnjoj memoriji, najčešće na disku. Postoje razne vrste datoteka, a mi ćemo sada pokazati kako C# radi sa jednom posebnom vrstom datoteka koje se zovu *tekstualne datoteke* ili *tekstovi*.

Tekstualna datoteka je niz karaktera koji je smešten na disk i ima svoje ime. Na primer, kada u nekom editoru kao što je Notepad otkucamo neki tekst i "snimimo" ga, operativni sistem ga smesti u tekstualnu datoteku na disku. Programski jezik C# ima način da pristupa takvim datotekama, da ih kreira i da ih čita. U programskom jeziku C# se podaci iz datoteke mogu ili samo čitati, ili je moguć samo upis u datoteku. Nije moguće istovremeno i pisati i čitati iz iste datoteke.

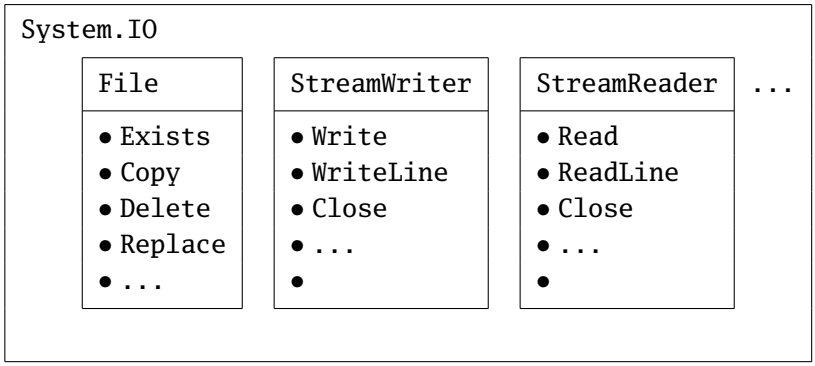
Da bismo u jeziku C# radili sa datotekama prvo moramo da uvezemo biblioteku `System.IO`, pa programi koji rade sa datotekama obično počinju ovako:

```
using System;
using System.IO;
...
```

(`IO` potiče od engl. *input-output*). Biblioteka `System.IO` sadrži mnogo klasa za rad sa datotekama, a mi ćemo koristiti sledeće tri:

- `File` – implementira opšte procese u vezi sa datotekama;

- `StreamWriter` – implementira procese vezane za pisanje podataka u datoteku;
- `StreamReader` – implementira procese vezane za čitanje podataka iz datoteke.



Neki od metoda klase `File` su:

- `File.Exists("ime.txt")` proverava da li postoji datoteka sa imenom `"ime.txt"` i vraća `true` odnosno `false`;
- `File.Copy("staro.txt", "novo.txt")` pravi kopiju datoteke `"staro.txt"` i kopija će se zvati `"novo.txt"` (datoteka `"staro.txt"` i dalje postoji);
- `File.Delete("ime.txt")` briše datoteku `"ime.txt"`;
- `File.Replace("staro.txt", "novo.txt", "bekap.txt")` prvo bekapuje datoteku `"novo.txt"` tako što joj promeni ime u `"bekap.txt"`, a onda datoteci `"staro.txt"` promeni ime u `"novo.txt"` (ukoliko datoteka `"novo.txt"` ne postoji pre početka operacije nema bekapovanja, naravno).

Pisanje u datoteku. Klasa `StreamWriter` se koristi za upisivanje u tekstualnu datoteku (engl. *stream* znači "tok", tako da ime ove klase znači "tok za upis (podataka)"). Programi koji pišu podatke u tekstualnu datoteku imaju sledeću strukturu:

- pre početka upisa napravimo novu instancu klase `StreamWriter` kojoj prosledimo ime datoteke u koju ćemo upisivati podatke;
- podatke upisujemo u datoteku metodima `Write` i `WriteLine` kao kada pišemo podatke na konzolu;
- na kraju zatvorimo datoteku metodom `Close`.

Primer. Napisati C# program koji učitava od korisnika prirodan broj n i onda u tekstualnu datoteku `"brojevi.txt"` upisuje sve brojeve iz skupa $\{1,2,\ldots,n\}$ koji nisu veći od sume kvadrata svojih cifara.

⟨C# fajl⟩

```
using System;
using System.IO;

class UpisUDatoteku {
  static void Main() {
    Console.Write("n -> ");
    int n = int.Parse(Console.ReadLine());

    StreamWriter txtFile = new StreamWriter("brojevi.txt");
    txtFile.WriteLine("Trazeni brojevi su:");
    for(int i = 1; i <= n; i++) {
      int sum = 0;
      for(int k = i; k > 0; k /= 10) {
        sum += (k % 10) * (k % 10);
      }
      if(i <= sum) {
        txtFile.WriteLine(i);
      }
    }
    txtFile.Close();
  }
}
```

Čitanje iz datoteke. Kao što ćemo videti čitanje iz datoteke je nešto komplikovanije. Klasa `StreamReader` se koristi za čitanje podataka iz tekstualne datoteke (ime ove klase znači "tok za čitanje (podataka)"). Programi koji čitaju podatke iz tekstualne datoteke imaju sledeću strukturu:

- pozivom metoda `File.Exists` proverimo da li datoteka iz koje želimo da čitamo podatke postoji;
- pre početka napravimo novu instancu klase `StreamReader` kojoj prosledimo ime datoteke iz koje čitamo podatke;
- podatke čitamo metodima `Read` i `ReadLine` kao kada čitamo podatke sa konzole;
- završavamo sa čitanjem kada u datoteci više nema podataka, što nam javlja logička vrednost `EndOfStream`;
- na kraju zatvorimo datoteku metodom `Close`.

Primer. Napisati C# program koji iz datoteke `"brojevi.txt"` čita red po red teksta i ispisuje ih sve na konzolu.

⟨C# fajl⟩

```
using System;
using System.IO;

class CitanjeIzDat {
  static void Main() {
    string fname = "brojevi.txt";
    if(File.Exists(fname)) {
      StreamReader txtFile = new StreamReader(fname);
      string s;
      while(!txtFile.EndOfStream) {
        s = txtFile.ReadLine();
        Console.WriteLine(s);
      }
      txtFile.Close();
    }
    else {
      Console.WriteLine("Fajl {0} ne postoji", fname);
    }
  }
}
```

Kako nam prilikom čitanja iz datoteke ime datoteke treba na više mesta (provera da li datoteka postoji, otvaranje toka podataka itd.), da bi se umanjila mogućnost greške najzgodnije je napraviti promenljivu (recimo `fname`) koja sadrži ime datoteke i onda nju koristiti u svim metodima koji zahtevaju da se navede ime datoteke.

Primer. Napisati C# program koji određuje dužinu svakog reda tekstualne datoteke `"ulaz.txt"` i dobijene brojeve upisuje u datoteku `"izlaz.txt"`, red po red. Na primer,

ulaz.txt	izlaz.txt
Biti il' ne biti,	17
pitanje je sad!	15
Jel' lepse u dusi trpeti	24
pracke i strele sudbe obesne,	29
ili na oruzje protiv mora bede	30
podici se i borbom	18
uciniti im kraj?	16

⟨C# fajl⟩

```
using System;
using System.IO;

class CitajPisi {
  static void Main() {
    if(File.Exists("ulaz.txt")) {
      StreamReader ulaz  = new StreamReader("ulaz.txt");
      StreamWriter izlaz = new StreamWriter("izlaz.txt");
      string s;
      while(!ulaz.EndOfStream) {
        s = ulaz.ReadLine();
        izlaz.WriteLine(s.Length);
      }
      ulaz.Close();
      izlaz.Close();
    }
    else {
      Console.WriteLine("Fajl 'ulaz.txt' ne postoji");
    }
  }
}
```

Datoteke sa komplikovanim imenom. U primerima koje smo videli datoteke sa podacima su imale jednostavna imena (`"brojevi.txt"`, `"ulaz.txt"` i slično). Da bi program našao ovako imenovanu datoteku ona mora da se nalazi u istom direktorijumu kao i odgovarajuća `.cs` datoteka u kojoj je program koji čita podatke.

Ukoliko program treba da čita podatke iz datoteke koja nije u istom direktorijumu, ime datoteke treba da sadrži *ceo put* do datoteke sa podacima, na primer ovako pod Windows operativnim sistemom:

```
c:\users\dm\primeri\ulaz.txt
```

ili, na primer, ovako ukoliko radimo pod nekom verzijom Unix operativnog sistema (gde spadaju sve verzije Linuxa, macOS, Android i iOS):

```
/usr/dm/primeri/ulaz.txt
```

U slučaju da radimo pod nekom verzijom operativnog sistema Unix nema nikakvih problema – prosto navedemo ime datoteke sa celom putanjom:

```
using System;
using System.IO;
```

```
class CitajPisiPodUnixom {
  static void Main() {
    string fname = "/usr/dm/primeri/ulaz.txt";
    if(File.Exists(fname)) {
      ...
```

Ukoliko radimo pod operativnim sistemom Windows, stvari se malo komplikuju, zato što Windows koristi simbol \ kao separator u putanji do datoteke, dok C# koristi simbol \ kao *escape* karakter. Zato se u putanji do datoteke svako pojavljivanje simbola \ mora zameniti *escape* sekvencom \\, ovako:

```
using System;
using System.IO;

class CitajPisiPodWindowsom {
  static void Main() {
    string fname = "c:\\users\\dm\\primeri\\ulaz.txt";
    if(File.Exists(fname)) {
      ...
```

Baš je interesantno da je u programskom jeziku C# (čiji proizvođač je Microsoft) jednostavnije pristupati datotekama kada se radi pod nekom verzijom Unixa, nego kada se radi pod Windowsom (čiji proizvođač je takođe Microsoft). Da bi se situacija pojednostavila i za Windows programere, u programskom jeziku C# postoji još jedna konvencija za zapisivanje stringova – *raw strings* (engl. sirovi stringovi):

☞ *ako zapis stringa počinje kombinacijom simbola* `@"` *onda se radi o stringu u kome* escape *sekvence nisu dozvoljene, tako da se svako pojavljivanje simbola* \ *tretira upravo kao taj simbol.*

Zato bi programer koji programira Windows aplikacije prethodni primer napisao ovako:

```
using System;
using System.IO;

class CitajPisiPodWindowsom2 {
  static void Main() {
    string fname = @"c:\users\dm\primeri\ulaz.txt";
    if(File.Exists(fname)) {
      ...
```

Zadaci.

7.56. Napisati C# program koji u datoj tekstualnoj datoteci utvrđuje dužinu najdužeg reda.

7.57. Napisati C# program koji u datoj tekstualnoj datoteci utvrđuje broj slova koja nisu razmaci i specijalni simboli, broj reči i broj redova.

7.58. Napisati C# program koji od tekstualne datoteke pravi novu tako što na početak svakog reda dopiše redni broj tog reda. Na primer,

`ulaz.txt`

```
Biti il' ne biti,
pitanje je sad!
Jel' lepse u dusi trpeti
pracke i strele sudbe obesne,
ili na oruzje protiv mora bede
podici se i borbom
uciniti im kraj?
```

`izlaz.txt`

```
1: Biti il' ne biti,
2: pitanje je sad!
3: Jel' lepse u dusi trpeti
4: pracke i strele sudbe obesne,
5: ili na oruzje protiv mora bede
6: podici se i borbom
7: uciniti im kraj?
```

7.59. Napisati C# program koji od tekstualne datoteke pravi novu tako što na početak svakog petog reda dopiše redni broj tog reda.

7.60. U tekstualnoj datoteci `ulaz.txt` se nalazi više redova teksta, a svaki red se sastoji iz niza brojeva razdvojenih zarezima. Napisati C# program koji čita ovu tekstualnu datoteku, i u tekstualnu datoteku `izlaz.txt` upisuje, red po red, zbirove brojeva iz svakog reda. Dakle, prvi red datoteke `izlaz.txt` sadrži zbir brojeva iz prvog reda datoteke `ulaz.txt`, i tako redom.

7.61. Napisati C# program koji formira novu tekstualnu datoteku nadovezivanjem dve tekstualne datoteke.

7.62. Napisati C# program koji upoređuje dve tekstualne datoteke i ispisuje prvi red u kome je detektovana razlika.

7.63. Jedan od najstarijih metoda šifrovanja poruka sastoji se u tome da se svako slovo poruke na sistematičan način zameni nekim drugim slovom. Na

primer, ako nam je data tabela zamene:

Slovo	a b c d e f g h i j k l m n o p q r s t u v w x y z
Zamena	q w e r t y u i o p a s d f g h j k l z x c v b n m

onda poruka *Sir ima miris* postaje *Lok odq dokol*. Niz od 26 slova

qwertyuiopasdfghjklzxcvbnm

u tabeli zamene zove se *ključ šifre*.

(*a*) Tekstualna datoteka `kljuc.txt` sadrži jedan red teksta koji predstavlja ključ šifre. Napisati C# program koji tekst u datoteci `poruka.txt` šifruje tim ključem i šifrovanu poruku upisuje u datoteku `sifra.txt`.

(*b*) Tekstualna datoteka `kljuc.txt` sadrži jedan red teksta koji predstavlja ključ šifre. Napisati C# program koji dešifruje tekst u datoteci `sifra.txt` i dobijenu poruku upisuje u datoteku `original.txt`.

7.64. Napisati C# program koji od tekstualne datoteke pravi novu tako što na početak svakog reda dopiše redni broj tog reda kao i ukupan broj redova u datoteci. Na primer,

`ulaz.txt`

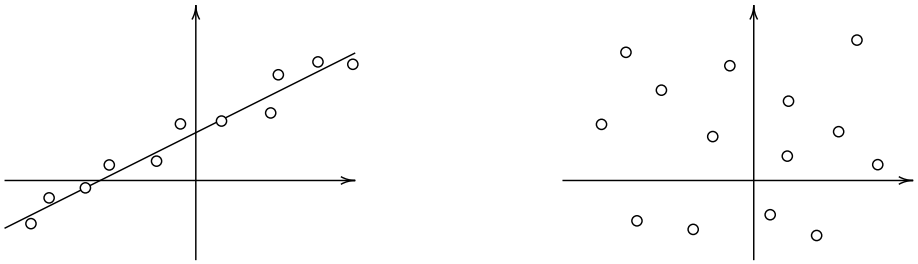
```
Biti il' ne biti,
pitanje je sad!
Jel' lepse u dusi trpeti
pracke i strele sudbe obesne,
ili na oruzje protiv mora bede
podici se i borbom
uciniti im kraj?
```

`izlaz.txt`

```
1/7: Biti il' ne biti,
2/7: pitanje je sad!
3/7: Jel' lepse u dusi trpeti
4/7: pracke i strele sudbe obesne,
5/7: ili na oruzje protiv mora bede
6/7: podici se i borbom
7/7: uciniti im kraj?
```

7.65. *Linearna regresija* je jedan metod obrađivanja eksperimentalnih rezultata za koje pretpostavljamo da se ponašaju po linearnom modelu $y = ax + b$. Pretpostavimo da smo nizom merenja neke veličine u tačkama $x_1, x_2, \ldots, x_n$ dobili izmerene vrednosti $y_1, y_2, \ldots, y_n$. Nas interesuje da li se izmerene vrednosti grupišu oko neke prave (u kom slučaju fenomen možemo opisati

linearnim modelom) ili ne. Na primer, na slici ispod, podaci levo se grupišu oko neke prave, dok za podatke desno to nije slučaj:



Broj koji pokazuje da li se dati podaci ponašaju po linearnom modelu ili ne zove se *koeficijent korelacije* i računa se po formuli:

$$r = \frac{n \cdot \sum_{i=1}^{n} x_i y_i - \left(\sum_{i=1}^{n} x_i\right) \cdot \left(\sum_{i=1}^{n} y_i\right)}{\sqrt{n \cdot \sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2} \cdot \sqrt{n \cdot \sum_{i=1}^{n} y_i^2 - \left(\sum_{i=1}^{n} y_i\right)^2}}$$

To je realan broj iz intervala $[-1,1]$. Ukoliko je r blizu 1 ili -1, radi se o fenomenu koji se može dobro opisati linearnim modelom (i tada za $r \approx 1$ imamo da će odgovarajuća prava biti rastuća, dok za $r \approx -1$ prava opada), a u ostalim slučajevima se linearni model ne preporučuje.

Nezavisno od vrednosti broja r, za svaki niz eksperimentalnih podataka $(x_1,y_1),\ldots,(x_n,y_n)$ možemo odrediti jednačinu prave koja najbolje aproksimira dati niz brojeva. Ona ima oblik $y = ax + b$ gde je

$$a = \frac{n \cdot \sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \cdot \sum_{i=1}^{n} y_i}{n \cdot \sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2} \quad \text{i} \quad b = \frac{\sum_{i=1}^{n} y_i - a \cdot \sum_{i=1}^{n} x_i}{n}$$

Prvi red tekstualne datoteke `podaci.txt` sadrži ceo broj n, nakon čega sledi još n redova koji sadrže par realnih brojeva razdvojen zarezom, na primer ovako:

```
3
-12.5,29.6
-15.5,31.28
-7.951,18.1
```

Napisati C# program koji iz tekstualne datoteke `podaci.txt` učitava ceo broj n, potom n parova realnih brojeva $(x_1, y_1), \ldots, (x_n, y_n)$ i onda računa i štampa veličine r, a i b.

Glava 8

Matrice

Matrica je pravougaona šema brojeva ili nekih drugih objekata. Dimenzije matrice zovemo još i *format matrice*. Na primer,

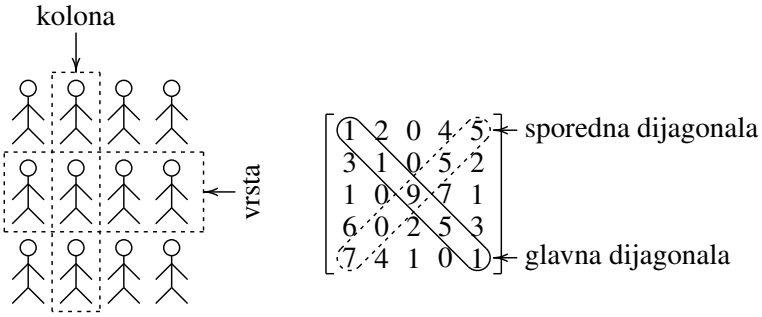
Matrica brojeva formata 3×5:

$$\begin{bmatrix} 0.12 & 3.35 & 12.47 & -1.12 & 9.96 \\ -2.44 & 0.00 & 9.67 & 12.36 & 3.37 \\ 9.85 & 3.31 & -1.00 & -2.25 & 2.78 \end{bmatrix}$$

Matrica slova formata 5×3:

$$\begin{bmatrix} a & b & c \\ f & e & d \\ g & h & i \\ l & k & j \\ m & n & o \end{bmatrix}$$

Matrica formata $m \times n$ ima m vrsta i n kolona. Na slici dole levo prikazana je matrica (čovečuljaka) formata 3×4.



Matrica kod koje su dimenzije jednake zove se *kvadratna matrica*. Kvadratna matrica ima dve dijagonale, *glavnu* i *sporednu*.

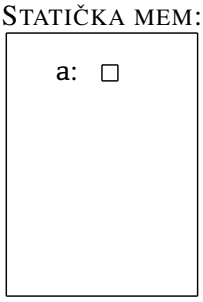
Kao i ostale nizove podataka u programskom jeziku C# elemente matrice ćemo numerisati počev od nule. Na primer, elementi neke matrice formata 3×4 će biti numerisani ovako:

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \end{bmatrix}$$

☞ *U skladu sa dogovorenim standardom za numeraciju, elementi matrice formata $n \times n$ na glavnoj dijagonali imaju oblik a_{ii}, $0 \leqslant i < n$, dok elementi na sporednoj dijagonali imaju oblik $a_{i,n-1-i}$, $0 \leqslant i < n$.*

8.1 Deklaracija matrice

U programskom jeziku C# matrice se deklarišu kao nizovi sa dva indeksa. Kao i kod nizova, odgovarajuća promenljiva je *referenca*, odnosno, "strelica" koja pokazuje na neki deo memorije. Celobrojna matrica se deklariše na sledeći način:

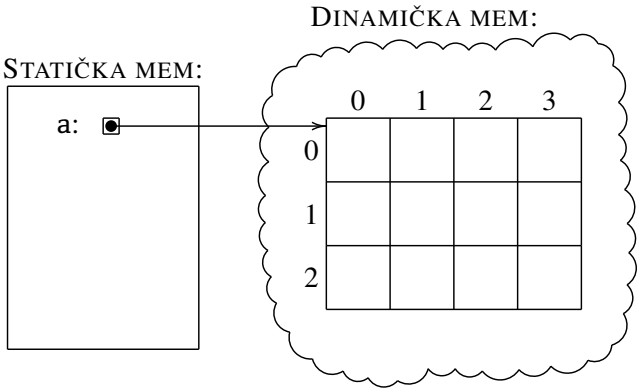



```
int[ , ] a;
```

Kao i kod nizova, ova deklaracija znači *da će* `a` *jednog dana biti matrica celih brojeva*. Za promenljivu `a` je rezervisan prostor u koji može da se upiše samo referenca na buduću matricu. Kao i kod nizova, prostor za elemente matrice se rezerviše pomoću naredbe `new`, na primer ovako:

```
a = new int[3, 4];
```

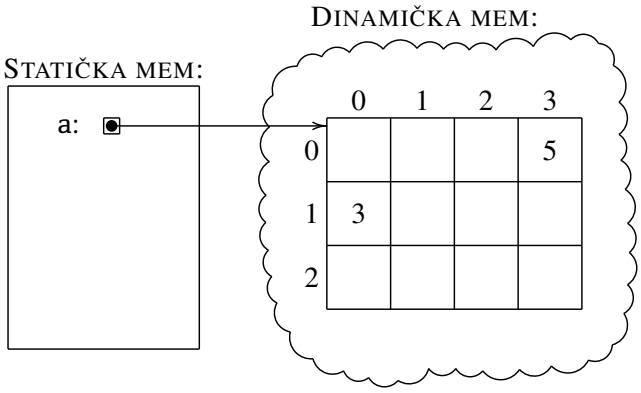
Situacija u memoriji nakon izvršenja ove naredbe je prikazana na slici ispod.



Elementima matrice se pristupa tako što se iza imena matrice u uglastim zagradama navedu dva indeksa: redni broj vrste i redni broj elementa u toj vrsti. Na primer, nakon gornje deklaracije i nakon naredbi

```
a[0, 3] = 5;
a[1, 0] = 2 * a[0, 3] - 7;
```

situacija u memoriji izgleda ovako:



Primer. U svakom od svojih p pogona korporacija "GigaKorp" proizvodi svaki od svojih n modela automobila. Pošto su pogoni locirani u različitim državama cena rada i cena nabavke/proizvodnje različitih delova automobila varira, pa se zato i proizvodna cena istog modela menja od pogona do pogona. Korporacija "GigaKorp" želi da smanji troškove proizvodnje tako što će odustati od ideje da u svakom pogonu proizvodi svaki model. Njihova nova poslovna politika je da će se svaki model automobila proizvoditi samo u onim pogonima u kojima je proizvodna cena tog modela automobila najmanja. Napisati C# program koji učitava proizvodne cene modela automobila po pogonima koji su dati ovakvom tabelom:

PRODAJA	Pogon 1	Pogon 2	Pogon 3	$\cdots$	Pogon p
Model 1	c_{11}	c_{12}	c_{13}	$\cdots$	c_{1p}
Model 2	c_{21}	c_{22}	c_{23}	$\cdots$	c_{2p}
Model 3	c_{31}	c_{32}	c_{33}	$\cdots$	c_{3p}
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$
Model n	c_{n1}	c_{n2}	c_{n3}	$\cdots$	c_{np}

gde je c_{ij} cena proizvodnje modela i u pogonu j, i potom štampa izveštaj koji za svaki model automobila sadrži spisak pogona u kojima je njegova proizvodnja najisplativija.

Da bismo rešili ovaj zadatak predstavićemo podatke o proizvodnim cenama matricom:

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{np} \end{bmatrix}.$$

i onda ćemo za svaku vrstu matrice pronaći minimum, kao i u kojim kolonama se on javlja. To su podaci koje treba ispisati.

⟨C# fajl⟩

```
using System;

class GigaKorp {
  static void Main() {
    Console.Write("Broj modela: ");
    int n = int.Parse(Console.ReadLine());
    Console.Write("Broj pogona: ");
    int p = int.Parse(Console.ReadLine());

    double[,] c = new double[n, p];
    Console.WriteLine("Unesite proizvodne cene: ");
    for(int i = 0; i < n; i++) {
      Console.WriteLine("Model {0}:", i + 1);
      for(int j = 0; j < p; j++) {
        Console.Write("  pogon {0}: ", j + 1);
        c[i, j] = double.Parse(Console.ReadLine());
      }
    }

    Console.WriteLine("Izvestaj");
    for(int i = 0; i < n; i++) {
      Console.Write("  Model {0} u pogonima:", i + 1);
      double min = c[i, 0];
      for(int j = 1; j < p; j++) {
        if(c[i, j] < min) { min = c[i, j]; }
      }
      for(int j = 0; j < p; j++) {
        if(c[i, j] == min) {
          Console.Write(" {0}", j + 1);
        }
      }
```

```
            Console.WriteLine();
        }
    }
}
```

Primer. U gradu u kome je osnovana, maloprodajna firma "FirmaKorp" ima n prodavnica. U svakoj prodavnici se može kupiti neki od m artikala. Na kraju meseca uprava firme dobije izveštaj o prodaji kao i cenovnik za taj mesec u obliku dve tabele:

PRODAJA	Artikl 1	Artikl 2	Artikl 3	$\cdots$	Artikl m
Prodavnica 1	k_{11}	k_{12}	k_{13}	$\cdots$	k_{1m}
Prodavnica 2	k_{21}	k_{22}	k_{23}	$\cdots$	k_{2m}
Prodavnica 3	k_{31}	k_{32}	k_{33}	$\cdots$	k_{3m}
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$
Prodavnica n	k_{n1}	k_{n2}	k_{n3}	$\cdots$	k_{nm}

	Cena
Artikl 1	c_1
Artikl 2	c_2
Artikl 3	c_3
$\vdots$	$\vdots$
Artikl m	c_m

gde je k_{ij} broj koji kaže koliko je u prodavnici i prodato artikala j, a c_j je cena artikla j. Napisati C# program koji od korisnika učitava tabelu sa informacijama o prodaji, potom učitava cenovnik i onda štampa izveštaj o zaradi svake prodavnice za taj mesec.

Lako se vidi da se zarada i-te prodavnice računa na sledeći način:

$$z_i = k_{i1} \cdot c_1 + k_{i2} \cdot c_2 + k_{i3} \cdot c_3 + \ldots + k_{im} \cdot c_m = \sum_{j=1}^{m} k_{ij} \cdot c_j.$$

Ovo je veoma važna operacija koja se zove *proizvod matrice i vektora* i kraće se zapisuje ovako:

$$\begin{bmatrix} k_{11} & k_{12} & \cdots & k_{1m} \\ k_{21} & k_{22} & \cdots & k_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ k_{n1} & k_{n2} & \cdots & k_{nm} \end{bmatrix} \cdot \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix}$$

gde je

$$z_i = \sum_{j=1}^{m} k_{ij} \cdot c_j.$$

Rešenje problema se, dakle, svodi na računanje proizvoda matrice i vektora.

⟨C# fajl⟩

```
using System;

class FirmaKorp {
  static void Main() {
    Console.Write("Broj prodavnica: ");
    int n = int.Parse(Console.ReadLine());
    Console.Write("Broj artikala: ");
    int m = int.Parse(Console.ReadLine());

    double[,] k = new double[n, m];
    Console.WriteLine("Unesite podatke o prodaji: ");
    for(int i = 0; i < n; i++) {
      Console.WriteLine("Prodavnica {0}:", i + 1);
      for(int j = 0; j < m; j++) {
        Console.Write("  artikl {0}: ", j + 1);
        k[i, j] = double.Parse(Console.ReadLine());
      }
    }

    double[]  c = new double[m];
    Console.WriteLine("Unesite cene artikala: ");
    for(int j = 0; j < m; j++) {
      Console.Write("  artikl {0}: ", j + 1);
      c[j] = double.Parse(Console.ReadLine());
    }

    Console.WriteLine("Zarada po prodavnicama:");
    for(int i = 0; i < n; i++) {
      double z = 0.0;
      for(int j = 0; j < m; j++) {
        z += k[i, j] * c[j];
      }
      Console.WriteLine("  Prodavnica {0}: {1:0.00}", i + 1, z);
    }
  }
}
```

Primer. U svakom od svojih p pogona korporacija "GigaKorp" proizvodi svaki od svojih n modela automobila. Proizvodni proces svakog modela se sastoji od m faza. Prilikom proizvodnje modela i faza j traje t_{ij} minuta. Pošto su pogoni locirani u različitim državama cena rada i cena nabavke repromaterijala varira, tako da faza j u pogonu k košta c_{jk} dolara na sat. Ovi podaci se mogu predstaviti u obliku dve tabele, trajanje svake faze u minutima:

VREME (min)	Faza 1	Faza 2	Faza 3	⋯	Faza m
Model 1	t_{11}	t_{12}	t_{13}	⋯	t_{1m}
Model 2	t_{21}	t_{22}	t_{23}	⋯	t_{2m}
Model 3	t_{31}	t_{32}	t_{33}	⋯	t_{3m}
⋮	⋮	⋮	⋮		⋮
Model n	t_{n1}	t_{n2}	t_{n3}	⋯	t_{nm}

i cena minuta po pogonima:

CENA ($/min)	Pogon 1	Pogon 2	Pogon 3	⋯	Pogon p
Faza 1	c_{11}	c_{12}	c_{13}	⋯	c_{1p}
Faza 2	c_{21}	c_{22}	c_{23}	⋯	c_{2p}
Faza 3	c_{31}	c_{32}	c_{33}	⋯	c_{3p}
⋮	⋮	⋮	⋮		⋮
Faza m	c_{m1}	c_{m2}	c_{m3}	⋯	c_{mp}

Napisati C# program koji od korisnika učitava podatke date u ove dve tabele i za svaki od pogona određuje proizvodnu cenu svakog modela.

Proizvodna cena i-tog modela u pogonu j se računa na sledeći način:

$$q_{ij} = t_{i1} \cdot c_{1j} + t_{i2} \cdot c_{2j} + t_{i3} \cdot c_{3j} + \ldots + t_{im} \cdot c_{mj} = \sum_{k=1}^{m} t_{ik} \cdot c_{kj}.$$

Ovo je veoma važna operacija koja se zove *proizvod dve matrice* i kraće se zapisuje ovako:

$$\begin{bmatrix} t_{11} & t_{12} & \cdots & t_{1m} \\ t_{21} & t_{22} & \cdots & t_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ t_{n1} & t_{n2} & \cdots & t_{nm} \end{bmatrix} \cdot \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{bmatrix} = \begin{bmatrix} q_{11} & q_{12} & \cdots & q_{1p} \\ q_{21} & q_{22} & \cdots & q_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ q_{n1} & q_{n2} & \cdots & q_{np} \end{bmatrix}$$

gde je

$$q_{ij} = \sum_{k=1}^{m} t_{ik} \cdot c_{kj}.$$

Rešenje problema se, dakle, svodi na računanje proizvoda dve matrice.

⟨C# fajl⟩

```
using System;

class GigaKorp2 {
  static void Main() {
    Console.Write("Broj modela: ");
    int n = int.Parse(Console.ReadLine());
    Console.Write("Broj proizvodnih faza: ");
    int m = int.Parse(Console.ReadLine());
    Console.Write("Broj pogona: ");
    int p = int.Parse(Console.ReadLine());

    double[,] t = new double[n, m];
    Console.WriteLine("Trajanje svake faze za svaki od modela: ");
    for(int i = 0; i < n; i++) {
      Console.WriteLine("Model {0}:", i + 1);
      for(int j = 0; j < m; j++) {
        Console.Write("  faza {0}: ", j + 1);
        t[i, j] = double.Parse(Console.ReadLine());
      }
    }

    double[,] c = new double[m, p];
    Console.WriteLine("Cena svake faze za svaki od pogona: ");
    for(int i = 0; i < m; i++) {
      Console.WriteLine("Faza {0}:", i + 1);
      for(int j = 0; j < p; j++) {
        Console.Write("  pogon {0}: ", j + 1);
        c[i, j] = double.Parse(Console.ReadLine());
      }
    }

    Console.WriteLine("Proizodna cena modela po pogonima:");
    for(int i = 0; i < n; i++) {
      Console.WriteLine("Model {0}:", i + 1);
      for(int j = 0; j < p; j++) {
        double q = 0.0;
        for(int k = 0; k < m; k++) { q += t[i, k] * c[k, j]; }
        Console.WriteLine("  pogon {0}: {1}", j + 1, q);
      }
    }
  }
}
```

Videli smo ranije da je moguće nizu dodeliti vrednost odmah prilikom deklaracije tipa. To je moguće i u slučaju deklaracije matrica, i tada se vrednost matrice promenljivoj dodeljuje ovako:

```
int[,] a = {{1, 2, 3, 4}, {5,  6, 7, 8}, {9, 10, 11, 12}};
```

Prethodna naredba deklariše matricu formata 3×4 i njenim elementima odmah dodeljuje vrednosti vrstu po vrstu (dakle, navedeni spisak se tretira kao spisak vrsta matrice). Ovo je isto kao da smo napisali sledeći kôd (samo kraće i elegantnije):

```
int[,] a = new int[3,4];
a[0,0] = 1;
a[0,1] = 2;
// ... itd ...
a[2,3] = 12;
```

Zadaci.

8.1. Napisati C# program koji od korisnika učitava vrednost proizvedene robe u m pogona tokom n meseci:

Vrednost robe	Pogon 1	Pogon 2	Pogon 3	$\cdots$	Pogon m
Mesec 1	c_{11}	c_{12}	c_{13}	$\cdots$	c_{1m}
Mesec 2	c_{21}	c_{22}	c_{23}	$\cdots$	c_{2m}
Mesec 3	c_{31}	c_{32}	c_{33}	$\cdots$	c_{3m}
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$
Mesec m	c_{n1}	c_{n2}	c_{n3}	$\cdots$	c_{nm}

i onda određuje i štampa: mesec u kome je produktivnost bila najmanja, najproduktivniji pogon, kao i redne brojeve pogona sa natprosečnom proizvodnjom.

8.2. Školski dnevnik nije ništa drugo do jedna matrica kod koje vrste odgovaraju učenicima jednog odeljenja, a kolone predmetima. U preseku i-te vrste i j-te kolone nalazi se broj koji predstavlja ocenu koja je i-tom učeniku zaključena iz j-tog predmeta. Napisati C# program koji iz tekstualne datoteke `ocene.txt` učitava podatke o učenicima jednog odeljenja i njihovim ocenama, a onda računa prosek za svakog učenika i svaki predmet, i njih upisuje u tekstualnu datoteku `proseci.txt`, svaki prosek u novom redu. Datoteka `ocene.txt` ima sledeću strukturu: u prvom redu se nalaze dva pozitivna cela broja N (broj učenika u odeljenju) i K (broj predmeta)

koji su razdvojeni jednim razmakom. Nakon prvog reda, u datoteci sledi još N redova, a svaki od njih sadrži tačno K brojeva koji su svi međusobno razdvojeni po jednim razmakom (ocene odgovarajućeg učenika).

8.3. Osiguravajuća kompanija svakom od svojih n klijenata pruža usluge osiguranja od svakog od m rizika. Kompanija ima tablicu u kojoj je za svakog klijenta i za svaki rizik procenjena verovatnoća p_{ij} isplate premije osiguranja kijentu i za vrstu rizika j. Osim toga, za svaki rizik se zna visina premije s_j koju kompanija isplaćuje:

Verovatnoća	Rizik 1	Rizik 2	Rizik 3	$\cdots$	Rizik m
Klijent 1	p_{11}	p_{12}	p_{13}	$\cdots$	p_{1m}
Klijent 2	p_{21}	p_{22}	p_{23}	$\cdots$	p_{2m}
Klijent 3	p_{31}	p_{32}	p_{33}	$\cdots$	p_{3m}
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$
Klijent n	p_{n1}	p_{n2}	p_{n3}	$\cdots$	p_{nm}

Rizik	Premija
Rizik 1	s_1
Rizik 2	s_2
Rizik 3	s_3
$\vdots$	$\vdots$
Rizik m	s_m

Napisati C# program koji učitava podatke iz ove dve tabele i potom ispisuje sve klijente koji imaju maksimalan rizik za kompaniju. Rizik klijenta i se računa po formuli

$$r_i = \sum_{j=1}^{m} p_{ij} s_j.$$

8.4. Množenje matrica ⟨LINK⟩

8.5. Relacija zavisnosti ⟨LINK⟩

8.6. Matrica $n \times n$ celih brojeva je magični kvadrat ako su zbirovi svih vrsta i svih kolona isti. Magični kvadrat je *jak magični kvadrat* ako su elementi matrice svi brojevi iz skupa $\{1, 2, \ldots, n^2\}$. Napisati program koji proverava da li je data matrica magični kvadrat, i ako jeste, da li je jak magični kvadrat.

8.7. Napisati C# program koji od korisnika učitava matricu formata $m \times n$ za koju se zna da je popunjena na neki način brojevima 1, 2, 4, 8, i potom štampa broj podmatrica formata 2×2 kod kojih su svi elementi različiti.

8.8. Broj bombi u okolini ⟨LINK⟩

8.9. Turnir ⟨LINK⟩

8.10. Preslikavanja matrice ⟨LINK⟩

8.11. Da li se dame napadaju ⟨LINK⟩

8.12. Najveća dijagonala ⟨LINK⟩

8.13. Spiralni ispis matrice ⟨LINK⟩

8.14. Za element a_{ij} matrice A kažemo da je *lokalni minimum* matrice ako je

$$a_{ij} < a_{i-1,j}, \quad a_{ij} < a_{i+1,j}, \quad a_{ij} < a_{i,j-1} \text{ i } a_{ij} < a_{i,j+1}.$$

Napisati C# program koji od korisnika učitava realnu matricu A formata $m \times n$ gde je $m \geqslant 3$ i $n \geqslant 3$ i onda određuje i štampa indekse svih lokalnih minimuma te matrice.

8.15. Element neke matrice se zove *sedlasta tačka* ako je on istovremeno najmanji u svojoj vrsti i najveći u svojoj koloni (sedlasta tačka tipa α), ili obrnuto, najmanji u svojoj koloni i najveći u svojoj vrsti (sedlasta tačka tipa β). Napisati program koji od korisnika učitava realnu matricu A formata $n \times n$ i računa i štampa matricu S istog formata koja je definisana na sledeći način:

$$s_{ij} = \begin{cases} 1, & a_{ij} \text{ je sedlasta tačka tipa } \alpha, \\ -1, & a_{ij} \text{ je sedlasta tačka tipa } \beta, \\ 0, & \text{inače.} \end{cases}$$

8.16. Planinarska ekspedicija se kretala pravougaonim terenom i u pravilnim intervalima merila je nadmorsku visinu. Prikupljene nadmorske visine su smeštene u matricu dimenzija $m \times n$. Kada se nalaze u nekoj tački planinari mogu da gledaju na istok, zapad, sever i jug, ali i na severoistok, severozapad, jugoistok i jugozapad.

Napisati C# program koji od korisnika učitava matricu sa nadmorskim visinama, potom cele brojeve p i q i onda određuje i štampa:

(a) broj vrhova (vrh je tačka koja ima veću nadmorsku visinu od svih osam tačaka koje je okružuju);

(b) najstrmije mesto (što su dve susedne tačke kod kojih je razlika u visinama najveća moguća);

(c) najravniju visoravan formata $p \times q$ (što je pravougaonik dimenzija $p \times q$ koji ima najmanju moguću visinsku razliku).

8.17. Populacija grada ⟨LINK⟩

8.18. Fibonačijevi podnizovi ⟨LINK⟩

8.19. Najveci NZD ⟨LINK⟩

8.20. Šahovske figure ⟨LINK⟩

8.21. Kvadratna matrica A je *strogo dijagonalno dominantna* ako je

$$|a_{ii}| > \sum_{\substack{j=0\ldots n-1 \\ j\neq i}} |a_{ij}|$$

za sve i. Napisati C# program koji utvrđuje da li je data kvadratna matrica strogo dijagonalno dominantna.

8.22. Napisati program koji za datu kvadratnu matricu računa sumu elemenata ispod glavne dijagonale.

8.23. Trouglovi i široka dijagonala ⟨LINK⟩

8.24. Napisati program koji za datu kvadratnu matricu reda n (što znači da je matrica formata $n \times n$) računa niz $d_0, \ldots, d_{n-1}$, gde je: d_0 =suma elemenata na glavnoj dijagonali, d_1 =suma elemenata na prvoj dijagonalnoj paraleli donjeg trougla matrice, d_2 =suma elemenata na drugoj dijagonalnoj paraleli donjeg trougla matrice, itd.

8.25. "Zarotirati" kvadratnu matricu za 90°.

8.26. Upisati brojeve 1, 2, ..., n^2 u kvadratnu matricu $n \times n$ po kolonama. Na primer, za $n = 4$ dobija se

$$\begin{array}{cccc} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{array}$$

8.27. *Latinski kvadrat reda n* je kvadratna matrica $S = [s_{ij}]$ formata $n \times n$ čiji elementi su iz skupa $\{1,2,\ldots,n\}$ i koja ima osobinu da su joj u svakoj vrsti svi elementi različiti i da su joj u svakoj koloni svi elementi različiti.

(a) Napisati C# program koji ispituje da li je data kvadratna matrica latinski kvadrat.

(b) Za latinski kvadrat S kažemo da je *dvostruko standardan* ako su mu elementi prve vrste i prve kolone poređani po veličini. Na primer,

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 1 & 2 & 3 \end{bmatrix}$$

Svaki latinski kvadrat se premeštanjem vrsta i kolona može dovesti u dvostruko standardan oblik. Napisati program koji dati latinski kvadrat dovodi u dvostruko standardan oblik.

(*c*) Za latinske kvadrate $S = [s_{ij}]$ i $T = [t_{ij}]$, oba reda n, kažemo da su *ortogonalni* ako je $\{(s_{ij}, t_{ij}) : 0 \leqslant i, j \leqslant n-1\} = \{(i, j) : 0 \leqslant i, j \leqslant n-1\}$. Na primer, latinski kvadrati

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{bmatrix} \quad \text{i} \quad \begin{bmatrix} 3 & 1 & 2 \\ 2 & 3 & 1 \\ 1 & 2 & 3 \end{bmatrix}$$

su ortogonalni, dok

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \\ 2 & 3 & 1 \end{bmatrix} \quad \text{i} \quad \begin{bmatrix} 2 & 3 & 1 \\ 1 & 2 & 3 \\ 3 & 1 & 2 \end{bmatrix}$$

to nisu. Napisati C# program koji proverava da li su dva latinska kvadrata istog reda ortogonalni.

(*d*) Za latinske kvadrate $S = [s_{ij}]$ i $T = [t_{ij}]$, oba reda n, kažemo da su *izotopni* ako postoji bijekcija α skupa $\{1, 2, \ldots, n\}$ takva da je $t_{ij} = \alpha(s_{ij})$ za sve i i j. Na primer, sledeća dva latinska kvadrata su izotopni

$$\begin{bmatrix} 4 & 1 & 3 & 2 \\ 3 & 2 & 1 & 4 \\ 1 & 4 & 2 & 3 \\ 2 & 3 & 4 & 1 \end{bmatrix} \sim \begin{bmatrix} 3 & 4 & 2 & 1 \\ 2 & 1 & 4 & 3 \\ 4 & 3 & 1 & 2 \\ 1 & 2 & 3 & 4 \end{bmatrix},$$

jer permutacija

$$\alpha = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 2 & 3 \end{pmatrix}$$

daje

$$\begin{bmatrix} \alpha(4) & \alpha(1) & \alpha(3) & \alpha(2) \\ \alpha(3) & \alpha(2) & \alpha(1) & \alpha(4) \\ \alpha(1) & \alpha(4) & \alpha(2) & \alpha(3) \\ \alpha(2) & \alpha(3) & \alpha(4) & \alpha(1) \end{bmatrix} = \begin{bmatrix} 3 & 4 & 2 & 1 \\ 2 & 1 & 4 & 3 \\ 4 & 3 & 1 & 2 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

Napisati C# program koji za latinske kvadrate S i T utvrđuje da li su izotopni.

8.28. Na izborima za parlament Republike Srbije struktura parlamenta se utvrđuje po principu najvećih količnika (Dontov proporcionalni sistem; dobio ime po belgijskom matematičaru Viktoru Dontu) koji radi na sledeći način. Neka parlament ima N mesta, neka je na parlamentarnim izborima učestvovalo K stranaka/koalicija, i neka su one osvojile $g_1, g_2, \ldots, g_K$ glasova, redom. Tada se formira tabela sa N vrsta i K kolona koja se popuni tako što se u j-tu vrstu upišu brojevi

$$\frac{g_1}{j}, \frac{g_2}{j}, \ldots, \frac{g_K}{j},$$

što ne moraju nužno biti celi brojevi. Potom se u tako formiranoj tabeli brojeva zaokruži najvećih N brojeva, a stranka/koalicija je osvojila onoliko mesta u parlamentu koliko u njenoj koloni ima zaokruženih brojeva. Pri tome, ako su neka dva količnika jednaka, prednost dajemo onom količniku koji je u koloni sa manjim rednim brojem (drugim rečima, prednost dajemo onoj stranki/koaliciji koja se ranije javlja na izbornom spisku).

Ulazna datoteka `ZAD.TXT` ima tačno dva reda:

$$\begin{array}{cccc} N & K & & \\ g_1 & g_2 & \cdots & g_K \end{array}$$

U svakom redu brojevi su razdvojeni tačno jednom prazninom i pri tome je $1 \leqslant N \leqslant 500$, $1 \leqslant K \leqslant 25$ i $1 \leqslant g_j \leqslant 6\,000\,000$ za sve $j \in \{1, \ldots, K\}$.

Izlazna datoteka `RES.TXT` ima tačno jedan red u kome se nalazi K brojeva

$$s_1 \quad s_2 \quad \ldots \quad s_K$$

gde su svaka dva susedna razdvojena tačno jednom prazninom. Broj s_j predstavlja broj poslaničkih mesta u parlamentu koja su pripala j-toj stranki/koaliciji.

Na primer, ako skupština ima 20 mesta, a na izbore je izašlo 4 stranke/koalicije koje su osvojile, redom, 120, 100, 40 i 20 glasova, odgovarajuća situacija je prikazana u Tabeli 8.1.

8.29. Napisati C# program koji rešava osmosmerku. Program od korisnika učitava matricu slova i niz stringova. Svaki od datih stringova se pojavljuje u matrici tačno jednom, i to na uzastopnim poljima u nekom od osam mogućih smerova. Program ispisuje slova koja preostaju u matrici (red po red, sleva na desno) kada se precrtaju pojavljivanja svih stringova.

1:	(120.00)	(100.00)	(40.00)	(20.00)
2:	(60.00)	(50.00)	(20.00)	10.00
3:	(40.00)	(33.33)	(13.33)	6.67
4:	(30.00)	(25.00)	10.00	5.00
5:	(24.00)	(20.00)	8.00	4.00
6:	(20.00)	(16.67)	6.67	3.33
7:	(17.14)	(14.29)	5.71	2.86
8:	(15.00)	12.50	5.00	2.50
9:	(13.33)	11.11	4.44	2.22
10:	12.00	10.00	4.00	2.00
11:	10.91	9.09	3.64	1.82
12:	10.00	8.33	3.33	1.67
13:	9.23	7.69	3.08	1.54
14:	8.57	7.14	2.86	1.43
15:	8.00	6.67	2.67	1.33
16:	7.50	6.25	2.50	1.25
17:	7.06	5.88	2.35	1.18
18:	6.67	5.56	2.22	1.11
19:	6.32	5.26	2.11	1.05
20:	6.00	5.00	2.00	1.00
Broj mesta:	9	7	3	1

Tabela 8.1: Dontov proporcionalni sistem za skupštinu sa 20 mesta i četiri stranke/koalicije

8.30. Dečija igračka ima oblik matrice sa m redova i n kolona. Jedno polje matrice je prazno, a na ostalim mestima su pločice sa brojevima od 1 do $mn-1$. Prazno polje je označeno brojem 0. Pločica pored prazne pozicije može sa njom da zameni mesto. Pomeranje neke pločice na levo, na desno, naviše, odnosno naniže, ćemo označiti slovima L, D, V, N, tim redom. Napisati C# program koji od korisnika učitava dimenzije igračke, početni raspored brojeva i niz poteza predstavljen slovima L, D, V, N, i prikazuje raspored pločica na igrački kada se izvrši navedeni niza poteza. Potezi se unose sve dok korisnik ne unese reč `KRAJ`.

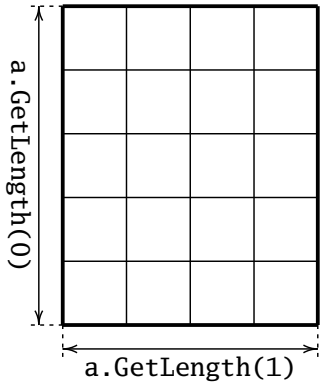
8.2 Matrice i metodi

Uvek je korisno neke standardne procedure sa objektima koji su nam interesantni implementirati u obliku metoda kako bi se njihova upotreba pojednostavila i ubrzala. U ovom odeljku ćemo pokazati nekoliko takvih metoda za rad sa matricama.

Kada želimo da prosledimo matricu nekom metodu, recimo ovako:

```
static void M(double[,] a)
```

u implementaciji metoda će nam skoro sigurno trebati način da dođemo do njenih dimenzija. Tome služi ugrađeni metod `GetLength`: izraz `a.GetLength(0)` vraća prvu dimenziju matrice (broj vrsta), dok izraz `a.GetLength(1)` vraća drugu dimenziju matrice (broj kolona).



Primer: Učitavanje i ispis matrice. Matricu učitavamo tako što učitamo njen format, a onda i elemente, jedan po jedan.

```
static double[,] Ucitaj() {
  Console.Write("m -> ");
  int m = int.Parse(Console.ReadLine());
  Console.Write("n -> ");
  int n = int.Parse(Console.ReadLine());
  double[,] rez = new double[m,n];
  for(int i = 0; i < m; i++) {
    for(int j = 0; j < n; j++) {
```

```
        Console.Write("({0}, {1}) -> ", i, j);
        rez[i,j] = double.Parse(Console.ReadLine());
      }
    }
    return rez;
  }
```

Matricu možemo da ispišemo tako što prosto izlistamo sve njene elemente

```
  static void Ispisi(double[,] a) {
    for(int i = 0; i < a.GetLength(0); i++) {
      for(int j = 0; j < a.GetLength(1); j++) {
        Console.WriteLine("({0}, {1}) -> {2}",
                          i,   j,   a[i,j]);
      }
    }
  }
```

ili, ako se radi o manjoj matrici, tako što je ispišemo kao tabelicu:

```
  static void IspisiMalu(double[,] a) {
    for(int i = 0; i < a.GetLength(0); i++) {
      for(int j = 0; j < a.GetLength(1); j++) {
        Console.Write("{0,8:N2}", a[i,j]);
      }
      Console.WriteLine();
    }
  }
```

Format `{0,8:N2}` koji se javlja u naredbi `Console.Write` znači da se argument sa rednim brojem 0 ispisuje na ukupno 8 mesta, da je to broj (`N` – *numeric*) i da ga treba zaokružiti na 2 decimale.

Primer: Sabiranje matrica. Matrice $A = [a_{ij}]$ i $B = [b_{ij}]$ istog formata $m \times n$ se mogu sabirati tako što se saberu odgovarajući elementi, i rezultat je ponovo matrica formata $m \times n$. Na primer:

$$\begin{bmatrix} 2 & 3 & -1 \\ 0 & 1 & 6 \end{bmatrix} + \begin{bmatrix} -2 & 0 & 2 \\ 1 & 9 & 4 \end{bmatrix} = \begin{bmatrix} 0 & 3 & 1 \\ 1 & 10 & 10 \end{bmatrix}.$$

Uopšteno, ako su $A = [a_{ij}]$ i $B = [b_{ij}]$ matrice istog formata, i ako je $C = A + B$ tada je $C = [c_{ij}]$ matrica istog formata kao polazne matrice, a njeni elementi su

$$c_{ij} = a_{ij} + b_{ij}.$$

Metod `IstiFormat` proverava da li su matrice istog formata, dok metod `Plus` sabira matrice `a` i `b` i vraća njihov zbir kao rezultat.

```
static bool IstiFormat(double[,] a, double[,] b) {
  return a.GetLength(0) == b.GetLength(0)
         && a.GetLength(1) == b.GetLength(1);
}

static double[,] Plus(double[,] a, double[,] b) {
  double[,] rez = new double[a.GetLength(0), a.GetLength(1)];
  for(int i = 0; i < rez.GetLength(0); i++) {
    for(int j = 0; j < rez.GetLength(1); j++) {
      rez[i,j] = a[i,j] + b[i,j];
    }
  }
  return rez;
}
```

Primer: Množenje matrice brojem. Matrica se množi brojem tako što se svaki element matrice pomnoži tim brojem. Na primer:

$$3 \cdot \begin{bmatrix} 1 & 2 & 1 & 3 \\ 0 & 1 & 1 & 1 \\ 2 & 1 & 3 & 0 \end{bmatrix} = \begin{bmatrix} 3 & 6 & 3 & 9 \\ 0 & 3 & 3 & 3 \\ 6 & 3 & 9 & 0 \end{bmatrix}.$$

Implementacija ovog postupka je veoma jednostavna.

```
static double[,] Puta(double k, double[,] a) {
  double[,] rez = new double[a.GetLength(0), a.GetLength(1)];
  for(int i = 0; i < rez.GetLength(0); i++) {
    for(int j = 0; j < rez.GetLength(1); j++) {
      rez[i,j] = k * a[i,j];
    }
  }
  return rez;
}
```

Primer: Množenje matrica. Matrica A formata $m \times p$ i matrica B formata $q \times n$ su *kompatibilne* ako je $p = q$, dakle, ako je broj kolona matrice A jednak broju vrsta matrice B.

Proizvod matrice $A = [a_{ij}]$ formata $m \times p$ i matrice $B = [b_{ij}]$ formata $p \times n$ (koje moraju biti kompatibilne!) je matrica $C = [c_{ij}]$ formata $m \times n$ čiji elementi se računaju po formuli:

$$c_{ij} = \sum_{k=0}^{p-1} a_{ik} b_{kj}.$$

Dakle, i-ta vrsta matrice A i j-ta kolona matrice B daju element na mestu ij matrice C:

$$\begin{bmatrix} \vdots & \vdots & \cdots & \vdots \\ a_{i0} & a_{i1} & \cdots & a_{i,p-1} \\ \vdots & \vdots & \cdots & \vdots \end{bmatrix} \cdot \begin{bmatrix} \cdots & b_{0j} & \cdots \\ \cdots & b_{1j} & \cdots \\ & \vdots & \\ \cdots & b_{p-1,j} & \cdots \end{bmatrix} = \begin{bmatrix} \vdots & \vdots & \vdots \\ \cdots & c_{ij} & \cdots \\ \vdots & \vdots & \vdots \end{bmatrix}$$

Metod `Kompatibilne` proverava da li su matrice kompatibilne, dok metod `Puta` množi matrice `a` i `b` i vraća njihov zbir kao rezultat.

```
static bool Kompatibilne(double[,] a, double[,] b) {
  return a.GetLength(1) == b.GetLength(0);
}

static double[,] Puta(double[,] a, double[,] b) {
  double[,] rez = new double[a.GetLength(0), b.GetLength(1)];
  for(int i = 0; i < rez.GetLength(0); i++) {
    for(int j = 0; j < rez.GetLength(1); j++) {
      rez[i,j] = 0.0;
      for(int k = 0; k < a.GetLength(1); k++) {
        rez[i,j] += a[i,k] * b[k,j];
      }
    }
  }
  return rez;
}
```

Primer: Jedinična matrica. *Jedinična matrica reda n* je matrica formata $n \times n$ koja na glavnoj dijagonali ima jednice, dok su svi ostali elementi ove matrice jednaki nuli:

$$E_5 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

Ova matrica se zove jedinična zato što je $A \cdot E = A$ i $E \cdot B = B$ kadgod su A i E kompatibilne matrice, i E i B kompatibilne matrice. Metod `JedinicnaMat` vraća jediničnu matricu reda n:

```
static double[,] Jedinicna(int n) {
  double[,] rez = new double[n, n];
  for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
      if(i == j) { rez[i,j] = 1.0; }
      else { rez[i,j] = 0.0; }
    }
  }
  return rez;
}
```

Zadaci.

8.31. Trag kvadratne matrice $A = [a_{ij}]$ formata $n \times n$ je zbir elemenata glavne dijagonale, dakle, broj $\sum_{i=0}^{n-1} a_{ii}$. Napisati metod

```
static double Trag(double[,] A)
```

koji računa trag date kvadratne matrice.

8.32. Napisati metod `static double MaxSporedni(double[,] A)` koji za datu kvadratnu matricu računa vrednost najvećeg elementa na sporednoj dijagonali.

8.33. Napisati metod `static double SumAll(double[,] A)` koji za datu matricu `A` računa sumu svih njenih elementa.

8.34. Neka je A matrica formata $m \times n$. Matrica B formata $n \times m$ se zove *transponovana matrica matrice* A, i označava sa A^T, ako je $a_{ij} = b_{ji}$ za sve i i j. Napisati metod `static double[,] Transponuj(double[,] A)` koji transponuje matricu `A`.

8.35. Kvadratna matrica H reda n (tj. formata $n \times n$) se zove *Adamarova matrica* (J. Hadamard) ako su njeni elementi samo 1 ili -1 i ako je

$$H \cdot H^T = \begin{bmatrix} n & 0 & 0 & \dots & 0 \\ 0 & n & 0 & \dots & 0 \\ 0 & 0 & n & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & n \end{bmatrix}.$$

Napisati C# program koji od korisnika učitava kvadratnu matricu i proverava da li je to Adamarova matrica.

8.36. Adamarove matrice su veoma retke. Jedan niz H_1, H_2, H_3, ..., H_n, ..., Adamarovih matrica reda 2, 4, 8, 16, 32, ..., 2^n, ... se može konstruisati na sledeći način:

$$H_1 = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \text{ i } H_{n+1} = \begin{bmatrix} H_n & H_n \\ -H_n & H_n \end{bmatrix}.$$

Na primer,

$$H_2 = \begin{bmatrix} H_1 & H_1 \\ -H_1 & H_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 \end{bmatrix}.$$

Napisati C# program koji od korisnika učitava prirodan broj n i potom računa i ispisuje Adamarovu matricu H_n definisanu na ovaj način.

8.37. Napisati metod

```
static double[,] Izbaci(double[,] A, int p, int q)
```

koji od date matrice $A = [a_{ij}]_{m\times n}$ formira matricu $B = [b_{ij}]_{(m-1)\times(n-1)}$ izbacivanjem elemenata p-vrste i q-te kolone matrice A.

8.38. Napisati metod

```
static int Podmat(double[,] A, double[,] S)
```

koji utvrđuje koliko se puta matrica `S` javlja kao podmatrica matrice `A`.

8.39. Napisati C# program koji od korisnika učitava kvadratnu matricu A, pozitivan ceo broj k i potom računa i štampa vrednost matrice S, gde je

(a) $S = E + A + A^2 + A^3 + \ldots + A^k$

(b) $S = E + \frac{1}{1!}A + \frac{1}{2!}A^2 + \frac{1}{3!}A^3 + \ldots + \frac{1}{k!}A^k$

(Napomena: E je jedinična matrica odgovarajućeg reda; pored toga, $A^2 = A \cdot A$, $A^3 = A \cdot A \cdot A$, itd.)

8.40. Napisati C# program koji od korisnika učitava pozitivan ceo broj k i kvadratne realne matrice A i B istog reda, i potom računa i štampa vrednost matrice S gde je

$$S = E + A + AB + ABA + ABAB + ABABA + \ldots + \underbrace{ABAB\ldots}_{k \text{ faktora}}.$$

8.41. Ispitati jednakost dve matrice. (Matrice $A = [a_{ij}]_{m\times n}$ i $B = [b_{ij}]_{p\times q}$ su jednake ako su istih dimenzija i ako je $a_{ij} = b_{ij}$ za sve vrednosti i i j.)

8.42. Ispitati da li je data kvadratna matrica simetrična. (Kvadratna matrica $A = [a_{i,j}]_{n\times n}$ je simetrična ako je $a_{ij} = a_{ji}$ za sve vrednosti i i j.)

8.43. Napisati program koji za dato n generiše magični kvadrat reda n.

8.44. Operacija $\triangle$ je zadata na skupu $\{0,1,\ldots,n-1\}$ pomoću matrice A dimenzija $n\times n$ tako da je

$$i\triangle j = A[i,j].$$

(Ova matrica se zove *Kejlijeva tablica* operacije.) Napisati C# metod

```
static bool Asocijativna(int[,] A)
```

koji proverava da li je operacija predstavljena matricom A asocijativna, odnosno, da li za svako $i,j,k \in \{0,1,\ldots,n-1\}$ važi:

$$(i\triangle j)\triangle k = i\triangle(j\triangle k).$$

8.45. Binarna relacija ρ na skupu $\{0,1,\ldots,n-1\}$ može da se predstavi matricom logičkih vrednosti `A` dimenzija $n\times n$ tako da je

i i j su u relaciji ρ ako i samo ako je `A[i, j] == true`.

(a) Za relaciju ρ kažemo da je *refleksivna* ako je svako $i \in \{0,1,\ldots,n-1\}$ u relaciji sa sobom. Napisati C# metod

```
static bool Refleksivna(bool[,] A)
```

koji proverava da li je relacija predstavljena matricom `A` refleksivna.

(b) Za relaciju ρ kažemo da je *simetrična* ako za svako $i,j \in \{0,1,\ldots,n-1\}$ važi sledeće: ako je i u relaciji sa j, onda je i j u relaciji sa i. Napisati C# metod

```
static bool Simetrična(bool[,] A)
```

koji proverava da li je relacija predstavljena matricom `A` simetrična.

(c) Za relaciju ρ kažemo da je *antisimetrična* ako za svako $i,j \in \{0,1,\ldots,n-1\}$ važi sledeće: ako je i u relaciji sa j i $i \neq j$ onda je j *nije* u relaciji sa i. Napisati C# metod

```
static bool Antisimetrična(bool[,] A)
```

koji proverava da li je relacija predstavljena matricom `A` antisimetrična.

(d) Za relaciju ρ kažemo da je *tranzitivna* ako za sve $i,j,k \in \{0,1,\ldots,n-1\}$ važi sledeće: ako je i u relaciji sa j i j u relaciji sa k, onda je i i u relaciji sa k. Napisati C# metod

```
static bool Tranzitivna(bool[,] A)
```

koji proverava da li je relacija predstavljena matricom A tranzitivna.

(*e*) *Refleksivno (odnosno, simetrično) zatvorenje relacije* ρ je najmanje moguće proširenje relacije ρ do relacije koja je refleksivna (odnosno, simetrična). Napisati C# metode

```
static bool[,] RflZatvorenje(bool[,] A)
static bool[,] SimZatvorenje(bool[,] A)
```

koji za datu relaciju predstavljenu matricom određuju njeno refleksivno, odnosno simetrično zatvorenje.

Glava 9

Rekurzija i globalne promenljive

U ovoj glavi uvodimo rekurziju kao jednu izuzetno značajnu programersku tehniku. Za metod kažemo da je *rekurzivan* ako se u njegovom telu javlja poziv tog istog metoda za neke druge, obično "manje" vrednosti parametara. S druge strane, ako jedan metod poziva drugi koji, opet, poziva onaj prvi, kažemo da se radi o *uzajamnoj rekurziji*. Uzajamnu rekurziju demonstriramo na primeru *parsiranja* izraza. To je pogodna prilika i da se upoznamo sa konceptom statičkih promenljivih.

9.1 Rekurzija

Za neki metod kažemo da je *rekurzivan* ako se u njegovom telu javlja poziv tog istog metoda za neke druge, obično "manje" vrednosti parametara. Uopšte, rekurzivni metodi se pišu kada posao za n možemo nekako da odradimo preko analognih poslova za neke druge vrednosti strogo manje od n. Prilikom pisanja rekurzivnih metoda moramo imati precizno formulisane odgovore na sledeća pitanja:

- Kako se tačno posao za n može svesti na poslove istog tipa, ali za manje vrednosti?
- Kada se izlazi iz rekurzije, odnosno, kako izgleda rezultat rada metoda za prvih nekoliko vrednosti broja n koje se prirodno mogu pojaviti?

Kada nam je ovo jasno, rekurzivni metod izgleda ovako:

```
static ... M(int n) {
  if (n == neka od početnih vrednosti za koje je sve trivijalno) {
    odradi posao za odgovarajuću početnu vrednost
  }
  else {
    svedi posao za n na poslove za vrednosti manje od n
    pozivajući isti ovaj metod
  }
}
```

Primer. Napisati metod koji rekurzivno računa $n!$.

Ovde prvo treba da shvatimo kako se $n!$ može izračunati preko manjih vrednosti. Primetimo da je $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1 = n \cdot (n-1)!$ Eto načina! Faktorijel broja n se računa tako što se prvo izračuna faktorijel broja $n-1$, pa se on pomnoži sa n. Početni uslov je, naravno, $0! = 1$. Evo metoda koji direktno implementira ovu formulu:

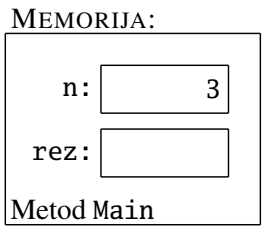
```
static int Fakt(int n) {
  if(n == 0) {
    return 1;
  }
  else {
    return n * Fakt(n - 1);
  }
}
```

Pogledajmo kako se izvršava ovaj program na primeru računanja faktorijela broja 3. Ispod je dat C# program koji od korisnika učitava n i potom računa i štampa $n!$ pri čemu smo metod koji računa $n!$ napisali malo drugačije kako bismo lakše ispratili dinamiku.

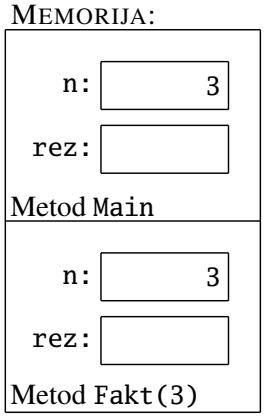
```
class TestirajFakt {
  static void Main() {
    int n, rez;
    Console.Write("n = ");
    n = int.Parse(Console.ReadLine());
    rez = Fakt(n);
    Console.WriteLine(rez);
  }

  static int Fakt(int n) {
    int rez;
    if(n == 0) { rez = 1; }
```

```
    else { rez =  n * Fakt(n - 1); }
    return rez;
  }
}
```

Kada startujemo program, za potrebe metoda `Main` u memoriji će biti napravljene dve kućice: `n` i `rez`. Potom će program učitati od korisnika neki broj i upisati ga u fioku `n`. Recimo da je korisnik uneo broj 3.



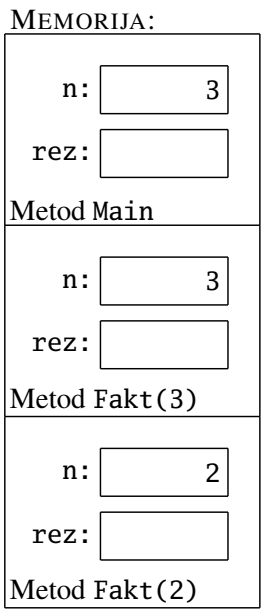
Glavni program potom pređe na izvršavanje sledeće naredbe, što je poziv metoda `Fakt` sa argumentom 3. Za potrebe metoda `Fakt` u memorji se otvore dve nove fioke, koje se takođe zovu `n` i `rez`, ali su to lokalne promenljive metoda `Fakt`.




```
using System;
class TestirajFakt {
  static void Main() {
    int n, rez;
    Console.Write("n = ");
    n = int.Parse(
      Console.ReadLine());
    rez = Fakt(n);
    Console.WriteLine(rez);
  }

  static int Fakt(int n) {
    ...
  }
}
```

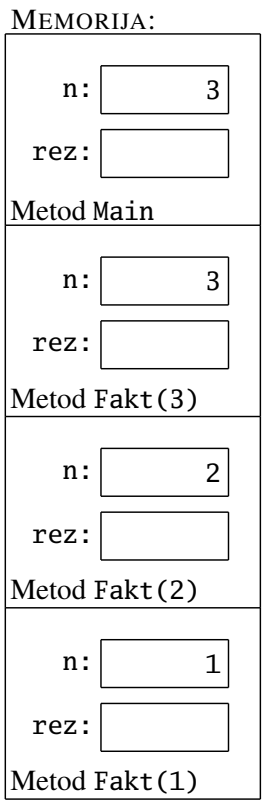
```
using System;
class TestirajFakt {
  static void Main() {
    int n, rez;
    Console.Write("n = ");
    n = int.Parse(
      Console.ReadLine());
    rez = Fakt(n);
    Console.WriteLine(rez);
  }

  static int Fakt(int n) {
    int rez;
    if(n == 0) {
      rez = 1;
    }
    else {
      rez =  n * Fakt(n - 1);
    }
    return rez;
  }
}
```

Kako je $n = 3$ izvršiće se else-grana u kojoj metod Fact poziva sam sebe, ali ovaj put za vrednost $n - 1 = 2$.



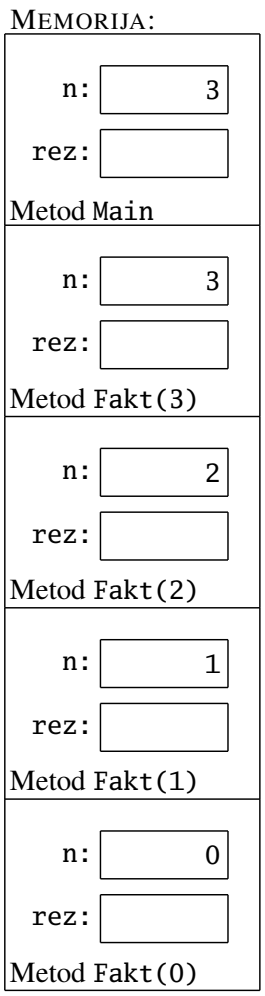

```
using System;
class TestirajFakt {
  static void Main() {
    int n, rez;
    Console.Write("n = ");
    n = int.Parse(
      Console.ReadLine());
    rez = Fakt(n);
    Console.WriteLine(rez);
  }

  static int Fakt(int n) {
    int rez;
    if(n == 0) {
      rez = 1;
    }
    else {
      rez =  n * Fakt(n - 1);
    }
    return rez;
  }
}
```

Kako je $n = 2$ izvršiće se else-grana u kojoj metod Fact poziva sam sebe, ali ovaj put za vrednost $n - 1 = 1$.



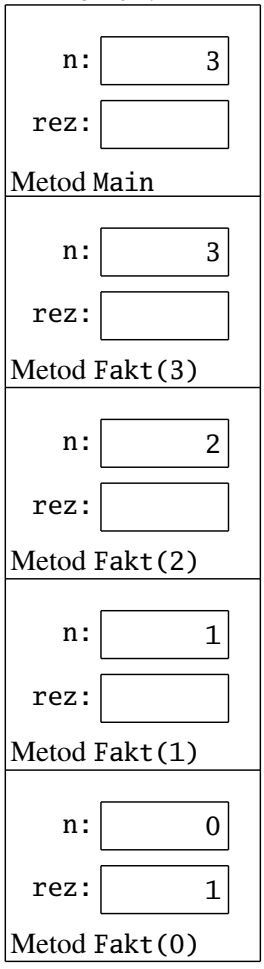

```
using System;
class TestirajFakt {
  static void Main() {
    int n, rez;
    Console.Write("n = ");
    n = int.Parse(
      Console.ReadLine());
    rez = Fakt(n);
    Console.WriteLine(rez);
  }

  static int Fakt(int n) {
    int rez;
    if(n == 0) {
      rez = 1;
    }
    else {
      rez =  n * Fakt(n - 1);
    }
    return rez;
  }
}
```

Kako je $n = 1$ izvršiće se `else`-grana u kojoj metod `Fact` poziva sam sebe, ali ovaj put za vrednost $n - 1 = 0$.



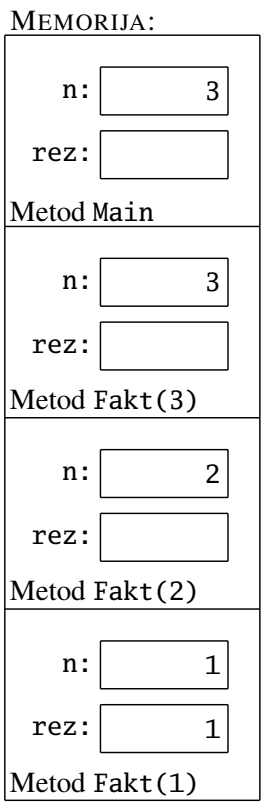

```
using System;
class TestirajFakt {
  static void Main() {
    int n, rez;
    Console.Write("n = ");
    n = int.Parse(
      Console.ReadLine());
    rez = Fakt(n);
    Console.WriteLine(rez);
  }

  static int Fakt(int n) {
    int rez;
    if(n == 0) {
      rez = 1;
    }
    else {
      rez =  n * Fakt(n - 1);
    }
    return rez;
  }
}
```

Sada je $n = 0$ – dostigli smo uslov za izlazak iz rekurzije, pa se promenljiva `rez` koja odgovara ovom pozivu metoda dobija vrednost 1.



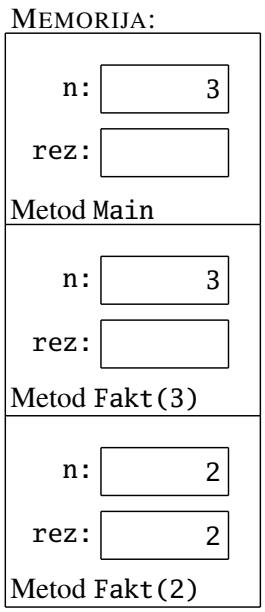

```
using System;
class TestirajFakt {
  static void Main() {
    int n, rez;
    Console.Write("n = ");
    n = int.Parse(
      Console.ReadLine());
    rez = Fakt(n);
    Console.WriteLine(rez);
  }

  static int Fakt(int n) {
    int rez;
    if(n == 0) {
      rez = 1;
    }
    else {
      rez =  n * Fakt(n - 1);
    }
    return rez;
  }
}
```

Naredbom `return` se završava poziv metoda `Fakt(0)` i rezultat se vraća u poziv metoda `Fakt(1)`.



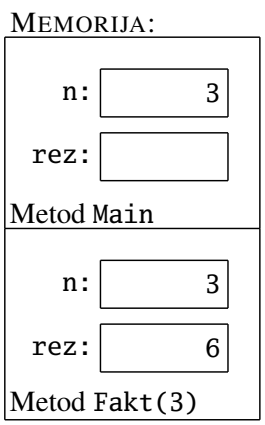

```
using System;
class TestirajFakt {
  static void Main() {
    int n, rez;
    Console.Write("n = ");
    n = int.Parse(
      Console.ReadLine());
    rez = Fakt(n);
    Console.WriteLine(rez);
  }

  static int Fakt(int n) {
    int rez;
    if(n == 0) {
      rez = 1;
    }
    else {
      rez =  n * Fakt(n - 1);
    }
    return rez;
  }
}
```

Naredbom `return` se završava poziv metoda `Fakt(1)` i rezultat se vraća u poziv metoda `Fakt(2)`.



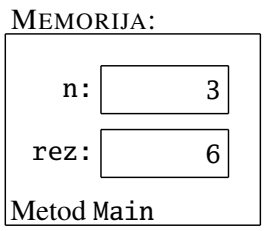

```
using System;
class TestirajFakt {
  static void Main() {
    int n, rez;
    Console.Write("n = ");
    n = int.Parse(
      Console.ReadLine());
    rez = Fakt(n);
    Console.WriteLine(rez);
  }

  static int Fakt(int n) {
    int rez;
    if(n == 0) {
      rez = 1;
    }
    else {
      rez =  n * Fakt(n - 1);
    }
    return rez;
  }
}
```

Naredbom `return` se završava poziv metoda `Fakt(2)` i rezultat se vraća u poziv metoda `Fakt(3)`.



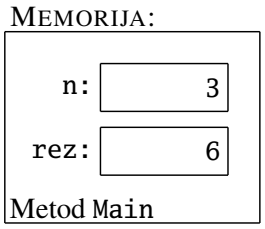

```
using System;
class TestirajFakt {
  static void Main() {
    int n, rez;
    Console.Write("n = ");
    n = int.Parse(
      Console.ReadLine());
    rez = Fakt(n);
    Console.WriteLine(rez);
  }

  static int Fakt(int n) {
    int rez;
    if(n == 0) {
      rez = 1;
    }
    else {
      rez =  n * Fakt(n - 1);
    }
    return rez;
  }
}
```

Naredbom `return` se završava poziv metoda `Fakt(3)` i rezultat se vraća u poziv metoda `Main`.




```
using System;
class TestirajFakt {
  static void Main() {
    int n, rez;
    Console.Write("n = ");
    n = int.Parse(
      Console.ReadLine());
    rez = Fakt(n);
    Console.WriteLine(rez);
  }

  static int Fakt(int n) {
    int rez;
    if(n == 0) {
      rez = 1;
    }
    else {
      rez =  n * Fakt(n - 1);
    }
    return rez;
  }
}
```

Naredba `WriteLine` konačno ispisuje rezultat rada programa.




```
using System;
class TestirajFakt {
  static void Main() {
    int n, rez;
    Console.Write("n = ");
    n = int.Parse(
      Console.ReadLine());
    rez = Fakt(n);
    Console.WriteLine(rez);
  }

  static int Fakt(int n) {
    int rez;
    if(n == 0) {
      rez = 1;
    }
    else {
      rez =  n * Fakt(n - 1);
    }
    return rez;
  }
}
```

☞ *Sa upotrebom rekurzije treba biti obazriv. Rekurzivni programi mogu da budu neefikasniji od odgovarajućih nerekurzivnih verzija.*

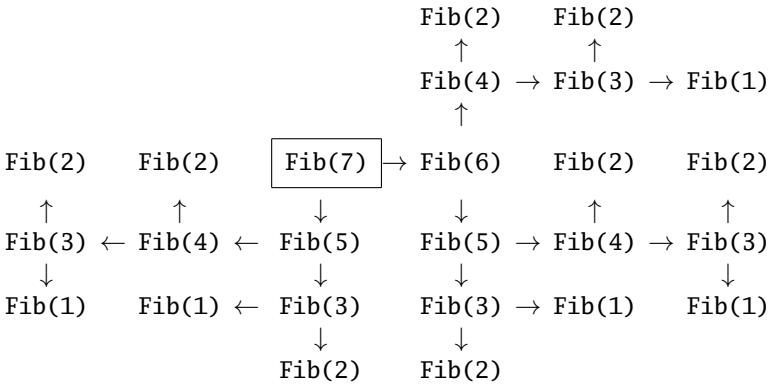
Primer. Napisati metod koji rekurzivno računa n-ti Fibonačijev broj. Fibonačijevi brojevi se definišu na sledeći način:

$$F_0 = 0, \quad F_1 = 1,$$
$$F_n = F_{n-1} + F_{n-2}, \text{ za } n \geqslant 2.$$

Vidimo da se posao računanja Fibonačijevog broja F_n se može svesti na računanje Fibonačijevih brojeva za manje vrednosti: $F_n = F_{n-1} + F_{n-2}$, dok za prve dve vrednosti imamo: $F_0 = 0$ i $F_1 = 1$.

```
static int Fib(int n) {
  if(n == 0 || n == 1) { return n; }
  else { return Fib(n - 1) + Fib(n - 2); }
}
```

Rekurzivno računanje Fibonačijevih brojeva implementirano na ovaj na čin je neuporedivo sporije od računanja pomoću "for" ciklusa koga smo videli ranije. Analizom ovog (inače korektnog) rešenja vidimo da računanje vrednosti `Fib(n)` već za relativno male vrednosti `n` zahteva veoma mnogo vremena. Naime, ako pogledamo stablo rekurzivnih poziva metoda `Fib` prilikom računanja vrednosti izraza `Fib(7)` dobijamo sledeće:



Lako se pokazuje (indukcijom) da stablo rekurzivnih poziva metoda `Fib` prilikom računanja vrednosti izraza `Fib(n)` ima $\geqslant F_n$ elemenata, a poznato je na osnovu Bineove formule da je

$$F_n \approx \left(\frac{1+\sqrt{5}}{2}\right)^n \approx 1{,}618^n.$$

Dakle, stablo rekurzivnih poziva raste eksponencijalno, pa se tako ponaša i vreme izvršavanja ovog programa.

Osim toga, ako se izlazni kriterijumi rekurzije ne postave kako treba, program veoma lako može da ostane zaglavljen u beskonačnoj rekurziji i da pukne kada potroši sve resurse sistema. Međutim, kada se pažljivo implementira, rekurzija daje efikasna i elegantna rešenja.

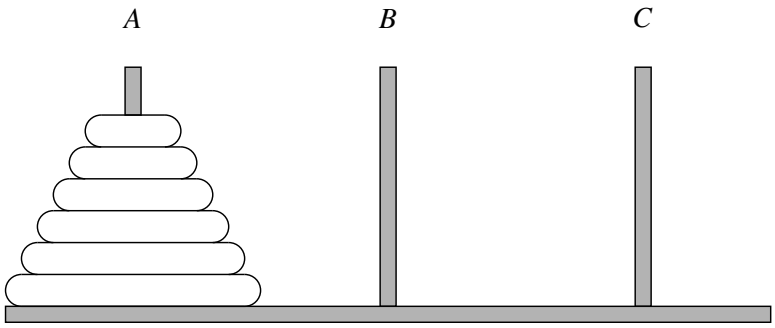
Primer. Napisati metod `void IspisiBin(int n)` koja dati nenegativan ceo broj ispisuje u binarnom sistemu.

Pre svega treba da shvatimo kako se binarni zapis broja n može dobiti na osnovu binarnog zapisa nekog manjeg broja. Pogledajmo kako teče konverzija broja n u osnovu 2 "peške": prvo n podelimo sa dva; ostatak predstavlja poslednju cifru binarnog zapisa broja, a količnik dalje konvertujemo u binarnu osnovu.

Eto pravila! Konverzija broja n u binarni zapis ide, dakle, ovako: prvo se `n / 2` konvertuje u binarni zapis, pa se na to dopiše `n % 2`. Izlazni kriterijum je jednostavan: ako je $n \in \{0,1\}$, taj broj je već zapisan u binarnom sistemu pa ga samo ispišemo.

```
static void IspisiBin(int n) {
  if(n == 0 || n == 1) {
    Console.Write(n);
  }
  else {
    IspisiBin(n / 2);
    Console.Write(n % 2);
  }
}
```

Primer: Hanojske kule. Hanojske kule predstavljaju staru igru kod koje su nam data tri štapa, A, B i C, i n diskova različitih prečnika naslaganih na štap A kao na Sl. 9.1. Cilj igre je da se ovih n diskova premeste sa A na C koristeći B kao pomoćni štap, uz ograničenje da se u svakom koraku sme premestiti samo jedan disk, pri čemu ni u jednom trenutku nije dozvoljeno da se veći disk nalazi na manjem. Napisati C# program koji rešava ovaj problem. Koliko poteza je potrebno da se premeste svih n diskova?



Slika 9.1: Hanojske kule

Rešenje. Da bismo premestili n diskova sa A na C, možemo koristiti sledeći rekurzivni pristup.

- Ako je $n = 1$ onda je lako: premestimo disk sa A na C.
- Ako je $n \geqslant 2$ onda:

- prvo premestiti gornjih $n-1$ diskova sa A na B koristeći C kao pomoćni štap;
- potom premestiti najveći disk sa A na C;
- i na kraju premestiti $n-1$ diskova sa B na C koristeći A kao pomoćni štap.

Evo C# metoda koji implementira ovu strategiju:

```
static void prebaci(int n, char A, char B, char C) {
  if(n == 1) {
    Console.WriteLine("prebaci disk sa {0} na {1}", A, C);
  }
  else {
    prebaci(n - 1, A, C, B);
    Console.WriteLine("prebaci disk sa {0} na {1}", A, C);
    prebaci(n - 1, B, A, C);
  }
}
```

Ako pozovemo metod sa `prebaci(3, 'A', 'B', 'C');` dobićemo ovakav izlaz:

```
prebaci disk sa A na C
prebaci disk sa A na B
prebaci disk sa C na B
prebaci disk sa A na C
prebaci disk sa B na A
prebaci disk sa B na C
prebaci disk sa A na C
```

Sada ćemo da odredimo koliko koraka je potrebno izvršiti da bi se ova strategija realizovala. Neka je sa h_n označen broj koraka potrebnih da se premeste svih n diskova. Analiza strategije nam daje sledeće:

Akcija	*Br. koraka*
premestiti gornjih $n-1$ diskova sa A na B	h_{n-1}
premestiti najveći disk sa A na C	1
premestiti $n-1$ diskova sa B na C	h_{n-1}

Odatle je $h_n = 2 \cdot h_{n-1} + 1, \quad n \geqslant 2$ i $h_1 = 1$. Jedinstveno rešenje ove rekurentne veze je $h_n = 2^n - 1$.

Primer. Napisati C# program koji od korisnika učitava prirodne brojeve n i M, potom n brojeva $c_0, c_1, \ldots, c_{n-1}$ i proverava da li se umetanjem znakova $+$ ili $-$ između unetih brojeva, bez menjanja redosleda unetih brojeva, može dobiti aritmetički izraz čija vrednost je M. Na primer za $n = 7$, $M = 14$ i brojeve 1, 4, 7, 5, 8, 1 izraz $1+4+7-5+8-1$ ima vrednost 14.

Rešenje. Da bismo od brojeva $c_0, \ldots, c_{n-2}, c_{n-1}$ napravili broj M pokušavamo prvo da ispred poslednjeg broja u nizu stavimo znak $+$ i da od brojeva $c_0, \ldots, c_{n-2}$ rekurzivnim pozivom napravimo broj $M - c_{n-1}$. Ako to ne uspe, pokušavamo potom da ispred poslednjeg broja u nizu stavimo znak $-$ i da od brojeva $c_0, \ldots, c_{n-2}$ rekurzivnim pozivom napravimo broj $M + c_{n-1}$. Ako ni to ne uspe, problem nema rešenje.

Evidenciju o operacijama koje smo umetnuli ispred brojeva vodimo u pomoćnom nizu karaktera `op` ovako: `op[i] == '+'` znači da smo ispred i-tog broja umetnuli znak $+$; dok `op[i] == '-'` znači da smo ispred i-tog broja umetnuli znak $-$.

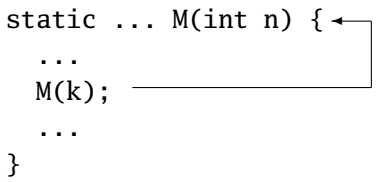
⟨C# fajl⟩

```
using System;
class MiniSlagalica {
  static void Main() {
    Console.WriteLine("n -> ");
    int n = int.Parse(Console.ReadLine());
    var a = new int[n];
    Console.WriteLine("Brojevi -> ");
    for(int i = 0; i < n; i++) {
      a[i] = int.Parse(Console.ReadLine());
    }
    Console.WriteLine("Rez -> ");
    int M = int.Parse(Console.ReadLine());
    var op = new char[n];
    bool moze = Pokusaj(a, op, n - 1, M);
    if(moze) {
      IspisiResenje(a, op, n, M);
    }
    else {
      Console.WriteLine("Ne moze");
    }
  }

  static bool Pokusaj(int[] a, char[] op, int i, int M) {
    if(i == 0) { return a[0] == M; }
    op[i] = '+';
    bool moze = Pokusaj(a, op, i - 1, M - a[i]);
```

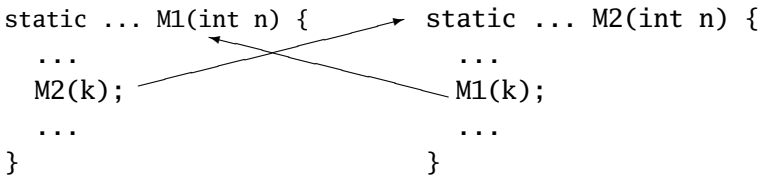
```
    if(moze) { return true; }
    op[i] = '-';
    moze = Pokusaj(a, op, i - 1, M + a[i]);
    if(moze) { return true; }
    return false;
  }

  static void IspisiResenje(int[] a, char[] op, int n, int M) {
    Console.Write(a[0]);
    for(int i = 1; i < n; i++) {
      Console.Write("{0}{1}", op[i], a[i]);
    }
    Console.WriteLine("={0}", M);
  }
}
```

9.2 Uzajamna rekurzija

Kao što smo videli, rekurzivni metod ima ovakvu strukturu:



Često se, međutim, dešava da je potrebno napisati dve (ili više) procedura koje pozivaju jedna drugu:



Nijedan od metoda `M1` i `M2` nije na prvi pogled rekurzivan, ali cela konstrukcija jeste vrsta rekurzije. U situaciji kao što je ova kažemo da se radi o *uzajamnoj rekurziji*.

Ideju uzajamne rekurzije ćemo demonstrirati na primeru parsera za algebarske izraze. *Parsiranje* (engl. *parsing*) je proces tokom koga računar učitava niz karaktera formiran po nekim unapred određenim pravilima i pokušava da na pravilan način interpretira značenje tog niza znakova. Na primer, svaki kompajler kao svoj osnovni deo ima parser, odnosno, program koji proverava da li je uneti niz karaktera formiran prema pravilima datog programskog jezika. U slučaju kompajlera,

značenje unetog niza znakova predstavlja mašinski kôd koga kompajler generiše ako se ispostavi da je uneti niz znakova smislen.

Međutim, pre nego što pokažemo primer parsera celobrojnih algebarskih izraza objasnićemo još jedan koncept programskog jezika C#: statičke promenljive.

9.3 Statičke promenljive

Statičke promenljive su promenljive koje pripadaju klasi. One se deklarišu na nivou klase tako što se ispred uobičajene deklaracije promenljive doda reč `static`, na primer ovako:

```
using System;
class PrimerStatProm {
  static int N = 0;

  static void Main() {
    N = int.Parse(Console.ReadLine());
    Console.WriteLine("Sada je N = {0}", N);
  }
}
```

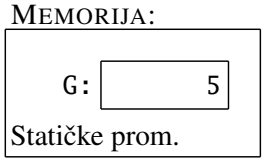
Statičke promenljive su vidljive u svim metodima te klase i svi metodi mogu da im pristupaju i da ih menjaju. Zato za njih kažemo još i da su *globalne*. S druge strane, promenljive deklarisane unutar metoda kao i argumenti metoda se zovu *lokalne promenljive*. Na primer:

```
using System;
class PrimerStatProm {
  static int N = 0; // globalna promenljiva

  static void Main() {
    int k; // lokalna promenljiva
    k = int.Parse(Console.ReadLine());
    N = k * k; // menjamo globalnu promenljivu
    Console.WriteLine("Sada je N = {0}", N);
  }
}
```

Statičke promenljive postoje sve dok program radi, a lokalne promenljive postoje samo dok je odgovarajući metod aktivan. Zato su statičke promenljive i dobile ime: one su statičke jer postoje sve vreme izvršavanja programa. Prostor za lokalne

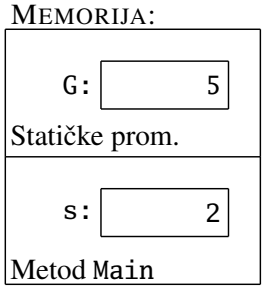
promenljive metoda se rezerviše kada se metod aktivira, i uklanja se iz memorije čim se metod završi. Zato su lokalne promenljive na raspolaganju samo unutar metoda. Kažemo još da se *lokalne promenljive ne vide spolja*. Pogledajmo jedan mali primer.

U programu pored `G` je statička (globalna) promenljiva i čim se program pokrene ona bude inicijalizovana na 5.



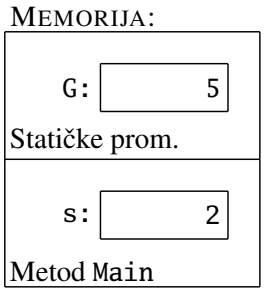

```
using System;
class StatProm1 {
  static int G = 5;
  static void Main() {
    int s = 2;
    M(s);
    M(s);
  }

  static void M(int n) {
    int k = int.Parse(;
      Console.ReadLine());
    G += n * k;
  }
}
```

Nakon inicijalizacije statičkih promenljivih se pokrene metod `Main`. On ima jednu *lokalnu* promenljivu `s` koja odmah bude inicijalizovana na 2.



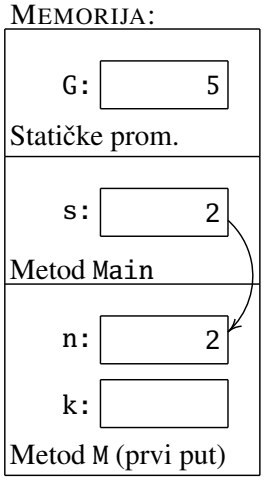

```
using System;
class StatProm1 {
  static int G = 5;
  static void Main() {
    int s = 2;
    M(s);
    M(s);
  }

  static void M(int n) {
    int k = int.Parse(;
      Console.ReadLine());
    G += n * k;
  }
}
```

U sledećem koraku metod `Main` poziva metod `M` sa argumentom `s`.



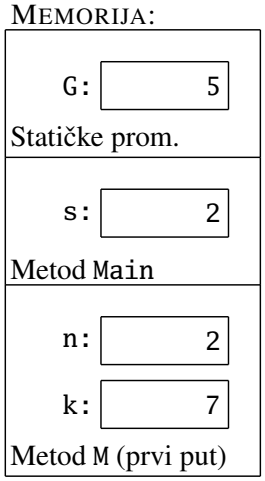

```
using System;
class StatProm1 {
  static int G = 5;
  static void Main() {
    int s = 2;
    M(s);
    M(s);
  }

  static void M(int n) {
    int k = int.Parse(;
      Console.ReadLine());
    G += n * k;
  }
}
```

Tokom aktivacije metoda `M` rezerviše se prostor za njegove (lokalne!) promenljive i mehanizam prenosa parametara inicijalizuje `n` na vrednost promenljive `s`.



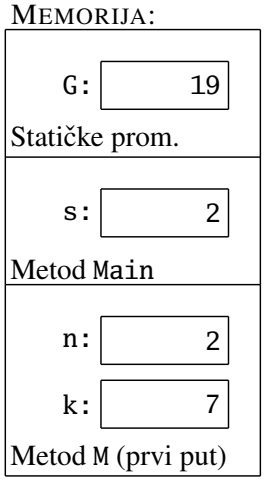

```
using System;
class StatProm1 {
  static int G = 5;
  static void Main() {
    int s = 2;
    M(s);
    M(s);
  }

  static void M(int n) {
    int k = int.Parse(;
      Console.ReadLine());
    G += n * k;
  }
}
```

U sledećem koraku metod `M` od korisnika učitava neki ceo broj. Recimo da je korisnik uneo broj 7.




```
using System;
class StatProm1 {
  static int G = 5;
  static void Main() {
    int s = 2;
    M(s);
    M(s);
  }

  static void M(int n) {
    int k = int.Parse(;
      Console.ReadLine());
    G += n * k;
  }
}
```

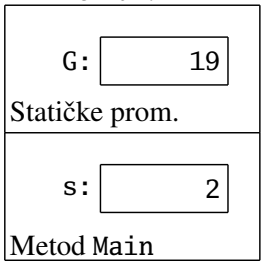
Poslednja naredba metoda `M` na *globalnu* promenljivu `G` dodaje vrednost izraza `n * k`. Tako `G` dobija vrednost 19.




```
using System;
class StatProm1 {
  static int G = 5;
  static void Main() {
    int s = 2;
    M(s);
    M(s);
  }

  static void M(int n) {
    int k = int.Parse(;
      Console.ReadLine());
    G += n * k;
  }
}
```

Nakon toga se metod `M` završava, njegove lokalne promenljive se uklanjaju iz memorije i kontrola se vraća metodu `Main`. Primetimo da je *globalna* promenljiva `G` zadržala svoje novo stanje!

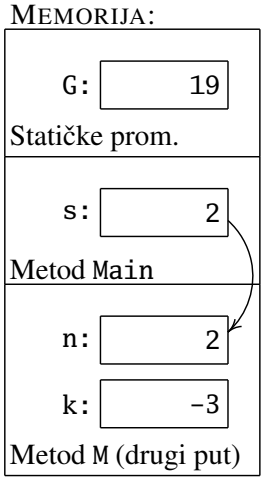
MEMORIJA:




```
using System;
class StatProm1 {
  static int G = 5;
  static void Main() {
    int s = 2;
    M(s);
    M(s);
  }

  static void M(int n) {
    int k = int.Parse(;
      Console.ReadLine());
    G += n * k;
  }
}
```

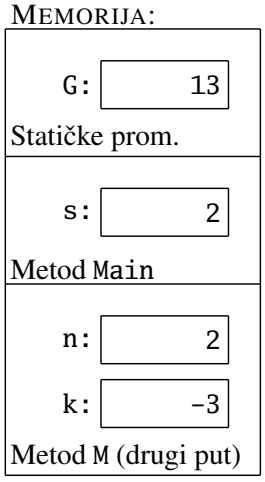
Kao narednu instrukciju metod `Main` po drugi put zove metod `M`. Ponovo se rezerviše prostor za lokalne promenljive metoda `M`, prenose se parametri i od korisnika se učitava neka vrednost. Recimo da je korisnik sada uneo broj -3.

MEMORIJA:



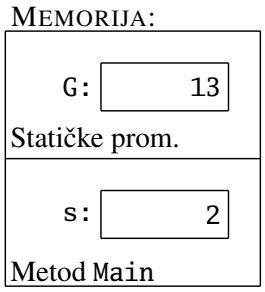

```
using System;
class StatProm1 {
  static int G = 5;
  static void Main() {
    int s = 2;
    M(s);
    M(s);
  }

  static void M(int n) {
    int k = int.Parse(;
      Console.ReadLine());
    G += n * k;
  }
}
```

Poslednja instrukcija metoda `M` na promenljivu `G` dodaje vrednost izraza `n * k`, što je u ovom slučaju -6, tako da promenljiva `G` sada sadrži broj 13.



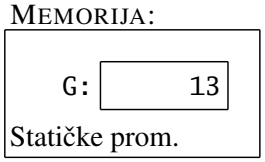

```
using System;
class StatProm1 {
  static int G = 5;
  static void Main() {
    int s = 2;
    M(s);
    M(s);
  }

  static void M(int n) {
    int k = int.Parse(;
      Console.ReadLine());
    G += n * k;
  }
}
```

Nakon toga se metod `M` završava, njegove lokalne promenljive se uklanjaju iz memorije i kontrola se vraća metodu `Main`. Primetimo da je *globalna* promenljiva `G` i ovaj put zadržala svoje novo stanje!




```
using System;
class StatProm1 {
  static int G = 5;
  static void Main() {
    int s = 2;
    M(s);
    M(s);
  }

  static void M(int n) {
    int k = int.Parse(;
      Console.ReadLine());
    G += n * k;
  }
}
```

Konačno, završava se i metod `Main` zbog čega se i njegove promenljive uklanjaju iz memorije.




```
using System;
class StatProm1 {
  static int G = 5;
  static void Main() {
    int s = 2;
    M(s);
    M(s);
  }

  static void M(int n) {
    int k = int.Parse(;
      Console.ReadLine());
    G += n * k;
  }
}
```

Statičke promenljive se uklanjaju iz memorije tek kada se zaši ceo program. Važno je uočiti sledeću ključnu razliku između promenljivih metoda `Main` i statičkih promenljivih:

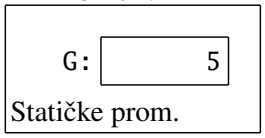
☞ *Iako imaju manje-više isti životni ciklus, promenljive metoda* `Main` *su vidljive samo u metodu* `Main`*, dok su statičke promenljive vidljive u* ***svim*** *metodima.*

U prethodnom primeru smo videli i to da svaki metod klase može da menja vrednosti statičkih promenljivih. Tada se kaže da metod pravi *bočne efekte* (engl. *side effects*). Ovu mogućnost treba koristiti umereno i veoma pažljivo zato što prekomerna upotreba bočnih efekata može da dovede do nejasnih i nečitkih programa, kao i do grešaka koje se veoma teško otkrivaju.

Lokalna promenljiva može da ima isto ime kao neka statička promenljiva. Tada se unutar metoda odgovarajuća statička promenljiva "ne vidi" zato što je "prekrivena" istoimenom lokalnom promenljivom. Evo primera.

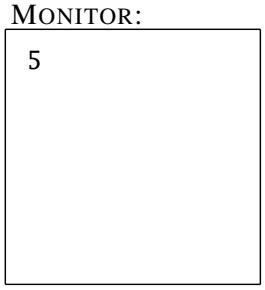
U primeru pored klasa ima statičku promenljivu G, ali i metod M ima lokalnu promenljivu G. Kada se program pokrene rezerviše se prostor za statičku promenljivu G koja ujedno dobije vrednost 5.

MEMORIJA:



Nakon toga se aktivira metod Main čija prva naredba ispiše vrednost promenljive G. Kako metod Main nema lokalnu promenljivu koja se zove G, a postoji statička (globalna) promenljiva sa tim imenom, ispisuje se vrednost statičke promenljive G:

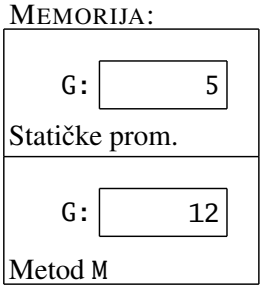
MONITOR:



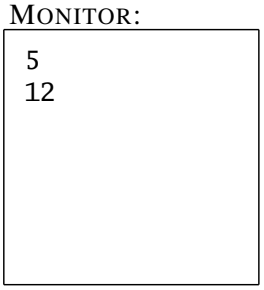

```
using System;
class StatProm2 {
  static int G = 5;
  static void Main() {
    Console.WriteLine(G);
    M();
    Console.WriteLine(G);
  }

  static void M() {
    int G = 12;
    Console.WriteLine(G);
  }
}
```

```
using System;
class StatProm2 {
  static int G = 5;
  static void Main() {
    Console.WriteLine(G);
    M();
    Console.WriteLine(G);
  }

  static void M() {
    int G = 12;
    Console.WriteLine(G);
  }
}
```

Sledi poziv metoda `M` za čije lokalne promenljive se u fazi aktivacije metoda rezerviše memorijski prostor. On ima jednu lokalnu promenljivu koja se zove `G` i koja odmah dobija vrednost 12.



Kako metod `M` ima svoju promenljivu `G`, naredba `WriteLine` u metodu `M` ispisuje vrednost *lokalne* promenljive `G`.

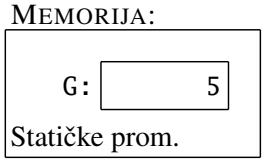



```
using System;
class StatProm2 {
  static int G = 5;
  static void Main() {
    Console.WriteLine(G);
    M();
    Console.WriteLine(G);
  }

  static void M() {
    int G = 12;
    Console.WriteLine(G);
  }
}
```

```
using System;
class StatProm2 {
  static int G = 5;
  static void Main() {
    Console.WriteLine(G);
    M();
    Console.WriteLine(G);
  }

  static void M() {
    int G = 12;
    Console.WriteLine(G);
  }
}
```

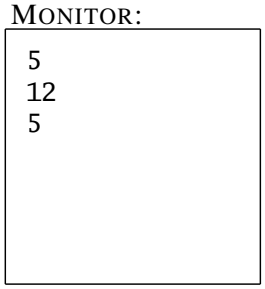
Primetimo kako je lokalna promenljiva metoda `M` "prekrila" statičku promenljivu `G`!

Kada se metod M završi iz memorije se uklanjaju lokalne promenljive koje su mu pripadale:




```
using System;
class StatProm2 {
  static int G = 5;
  static void Main() {
    Console.WriteLine(G);
    M();
    Console.WriteLine(G);
  }

  static void M() {
    int G = 12;
    Console.WriteLine(G);
  }
}
```

Druga WriteLine naredba metoda Main ponovo ispisuje 5 zato što metod Main nema lokalnu promenljivu koja se zove G, a postoji statička (globalna) promenljiva sa tim imenom:




```
using System;
class StatProm2 {
  static int G = 5;
  static void Main() {
    Console.WriteLine(G);
    M();
    Console.WriteLine(G);
  }

  static void M() {
    int G = 12;
    Console.WriteLine(G);
  }
}
```

I za kraj samo još jedna kratka terminološka diskusija:

☞ *Pojam "statički" se u programskom jeziku C# koristi u raznim kontekstima i ima više značenja.* ***Statičke promenljive*** *o kojima smo govorili u ovom odeljku ne moraju biti u direktnoj vezi sa* ***statički alociranim promenljivim*** *o kojima smo govorili ranije, recimo kada smo pričali o nizovima.*

Statička promenljiva, u smislu promenljive koja je vidljiva u svim metodima klase, može biti alocirana u statičkoj memoriji, ali može biti alocirana i u dinamičkoj memoriji. S druge strane, lokalna promenljiva metoda se ne deklariše kao statička, iako se prostor za nju alocira u statičkoj memoriji.

9.4 Primer uzajamne rekurzije: parsiranje

Kao primer uzajamne rekurzije i kao demonstraciju rada sa statičkim promenljivim u ovom odeljku ćemo pokazati jednostavan program koji parsira celobrojne aritmetičke izraze i računa vrednost onih koji su korektno formirani. Pravila koja opisuju korektne celobrojne aritmetičke izraze se mogu zapisati ovako:

$$\langle \textit{Cifra} \rangle \equiv \text{“0”} \mid \text{“1”} \mid \text{“2”} \mid \ldots \mid \text{“9”}$$

$$\langle \textit{OpMnoženja} \rangle \equiv \text{“}*\text{”} \mid \text{“}/\text{”} \mid \text{“}\%\text{”}$$

$$\langle \textit{OpSabiranja} \rangle \equiv \text{“}+\text{”} \mid \text{“}-\text{”}$$

$$\langle \textit{Broj} \rangle \equiv \langle \textit{Cifra} \rangle \ \{ \ \langle \textit{Cifra} \rangle \ \}$$

$$\langle \textit{Činilac} \rangle \equiv \langle \textit{Broj} \rangle \mid \text{“(”} \ \langle \textit{Izraz} \rangle \ \text{“)”}$$

$$\langle \textit{Sabirak} \rangle \equiv \langle \textit{Činilac} \rangle \ \{ \ \langle \textit{OpMnoženja} \rangle \ \langle \textit{Činilac} \rangle \ \}$$

$$\langle \textit{Izraz} \rangle \equiv [\text{“}+\text{”} \mid \text{“}-\text{”}] \ \langle \textit{Sabirak} \rangle \ \{ \ \langle \textit{OpSabiranja} \rangle \ \langle \textit{Sabirak} \rangle \ \}$$

Vertikalna crta u gornjim zapisima označava alternativu, jednu od ponuđenih mogućnosti. Vitičaste zagrade { i } označavaju da se izraz u zagradi može pojaviti proizvoljno mnogo puta, ali se i ne mora pojaviti. Uglaste zagrade [i] označavaju da je izraz u zagradi opcioni – može se pojaviti, a ne mora.[1] Navedena pravila dakle znače sledeće:

- cifra je neki od karaktera 0, ..., 9;
- operator množenja je neki od karaktera $*$ (množenje), $/$ (količnik pri celobrojnom deljenju) i % (ostatak pri celobrojnom deljenju);
- operator sabiranja je jedan od karaktera $+$ (sabiranje) ili $-$ (oduzimanje);
- broj je proizvoljan niz cifara koji ima bar jednu cifru;
- činilac je broj ili izraz okružen zagradama;
- sabirak je činilac ili niz činilaca razdvojenih operatorom množenja;
- izraz može imati opcioni predznak $+$ ili $-$, nakon čega sledi sabirak, ili niz sabiraka razdvojenih operatorima sabiranja.

[1] Ova konvencija za zapisivanje formalnih (tj. veštačkih) jezika se zove EBNF što je skraćenica od *Extended Backus-Naur Formalism.*

Jedna od osnovnih tehnika konstrukcije kompajlera koju ćemo ovde demonstrirati se zove *tehnika rekurzivnog spusta* (engl. *recursive descent*) prema kojoj svakom pravilu dodelimo jedan metod koji prihvata karaktere ili poziva druge metode onim redom kojim su oni navedeni u odgovarajućem pravilu. U našem slučaju svakom pravilu ćemo dodeliti jedan metod koji vraća vrednost dela izraza koga je pročitao iz ulazne linije i prepoznao.

Krenućemo sada sa prezentovanjem parsera za celobrojne izraze. Kôd ćemo detaljno komentarisati, a integralna verzija se može dobiti klikom na link pored. Da bi unos izraza bio udobniji za korisnika napisaćemo parser koji ignoriše praznine. Tako korisnik može da unese svaki od sledećih izraza:

⟨C# fajl⟩

```
(1+2)*(5+6)-11
    (1 + 2)*(5 + 6) - 11
(1 + 2)*   (5 + 6) - 11
( 1 + 2 )  *  ( 5+6   ) - 11
```

i svi će oni biti prihvaćeni kao korektni izrazi čija vrednosti je 22.

Program radi tako što učita izraz u string i onda se kreće po stringu sa željom da razume njegovu strukturu. Program zato ima statičku promenljivu `S` koja sadrži niz simbola koje je uneo korisnik, kao i statičku promenljivu `I` koja sadrži sledeću poziciju u stringu `S` koju treba analizirati:

```
using System;

class Parser {
  static string S;
  static int    I;
```

Pošto ćemo u nekim situacijama morati da donosimo odluke o tome koje pravilo treba primeniti (tj. koji metod pozvati), kako bi se nastavilo sa parsiranjem izraza standardna preporuka je da se prilikom implementacije parsera tehnikom rekurzivnog spusta uvek gleda jedan simbol unapred. Taj simbol se zove *lookahead* (što bi se sa engleskog prevelo otprilike kao "virni unapred") i to je u našem slučaju `S[I]`. Zato nijedan od metoda u ovom programu nema argumente: svi metodi čitaju i menjaju statičke promenljive `S` i `I` i na taj način komuniciraju.

Glavni program ima veoma jednostavnu strukturu:

```
  static void Main() {
    Console.WriteLine("Unesi celobrojni izraz");
    S = Console.ReadLine();
    I = 0;
    PreskociPraznine();
    int rez = Izraz();
```

```
    if(I < S.Length) { GRESKA("Neocekivani simbol"); }
    Console.WriteLine(rez);
  }
```

Nakon što niz simbola koji predstavlja celobrojni izraz učitamo u string `S` postavimo vrednost promenljive `I` na 0 jer je `S[0]` prvi karakter koga treba analizirati i pozivom metoda `PreskociPraznine` preskocimo praznine koje je korisnik možda uneo na početku izraza.

Nakon toga pozovemo metod `Izraz` koji implementira pravilo $\langle Izraz \rangle$. Ovaj metod vrati vrednost izraza i ona bude smeštena u promenljivu `rez`. Ako se metod `Izraz` završio a proces parsiranja stringa `S` nije stigao do kraja stringa `S` onda treba prijaviti grešku. Na primer, to se može desiti ako korisnik iza izraza unese još neki simbol, recimo ovako: `12 + 3 ;`. Ako to nije slučaj, glavni metod ispisuje vrednost koju je metod `Izraz` vratio.

Metod `GRESKA` ispisuje poruku o vrsti greške koja je identifikovana i prekida rad programa:

```
  static void GRESKA(string poruka) {
    Console.WriteLine("Greska na poziciji {0}: {1}", I + 1, poruka);
    System.Environment.Exit(0);
  }
```

Naredba `System.Environment.Exit(0);` prekida izvršavanje programa. Ovaj pristup obradi grešaka (prekidamo sa izvršavanjem programa čim naiđemo na prvu grešku) će nam u mnogome olakšati pisanje parsera.

Metod `PreskociPraznine` ima direktnu i jasnu implementaciju:

```
  static void PreskociPraznine() {
    while(I < S.Length && S[I] == ' ') { I++; }
  }
```

Prelazimo sada na implementacija pravila koje smo naveli na početku odeljka. Prva tri pravila

$$\langle Cifra \rangle \equiv \text{“0”} \mid \text{“1”} \mid \text{“2”} \mid \ldots \mid \text{“9”}$$

$$\langle OpMnoženja \rangle \equiv \text{“}*\text{”} \mid \text{“}/\text{”} \mid \text{“}\%\text{”}$$

$$\langle OpSabiranja \rangle \equiv \text{“}+\text{”} \mid \text{“}-\text{”}$$

implementiramo kao metode koji vraćaju logičku vrednost ukoliko su prepoznali odgovarajući simbol:

```
static bool Cifra() {
  return '0' <= S[I] && S[I] <= '9';
}

static bool OpMnozenja() {
  return S[I] == '*' || S[I] == '/' || S[I] == '%';
}

static bool OpSabiranja() {
  return S[I] == '+' || S[I] == '-';
}
```

Pravilo koje opisuje brojeve

$$\langle Broj \rangle \equiv \langle Cifra \rangle \ \{ \ \langle Cifra \rangle \ \}$$

implementiramo metodom koji procesira niz cifara i odmah računa i kao rezultat svog rada vraća odgovarajuću celobrojnu vrednost:

```
static int Broj() {
  int n = 0;
  while(I < S.Length && Cifra()) {
      n = 10 * n + ((int)S[I] - (int)'0');
      I++;
  }
  PreskociPraznine();
  return n;
}
```

Pošto se iza zapisa broja mogu pojaviti praznine koje po dogovoru preskačemo, pre nego što vrati odgovarajuću celobrojnu vrednost poziva se metod `PreskociPraznine`.

Činilac je podizraz koji učestvuje u građenju izraza povezanih multiplikativnim operatorima. Pravilo koje opisuje činioce je jednostavno – to može biti broj ili izraz okružen zagradama:

$$\langle \check{C}inilac \rangle \equiv \langle Broj \rangle \mid \text{“(”} \ \langle Izraz \rangle \ \text{“)”}$$

Zato metod `Cinilac` razlikuje dva slučaja: ako je `S[I]` (*lookahead* simbol) cifra, poziva se metod koji procesira brojeve, a ako je `S[I]` otvorena zagrada poziva se pravilo koje procesira izraze, s tim da nakon toga moramo proveriti da li sledi odgovarajuća zatvorena zagrada:

```
static int Cinilac() {
  if(I == S.Length) { GRESKA("Neocekivani kraj izraza"); }
  if(Cifra()) { return Broj(); }
```

```
    if(S[I] == '(') {
      I++; PreskociPraznine();
      int n = Izraz();
      if(I < S.Length && S[I] == ')') { I++; PreskociPraznine(); }
      else { GRESKA("Ocekujem )"); }
      return n;
    }
    GRESKA("Neocekivani simbol");
    return 0;
  }
```

Sabirak je podizraz koji učestvuje u građenju izraza povezanih aditivnim operatorima. Pravilo koje opisuje sabirke očekuje činilac, nakon čega može više puta da se javi multiplikativni operator praćen novim činiocem:

$$\langle Sabirak \rangle \equiv \langle Činilac \rangle \ \{ \ \langle OpMnoženja \rangle \ \langle Činilac \rangle \ \}$$

Moramo voditi računa o tome da deljenje nulom nije definisano:

```
  static int Sabirak() {
    int n = Cinilac();
    while(I < S.Length && OpMnozenja()) {
      char op = S[I];
      I++; PreskociPraznine();
      int k = Cinilac();
      switch(op) {
        case '*':
          n *= k;
          break;
        case '/':
          if(k == 0) { GRESKA("Deljenje nulom"); }
          n /= k;
          break;
        case '%':
          if(k == 0) { GRESKA("Deljenje nulom"); }
          n %= k;
          break;
      }
    }
    return n;
  }
```

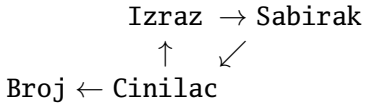
Konačno, izraz počinje sabirkom koji može imati predznak, nakon čega može više puta da se javi aditivni operator praćen novim sabirkom:

$$\langle Izraz \rangle \equiv [\text{“}+\text{”} \mid \text{“}-\text{”}] \ \langle Sabirak \rangle \ \{ \ \langle OpSabiranja \rangle \ \langle Sabirak \rangle \ \}$$

Odgovarajući metod koji vodi računa o svim ovim detaljima izgleda ovako:

```
static int Izraz() {
  if(I == S.Length) { GRESKA("Neocekivani kraj izraza"); }
  bool znak = false;
  if(S[I] == '+' || S[I] == '-') {
    znak = S[I] == '-';
    I++; PreskociPraznine();
  }
  int n = Sabirak();
  if(znak) { n = -n; }
  while(I < S.Length && OpSabiranja()) {
    char op = S[I];
    I++; PreskociPraznine();
    int k = Sabirak();
    if(op == '+') { n += k; }
    else { n -= k; }
  }
  PreskociPraznine();
  return n;
}
```

Ako pogledamo četiri najinteresantnija metoda koji i predstavljaju srce algoritma vidimo da oni zavise jedan od drugog ovako:



Ovo je tipična situacija u kojoj uzajamna rekurzija nudi najelegantniju implementaciju.

Zadaci.

9.1. Napisati rekurzivan C# metod koji računa n-ti element niza brojeva b_n koji je zadat ovako: $b_0 = 0$, $b_1 = 2$, $b_2 = 3$, i $b_n = n^2 b_{n-1} + b_{n-3}^2 - 3n$.

9.2. Napisati rekurzivan C# metod koji računa n-ti element niza brojeva d_n koji je zadat ovako: $d_0 = 0$, $d_1 = 5$, i

$$d_n = \begin{cases} d_{n-1} + 2d_{n-2}, & n \text{ parno} \\ 4d_{n-1} - nd_{n-2}, & n \text{ neparno.} \end{cases}$$

9.3. Napisati C# metod koji računa *NZD* dva pozitivna broja na sledeći način:

$$NZD(m,n) = \begin{cases} m, & m = n \\ NZD(m-n,n), & m > n \\ NZD(n-m,m), & n > m \end{cases}$$

9.4. Napisati rekurzivan metod `void IspisiUOsnovi(int n, int b)` koji nenegativan ceo broj `n` ispisuje u osnovi `b`.

9.5. Za realan broj x i prirodan broj n, broj x^n se može izračunati ovako:

$$x^n = \begin{cases} 1, & n = 0 \\ x, & n = 1 \\ x \cdot (x^k)^2, & n = 2k+1 \\ (x^k)^2, & n = 2k. \end{cases}$$

Napisati C# metod koji rekurzivno računa x^n.

9.6. Napisati C# program koji rekurzivno računa količnik m/n celih brojeva m i n i ispisuje njegovih prvih k decimala.

9.7. Verižni razlomak je razlomak oblika

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{\ddots \\ a_{n-1} + \cfrac{1}{a_n}}}}}$$

gde su a_k neki ne-nula realni brojevi.

(a) Napisati C# program koji od korisnika učitava n, potom $n+1$ realnih brojeva a_0, a_1, …, a_n, i onda računa odgovarajući verižni razlomak *nerekurzivno* (obična "for" petlja).

(b) Napisati C# program koji od korisnika učitava n, potom $n+1$ realnih brojeva a_0, a_1, …, a_n, i onda računa odgovarajući verižni razlomak *rekurzivno*.

9.8. Napisati C# metod koji za dato n ispisuje niz brojeva koji se formira na sledeći način:

$n = 1:$ 1
$n = 2:$ 1, 2, 1
$n = 3:$ 1, 2, 1, 3, 1, 2, 1
$n = 4:$ 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1

Pravilo za formiranje niza izgleda ovako:

$$\boxed{\text{niz za } n} = \boxed{\text{niz za } n-1}, \quad n, \quad \boxed{\text{niz za } n-1}.$$

9.9. Grejov kôd (Gray code) reda n je niz koji se sastoji od svih 01-reči dužine n (kojih ima 2^n) sa osobinom: svake dve susedne reči se razlikuju samo na jednom mestu. Na primer, Grejov kôd reda 4 izgleda ovako:

0000	0110	1100	1010
0001	0111	1101	1011
0011	0101	1111	1001
0010	0100	1110	1000

Napisati C# program koji od korisnika učitava pozitivan ceo broj n i potom štampa Grejov kôd reda n, svaku reč u novom redu.

(Napomena: Naredni element Grejovog koda se formira tako što se u prethodnom elementu komplementira jedan bit. Niz rednih brojeva bitova koje treba komplementirati je upravo niz koji je formiran u prethodnom zadatku.)

9.10. Particija broja n je svaki način da se broj n predstavi u obliku zbira prirodnih brojeva. Na primer, sve particije broja 5 su

$$\begin{aligned} 5 &= 1+1+1+1+1 \\ &= 1+1+1+2 \\ &= 1+2+2 \\ &= 1+1+3 \\ &= 2+3 \\ &= 1+4 \\ &= 5 \end{aligned}$$

Napisati C# program koji za dati prirodan broj n ispisuje broj particija

broja n. Broj particija broja n jednak je $f(n,n)$, gde je

$$f(m,n) = \begin{cases} 1, & m = 1 \text{ ili } n = 1 \\ f(m,m), & m < n \\ f(m,m-1)+1, & m = n \\ f(m,n-1)+f(m-n,n), & m > n. \end{cases}$$

9.11. Napisati C# metod koji rekurzivno računa binomni koeficijent $\binom{n}{k}$ koristeći sledeće:

$$\binom{n}{k} = \begin{cases} 0, & n < k, \\ 1, & n = k \text{ ili } k = 0, \\ n, & k = 1, \\ \frac{n}{k} \cdot \binom{n-1}{k-1}, & \text{inače.} \end{cases}$$

9.12. Napisati C# program koji od korisnika učitava prirodne brojeve n i k, potom k različitih prirodnih brojeva a_1, a_2, ..., a_k, i ispisuje sve načine na koje se dati broj n može predstaviti kao zbir nekih od brojeva a_1, ..., a_k. Svaki broj a_i se može pojaviti u reprezentaciji proizvoljan broj puta.

9.13. Napisati C# program koji od korisnika učitava prirodne brojeve n i M, potom n brojeva c_0, c_1, ..., c_{n-1} i proverava da li se umetanjem znakova $+$ i $-$ ispred ili između nekih od unetih brojeva, ali bez menjanja redosleda unetih brojeva, može dobiti aritmetički izraz čija vrednost je M. Na primer za $n = 4$, $M = 989$ i brojeve 1, 10, 100, 1000 izraz $-1-10+1000$ ima vrednost 989. Primetimo da u ovom primeru nismo upotrebili broj 100.

9.14. Napisati C# program koji od korisnika učitava prirodne brojeve n i M, potom n cifara c_1, ..., c_n i proverava da li se umetanjem znakova $+$ između nekih cifara u nizu može dobiti aritmetički izraz čija vrednost je M. Na primer za $n = 8$, $M = 1000$ i cifre 8 8 8 8 8 8 8 8 (osam osmica), umetanjem znakova $+$ na sledeći način: $888+88+8+8+8$ dobija se izraz čija vrednost je 1000.

9.15. *Kontra-Paskalov trougao* je trougao brojeva koji se formira po sledećim pravilima:

- pođemo od proizvoljnog niza celih brojeva $(a_0, a_1, \ldots, a_{n-1})$;
- svaki naredni niz je za jedan kraći od prethodnog i dobija se ovako: od niza $(a_0, a_1, \ldots, a_{n-1})$ se napravi niz $(a'_0, a'_1, \ldots, a'_{n-2})$ tako što se stavi da je $a'_0 = a_0 + a_1$, $a'_1 = a_1 + a_2$, ..., $a'_{n-2} = a_{n-2} + a_{n-1}$;

- ovaj proces se ponavlja sve dok ne dođemo do niza dužine 1.

Na primer, za niz brojeva 1 2 3 4 5 kontra-Paskalov trougao izgleda ovako:

```
1  2  3  4  5
3  5  7  9
8  12 16
20 28
48
```

Napisati C# program koji od korisnika učitava niz celih brojeva i onda ispisuje njegov kontra-Paskalov trougao.

9.16. Rekamanov niz (B. Recamán) je niz brojeva definisan ovako: $a_1 = 1$, dok za $n \geqslant 2$ stavimo

$$a_n = \begin{cases} a_{n-1} - n, & a_{n-1} - n > 0 \text{ i broj } a_{n-1} - n \text{ se ne javlja ranije u nizu,} \\ a_{n-1} + n, & \text{inače.} \end{cases}$$

Napisati C# program koji od korisnika učitava ceo broj $n \geqslant 1$ i onda računa i štampa a_n.

9.17. *Katalanovi brojevi* (*Catalan numbers*) su brojevi definisani na sledeći način:

$$C_0 = 1$$

$$C_n = \sum_{i=0}^{n-1} C_i C_{n-1-i}.$$

Napisati C# program koji od korisnika učitava ceo broj n i računa n-ti Katalanov broj.

9.18. Napisati C# program koji računa vrednost algebarskog izraza u kome učestvuju realni brojevi. Definicija algebarskog izraza je ista kao gore, s tim da je sada

$$\langle Broj \rangle \equiv [\ \text{“+”} \mid \text{“−”}\ ]\ \langle Cifra \rangle\ \{\ \langle Cifra \rangle\ \}\ [\ \text{“.”}\ \langle Cifra \rangle\ \{\ \langle Cifra \rangle\ \}\ ]$$

9.19. Programski jezik J_0 ima sledeće tri naredbe:

`print` ⟨*Prom*⟩	ispisuje vrednost promenljive ⟨*Prom*⟩
`let` ⟨*Prom*⟩ = ⟨*Izraz*⟩	računa vrednost izraza ⟨*Izraz*⟩ i dodeljuje ga promenljivoj ⟨*Prom*⟩
`stop`	zaustavlja rad programa.

pri čemu je

$$\langle Prom \rangle \equiv \text{“a”} \mid \text{“b”} \mid \ldots \mid \text{“z”}$$

a $\langle Izraz \rangle$ predstavlja algebarski izraz u kome učestvuju realni brojevi i imena promenljivih. To znači da se definicija razlikuje od one o kojoj smo do sada govorili u sledećem:

$$\langle Broj \rangle \equiv [\ \text{“+”} \mid \text{“−”}\ ]\ \langle Cifra \rangle\ \{\ \langle Cifra \rangle\ \}\ [\ \text{“.”}\ \langle Cifra \rangle\ \{\ \langle Cifra \rangle\ \}\ ]$$

$$\langle Činilac \rangle \equiv \langle Broj \rangle \mid \langle Prom \rangle \mid \text{“(”}\ \langle Izraz \rangle\ \text{“)”}$$

Napisati C# program koji od korisnika učitava niz naredbi programskog jezika J_0 i izvršava ih jednu po jednu, sve do pojave naredbe `stop` koja prekida rad programa.

Glava 10

Klase

U modernim programskim jezicima klase se koriste

- kao oblik organizovanja kompleksnog *programskog kôda* u celine koje se lakše pišu i testiraju,
- kao oblik organizovanja kompleksnih *podataka* u logične i funkcionalne celine, i
- kao osnova koja omogućuje da se kôd koji je razvijen i istestiran iskoristi za razvoj novih delova sistema:
  - kroz pružanje standardizovanih usluga u vidu bibliotečkih klasa, i
  - kroz mehanizam nasleđivanja (što je jedna od ključnih ideja objektno-orijentisanog programiranja) koji omogućuje da se klasa koju smo razvili "doradi" i tako prilagodi novim zahtevima.

Klase koje služe kao oblik organizovanja programskog koda smo do sada već videli. U ovoj glavi ćemo se upoznati sa ostalim važnim funkcijama klasa.

10.1 Objekti i klase

U ovom odeljku počinjemo priču o klasama kao obliku organizovanja podataka u logične celine, što predstavlja prvi korak ka implementiranju objektno-orijentisane programske paradigme. Često je veoma korisno da se nekoliko podataka tretira kao jedna celina sa stanovišta programskog jezika. Tako se, recimo, podaci o nekoj osobi kao što je ime, prezime, datum rođenja, adresa, pol i JMBG mogu spakovati u jedan "paket podataka", umesto da ih tretiramo kao šest nezavisnih promenljivih. Do sada smo videli nizove i matrice kao strukture podataka kod

kojih su svi elementi istog tipa. Sada ćemo pokazati kako se klase u C# koriste da se elementi ne nužno istog tipa organizuju u pakete podataka.

Objekat je paket podataka (koji može da ima svoj tajni život, ali o tome nešto kasnije). Možemo ga zamisliti kao jednu skupinu podataka i procedura za manipulaciju tim podacima. Procedure koje su deo objekta se zovu *metodi* (to su metodi za menjanje stanja objekta), dok podaci opisuju *stanje objekta*.

Klasa se može shvatiti kao opis jedne vrste objekata. Objekti koji pripadaju nekoj klasi zovu se *instance* te klase, a klasa opisuje koja polja i koje metode će imati sve njene instance. Evo primera jedne klase koja opisuje podatke o nekoj osobi:

```
class Osoba {
  public string ime, prezime, JMBG;
  public int dan, mesec, godina;

  public Osoba(string i, string p, string J) {
    ime = i; prezime = p; JMBG = J;
    dan = int.Parse(J.Substring(0, 2));
    mesec = int.Parse(J.Substring(2, 2));
    godina = int.Parse(J.Substring(4, 3));
    if(godina > 900) { godina += 1000; }
    else { godina += 2000; }
  }
}
```

Dakle, podaci koje registrujemo za jednu osobu su ime, prezime, JMBG i datum (dan, mesec i godina) rođenja. Uočavamo da su svi elementi u opisu klase `Osoba` označeni kao javni, `public`, kako bi druge klase mogle da im pristupaju. Element koji nije označen kao `public` se vidi samo unutar svoje klase i druge klase (recimo, klasa koja sadrži `Main` metod i koristi usluge ove klase) ih ne vide. Ovu klasu možemo da koristimo unutar neke druge klase, recimo ovako:

```
using System;

class Osoba { ... }

class Primer1 {
  static void Main() {
    Osoba Ana = // ovde nekako popunimo podatke o osobi
    Console.WriteLine("Podaci o osobi: {0} {1}", Ana.ime, Ana.prezime);
    Console.WriteLine("Datum rodjenja: {0}.{1}.{2}.",
                      Ana.dan, Ana.mesec, Ana.godina);
  }
}
```

Vidimo da se klasa `Osoba` ponaša kao novi tip podataka, a da se instanca `Ana` ove klase ponaša kao promenljiva tipa `Osoba`. Poljima koji sadrže podatke o ovoj osobi pristupamo ovako

```
ImeObjekta.ImePolja
```

recimo: `Ana.ime` ili `Ana.godina`. Zato što su sva ova polja označena kao `public` u definiciji klase `Osoba` možemo da im pristupimo iz klase `Primer1`.

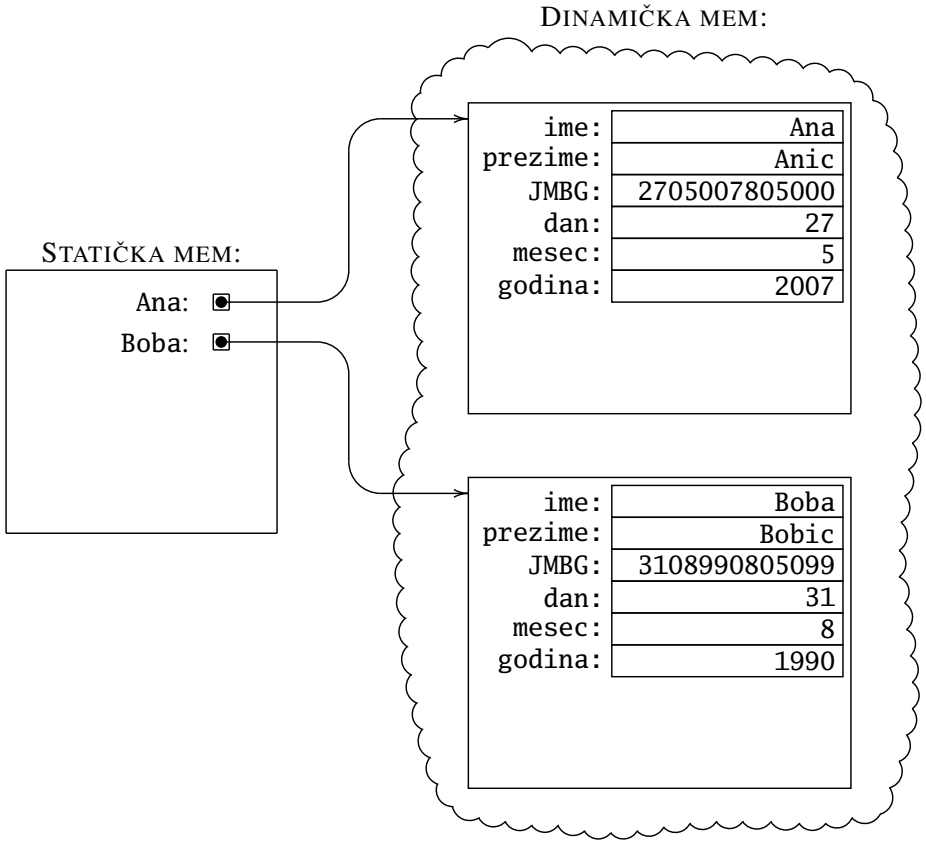
10.2 Konstruktori

Pozabavićemo se sada problemom kako napraviti instancu klase. Programski jezik C# podržava strategiju *objekti su reference* (engl. *reference semantics*). To znači da se u statičkoj memoriji čuva samo *referenca* na objekat, dok se memorijski prostor neophodan da objekat zapamti svoje stanje alocira u dinamičkoj memoriji, kao u slučaju matrica i nizova. Dakle, instance klase su *dinamičke*, nalaze se u dinamičkoj memoriji i za njih moramo eksplicitno alocirati prostor.

Tako dolazimo do pojma konstruktora. *Konstruktor* je metod koji se zove isto kao klasa, poziva se eksplicitno, i u kome se navodi niz instrukcija koje postavljaju stanje objekta na početne vrednosti. Kao što smo već videli, ime klase čiju instancu pravimo se tada koristi kao oznaka tipa promenljive u koju smeštamo referencu na objekat. Konstruktori se pozivaju koristeći ključu reč `new`, kako pokazuje sledeći primer:

```
using System;

class Osoba { ... }

class Primer2 {
  static void Main() {
    Osoba Ana = new Osoba("Ana", "Anic", "2705007805000");
    Osoba Boba = new Osoba("Boba", "Bobic", "3108990805099");
    Console.WriteLine("Podaci o osobi: {0} {1}",
                      Ana.ime, Ana.prezime);
    Console.WriteLine("Datum rodjenja: {0}.{1}.{2}.",
                      Ana.dan, Ana.mesec, Ana.godina);
    Console.WriteLine("Podaci o osobi: {0} {1}",
                      Boba.ime, Boba.prezime);
    Console.WriteLine("Datum rodjenja: {0}.{1}.{2}.",
                      Boba.dan, Boba.mesec, Boba.godina);
  }
}
```

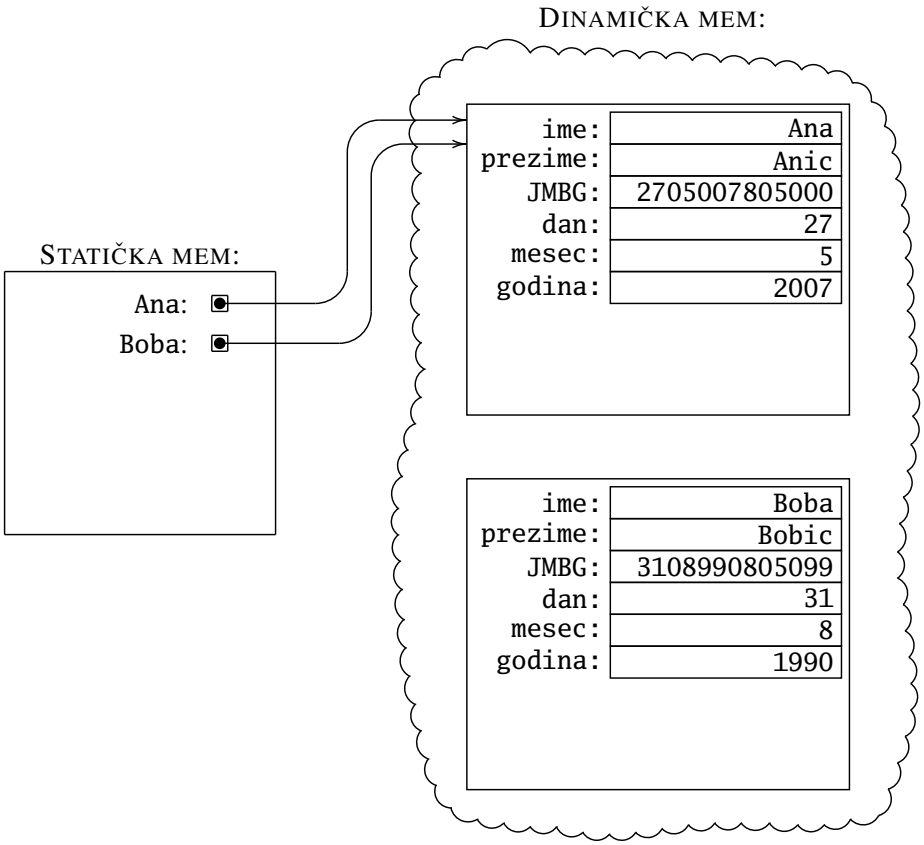
Nakon prve dve naredbe stanje u memoriji je ovakvo:



S obzirom na to da objekti predstavljaju reference na podatke koji se nalaze u dinamičkoj memoriji, važno je da se podsetimo kako se ponaša operacija dodele = i operator poređenja ==. Ako bismo izvršili naredbu

```
Boba = Ana;
```

stanje u memoriji bi izgledalo ovako:

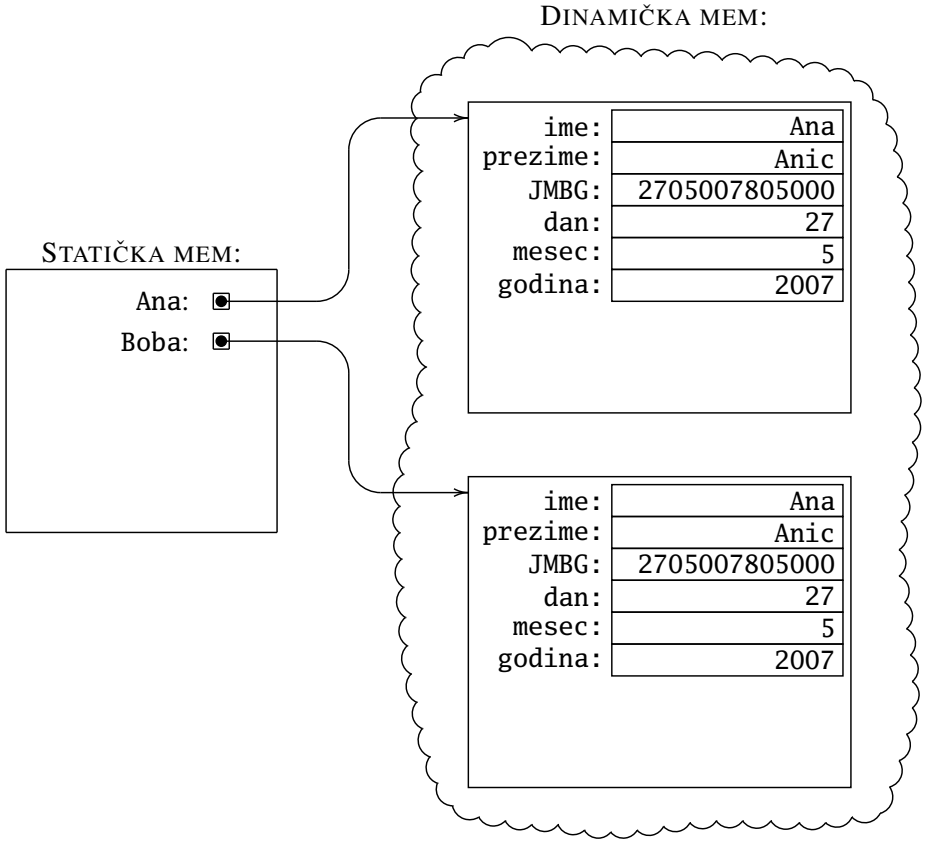


i tada je `Ana == Boba` jer obe promenljive referenciraju isti blok podataka u dinamičkoj memoriji.

S druge strane, nakon naredbi

```
Osoba Ana = new Osoba("Ana", "Anic", "2705007805000");
Osoba Boba = new Osoba("Ana", "Anic", "2705007805000");
```

stanje u memoriji izgleda ovako:



Sada je `Ana != Boba` zato što promenljive referenciraju različite blokove podataka u dinamičkoj memoriji, iako ti blokovi sadrže iste podatke.

10.3 Metodi

Metodi su procedure koje omogućuju pristup poljima objekta ili menjaju njegovo stanje. Pošto su objekti u C# implementirani kao reference, razmatranja sa kraja prethodnog odeljka pokazuju zašto je uvek korisno implementirati metod `Equals` koji proverava da li dva objekta sadrže iste podatke, i metod `Copy` koji pravi kopiju objekta. Klasa `OsobaV2` sadrži podatke o osobi i pored konstruktora (što je metod koji se zove isto kao i klasa) sadrži i ova dva metoda:

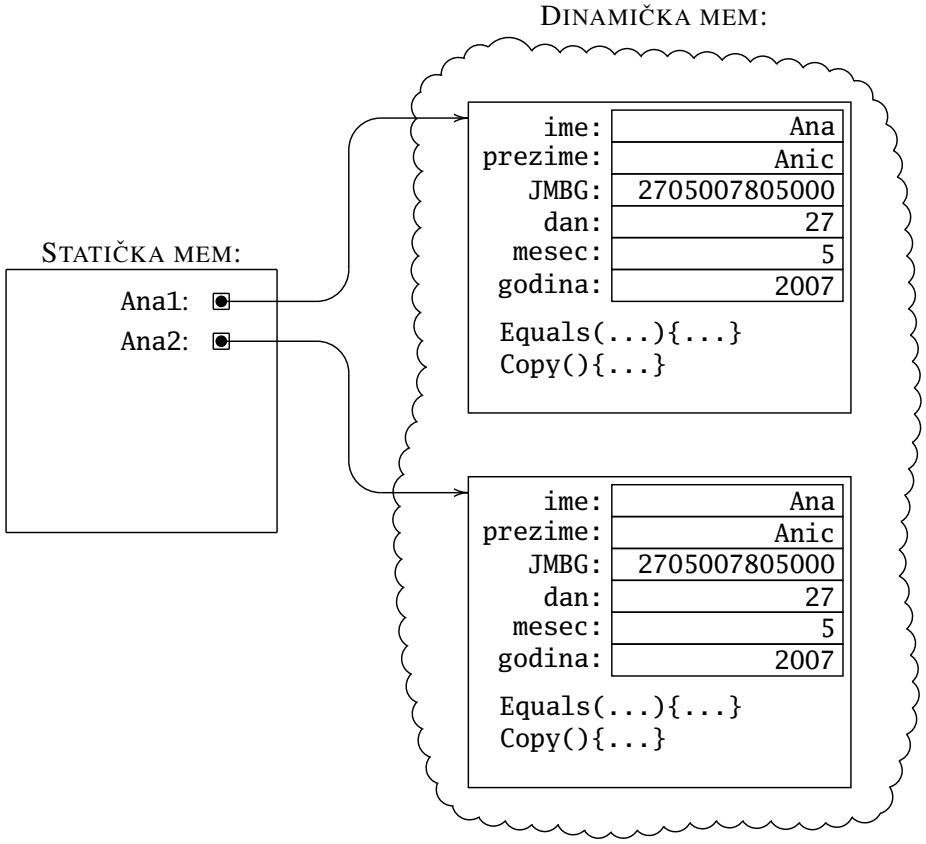
```
class OsobaV2 {
  public string ime, prezime, JMBG;
  public int dan, mesec, godina;

  public OsobaV2(string i, string p, string J) {
    ime = i; prezime = p; JMBG = J;
    dan = int.Parse(J.Substring(0, 2));
    mesec = int.Parse(J.Substring(2, 2));
    godina = int.Parse(J.Substring(4, 3));
    if(godina > 900) { godina += 1000; }
    else { godina += 2000; }
  }

  public bool Equals(OsobaV2 S) {
    return ime == S.ime && prezime == S.prezime
           && JMBG == S.JMBG;
  }

  public OsobaV2 Copy() {
    return new OsobaV2(ime, prezime, JMBG);
  }
}
```

Evo jednostavnog primera:

```
class PrimerSaOsobama {
  static void Main() {
    OsobaV2 Ana1 = new OsobaV2("Ana", "Anic", "2705007805000");
    OsobaV2 Ana2 = Ana1.Copy();
    if(Ana1 == Ana2) {
      Console.WriteLine("Ista referenca");
    }
    else {
      Console.WriteLine("Nije ista referenca");
    }
```

```
    if(Ana1.Equals(Ana2)) {
      Console.WriteLine("Isti sadrzaj");
    }
    else {
      Console.WriteLine("Nije isti sadrzaj");
    }
  }
}
```

Objekat `Ana1` napravimo pozivom konstruktora, dok objekat `Ana2` napravimo kao kopiju objekta `Ana1`. Posle ove dve naredbe stanje u memoriji izgleda ovako:



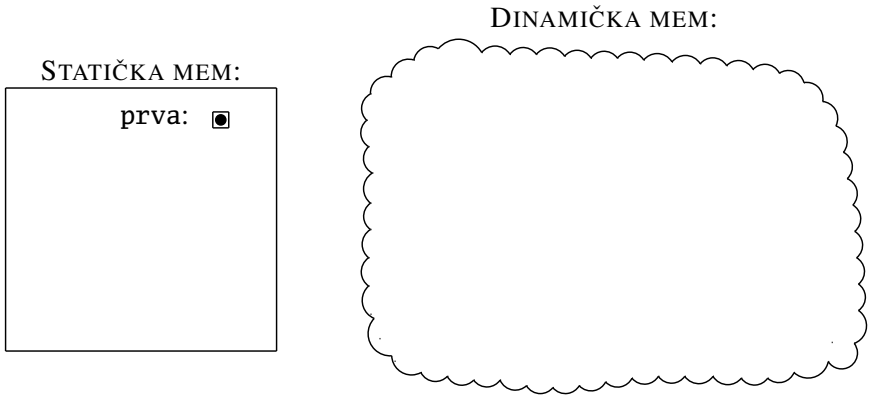
Vidimo da je objekat određen ne samo stanjem svojih polja, već i metodima koji se mogu izvršiti nad tim objektom. Tako naredba `Ana1.Copy()` pozove metod `Copy` nad objektom `Ana1` i napravi kopiju objekta, koju potom smestimo u promenljivu `Ana2`. Jasno je da su reference u promenljivim `Ana1` i `Ana2` različite, pa će program ispisati "`Nije ista referenca`". S druge strane naredba `Ana1.Equals(Ana2)`

će nad objektom `Ana1` izvršiti metod `Equals(Ana2)` koji će stanje objekta `Ana1` uporediti sa stanjem objekta `Ana2`. Kako su sadržaji predstavljeni sa ova dva objekta isti (što nije ni čudo, kada smo napravili kopiju), program će kao drugu poruku ispisati `"Isti sadrzaj"`.

Postoje situacije kada želimo da deklarišemo promenljivu koja će sadržati objekat neke klase, ali ne želimo u tom trenutku da joj dodelimo vrednost. Tada možemo da joj dodelimo vrednost `null`, što označava *praznu referencu*, referencu koja ni na šta ne pokazuje. Na primer, nakon deklaracije

```
Osoba prva = null;
```

stanje u memoriji izgleda ovako:



Možemo proveriti da li je neka referenca "prazna" ili se radi o referenci na neke podatke u dinamičkoj memoriji:

```
if(prva == null) {
  // inicijalizujemo promenljivu
}
else {
  // radimo nesto sa podacima na koje prva pokazuje
}
```

Naravno, kada nam to bude potrebno, promenljivoj `prva` možemo dodeliti neku "konkretnu" referencu na neki od uobičajenih načina, recimo, pozivom konstruktora:

```
prva = new Osoba("Prvan", "Prvanovic", "0101001800001")]
```

Primer. Kada se nakon parlamentarnih izbora utvrdi ko su novi poslanici, saziva se konstitutivna sednica na kojoj se verifikuju mandati novih poslanika i odabere predsednik parlamenta, njegovi zamenici i ostali organi parlamenta koji su predviđeni ustavom. Poštujući stare plemenske običaje, do izbora predsednika parlamenta i njegovih zamenika sednicom rukovodi najstariji poslanik, a pomažu mu dvoje do troje najmlađih. Napisati C# program koji od korisnika učitava ceo broj $n \geqslant 5$, potom podatke o n poslanika koji čine saziv novog parlamenta i određuje najstarijeg i dvoje najmlađih poslanika.

⟨C# fajl⟩

```
using System;

class Osoba {
  public string ime, prezime, JMBG;
  public int dan, mesec, godina; // datum rodjenja

  public Osoba(string i, string p, string j) {
    ime = i; prezime = p; JMBG = j;
    dan = int.Parse(JMBG.Substring(0, 2));
    mesec = int.Parse(JMBG.Substring(2, 2));
    godina = int.Parse(JMBG.Substring(4, 3));
    if(godina > 900) { godina += 1000; }
    else { godina += 2000; }
  }

  public void Ispisi() {
    Console.WriteLine(prezime + ", " + ime + ", " + JMBG);
  }

  public bool MladjaOd(Osoba S) {
    if(godina > S.godina) { return true; }
    if(godina < S.godina) { return false; }
    if(mesec > S.mesec) { return true; }
    if(mesec < S.mesec) { return false; }
    if(dan > S.dan) { return true; }
    return false;
  }
}

class Zadatak {
  static void Main() {
    // ucitavanje osoba
    Console.Write("Unesi broj osoba -> ");
    int n = int.Parse(Console.ReadLine());
    Osoba najstarija = UcitajOsobu(1);
```

```
    Osoba najmladja1 = najstarija;
    Osoba najmladja2 = null;
    for(int i = 1; i < n; i++) {
      Osoba s = UcitajOsobu(i + 1);
      if(najstarija.MladjaOd(s)) { najstarija = s; }

      if(najmladja2 == null) {
        if(najmladja1.MladjaOd(s)) { najmladja2 = s; }
        else {
          najmladja2 = najmladja1;
          najmladja1 = s;
        }
      }
      else if(s.MladjaOd(najmladja1)) {
        najmladja2 = najmladja1;
        najmladja1 = s;
      }
      else if(s.MladjaOd(najmladja2)) { najmladja2 = s; }
    }

    Console.WriteLine("Najstarija osoba:");
    najstarija.Ispisi();
    Console.WriteLine("Dve najmladje osobe:");
    najmladja1.Ispisi();
    najmladja2.Ispisi();
  }

  static Osoba UcitajOsobu(int i) {
    Console.WriteLine("Podaci o osobi {0}", i);
    Console.Write("  Ime -> ");
    string ime = Console.ReadLine();
    Console.Write("  Prezime -> ");
    string prezime = Console.ReadLine();
    Console.Write("  JMBG -> ");
    string jmbg = Console.ReadLine();
    return new Osoba(ime, prezime, jmbg);
  }
}
```

10.4 Statički i dinamički kontekst

Pogledajmo još jednu varijantu klase Osoba:

```
class Osoba {
  //---------------------------------------------
  // staticki kontekst
  public static int TekucaGodina = 2019;
  public static int BrojOsoba = 0;

  public static void Izvestaj() {
    Console.WriteLine("Tekuca godina = {0}", TekucaGodina);
    Console.WriteLine("Broj osoba    = {0}", BrojOsoba);
  }

  //---------------------------------------------
  // dinamicki kontekst
  public string ime, prezime, JMBG;
  public int dan, mesec, godina;

  public Osoba(string i, string p, string J) {
    BrojOsoba++;
    ime = i; prezime = p; JMBG = J;
    dan = int.Parse(J.Substring(0, 2));
    mesec = int.Parse(J.Substring(2, 2));
    godina = int.Parse(J.Substring(4, 3));
    if(godina > 900) { godina += 1000; }
    else { godina += 2000; }
  }

  public bool MladjaOd(Osoba S) {
    if(godina < S.godina) { return true; }
    if(godina > S.godina) { return false; }
    if(mesec < S.mesec) { return true; }
    if(mesec > S.mesec) { return false; }
    if(dan < S.dan) { return true; }
    return false;
  }

  public int Starost() {
    return TekucaGodina - godina;
  }
}
```

Uočava se da ova klasa ima *statički kontekst* i *dinamički kontekst*. Oni ne moraju biti ovako lepo razdvojeni već mogu da se mešaju u kodu.

- Statički kontekst čine sve deklaracije koje su označene kao `static`, dok
- dinamički kontekst čine sve deklaracije koje *nisu* označene kao `static`.

Dakle, u navedenom primeru statički kontekst čine polja `TekucaGodina` i `BrojOsoba` tipa `int`, i metod `Izvestaj`, dok dinamički kontekst čine polja `ime`, `prezime` i `JMBG` tipa `string`, polja `dan`, `mesec` i `godina` tipa `int`, konstruktor `Osoba` i metodi `MladjaOd` i `Starost`.

☞ *Statički kontekst pripada klasi, a dinamički kontekst pripada instancama te klase.*

Pošto statički kontekst "pripada klasi", elementi statičkog konteksta koriste "nezavisno od objekata". Zato se elementima statičkog konteksta pristupa tako što se navede

`ImeKlase.ImeElementa`

na primer ovako:

```
Osoba.TekucaGodina = 2020;
Osoba.Izvestaj();
```

Dinamički kontekst je vezan za instance klase i svaka instanca klase ima svoj dinamički konstekst (zato je *dinamički* – menja se od instance do instance). Elementima dinamičkog konteksta se pristupa tako što se navede

`ImeObjekta.ImeElementa`

Na primer, ako pretpostavimo da su `Ana` i `Boba` instance klase `Osoba`, možemo da napišemo sledeći kôd:

```
if(Ana.MladjaOd(Boba)) {
  Console.WriteLine("Ana ima {0} godina", Ana.Starost());
  Console.WriteLine("i ona je mladja od Bobe");
}
```

Konstrukcija `Ana.Starost()` znači "u kontekstu instance `Ana` aktiviraj metod `Starost()`".

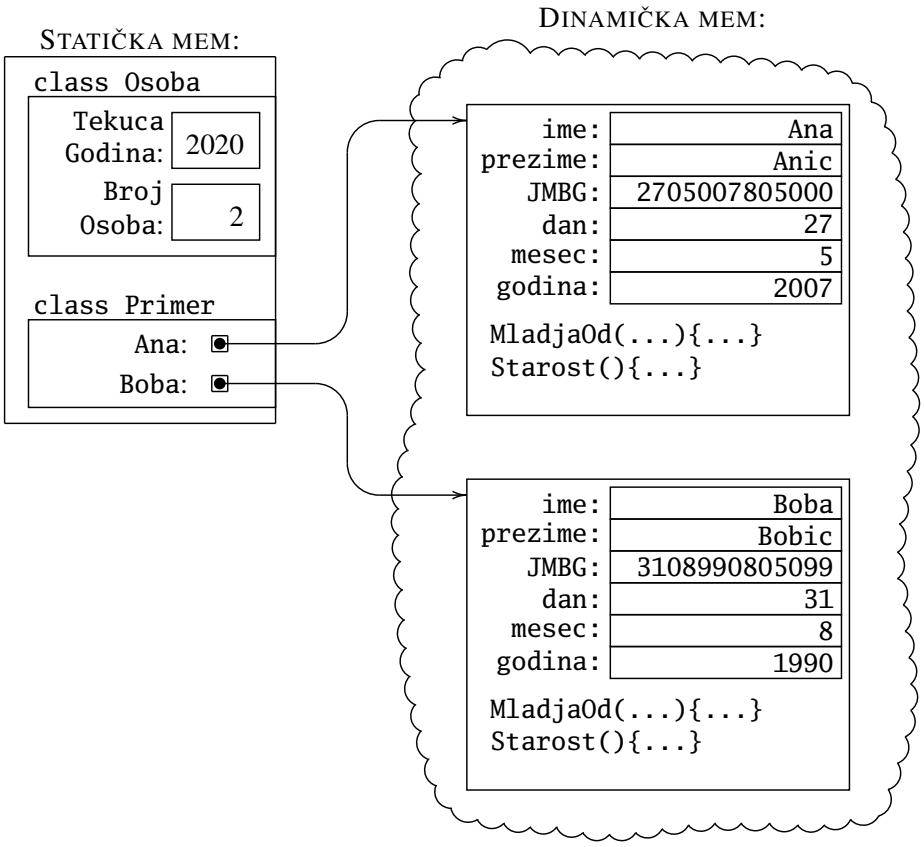
Sva polja dinamičkog konteksta (u našem primeru su to `ime`, `prezime`, `JMBG`, `dan`, `mesec` i `godina`) se sasvim lepo vide u telu metoda `MladjaOd` i `Starost` koji takođe pripadaju dinamičkom kontekstu, a nijedan statički metod klase `Osoba` ih ne vidi. S druge strane, svi elementi statičkog konteksta se vide u svim elementima dinamičkog konteksta. Možemo o tome da razmišljamo i na sledeći način:

☞ *Klasa ne zna šta se dešava unutar njenih instanci jer svaka instanca vodi svoj privatni život, ali zato instance vide sve što se dešava unutar klase jer ona nudi servise svojim instancama.*

Dakle, dinamički kontekst klase određuje koje elemente će imati instance te klase. Na primer, instance klase `Osoba` će imati polja `ime`, `prezime` i `JMBG` tipa `string`, polja `dan`, `mesec` i `godina` tipa `int` (koji sadrže datum rođenja osobe) i metode `MladjaOd` koji proverava da osoba na koju je ovaj metod primenjen mlađa od osobe S koja je prosleđena kao argument metoda, i `Starost` koji računa starost osobe u godinama, u odnosu na tekuću godinu. Ako ovu klasu upotrebimo, recimo, ovako:

```
using System;

class Osoba {
  ...
}

class Primer {
  static void Main() {
    Osoba.TekucaGodina = 2020;
    Osoba Ana = new Osoba("Ana", "Anic", "2705007805000");
    Osoba Boba = new Osoba("Boba", "Bobic", "3108990805099");
    if(Ana.MladjaOd(Boba)) {
      Console.WriteLine("Ana ima {0} godina", Ana.Starost());
      Console.WriteLine("i ona je mladja od Bobe");
    }
    else {
      Console.WriteLine("Boba ima {0} godina", Boba.Starost());
      Console.WriteLine("i ona nije starija od Ane");
    }
  }
}
```

nakon prve tri naredbe stanje u memoriji je sledeće:



i program ispisuje da Ana ima 13 godina i da je mlađa od Bobe.

10.5 Enkapsulacija – učaureni podaci

Kao uvod u priču o enkapsulaciji pogledajmo primer klase `Frac` koja opisuje razlomke (engl. *fraction* = razlomak). Integralna verzija kôda koga u ovom odeljku izlažemo deo po deo uz detaljne komentare dostupna je klikom na link pored. ⟨C# fajl⟩

```
class Frac {
  int a, b; // razlomak je oblika a/b gde je b > 0

  public int Num() { return a; }

  public int Den() { return b; }
```

Ova deklaracija znači da je `Frac` klasa čije instance će imati polja `a` i `b`, što predstavljaju brojilac i imenilac razlomka. Polja `a` i `b` *nisu označena* kao `public`, što znači da se radi o poljima koja su *privatna za klasu*: ta dva polja se vide samo unutar klase `Frac` i ako neka druga klasa koristi klasu `Frac` neće imati pristup ovim poljima. Da bi određena polja klase bila privatna dovoljno je da ne navedemo reč `public`. Međutim, ako želimo da naglasimo da su ta polja privatna, možemo umesto `public` da napišemo `private` (time pokazujemo da nismo omaškom zaboravili da proglasimo ta polja za javna). Tako, ako neka klasa želi da pročita brojilac ili imenilac razlomka moraće da pozove neki od *javnih metoda* `Num()` ili `Den()` (engl. *numerator* = brojilac, *denominator* = imenilac).

Ovo može da deluje čudno: prvo smo sakrili polja `a` i `b` tako što smo ih deklarisali kao privatna (to jest nismo ih deklarisali kao javna), a sada pišemo trivijalne "nepotrebne" i "besmislene" metode `Num()` ili `Den()` koji nam vraćaju njihove vrednosti. Zašto to radimo?

Efikasno programiranje u velikim timovima koji pišu velike projekte se bazira na korišćenju *specifikacije, bez virenja u implementaciju*. Ako dozvolimo programerima iz nekog drugog tima da direktno čačkaju po sadržajima naših objekata mogu da nastanu greške koje se užasno teško nalaze i ispravljaju. Praksa je pokazala da kaogod što je decu nemoguće odvići od čačkanja nosa, tako je programere nemoguće odvići od čačkanja po objektima. Uvek se nađe neki mračni kutak svesti iz koga iskoči ideja:

> *"Zašto da gubim vreme i da ovu trivijalnu stvar radim pozivom metoda* `a.SetLeftChild(p)` *kada je mnogo brže reći*
>
> ```
> a.struct[x].left = p
> ```
>
> *Sada ću ja to časkom, da niko ne vidi!"*

Zato je za pojam objektno-orijentisanog programiranja neraskidivo vezan pojam *sakrivanja informacija* (*information hiding*). Programeru se stavi na raspolaganje ime klase i paket metoda, uz objašnjenje šta koji metod radi (to se zove *specifikacija klase*). Implementacija klase se *sakrije* tako što se svi interni elementi deklarišu kao privatni. Programer iz drugog tima koji koristi ovu klasu ne može da menja stanja objekata direktno, već isključivo pozivanjem metoda koji su u klasi obezbeđeni i koje kontroliše tim koji je odgovoran za razvoj klase. Tako, potencijalno zločesti (ili prosto nesmotreni) korisnik i dalje vidi strukturu objekata u klasi, ali ne može da pristupi direktno poljima objekta. Jedini način da koristi objekte se svodi na pozive metoda koje klasa nudi jer kompajler vrši striktnu kontrolu privatnosti.

Svaki objekat koji drži do sebe mora svoju unutrašnjost držati daleko od prljavih prstiju okolnih korisnika (ili, *klijenata*). Zato onaj deo objekta koji opisuje strukturu promenljivih koje čuvaju stanje objekta ne sme biti dostupan klijentima.

Ako neko želi da proveri i/ili da promeni stanje objekta, mora to učiniti pozivanjem odgovarajućeg metoda. Na taj način objekat ima potpunu kontrolu nad samim sobom i može da štiti svoj integritet. To je u skladu sa opštom idejom o skrivanju podataka (*information hiding*) i zove se još i *encapsulation* (učaurenost). Podaci su učaureni u objektu i do njih se ne može doći direktno.

U skladu sa preporukama o sakrivanju informacija, polja `a` i `b` su deklarisana kao privatna i njima mogu da pristupe samo metodi ove klase. Metodi `Num()` i `Den()` su javno dostupni. Pomoću njih se ne može menjati stanje objekta, već se može samo pročitati imenilac i brojilac razlomka. Na primer u sledećem programu:

```
using System;

class Frac { ... }

class Primer1 {
  static void Main() {
    Frac R = new Frac(5, 9);
    Console.WriteLine(R.Num()); // ispisuje brojilac razlomka R
    Console.WriteLine(R.a); // GRESKA! Polje a je privatno
                            // i ne vidi se izvan klase Frac!
  }
}
```

prevodilac prijavljuje grešku u poslednjoj `WriteLine` naredbi. Da je polje `a` bilo deklarisano kao `public` onda bismo iz programa `Primer1` mogli i njemu da pristupimo i da mu čitamo ili menjamo vrednost. No, to nije moguće pošto polje nije deklarisano kao javno dostupno.

Klasa `Frac` ima dva konstruktora. Prvi konstruktor ima samo jedan argument `n` i on od celog broja n pravi razlomak $n/1$:

```
public Frac(int n) {
  a = n; b = 1;
}
```

Drugi konstruktor ima dva celobrojna argumenta `p` i `q` koji predstavljaju brojilac i imenilac razlomka. Pri tome vodi računa da q ne sme biti 0, i da smo se na početku dogovorili da uvek držimo $q > 0$:

```
public Frac(int p, int q) {
  if(q == 0) { throw new Exception("Imenilac razlomka je 0"); }
  if(q < 0) { p = -p; q = -q; }
  int d = GCD(Math.Abs(p), q);
  a = p / d; b = q / d;
}
```

```
static int GCD(int a, int b) {
  // znamo da je a >= 0 i b >= 0
  if(a == 0) { return b; }
  if(b == 0) { return a; }
  int r;
  do {
    r = a % b; a = b; b = r;
  } while(r > 0);
  return a;
}
```

Evo nekoliko ključnih novih koncepata koje ovaj primer uvodi:

(1) Ako je $q = 0$ konstruktor generiše izuzetak naredbom

```
throw new Exception...
```

Time se prekida izvršavanje metoda i *run-time* sistem kreće u potragu za metodom koji ume da se snađe sa ovakvom situacijom. Ako se takav metod ne pronađe, *run-time* sistem prijavljuje grešku korisniku.

(2) Ako je $q < 0$, konstruktor promeni znak i imeniocu i brojiocu. Osim toga, konstruktor još i skrati razlomak. Za skraćivanje razlomka se koristi metod `GCD` koji je statički metod. To znači da je on vezan za klasu i poziva se iz statičkog konteksta, dakle, kako smo i navikli u dosadašnjem radu.

(3) Metod `GCD` *nije* deklarisan kao javni. To znači da se on vidi samo unutar klase `Frac` i druge klase koje koriste klasu `Frac` ga ne vide.

Razlomci predstavljaju vrstu brojeva. Želimo zato da ih učitavamo, ispisujemo i da sa njima računamo na način na koji to radimo i sa drugim brojevima. Učitavanje je lako – dovoljno je da napišemo statički metod `Parse` koji će string konvertovati u razlomak. Razlomak treba uneti u obliku

```
"brojilacSaZnakom/imenilac"
```

na primer ovako: `-25/17`.

```
public static Frac Parse(string s) {
  string t = "";
  foreach(char c in s) {
    if(!Char.IsWhiteSpace(c)) { t += c; }
  }
```

```
    if(String.IsNullOrEmpty(t)) {
      throw new Exception("Nije dobar zapis razlomka");
    }
    int k = t.IndexOf('/');
    if(k == -1) { return new Frac(int.Parse(t)); }
    return new Frac(int.Parse(t.Substring(0, k)),
                    int.Parse(t.Substring(k+1)));
  }
```

Sada imamo nekoliko načina da formiramo razlomak – pozivom jednog od konstruktora ili učitavanjem od korisnika, kao u sledećem malom primeru:

```
using System;

class Frac { ... }

class Primer2 {
  static void Main() {
    Frac R = new Frac(5, 9); // razlomak 5/9
    Frac S = new Frac(7); // ceo broj 7 kao razlomak
    Frac T = Frac.Parse(Console.ReadLine());
    // razlomak ucitan od korisnika
  }
}
```

Postoji mogućnost i da razlomak ispišemo pozivom naredbe `WriteLine` kao što smo ispisivali cele i realne brojeve. Zapravo, C# sistem ima način da ispiše bilo koji objekat koji se prosledi naredbi `WriteLine` tako što pozove metod `ToString`. U jeziku C# je standard da se u svakoj klasi koju definišemo *implicitno* (dakle, prikriveno, bez *eksplicitnog* znanja korisnika) definiše i metod `ToString` koji dati objekat na neki način konvertuje u string. Mi ćemo zato morati da *redefinišemo* metod `ToString` tako što ćemo ga označiti ključnom rečju `override`:

```
public override string ToString() {
  if(a == 0) { return "0"; }
  if(b == 1) { return a.ToString(); }
  return a.ToString() + "/" + b.ToString();
}
```

Ključna reč `override` znači: “Znam da je C# sistem već implementirao ovaj metod, ali ja želim da napišem svoj koji će se pozivati *umesto* sistemskog.” Zato sada možemo da napišemo ovakav kôd:

```
using System;

class Frac { ... }

class Primer3 {
  static void Main() {
    Frac r = Frac.Parse(Console.ReadLine());
    Console.WriteLine("Uneo si razlomak {0}!", r);
  }
}
```

Slede tri metoda čija uloga je očigledna:

```
public Frac Copy() { return new Frac(a, b); }

public bool Equals(Frac r) {
  return a == r.a && b == r.b;
}

public int Compare(Frac r) {
  if(Equals(r)) { return 0; }
  if(a * r.b < r.a * b) { return -1; }
  return 1;
}
```

Metod `Copy` pravi kopiju objekta, metod `Equals` proverava da li su dve instance klase `Frac` jednake *po sadržaju*, dok metod `Compare` poredi dva razlomka i vraća ceo broj ovako:

- `s.Compare(r)` vraća 0 ako su razlomci `s` i `r` jednaki po sadržaju;
- `s.Compare(r)` vraća -1 ako je razlomak `s` manji od razlomka `r`;
- `s.Compare(r)` vraća 1 ako je razlomak `s` veći od razlomka `r`.

U implementaciji metoda `Comapre` koji poredi dva razlomka proveravali smo da li važi da je $\frac{a_1}{b_1} < \frac{a_2}{b_2}$ tako što smo proverili da li je $a_1 \cdot b_2 < a_2 \cdot b_1$. Dakle, korstili smo da je

$$\frac{a_1}{b_1} < \frac{a_2}{b_2} \text{ ako i samo ako } a_1 \cdot b_2 < a_2 \cdot b_1,$$

što je tačno *samo pod uslovom da je* $b_1 > 0$ *i* $b_2 > 0$*!*

Dakle, zahtev da imenilac razlomka uvek mora biti pozitivan koga smo postavili na samom početku implementacije klase `Frac` je *od presudne važnosti* za korektnu implementaciju klase. Zato *ne smemo da dozvolimo da korisnik postavlja imenilac i brojilac razlomka kako želi*, odnosno, ne smemo dozvoliti da imenilac i

brojilac razlomka budu javna polja. Ova dva podatka su enkapsulirana u klasi tako što su odgovarajuća polja privatna, i jedini način da im se pristupi iz neke druge klase je kroz metode, koji će uz put proveriti da li su uslovi koje smo postavili i ispunjeni.

Konačno, napisaćemo i niz metoda koji nam omogućuju da pišemo algebarske izraze sa razlomcima kao što to radimo sa celim i realnim brojevima. Za to je potrebno da definišemo niz varijanti standardnih operatora +, −, ∗ i / kako sledi. Prvo ćemo definisati varijantu koja sabira dva razlomka:

```
public static Frac operator + (Frac r, Frac s) {
  return new Frac(r.Num()*s.Den() + s.Num()*r.Den(), r.Den()*s.Den());
}
```

Potom definišemo zbir celog broja i razlomka tako što ceo broj konvertujemo u razlomak i pozovemo prethodni operator:

```
public static Frac operator + (Frac r, int s) {
  return r + new Frac(s);
}

public static Frac operator + (int r, Frac s) {
  return new Frac(r) + s;
}
```

Sada po analogiji definišemo varijante binarnog operatora −:

```
public static Frac operator - (Frac r, Frac s) {
  return new Frac(r.Num()*s.Den() - s.Num()*r.Den(), r.Den()*s.Den());
}

public static Frac operator - (Frac r, int s) {
  return r - new Frac(s);
}

public static Frac operator - (int r, Frac s) {
  return new Frac(r) - s;
}
```

kao i unarni operator promene znaka:

```
public static Frac operator - (Frac r) {
  return new Frac(-r.Num(), r.Den());
}
```

Slede varijante operatora * i /:

```
public static Frac operator * (Frac r, Frac s) {
  return new Frac(r.Num()*s.Num(), r.Den()*s.Den());
}

public static Frac operator * (Frac r, int s) {
  return r * new Frac(s);
}

public static Frac operator * (int r, Frac s) {
  return new Frac(r) * s;
}

public static Frac operator / (Frac r, Frac s) {
  return new Frac(r.Num()*s.Den(), r.Den()*s.Num());
}

public static Frac operator / (Frac r, int s) {
  return r / new Frac(s);
}

public static Frac operator / (int r, Frac s) {
  return new Frac(r) / s;
}
```

Klasu završavamo implicitnim operatorom konverzije celog broja u razlomak koji nam omogućuje da u istom izrazu mešamo cele brojeve i razlomke:

```
public static implicit operator Frac(int n) {
   return new Frac(n);
}
```

Primer. Napisati C# program koji od korisnika učitava n, potom n razlomaka r_1, $r_2, \ldots, r_n$ i potom računa i štampa vrednost izraza

$$\sum_{k=1}^{n} \frac{(-1)^k}{k} \cdot r_k.$$

⟨C# fajl⟩

```
using System;
using static Frac;

class PrimerSaRazlomcima {
```

```
    static void Main() {
      int n = int.Parse(Console.ReadLine());
      Frac sum = 0; // implicitna konverzija int -> Frac
      int znak = 1;
      for(int k = 1; k <= n; k++) {
        znak = -znak;
        Frac r = Frac.Parse(Console.ReadLine());
        sum += r * new Frac(znak, k);
      }
      Console.WriteLine("Rezultat je {0}", sum);
    }
  }
```

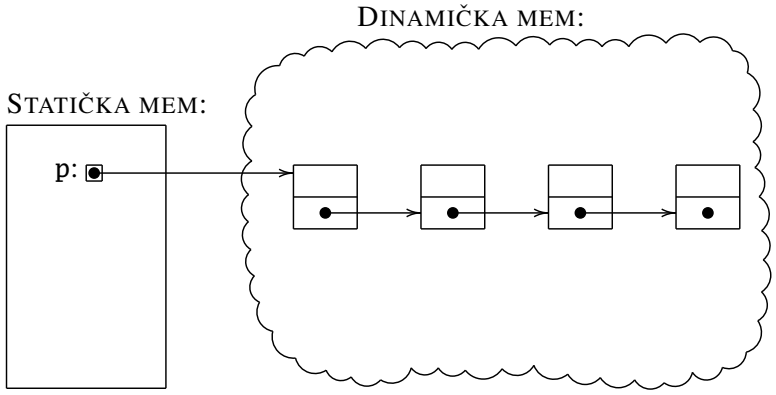
10.6 Bibliotečke klase

Klasa `Frac` je veoma dugačka i bilo bi besmisleno da moramo da je prekucavamo svaki put kada želimo da radimo sa razlomcima. Srećom, klase možemo da koristimo kao posebne kompilacione jedinice, odnosno, kao biblioteke korisnih metoda koje se kasnije mogu koristiti u drugim klasama. Time što korisne metode grupišemo u biblioteke štedimo vreme prilikom pisanja programa, ali i vreme potrebno za testiranje programa: biblioteka se jednom detaljno istestira i onda znamo da su metodi koje ona nudi drugim klasama pouzdani. Prema tome, ukoliko se u programu koji se sastoji iz više klasa pojavi greška, znamo gde tu grešku nećemo da tražimo (bar dok ne proverimo sumnjivija mesta).

Kako se u zavisnosti od uloge koju klasa ima kompajler mora pozivati na različite načine, u ovom odeljku ćemo se osvrnuti i na nekoliko tehničkih pitanja u vezi sa strukturom C# kompajlera. Konkretno, pokazaćemo kako se koriste opcije prilikom poziva kompajlera i koje opcije treba uključiti da bi kompajler preveo kôd na odgovarajući način. Okruženja poput Visual Studija sakrivaju od korisnika direktno upravljanje radom kompajlera. Mi ćemo, međutim, poći od pretpostavke da korisnik poziva kompajler iz komandne linije, i zato ćemo na početku svakog listinga u prvom redu navesti komentar koji sadrži opcije za prevođenje tog koda. Primer upotrebe bibliotečkih klasa pokazaćemo na primeru klase koja implementira celobrojne liste.

Strukture podataka kao što su nizovi se zovu još i *statičke* strukture podataka zato što se broj elemenata takve strukture ne može menjati tokom izvršavanja programa. Klase i pokazivačka semantika nam omogućuju da se upoznamo sa jednom novom vrstom struktura podataka koje zovemo *dinamičke* strukture podataka. Dinamičke strukture podataka mogu da menjaju broj svojih elemenata tokom izvršenja programa. Liste predstavljaju najjednostavniju dinamičku strukturu podataka.

Lista je niz kućica u dinamičkoj memoriji koje pokazuju svaka na onu iza sebe kao na sledećoj slici:



Promenljiva u statičkoj memoriji pokazuje na prvi element liste koji se zove *glava liste*, a svaki element liste potom pokazuje na onaj iza njega. Poslednji element liste ne pokazuje ni na šta i zato je u odgovarajuće polje upisana prazna referenca `null`. Na primer, lista celih brojeva može da se opiše ovako:

```
class ListInt {
  public int info;
  public ListInt next;

  public ListInt(int n, ListInt nxt) {
    info = n; next = nxt;
  }

  // jos neki staticki metodi
}
```

Polje `info` sadrži koristan podatak koji je u ovom slučaju neki ceo broj, dok polje `next` sadrži referencu na narednu kućicu u listi, ili `null` ukoliko se radi o poslednjoj kućici u listi. Listu čiji elementi su 2, 7 i 9 možemo napraviti, recimo, ovako:

```
ListInt a = new ListInt(2, new ListInt(7, new ListInt(9, null)));
```

Statički metod koji ispisuje sve elemente celobrojne liste izgleda ovako:

```
public static int Length(ListInt p) {
  int br = 0;
  while(p != null) {
    br++;
    p = p.next;
  }
  return br;
}
```

On radi tako što inicijalizuje brojač na 0, pa sve dok referenca p pokazuje na neku kućicu u listi uveća brojač za jedan i onda se naredbom

```
p = p.next
```

premesti na sledeću kućicu u listi.

Evo sada cele bibliotečke klase ListInt koja se nalazi u datoteci Liste.cs: (Primetimo da ime fajla – Liste.cs – i ime klase – ListInt – ne moraju biti u vezi; štaviše, jedan fajl može sadržati više klasa.) Metod Length računa dužinu liste, metod Equal proverava da li su dve liste jednake po sadržaju, metod Copy pravi kopiju liste, dok metod Rev obrće listu tako što preuredi reference u listi koja mu je prosleđena.

⟨C# fajl⟩

```
//compiler options: /t:library
using System;

public class ListInt {
  public int info;
  public ListInt next;

  public ListInt(int n, ListInt nxt) {
    info = n;
    next = nxt;
  }

  public static int Length(ListInt p) {
    int br = 0;
    while(p != null) {
      br++;
      p = p.next;
    }
    return br;
  }

  public static bool Equal(ListInt p, ListInt q) {
    while(p != null && q != null) {
      if(p.info != q.info) { break; }
      p = p.next;
      q = q.next;
    }
    return p == null && q == null;
  }

  public static ListInt Copy(ListInt p) {
    if(p == null) { return null; }
```

```
    ListInt first = new ListInt(p.info, null);
    ListInt last = first;
    p = p.next;
    while(p != null) {
      last.next = new ListInt(p.info, null);
      last = last.next;
      p = p.next;
    }
    return first;
  }

  public static ListInt Rev(ListInt p) {
    if(p == null || p.next == null) { return p; }
    ListInt q = p.next;
    p.next = null;
    ListInt r;
    do {
      r = p;
      p = q;
      q = q.next;
      p.next = r;
    } while(q != null);
    return p;
  }
}
```

Bibliotečke klase se razlikuju od klasa koje smo do sada videli na više načina.

(1) Bibliotečke klase nemaju `Main` metod zato što one ne predstavljaju program koji treba da se izvršava nezavisno. One predstavljaju komponente koje će biti uključene u druge klase.

(2) Cela klasa je deklarisana kao `public` što znači da *svaka druga klasa* može da iskoristi njene usluge.

(3) Kao i ranije, svaki metod čije usluge želimo da ponudimo okruženju mora biti deklarisan kao `public`. Ako neki metod nije deklarisan kao `public`, njega mogu da pozivaju drugi metodi iz iste klase, ali on ostaje nevidljiv za druge klase.

Prilikom prevođenja bibliotečke klase ne generiše se program pošto klasa nema `Main` metod. Zato je potrebno pozvati prevodilac sa odgovarajućim dodatnim parametrom, recimo ovako:

```
csc /t:library Liste.cs
```

a prevodilac će generisati fajl `Liste.dll` (pod operativnim sistemom Windows). Opcija `/t:library` znači: *target = library*.

☞ *Prilikom prevođenja navodi se ime fajla, a ne ime klase.*

Primer. Napisati C# program koji od korisnika učitava niz brojeva čija dužina nije unapred poznata i onda proverava da li je to palindrom. Korisnik završava unos niza brojeva tako što unese reč `kraj`.

Pošto ne znamo unapred maksimalnu dužinu niza, nećemo u rešenju koristiti nizove celih brojeva već liste. Rešenje koje ćemo pokazati nije najefikasnije, ali demonstrira upotrebu statičkih metoda klase `ListInt`.

⟨C# fajl⟩

```
//compiler options: /t:exe /reference:Liste.dll
using System;
using static ListInt;

class Palindrom {
  static void Main() {
    ListInt p = null;
    while(true) {
      string s = Console.ReadLine();
      if(s == "kraj") { break; }
      int n = int.Parse(s);
      p = new ListInt(n, p);
    }

    ListInt r = ListInt.Rev(ListInt.Copy(p));
    if(ListInt.Equal(p, r)) {
      Console.WriteLine("Lista je palindrom");
    }
    else {
      Console.WriteLine("Lista nije palindrom");
    }
  }
}
```

Vidimo da se prilikom uključivanja bibliotečke klase koristi direktiva `using static`. Kako se klasa `ListInt` nalazi u datoteci `Liste.cs`, prilikom prevođenja primera potrebno je prevodilac pozvati ovako:

```
csc /t:exe /reference:Liste.dll ListePrimer.cs
```

Opcija `/reference:` sadrži listu biblioteka koje sadrže nestandardne klase koje se u programu koriste, i tako prevodilac zna gde sve treba da pogleda kako bi našao klasu `ListInt`. Opcija `/t:exe` znači: *target = executable*. Ova komanda će prevesti klasu `ListePrimer.cs`, tokom prevođenja će pronaći klasu `ListInt` u biblioteci `Liste.dll` i uključiti je u kôd, i na kraju generisati izvršni kôd `ListePrimer.exe`.

10.7 Generičke klase

Do sada smo radili sa listama celih brojeva, ali se može desiti da nam za rešavanje nekog problema trebaju, recimo, liste karaktera.

Primer. Napisati C# program koji od korisnika učitava string i proverava da li su u njemu parovi zagrada `( i )`, `[ i ]`, `{ i }`, `i < i >` izbalansirani, što znači da

- svaka otvorena zagrada ima svoju zatvorenu zagradu "drugaricu", i
- parovi zagrada različitog tipa ne smeju da se "prepliću".

Na primer, u sledećim stringovima zagrade nisu izbalansirane:

String	Zagrade nisu izbalansirane jer...
`<(a)c>[{}(]x`	zagrada `(` nema svoju zatvorenu zagradu "drugaricu";
`<()>b{v}([]n])`	zagrada `]` nema svoju otvorenu zagradu "drugaricu";
`g((h)[)]a`	zagrade različitih tipova se "prepliću".

Rešenje. Problem se najjednostavnije može rešiti tako što se formira lista karaktera koja vodi računa o otvorenim zagradama. Nakon toga prođemo redom kroz sve karaktere u stringu i ponašamo se ovako:

- karaktere koji nisu zagrade ignorišemo;
- ako karakter predstavlja otvorenu zagradu, stavimo ga na početak liste;
- ako karakter predstavlja zatvorenu zagradu onda:

  ako je lista prazna ili prvi element liste nije odgovarajuća otvorena zagrada prekinemo sa izvršavanjem i prijavljujemo da zagrade nisu izbalansirane; u suprotnom uklonimo prvi element liste i nastavljamo dalje.

Na kraju, ako je lista prazna prijavimo da su zagrade izbalansirane, a ako je lista neprazna prijavimo da zagrade nisu izbalansirane.

```
using System;

class ListChar {
  public char info;
  public ListChar next;

  public ListChar(char c, ListChar nxt) {
    info = c;
    next = nxt;
  }
  ...
} // class ListChar

class Primer {
  static void Main() {
    string s = Console.ReadLine();
    if(Balans(s)) { Console.WriteLine("Izbalansirane"); }
    else { Console.WriteLine("Nisu izbalansirane"); }
  }

  static bool OtvorenaZag(char c) {
    return c == '(' || c == '[' || c == '{' || c == '<';
  }

  static bool ZatvorenaZag(char c) {
    return c == ')' || c == ']' || c == '}' || c == '>';
  }

  static bool Odgovaraju(char c, char d) {
    return c == '(' && d == ')' || c == '[' && d == ']' ||
           c == '{' && d == '}' || c == '<' && d == '>';
  }

  static bool Balans(string s) {
    ListChar p = null;
    foreach(char c in s) {
      if(OtvorenaZag(c)) {
        p = new ListChar(c, p);
      }
      else if(ZatvorenaZag(c)) {
        if(p == null || !Odgovaraju(p.info, c)) { return false; }
        p = p.next;
      }
    }
    return p == null;
```

```
  }
} // class Primer
```

Nije teško zaključiti da klasa `ListChar` jako liči na klasu `ListInt` i da su tipične operacije sa listama karaktera potpuno analogne odgovarajućim operacijama nad listama celih brojeva, kako je pokazano na Sl. 10.1 i Sl. 10.2. Ako nam za rešavanje nekog problema zatreba, recimo, lista stringova, onda bismo ponovo morali da pišemo istu klasu koja se od ove dve razlikuje samo po tome što na nekoliko ključnih mesta treba umesto tipa `int` staviti tip `string`.

Logično je, zato, da u svim savremenim programskim jezicima postoji mehanizam koji nam omogućuje da napišemo jednu *generičku* klasu koja radi za neki *generički tip* `T` i da potom u programima koje pišemo pozivamo tu generičku klasu za ovaj tip podataka ili za onaj tip podataka. Mehanizam koji je usvojen u C# da bi se ovaj problem rešio zasniva se na *parametrizovanim klasama*, što su klase koje imaju parametar umesto koga se u glavnom programu može staviti bilo koji tip. Parametrizovana klasa `List<T>` data je na Sl. 10.3. Iz prvog reda deklaracije

```
class List<T> {
```

vidimo da se argument parametrizovane klase piše u uglastim zagradama i *nema deklaraciju tipa* jer znamo da to može biti samo neki tip. Svuda u kodu se potom umesto `ListInt` i `ListChar` javlja `List<T>`. Jedina razlika u odnosu na klase `ListInt` i `ListChar` je u metodu `Equal`: kod klasa `ListInt` i `ListChar` jednakost celih brojeva, odnosno karaktera, proveravamo operatorom ==, dok u klasi `List<T>` moramo da zovemo sistemski operator `Object.Equals(...)` jer prilikom kompajliranja parametrizovane klase kompajler zapravo ne zna koji tip će se u primenama klase `List<T>` javiti umesto parametra `T`. Metod `Balans` iz prethodnog primera se sada može napisati ovako:

```
static bool Balans(string s) {
  List<char> p = null;
  foreach(char c in s) {
    if(OtvorenaZag(c)) {
      p = new List<char>(c, p);
    }
    else if(ZatvorenaZag(c)) {
      if(p == null || !Odgovaraju(p.info, c)) { return false; }
      p = p.next;
    }
  }
  return p == null;
}
```

```
public class ListInt {
  public int  info;
  public ListInt next;

  public ListInt(int n, ListInt nxt) {
    info = n;
    next = nxt;
  }

  public static int Length(ListInt p) {
    int br = 0;
    while(p != null) {
      br++;
      p = p.next;
    }
    return br;
  }

  public static bool Equal(ListInt p, ListInt q) {
    while(p != null && q != null) {
      if(p.info != q.info) { break; }
      p = p.next;
      q = q.next;
    }
    return p == null && q == null;
  }

  public static ListInt Copy(ListInt p) {
    if(p == null) { return null; }
    ListInt first = new ListInt(p.info, null);
    ListInt last = first;
    p = p.next;
    while(p != null) {
      last.next = new ListInt(p.info, null);
      last = last.next;
      p = p.next;
    }
    return first;
  }
 ...
}
```

Slika 10.1: Deo klase `ListInt`

```
public class ListChar {
  public char info;
  public ListChar next;

  public ListChar(char c, ListChar nxt) {
    info = c;
    next = nxt;
  }

  public static int Length(ListChar p) {
    int br = 0;
    while(p != null) {
      br++;
      p = p.next;
    }
    return br;
  }

  public static bool Equal(ListChar p, ListChar q) {
    while(p != null && q != null) {
      if(p.info != q.info) { break; }
      p = p.next;
      q = q.next;
    }
    return p == null && q == null;
  }

  public static ListChar Copy(ListChar p) {
    if(p == null) { return null; }
    ListChar first = new ListChar(p.info, null);
    ListChar last = first;
    p = p.next;
    while(p != null) {
      last.next = new ListChar(p.info, null);
      last = last.next;
      p = p.next;
    }
    return first;
  }
 ...
}
```

Slika 10.2: Deo klase ListChar

```
public class List<T> {
  public T  info;
  public List<T> next;

  public List(T n, List<T> nxt) {
    info = n;
    next = nxt;
  }

  public static int Length(List<T> p) {
    int br = 0;
    while(p != null) {
      br++;
      p = p.next;
    }
    return br;
  }

  public static bool Equal(List<T> p, List<T> q) {
    while(p != null && q != null) {
      if(!Object.Equals(p.info, q.info)) { break; }
      p = p.next;
      q = q.next;
    }
    return p == null && q == null;
  }

  public static List<T> Copy(List<T> p) {
    if(p == null) { return null; }
    List<T> first = new List<T>(p.info, null);
    List<T> last = first;
    p = p.next;
    while(p != null) {
      last.next = new List<T>(p.info, null);
      last = last.next;
      p = p.next;
    }
    return first;
  }
 ...
}
```

Slika 10.3: Deo generičke klase `List` čiji parametar `T` predstavlja neki tip

Primer. Koristeći činjenicu da u klasi `List<T>` koja se nalazi u fajlu `GenListe.cs` (do koga se može doći klikom na link koji se nalazi pored ovog pasusa) postoje metodi `Copy` koji pravi kopiju liste, `Equal` koji proverava da li su dve liste jednake i `Rev` koji okreće listu, napisati C# program koji od korisnika učitava niz celih brojeva i proverava da li je on palindrom.

⟨C# fajl⟩

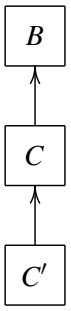
⟨C# fajl⟩

```
//compiler options: /reference:GenListe.dll
using System;
using static List<int>;

class Palindrom {
  static void Main() {
    Console.WriteLine("Unesi niz celih brojeva; STOP za kraj");
    List<int> p = null;
    while(true) {
      string s = Console.ReadLine();
      if(s != "STOP") { break; }
      p = new List<int>(int.Parse(s), p);
    }
    List<int> r = List<int>.Rev(List<int>.Copy(p));
    if(List<int>.Equal(p, r)) {
      Console.WriteLine("Jeste palindrom");
    }
    else {
      Console.WriteLine("Nije palindrom");
    }
  }
}
```

10.8 Objektno-orijentisano programiranje

Osnovna odlika objektno-orijentisanih sistema je nasleđivanje. *Nasleđivanje* je relacija među klasama koja omogućuje da se definicija i implementacija jedne klase bazira na definiciji i implementaciji neke već postojeće klase. Na taj način se štedi vreme, novac, ali i memorijski prostor.

Prilikom nasleđivanja, *potklasa* nasleđuje i metode i strukturu objekata od svoje *nadklase*. Činjenica da potklasa nasleđuje metode znači da njih ne treba pisati ponovo. Treba dopisati samo nove stvari, kao i one koje se razlikuju. Ova osobina se zove *code reuse* (jer potklasa koristi kôd iz nadklase).



Nasleđivanje je relacija među klasama koja uređuje klase u hijerarhijsku strukturu. Ako je klasa C nasledila klasu B onda ćemo to označiti sa $C : B$. Za klasu B kažemo da je *bazna klasa* za klasu C, ili da je *nadklasa* klase C, dok za klasu C kažemo da je *potklasa* klase B. Potklasa može biti *direktna* ili *indirektna*. Na primer, ako je $C' : C$ i $C : B$ onda je C direktna potklasa klase B, dok je C' potklasa klase B, mada ne direktna.

Ideje na kojima počiva objektno-orijentisano programiranje su prilično stare[1], ali tek krajem prošlog veka postaju popularne. Moglo bi se čak reći da se desilo i nešto nepoželjno: ne samo da je objektno-orijentisano programiranje postalo priznato kao elegantna i efikasna tehnika programiranja, već je postalo moda. To nije dobro. Ima mnogo vrsta problema koji se mogu brže i jednostavnije rešiti klasičnim metodama. S druge strane, projektovanje velikih softverskih sistema može biti mnogo lakše ukoliko se projektantski timovi opredele za objektno-orijentisanu metodologiju.

Objektno orijentisano programiranje ne treba gurati tamo gde mu nije mesto. Na primer, programi napisani uz upotrebu objektno-orijentisanih programskih jezika se izvršavaju sporije iz najmanje dva razloga: dinamičkog vezivanja metoda i kupljenja đubreta (engl. *garbage collection*). Zato ova metodologija uglavnom nije pogodna za *real-time* aplikacije i softverske sisteme kod kojih je vreme izvršavanja kritično. Objektno-orijentisana metodologija razvoja aplikacija je korisna i dobra metodologija, ali nikako ne i najbolja. Sve to treba imati u vidu kada se bira alat pri ulasku u veliki projekt.

10.9 Nasleđivanje

Krenimo od klase `Figura` koja opisuje figure u ravni. Za svaku figuru pamtimo poziciju centra figure u promenljivim `centar_X` i `centar_Y`, pamtimo ime figure i boju figure. Sve ove promenljive su privatne, i time zaštićene od neovlašćenog pristupa:

```
class Figura {
  private int centar_X, centar_Y;
  private string ime_figure;
  private string boja_figure;

  public Figura(string i, int x, int y, string b) {
    centar_X = x; centar_Y = y;
```

[1] kao prvi objektno-orijentisani jezik smatra se Simula-67 koja je nastala 1967. godine

```
    ime_figure = i; boja_figure = b;
  }

  public string Ime() {
    return ime_figure;
  }

  public void PomeriZa(int dx, int dy) {
    centar_X += dx; centar_Y += dy;
  }

  public void Ispisi() {
    Console.WriteLine("Objekat: {0}", ime_figure);
    Console.WriteLine("  centar: ({0}, {1})", centar_X, centar_Y);
    Console.WriteLine("  boja: {0}", boja_figure);
  }
}
```

Da bismo pojednostavili kôd nećemo crtati figure nego ćemo ispisivati informacije o njima. Metod `Ispisi` ispisuje osnovne informacije o figuri, dok metod `PomeriZa` pomera figuru za dati vektor. Metod `Ime` vraća ime figure. Evo malog primera upotrebe ove klase:

```
  static void Main() {
    Figura F = new Figura("Figura 1", 10, 15, "crvena");
    F.PomeriZa(-10, 5);
    Console.WriteLine("Figura {0} je pomerena!", F.Ime());
    F.Ispisi();
  }
```

Program će ispisati:

```
Figura Figura 1 je pomerena!
Objekat: Figura 1
  centar: (0, 20)
  boja: crvena
```

Želeli bismo sada da napravimo klasu `Krug` koja će predstavljati krugove. Jedan način je da uzmemo klasu `Figura` i da je malo preradimo. Na taj način dobijamo dve klase kod kojih je 90% kôda identično. Gubimo vreme i memorijski prostor. Zar ne bi bilo lepo kada bismo mogli da kažemo da je klasa `Krug` "veoma slična" sa klasom `Figura`, pa da dopišemo samo razliku?

Ne samo da bi bilo lepo, nego je to i moguće! Mehanizam koji nam to omogućuje zove se *nasleđivanje* (engl. inheritance). Napravićemo novu klasu i, umesto da

prekucavamo kôd iz klase `Figura`, reći ćemo da klasa `Krug` nasleđuje sve osobine klase `Figura`, pa ćemo samo dopisati nove (specifične) metode i redefinisati neke stare. Recimo, ovako:

```
class Krug : Figura {
  private int r_kruga;

  public Krug(string i, int x, int y, string b, int r)
         : base(i, x, y, b)
  {
    r_kruga = r;
  }
}
```

Konstrukcija `Krug : Figura` znači da klasa `Krug` *nasleđuje* klasu `Figura`, sva njena polja i metode, i dodaje novo polje `r` koje sadrži poluprečnik kruga. Konstruktoru klase `Krug` treba jedan parametar više zato što nam za krug treba podatak više u odnosnu na konstruktor klase `Figura`. S durge strane, on u suštini radi isto što i stari konstruktor klase `Figura`, s tim da dodatno inicijalizuje polje `r`. Zato smo u deklaraciji konstruktora klase `Krug` naveli

```
: base(i, x, y, b)
```

što znači "pozovi prvo konstruktor iz nadklase (engl. *base class*, odatle ime) sa navedenim parametrima, pa kada on inicijalizuje svoja polja, ti dodatno inicijalizuj još polje `r`." Evo malog primera upotrebe klase `Krug`:

```
  static void Main() {
    Krug K = new Krug("Krug 1", 10, 15, "crvena", 22);
    K.Ispisi();
  }
```

Program će ispisati:

```
Objekat: Krug 1
  centar: (10, 15)
  boja: crvena
```

Hm... gde je informacija o poluprečniku kruga? Ako pažljivo pogledamo kôd videćemo da je program uradio tačno ono što smo mu rekli, a ne ono što bismo mi voleli da se desilo. Metod `Ispisi` ispisuje ime, centar i boju figure.

Problem je u tome što klasa `Figura` ne može da zna ko će je sve naslediti i šta će nove klase želeti da urade. Mi bismo mogli u klasi `Figura` da napišemo novi

metod, recimo IspisiKrug, koji ispisuje sve podatke o krugu, ali bi bilo mnogo bolje kada bismo imali mehanizam da iz klase Krug doradimo metod Ispisi() koji je definisan u klasi Figura. Da bi se rešio ovaj problem uvedeni su *virtuelni metodi*.

10.10 Virtuelni metodi

Virtuelni metod je metod koji može biti promenjen ili na neki drugi način dorađen u nekoj klasi koja nasleđuje ovu klasu. Ako metod nije označen kao virtuelan, klase koje nasleđuju tekuću klasu *ne smeju da menjaju* značenje ovog metoda. Klasu Figura, zato, treba napisati na sledeći način:

```
class Figura {
  private int centar_X, centar_Y;
  private string ime_figure;
  private string boja_figure;

  public Figura(string i, int x, int y, string b) {
    centar_X = x; centar_Y = y;
    ime_figure = i; boja_figure = b;
  }

  public string Ime() {
    return ime_figure;
  }

  public void PomeriZa(int dx, int dy) {
    centar_X += dx; centar_Y += dy;
  }

  public virtual void Ispisi() {
    Console.WriteLine("Objekat: {0}", ime_figure);
    Console.WriteLine("  centar: ({0}, {1})", centar_X, centar_Y);
    Console.WriteLine("  boja: {0}", boja_figure);
  }
}
```

Metod Ispisi je označen kao virtuelan i njega smemo kasnije da dorađujemo. S druge strane, metod PomeriZa ne sme biti promenjen ni u jednoj klasi koja nasleđuje klasu Figura zato što nije označen kao virtuelan. Klasa Krug se sada može napisati ovako:

```
class Krug : Figura {
```

```
    private int r_kruga;

    public Krug(string i, int x, int y, string b, int r)
           : base(i, x, y, b)
    {
      r_kruga = r;
    }

    public override void Ispisi() {
      base.Ispisi();
      Console.WriteLine("  poluprecnik: {0}", r_kruga);
    }
  }
```

Primetimo da je u klasi `Krug` metod `Ispisi` *redefinisan* tako što smo ispred njegove deklaracije stavili ključnu reč `override`. Time smo naglasili kompajleru da želimo da doradimo odgovarajući metod iz nadklase. Nova definicija metoda `Ispisi` počinje naredbom

```
base.Ispisi();
```

koja pozove metod `Ispisi` iz nadklase. Ovaj metod ispiše ime, položaj centra i boju objekta, pa metod `Ispisi` iz klase `Krug` treba još samo da ispiše podatak o poluprečniku kruga. Ako sada startujemo program:

```
static void Main() {
  Krug K = new Krug("Krug 1", 10, 15, "crvena", 22);
  K.Ispisi();
}
```

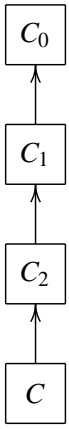
on će ispisati:

```
Objekat: Krug 1
  centar: (10, 15)
  boja: crvena
  poluprecnik: 22
```

Prve tri linije izveštaja je ispisao metod `Ispisi` klase `Figura`, dok je poslednju liniju izvestaja ispisao metod `Ispisi` klase `Krug`.

10.11 Vezivanje metoda

Neka je X objekat klase C i neka je $C : C_2 : C_1 : C_0$. Ako objektu X uputimo poruku $m()$ na uobičajeni način $X.m()$ potrebno je naći klasu u kojoj je implementiran metod m. Prvo se pregleda klasa C. Ako u njoj postoji metod sa imenom "m", onda se poruka $m()$ protumači kao poziv metoda "m" iz klase C. No, ako u klasi C ne postoji metod sa imenom "m", objekt neće odbaciti poruku, već će proveriti da li u nekoj od nadklasa postoji odgovarajući metod. Potraga za metodom se nastavlja u njegovoj direktnoj nadklasi (to je, u ovom primeru, klasa C_2). Ako u klasi C_2 postoji metod koji se zove "m", onda je sve u redu. Poziv $X.m()$ se protumači kao primena metoda "m" iz klase C_2 na objekt X. U suprotnom se proverava da li u klasi C_1 postoji metod "m", itd... Potraga se nastavlja sve dok se ne pronađe odgovarajući metod, ili dok ne stignemo do vrha hijerarhije. Ako nigde nema željenog nam metoda, on se odbacuje (uz odgovarajuće posledice).

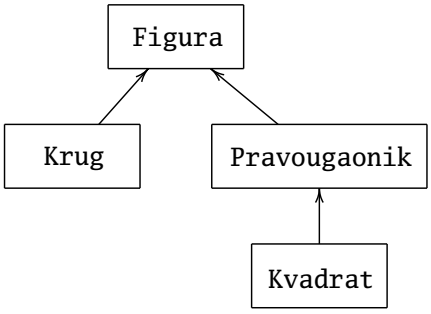


Jedna od lepih (ali i zapetljanijih) stvari vezanih za nasleđivanje je da se promenljivoj koja je deklarisana kao promenljiva klase C_1 može dodeliti objekt bilo koje klase C_2 za koju je $C_2 : C_1$. Na primer, ako imamo da je

```
Figura F;
Krug K;
```

onda je dodela `F = K` ispravna, jer je `Krug` potklasa klase `Figura` (krug je specijalna figura), dok dodela `K = F` *nije ispravna* jer nije tačno da je svaka figura ujedno i krug.

Evo sada složenijeg primera. Neka je data klasa `Figura` i njene potklase `Krug`, `Pravougaonik` i `Kvadrat` koje su organizovane ovako:



i koje su definisane na sledeći način:

⟨C# fajl⟩

```
using System;

class Figura {
  private int centar_X, centar_Y;
  private string ime_figure;
  private string boja_figure;

  public Figura(string i, int x, int y, string b) {
    centar_X = x; centar_Y = y;
    ime_figure = i; boja_figure = b;
  }

  public string Ime() {
    return ime_figure;
  }

  public void PomeriZa(int dx, int dy) {
    centar_X += dx; centar_Y += dy;
  }

  public virtual void Ispisi() {
    Console.WriteLine("Objekat: {0}", ime_figure);
    Console.WriteLine("  centar: ({0}, {1})", centar_X, centar_Y);
    Console.WriteLine("  boja: {0}", boja_figure);
  }
}

class Krug : Figura {
  private int r_kruga;

  public Krug(string i, int x, int y, string b, int r)
         : base(i, x, y, b)
  {
    r_kruga = r;
  }

  public override void Ispisi() {
    base.Ispisi();
    Console.WriteLine("  poluprecnik: {0}", r_kruga);
  }
}

class Pravougaonik : Figura {
```

```
  private int duzina, sirina;

  public Pravougaonik(string i, int x, int y, string b, int d, int s)
         : base(i, x, y, b)
  {
    duzina = d;
    sirina = s;
  }

  public override void Ispisi() {
    base.Ispisi();
    Console.WriteLine("  dimenzije: {0} x {1}", duzina, sirina);
  }
}

class Kvadrat : Pravougaonik {
  public Kvadrat(string i, int x, int y, string b, int a)
         : base(i, x, y, b, a, a)
  { }
}

class GlavniProgram {
  static void Main() {
    Krug K = new Krug("Krug 1", 10, 15, "plava", 100);
    Pravougaonik P = new Pravougaonik("Pravougaonik 1", 27, 39,
                                      "zelena", 35, 45);
    Kvadrat Q = new Kvadrat("Kvadrat 1", 0, 0, "crvena", 30);

    Figura F;
    F = K; F.Ispisi();
    F = P; F.Ispisi();
    F = Q; F.Ispisi();
  }
}
```

Klasa `Kvadrat` je zanimljiva i zaslužuje komentar. Ona je potklasa klase `Pravougaonik` i zapravo ništa ne dodaje ili menja u klasi `Pravougaonik`, već samo obezbeđuje da objekat bude inicijalizovan tako da su mu dužina i širina jednaki. Sve ostale osobine nasleđuje od klase `Pravougaonik`.

Kada izvršimo program (zapravo, metod `Main` klase `GlavniProgram`), računar će ispisati:

```
Objekat: Krug 1
  centar: (10, 15)
  boja: plava
```

```
  poluprecnik: 100
Objekat: Pravougaonik 1
  centar: (27, 39)
  boja: zelena
  dimenzije: 35 x 45
Objekat: Kvadrat 1
  centar: (0, 0)
  boja: crvena
  dimenzije: 30 x 30
```

Iako smo svaki put pozvali `F.Ispisi();` gde je `F` objekat klase `Figura`, pozvana su tri različita metoda. Mehanizam koji realizuje ovo ponašanje se zove *dinamičko vezivanje metoda*. Ideja dinamičkog vezivanja je u tome da svaki objekat sa sobom nosi svoj paket metoda i zato će u svakom slučaju biti pozvan tačno onaj metod koji odgovara klasi kojoj objekat pripada. Recimo, nakon naredbi

```
F = Q; F.Ispisi();
```

u promenljivoj `F` se nalazi objekat klase `Kvadrat`. Zato potraga za metodom `Ispisi` kreće iz klase `Kvadrat`. Kako ona ne sadrži metod sa imenom `Ispisi` prelazimo na njenu direktnu nadklasu (klasa `Pravougaonik`). Tamo nalazimo metod sa imenom `Ispisi` i sistem njega aktivira. A izvršavanje tog metoda pozivom

```
base.Ispisi();
```

aktivira i metod `Ispisi` klase `Figura`.

Dinamičko vezivanje je jako zgodna stvar. Sada ćemo demonstrirati jednu interesantnu primenu. Posmatrajmo niz ravanskih figura:

```
Figure[] F;
```

i pretpostavimo da smo na neki način učitali figure u taj niz, recimo ovako:

```
int N = int.Parse(Console.ReadLine());
for(int i = 0; i < N; i++) {
  Console.Write("Unesi vrstu figure -> ");
  string s = Console.ReadLine();
  if(s == "Krug") {
    // na neki nacin ucitaj krug u F[i]
  }
  else if(s == "Pravougaonik") {
    // na neki nacin ucitaj pravougaonik u F[i]
  }
  else {
    // na neki nacin ucitaj kvadrat u F[i]
  }
}
```

Ako treba ispisati podatke o svih *N* figura ovo je najelegantniji način da se to učini:

```
for(int i = 0; i < N; i++) { F[i].Ispisi(); }
```

Ako treba pomeriti celu konstelaciju figura za vektor `(dx, dy)`, to se može učiniti ovako:

```
for(int i = 0; i < N; i++) { F[i].PomeriZa(dx, dy); }
```

Uočimo da se metod `PomeriZa` nalazi u klasi `Figura`. On nije reimplementiran ni u jednoj od potklasa koje koristimo, ali nasleđivanje obezbeđuje da on ipak bude pronađen i primenjen kako treba.

10.12 Apstraktne klase i apstraktni metodi

U sledećem koraku želimo da sistemu dodamo metod koji računa površinu figure. Problem je u tome što u klasi `Figure` nemamo nikakvu informaciju o geometriji figure. Zato računanje površine moramo implementirati u pojedinačnim klasama gde znamo o kojoj figuri je reč. S druge strane, ako se metod koji računa površinu pojavljuje samo u klasama kao što su `Pravougaonik` i `Krug`, onda sledeći zapis postaje nemoguć

```
Figure[] F;
// na neki nacin ucitamo figure u niz F
for(int i = 0; i < F.Length; i++) {
  Console.WriteLine("Povrsina figure {0} je {1}",
                    F[i].Ime(), F[i].Povrsina());
}
```

zato što klasa `Figura` nema informaciju o tome šta će sve jednog dana biti implementirano u njenim potklasama.

Rešenje ovog problema koristi dosetku koja se zove *apstraktni metodi*. Apstraktni metodi se javljaju u onim situacijama u kojima se metod implementira na potpuno različite načine u različitim potklasama te klase. Površina kruga i pravougaonika se računa na potpuno različite načine. Jasno je da nije moguće da se u klasi `Figura` obezbedi metod koji bi bio u stanju da izračuna površinu svega što može da bude ravanska figura. Zato se u klasi `Figura` samo označi da postoji metod `Povrsina` koji moraju imati sve potklase klase `Figura` i da će on biti implementiran tek tamo. Najviše što se može očekivati od klase `Figura` je to da nagovesti potrebu za ovim metodom. Klasa `Figura` ništa ne može da kaže o njegovoj implementaciji. Apstraktni metod nema telo, već samo specifikaciju metoda. Na primer ovako:

```
public abstract double Povrsina();
```

Deklaracija `abstract` predstavlja obećanje prevodiocu. Ona znači da će metod `Povrsina` biti definisan negde u nekoj potklasi, ali da ćemo se na njega pozivati pre nego što bude implementiran.

Apstraktna klasa je klasa koja ima bar jedan apstraktni metod. Klasa koja nema apstraktne metode se zove *konkretna klasa.* Apstraktna klasa mora biti deklarisana kao apstraktna odmah na početku. Evo primera klase `Figura` koja je sada deklarisana kao apstraktna klasa jer ima apstraktni metod `Povrsina` (u njene tri potklase je taj metod implementiran i zato te tri nisu apstraktne):

⟨C# fajl⟩

```
using System;

abstract class Figura {
  private int centar_X, centar_Y;
  private string ime_figure;
  private string boja_figure;

  public Figura(string i, int x, int y, string b) {
    centar_X = x; centar_Y = y;
    ime_figure = i; boja_figure = b;
  }

  public string Ime() {
    return ime_figure;
  }

  public void PomeriZa(int dx, int dy) {
    centar_X += dx; centar_Y += dy;
  }

  public virtual void Ispisi() {
    Console.WriteLine("Objekat: {0}", ime_figure);
    Console.WriteLine("  centar: ({0}, {1})", centar_X, centar_Y);
    Console.WriteLine("  boja: {0}", boja_figure);
  }

  public abstract double Povrsina();
}

class Krug : Figura {
  private int r_kruga;

  public Krug(string i, int x, int y, string b, int r)
         : base(i, x, y, b)
  {
```

```
    r_kruga = r;
  }

  public override void Ispisi() {
    base.Ispisi();
    Console.WriteLine("  poluprecnik: {0}", r_kruga);
  }

  public override double Povrsina() {
    return r_kruga * r_kruga * Math.PI;
  }
}

class Pravougaonik : Figura {
  private int duzina, sirina;

  public Pravougaonik(string i, int x, int y, string b, int d, int s)
         : base(i, x, y, b)
  {
    duzina = d;
    sirina = s;
  }

  public override void Ispisi() {
    base.Ispisi();
    Console.WriteLine("  dimenzije: {0} x {1}", duzina, sirina);
  }

  public override double Povrsina() {
    return duzina * sirina;
  }
}

class Kvadrat : Pravougaonik {
  public Kvadrat(string i, int x, int y, string b, int a)
         : base(i, x, y, b, a, a)
  { }
}

class GlavniProgram {
  static void Main() {
    Figura[] F = new Figura[3];
    F[0] = new Krug("Krug 1", 10, 15, "plava", 100);
    F[1] = new Pravougaonik("Pravougaonik 1", 27, 39,
                            "zelena", 35, 45);
```

```
    F[2] = new Kvadrat("Kvadrat 1", 0, 0, "crvena", 30);
    for(int i = 0; i < 3; i++) {
      Console.WriteLine("Povrsina figure {0} je {1}",
                        F[i].Ime(), F[i].Povrsina());
    }
  }
}
```

Kako smo klasu `Figura` deklarisali kao apstraktnu klasu sa apstraktnim metodom `Povrisna`, u glavnom programu slobodno smemo da napišemo:

```
for(int i = 0; i < 3; i++) {
  Console.WriteLine("Povrsina figure {0} je {1}",
                    F[i].Ime(), F[i].Povrsina());
}
```

☞ *Veoma je važno istaći da apstraktne klase ne mogu da imaju instance!*

Razlog je veoma jednostavan: instanca ne zna kako radi nijedan od apstraktnih metoda klase. Tek *konkretne* klase koje naslede apstraktnu klasu i implementiraju svaki od njenih apstraktnih metoda mogu da se instanciraju.

Kada se prevodi program napisan na nekom objektno-orijentisanom jeziku koji ima apstraktne metode, statička provera tipova utvrđuje da li su ispunjena sva obećanja koja je programer dao. Dakle, prevodilac proverava da li postoji bar jedna implementacija za svaki apstraktni metod koji se javlja u programu. Naravno, može ih postojati više. Dinamičko vezivanje metoda određuje koju implementaciju apstraktnog metoda treba shvatiti kao interpretaciju poruke.

Potklasa apstraktne klase ne mora ponuditi implementaciju za apstraktne metode. Štaviše, potklasa apstraktne klase može dodati i nove apstraktne metode. Bitno je samo to da za svaku apstraktnu klasu postoji njena potklasa koja je konkretna. Time se obezbeđuje da svaki apstraktni metod bude nekad i negde implementiran.

Takođe, moguće je da potklasa neke konkretne klase bude apstraktna. Za nju tada važi sve što važi i za ostale apstraktne klase.

Zadaci.

10.1. Napisati C# klasu `Poligon` koja podržava rad sa poligonima. Klasa treba da ima sledeće metode:

- konstruktor `Poligon(string fname)` koji iz datoteke `fname` učitava broj i koordinate temena poligona;
- `double Obim()` koji računa obim poligona;

- `double Povrsina()` koji računa površinu poligona koristeći sledeću formulu:
$$P = \frac{1}{2} \cdot \left| \sum_{i=0}^{n-1} (x_i y_{i\oplus 1} - x_{i\oplus 1} y_i) \right|,$$
gde je n broj temena poligona, (x_i, y_i) koordinate i-tog temena poligona, a $\oplus 1$ uvećavanje za 1 po modulu n;
- `void Dijametar()` koji računa dijametar poligona (dijametar figure je najveće rastojanje koje postižu dve tačke te figure);
- `Poligon BoundingBox()` koji određuje *bounding box* poligona, što je najmanji pravougaonik koji sadrži poligon a čije strane su paralelne koordinatnim osama.

10.2. Napisati C# klasu `Date` koja podržava rad sa datumima. Datum treba da bude predstavljen preko tri celobrojna polja `day`, `month`, `year` i treba da podržava sledeće metode:

- konstruktor `Date(int d, int m, int g)` koji postavlja datum i koji treba da proveri da li je datum korektan,
- `void Write()` koji ispisuje vreme u obliku `d-m-g`,
- `void Add(int n)` koji na datum nad kojim je metod primenjen dodaje `n` dana na datum,
- `void Sub(int n)` koji od datuma nad kojim je metod primenjen oduzima `n` dana od datuma,
- `bool InBetween(Date p, Date q)` koji proverava da li se datum nad kojim je metod primenjen nalazi između datuma `p` i datuma `q`,
- `bool Before(Date p)` koja proverava da li je datum nad kojim je metod primenjen raniji u odnosu na `p`,
- `bool Equal(Date p)` koja proverava da li je datum nad kojim je metod primenjen jednak datumu `p`.

10.3. Napisati C# program koji učitava n događaja i sortira ih po datumu održavanja. Događaj je opisan kao klasa `Event` koja ima dva polja: `string name` što predstavlja ime događaja i `Date d` koje predstavlja datum održavanja događaja.

10.4. Napisati metod `static Frac Mid3(Frac a, Frac b, Frac c)` koja vraća srednji od tri data razlomka.

10.5. Napisati C# program koji od korisnika učitava pozitivan ceo broj n, potom n razlomaka $a_1, \ldots, a_n$ i računa i štampa vrednost sledećeg izraza u obliku skraćenog razlomka: $-a_1 + a_2 - a_3 + a_4 + \ldots + (-1)^n a_n$.

10.6. Napisati C# program koji od korisnika učitava ceo broj $n \geqslant 3$, potom n razlomaka $q_1, \ldots, q_n$ i računa i štampa vrednost izraza

$$q_1 + q_1q_2 + q_1q_2q_3 + \ldots + q_1q_2 \ldots q_n.$$

10.7. Napisati C# program koji od korisnika učitava pozitivan ceo broj n, potom n razlomaka $a_1, \ldots, a_n$ i uređuje i štampa date razlomke po veličini, od najmanjeg ka najvećem (sortiranje razlomaka).

10.8. Polinom $a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1x + a_0$, $a_{n-1} \neq 0$, sa celobrojnim koeficijentima može da se opiše nizom $(a_0, a_1, \ldots, a_{n-1})$ celih brojeva na očigledan način. Napisati C# metod `static Frac Eval(int[] p, Frac x)` koji za dati razlomak `x` računa vrednost polinoma `p`.