REPUBLIKA SLOVENIJA
MINISTRSTVO ZA ŠOLSTVO IN ŠPORT

Naložba v vašo prihodnost
OPERACIJO DELNO FINANCIRA EVROPSKA UNIJA
Evropski socialni sklad

VIŠJEŠOLSKI STROKOVNI PROGRAM
INFORMATIKA

PROGRAMIRANJE 1

Višješolski strokovni program: Informatika
Učbenik: Programiranje 1
Gradivo za 1. letnik

Avtorja:

Matija Lokar
Univerza v Ljubljani
Fakulteta za matematiko in fiziko

Srečo Uranič
TEHNIŠKI ŠOLSKI CENTER KRANJ
Višja strokovna šola

Kranj, 2008

Gradivo je sofinancirano iz sredstev projekta Impletum 'Uvajanje novih izobraževalnih programov na področju višjega strokovnega izobraževanja v obdobju 2008–11'.

Projekt oz. operacijo delno financira Evropska unija iz Evropskega socialnega sklada ter Ministrstvo RS za šolstvo in šport. Operacija se izvaja v okviru Operativnega programa razvoja človeških virov za obdobje 2007–2013, razvojne prioritete 'Razvoj človeških virov in vseživljenjskega učenja' in prednostne usmeritve 'Izboljšanje kakovosti in učinkovitosti sistemov izobraževanja in usposabljanja'.

Vsebina tega dokumenta v nobenem primeru ne odraža mnenja Evropske unije. Odgovornost za vsebino dokumenta nosi avtor.

Projekt **Impletum**

Uvajanje novih izobraževalnih programov na področju višjega strokovnega izobraževanja v obdobju 2008–11

Konzorcijski partnerji:

Operacijo delno financira Evropska unija iz Evropskega socialnega sklada ter Ministrstvo RS za šolstvo in šport. Operacija se izvaja v okviru Operativnega programa razvoja človeških virov za obdobje 2007–2013, razvojne prioritete 'Razvoj človeških virov in vseživljenjskega učenja' in prednostne usmeritve 'Izboljšanje kakovosti in učinkovitosti sistemov izobraževanja in usposabljanja'.

KAZALO

PREDGOVOR

Gradivo *Programiranje 1* je namenjeno študentom 1. letnika višješolskega študija Informatika. Pokriva vsebinski del naveden v katalogu znanja, pri čemer sva kot izbrani programski jezik vzela programski jezik C#. V uvodnem delu je opisano tudi razvojno okolje Microsoft Visual C# 2008 Express Edition. Vendar samo gradivo ni vezano na uporabo tega okolja in se lahko uporabljajo tudi druga.

Pri sestavljanju gradiva sva imela v mislih predvsem začetnike, ki se s programskimi jeziki srečujejo prvič. Zato je v besedilu veliko primerov programov in zgledov. Prav tako (namerno) zamolčiva marsikaj, vrsto stvari pa poenostaviva. Izkušnje namreč kažejo, da predvsem za začetnika te stvari niso pomembne. Še več, pogosto le motijo, saj preusmerjajo pozornost na manj važne stvari. Študente, ki bi o posamezni tematiki radi zvedeli več, vabiva, da si ogledajo tudi gradivo, ki je navedeno v literaturi.

Samo gradivo je nastalo na osnovi zapiskov obeh avtorjev, črpala pa sva tudi iz že objavljenih gradiv, pri katerih sva bila »vpletena«. V gradivu je koda, napisana v programskem jeziku nekoliko osenčena, saj sva jo na ta način želela tudi vizualno ločiti od teksta gradiva. Žal predpisi oblikovanja v sklopu projekta ne dovoljujejo, da bi za kodo uporabila kako drugo pisavo, na primer Courier.

V besedilu so številni zgledi, zaradi pomanjkanja prostora pa sva se odločila, da bova vaje in rešitve vaj zapisala v ločeno gradivo. Ker se programiranja seveda ne da naučiti le s prepisovanjem tujih programov, pričakujeva, da bodo študenti poleg skrbnega študija zgledov, pisali programe tudi sami. Zato jim na več mestih predlagava, da rešijo naloge, ki so objavljene v več, na spletu dosegljivih, zbirkah nalog. Predvsem so to gradiva iz projekta UP - Kako poučevati začetni tečaj programskega jezika, sklop interaktivnih gradiv, ki so dosegljiva na naslovu http://up.fmf.uni-lj.si/ in gradiva v sklopu *Wiki C#*, dostopna na naslovu http://penelope.fmf.uni-lj.si/C_sharp. V kratkem pa naj bi bila na voljo tudi zbirka nalog Programiranje 1 - zbirka rešenih nalog v C#. Zbirka ima poglavja razporejena na identičen način kot to besedilo, zato ne bo težav najti primerne naloge.

Predavateljem, ki pa bodo uporabljati ta učbenik pri svojem predmetu, pa priporočava tudi uporabo spletne učilnice, za katero avtorja meniva, da je postala nepogrešljiv pripomoček pri poučevanju programiranja. Tako je v sklopu gradiv projekta *UP* na voljo velika zbirka vprašanj, podprtih z avtomatskim preverjanjem pravilnosti. Iz njih lahko sestavljamo različne kvize. Ta vprašanja so na voljo bodisi za neposredno vključitev v spletne strani, bodisi v obliki *MoodleXML* in jih je mogoče uvoziti v spletno učilnico, podprto s sistemom *Moodle*.

Gradivo sva opremila tudi z motivirajočo slikovno podporo za nekatere bistvene poudarke (Vir: http://commons.wikimedia.org/wiki/Crystal_Clear) in sicer:

- Slikovna podpora za povzetke:

- Slikovna podpora za dodatne naloge, vprašanja, gradiva oz. reference:

- Slikovna podpora za dodatne informacije in dodatna gradiva:

V Kranju, december 2008

1 UVOD - ZAKAJ UČENJE PROGRAMIRANJA

Pogosta trditev, ki odvrača od učenja programiranja je mnenje, da uporabniku računalnika danes ni potrebno znati programirati. Pravijo: *»Saj imamo za vse, kar želimo narediti z računalnikom, na voljo ustrezna orodja.«* Znanje programiranja je po mnenju mnogih povsem odveč. Razmišljajo: »*Programiranje je le zelo specialistično znanje skupinice strokovnjakov, ki pišejo programe, ki jih potem običajni uporabniki uporabljamo.«*

Vendar tako razmišljanje ni utemeljeno. Razlogov, zakaj je vseeno dobro poznati vsaj osnove programiranja, je veliko:

- učenje programiranja je privajanje na algoritmični način razmišljanja. Mirno lahko trdimo, da je znanje algoritmičnega razmišljanja nujna sestavina sodobne funkcionalne pismenosti. Danes ga potrebujemo praktično na vsakem koraku. Uporabljamo oz. potrebujemo ga:
  - ko sledimo postopku za pridobitev denarja z bankomata;
  - pri vsakršnem delu z računalnikom;
  - ko se odločamo za podaljšanje registracije osebnega avtomobila;
  - omogoča nam branje navodil, postopkov (pogosto so v obliki "kvazi" programov, diagramov poteka);
- za umno naročanje ali izbiranje programske opreme;
  - da znamo pravilno predstaviti (opisati, zastaviti, ...) problem, ki ga potem programira nekdo drug;
- osnovne programske tehnike potrebujemo za pisanje makro ukazov v uporabniških orodjih;
- za potrebe administracije – delo z več uporabniki;
- za kreiranje dinamičnih spletnih strani;
- za popravljanje "tujih" programov.

Splošno znani pregovor pravi: »*Več jezikov znaš, več veljaš!*«. Zakaj med "tujimi" jeziki ne bi bil tudi kak programski jezik? Če bi se tolikokrat "pogovarjali" s Kitajcem, kot se z računalnikom, se ne bi naučili nekega skupnega jezika, ki bi ga "obvladala" oba? Tudi "napredni uporabnik" se slej kot prej sreča s programiranjem. In prav z učenjem programiranja najlažje osvojimo ta način razmišljanja.

Zaradi vsega tega je danes za vsakega uporabnika računalnikov skoraj nujno, da je med jeziki, ki jih obvlada, tudi nek programski jezik. Teh je zelo veliko. Med njimi je tudi programski jezik C#, o katerem bo govora tukaj.

Podrobneje pa si o tem, kakšni so razlogi za učenje programiranja in kako ti znanje že enega programskega jezika lahko zelo olajša delo z računalnikom, preberite v Lokar, Osnove programiranja: programiranje – zakaj in vsaj kaj. Premislite, če so tam navedeni razlogi smiselni in če ste pri svojem delu z računalnikom že naleteli v tej knjižici navedene zglede.

2 VISUAL C# IN .NET FRAMEWORK

C# je objektno usmerjen programski jezik (karkoli pač že to pomeni), ki so ga razvili v podjetju Microsoft. Jezik izvira pretežno iz programskih jezikov C++, Visual Basic in java. Trenutno obstaja v različici 3. Namenjen je pisanju programov v okolju .NET in je primeren za razvoj najrazličnejše programske opreme. Poleg tega je zasnovan na tak način, da ga je mogoče preprosto izboljševati in razširjati, brez nevarnosti, da bi s tem izgubili združljivost z obstoječimi programi. C# je oblikovan za delo z Microsoftovo platformo .NET (.NET Framework). .NET Framework je skupek tehnik in tehnologij, ki predstavlja ogrodje za različna programska orodja in aplikacije za osebne računalnike, dlančnike, pametne telefone, razne vgrajene sisteme … . To ogrodje je sestavni del operacijskega sistema Windows, oziroma ga lahko temu operacijskemu sistemu dodamo. Ogrodje je sestavljeno iz velikega števila rešitev za različna programska področja kot so gradnja uporabniških vmesnikov, dostopanje do podatkov, kriptografija, razvoj mrežnih aplikacij, numerični algoritmi, omrežne komunikacije … . Programerji pri razvoju programov metode, ki so uporabljene v teh rešitvah, kombinirajo z lastno kodo.

2.1 RAZVOJ PROGRAMSKE REŠITVE V OKOLJU MICROSOFT VISUAL STUDIO .NET

Za pisanje programov v jeziku C# (in tudi ostalih jezikih v okolju .NET) je Microsoft razvil razvojno okolje, ki se imenuje **Microsoft Visual Studio.NET**. Združuje zmogljiv urejevalnik kode, prevajalnik, razhroščevalnik, orodja za dokumentacijo programov in druga orodja, ki pomagajo pri pisanju programskih aplikacij. Poleg tega okolje nudi tudi podporo različnim programskim jezikom, kot so na primer C#, C++ in Visual Basic.

Microsoft Visual Studio.Net obstaja v več različicah. Za spoznavanje z jezikom C# zadošča brezplačna različica Visual C# Express Edition. Ta podpira le razvoj programov v jeziku C#. Prenesemo jo lahko z Microsoftovih spletnih strani. Ker se točen naslov, na katerem to različico dobimo, občasno spreminja, naj naslov bralec poišče kar sam. Če v poljubni iskalnik vpišemo ključne besede "Visual Studio Express Download", bomo program zagotovo našli. V besedilu bomo več ali manj opisovali v trenutku pisanja najnovejšo različico tega okolja, Visual C# 2008 Express Edition. Vendar praktično vse povedano velja tudi za starejše različice tega okolja. Seveda pa obstajajo tudi druga razvojna okolja za pisanje programov v C#, npr. okolje **SharpDevelop** (http://www.icsharpcode.net/OpenSource/SD/).

Ker se sam način učenja programskega jezika ne veže na okolje, je povsem vseeno, katero razvojno okolje bo bralec uporabljal pri učenju jezika.

Programske rešitve običajno ne sestavlja le ena datoteka z izvorno kodo, ampak je datotek več. Skupek datotek in nastavitev, ki rešujejo določen problem, imenujemo **projekt**. Glede na to, za kakšno vrsto programske rešitve gre, imamo v okolju Visual C# Express vnaprej pripravljenih več različnih tipov projektov:

- **Console Application** (ali konzolne aplikacije) – namenjene gradnji aplikacij, ki ne potrebujejo grafičnega vmesnika. Izvajajo se preko ukaznega okna ali kot mu tudi rečemo *konzole* (t.i. »DOS-ovskih« aplikacij).
- **Windows Forms Application** (ali namizne aplikacije) – namenjene za gradnjo namiznih aplikacij s podporo grafičnih gradnikov.
- **Windows Presentation Foundation (WPF) Application** – namenjen za gradnjo programov, ki tečejo v okolju Windows, zasnovanih na uporabi najnovejših gradnikov okolja WPF.
- **Windows Presentation Foundation (WPF) Browser Application** – namenjen za programiranje programov, ki tečejo v spletnih brskalnikih (npr. Internet Explorer, Firefox).
- **Class Library** (ali knjižnice) – namenjene gradnji knjižnic razredov.
- **Empty Project** (ali prazen projekt) – namenjen gradnji aplikacij brez vnaprej določenega vzorca.

Izbira projekta določa, kakšno bo vnaprej pripravljeno ogrodje programov, katere knjižnice se bodo naložile in podobno.

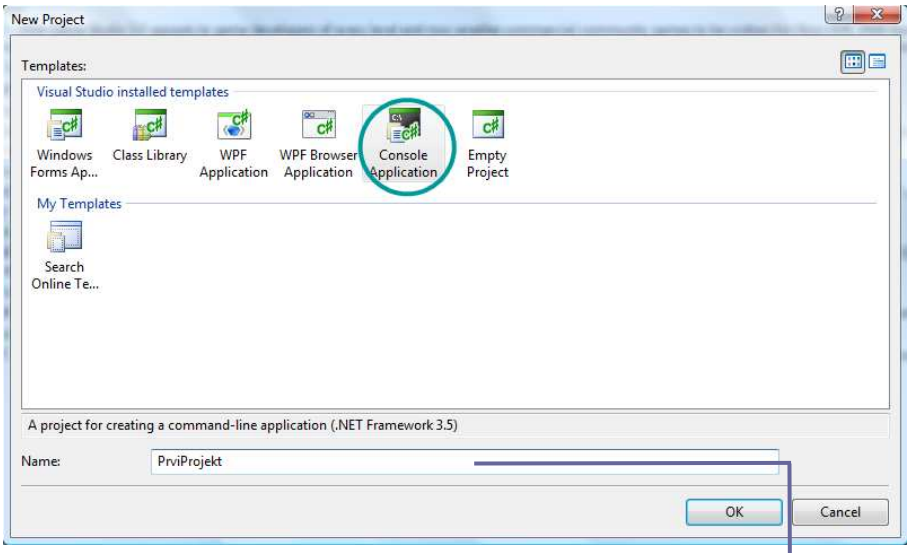
Za spoznavanje osnov programskega jezika C# bomo uporabljali več ali manj tip projekta *Console Application*. V nadaljevanju bomo opisali, kako tak projekt ustvarimo. Pri tem se bomo omejili le na najnujnejše funkcije, ki jih okolje ponuja. Nekaj več o tem lahko zveste med drugim tudi v Uranič S., Microsoft C#.NET, dostopno na naslovu http://uranic.tsckr.si/C%23/C%23.pdf

2.2 RAZVOJ NOVEGA PROJEKTA V RAZLIČICI EXPRESS EDITION

Po zagonu **MVCEE** (Microsoft Visual C# 2008 Express Edition)) se odpre začetna stran okolja. V zgornjem levem delu je okno, v katerem je seznam projektov, s katerimi smo nazadnje delali. Preostala okna v tem okolju nam služijo za pomoč pri delu in za informiranje o novostih, povezanih z okoljem MVCEE.

Slika 1: Razvojno okolje MVCEE (Microsoft Visual C# 2008 Express Edition)

Za začetek novega projekta v okolju Visual C# Express Edition odprimo meni *File -> New Project…* . Odpre se okno *New Project*. V tem oknu med *Templates* (vzorci) izberemo (kliknemo) tip projekta *Console Application* ter določimo ime (*Name*) projekta.



Slika 2: Začetek pisanja prve konzolne aplikacije

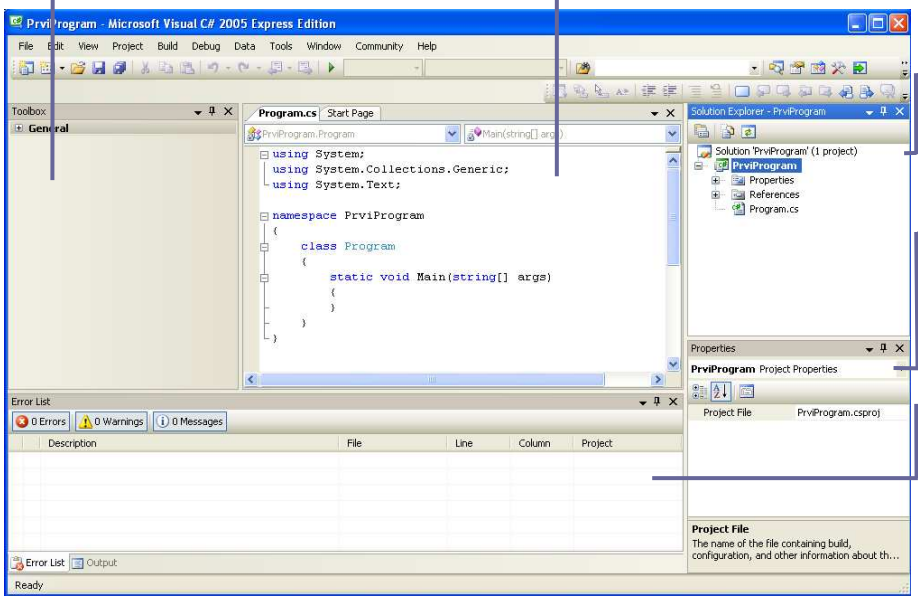
Na dnu okna napišimo še ime naše prve konzolne aplikacije, npr. **Prvi Projekt**. S klikom na gumb OK se ustvari nov projekt.

2.3 PISANJE PROGRAMSKE KODE

Ko smo ustvarili nov projekt, se odprejo osnovna okna.

Toolbox (objekti za obrazce)

Editor (urejevalnik -okno za pisanje/urejanje kode)



Solution Explorer (raziskovalec rešitve)

Properties (lastnosti objektov)

Error list (seznam napak), ali pa **Task list** (seznam opravil, potrebnih za dokončanje projekta)

Slika 3: Najpomembnejša okna razvojnega okolja

Pogljemo si kratek opis osnovnih delov razvojnega okolja:

- **Editor** ali **urejevalnik** je okno, namenjeno pisanju in urejanju programske kode. Pri pisanju daljših delov programov je zelo priročna načrtovalna črta (*outlining*). Črto sestavljajo + in – na levi strani urejevalnika. Kliki na te oznake omogočajo skrivanje in prikazovanje delov kode. Pri tem so sestavni deli logični deli kode, kot so posamezne metode, razredi in podobno. V primeru, da je del kode skrit (+ na levi), je naveden le njegov povzetek (začetek, ki mu sledijo …). Pregled vsebine lahko vidimo tako, da se z miško premaknemo na ta povzetek. Če na + kliknemo, se koda prikaže. Obratno s klikom na – kodo skrijemo.

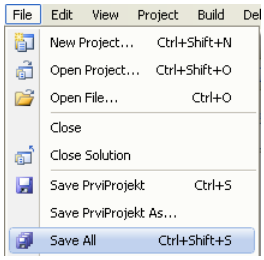
- **Soution Explorer** ali **raziskovalec rešitve** je okno, ki prikazuje in omogoča dostop do vseh datotek znotraj projekta. Datoteke lahko dodajamo, jih brišemo ali spreminjamo.

- **Properties** ali **lastnosti objektov** je okno, v katerem lahko spreminjamo in nastavljamo lastnosti objektov. Pri izdelavi konzolne aplikacije je to okno prazno in ga ne potrebujemo.

- **Output** ali **izpis** je okno, kjer dobimo sporočila o morebitnih napakah ali opozorilih, ki so se pojavili med preverjanjem kode.

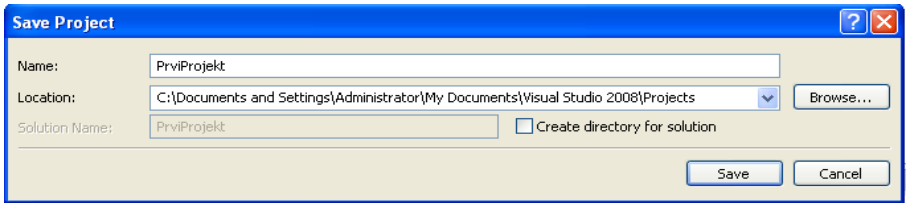
Kot vidimo, že samo okolje v skladu z izbranim tipom projekta zgradi osnovno "okostje" programa. V to "okostje" lahko začnemo zapisovati kodo programa.

Priporočljivo je, da nov projekt takoj shranimo. To storimo tako, da se postavimo na meni *File* in kliknemo izbiro *Save All*.



Slika 4: Meni za shranjevanje projekta

Prikaže se okno *Save Project*. V tem oknu določimo imenik (*Location*), kjer bodo datoteke novega projekta. Poleg imenika lahko v tem oknu določimo tudi novo ime projekta. To naredimo tako, da v okence *Name* vnesemo novo ime.



Slika 5: Okno za shranjevanje projekta

Sedaj v pripravljeno okostje programa dodamo našo kodo. Nato ponovno shranimo vse skupaj, pri čemer seveda ponujenih nastavitev (ime, imenik, …) ni smiselno spreminjati. Nato moramo kodo programa samo še prevesti in zagnati.

2.4 OKOSTJE PROGRAMOV

Ko v okolju Microsoft Visual C# 2008 Express Edition ustvarimo nov projekt, se v oknu *Editor* prikaže osnovna zgradba programa v jeziku C#.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
   class Program
   {
      static void Main(string[] args)
      {
      }
   }
}
```

Pomen posameznih delov v tem trenutku ni pomemben. Važno je le to, da si zapomnimo, da moramo vse, kar bomo pisali, napisati znotraj metode *Main*, med zavite oklepaje.

```
static void Main(string[] args)
{
    Console.WriteLine("Pozdravljen svet!");
}
```

S tem smo napisali ukaz, ki na zaslon izpiše sporočilo kot je med dvojnimi narekovaji. Pri pisanju lahko opazimo zanimivo lastnost urejevalnika, ki nam je lahko v veliko pomoč. Ko za besedo *Console* napišemo piko, se nam odpre t.i. meni *IntelliSense* (kontekstno občutljiva pomoč). Le-ta nam prikaže ustrezne možnosti, ki so v tem trenutku na voljo. S puščicami (če je možnosti preveč, se splača natipkati še prvih par črk ukaza, v našem primeru ukaza *WriteLine*) izberemo ustrezno izbiro in pritisnemo na <Enter> ali dvakrat kliknimo z miško na metodo - metoda je s tem dodana programski kodi. Dodana sta že oklepaj in zaklepaj. Znotraj oklepaja v narekovajih napišimo besedilo, ki nam ga bo program izpisal na zaslon - v našem primeru bo to besedilo "Pozdravljen svet!". Za oklepajem dodajmo še podpičje, saj s tem zaključimo ta ukaz (dvopičje je znak za konec stavka).

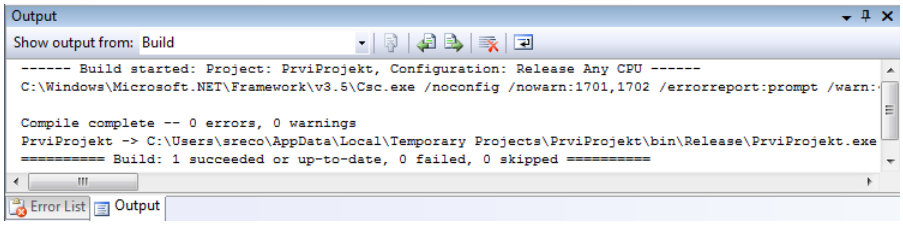
Paziti moramo, da napišemo črke metod točno tako, kot je navedeno, saj je C# "občutljiv" glede uporabe velikih in malih črk.

2.5 PREVAJANJE IN ZAGON PROGRAMOV

Pred zagonom programa moramo program prevesti. Poglejmo, kako to naredimo v okolju Microsoft Visual C# 2008 Express Edition.

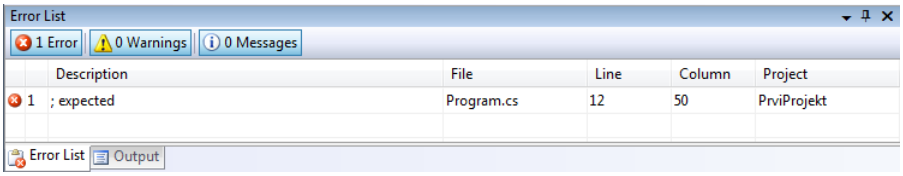
2.5.1 Prevajanje programov

Program po zapisu prevedemo. To storimo tako, da se postavimo na meni *Build* in izberemo *Build Solution*. Po končanem prevajanju se v oknu *Output* izpiše povzetek.



Slika 6: Okno *Output* s povzetkom naše rešitve

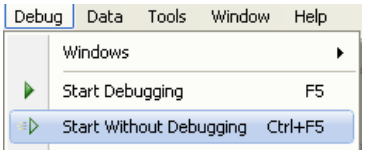
V povzetku je zapisan podatek o številu napak (*errors*). Če je število napak 0, je program sintaktično pravilen. V primeru, da so v programu napake, se prevajanje prekine in prevajalnik nam v oknu *Error List* sporoči opis sintaktične nepravilnosti in številko vrstice, kjer se napaka nahaja.



Slika 7: Okno Error List

2.5.2 Zagon programa

Program, ki je preveden brez sintaktičnih napak, lahko zaženemo. To storimo tako, da se postavimo na meni *Debug* in izberemo opcijo *Start Without Debugging*.



Slika 8: Zagon programa

S tem zaženemo program. Če smo dovolj hitri, bomo opazili, da se prikaže konzolno okno, v katerem se prikazujejo rezultati napisanega programa. Ko se namreč program izvede do konca, takoj zapre konzolno okno, ki ga je prej odprl, da bi izpisal rezultate. To nas seveda moti. Zato dopolnimo program tako, da bo tik pred zaključkom "počakal" na pritisk na poljubno tipko. Na koncu vsakega programa (torej tik pred obema *}*) bomo torej napisali še

```
Console.ReadKey();
```

Če pa želimo, da bi se ob zaključku programa na ekranu izpiše še obvestilo, da je program končan, potem zapišemo takole:

```
static void Main(string[] args)
{
```

```
    Console.WriteLine("Pozdravljen svet!");
    Console.WriteLine("Konec programa. Pritisni poljubno tipko ..." );
    Console.ReadKey();
}
```

Zadnji ukaz pa zahteva pritisk poljubne tipke. Ko jo pritisnemo, je tudi ta ukaz, s tem pa tudi program končan. Zato se konzolno okno takrat zapre.

Slika 9: Izgled konzolnega okna

Povzemimo in si še enkrat poglejmo, kako dosežemo, da se naš projekt prevede v izvršno datoteko .exe oz. samostojen program. Kliknimo na meni *Build* → *Build Solution*. Če se bo naša programska koda pravilno prevedla, se bo v statusni vrstici skrajno levo spodaj izpisalo »Build succedded«. Izvršna datoteka se nahaja v podmapi *bin\Debug* tiste mape, kamor smo shranili naš projekt. Da pa nam vsakič, ko želimo prevesti kodo, ni potrebno odpirati izvršne datoteke, našo aplikacijo lahko zaženemo preprosto s klikom na gumb z zeleno puščico, ki je približno pod izbiro *Tools*. Pokaže se okno aplikacije. Tega lahko zapremo, minimiziramo, maksimiziramo ali razširjamo. Izvajanje aplikacije prekinemo s klikom na gumb z modro obarvanim štirikotnikom (oz. Izbira *Debug* → *Stop Debugging*).

2.6 NAPAKE PRI PREVAJANJU

Pri pisanju kode se seveda lahko zgodi, da naredimo kakšno napako. V svetu programiranja napakam v programu rečemo tudi hrošči (*bugs*). Poznamo tri vrste napak:

- Napake pri prevajanju (**Compile Time Error**);
- Napake pri izvajanju (**Run Time Error**);
- Logične napake (**Logical Error**).

Prvo vrsto napak odkrije že prevajalnik in nas nanje opozori. Dokler ima program te, sintaktične napake, prevajalnik programa tako ali tako sploh ne bo prevedel. Pri napakah med izvajanjem se program zaustavi (ali kot tudi rečemo, sesuje). A običajno tudi takrat dobimo ustrezno obvestilo, s katerim si pomagamo pri odpravljanju tovrstnih napak. Bolj problematične so logične napake. Te napake so pomenske, njihovo odpravljanje pa zaradi tega veliko težje in dolgotrajnejše.

Na primeru naše prve konzolne aplikacije si poglejmo prikaz in odpravljanje prvih dveh vrst napak. Recimo, da smo se pri pisanju programske kode zmotili in namesto pravilnega zapisa

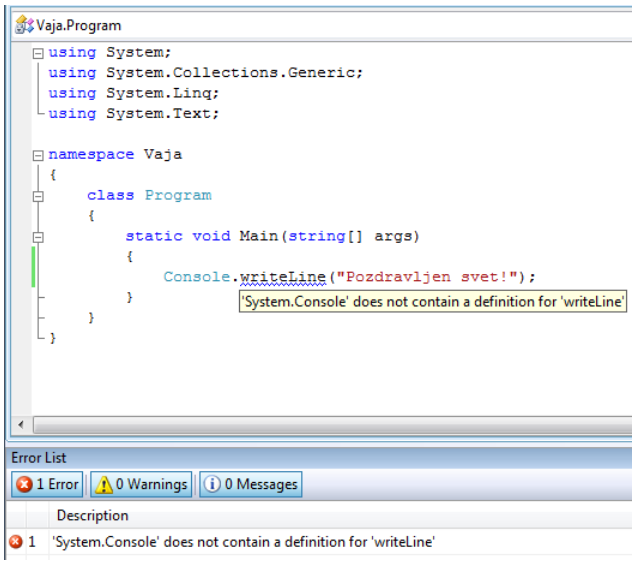
```
Console.WriteLine("Pozdravljen svet!");
```

zapisali stavek **WriteLine** z malo začetnico takole:

```
Console.writeLine("Pozdravljen svet!");
```

Ko poženemo prevajanje s klikom na opcijo *Debug → Start Debugging* (tipka F5) ali pa *Start Without Debugging* (Ctrl-F5), nam Visual C# v oknu *Error list* (seznam napak) prikaže za kakšno napako gre in kje se nahaja (oz. kakšne so napake in kje se nahajajo, če jih je več). Taki vrsti napake pravimo napaka pri prevajanju (*Compile Time Error*), ker se naš program zaradi napake sploh ni mogel prevesti.

Poleg tega nam Visual C# podčrta del kode, kjer se napaka nahaja. Če se z miško premaknemo nad podčrtani del kode, kjer je napaka, nam Visual C# v okvirčku pod to besedo izpiše za kakšno napako gre.



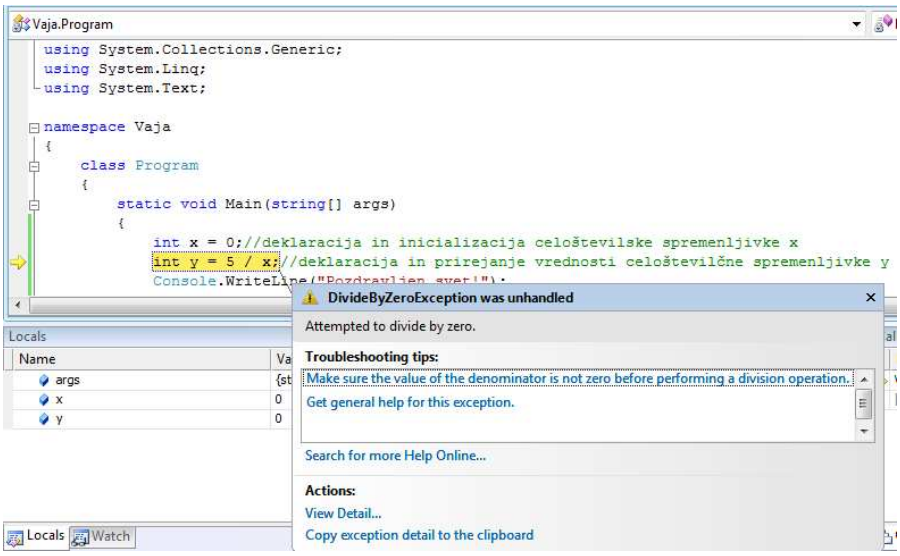
Slika 10: Opis napake v programu

Opozoriti velja, da napaka ni nujno točno na tistem mestu, kjer prevajalnik misli da je. Napako smo lahko storili že prej. Zato se takrat, ko se nam na označenem mestu zdi, da je vse zapisano prav, splača pogledati od tam proti začetku programa.

Zgodi se tudi, da se naš program prevede, a do napake pride pri samem izvajanju programa. Kot primer take napake poglejmo klasično napako, ko delimo z nič. V našo začetno aplikacijo dodajmo še dva stavka takole:

```
int x = 0;  // deklaracija in inicializacija celoštevilske spremenljivke x
int y = 5 / x;  // deklaracija in prirejanje vrednosti celošt.spremenljivke y
Console.WriteLine("Pozdravljen svet!");
```

Ko poženemo prevajanje, se program sicer prevede in se prične izvajati. V stavku *int y = 5 / x* pa smo zahtevali deljenje z 0 (ker je pač vrednost spremenljivke x enaka 0), kar pa je strogo prepovedana operacija. Izvajanje programa se zaradi tega ustavi in na ekranu dobimo približno takole sliko(Slika 11) z obvestilom o napaki. Taki napaki pravimo napaka med izvajanjem programa (*Run Time Error*). Program moramo seveda popraviti tako, da se izognemo takim operacijam.



Slika 11: Napaka med izvajanjem programa

Preden nadaljujemo, si oglejmo zanimivo napako. Gremo v urejevalnik, poskusimo prevesti in zagnati program, sistem pa javi napako. Gledamo, gledamo, a napake ne vidimo. Vse je videti OK. A pri poskusu prevajanja sistem vztraja in izpisuje napako. Vzrok je enostavno ta, da smo pozabili zapreti konzolno okno. Zato se prejšnje različica tega programa "še izvaja" in datoteke *exe* ne moremo prepisati z novo različico.

Če torej sistem javi tako napako, preverite, če nimate slučajno odprtega kakega nepotrebnega konzolnega okna!

Sedaj, ko smo spoznali osnove dela v tem okolju, je zadnji čas, da si tudi na svojem računalniku namestite to okolje. Na spletu poiščite ustreznen namestitveni program (bodisi za MSCSEE, bodisi za SharpDevelop) in si ga namestite na svoj računalnik. Tako boste pripravljeni, da napišete svoje prve programe.

2.7 KOMENTARJI

Komentarji so na poseben način označeni deli besedila, ki niso del programske kode. V komentarje zapisujemo razne opazke ali pa jih uporabljamo za lažje iskanje delov programa in za izboljšanje preglednosti programske kode. Prevajalnik komentarje ne prevaja, tako da ti ne vplivajo na velikost izvršne datoteke. Komentarje v C# označujemo na dva načina:

- s paroma znakov /* in */ - večvrstični komentar;
- z dvema poševnicama // - enovrstični komentar;

```
// To je enovrstični komentar;

/*
 To pa je
  večvrstični
  komentar!
```

```
*/
```

Vsak "spodoben" program vsebuje komentarje. Na ta način si olajšamo razumevanje programa, iskanje napak in kasnejše morebitno spreminjanje programa. Komentarje pišemo sproti ob kodi in vedno napišemo kaj v določenem delu programa želimo narediti in zakaj. Kvalitetni komentarji so zelo pomembni, zato bomo večkrat omenili, kako jih napišemo.

3 IZPISOVANJE NA ZASLON

3.1 UVOD

Sedaj, ko vemo, kako okvirno napišemo program, si oglejmo nekaj enostavnih programov. Pri tem bomo spustili zgornje vrstice (z *using* in *namespace*), saj nam bodo vedno ustrezale točno take, kot jih ponudi že samo okolje. Prav tako ne bomo navajali zadnje vrstice z *}*. Če nam torej okolje ponudi

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Vaja
{
   class Program
   {
      static void Main(string[] args)
      {

      }
   }
}
```

bomo v naših zgledih temneje označeni del spustili in napisali le

```
static void Main(string[] args)
{

}
```

ali podobno. Če bo potrebno (zaradi razlage), bomo navedli še številke vrstic, ki jih seveda ne pišemo. Prav tako bomo običajno spustili še dodatek

```
Console.WriteLine("Konec programa. Pritisni poljubno tipko ...");
Console.ReadKey();
```

ki poskrbi, da se konzolno okno ne zapre prehitro.

3.2 IZPISOVANJE NIZOV

Osnovni način izpisovanja na zaslon je z metodo *WriteLine()* oziroma z metodo *Write()*. Obe metodi izpišeta tisto, kar je podano med narekovaji (") znotraj okroglih oklepajev. V našem prvem programu se je tako izpisalo *Pozdravljen, svet!*. Edina razlika med metodama *WriteLine()* in *Write()* je, da prva po izpisu prestavi izpisno mesto (položaj začetka naslednjega izpisa) na začetek nove vrstice, pri drugi pa ostane izpisno mesto v isti vrstici desno od zadnjega izpisanega znaka.

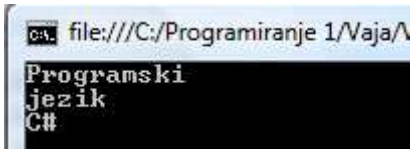
Spremenimo naš prvi program tako, da bo namesto *Pozdravljen, svet!* izpisal *Programski jezik C#*. Spremeniti moramo le tekst med znakoma " pri metodi *WriteLine()*.

```
static void Main(string[] args)
{
    //  Program, ki na zaslon izpiše niz "Programski jezik C#".
    Console.WriteLine("Programski jezik C#");
}
```

Kako pa bi napisali vsako besedo v svojo vrstico?

```
static void Main(string[] args)
{
    Console.WriteLine("Programski");
    Console.WriteLine("jezik");
    Console.WriteLine("C#");
}
```

Prevedemo in poženemo:



Slika 12: Izpis več besed

Nize združujemo (stikamo) s tako imenovanim združitvenim operatorjem '+'. Na ta način pogosto razdelimo nize v bolj pregledne, krajše nize. Pogosto pri oblikovanju izpisov uporabimo tudi poseben znak *\n* – ta znak povzroči preskok v novo vrsto. Naslednji izpis bo zato identičen tistemu na sliki 13.

```
Console.WriteLine("Programski\njezik\nC#");
```

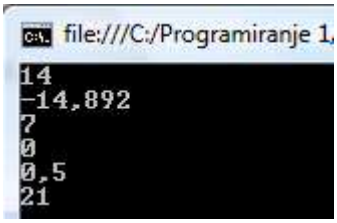
Podobno z \" dosežemo, da je tudi " lahko sestavni del niza, z \\ pa v niz zapišemo \. V kodi jezika C# lahko vsak znak (tudi znake '\', narekovaj in dvojni narekovaj) zapišemo tudi kot običajen znak, če le pred nizom uporabimo znak '@'. S tem povemo, da se morajo vsi znaki v nizu jemati dobesedno. Torej sta @"C:\dat\bla.txt" in "C:\\dat\\bla.txt" enaka niza, le drugače zapisana.

3.3 IZPISOVANJE ŠTEVIL

V prejšnjih primerih smo si pogledali, kako izpisujemo nize. Poglejmo še nekaj primerov, kako bi izpisali števila.

```
static void Main(string[] args)
{
    Console.WriteLine(14);
```

```
    Console.WriteLine(-14.892);  // decimalno ločilo je pika
     // vrednost izraza se računa tako kot v matematiki
    Console.WriteLine(1 + 2 * 3);
    Console.WriteLine(1 / 2);  // celoštevilsko deljenje
    Console.WriteLine(1.0 / 2);  // "pravo" deljenje
    Console.WriteLine((1+2)*(3+4)); // oklepaji imajo prioriteto
}
```



Slika 13: Izpisovanje števil

Kot vidimo, lahko s pomočjo programov tudi kaj izračunamo! Pri številih uporabljamo decimalno piko, za množenje je potrebno uporabiti *. Deljenje z operatorjem / je včasih "čudno" (1/2 je 0, 1.0/2 pa 0.5). Podrobnosti si bomo ogledali kasneje, ko si bomo ogledali t.i. podatkovne tipe.

Včasih pa pri izpisovanju »mešamo« nize in števila

```
Console.WriteLine("Starost: " + 42);
```

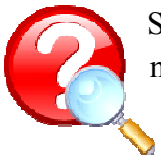
Dobimo izpis Starost: 42. Če namreč »seštevamo« niz in število, se število pretvori v ustrezen niz, nato pa se niza stakneta. Če napišemo *"12"* + *"13"* bomo dobili niz *"1213"*. Če pa + uporabimo med števili (12 + 13), ima učinek, ki smo ga navajeni iz računstva – sešteje števili. Operator *'+'* torej nastopa v dveh vlogah – kot operator seštevanja in za stikanje (združevanje) nizov. Če sta oba operanda števili ali če sta oba operanda niza, je pomen jasen. Če pa je eden od operandov niz in drugi število, se število pretvori v niz (*12* v niz *"12"*) in *'+'* je operator združevanja nizov.

```
Console.WriteLine("Stevilo" + 1 + 2);  //  Izpis: Stevilo12
Console.WriteLine(1 + 2 + "Stevilo");  //  Izpis: 3Stevilo
Console.WriteLine(1 + "Stevilo" + 2);   //  Izpis: 1Stevilo2
```

3.4 SEMANTIČNE NAPAKE

Sintaktično pravilen program pa ima lahko t.i. semantične (pomenske) napake. Program je napisan pravilno (prevajalnik se ne pritoži) in program lahko izvedemo, a rezultati niso taki, kot jih pričakujemo. Naslednji vrstici sta sicer sintaktično pravilni, a izpis pomensko napačen.

```
Console.WriteLine("Vsota stevil 2 in 3 je " + 2 * 3); //  Izpis: Vsota števil 2 in 3 je 6
Console.WriteLine("Vsota števil 2 in 3 je " + 2 + 3);  //  Izpis: Vsota števil 23
```



Sedaj je že zadnji čas, da tudi sami napišete nekaj programov. Izberite si naloge z naslova http://penelope.fmf.uni-lj.si/C_sharp/index.php/Vaje/Vaje2007_8/1VajeLj in samostojno rešite nekaj nalog. Prav tako pa so na voljo naloge iz začetnega poglavja zbirke nalog Lokar M. in Uranič S., Programiranje 1 - zbirka rešenih nalog v C#.

3.5 OBLIKA PROGRAMOV

Sedaj smo napisali že nekaj programov. Pri tem smo vse pisali v določeni obliki. Tako smo na določenih mestih uporabili presledke, prehode v novo vrsto, zamikanje ali pa znake okoli operatorjev.

Pisanje programov "lepo" ima določen pomen. Je sicer povsem nepomembno za prevajalnik, a zelo pomembno za človeka. Če programe pišemo na določen način (uporabljamo določen stil), se v programu lažje znajdemo pa še koda nam je bolj pregledna.

Tako je spodaj napisan sintaktično pravilen, a nepregleden program:

```
static void Main(string[] args
) {
      Console.WriteLine
("Vsota števil 2 in 3 je " + (2
* 3));Console.WriteLine("Konec ...");     Console.ReadKey()
; }
      }
```

Ta isti program lahko napišemo pregledneje:

```
static void Main(string[] args)
{
   Console.WriteLine("Vsota števil 2 in 3 je " + (2+ 3));
   Console.WriteLine("Konec ...");
   Console.ReadKey();
}
```

3.6 POVZETEK

Pokazala sva le osnovno uporabo metod *Write()* in *WriteLine()*. Med oba oklepaja je potrebno le zapisati poljuben string, ki ga želimo izpisati oziroma številski izraz, ki ga želimo najprej izračunati in nato izpisati. Pogosto pa želimo izpis tudi oblikovati, oziroma ga formatirati. V takih primerih lahko uporabimo drug način uporabe teh dveh stavkov in to tako, da znotraj okroglih oklepajev uporabimo poseben sistem izpisovanja in oblikovanja spremenljivk, ki vključuje uporabo zavitih oklepajev. Več o taki uporabi stavkov *Write()* in *WriteLine()* pa je zapisano npr. v gradivu Uranič S., Microsoft C#.NET, (online). Kranj, 2008. Dostopno na naslovu *http://uranic.tsckr.si/C%23/C%23.pdf*, str. 28 − 31.

4 SPREMENLJIVKE IN PODATKOVNI TIPI V C#

4.1 UVOD

Programi upravljajo s podatki, ki so shranjeni v pomnilniku. Zato moramo vedeti, kje ti podatki so. V C# se za naslove podatkov uporabljajo imena. Takšno ime (naslov), ki služi za klic podatkov shranjenih v spominu, se imenuje **spremenljivka**.

Spremenljivko si lahko predstavljamo kot tablo, na kateri je zapisan nek podatek. Spremenljivka se neposredno nanaša na tablo in posredno na podatek. Med izvajanjem programa se podatek na tabli lahko spreminja. Na tabli (v spremenljivki) bodo različne vrednosti (podatki), toda vedno bo to ista tabla. Vedno lahko pogledamo, kakšno vrednost ima podatek na tabli.

4.2 IMENA SPREMENLJIVK

Vsem spremenljivkam moramo določiti ime, da se lahko kasneje nanj sklicujemo in tako pridemo do vrednosti, ki jo posamezna spremenljivka vsebuje. Imena spremenljivk morajo biti smiselna: iz imena naj bo razvidno, kaj pomeni podatek, ki ga hranimo v spremenljivki. Uporaba takih imen nam olajša razumevanje programske kode.

Prva črka v imenu spremenljivke naj bo poljubna **mala** črka abecede[1.] Tej črki sledi poljubno zaporedje črk (malih in velikih), števk in podčrtajev (_). Tako so imena npr. *vhodni_niz, niz1, niz_2* ipd.

Če je ime sestavljeno iz več besed, ne smemo uporabljati presledkov. Zato ga zaradi boljše čitljivosti zapišemo bodisi tako, da vsako naslednjo besedo začnemo z veliko črko (*dolgoIme*), ali pa besede povežemo s podčrtajem *(dolgo_ime)*. Znotraj programa zaradi preglednosti uporabljajmo samo en način.

Ne pozabimo, da C# loči med malimi in velikimi črkami. Zato so *ime*, *Ime*, *iMe*, *IME* štiri različna imena.

Pri imenovanju spremenljivk moramo upoštevati določena pravila. Tu bomo navedli zelo poenostavljeno različico teh pravil:

- prvi znak mora biti črka;
- ime je lahko sestavljeno le iz črk angleške abecede[2] in števk;
- ne sme biti enako drugemu imenu;
- ne sme biti enako rezerviranim besedam.

Pri imenih se upošteva (razlikuje), ali je črka mala ali velika. Ime je sestavljeno iz ene besede. Če je besed več, jih zlepimo. Zlepimo jih tako, da naslednjo besedo začnemo z veliko

[1] Pravila jezika C# dopuščajo določene druge znake, a ... .
[2] Zaradi določenih težav z nekaterimi orodji se šumniki praviloma ne uporabljajo, čeprav C# podpira poljubne znake v kodi UNICODE.

začetnico, npr. *vsotaSodihStevil*. Ne uporabljamo imen, ki se med seboj razlikujejo le v uporabljenih velikih oz. malih črkah. Tako ni dobro, da bi v programu imeli tako spremenljivko *Vsota* kot tudi spremenljivko *vsota*.

Vsako spremenljivko pred prvo uporabo najavimo, ali kot rečemo, **deklariramo**. S tem v pomnilniku rezerviramo prostor ustrezne velikosti, ki je določen glede na tip spremenljivke. Kaj je tip, si bomo ogledali v nadaljevanju. Zaenkrat povejmo le, da z njim določimo, kakšne vrednosti lahko hranimo v spremenljivki.

Vsako v programu uporabljeno spremenljivko moramo najaviti oziroma deklarirati.

4.3 DEKLARACIJSKI STAVEK

Spremenljivke najavimo z **deklaracijskim stavkom**. Ta je oblike

podatkovniTip imeSpremenljivke;

Primeri:

```
int stevilo;
double a, b, c;
string niz;
```

Spremenljivki ob najavi lahko priredimo začetno vrednost.

podatkovniTip spremenljivka1 = začetnaVrednost;

Primeri:

```
int stevilo = 0;
char znak = 'A';
string niz = "Vhodni niz";
```

Spremenljivko deklariramo samo enkrat. Kot smo omenili, to običajno naredimo na samem začetku, ni pa to nujno. Obvezno pa mora biti spremenljivka deklarirana pred prvo uporabo, drugače nam prevajalnik javi napako.

4.4 PRIREDITVENI STAVEK

Vrednost, zapisano v spremenljivki, spreminjamo (ali nastavimo prvič, če tega ob deklaraciji nismo naredili) s pomočjo prireditvenega stavka. Prireditveni stavek vsebuje prireditveni operator (=). Ta priredi vrednost desne strani spremenljivki na levi strani.

Prireditven stavek je oblike

spremenljivka = izraz;

Tu je *izraz* lahko konstanta, ki jo želimo prirediti spremenljivki ali pa predstavlja izraz z operatorji, metodami, ...

Najprej se izračuna vrednost izraza. Dobljena vrednost se shrani v spremenljivko. Če spremenljivka nastopa v izrazu, pomeni, da vzamemo vrednost, ki jo hranimo v spremenljivki. Denimo, da v spremenljivki *x* hranimo število.

```
y = 3 * x + 5;
```

Pomen tega stavke je:

Izračunamo izraz: 3 krat število, ki je shranjeno v *x* in to povečamo za 5. Dobljeni rezultat shranimo v *y*. Seveda smo morali prej obe spremenljivke (*x* in *y*) deklarirati in povedati, da v njih hranimo števila.

Če torej uporabimo ime spremenljivke na desni strani (v izrazu), mislimo na vrednost, shranjeno v tej spremenljivki. Tako *a* = *b* ne pomeni, da je *a* enako *b*, ampak da vrednost shranjeno v *b* priredimo *a*–ju. V spremenljivko *a* torej shranimo vrednost, kot je v spremenljivki *b*.

Če napišemo

```
x = x + 1;
```

to pomeni, da vrednost shranjeno v *x* povečamo za 1. Zakaj? Poglejmo, kako se ta stavek izvede. Najprej izračunamo izraz, torej tisto, kar je shranjeno v *x*, povečamo za 1. Dobljeni rezultat shranimo v *x*. To, da je prej pri računanju nastopala spremenljivka *x*, sploh ni pomembno.

4.5 IZPIS SPREMENLJIVK

Kaj se zgodi, če napišemo

```
Console.WriteLine(x);
```

kjer je *x* neka spremenljivka, v kateri hranimo število?

Ukaz pomeni: izpiši vrednost izraza. Vrednost izraza *x* pa je vrednost spremenljivke *x*, torej število. Že prej pa smo videli, da se ob izpisovanju število pretvori v niz (zapis števila kot zaporedja znakov). Zato gornji ukaz izpiše kot niz tisto število, ki je v tem trenutku v spremenljivki *x*.

Če napišemo

```
Console.WriteLine(x + 1);
```

se najprej izračuna vrednost izraza (tisto, kar je v *x*, povečamo za 1). Dobljena vrednost je število. To se pretvori v niz, ki se izpiše.

Seveda lahko v izrazih kombiniramo nize, spremenljivke in števila. Če torej napišemo

```
int x = 10;
```

```
Console.WriteLine (x + " + " + " 1 = " + x + 1);
```

bomo dobili kot izpis

10 + 1 = 101

Še vemo zakaj? Če pa bi napisali

```
int x = 10;
Console.WriteLine (x + " + " + " 1 = " + (x + 1));
```

bi bili izpis verjetno bolj pričakovan

10 + 1 = 11

4.6 ZGLEDI

4.6.1 Pleskanje stanovanja

To jesen smo porabili veliko časa in denarja za preurejanje stanovanja in do popolnosti manjka le še na novo prepleskana dnevna soba. Ta meri 6 x 3 m. Je pa precej visoka, kar 3 m. V sobi sta tudi dve veliki okni s skupno površino 5 m^2 (ki jih ne bomo pleskali in moramo njihovo površino odšteti od površine vseh sten in stropa). Z enim kilogramom barve prepleskamo 8 m^2. Koliko kilogramov barve moramo najmanj kupiti, če nameravamo nanesti dva nanosa barve?

Namig: Najprej bomo izračunali površino vseh sten in stropa. Od tega bomo odšteli površino oken. To bomo izpisali. Nato bomo površino množili s številom nanosov in delili z 8. Za vsak primer bomo še prišteli 1 l barve.

Rešitev:

```
static void Main(string[] args)
{
    //  podatki
   int visina = 3;
   int sirina = 3;
   int dolzina = 6;
   int povrsinaOken = 5;
   int prekrivnostBarve = 8;  //  koliko m2 z enim litrom
   int steviloNanosov = 2;

    // izračun površine za barvanje
   int povrsina = dolzina * sirina +  // strop
      visina * sirina * 2 +  // dve ožji steni
      visina * dolzina * 2  // dve širši steni
      - povrsinaOken;  //  oken ne barvamo

   Console.WriteLine("Površina sten za barvanje: " + povrsina);
```

```
    // izpis in izračun količine barve
    int kolicinaBarve = povrsina * steviloNanosov / prekrivnostBarve + 1;
    //  +1 - da zaokrožimo navzgor
    Console.WriteLine("Potrebna količina barve za dva premaza: " +
                      kolicinaBarve + " litrov");

    Console.ReadKey(); // da se konzolno okno ne zapre prehitro
}
```

4.6.2 Hišnik čisti bazen

Hišnik mora vsak mesec očistiti bazen. Da ga lahko očisti, mora najprej iz njega izčrpati vodo. Ker je bolj lene sorte, bi med iztekanjem vode (ki traja kar nekaj časa) raje odšel v bližnjo kavarno na pogovor s prijateljem, namesto da bi stražil bazen. Napišimo mu program, ki bo za bazen velikosti 2.3 m x 1.6 m x 9.5 m izračunal, koliko sekund se bo praznil bazen, če vsako sekundo izteče iz bazena 23 l vode. Če si že pozabili: 1 dm^3 vode = 1 l vode.

Namig: Izračunali bomo volumen bazena kar v kubičnih decimetrih. Če bomo volumen delili s 23, bomo dobili število sekund.

Rešitev:

```
public static void Main(string[] args)
{
    Console.WriteLine("Praznjenje bazena");

    int sirina = 23;  // mere v dm
    int globina = 16;
    int dolzina = 95;
    int prazni = 23;  //  koliko litrov na sekundo

    int volumen = sirina * globina * dolzina;
    int sekunde = volumen / prazni;  //  koliko sekund se prazni

    Console.WriteLine("\n   Bazen se prazni " + sekunde + " sekund.");

    Console.Write("\n\nTipka za nadaljevanje . . . ");
    Console.ReadKey();
}
```

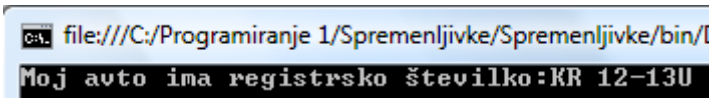
4.7 PODATKOVNI TIPI

Podatkovni tip pove prevajalniku, koliko pomnilnika naj rezervira za spremenljivko in katere so operacije, ki jih lahko s temi spremenljivkami izvajamo.

4.7.1 Nizi

Če želimo kak niz (torej zaporedje znakov med dvema dvojnima narekovajema ") shraniti v spremenljivko, to spremenljivko deklariramo, da je tipa **string**. Podatkovni tip *string* (niz) torej označuje zaporedje znakov.

```
string regObmocje = "KR";
string stevilka = "12-13U";
string registrska = regObmocje + " " + stevilka;
Console.WriteLine("Moj avto ima registrsko številko:" + registrska);
```



Slika 14: Izpis spremenljivk

4.7.2 Cela števila

Cela števila (ang. *integer*) označimo s tipom **int**. Seveda ne gre za poljubna cela števila, kot jih poznamo iz matematike, saj v končni pomnilnik računalnika ne moremo shraniti neskončno velikega števila.

Nad celimi števili lahko izvajamo vse osnovne računske operacije (seštevanje, odštevanje, množenje in deljenje). Za množenje uporabljamo znak *. Vsota, razlika in produkt dveh celih števil so definirani na običajen način, deljenje pa je celoštevilsko. To pomeni, da pri deljenju dveh celih števil C# zanemari vsa decimalna mesta (tako je kvocient dveh celih števil vedno celo število). Poleg deljenja pozna C# še operator %, ki vrne ostanek pri deljenju (zopet celo število).

Tabela 1: Osnovni operatorji

Operator	Pomen	Primer	Rezultat
+	vsota	3 + 2	5
-	razlika	3 - 2	1
*	množenje	3 * 2	6
/	celoštevilsko deljenje	5 / 3	1
%	celoštevilski ostanek pri deljenju	5 % 3	2

4.8 ZGLEDI

4.8.1 Trimestno število izpisano po vrsticah

Trimestno število 376 izpišimo tako, da bodo stotice izpisane v svoji vrstici, nato v svoji vrstici desetice in nato še enice. Programa napišimo tako, da bo za enak izpis nekega drugega trimestnega števila potrebno spremeniti le število 376 v drugo, ostale spremembe v programu pa ne bodo potrebne.

***Namig*:** S pomočjo celoštevilskega deljenja z 10 in 100 in ostanka pri deljenju z 10 ali 100 bomo iz števila izluščili posamezno števko.

Rešitev:

```
static void Main(string[] args)
{
    int stevilo = 376;
     //  izračunamo in izpišemo stotice
    int stotice = stevilo / 100;
    Console.WriteLine(stotice);
     //  izračunamo in izpišemo desetice
    int desetice = (stevilo % 100) / 10;
    Console.WriteLine(desetice);
     //  izračunamo in izpišemo enice
    int enice = stevilo % 10;
    Console.WriteLine(enice);
}
```

4.8.2 Obrnjeno število

Napišimo program, ki bo trimestno celo število 138 izpisal obrnjeno, torej 831.

```
1:      static void Main(string[] args)
2:      {
3:              int trimestnoStevilo = 138;
4:              int enice = trimestnoStevilo  % 10;
5:              int dvomestnoStevilo = trimestnoStevilo / 10;
6:              int desetice = dvomestnoStevilo % 10;
7:              int stotice = dvomestnoStevilo / 10;
8:              Console.WriteLine("Število " + trimestnoStevilo);
9:              Console.WriteLine("Obrnjeno stevilo " + enice + desetice + stotice);
10:             Console.ReadKey();
11:     }
```

Kaj bo program izpisal? Pa poglejmo:

Spremenljivki *trimestnoStevilo* priredimo vrednost *138*, torej število, ki ga želimo izpisati v obratnem vrstnem redu.
Sedaj moramo priti do stotic, desetic in enic(vrstice 4. – 7.). Pomagamo si z operatorjema */* in *%*. Najlažje pridemo do enic. Izračunamo ostanek pri deljenju števila z *10* (vrstica 4). Sedaj enic ne potrebujemo več. Zato jih "odrežemo" in dobljeno shranimo v spremenljivko *dvomestnoStevilo* (vrstica 5). Desetice prvotnega števila so v novem številu enice. Te pa že znamo "izluščiti" – zopet si pomagamo z ostankom pri deljenju z *10*. Stotice lahko izračunamo bodisi iz dvomestnega števila z deljenjem z *10* (vrstica 7) ali pa iz trimestnega števila z deljenjem s *100* (*trimestnoStevilo / 100*). Na koncu še dobljene vrednosti izpišemo.
Če želimo izpisati na tak način katero druigo tromestno število, denimo 254, moramo psremeniti le 3. stavek in spremenljivki trimestnoStevilo prirediti vrednost 254.

4.8.3 Zamenjava spremenljivk

Velikokrat želimo zamenjati vrednost dveh spremenljivk med sabo. Kako to izvedemo, pokažimo na primeru.

```
int x = 4;
int y = 5;
int pomozna = x; // x shranimo v pomozna
x = y;  // y zapišemo v x
y = pomozna; // pomozna zapišemo v y
```

4.9 REALNA (DECIMALNA) ŠTEVILA

Realna števila shranjujemo v spremenljivkah tipa **double**. Sestavljena so iz celega dela in decimalne vrednosti, ki ju loči **decimalna pika** (ne vejica!).

Primeri:

```
double dolzinaX = 3.4;
double cenaEnote = 1.9;
double faktorPodrazitve = 1.1;
double novaCenaEnote = cenaEnote * faktorPodrazitve;
```

Spremenljivke tipa *double* lahko zavzamejo zelo majhne (do okvirno 5.0 × 10-324) in zelo velike vrednosti (1.7 × 10308) z natančnostjo približno 15 mest. Tudi nad realnimi števili lahko izvajamo vse osnovne operacije (seštevanje, odštevanje, množenje in deljenje). Vsota, razlika, produkt in deljenje dveh realnih števil so definirani na običajen način. Če operator (+, -, /, *) povezuje celo in realno število, se celo število najprej pretvori v realno. Rezultat je takrat realno število.

4.10 FUNKCIJE – RAZRED MATH

Pogosto moramo izračunati na primer kvadratni koren nekega decimalnega števila, ali pa absolutno vrednost števila. Jezik C# pozna cel kup različnih matematičnih funkcij. Zbrane so v t.i. razredu *Math*. Razred *Math* vsebuje številne metode za uporabo matematičnih funkcij. Nekatere od njih so zbrane v spodnji tabeli:

Tabela 2: Število PI in Matematične funkcije

Konstanta	**Pomen**
PI	Iracionalno število PI.
Metoda	**Pomen**
Abs	Absolutna vrednost.
Acos	Arkus kosinus – kot (v radianih), katerega kosinus je argument metode.
Asin	Arkus sinus – kot, katerega sinus je argument metode.
Atan	Arkus tangens – kot, katerega tangens je argument metode.
Cos	Kosinus kota.
Max	Metoda vrne večjega izmed dveh števil, ki nastopata kot argument metode.
Min	Metoda vrne manjšega izmed dveh števil, ki nastopata kot argument metode.
Pow	Metoda za izračun potence.

Round	Zaokroževanje.
Sqrt	Kvadratni koren.
Tan	Tangens.
Truncate	Celi del števila (odrežemo decimalke, rezultat je celo število!).

Nekaj primerov:

```
static void Main(string[] args)
{
   int negativno = -300;
   int pozitivno = Math.Abs(negativno);
   Console.WriteLine(pozitivno);   // izpis 300

   double alfa = Math.Acos(-1);
   Console.WriteLine(alfa);   // izpis 3,141592... = iracionalno število PI

   double beta = Math.Asin(0.5);
   Console.WriteLine(beta);   // izpis 0,52359... = iracionalno število PI/6

   double gama = Math.PI / 2;
   double cosgama = Math.Cos(gama);
   Console.WriteLine(cosgama);   // izpis 0

   double vecje = Math.Max(235.8, 100.7);
   Console.WriteLine(vecje);   // izpis 235.8

   double manjse = Math.Min(235.8, 100.7);
   Console.WriteLine(manjse);   // izpis 100.7

   double eksponent = 3;
   double osnova = 5;
   double potenca = Math.Pow(osnova, eksponent);  // izračun potence:
   Console.WriteLine(potenca);   // izpis 125

   double decimalno = 245.67843;
   Console.WriteLine(Math.Round(decimalno));  // izpis 246
   Console.WriteLine(Math.Round(decimalno, 2));  // izpis 245,68
   Console.WriteLine(Math.Round(decimalno, 3));  // izpis 245,678

   double kvadrat = 625;
   Console.WriteLine(Math.Sqrt(kvadrat));  // izpis 25
}
```

4.11 ZGLEDA

4.11.1 Plačilo bencina

Bencin se je podražil za 0,2 €. Pred podražitvijo, ko je bil bencin še 1,17 €, smo za poln tank plačali 40,5 €. Izračunajmo, koliko bomo po podražitvi plačali za poln tank.

Novo ceno bomo izračunali tako, da bomo volumen tanka pomnožili z novo ceno za liter bencina. Volumen tanka dobimo tako, da staro ceno polnega tanka delimo s staro ceno za liter bencina.

```
public static void Main(String[] args)
{
   double staraCena = 1.17;
   double novaCena = staraCena + 0.2;
   double stariZnesek = 40.5;
   double noviZnesek;
   double volumenTanka = stariZnesek / staraCena;
    //  cena bencina po podražitvi
   noviZnesek = volumenTanka * novaCena;

   Console.WriteLine("Za poln tank bomo plačali " + noviZnesek + " €.");
   Console.ReadKey();
}
```

4.11.2 Plačilo mesečne vozovnice

Mesečna vozovnica za avtobus stane 68,6 €. Izračunajmo, koliko bo stala po 5-odstotni podražitvi.

```
public static void Main(String[] args)
{
  double cenaMesecne = 68.6;
  double podrazitev = 5.0 / 100;
  double novaCena;

   //  cena mesečne vozovnice po podražitvi
  novaCena = cenaMesecne + cenaMesecne * podrazitev;

  Console.WriteLine("Mesečna vozovnica bo stala " +  novaCena + " €.");
}
```

4.12 PRETVARJANJE MED VGRAJENIMI PODATKOVNIMI TIPI

Večkrat tip vrednosti ne ustreza željnemu. Takrat ga je potrebno pretvoriti v drugi tip. Včasih to stori prevajalnik, včasih pretvorbo zahteva programer. Avtomatične pretvorbe (tiste, ki jih

opravi prevajalnik) iz tipa *double* in tipa *int* v tip *string* ter iz tipa *int* v tip *double* smo že srečali v več zgledih.

4.12.1 Pretvarjanje iz tipa int v tip double

Če želimo celo število pretvoriti v realno, napišemo pred celim številom v okroglih oklepajih *double*.

(double)celoStevilo

Število tipa *int* se pretvori v število tipa *double* tudi tako, da se zadaj doda *.0* (decimalni del).

4.12.1.1 Zgled − plačilo sira

V Estoniji v trgovini stane 1 kilogram sira 1150 kron. Manja kupi 20 dekagramov sira. Koliko bo Manja plačala za sir.

```
static void Main(string[] args)
{
    int teza1 = 100;  //  teža v dekagramih
    int cena1 = 1150;  //  cena za 1 kilogram
    int teza2 = 20;  //  teža v dekagramih
    double cena2 = teza2 * cena1 / (double)teza1;       //  cena sira
    Console.WriteLine(teza2 + " dag sira v Estoniji stane " + cena2 + " kron.");
}
```

Ceno sira dobimo tako, da ceno za 1 dekagram (*cena1 / (double)teza1 = 11.5*) pomnožimo s težo kupljenega sira. Če želimo pri deljenju dveh celih števil dobiti realno število*(11.5)*, moramo vsaj eno od števil spremeniti v realno. Z *(double)teza1* smo celoštevilčno vrednost, ki je shranjena v spremenljivki *teza1,* spremenili v realno število. Ko se izračuna desni del, se priredi realni spremenljivki *cena2*.

Na koncu še izpišemo vrednosti spremenljivk *teza2* in *cena2* opremljeni z ustreznim besedilom.

4.12.2 Pretvarjanje iz tipa double v int

Če želimo realno število pretvoriti v celo, napišemo pred realnim številom v okroglih oklepajih *int*.

(int)realnoStevilo

Sprememba tipa iz realnega v celo število se opravi tako, da se preprosto odreže decimalni del.

4.12.2.1 Zgled −Višina vode

V valjast sod s premerom 100 centimetrov nalijemo 100 litrov vode. Izračunajmo, kolikšno višino doseže voda. Rezultat naj bo zaokrožen na eno decimalno mesto.

Namig: Pri računanju si pomagamo s konstanto *PI* ter metodo *Pow()*, ki ju najdemo v razredu *Math*. V konstanti *PI* je shranjena vrednost π, uporabimo jo z navedbo *Math.PI*.
Da bomo število zaokrožili na eno decimalko, število najprej pomnožimo z 10, ga zaokrožimo ter ga spremenimo v tip *int*. S tem smo odrezali odvečne decimalne vrednosti. Nato število delimo z 10 in tako dobimo število, ki je zaokroženo na eno decimalno mesto.

```
static void Main(string[] args)
{
    double volumenVode = 100000;  //  volumen vode v cm3
    double polmerSoda = 50;  //  polmer soda v cm
    double visinaVode;  //  višina vode v cm
    double ploscinaKroga;  //   ploščina sodovega dna v cm2

    ploscinaKroga = Math.PI * Math.Pow(polmerSoda, 2);
    visinaVode = volumenVode / ploscinaKroga;

     //  zaokrožimo na eno decimalno mesto
    visinaVode = (int)(Math.Round(visinaVode * 10));
    visinaVode = visinaVode / 10;
    Console.WriteLine("Voda stoji " + visinaVode + " cm visoko.");
}
```

4.12.3 Pretvarjanje iz niza (string) v celo število (int)

Če so v nizu zapisane le števke (in kot prvi znak morebiti – ali +), ga v C# lahko pretvorimo v število na dva načina:

- z metodo *int.Parse();*
- z eno od metod *Convert.ToInt16()*,*Convert.ToInt32()* in *Convert.ToInt64()*(odvisno od tega, s kako velikim celim številom imamo opravka).

Primer:

```
string niz = "2008";
int leto = int.Parse(niz); //  Vrednost spremenljivke je 2008
```

4.12.4 Pretvarjanje iz niza v realno število

Če je v nizu zapisano realno število, ga v jeziku C# pretvorimo v število z metodo *double.Parse()*, ali pa z metodo *Convert.ToDouble()*.

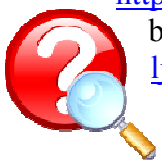
Primer:

```
 //  V nizu uporabimo decimalno vejico
string niz1 = "23,4";
double prvoStevilo = double.Parse(niz1);  //  Vrednost spremenljivke je 23,4
```

4.13 POVZETEK

V poglavju so zapisani le osnovni podatkovni tipi. Predvsem za numerične podatke vsebuje C# tudi številne druge podatkovne tipe, ki so specifični za določen tip numeričnega podatka. Tako za decimalna števila lahko uporabljamo še tip *float*, za velika cela števila tip *long* in še številne druge. Več o teh tipih, kot tudi o njihovih zalogah vrednosti, pa si lahko preberete kar v datoteki s pomočjo, ki je sestavni del razvojnega okolja.

Že z doslej osvojenim znanjem lahko rešimo številne zanimive naloge. Iz zbirke Lokar M. in Uranič S., Programiranje 1 - zbirka rešenih nalog v C# si izberite tiste iz poglavja Enostavni programi, pri katerih vas sam problem najbolj pritegne in napišite ustrezne programe. Še bistveno več nalog pa lahko najdete na WIKI DIRI 06/07. Tam so sicer naloge reševane s programskim jezikom java, a problemi so vsi taki, da jih lahko rešimo tudi z jezikom C#. In če se bo zataknilo, si boste lahko pomagali s tam objavljenimi rešitvami. Videli boste namreč, da sta si jezik java in C# dokaj podobna. Ustrezne naloge iz tega wikija najdete na naslovu http://penelope.fmf.uni-lj.si/diri0607/index.php/Kategorija:Spremenljivke. Da pa



bomo malo obnovili še »teoretično znanje« na naslovu http://up.fmf.uni-lj.si/branje-4558b1a4/index.html najdete kviz iz doslej obravnavane tematike.

5 BRANJE

5.1 UVOD

Napisali smo že program (**5.8.2**), s katerim smo število 138 izpisali kot 831. Pri tem smo pazili, da smo program napisali tako, da smo, če smo želeli uporabiti drugo trimestno število, naredili popravek le na enem mestu (tam, kjer smo število 138 shranili v spremenljivko). Navkljub temu, pa če smo želeli delati z drugim številom, je bilo potrebno:

- popraviti program
- ponovno prevajanje programa
- izvedba programa

Uporabnik našega programa torej potrebuje izvorno kodo, kot tudi znanje popravljanja kode in prevajanja.

Če želimo, da uporabnik sam vnaša določene podatke v naš program, moramo spoznati način, kako lahko spremenljivki priredimo vrednost, ki jo uporabnik vnese s pomočjo tipkovnice. Najprej si bomo pogledali, kako bi prebrali in izpisali niz, ki ga vpiše uporabnik.

5.2 METODA ZA BRANJE PODATKOV

Za branje podatkov iz konzole v jeziku C# uporabimo metodo *ReadLine()*, ki pripada razredu *Console*. V C# podatek torej preberemo z **Console.ReadLine();**

Rezultat te metode je niz, ki vsebuje tisto zaporedje znakov, ki ga natipkamo. Shranili ga bomo v spremenljivko tipa *string*.

```
string vnos = Console.ReadLine();
```

Ko se izvaja ta stavek, se program "ustavi" in čaka, da nekaj natipkamo. Ko natipkamo določene znake in pritisnemo na tipko Enter, se natipkani znaki (brez Enter) shranijo v niz (v našem primeru v spremenljivko *vnos*).

Da vemo, da program čaka na nas, pred branjem z metodo *Write* (ali *WriteLine*) na zaslon izpišemo ustrezno opozorilo:

```
Console.Write("Vnesi ime avtorja: ");
 // preberemo stavek in ga shranimo v spremenljivko tipa string
string avtorIme = Console.ReadLine();
```

Branje podatkov iz konzole je možno le preko tipa niz (*string*). Vse podatke je potrebno kasneje pretvoriti iz niza v obliko, ki jo potrebujemo. Za pretvarjanje lahko porabimo metodo *Parse* ali pa metodo *Convert*.

Primer:

```
Console.Write("Vnesi še eno celo število: ");
int st = int.Parse(Console.ReadLine());
```

Lahko pa gremo tudi korak za korakom in podatek najprej preberemo v spremenljivko tipa *string*.

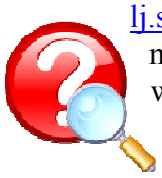
Primer:

```
Console.Write("Vnesi še eno celo število: ");
string branje = Console.ReadLine();
int st = int.Parse(branje);
```

Sedaj že vemo dovolj, da lahko program, s katerim smo trimestno število izpisali obrnjeno, spremenimo tako, da bomo število prebrali. Na ta način bo naš program uporaben za obračanje poljubnega trimestnega števila, saj se bo šele ob zagonu programa "odločilo", katero trimestno število obračamo.

```
static void Main(string[] args)
{
    Console.Write("Vnesi trimestno celo število: ");
    string beri = Console.ReadLine();
    int trimestnoStevilo = int.Parse(beri);

    int enice = trimestnoStevilo % 10;
    int dvomestnoStevilo = trimestnoStevilo / 10;
    int desetice = dvomestnoStevilo % 10;
    int stotice = dvomestnoStevilo / 10;

    Console.WriteLine("Število " + trimestnoStevilo);
    Console.WriteLine ("Obrnjeno število " + enice + desetice + stotice);
}
```

Vidimo, da smo dodali vrstici, kjer smo prebrali število kot niz. Sledi še popravek v vrstici kjer namesto števila 138 v spremenljivko *trimestnoStevilo* shranili število, ki ga dobimo s pretvorbo niza *beri* v tip *int*. Vse ostalo je ostalo nespremenjeno. To pove, da smo prvotni program res napisali tako, da je deloval za poljubno trimestno število.



Najprej malo obnovimo znanje. V ta namen rešite kviz, dostopen na naslovu http://up.fmf.uni-lj.si/branje-4558b1a4/index.html. Sedaj, ko smo osvežili teoretično znanje, pa napišimo še nekaj programov. Ustrezne najdete na primer na prej omenjenem wikiju, na naslovu http://penelope.fmf.uni-lj.si/diri0607/index.php/Kategorija:Branje.

6 POGOJNI STAVKI

6.1 UVOD

Problemov, ki se jih da rešiti s preprostimi programi, kjer se stavki izvajajo lepo po vrsti, eden za drugim, je razmeroma malo. Vsi stavki, ki smo jih spoznali do sedaj, so se izvajali zaporedoma. Najprej se je izvedel prvi stavek v metodi Main, nato naslednji in tako naprej. Pogosto pa se moramo odločiti med več možnostmi, oziroma izvršiti določene stavke samo, če je izpolnjen nek pogoj. To nam omogoča pogojni stavek, v katerem se na osnovi pravilnosti oz. nepravilnosti nekega pogoja program razveji na dve veji.
Za pogoje je značilna uporaba operatorjev za primerjavo ter logičnih operatorjev za povezovanje manjših pogojev v daljše logične izraze.

6.2 LOGIČNE VREDNOSTI IN LOGIČNE SPREMENLJIVKE

Logične spremenljivke so spremenljivke, ki lahko zavzamejo le eno izmed dveh logičnih vrednosti, **true** ali pa **false**. Označimo jih s tipom **bool**. V spremenljivki tipa **bool** torej lahko shranjujemo le vrednost pravilno **(true)** ali pa napačno **(false)**.

Primera:

```
bool trditev = true;
bool konec = false;
```

6.3 OSNOVNE LOGIČNE OPERACIJE

Logična operacija je operacija med logičnimi vrednostmi. Rezultat logične operacije je logična vrednost **true** (pravilno) ali **false** (napačno). Logične vrednosti lahko združujemo z logičnimi operatorji: **in**, **ali** ter **negacijo**. Naj bosta *a* in *b* logična izraza. Poglejmo si logične operacije med njima:

Tabela 3: Logični operatorji

Logični operator	Pomen	Logična operacija	Pomen	Opis
&&	Logični in (**and**).	a && b	logični in	Vrne *true*, kadar sta oba pogoja resnična (imata vrednost *true*).
||	Logični ali (**or**).	a || b	logični ali	Vrne *false*, kadar je vsaj en pogoj resničen.
!	Negacija (**not**).	!a	logična negacija	Vrne *false*, če je *a true* in *true*, če je a *false*.

Če je združenih več pogojev, lahko uporabimo oklepaje. Vrstni red operacij je takšen, kot to velja pri matematiki. Zato se && izvede pred ||. Kadar nismo prepričani, kakšna je prioriteta operatorjev, uporabimo oklepaje, da zagotovimo vrednotenje v želenem vrstnem redu.

Opozoriti velja, da se izrazi, v katerih uporabljamo operatorja && in ||, vrednotijo vedno od leve proti desni in sicer toliko časa, da je vrednost celega izraza določena.

6.4 PRIMERJALNE OPERACIJE

S primerjalnimi operatorji primerjamo dva izraza. Rezultat primerjalnih operatorjev je vedno tipa *bool*. Najpogosteje se z njimi srečujemo v zankah in pogojnih stavkih.

Naj bosta *a* in *b* izraza, katerih vrednosti sta števili. Poglejmo si primerjalne operacije med njima

Tabela 4: Primerjalne operacije

Primerjalna operacija	Pomen	Opis
a = = b	enako	Pogoj je izpolnjen (ima vrednost true), če sta vrednosti obeh izrazov enaki.
a != b	različno	Pogoj je izpolnjen, če sta vrednosti obeh izrazov različni.
a > b	večje	Pogoj je izpolnjen, če je vrednost levega izraza večja od vrednosti desnega izraza.
a >= b	večje ali enako	Pogoj je izpolnjen, če je vrednost levega izraza večja ali enaka od vrednosti desnega izraza.
a < b	manjše	Pogoj je izpolnjen, če je vrednost levega izraza manjša od vrednosti desnega izraza
a <= b	manjše ali enako	Pogoj je izpolnjen, če je vrednost levega izraza manjša ali enaka od vrednosti desnega izraza.

6.5 POGOJNI STAVEK IF

Pogojni stavek *if* uporabimo, kadar želimo izvesti določene stavke (oz. stavek) samo v primeru, ko je izpolnjen nek pogoj. Za besedo *if* v oklepajih zapišemo *pogoj*. To je poljuben logični izraz. Sledi mu blok stavkov (oz. stavek), ki jih želimo izvesti v primeru, če je pogoj resničen.

```
if (pogoj)
{
   stavek1;
   stavek2;
   ...
   stavekn;
}
```

Če je blok sestavljen samo iz enega stavka, lahko oklepaje izpustimo.

```
if (pogoj) stavek;
```

Kljub temu je bolje, da zaradi boljše preglednosti zavite oklepaje *{}* ohranimo.

6.6 ZGLEDI

6.6.1 Ali je prvo število manjše ali enako drugemu številu?

Napišimo preprost program, ki bo primerjal celi števili, ki sta shranjeni v dveh spremenljivkah. V primeru, da je prvo število manjše ali enako drugemu, bo izpisal "vrednost prvega števila" *je manjše ali enako* "od vrednosti drugega števila". Če je prva vrednost *5*, druga pa *7*, bo program izpisal *"5 je manjše ali enako 7."*.

```
1:      static void Main(string[] args)
2:      {
3:          Console.Write("Prvo število: ");
4:          String beri = Console.ReadLine();
5:          int prvoStevilo = int.Parse(beri);
6:          Console.Write("Drugo število:");
7:          int drugoStevilo = int.Parse(Console.ReadLine());
8:          if(prvoStevilo <= drugoStevilo)
9:            Console.WriteLine (prvoStevilo +" je manjše ali "+"enako "+drugoStevilo+".");
10:         if(prvoStevilo > drugoStevilo)
11:           Console.WriteLine (prvoStevilo + " je večje od " + drugoStevilo + ".");
12:     }
```

Opis programa.
V 3. in 4. vrstici prikažemo sporočilo, ki uporabniku pove, naj vnese prvo število. Ko uporabnik zapiše niz in pritisne tipko Enter, se zapisani niz prenese v niz *beri*. S klicem metode *int.Parse(beri)* niz *beri* pretvorimo v celoštevilčno vrednost, ki se nato priredi spremenljivki *prvoStevilo*. V 6. vrstici ponovno prikažemo sporočilo. Ko pri izvajanju 7. vrstice uporabnik zapiše niz in pritisne tipko *Enter*, vneseni niz pretvorimo v celoštevilčno vrednost, ki jo shranimo v spremenljivko *drugoStevilo*. V 8. vrstici preverimo pogoj in če je resničen, se izpiše ustrezen tekst (vrstica 9). Isto ponovimo v 10. vrstici. Preveri se pogoj in če je resničen, se izpiše ustrezen tekst (vrstica 11).
Ker je v našem primeru lahko resničen samo en pogoj, bi bilo bolj smiselno uporabiti pogojni stavek v obliki *if–else*. Tega si bomo ogledali v naslednjem razdelku.

6.6.2 Dvig denarja na bankomatu

Na transakcijskem računu imamo dovoljen limit 500 €. Ker nam je zmanjkalo denarja, želimo dvigniti določen znesek. Če ne presegamo limita, nam bankomat izplača izbrani znesek. Napišimo program, ki bo v primeru opravljene transakcije izpisal dvignjen znesek, stanje na računu in razpoložljivo stanje na računu. Če transakcija ni dovoljena, bo izpisal le stanje in razpoložljivo stanje na računu. Stanje na računu in znesek dviga preberemo.

```
1:      static void Main(string[] args)
```

```
2:      {
3:          const int LIMIT = -500;  //  dovoljen limit
4:          Console.Write("Stanje na računu: ");
5:          String beri = Console.ReadLine();
6:          int stanje = int.Parse(beri);
7:          Console.Write("Znesek dviga: ");
8:          beri =  Console.ReadLine();
9:          int dvig = int.Parse(beri);
10:         if(stanje - dvig >= LIMIT)
11:         {
12:              Console.WriteLine ("Dvig: " + dvig + " €");
13:             stanje = stanje - dvig;
14:         }
15:         int razpolozljivoStanje = stanje + Math.Abs(LIMIT);
16:         Console.WriteLine ("Stanje na računu: " + stanje + " €");
17:         Console.WriteLine ("Razpoložljivo stanje: " + razpolozljivoStanje + " €");
18:     }
```

Opis programa.

V vrstici 3 deklariramo konstanto in ji priredimo vrednost. Sledi vnos (branje) podatkov. Ko podatke preberemo in jih pretvorimo v cela števila, se preveri pogoj v vrstici 10. Če je pogoj resničen *(stanje - dvig >= LIMIT)*, se izvedeta stavka znotraj *if* (vrstici 12 in 13). V 15. vrstici sledi izračun razpoložljivega stanja. Izračuna se vrednost izraza, ki se priredi spremenljivki *razpolozljivoStanje*. Pomagali smo si z metodo *Abs()* iz razreda *Math*, ki vrne absolutno vrednost števila. V našem primeru je to *10000*. Na koncu še izpišemo stanje na računu ter razpoložljivo stanje.

6.7 IF − ELSE

Prejšnja oblika pogojnega stavka je poskrbela, da se je določen ukaz izvedel le, če je bil pogoj izpolnjen. Nato smo v vsakem primeru nadaljevali z istim ukazom. Včasih pa želimo, da se takrat, ko pogoj ni izpolnjen, zgodi nekaj drugega. Takrat uporabimo stavek *if* v kombinaciji z *else*.

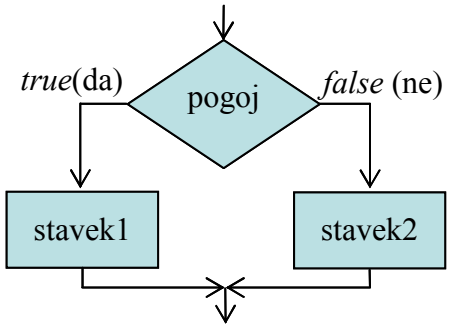
```
if (pogoj)
{
    stavekp1;
    stavekp2;
    ...
    stavekpn;
}
else
{
   stavekr1;
   stavekr2;
   ...
   stavekrk;
}
```

Če bo pogoj izpolnjen, se bodo izvedli *stavekp*$_1$ do *stavekp*$_n$ (oz. *stavek1* v spodnjem zgledu), sicer *stavekr1* do *stavekr* (oz. *stavek2* v spodnjem zgledu). Če sta bloka sestavljena samo iz enega stavka, lahko oklepaje izpustimo

```
if  (pogoj) stavek1;
else stavek2;
```

vendar jih zaradi boljše preglednosti praviloma vseeno zapišemo.

Shematično prikažemo stavek *if–else* takole:



Slika 15: Diagram poteka za stavek *if*

6.7.1 Praznovanje rojstnega dne

Janja ima rojstni dan 29. februarja. Če leto ni prestopno, ga praznuje 1. marca. Napišimo program, ki bo Janji pomagal ugotoviti, ali bo na določeno leto praznovala rojstni dan 29. februarja ali 1. marca. Leto je prestopno, kadar je deljivo s 4 in ne s 100 ali kadar je deljivo s 400.

```
static void Main(string[] args)
{
    Console.Write("Vpiši leto: ");
    string beri = Console.ReadLine();
    int leto = int.Parse(beri);

    if (((leto % 4 == 0) && (leto % 100 != 0))|| (leto % 400 == 0))
      Console.WriteLine ("Praznovanje bo 29. februarja.");
    else
      Console.WriteLine ("Praznovanje bo 1. marca.");
}
```

Prebrani niz smo pretvorili v celo število. Nato preverimo pogoj, ki je sestavljen iz dveh delov. Prvi del *((leto % 4 == 0) && (leto % 100 != 0))* je sestavljen iz dveh pod pogojev. Če sta pod pogoja resnična, se drugi del sploh ne preverja in izpiše se *"Praznovanje bo 29. februarja."*. V primeru, da prvi del ni resničen, se preveri drugi del. Če je resničen drugi del,

se prav tako izpiše *"Praznovanje bo 29. februarja."*. Če pa sta oba neresnična, se izvede *else* del. Izpiše se *"Praznovanje bo 1. marca."*.

6.8 GNEZDENI STAVEK IF

Gnezdeni stavek *if* je stavek *if* znotraj stavka *if*. Pri takih stavkih moramo biti pozorni na to, v katerem primeru se izvede veja *else* takega gnezdenega stavka. Pomagamo si seveda z oklepaji ***{ }***.
Pogosto imamo opraviti tudi s kombinacijo več pogojnih stavkov *if..else*. Znotraj pogojnega stavka imamo spet pogojni stavek. Kombinacijo običajno zapišemo v obliki

```
if (pogoj1)
{
    stavek1;
}
else if(pogoj2)
{
   stavek2;
}
…
else
{
  stavekn;
}
```

Če velja *pogoj1*, se izvede *stavek1*, sicer če velja *pogoj2*, se izvede *stavek2*, sicer …, sicer se izvede *stavekn*. V navedenem primeru se vedno izvede natanko en od stavkov *stavek1*, *stavek2*, ..., *stavekn*.

6.8.1 Telesna teža

Jelka je v lekarni dobila brošuro, v kateri je formula za izračun indeksa telesne mase. Indeks telesne mase je razmerje med telesno maso (v kg) in kvadratom višine (v m). S tem indeksom ugotovimo, če imamo čezmerno telesno maso. Napišimo program, ki bo prebral podatke o teži in višini in glede na izračunani indeks telesne mase (ITM) izpisal ustrezen tekst.
Če je ITM kot 18.5, naj program izpiše *"Premajhna telesna masa."*. V primeru, da je ITM med 18.5 in 24.9, naj se izpiše *"Normalna telesna masa."*. Če pa je ITM več kot 25, naj program izpiše *"Čezmerna telesna masa."*. Na koncu v oklepaju še izpišimo izračunani ITM.

```
1:      static void Main(string[] args)
2:      {
3:          double itm;
4:          Console.Write("Vnesi maso (v kg): ");
5:          string beri = Console.ReadLine();
6:          int masa = int.Parse(beri);
7:          Console.Write("Vnesi višino (v m):");
8:          beri = Console.ReadLine();
9:          double visina = double.Parse(beri);
10:         // izračunamo ITM
```

```
11:          itm = masa / visina / visina;
12:          if (itm < 18.5)
13:              Console.WriteLine ("Premajhna telesna teža." + " (itm = " + itm +")");
14:          else if (itm <= 24.9)   //  vemo da velja itm >= 18.5
15:              Console.WriteLine ("Normalna telesna teža." + " (itm = " + itm +")");
16:          else
17:              Console.WriteLine ("Čezmerna telesna teža." + " (itm = " + itm +")");
18:      }
```

V vrsticah 12 – 17 imamo primer gnezdenega stavka. Najprej se preveri prvi pogoj. Če je ITM manjši od *18.5* se izpiše *Premajhna telesna teža* in program se zaključi. Če to ni res (ITM je večji ali enak 18.5, preverimo drugi pogoj. Če je naš ITM manjši ali enak *24.9* (večji ali enak 18.5 je že!), se izpiše *Normalna telesna teža*, če pa ta pogoj ni izpolnjen, se izpiše *Cezmerna telesna teža*.

6.8.2 Dvomestno število

Prebrati moramo dvomestno število in izpisati njegove desetice in enice. Če vnešeno število ni dvomestno, naj se izpiše ustrezno obvestilo. Napišimo samo "glavni" del programa:

```
Console.Write("Vnesi število : ");
int stevilo = int.Parse (Console.ReadLine());
if (10 <= stevilo && stevilo <= 100)
{
    Console.WriteLine("desetice : " + stevilo / 10);
    Console.WriteLine("enice : " + stevilo % 10);
}
else  {
   Console.WriteLine("Število ni v dogovorjenih mejah!");
}
```

6.9 POVZETEK

Pogojni stavek srečamo v vseh programskih jezikih. Pojem pogojni govori o tem, da se bo nekaj zgodilo, če bo izpolnjen pogoj. S pomočjo pogojnega stavka v besedilo programa vstavimo stavek ali skupino stavkov, ki se bodo izvršili le ko bo pogoj izpolnjen. Pogoji so lahko sestavljeni iz več pogojev, ki jih med seboj povežemo z operatorjema && (logični **in**) in operatorjem || (logični **ali**), lahko pa tudi z negacijo **!** (**not**). Opozoriti velja, da se izrazi, v katerih uporabljamo operatorje **&&**, || in **!**, vrednotijo vedno od leve proti desni in sicer tako dolgo, da je vrednost celotnega izraza določena.

Pogojni stavek ima lahko tri oblike: prva je preprosta in vključuje samo en pogoj in en blok. Druga oblika ima vejo *else*, tretja pa omogoča nadaljnje vejitve *if* (gnezdeni stavek *if*). Kadar pa želimo program na enem mestu razvejiti na več vej, pa lahko (ni pa potrebno) uporabimo stavek *switch*. Več o tem stavku pa si lahko preberete npr. v gradivu Uranič S., Microsoft C#.NET. Dostopno je na naslovu *http://uranic.tsckr.si/C%23/C%23.pdf*, str. 36 − 39.

Za zaključek pa najprej rešite ustrezni kviz, ki je dosegljiv na naslovu http://up.fmf.uni-lj.si/pogojni_stavek-ef294d28/index.html. Oglejte si še animirani prikaz sestavljanje programa, ki poišče minimum treh števil. Animacija je dostopna na naslovu http://up.fmf.uni-lj.si/csmintrehstevil-968c715b/csmintrehstevil.htm. Nato pa napišite nekaj programov, kjer bo problem zahteval vejitev. Ustrezne probleme najdete na primer na http://penelope.fmf.uni-lj.si/diri0607/index.php/Kategorija:Pogojni_stavki, kjer med več kot 140 nalogami ne bo težko najti kakšne vam primerne.

7 ZANKE

7.1 UVOD

Denimo, da bi želeli izpisati vsa naravna števila med 1 in 20. Z znanjem, ki ga imamo, to ni nobena težava. Hitro je tu program:

```
static void Main(string[] args)
{
      Console.WriteLine("1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20");
}
```

Kaj pa, če bi želeli izpisati števila med 1 in 1000?

Janezek je bil v šoli poreden. Zelo poreden. In zato mu je učiteljica naročila, naj sestavi program, ki 5x izpiše "V šoli se moram lepo obnašati!" Janezek hitro vzame "okostje" programa in ga dopolni s

```
Console.WriteLine("V šoli se moram lepo obnašati!");
Console.WriteLine("V šoli se moram lepo obnašati!");
Console.WriteLine("V šoli se moram lepo obnašati!");
Console.WriteLine("V šoli se moram lepo obnašati!");
Console.WriteLine("V šoli se moram lepo obnašati!");
```

Ker pa se Janezek še vedno ne obnaša lepo, sledi nova kazen. Ker učiteljica ve, da je pomočnik ravnatelja Robert, ki na šoli med drugim opravlja tudi nalogo vzdrževalca programske opreme, uspel "sesuti" operacijski sistem tako, da se enostavno ne da več kopirati besedila, Janezku pa ni nič bolj zoprnega kot veliko tipkanja, je kazen vzgojna. Sedaj mora omenjeni stavek izpisati 100x.
Janezek je sprva obupan. Ampak bistra glavica, v Google natipka "C# ponavljanje istih ukazov" in kaj hitro se mu razvedri čelo …

Izpiši 100x isti stavek, seštej 10 števil, izpiši 20 zvezdic, nariši n krogov, seštej dosežene točke vsakega dijaka, izračunaj plačo zaposlenih, …

Kaj je skupno vsem tem problemom? Gre za ponavljanje. Izvajamo isti postopek, le da uporabljamo spremenjene podatke. Ponavljanju stavkov v programiranju pogosto rečemo **zanke**.

Zanke v programiranju uporabljamo takrat, kadar želimo enega ali več stavkov ponoviti večkrat zaporedoma. V C# poznamo naslednje zanke: *while*, *for*, do *while* in *foreach*. V tem poglavju si bomo ogledali zanki *while* in *for*. Zanki do *while* in *foreach* uporabljamo redkeje, saj za programiranje povsem zadoščala že zanka *while* (ali zanka *for*). Vsako zanko lahko izvedemo z zanko *while*. Kljub temu bomo spoznali še zanko *for*, saj jo srečamo zelo pogosto. Zanko *while* uporabljamo predvsem takrat, kadar število ponavljanj zanke ni vnaprej znano. Če pa je število prehodov zanke odvisno od nekega števca, običajno uporabimo zanko *for*.

7.2 ZANKA WHILE

Zanka *while* je zanka, ki jo verjetno uporabljamo najbolj pogosto. Uporabimo jo, kadar želimo, da se izvajanje določenih stavkov (oz. stavek) ponavlja, dokler je določen pogoj izpolnjen. Predvsem pa jo uporabljamo takrat, kadar število ponavljanj zanke ni vnaprej znano. Če pa je število prehodov zanke odvisno od nekega števca, običajno uporabimo zanko **for**.

Najpomembnejše značilnosti zanke *while* so

- pogoj preverimo na začetku zanke,
- zanka se izvaja, dokler je pogoj izpolnjen,
- če pogoj ni izpolnjen že na začetku, se zanka ne izvede niti enkrat,
- pogoj, ki ga preverjamo, je lahko sestavljen.

Struktura zanke *while*:

```
while (pogoj)
{
    stavek1;
    stavek2;
    ...
    stavekn;
}
```

Tako kot pri pogojnem stavku za besedo *if*, tukaj za besedo *while* v oklepajih zapišemo pogoj. Sledi mu blok stavkov (oz. stavek), ki jih želimo izvajati večkrat – toliko časa, kolikor časa je pogoj izpolnjen. Tem stavkom rečemo tudi telo zanke.

Zanka *while* se izvaja, dokler je pogoj resničen (ima vrednost *true*). Njegovo resničnost se preveri pred vsakim izvajanjem telesa zanke. To se zgodi tudi takoj na začetku, zato ni nujno, da se stavki (oz. stavek), ki mu sledijo, sploh izvedejo. Vmes med izvajanjem bloka stavkov se pogoj ne preverja, ampak kot smo dejali, le pred vsako ponovitvijo.

Če je blok sestavljen samo iz enega stavka, lahko oklepaje izpustimo.

```
while (pogoj) stavek;
```

Običajno pa, tako kot pri pogojnem stavku, tudi tu zaradi boljše preglednosti oklepaje ohranimo.

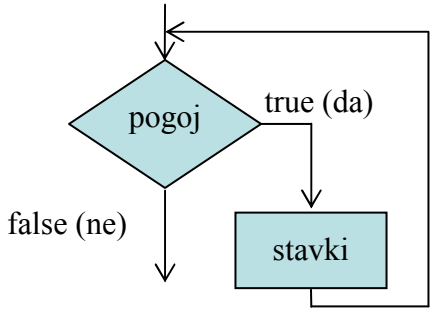
Na dolgo bi delovanje *while* zanke opisali takole:

- najprej se preveri pogoj,
- če ima pogoj vrednost *true* (je resničen), se izvedejo stavki telesa zanke,
- nato se ponovno preveri pogoj,
- če je še vedno izpolnjen, se znova izvedejo stavki telesa zanke,
- ponovno se preveri pogoj,
- ...

- če pogoj ob preverjanju nima več vrednosti *true* (ni več resničen), se konča izvajanje zanke *while*.

Ponavljaj:

če je **pogoj true,**
izvedi **stavek**,
sicer prekini izvajanje,



Slika 16: Diagram poteka zanke while

Na kratko pa zanko *while* lahko opišemo z: *Dokler je logični pogoj izpolnjen, izvajaj stavke v zavitih oklepajih.*

Poglejmo sedaj, kako bi rešili Janezkov problem.
kaj ponavljamo:
izpis stavka
pogoj:
dokler ne izpišemo 100 stavkov
torej moramo šteti izpisane stavke
(izpisanihStavkov < 100)
pred začetkom ponavljanja izpisanih stavkov še ni
izpisanihStavkov = 0;
v zanki
izpišemo stavek
števec izpisanih stavkov povečamo za 1
izpisanihStavkov = izpisanihStavkov + 1;

In ustrezni del programa

```
int izpisanihStavkov = 0;
while (izpisanihStavkov < 100)
{
   Console.WriteLine("V šoli se moram lepo obnašati!");
    //  nov izpisani stavek
   izpisanihStavkov = izpisanihStavkov + 1;
}
```

7.3 ZGLEDI

7.3.1 Izpis števil

Izpišimo števila od *1* do *10*. Vsako naj bo v svoji vrstici.

```
1:  static void Main(string[] args)
2:  {
3:      int stevilo = 1;
4:      while(stevilo <= 10)
5:      {
6:          Console.WriteLine(stevilo);
7:          stevilo = stevilo + 1;
8:      }
9:      Console.WriteLine("Konec programa");
10: }
```

Na začetku ob deklaraciji spremenljivki *stevilo* priredimo začetno vrednost *1* (vrstica 3). Nato se preveri pogoj (vrstica 4). Ker je vrednost v spremenljivki manjša ali enaka od *10* (*1 <= 10*), se izpiše vrednost spremenljivke *stevilo* (vrstica 6), torej *1*. Nato se izračuna desni del, ki se priredi spremenljivki na levi strani (*stevilo*). Vrednost v spremenljivki je sedaj *2*. Nato se zopet preveri pogoj v vrstici 5 in ker je resničen (*2 <= 10*), se izvede blok zanke. Izpiše se trenutna vrednost v spremenljivki *stevilo*, tokrat *2*. Povečamo vrednost spremenljivke *stevilo* na 3. Ponovno preverimo pogoj v vrstici 5. Ker je izpolnjen, ...
Zanka se torej ponavlja, dokler je vrednost v spremenljivki *stevilo* manjša ali enaka *10*. Ko je vrednost v spremenljivki enaka *10*, pogoj še vedno velja. Zato se izvede ukaz v vrstici 6 in na zaslon se izpiše *10*. Nato se vrednost spremenljivke *stevilo* poveča za *1*. Sedaj je vrednost v spremenljivki *stevilo* enaka *11*. Zopet se preveri pogoj v peti vrstici in ker ni več resničen (*11 <= 10*), se zanka zaključi. Program se nadaljuje v vrstici 9. Izpiše se besedilo *Konec programa.*

Stavek v sedmi vrstici *stevilo = stevilo + 1;* lahko krajše zapišemo kot *stevilo++;*.

Program bi lahko napisali tudi malce drugače

```
static void Main(string[] args)
{
  int stevilo = 0;
  while(stevilo < 10)
  {
      stevilo = stevilo + 1;
      Console.WriteLine (stevilo);
  }
  Console.WriteLine ("Konec programa.");
}
```

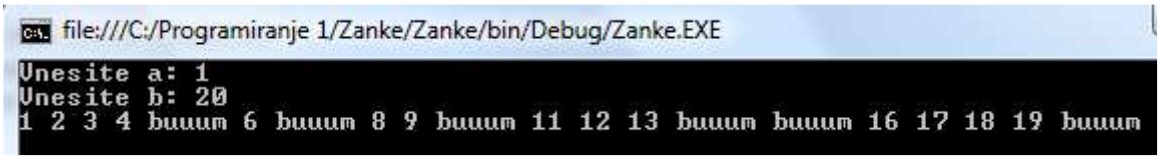
Če program izvedemo, vidimo, da nam da enak izpis kot prejšnji program. Spremembo smo naredili v vrednostih, ki jih zavzame števec. Pri prvi obliki programa je spremenljivka *stevilo* imela zaporedoma vrednosti 1, 2, 3, ..., 9, 10 in 11, pri drugi pa 0, 1, 2, 3, ..., 9 in 10.

7.3.2 Buuum

"Buuum" je igra z naslednjim pravilom: Sodelujoči zaporedoma govorijo števila, vendar morajo namesto večkratnikov števila pet in večkratnikov števila sedem reči *buuum*. Sestavimo program, ki bo po tem pravilu izpisal cela števila od a do b.

Pomagali si bomo s pogojnim stavkom, ki preveri, ali je število večkratnik števila 5 ali števila 7. Ko bo pogoj pogojnega stavka resničen, bomo izpisali besedo *buuum*.

```
static void Main(string[] args)
{
     // Preberemo interval
    Console.Write("Vnesite a: ");
    int a = int.Parse(Console.ReadLine());
    Console.Write("Vnesite b: ");
    int b = int.Parse(Console.ReadLine());
     // Drugi vnos je manjši od prvega
    if (b < a)
    {
        int t = a;
        a = b;
        b = t;
     }
     // Izpis besede buuum ali števil
    while(a <= b)
    {
            if((a % 5 == 0) || (a % 7 == 0))
                Console.Write("buuum");
            else
                Console.Write(a);
            Console.Write(" ");
       a = a + 1;
    }
}
```



Slika 17: Rezultat igre Buuum

Razlaga. Z branjem napolnimo spremenljivki *a* in *b*. Nato s pomočjo pogojnega stavka preverimo, če je vrednost spremenljivke b manjše od vrednosti spremenljivke *a*. Če je pogoj resničen, zamenjamo vrednost spremenljivke *a* z vrednostjo spremenljivke *b* in vrednost spremenljivke *b* z vrednostjo spremenljivke *a*. Pri zamenjavi vrednosti si pomagamo s pomožno spremenljivko *t*.

S pomočjo zanke *while* nato izpisujemo števila od *a* do *b*. Če je število večkratnik števila 5 ali števila 7, izpišemo besedo *buuum*, sicer izpišemo preverjeno število. Na koncu vsakega izpisa izpišemo presledek in se pomaknemo na naslednje število.

7.4 NESKONČNA ZANKA

Pri zanki *while* moramo paziti, da se zanka konča. V primeru nepravilne uporabe lahko zanka teče v neskončnost (se zacikla). Pravilno zapisana zanka bo torej taka, kjer pogoj nekoč le dobi vrednost *false*.

```
static void Main(string[] args)
{
    int stevilo = 1;
    while(stevilo > 0)
    {
        Console.WriteLine (stevilo);
        stevilo++;
    }
}
```

Program izpisuje števila v neskončnost in se sploh ne ustavi. Pogoj je vedno resničen, saj je vrednost spremenljivke *stevilo* vedno večja od *0*. No, če bi bili potrpežljivi, bi videli, da se bo program vseeno ustavil. Namreč ko bi prišlo do prekoračitve obsega celih števil, bi v spremenljivki *stevilo* dobili negativno število, ki bi ustavila zanko.

No, pri naslednjem zgledu pa se zanka zagotovo nikoli ne ustavi.

```
static void Main(string[] args)
{
    int stevilo = 1;
    while(stevilo < 10)
      Console.WriteLine(stevilo);
      stevilo++;
}
```

Tudi v tem primeru je pogoj vedno resničen, saj je vrednost spremenljivke *stevilo* vedno *1* in je vedno večja od *0*. Ker ni zavitih oklepajev, spada v *while* zanko samo stavek *Console.WriteLine(stevilo);* in ker je pogoj vedno resničen, se vrstica *stevilo++;* sploh ne izvede. Vrednost spremenljivke *stevilo* se torej nikoli ne spremeni in je pogoj vedno izpolnjen, iz česar sledi, da se program zacikla.

Povzetek:

Kako torej sestavljamo programe, ki potrebujejo zanke? Dobro premislimo o treh stvareh:

- Kaj naj se zgodi v tekoči ponovitvi zanke
  - Pišemo i-ti krog
  - Pregledujemo tekočo vrstico
- Kaj naj se zgodi na začetku (pred vstopom v zanko)
  - Posebni pogoji

    - Nastavitev števcev
    - Vrednost kontrolne spremenljivke mora biti taka, da se zanka sploh začne
- Dogajanje na koncu
    - Ali je potrebno z zadnjim elementom kaj posebnega narediti
    - Smo števec po "nepotrebnem" preveč povečali

7.5 POGOSTE NAPAKE

Premislimo, kaj naredijo naslednji delčki programa. Njihov učinek bomo le na kratko opisali, bralec pa naj dobro premisli, zakaj in kako!

```
int i = 100;
string odgovor = "";
while (i > 100)
{
    odgovor = odgovor + "riba raca rak ";
    i = i + 1;
}
Console.WriteLine(odgovor);
```

Ta del programa izpiše le prazen niz (torej dejansko ne izpiše nič, le v novo vrstico skoči).

```
string odgovor = "";
while (true)
{
   odgovor = odgovor + "plug pod klopjo, ";
   Console.WriteLine(odgovor);
}
Console.WriteLine("Končali smo.");
```

Te vrstice povzročijo neskončno zanko v kateri se izpisuje

plug pod klopjo,
plug pod klopjo, plug pod klopjo,
plug pod klopjo, plug pod klopjo, plug pod klopjo,
plug pod klopjo, plug pod klopjo, plug pod klopjo, plug pod klopjo,
...

```
string odgovor = "";
i = 1;
while (i <= 10)
   odgovor = odgovor + "plug pod klopjo, ";
   Console.WriteLine(odgovor);
   i = i + 1;
Console.WriteLine("Končali smo.");
```

Tudi tu gre za neskončno zanko. A ta ne izpiše nič, saj do *Console.WriteLine(odgovor);* sploh ne pride.

```
string odgovor = "";
int i = 1;
while (i <= 10) ;
{
    odgovor = odgovor + "plug pod klopjo, ";
    Console.WriteLine(odgovor);
    i = i + 1;
}
Console.WriteLine("Končali smo.");
```

Verjeli ali ne, tudi tu gre za neskončno zanko.

Najbolj pogoste napake pri *while* zanki so torej:

- Napačen pogoj: zanka se nikoli ne konča, ali pa se izvede nikoli
- Pozabljeni { }: pravilo je enako kot pri *if* stavku (zamikanje ne pomaga, prevajalniku je vseeno)
- Napačno postavljen znak podpičje takoj za pogojem
- Zanka ima v telesu "prazen" stavek

7.6 ZGLEDI

7.6.1 Vsota členov zaporedja

Izračunaj vsoto *N* členov zaporedja, če poznaš splošni člen *a[n]=(n+1)*(n-1); n = 1,2,3,...N*. Členi zaporedja so potemtakem: 0, 3, 8, 15, 24, 35 ... . Izpis naj vsebuje tudi člene zaporedja, ter vsoto prvih *N* členov.

Zapišimo le "zanimivi del" programa.

```
int n = 1;
int vsota = 0;
Console.WriteLine("Zaporedje ima splošni člen a[n]=(n+1)*(n-1); \n");
Console.Write("Vpiši število členov zaporedja: ");
int steviloClenov = int.Parse(Console.ReadLine());
while (n <= steviloClenov)  //  nismo še pregledali dovolj členov
{
    int clen;
    clen = (n + 1) * (n - 1);  //  izračun n-tega člena
    Console.WriteLine(n +". člen zaporedja je:" + clen);  // izpis člena
    vsota = vsota + clen;  // vsoto povečamo za n-ti člen
    n = n + 1;   // krajši zapis tega stavka je:    n++;
}
Console.WriteLine("\nVsota prvih "+ steviloClenov +  " clenov tega zaporedja je "+ vsota);
```

7.6.2 Izlušči sode števke

Napiši program, ki iz prebranega pozitivnega celega števila naredi novo število, v katerem so le sode števke danega števila. Iz 122436 torej naredimo število 2246. Če so vse števke danega števila lihe, je novo število enako 0.

Ideja programa je, da iz števila "luščimo" zadnje števke. Če so sode (ostanek pri deljenju z 2 je 0), jih dodamo na začetek novega števila. Da jih bomo lahko dodali na začetek, jih bomo morali pomnožiti z ustrezno potenco števila 10. Poglejmo, kako se bodo spreminjali glavni "igralci", če je vnešeno število 14556. Zapisali bomo stanje na začetku vsake ponovitve zanke (torej tik preden se preveri pogoj).

Tabela 5: "luščimo" zadnje števke

stevilo	novo število	cifra	faktor
14556	0	-	1
1455	6	6	10
145	6	5	10
14	6	5	10
1	46	4	100
0	46	1	100

Tudi tokrat bomo zapisali le "zanimivi" del.

```
1:      int novoStevilo = 0, cifra, faktor = 1;
2:      Console.Write("Vnesi poljubno pozitivno celo število: ");
3:      int stevilo = int.Parse(Console.ReadLine());
4:      while (stevilo > 0){
5:          cifra = stevilo % 10;
6:          if (cifra % 2 == 0)
7:          {
8:              novoStevilo = novoStevilo + cifra * faktor;
9:              faktor = faktor * 10;
10:         }
11:         stevilo = stevilo / 10;
12:     }
13:     Console.WriteLine("\nNovo število je " + novoStevilo);
```

Opis:
V vrstici 1 smo sočasno deklarirali tri spremenljivke: *novoStevilo*, *cifra* in *faktor*. Prvi in zadnji smo priredili tudi vrednost. Pogosto napačno mislimo, da smo v primeru kot je ta, tudi spremenljivki *cifra* priredili vrednost 1. Vendar prirejanje v takem primeru vedno velja le za zadnjo navedeno spremenljivko. Zato je morda bolj smiselno, da bi ta del zapisali kar kot

```
int novoStevilo = 0;
int cifra;
int faktor = 1;
```

in se s tem izognili morebitnim nesporazumom.
Razložimo še vrstico 3. V tej vrstici opravimo cel kup dela. Deklariramo spremenljivko *stevilo* (tipa *int*) in ji priredimo vrednost. To dobimo tako, da z metodo *Parse* v število pretvorimo niz, ki ga vnesemo preko tipkovnice (in ga posreduje metoda *ReadLine*). Ta vrstica se torej izvede takole:

- Najprej se pripravi prostor za spremnljivko stevilo in sicer tako, da se vanjo lahko shrani celo število (nekaj tipa *int*).
- Računalnik čaka, da vnesemo določeno zaporedje znakov.
- Ko pritisnemo na tipko *Enter*, se vnešeni niz posreduje metodi *Parse*.
- Ta metoda pretvori niz v celo število in ga vrne kot rezultat.
- Dobljeno število se shrani v spremenljivko stevilo.

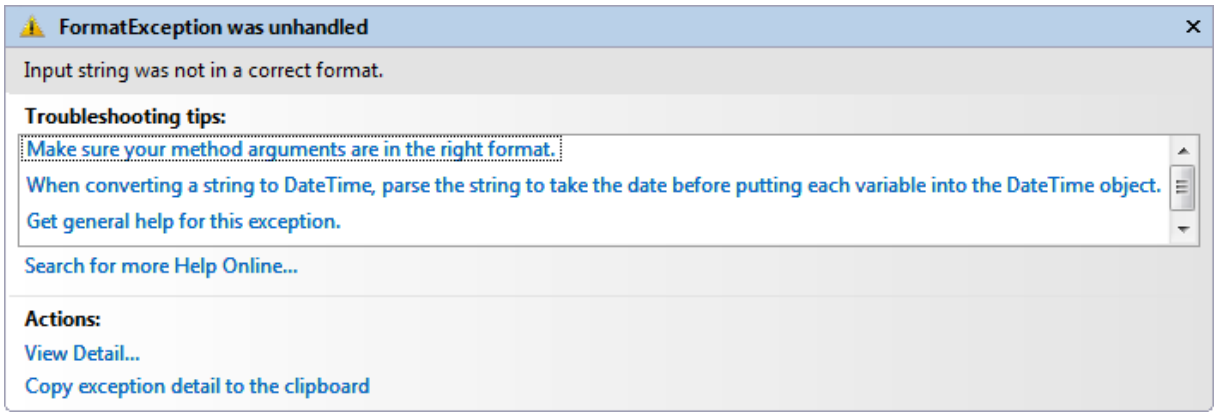
Glavno delo opravimo v zanki *while* (4 – 12). Zanko izvajamo toliko časa, dokler nam v spremenljivki stevilo "ne zmanjka" znakov. Ta spremenljivka bo za vnešeno število 122436 to zaporedoma 122436, 12243, 1224, 122, 12, 1 in na koncu 0. Takrat se bo izvajanje zanke končalo. Če pa bi bilo število enako 65, bi zanko izvedli le 2x, ko bi število postalo najprej 6 in nato 0. V vrstici 5 izluščimo tekočo števko. Če je soda (pogoj v vrstici 6), jo pomnožimo z ustrezno potenco števila 10. Če jo prištejemo obstoječemu novemu številu, smo jo s tem dodali ravno z leve. Sedaj še faktor pomnožimo z 10, da bomo naslednjo sodo števko spet dodali levo. V vrstici 11 pa "skrajšamo" število, torej odrežemo to števko, ki smo jo ravno obdelali.

Oglejmo si še eno obliko tega programa. Tu se bomo izognili potencam števila 10. Da pa bomo števke lahko "lepili" z leve, bomo uporabili kar nize. Novo število bomo torej sestavljali kot niz in ga na koncu z metodo *Parse* pretvorili v število.

```
1:      int novoStevilo, cifra;
2:      string noSt = "";
3:      Console.Write("Vnesi poljubno pozitivno celo število: ");
4:      int stevilo = int.Parse(Console.ReadLine());
5:      while (stevilo > 0){
6:          cifra = stevilo % 10;
7:          if (cifra % 2 == 0)
8:          {
9:              noSt = cifra + noSt;  //  prilepimo z leve
10:         }
11:         stevilo = stevilo / 10;
12:     }
13:     novoStevilo = int.Parse(noSt);
13:     Console.WriteLine("\nNovo število je " + novoStevilo);
```

Pozorni bralec se bo verjetno vprašal, kaj se bo zgodilo, če bo prebrano število npr. 1335, torej tako, samo z lihimi števkami. V prvi različici ni dileme, novo število bo (kot zahteva

naloga) enako 0. Kaj pa pri drugi obliki? Vprašanje je torej, kaj stori metoda *Parse*, če ji "podtaknemo" prazen niz. Žal ji to ni všeč in se pritoži z



Slika 18: Napaka v programu

Kako se rešiti iz tega položaja, presega naše dosedanje znanje. Namreč poznati bi morali način, kako primerjati nize. To pa si bomo ogledali šele v naslednjem razdelku.

7.6.3 Povprečje ocen

Denimo da imamo *n* predmetov. Iz ocen predmetov izračunajmo povprečje ocen, najmanjšo in največjo oceno. Ocene so cela števila med 1 in 10. Pomagali si bomo z metodama *Min()* in *Max()* iz razreda *Math*. Prva vrne manjše, druga pa večje število izmed dveh.

```
static void Main(string[] args)
{

    int vsota = 0;  //  seštevek ocen
    int stevec = 1;  //  števec predmetov
    int najnizja = 11;  //  najnižja ocena
    int najvisja = 0;  //  najvišja ocena
    Console.WriteLine("Število predmetov:" );
    string beri = Console.ReadLine();
    int steviloPredmetov = int.Parse(beri);

     //  seštevamo vnesene predmete ter iščemo najnižjo in najvišjo oceno
    while(stevec <= steviloPredmetov)
    {
        Console.WriteLine("Vnesi " + stevec + ". oceno:");
        beri = Console.ReadLine();
        int ocena = int.Parse(beri);
        vsota = vsota + ocena;
        najnizja = Math.Min(najnizja, ocena);
        najvisja = Math.Max(najvisja, ocena);
        stevec++;
    }
```

```
    //  povprečje ocen
    double povprecje = (double)vsota / steviloPredmetov;

    Console.WriteLine ("Tvoje povprečje je " + povprecje + ".");
    Console.WriteLine ("Zaokroženo povprečje je " + (int)(povprecje + 0.5) + ".");
    Console.WriteLine ("Najnižja ocena je " + najnizja);
    Console.WriteLine ("Najvišja ocena je " + najvisja);
}
```

Opis programa.

Najprej deklariramo ustrezne spremenljivke in jim priredimo začetne vrednosti. Preko tipkovnice smo prebrali število predmetov in jih shranili v spremenljivko *steviloPredmetov*. S pomočjo zanke *while* beremo ocene predmetov. Če je novo število (nova ocena) večje od dosedaj največjega, je med vsemi do sedanjimi števili največje prav to *(Math.Max(najvisja, ocena))*, če pa je manjše ali enako, pa doslej največje število ostane enako. Podobno velja tudi za najmanjše število.
Na vsakem koraku seštevamo ocene. Njihovo vsoto bomo potrebovali za izračun povprečja. Na koncu še izpišemo želene vrednosti. Pri računanju povprečja ne pozabimo, da imamo ves čas opraviti s celimi števili, zato moramo pred deljenjem opraviti ustrezno pretvorbo izraza.

7.7 STAVKA BREAK IN CONTINUE

Stavek **break** povzroči izstop iz (najbolj notranje) zanke tipa *for*, *while* ali *do while*. Stavek *continue* pa ima nasprotno vlogo. Pri zanki *while* skoči na pogoj zanke, ter sproži ponovno preverjanje pogoja. Če je ta še izpolnjen, se izvajanje zanke nadaljuje, sicer pa se zanka zaključi.

```
double stevilo;
while (true)
{
    Console.Write("Vnesi poljubno število :");
    stevilo = double.Parse(Console.ReadLine());
    if (stevilo == 0.0)
        continue;  // nazaj na začetek zanke
    Console.WriteLine("  Obratna vrednost števila "+ stevilo +" je "   + 1 / stevilo);
    break;  // izstop iz zanke
}
```

7.7.1 Trgovec

Z novim letom se je povečala stopnja davka na dodano vrednost (DDV) z 20% na 22%. Podjetje Trgovec d.o.o. mora v kratkem času spremeniti cene izdelkov tako, da se cena brez davka ne spremeni. To bi bilo sicer nepotrebno, a kaj, ko imajo v svojih podatkih le končne cene (torej cene z že upoštevanim davkom). Zato napišimo program, s katerim jim bomo pomagali. V program preko tipkovnice vnašamo cene, program pa izpisuje nove vrednosti. To počnemo, dokler ne vnesemo 0. Če je vnesena cena negativna, naj program izpiše "*Cena izdelka je narobe vnesena. Vnesi znova.*". Novo ceno zaokrožimo na dve mesti natančno.

Pomagali si bomo s stavkom *break*, ki poskrbi, da se zanka predčasno zaključi. Ko se namreč izvede stavek *break*, zapustimo stavek (zanko), v katerem se nahajamo.
Uporabili bomo neskončno zanko, v kateri je pogoj vedno *true* (resničen). Znotraj zanke bomo spraševali po ceni toliko časa, dokler ne vnesemo števila 0. Takrat bomo zanko prekinili s stavkom *break*.

```
static void Main(string[] args)
{
    // Deklaracija spremenljivk
   const double ddvPrvi = 1.20;  // DDV 20%
   const double ddvDrugi = 1.22;  // DDV 22%
   double cena;

    // Preračunavanje cene iz 20% DDV v 22% DDV
   while (true)
   {
      Console.Write("Vnesi ceno izdelka z 20% DDV: ");
      cena = double.Parse(Console.ReadLine());

      if (cena == 0)
      { // Konec vnosov
         break;
      }
      if (cena < 0)
      {
        Console.WriteLine("Cena izdelka je vnesena narobe. Vnesi znova.\n");
      }
      else
      {
        cena = ddvDrugi * cena / ddvPrvi;  // Nova cena
        cena = Math.Round(cena * 100) / 100.0;
        Console.WriteLine("Cena izdelka z 22% DDV je " + cena + ".\n");
      }
   }
}
```

Razlaga. Najprej deklariramo konstantni spremenljivki *ddvPrvi* in *ddvDrugi* in jima priredimo začetni vrednosti. V spremenljivki *ddvPrvi* hranimo 20% DDV, medtem, ko v spremenljivki *ddvDrugi* hranimo 22% DDV. Zatem najavimo spremenljivko *cena*. To spremenljivko potrebujemo za ceno posameznega izdelka. Nato vstopimo v zanko, katere pogoj je vedno *true*. Torej se načeloma zanka izvaja neskončno dolgo, saj bo pogoj ves čas izpolnjen. Znotraj zanke sprašujemo po ceni izdelka. Če je vnesena cena enaka 0, prekinemo izvajanje zanke *while*. Če je vnesena cena negativna, izpišemo opozorilno besedilo *Cena izdelka je vnesena narobe. Vnesi znova.*. Če pa je vnesena cena izdelka pozitivna, izračunamo novo ceno. Zaokrožimo jo na dve decimalni mesti ter jo izpišemo. Po izpisu opozorila ali nove cene se vrnemo na začetek zanke *while* in ponovimo postopek. Postopek ponavljamo

tako dolgo, dokler ni vnesena cena enaka 0. Takrat se izvede stavek *break* in izvajanje zanke se konča.

7.7.2 Števila deljiva s tri

Izpišimo števila od 1 do 30, ki so deljiva s številom tri. Števila naj bodo izpisana v isti vrstici in ločena s presledkom.

Pomagali si bomo z ukazom *continue*, ki poskrbi, da se posamezna ponovitev zanke predčasno prekine in se zanka nadaljuje z naslednjo ponovitvijo (če je pogoj za ponovitev zanke še izpolnjen).
Uporabili bomo pogojni stavek, s katerim bomo preverjali, če število ni deljivo s 3. Če bo pogoj resničen, bomo s pomočjo stavka *continue* takoj nadaljevali z naslednjo ponovitvijo stavkov telesa zanke.

```
static void Main(string[] args)
{
    // Deklaracija spremenljivke
    int stevilo = 0;
    const int zgMeja = 30;  // Zgornja meja števil

    // Izpis števil deljivih s tri
    while (stevilo <= zgMeja)
    {
       stevilo++;
       if (stevilo % 3 != 0) continue;
       Console.Write(stevilo + " ");
        // Izpis le, če je število deljivo s 3
    }
    // Premik v novo vrstico
    Console.WriteLine();
}
```

Razlaga.
Spremenljivko *stevilo* nastavimo na vrednost 0 in konstantno spremenljivko *zgMeja* na vrednost 30. Potem povečamo vrednost spremenljivke *stevilo* za 1. Če je vrednost spremenljivke deljivo s številom 3, se izpiše vrednost spremenljivke in presledek. Če vrednost spremenljivke ni deljivo s 3, pa se samo poveča vrednost spremenljivke *stevilo* za 1. V tem primeru s stavkom *continue* skočimo na začetek zanke.

7.8 ZANKA FOR

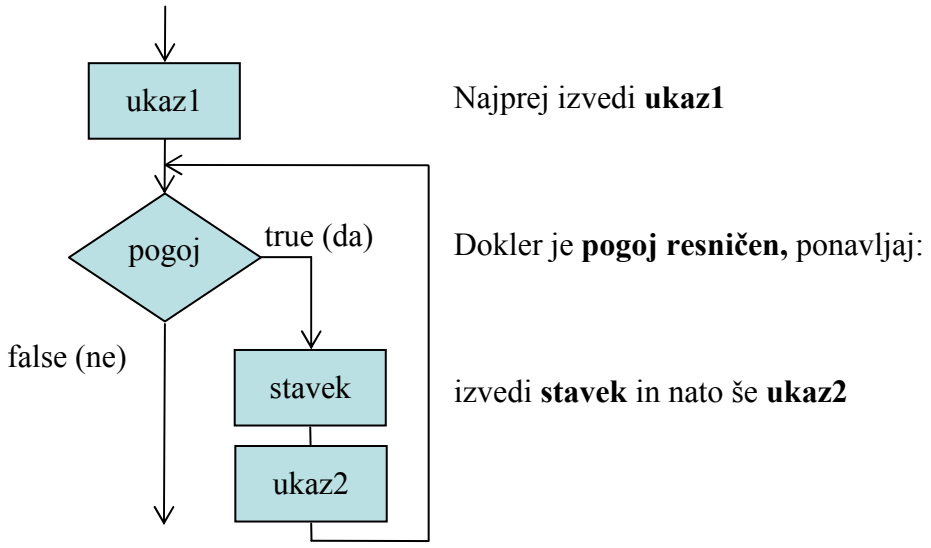
Zanka *for* je verjetno najpogosteje uporabljena zanka. Je zelo podobna zanki *while*. Oglejte si najprej animacijo, dostopno na naslovu http://up.fmf.uni-lj.si/cszankafor-6381e1fc/for_osnova.htm

Kot vidimo, za besedo *for* v oklepajih zapišemo tri dele: *ukaz1*, *pogoj* in *ukaz2,* ločene z podpičjem. Sledi mu blok stavkov (oz. stavek), ki jih želimo izvajati

večkrat – toliko časa, dokler je pogoj izpolnjen.
Preden se zanka sploh začne izvajati, se izvede *ukaz1*. Nato se preveri pogoj. Če je ta izpolnjen, se izvedejo stavki (oz. stavek), nato se izvede še *ukaz2*. Ponovno se preveri pogoj. Če je še vedno izpolnjen, se spet izvedejo stavki v telesu zanke in za njimi še *ukaz2*. Če pogoj ni izpolnjen, se zanka konča.

```
for(predZanko/ukaz1/; pogoj; poKoraku/ukaz2/)
{
    stavek1;
    stavek2;
    …
    stavekn;
}
```

Oblika zanke pomeni: "Izvedi stavek *predZanko*. Nato preveri pogoj. Če je izpolnjen, izvedi stavke stavek1, stavek2 … stavekn in nato še stavek *poKoraku*. Ponovno preveri pogoj. Če je še vedno izpolnjen, ponovno izvedi stavke stavek1, stavek2 … stavekn in še stavek *poKoraku*. Ponovno preveri pogoj. Ko pogoj ni izpolnjen, se zanka konča."



Slika 19: Diagram poteka za *for* zanko

Čeprav je v zanki *for ukaz2* napisan pred *stavek*, se najprej izvedejo *stavek* in šele nato *ukaz2*.

Katero zanko bomo uporabili, je odvisno od naše odločitve. Zanko *for* običajno uporabljamo, ko moramo šteti ponovitve. Stavek *predZanko* takrat običajno uporabimo za nastavitev števca, v pogoju preverjamo, če je števec že dosegel določeno mejo, v stavku *poKoraku* pa povečujemo (ali zmanjšujemo) števec.
Vsako zanko *for* lahko zapišemo z enakovredno zanko *while*:

```
ukaz1
while(pogoj)
{
```

```
    stavki;
          ukaz2;
}
```

Zanke *for* so torej programske strukture, za katere so značilni naslednji elementi:

- števec - spremenljivka katere vrednost se spreminja med izvajanjem zanke,
- začetna vrednost je vrednost, ki določa začetno stanje števca,
- končna vrednost je vrednost, pri kateri se izvajanje zanke konča in
- korak zanke je vrednost, ki se prišteje števcu v eni ponovitvi zanke.

V vsakega izmed treh glavnih delov stavka *for* lahko združimo več stavkov, ki morajo biti ločeni z vejico. Pomembno pa je, da se tako zapisani stavki izvajajo od **leve proti desni**.

Ker je korak zanke najpogosteje 1, bomo v tretjem delu najpogosteje srečali obliko *stevec++*, ki, kot že vemo pomeni *stevec = stevec + 1*.

7.8.1 Izpis števil

Popravimo program *IzpisStevil* iz prejšnjega razdelka tako, da bomo uporabili zanko *for*. Izpisati želimo števila od *1* do *10* vsakega v svoji vrstici.

```
1:      static void Main(string[] args)
2:      {
3:          for(int stevilo = 1; stevilo <= 10; stevilo++)
4:          {
5:                  Console.WriteLine (stevilo);
6:          }
7:          Console.WriteLine ("Konec zanke");
8:      }
```

Razlaga.
Spremenljivki stevilo se najprej priredi začetna vrednost 1(*int stevilo = 1*). Nato se preveri pogoj (*stevilo <= 10*). Če je resničen, se izvedejo stavki v telesu zanke, sicer ne. V našem primeru je pogoj izpolnjen (1 <= 10), zato se izvede stavek v 5. vrstici, ki izpiše vrednost spremenljivke (1). Nato se izvede *ukaz2*, ki vrednost spremenljivke *stevilo* poveča za ena. Preveri se pogoj in ker še vedno velja (2 <= 10), se s stavkom v 5. vrstici zopet izpiše vrednost spremenljivke (2). Na enak način se *for* zanka izvaja, dokler je vrednost v spremenljivki *stevilo* manjša ali enaka 10. Ko je v spremenljivki vrednost 10, je pogoj še vedno izpolnjen. Ponovno se izvede stavek *Console.WriteLine (stevilo);*, ki izpiše 10. Nato se z *ukazom2* (*stevilo++*) vrednost v spremenljivki *stevilo* zopet poveča za ena in postane 11. Preveri se pogoj. Ker 11 ni manjše od 10, pogoj ni več izpolnjen, zato se zanka zaključi in program se nadaljuje v vrstici 7. Izpiše se besedilo *Konec zanke*.

7.8.2 Pravokotnik

Napišimo program, ki vpraša po širini in višini pravokotnika, nato pa izriše pravokotnik sestavljen s samih zvezdic.

Tu bomo uporabili gnezdeno zanko *for*. Tako kot pri pogojnem stavku, so tudi stavki v telesu zanke *for* poljubni stavki. In če uporabimo spet zanko *for* (ali tudi katero drugo zanko), je ta znotraj druge zanke *for* ("v gnezdu"). Zato rečemo, da gre za gnezdeno zanko *for*.

```
static void Main(string[] args)
{
     //  Vnos dimenzij pravokotnika
    Console.Write("Vnesi višino: ");
    int visina = int.Parse(Console.ReadLine());
    Console.Write("Vnesi širino: ");
    int sirina = int.Parse(Console.ReadLine());

     //  Preverimo, obstoj pravokotnika
    if (0 < visina && 0 < sirina)
    {
       Console.WriteLine();
        //  Izpisovanje vrstic
       for (int i = 1; i <= visina; i++)
       {
           //  Izpis stolpcev
          for (int j = 1; j <= sirina; j++)
          {
             Console.Write("*");
          }
           //  Po izpisu stolpcev, se pomaknemo v novo vrstico
          Console.WriteLine();
       }
    }
 }
}
```

7.8.3 Zanimiva števila

Sestavi program ki poišče vsa naravna števila med 0 in 10000, ki so enaka vsoti kubov svojih števk. Eno od števil, ki ima zahtevano lastnost, je npr. 153: $153 = 1^3 + 5^3 + 3^3$.

Ideja programa je v tem, da s pomočjo štirikratne zanke ustvarimo vsa možna števila med 0 in 9999.

```
static void Main(string[] args)
{
    int i, j, k, l;   // števke
    double st, vk;     // število in vsota kubov števk
    for (i = 0; i < 10; i++)
      for (j = 0; j < 10; j++)
        for (k = 0; k < 10; k++)
          for (l = 0; l < 10; l++)
          {
```

```
            st = 1000 * i + 100 * j + 10 * k + l;  //  sestavimo število
            //  vsota kubov
            vk = Math.Pow(i, 3) + Math.Pow(j, 3) + Math.Pow(k, 3)+ Math.Pow(l, 3);
            if (st == vk)
               Console.Write(st+",   ");
        }
}
```

Ali znaš napisati program tako, da uporabiš le enojno zanko?

7.8.4 Vzorec

Napiši program ki izpiše naslednji vzorec. Primer: za n = 5 je izpis takle:

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

```
static void Main(string[] args)
{
    int i,j;
    Console.Write("Velikost vzorca (1 - 9): ");
    int n = int.Parse(Console.ReadLine());
    for(i = n; i > 0; i--)
    {   //  z zunanjo zanko izpisujemo vrstice
       for(j = n; j >= i; j--) //  izpis posameznih števil v vrstici
          Console.Write(i);
       Console.WriteLine();  //  zaključimo tekočo vrstico
   }
}
```

Tu smo uporabili operator --. Ta zmanjša vrednost spremenljivke za 1. Zapis

```
i--;
```

torej pomeni

```
i = i – 1;
```

7.9 POVZETEK

Zanki *while* in *for* sta najpogosteje uporabljeni zanki, ki omogočata, da se en in isti del programske kode izvede večkrat. Kadar je število ponavljanj znano v naprej uporabimo *for* zanko, sicer pa zanko *while*. Predvsem včasih, ko še ni bilo vizuelnega programiranja, se je veliko uporabljala tudi zanka *do while*, ki so jo programerji pisali npr. za oblikovanje programov, ki so vključevali razne menije ipd. Danes pa se v sodobnih programskih pristopih vse pogosteje uporablja tudi četrta

zanka − *foreach*, ki pa jo v tem gradivu predvsem zaradi pomanjkanja prostora ne omenjava. Zahtevnejši bralec bo več o tej zanki izvedel npr. v gradivu Uranič S., Microsoft C#.NET, (online). Kranj, 2008. Dostopno na naslovu *http://uranic.tsckr.si/C%23/C%23.pdf*, str. 70 − 73.

Za zaključek spet najprej rešite kviz. Tokrat tistega, ki je na naslovu http://up.fmf.uni-lj.si/zanke-7c2f2bb5/index.html. Seveda brez samostojnega pisanja programov ne bo šlo. Ustrezne probleme najdete tako v zbirki nalog v poglavju o zankah, kot tudi na primer na http://penelope.fmf.uni-lj.si/diri0607/index.php/Kategorija:Zanke.

8 NAKLJUČNA ŠTEVILA

8.1 UVOD

V programiranju večkrat potrebujemo naključna števila. Ta imajo tako pri programiranju, kot tudi v vsakdanjem življenju velik pomen, zato je zelo pomembno, da generatorji takih števil generirajo res čimbolj naključna števila. Z njimi se srečamo npr. pri izdelavi preprostega programa za simulacijo meta kocke, nenadomestljivo vlogo imajo pri izdelavi kriptografskih aplikacij (generiranje naključnih gesel), potrebujemo jih npr. pri izdelavi programa, ki bo npr. iz množice telefonskih številk izbral nekaj naključnih za potrebe neke ankete, ipd.

8.2 GENERIRANJE NAKLJUČNIH ŠTEVIL

V jeziku C# je za generiranje naključnih števil na voljo razred (kar pač je že to) *Random*. Metode razreda *Random* uporabljamo tako, da najprej ustvarimo objekt tipa *Random*. To storimo s pomočjo operatorja ***new***. Kaj dejansko to pomeni, bomo pojasnili kasneje, ko bomo govorili o razredih in objektih.

```
Random nakljucno = new Random(); // generiranje objekta nakljucno za
                                 // generiranje naklj.števil
```

Objekt *nakljucno* sedaj lahko uporabimo za generiranje naključnih števil. Ta objekt si lahko predstavljamo kot nekakšen boben, iz katerega potem z različnimi metodami "žrebamo" naključna števila. Najpogostejše metode razreda *Random* so prikazane v spodnji tabeli.

Tabela 6: Metode za generiranje naključnih števil

C#	Razlaga
Next(n)	Naključno celo število tipa int.
Next(int n)	Naključno število tipa int iz intervala [0, n).
Next(int n, int m)	Naključno število tipa int iz intervala [n, m).
NextDoube()	Naključno število tipa double iz intervala [0, 1).

Primer uporabe:

```
Random nakljucno = new Random();

int naklj = nakljucno.Next(); // naključno nenegativno število

int naklj1 = nakljucno.Next(5);  // naključno število med vključno 0 in  vključno 4

int naklj2 = nakljucno.Next(10, 20);   // naključno število med vključno 10 in vključno 19

double naklj3 = nakljucno.NextDouble(); // naklj.število tipa double na intervalu od 0.0 do 1.0
```

```
double naklj4 = nakljucno.NextDouble() * 1000;  // naključno število tipa
                                                // double na intervalu od 0.0 do 1000
```

8.3 ZGLEDI

8.3.1 Kocka

Marko bi rad zvedel, kolikokrat mora vreči kocko, da bo skupaj vrgel 100 pik ali več. A ve, da je metanje kocke naporno opravilo. Zato mu napišimo program, ki bo simuliral metanje kocke tako, da bo "metal" kocko toliko časa, dokler ne bo vsota vseh pik presegla 100. Program naj po vsakem metu izpiše vržemo število pik in skupno vsoto. Na koncu naj izpiše število metov kocke, potrebnih, da je skupna vsota presegla 100.

V zanki bomo z ustrezno metodo razreda *Random* določili število pik pri posameznem metu. Dobljene pike bomo nato prišteli k skupni vsoti vseh pik.

```
static void Main(string[] args)
{
      // Ustvarimo generator naključnih števil
      Random nakljucno = new Random();

      // Zgornja meja vseh pik
      const int zgMeja = 100;
      // Šteje število pik
      int vsota = 0;
      // Šteje število metov
      int stevec = 0;

      // Simulacija meta kocke
      while(vsota <= zgMeja){
          // Število pik ob posameznem metu kocke
          int met = nakljucno.Next(6) + 1;
          vsota = vsota + met;
          Console.WriteLine("Zadnji met: " + met + "\tVsota: " + vsota);
          stevec = stevec + 1;
      }
      // Izpis števila metov kocke
      Console.WriteLine("\nŠtevilo metov kocke je " + stevec + ".");
}
```

Razlaga.
Najprej ustvarimo generator naključnih števil. Nato deklariramo ustrezne spremenljivke in jim priredimo ustrezne začetne vrednosti. S pomočjo zanke *while* simuliramo metanje kocke. Pri vsakem "metu" kocke določimo število dobljenih pik. Te pike prištejemo skupni vsoti pik. Dobljene pike in skupno vsoto pik na vsakem koraku izpišemo. Na vsakem koraku tudi povečujemo števec, s katerim štejemo število metov kocke oz. korake zanke. Na koncu še izpišemo število vseh metov kocke.

8.3.2 Ugibanje števila

Napišimo program, ki si bo "izbral" naključno število med 1 in 100, mi pa ga moramo v čim manj poskusih uganiti. Program naj glede na naše ugibanje pove, ali je povedano (vneseno) število premajhno ali preveliko. Ko število uganemo, naj izpiše število poskusov.

V zanki bomo uporabniku ponudili vnos števila in nato glede na vneseno število izpisali ustrezno obvestilo.

```
static void Main(string[] args)
{
    int stevec = 0;  //  šteje število poskusov
    Random boben = new Random(); //  boben naključnih števil
    int stevilo = -1;  //  število, ki ga vnesemo
     //  število, ki ga ugibamo
    int nakljucnoStevilo = boben.Next(1, 101);

    while (nakljucnoStevilo != stevilo)
    {
        Console.Write("Vnesi število:");
        string beri = Console.ReadLine();
        stevilo = int.Parse(beri);
        stevec++;  //  nov poskus

        if (stevilo < nakljucnoStevilo)
        {
            Console.WriteLine("Število " + stevilo +" je premajhno!");
        }  //  if
        else if (stevilo > nakljucnoStevilo)
        {
           Console.WriteLine("Število " + stevilo + " je preveliko!");
        }  //  else
    }  //  while

    Console.WriteLine("\n");
    Console.WriteLine("Bravo! Uganil si!");
    Console.WriteLine("Število je " + nakljucnoStevilo + ".");
    Console.WriteLine("Število poskusov: " + stevec);
}  //  main
```

Opis programa:

Najprej definiramo *stevec* s katerim bomo šteli število poskusov in *stevilo,* v katerem bomo hranili vneseno število. Slednjemu dodelimo vrednost -1, zakaj? Odgovor dobimo v vrstici, kjer si podrobneje oglejmo pogoj zanke *while*. V pogoju primerjamo *nakljucnoStevilo*, v katerem hranimo naključno število med 1 in 100 in *stevilo*. Če sta vrednosti spremenljivk različni, vstopimo v zanko. Da je delovanje programa pravilno, moramo v zanko vstopiti vsaj enkrat. To bomo zagotovili le če, pogoj na začetku ne bo izpolnjen. Ker v spremenljivki

nakljucnoStevlo hranimo cela števila med 1 in 100, moramo začetno vrednost spremenljivke *stevilo* nastaviti na katerokoli število, ki ni med 1 in 100, na primer na -1.
Znotraj zanke *while* sprašujemo po tej naključni številki. Če jo uganemo, se ob naslednjem poskusu izvajanja zanke, le-ta konča. Če število ni enako izbranemu, program izpiše, ali je naše število premajhno ali preveliko in nam tako pomaga čim hitreje uganiti število. Ko uganemo, se izpiše koliko poskusov smo potrebovali, da smo prišli do želenega števila.

8.4 POVZETEK

Naključni generatorji števil so zelo pomembni za varno delovanje številnih uporabniških aplikacij. Ampak ali so števila res naključna? Vsi generatorji naključnih števil v resnici delujejo po določenem pravilu, oziroma algoritmu, ki pa je lahko dober ali pa slab. Več o generatorjih naključnih števil lahko najdete npr. na spletni strani http://wiki.fmf.uni-lj.si/wiki/Generator_naklju%C4%8Dnih_%C5%A1tevil.

Kar številne naloge, ki predvidevajo uporabo naključnih števil, boste našli na wikiju v kategoriji Naključna števila. Dostopne so na naslovu http://penelope.fmf.uni-lj.si/diri0607/index.php/Kategorija:Naključna_števila

9 ZNAKI (TIP CHAR)

9.1 UVOD

V spremenljivki lahko hranimo tudi posamezen znak. Med znake prištevamo vse velike in male črke, vse cifre, posebne znake (npr. znak vejica, pika, dvopičje, …), pa tudi posebne nevidne (kontrolne) znake (npr. znak za prelom vrstice, znak tabulator,…). Kljub temu, da se v naših programih pogosteje uporablja zaporedje znakov – *string*, pa so tudi znaki sami zelo pomembni. Kako bomo sicer zgenerirali neko naključno geslo ali pa kako bomo zakodirali neko besedilo, da ga nepooblaščeni bralec ne bo razumel?

9.2 PODATKOVNI TIP CHAR

Znakovni podatkovni tip je označen z rezervirano besedo **char** (ang. *character*). To so črke, številke, presledek, ločila, ... Znake pišemo med enojnimi narekovaji.

Primer:

```
char znak = 'n';
char oznakaVrat = 'N';
char zacetnica = 'M';
```

Primer uporabe:

```
static void Main(string[] args)
{
   char znak_a = 'a';
   Console.WriteLine(znak_a);
}
```

C# obravnava znake kot "majhna" števila. Zato smo v spremenljivko *znak_a* pravzaprav shranili kodo znaka mali a. Ta koda je neko naravno število. Na vsako spremenljivko tipa *char* lahko gledamo bodisi kot na znak, bodisi kot na število. Ker na znake lahko gledamo tudi kot na števila, smemo napisati tudi tak program.

```
static void Main(string[] args)
{
    char znak_a = 'a';
    Console.WriteLine(znak_a + znak_a);
    Console.ReadKey();
}
```

V spremenljivko tipa *char* pravzaprav shranimo kodo znaka. Zato smo v spremenljivko *znak_a* shranili kodo znaka mali a, ki je neko naravno število (v našem primeru *97*). In ker je med dvema "številoma" operator +, se izvede operacija seštevanja, kot smo že navajeni. Torej se seštejeta dve številski vrednosti malega znaka a in vsota (197) se izpiše.

Razlikovati moramo med znakom in nizom, ki vsebuje en znak. Tako znak 'm' ni enak nizu, ki vsebuje le znak m, torej "m".

```
string znakKotNiz = "m";
char znakKotZnak = 'm';
```

Primer:

Napišimo program, ki bo izpisal angleško abecedo. Prvič naprej, drugič pa v obratnem vrstnem redu. Izrabili bomo dejstvo, da so znaki "števila". Ker z njimi lahko računamo, to pomeni tudi, da jih lahko uporabimo kot števce v zankah.

```
1:  static void Main(string[] args)
2:  {
3:      for(char i = 'a'; i <= 'z'; i++) //  abeceda naprej
4:      {
5:          Console.Write (i + " ");
6:      } //  for
7:      Console.WriteLine ("\n");  //  vmesna prazna vrsta
8:      for(char j = 'z';  j >= 'a' ;  j--)   //  abeceda nazaj
9:      {
10:         Console.Write (j + " ");
11:     }
12: }
```

Opis programa.

Kot smo že rekli, lahko na znake gledamo kot na števila. Zato smo v spremenljivki *i* pravzaprav shranili kodo znaka mali *a*. Ta koda je neko naravno število. Če so znaki kodirani po standardu ASCII, se v a shrani število 97. Preveri se pogoj in ker je 97 <= 122 (koda znaka 'z') se izpiše znak s kodo 97, torej a. Vrednost v spremenljivki *i* se poveča za ena (*i*++). Zopet se preveri pogoj in ker je resničen, se izpiše znak s kodo 98. Na ta način se zanka *for* izvaja, dokler je vrednost v spremenljivki i manjša ali enaka 'z' (122). Ko je v spremenljivki *i* vrednost 122, je pogoj izpolnjen in 5. vrstica izpiše znak s kodo 122, torej z. Sedaj spremenljivka *i* dobi vrednost 123. Preveri se pogoj. Ker 123 ni manjše ali enako 122, pogoj ni več izpolnjen. Zato se zanka konča in program se nadaljuje v vrstici 7. Izpiše se prazna vrsta. Program se nadaljuje v vrstici 8. Če želimo izpisati abecedo v obratnem vrstnem redu, začnemo izpisovati znake od zadnjega proti prvemu. V spremenljivki *j* shranimo kodo znaka mali z (122). Preveri se pogoj in ker je 122 >= 97 (koda znaka mali a), se izpiše znak s kodo 122, torej z. Vrednost v spremenljivki j se zmanjša za ena (*j*--) in zopet se preveri pogoj. Ker je resničen, se izpiše znak s kodo 121. Na ta način se zanka *for* izvaja, dokler je vrednost v spremenljivki *j* večja ali enaka 'a' (97). Ko je v spremenljivki vrednost 97, je pogoj izpolnjen in 10. vrstica izpiše znak s kodo 97, torej a. Sedaj spremenljivka i dobi vrednost 96. Preveri se pogoj. Ker 96 ni večje ali enako 97, pogoj ni več izpolnjen, zato se zanka konča.

9.3 PRIMERJANJE ZNAKOV

Ker v spremenljivko tipa *char* shranimo kodo znaka, znake primerjamo tako kot števila z operatorjem enakosti (==). S tem se v bistvu primerja koda znaka. Ker ima vsak znak svojo

kodo, lahko dobimo vrednost *true* samo, kadar primerjamo enaka znaka, drugače dobimo *false*. Poglejmo si primer.

```
static void Main(string[] args)
{
    char znak_a = 'a';
    char znak_A = 'A';
    Console.WriteLine(znak_a == znak_A);
    Console.WriteLine(znak_a == 'a');
}
```

Razlaga.
Kot smo že povedali, C# primerja znake po njihovih celoštevilčnih kodah. Ker koda malega in velikega znaka a ni enaka, se izpiše *false*. V naslednji vrstici primerjamo vrednost spremenljivke *znak_a* in znakovne konstante *'a'*. Ker je njuna celoštevilčna koda enaka, se izpiše *true*.

Za primerjanje znakov je pomembno vedeti, kako so znaki zloženi v kodnih tabelah. Velja:

- 'a' < 'b' < 'c' < … < 'z'
- '0' < '1' < … < '9'
- 'A' < 'B' < 'C' < … < 'Z'

Vmes v teh treh zaporedjih ni nobenih drugih znakov. Če torej napišemo pogoj

```
('0' <= preverjaniZnak) && (preverjaniZnak <= '9')
```

bo ta imel vrednost *true* (bo torej izpolnjen), če je v spremenljivki *preverjaniZnak* števka.

Podobno z

```
('a' <= malaČrka) && (malaČrka <= 'z')
```

preverjamo, če je v spremenljivki *malaČrka* res mala črka. Pogoj pomeni, da je znak v tej spremenljivki "desno" od 'a' in "levo" od 'z'. In ker med 'a' in 'z' ni drugih znakov, temu pogoj ustrezajo le male črke.

9.4 PRETVARJANJE ZNAKOV

S pomočjo operatorja (char) celo število lahko pretvorimo v znak. Tako na primer velja

(char)('a') → 'a'

In ker z znaki lahko računamo in so, kot smo videli v zgornjem razdelku, črke lepo "zložene skupaj", je seveda res tudi, da

(char)('a' + 2) → 'c'

To lahko izrabimo, če moramo slučajno malo črko pretvoriti v veliko ali obratno. Kratek premislek pokaže, da velja:

(char)('A' + 'k' – 'a') → 'K'
(char)('a' + 'M' – 'A') → 'm'

Kako to? Če želimo malo črko 'k' pretvoriti v veliko, vemo, da bo razlika v kodi med 'k' in 'a' enaka razliki kod 'K' in 'A'. Torej moramo od kode 'A' iti naprej točno za razliko med kodo 'k' in 'a', da pridemo do kode znaka 'K'. Če je v spremenljivki *maliZnak* torej neka mala črka, bomo z

(char)('A' + maliZnak – 'a')

dobili ustrezno veliko črko.

Oglejmo si še zgled uporabe.

9.5 ZGLED

9.5.1 Geslo

Iz oddelka za varovanje podatkov smo dobili nalogo, da napišemo program za samodejno generiranje gesel. Sodelavci tega oddelka želijo gesla naslednje oblike:

- najprej dve naključni števki,
- potem tri naključne velike črke,
- nato še ena naključna števka in
- na koncu naključno trimestno število.

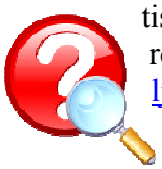
Pri generiranju naključnih črk si bomo pomagali z dejstvom, da je znak predstavljen kar s številom, ki predstavlja njegovo kodo. Z njim torej lahko računamo. Izraz *geslo.Next('A', 'Z' + 1)* je torej koda neke velike črke. Za pretvarjanje iz celega števila v znak bomo uporabili operator (*char*).

```
static void Main(string[] args)
{      // Ustvarimo generator naključnih števil
       Random geslo = new Random();
       // Določimo dve števki
      int stev1 = geslo.Next(10);
      int stev2 = geslo.Next(10);
       // Določimo tri naključne črke (velike tiskane)
      char c1 = (char)geslo.Next('A', 'Z' + 1);
      char c2 = (char)geslo.Next('A', 'Z' + 1);
      char c3 = (char)geslo.Next('A', 'Z' + 1);
       // Določimo naključno števko
      int stev3 = geslo.Next(10);
       // Določimo naključno tromestno število
      int s = geslo.Next(100, 1000);
       // Izpis gesla
     Console.WriteLine("Geslo: "+stev1+stev2 +c1 +c2 +c3+stev3+ s);
```

```
    Console.ReadKey();
}
```

Razlaga

Najprej ustvarimo generator naključnih števil. Nato generiramo posamezne elemente gesla. Za generiranje naključne črke uporabimo izraz *geslo.Next('A', 'Z' + 1)*. Z omenjenim izrazom določimo kodo črke. Ker se koda črke (tipa *int*) ne spremeni v črko (tip *char*) samodejno, pred omenjenim izrazom zapišemo operator (*char*). S tem operatorjem iz kode dobimo naključno črko, ki jo nato priredimo spremenljivki. Po generiranju združimo vse elemente in jih izpišemo na zaslon.



Preden si ogledamo znake v povezavi z večjo »enoto«, nizi, pa rešite vsaj tri naloge med tistimi, ki jih najdete v zbirki Lokar M. in Uranič S., Programiranje 1 - zbirka rešenih nalog v C# in na naslovu http://penelope.fmf.uni-lj.si/diri0607/index.php/Kategorija:Znaki.

10 NIZI (TIP STRING)

10.1 UVOD

Nizi (podatkovni tip *string*) so zelo pomemben podatkovni tip, ki pride v poštev vselej, kadar imamo opraviti z nenumeričnimi informacijami, kot so npr. imena oseb, krajev, planetov, … Niz sestavlja množica znakov, ki so med seboj povezani v celoto. V C# imamo vgrajene mehanizme za zelo učinkovito obravnavanje nizov, poleg tega pa ogromno število metod za manipulacijo z nizi.

10.2 DEKLARACIJA SPREMENLJIVKE TIPA NIZ

Nize (ang. *string*) uporabljamo pogosteje kot posamezne znake. Za delo z nizi je v jeziku C# definiran tip *string*. Željen niz navedemo med dvojnimi narekovaji. Podatkovni tip *string* (niz) torej označuje zaporedje znakov. Spremenljivke tega tipa v C# lahko inicializiramo takole:

```
string var5;
var5 = "Lokomotiva";
string niz = "Pozdravljeni!";
string presledek = " ";
```

Dva niza staknemo (združimo) z operatorjem za združevanje (+). To smo že večkrat srečali pri izpisovanju.

```
static void Main(string[] args)
{
    string niz1 = "Dobro ";
    string niz2 = "jutro.";
    string niz3 = niz1 + niz2;

    Console.WriteLine(niz3);
    Console.ReadKey();
}
```

Niz je lahko sestavljen iz enega ali več znakov, poznamo pa tudi t.i. prazen niz, ki ne vsebuje nobenega znaka.

```
string prazenNiz = "";
```

10.3 DOSTOP DO ZNAKOV V NIZU

Niz je lahko torej prazen, ali pa sestavljen iz enega ali več znakov. Vsak znak, ki je vsebovan v nizu, ima svoj indeks. Do posameznega znaka v nizu dostopamo s pomočjo oglatih oklepajev ([]), kjer navedemo indeks znaka. Z *niz[i]* torej dobimo znak z indeksom i iz niza *niz*.

- "Blabla"[3] → b
- string ime = "Matija";

- ime[1] → a

Indeksiranje znakov se prične z 0 in se konča z dolžina niza - 1.

Primer:

Niz	**"D**	**o**	**b**	**e**	**r**		**d**	**a**	**n**	**."**
Indeks	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Z uporabo oglatih oklepajev ne moremo spreminjati znakov v nizu, temveč jih lahko le beremo. Izraz *niz[indeks]* se torej ne sme pojaviti na levi strani prireditvenega stavka.

10.4 DOLŽINA NIZA

Da zvemo dolžino niza, uporabljamo lastnost *Length*.

Primer:

```
string n = "Rock Otočec";
int dolzina = n.Length;
```

Vrednost spremenljivke dolzina je 11.

- string priimek = "Lokar";
- priimek.Length → 5
- "matija".Length → 6
- (priimek + " " + "bla").Length → 9

10.5 ZGLEDI

10.5.1 Obratni izpis

Napišimo program, ki bo vneseni niz izpisal v obratnem vrstnem redu. Torej bo npr. niz *Matija* izpisal kot *ajitaM*.

Tu si bomo pomagali z *[i]*, ki vrne znak z indeksom *i* ter z lastnostjo *Length*, ki vrne dolžino niza. Ker v nizih znake indeksiramo od 0 dalje, nam *niz.[i]* vrne i+1-ti znak niza *niz*.

Ideja programa je v tem, da naredimo zanko, v kateri izpišemo posamezen znak. Začnemo pri zadnjem znaku in zmanjšujemo števec, ki pomeni indeks, dokler ne pridemo do vrednosti števca 0 (torej do zadnjega znaka), ko še zadnjič izvedemo zanko.

```
1:   static void Main(string[] args)
2:   {
3:      Console.WriteLine("Vnesi niz:");
4:      string beri = Console.ReadLine();
5:      Console.WriteLine("Vneseni niz je: " + beri);
6:
```

```
7:      for(int i = beri.Length - 1; i >= 0; i--)
8:      {
9:         Console.Write (beri[i]);
10:     }
11:     Console.WriteLine ("\n");
12:  }
```

Opis programa

Če želimo niz izpisati v obratnem vrstnem redu, bomo izpisovali znake od zadnjega proti prvemu. Recimo, da smo vnesli niz *"abeceda"*. Spremenljivki *i* se najprej priredi začetna vrednost *beri.Length - 1*. Dolžina niza je enaka številu vnesenih znakov, kar je v našem primeru 7. Ker se znaki štejejo od 0 dalje, ima naš zadnji znak indeks *Length - 1*. V našem primeru se spremenljivki *i* najprej priredi začetna vrednost 6. Nato se preveri pogoj. Ker je resničen *(6 >= 0)*, vstopimo v zanko in izpiše se znak na mestu *i (a)*. Nato se izvede ukaz2 *(i--)*, ki vrednost spremenljivke *i* zmanjša za ena. Preveri se pogoj in ker še vedno velja *(5 >= 0)*, se zopet izpiše znak na mestu *5* (*d*). Na ta način se zanka *for* izvaja, dokler je vrednost v spremenljivki *i* večja ali enaka *0*. Ko je v spremenljivki vrednost *0*, je pogoj še vedno izpolnjen in v vrstici 9 se izpiše znak na mestu *0,* torej prvi znak. Sedaj spremenljivka *i* dobi vrednost *-1*. Preveri se pogoj. Ker *-1* ni večje od *0*, pogoj ni več izpolnjen. Zanka se konča in program se nadaljuje v vrstici 11. Izpiše se prazna vrsta.

10.5.2 Obratni niz

Kaj pa, če bi rad opravili nekoliko drugačno nalogo in iz vnesenega niza naredili obrnjeni niz nekoliko drugače. Tudi tu bomo uporabili zanko, a za razliko od prej bomo šli kar od indeksa 0 naprej. Uporabili bomo tudi stikanje nizov. Če znak dodamo nizu, se znak pretvori v ustrezni niz in niza se stakneta.

- Začnemo s praznim nizom
  - string obrnjeniNiz = "";
- Zanka
- Pregledamo vse znake v nizu (dolžina niza)
  - while (i < niz.Length)
- Dodajamo na začetek
  - obrnjeniNiz = niz[i] + obrnjeniNiz;

```
static void Main(string[] args)
{
    Console.WriteLine("Vnesi niz:");
    string beri = Console.ReadLine();
    Console.WriteLine("Vneseni niz je: " + beri);
    int i = 0;
    string obrnjeniNiz = "";
    while (i < beri.Length)
    {
        obrnjeniNiz = beri[i] + obrnjeniNiz;
        i++;
    }
```

```
    Console.WriteLine("\nObrnjeni niz: " + obrnjeniNiz);
}
```

10.6 PRIMERJANJE NIZOV

Oglejmo si, kako lahko ugotovimo, da sta dva niza enaka (ang. *equals*)? V razredu string najdemo metodo *Equals()*, ki vrača rezultat tipa *bool*.

```
static void Main(string[] args)
{
   string niz1 = "Ana";
   string niz2 = "Zdravko";
   bool trditev = niz1.Equals(niz2);

   Console.WriteLine(trditev);  //  Izpis:  False
}
```

V programu smo uporabili metodo *Equals()*. Primerjali smo *niz1* in *niz2*. Ker sta različna, nam metoda vrne *false*.

Nize pa lahko glede enakosti primerjamo tudi z relacijskim operatorjem ==. Zgornji zgled bi torej lahko napisali kot

```
static void Main(string[] args)
{
    string niz1 = "Ana";
    string niz2 = "Zdravko";
    bool trditev = niz1 = = niz2;

   Console.WriteLine(trditev);
}
```

Večkrat bi radi niza primerjali (ang. *compare*) po abecednem redu. Tako je niz *"Matija"* je manjši kot niz *"Mojca",* ker sta prva znaka enaka, drugi znak pa je v prvem nizu ('a') manjši kot v drugem nizu ('o').

Tu ne moremo uporabiti relacijskih operatorjev <, >, <=, >=. Na voljo imamo metodo *String.Compare()*, ki vrača celoštevilčno vrednost. Tako

```
String.Compare(s1,"bla")
```

vrne 0, če je niz shranjen v *s1* enak nizu *bla*, neg. število, če je niz v *s1* manjši od niza *bla* in poz. število, če je večji.

```
int n = String.Compare("matija", "mojca");  //  n dobi vrednost -1
bool jeManj = String.Compare("Ana", "Zdravko") < 0;   // jeManj dobi vrednost True
```

Kako si zapomniti, kako to uporabiti? Zanima nas če velja

```
niz1 <= niz2
```

(če je torej niz1 manjši (prej po abecedi) ali enak nizu niz2). Namesto zgornjega izraza, ki ga prevajalnik "ne mara", napišemo izraz

```
String.Compare(niz1,niz2) <= 0
```

Za primerjanje nizov torej uporabimo ustrezen operator in primerjamo z 0.

Poglejmo, kako deluje, oziroma, kako jo uporabljamo.

```
static void Main(string[] args)
{
    string niz1 = "Ana";
    string niz2 = "Zdravko";
    int vrednost = String.Compare(niz1,niz2);

    Console.WriteLine(vrednost);
    Console.ReadKey();
}
```

Program prevedemo in poženemo: izpiše se vrednost *-1*.

Obstaja pa tudi malo drugačna metoda *CompareTo(),* ki se več ali manj razlikuje le po obliki (načinu klica). No resnici na ljubo je to "okleščena" različica metode *String.Compare()*, ki poleg tega, kar smo spoznali mi, "zna" še mnogo več. Ustrezna oblike te metode je

```
niz1.CompareTo(niz2)
```

kar ustreza klicu

```
String.Compare(niz1, niz2)
```

Prejšnje zglede navedimo še v novi obliki. Tako

```
S1.CompareTo("bla")
```

vrne 0, če je niz shranjen v *s1* enak nizu *bla*, neg. število, če je niz v *s1* manjši od niza *bla* in poz. število, če je večji.

```
"matija".CompareTo("mojca")  //  vrne negativno število
bool jeManj = n1.CompareTo(n2) < 0;
```

Poglejmo še enkrat, kaj nam pove metoda *CompareTo()*. Denimo, da primerjamo *niz1* in *niz2* z *niz1.CompareTo(niz2)*. Če je niz v spremenljivki *niz1* po abecedi pred nizom, ki je v spremenljivki *niz2*, vrne metoda negativno število. Če je *niz1* enak *niz2*, vrne metoda vrednost 0. Če je *niz1* po abecedi za *niz2*, vrne metoda pozitivno število.

10.7 METODE ZA DELO Z NIZI

Za delo z nizi poleg omenjenih metod, lastnosti in operatorja [] pogosteje uporabljamo še metode, ki so prikazane v spodnji tabeli. Da bomo imeli zbrane vse, sta v tabeli še metodi *Equals* in *CompareTo*. Vse te metode uporabljamo nad nizi. To pomeni, da jih kličemo kot *niz1.Metoda(niz2)*.

Tabela 7: Metode za delo z nizi

Metoda	Razlaga
Equals(niz)	S postopkom preverjamo enakost niza *niz* in niza, nad katerim uporabljamo metodo. Ukaz vrne vrednost *true*, če sta niza enaka.
CompareTo(niz)	S postopkom primerjamo dva niza po abecednem položaju. Če je vrnjena vrednost: • pozitivna, če je niz po abecedi pred nizom, nad katerim je uporabljena metoda. • negativna, če je niz po abecedi za uporabljenim nizom. • nič, če sta niz in uporabljeni niz enaka (ju sestavljajo isti znaki).
ToUpper()	Metoda spremeni v nizu, nad katerim je uporabljena,vse male črke v velike.
ToLower()	Metoda spremeni v nizu, nad katerim je uporabljena, vse velike črke v male.
Substring(int,int)	Metoda vrne podniz danega niza. Prvi argument pomeni začetni položaj (indeks) v nizu. Drugi argument pomeni dolžino podniza. Če drugega argumenta ni, metoda vrne vse znake do konca niza.
IndexOf(niz)	S postopkom preverimo, če je niz *podniz* v nizu, nad katerim metodo uporabljamo. Metoda vrne celo število, ki predstavlja na katerem indeksu se v nizu prvič kot *podniz* prične niz *niz*. Če *niz* ni *podniz* uporabljenega niza, je vrnjena vrednost -1.

Ilustrirajmo uporabo omenjenih metod z nekaj zgledi.

10.7.1 Samoglasniki

Sestavimo program, ki prebere niz in ga izpiše tako, da vse samoglasnike nadomesti z zvezdico (*).

Pomagali si bomo z metodo *IndexOf()*, s katero bomo preverili, če je posamezen znak prebranega niza podniz niza "aAeEiIoOuU ".
Posamezne znake prebranega niza, ki jih bomo bodisi spremenili bodisi ohranili, bomo dodali v pomožni niz. Pomožni niz bo na začetku programa prazen ("").

```
1:      static void Main(string[] args)
2:      {
```

```
3:          // Vnos niza
4:          Console.Write("Vnesi niz: ");
5:          string niz = Console.ReadLine();
6:          // Pomožne spremenljivke
7:          string samoglasniki = "aAeEiIoOuU";  // Niz samoglasnikov
8:          string novNiz = "";  // Nov niz
9:          int dolzina = niz.Length;  // Dolžina prebranega niza
10:         // Zamenjava samoglasnikov z zvezdico
11:         for (int i = 0; i < dolzina; i++)
12:         {
13:               char znak = niz[i];
14:               if (samoglasniki.IndexOf(znak) != -1)
15:               {  // Znak je samoglasnik
16:                     novNiz = novNiz + '*';  // Samoglasnik zamenjamo z *
17:               }
18:               else
19:                     novNiz = novNiz + znak; // Ostali znaki ostanejo nespremenjeni
20:         }
21:         // Izpis novega niza
22:         Console.WriteLine("Spremenjen niz: " + novNiz);
23:   }
```

Razlaga. Najprej preberemo niz, nato deklariramo niz samoglasnikov in prazen niz. Potem določimo dolžino niza niz (vrstica 9). Z zanko se sprehodimo preko vseh znakov v nizu. Znotraj zanke deklariramo spremenljivko znak in ji priredimo trenutni znak niza niz. V vrstici 14 preverimo, če je znak, katerega koda je shranjena v spremenljivki znak, podniz niza samoglasniki. Če je pogoj izpolnjen, se nizu *novNiz* doda znak '*' (vrstica 16), sicer pa se mu doda trenutno preverjeni znak (19). Na koncu izvedemo izpis niza *novNiz*.

10.8 POVZETEK

Povzemimo še vse skupaj v tabelo in dodajmo še nekaj uporabnejših metod.

Tabela 8: Še nekaj metod za delo z nizi

Indeks	**Razlaga**
[indeks]	Dostop do znaka na določeni poziciji.
Lastnost	**Razlaga**
Length	Število znakov nizu.
Metoda	**Razlaga**
StartsWith(string)	Vrne logično vrednost, ki označuje, ali se nek niz začenja z navedenim nizom.
EndsWith(string)	Vrne logično vrednost, ki označuje, ali se nek niz končuje z navedenim nizom.

IndexOf(niz)	S postopkom preverimo, če je niz podniz v nizu, nad katerim metodo uporabljamo. Metoda vrne celo število, ki predstavlja na katerem indeksu se v nizu prvič kot podniz prične niz *niz*. Če niz ni podniz uporabljenega niza, je vrnjena vrednost -1.
IndexOf(string, začetni indeks)	Vrne celo število ki predstavlja mesto (indeks), kjer se navedeni (pod)niz v nekem nizu od začetnega indeksa naprej prvič pojavi. Če navedenega (pod)niza ni, je vrnjena vrednost -1.
Insert(začetni indeks, string)	Vrne niz, v katerega je na navedeno mesto (začetni indeks) vrinjen navedeni niz.
Equals(niz)	S postopkom preverjamo enakost niza niz in niza, nad katerim uporabljamo metodo. Ukaz vrne vrednost true, če sta niza enaka.
CompareTo(niz)	S postopkom primerjamo dva niza po abecednem položaju. Če je vrnjena vrednost: • pozitivna, je niz po abecedi pred nizom, nad katerim je uporabljena metoda. • negativna, je niz po abecedi za uporabljenim nizom. • nič, sta niz in uporabljeni niz enaka (ju sestavljajo isti znaki).
ToUpper()	Metoda spremeni v nizu, nad katerim je uporabljena, vse male črke v velike.
ToLower()	Metoda spremeni v nizu, nad katerim je uporabljena, vse velike črke v male.
Substring(začetni_indeks)	Vrne del niza, ki se začne na navedenem mestu in vsebuje vse znake do konca niza.
Substring(začetni_indeks, dolžina)	Vrne del niza, ki se začne na navedenem mestu in ima navedeno dolžino.
PadLeft(skupna_dolžina)	Vrne niz, ki je DESNO poravnan in na levi strani zapolnjen s tolikšnim številom presledkov, da je skupno število znakov enako vrednosti skupna_dolžina.
PadRight(skupna_dolžina)	Vrne niz, ki je LEVO poravnan in na desni strani zapolnjen s tolikšnim številom presledkov, da je skupno število znakov enako vrednosti skupna_dolžina.
Remove(začetni_indeks, N)	Vrne niz, iz katerega je odstranjeno N znakov od mesta začetni_indeks naprej.
Replace(staristring, novistring)	Vrne niz, v katerem je na vsakem mestu, kjer je bil v starem nizu (pod)niz *stariniz*, sedaj (pod)niz *noviniz*.
Trim()	Vrne niz iz katerega so odstranjeni vsi vodilni in končni presledki.

Tabela nam bo prav prišla tako pri reševanju kviza kot pri pisanju programov. Ustrezni vir za kviz in ideje za naloge so gradiva projekta UP - Kako poučevati začetni tečaj programskega jezika, sklop interaktivnih gradiv. Kviz je dostopen na naslovu http://up.fmf.uni-lj.si/nizi-ed665584/index.html. Na naslovu http://up.fmf.uni-lj.si/index.html#h2_21 pa so v razdelku Naloge iz programiranja zbrane naloge, ki za reševanje predvidevajo uporabo nizov v jeziku C#.

11 TABELE

11.1 UVOD

Pogosto se srečamo z večjo količino podatkov istega tipa, nad katerimi želimo izvajati podobne (ali enake) operacije. Takrat si pomagamo s tabelami. Tabelo ali polje (ang. *array*) v jeziku C# uporabljamo torej v primerih, ko moramo na enak način obdelati skupino vrednosti istega tipa. Oglejmo si tak zgled. Tabelarične spremenljivke, kot jih poznamo iz srednješolske matematike, imajo vse enako ime (npr. tabela), razlikujejo pa se po **indeksu**. Zaradi tega vsako od njih obravnavamo kot samostojno spremenljivko.

Dana je naloga: *Preberi 5 števil in jih izpiši v obratnem vrstnem redu*. Vsa števila moramo prebrati (shraniti), ker jih bomo potrebovali šele, ko bodo prebrana vsa. Ustrezen bo na primer tak program:

```
static void Main(string[] args)
{
    // branje
    Console.WriteLine("Vnesi število:");
    string beri = Console.ReadLine();
    int x1 = int.Parse(beri);
    Console.WriteLine("Vnesi število:");
    beri = Console.ReadLine();
    int x2 = int.Parse(beri);
    Console.WriteLine("Vnesi število:");
    beri = Console.ReadLine();
    int x3 = int.Parse(beri);
    Console.WriteLine("Vnesi število:");
    beri = Console.ReadLine();
    int x4 = int.Parse(beri);
    Console.WriteLine("Vnesi število:");
    beri = Console.ReadLine();
    int x5 = int.Parse(beri);
    // izpis
    Console.WriteLine("Števila v obratnem vrstnem redu: ");
    Console.WriteLine(x5);
    Console.WriteLine(x4);
    Console.WriteLine(x3);
    Console.WriteLine(x2);
    Console.WriteLine(x1);
}
```

Če malo pomislimo, vidimo, da v našem program pravzaprav ponavljamo dva sklopa ukazov. Prvi je

```
Console.WriteLine("Vnesi število:");
beri = Console.ReadLine();
int x2 = int.Parse(beri);
```

in drugi

```
Console.WriteLine(x3);
```

Vsak sklop ponovimo 5x. Razlikujejo se le po tem da v njih nastopa enkrat *x2*, drugič *x3*, tretjič *x4* … To nam da misliti, da bi lahko uporabili zanko:

zanka
    preberi *i*-to število
    shrani ga v *xi*
    Povečaj *i* za 1
Naj bo *i* = prebrano število števil
zanka
    izpiši *xi*
    zmanjšaj *i* za 1

```
static void Main(string[] args)
{
    //  branje
   string beri;
   for (int i = 1; i <= 50; i++)
   {
      Console.WriteLine("Vnesi število:");
      beri = Console.ReadLine();
      int xi = int.Parse(beri);
   }
   for (int i = 50; i >= 1; i--)
      Console.WriteLine(xi);
}
```

Prevajalniku tisti *xi* ni najbolj všeč in v drugi zanki pravi, da "ne obstaja". Razlog je v tem, da so tisti naši *xi* deklarirani v prvi zanki in jih "drugje ni". Torej jih moramo deklarirati zunaj.

Spremenimo program takole:

```
static void Main(string[] args)
{
    //  branje
   string beri;
   int x1, x2, x3, x4, x5;
   for (int i = 1; i <= 5; i++)
   {
      Console.WriteLine("Vnesi število:");
      beri = Console.ReadLine();
      xi = int.Parse(beri);
   }
   for (int i = 5; i >= 1; i--)
```

```
        Console.WriteLine(xi);
}
```

A prevajalnik pravi, da ne pozna spremenljivke *xi*? No, morda pa bi bilo dobro deklarirati še to in dodamo

```
int xi;
```

A ko poženemo program, le ta 5x izpiše zadnje vneseno število. Kaj se dogaja?

11.2 INDEKSI

Mi bi v prejšnjem zgledu želeli, da *xi* pomeni *x1*, *x2*, *x3*, ... (glede na vrednost *i*). Prevajalnik pa trmasto vztraja, da je to spremenljivka z imenom *xi* (torej ena sama). Radi bi torej, da so 1, 2, 3 indeksi spremenljivke. Kako pa indekse napišemo? Indeks napišemo za imenom tabele med oglatimi oklepaji (*[]*). Tako zapis *igralci[4]* pomeni element z indeksom 4 tabele *igralci*.

Torej bi prejšnji zgled morali napisati kot

```
static void Main(string[] args)
{
    //  branje
   string beri;
   int x1, x2, x3, x4, x5;
   for (int i = 1; i <= 5; i++)
   {
        Console.WriteLine("Vnesi število:");
        beri = Console.ReadLine();
        x[i] = int.Parse(beri);
   }
   for (int i = 5; i >= 1; i--)
   {
      Console.WriteLine(x[i]);
    }
}
```

Vendar žal prevajalnik še vedno ni zadovoljen. Namreč pozna le spremenljivke *x1*, *x2*, ... ne ve pa, da je *x* ime tabele, kjer bi radi imeli 5 indeksov (torej prostor za 5 števil).

11.3 DEKLARACIJA TABELE

Prvi korak pri ustvarjanju tabele je **najava** le te. To storimo tako, da za tipom tabele navedemo oglate oklepaje in ime tabele.

podatkovniTip[] imeTabele;

Z zgornjim stavkom zgolj napovemo spremenljivko, ki bo hranila naslov bodoče tabele, ne zasedemo pa še nobene pomnilniške lokacije, kjer bodo elementi tabele dejansko shranjeni. Potrebno količino zagotovimo in s tem tabelo dokončno pripravimo z ukazom *new*:

imeTabele = new podatkovniTip[velikost];

Ukaz *new* zasede dovolj pomnilnika, da vanj lahko shranimo *dolzina* spremenljivk ustreznega tipa in vrne njegov naslov. Velikost tabele je enaka številu elementov, ki jih lahko hranimo v tabeli.

Napoved in zasedanje pomnilniške lokacije lahko združimo v en stavek:

podatkovniTip[] imeTabele = new podatkovniTip[velikost];

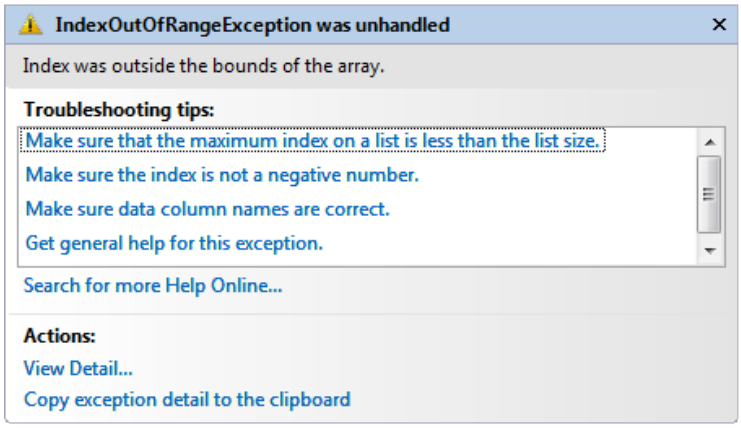
Primeri najave tabele:

```
 //  tabela 10 celih števil
int[] stevila = new int[10];
 //  tabela 3 realnih števil
double[] cena = new double[3];
 //  tabela 4 znakov
char[] tabelaZnakov = new char[4];
 //  tabela 500 nizov
string[] ime = new string[500];
```

Sedaj lahko končno naš zgled napišemo prav.

```
static void Main(string[] args)
{
    //  branje
   string beri;
   int[] x = new int[5];
   for (int i = 1; i <= 5; i++)
   {
      Console.WriteLine("Vnesi število:");
      beri = Console.ReadLine();
      x[i] = int.Parse(beri);
   }
   for (int i = 5; i >= 1; i--)
      Console.WriteLine(x[i]);
}
```

A med izvajanjem se program "sesuje" s sporočilom



Slika 20: Napaka pri prekoračitvi indeksa

Zakaj? Grafično si lahko enodimenzionalno tabelo predstavljamo takole:

```
int[] tab = new int[5]; // ime tabelarične spremenljivke je tab
```

tab[0]	tab[1]	tab[2]	tab[3]	tab[4]

```
tab[4] = 100;
```

peti element v tabeli (indeksi se označujejo od 0 naprej !!!)

ime tabele (polja) indeks vrednost

Vsaka tabelarična spremenljivka ima svoj indeks, preko katerega ji določimo vrednost (jo inicializiramo) ali pa jo uporabljamo tako kot običajno spremenljivko. Začetni indeks je enak nič (0) končni indeks pa je za ena manjši kot je dimenzija tabele (dimenzija – 1).

S posameznimi elementi tabele lahko operiramo enako kot s spremenljivkami tega tipa. Tako je v spodnjem zgledu npr. *tabela[4]* povsem običajna spremenljivka tipa *int*.

Primer:

```
// tabelo napolnimo z vrednostmi
tabela[0] = 10;
tabela[1] = 20;
tabela[2] = 31;
tabela[3] = 74;
tabela[4] = 97;
tabela[5] = 31;
```

Posameznim tabelaričnim spremenljivkam (komponentam) lahko prirejamo vrednost, ali pa jih uporabljamo v izrazih, npr.:

```
int[] tab = new int[5];
```

```
tab[0] = 100;
tab[1] = tab[0]-20;      //  tab[1] dobi vrednost 80
tab[2] = 300;
tab[3] = 400;
tab[4] = tab[1] + tab[3];  //  tab[4] dobi vrednost 480
```

Izgled tabele po izvedbi zgornjih stavkov je torej takle:

100	80	300	400	480

Pri uporabi indeksov pa moramo paziti, da ne zaidemo izven predpisanih meja. In ker smo v našem primeru uporabljali *x[5]* (ko je bil *i* == 5), smo dobili napako *IndexOutOfRangeException*, ki vedno pomeni, da smo zašli izven tabele.

Če sedaj upoštevamo povedano, vidimo, da bomo namesto indeksov 1, 2, 3, 4 in 5 uporabili indekse 0, 1, 2, 3, in 4. S programom

```
static void Main(string[] args)
{
    //  branje
   string beri;
   int[] x = new int[5];
   for (int i = 0; i <= 4; i++)
   {
      Console.WriteLine("Vnesi število:");
      beri = Console.ReadLine();
      x[i] = int.Parse(beri);
   }
   for (int i = 4; i >= 0; i--)
      Console.WriteLine(x[i]);
}
```

pa končno dobimo želeni rezultat!

Najbolj uporabna lastnost tabel je ta, da jih zlahka uporabljamo v zankah. S spreminjanjem indeksov v zanki poskrbimo, da se operacije izvedejo nad vsemi elementi tabele.

Deklarirajmo tabelo 100 celih števil in jo inicializirajmo tako, da bodo imeli vsi elementi tabele vrednost 1.

```
int[] tabela = new int[100];

for (int i = 0; i < 100; i++)
{
   tabela[i] = 1;
}
```

Velikost tabele lahko določimo med samim delovanjem programa. Dimenzijo tabele torej lahko določi uporabnik, glede na svoje potrebe:

```
Console.Write("Določi dimenzijo tabele: ");
 // Dimenzijo tabele določi uporabnik med izvajanjem!!!
int dimenzija = int.Parse(Console.ReadLine());
int[] tabela = new int[dimenzija];
```

Sočasno z najavo lahko elementom določimo tudi začetne vrednosti. Te zapišemo med zavita oklepaja. Posamezne vrednosti so ločene z vejico.

Primer:

```
int[] stevila = {1, 2, 3, 6, 8, 12, 14, 4, 7, 9};
```

Ob naštevanju vrednosti ne moremo navesti dolžine tabele, saj jo prevajalnik izračuna sam. Paziti moramo, da vsi elementi ustrezajo izbranemu tipu.

Povzemimo vse načine deklaracije tabele:

1. način: Najprej najavimo ime tabele. Nato z operatorjem *new* dejansko ustvarimo tabelo.

Primer:

```
int[] tabela = new int[5];  //  ustvarimo tabelo dolžine 5
int[] tab1;  //  Vemo, da je tab1 ime tabele celih števil,
 //  tabela kot taka pa še ne obstaja
tab1 = new int[20];  //  tab1 kaže na tabelo 20 celih števil
```

Seveda ni nujno, da si ukaza sledita neposredno drug za drugim. Prav tako ni nujno, da za določanje velikosti uporabimo konstanto. Napišemo lahko poljubni izraz.

```
tab1 = new int[2 + 5];  //  tab1 kaže na tabelo 7 celih števil
```

kjer lahko nastopajo tudi spremenljivke, ki imajo v tistem trenutku že prirejeno vrednost

```
int vel = 10;
tab1 = new int[2 * vel];  //  tab1 kaže na tabelo 20 celih števil
```

ki smo jo seveda lahko tudi prebrali

```
int[] tab = new int[int.Parse(Console.ReadLine())];
```

No, zgornjo "kolobocijo" raje napišimo kot

```
string beri = Console.ReadLine();
int velTab = int.Parse(beri);
```

```
int[] tab = new int[velTab];
```

2. način: Oba zgornja koraka združimo v enega.

Primera:

```
int[] tab1 = new int[2];  //  Vsi elementi so samodejno nastavljeni na 0
double[]tab2 = new double[5]; // Vsi elementi so samodejno nastavljeni na 0.0
```

Tako v 1. in 2. načinu velja, da se, ko z *new* ustvarimo tabelo, vsi elementi samodejno nastavijo na začetno vrednost, ki jo določa tip tabele (npr. *int* - 0, *double* - 0.0, *bool* - *false*).

3. način: Tabelo najprej najavimo, z operatorjem *new* zasedemo pomnilniško lokacijo, nato pa elementom določimo začetno vrednost.

Primer:

```
int[] tab3 = new int[] { -7, 5, 65 };  //  Elementi so -7, 5 in 65
```

lahko pa smo tudi "pridni" in velikost povemo še sami

Primer:

```
int[] tab3 = new int[3] { -7, 5, 65 };  //  Elementi so -7, 5 in 65
```

4. način: Tabelo najprej najavimo, nato pa elementom določimo začetno vrednost. Velikosti tabele ni potrebno podati, saj jo prevajalnik izračuna sam.

Primer:

```
char[] tab4 = { 'a', 'b', 'c' };  //  Elementi so 'a', 'b', in 'c'
```

Omenimo še enkrat, da dolžino tabele *stevila* (število elementov, ki jih lahko shranimo v tabelo) dobimo z izrazom *stevila.Length*.

11.4 INDEKSI V NIZIH IN TABELAH

Izraz, s katerim dostopamo do *i*-tega elementa tabele (*tab[i]),* se v jeziku C# po videzu ujema z izrazom za dostopanje do *i*-tega znaka v nizu (*niz[i]*). Izraza pa se ne ujemata v uporabi. Pri tabelah uporabljamo izraz za pridobitev in spremembo elementa tabele, medtem, ko ga pri nizih uporabljamo le za pridobitev znaka. V primeru, da izraz uporabimo za spremembo določenega znaka v nizu, nam prevajalnik vrne napako.

```
C:\Documents and Settings\Administrator\Desktop\VajeC#\Test12\Test12\
Program.cs(10,13): error CS0200: Property or indexer 'string.this[int]' cannot be
assigned to -- it is read only
```

Primer:

```
// Deklaracija tabele in niza
int[] tab = { 1, 6, 40, 15 };
string niz = "Trubarjevo leto 2008.";

// Pridobivanje
int el = tab[2];  // Vrednost spremenljivke je 6.
char z = niz[2];  // Vrednost spremenljivke je koda zanka 'u'.

// Spreminjanje
tab[1] = 5;  // Spremenjena tabela je [1, 5, 40, 15].
niz[18] = '9';  // Izpiše se napaka.
```

Poglejmo si preprost primer tabele imen. Napolnimo tabelo z imeni in izpišimo njeno dolžino.

```
1: static void Main(string[] args)
2: {
3:     string[] imena={"Jan", "Matej", "Ana", "Grega", "Sanja"};
4:     Console.WriteLine("Dolžina tabele je " + imena.Length + ".");
5:     Console.WriteLine("Ime na sredini tabele je  " +  imena[imena.Length / 2] + ".");
6: }
```

Opis programa

V 3. vrstici smo najavili tabelo *imena* tipa *string[]* in jo napolnili z elementi. Zatem smo izpisali njeno dolžino z ukazom *imena.length*. V 4. vrstici želimo izpisati element, ki se nahaja na sredini naše tabele. Izraz *imena.length / 2* nam vrne *2*, element, ki se nahaja na mestu *imena[2]* je *Ana.*

Podrobneje si poglejmo, kako indeksiramo podatke v tabeli *imena.*

Prvi element tabele ima indeks *0* in vsebuje niz *"Jan"*, drugi ima indeks *1* in vsebuje niz *"Matej"*, tretji ima indeks *2* in vsebuje niz *"Ana"*, četrti ima indeks *3* in vsebuje niz *"Grega"* in peti ima indeks *4* in vsebuje niz *"Sanja"*. Tabelo *imena* si lahko predstavljamo takole:

Tabela 9: Indeksi v tabeli

Indeks	0	1	2	3	4
Vrednost	Jan	Matej	Ana	Grega	Sanja

11.5 NEUJEMANJE TIPOV PRI DEKLARACIJI

Kaj se bo zgodilo, če napišemo naslednji program.

```
1:   static void Main(string[] args)
2:  {
3:      string[] stevila = { 1, 2, 3 };
4:      Console.WriteLine("Dolžina tabele je " +  stevila.Length + ".");
```

```
5:  }
```

Program se ne prevede. Če pogledamo vrstico 3, smo deklarirali tabelo nizov, tej tabeli pa smo priredili celoštevilske vrednosti. Ker se tipa ne ujemata, bo prevajalnik javil:

Error 3 Cannot implicitly convert type 'int' to 'string'

Napako popravimo tako, da napišemo ustrezen (pravilen) tip.

```
1:    static void Main(string[] args)
2:  {
3:        int[] stevila = { 1, 2, 3 };
4:        Console.WriteLine("Dolžina tabele je " +stevila.Length + ".");
5:  }
```

11.6 IZPIS TABELE

Poskusimo izpisati vse elemente tabele *imena* na naslednji način.

```
static void Main(string[] args)
{
    string[] imena={"Jan", "Matej", "Ana", "Grega", "Sanja"};
    Console.WriteLine(imena);
}
```

Program prevedemo in poženemo: dobimo čuden izpis (*System.String[]*). V spremenljivki *imena* namreč ne hranimo tabele imen, ampak le naslov, kje se tabela nahaja. Z ukazom *Console.WriteLine(imena)* izpišemo naslov, kjer se nahaja naša tabela, in ne posameznih elementov. Če želimo izpisati vrednosti slednjih, jih moramo obravnavati vsakega posebej. Omenili smo že, da do *i*–tega elementa (natančneje, do elementa z indeksom i) dostopamo z izrazom *imeTabele[i]*. Popravimo program. Za izpis vseh elementov tabele bomo uporabili zanko *for*.

```
static void Main(string[] args)
{
   string[] imena={"Jaka", "Matej", "Ana", "Grega", "Sanja"};

   for(int i = 0; i < imena.Length; i++)
       Console.WriteLine(imena[i]);
}
```

Opis programa:

Najprej definiramo tabelo tipa *string* in jo napolnimo s podatki. Njeno vsebino bomo izpisali na ekran s pomočjo zanke *for*. Definiramo spremenljivko *i* in ji priredimo vrednost *0*, zatem preverimo če je *i* manjše od dolžine tabele, ki je v našem primeru *5* (pogoj je izpolnjen), izpišemo spremenljivko, ki se nahaja na mestu *imena[0]* to je *Jaka*. Vrednost *i* povečamo za ena in ponovno preverimo resničnost pogoja (*i < imena.Length*), ker je pogoj ponovno resničen izpišemo vrednost spremenljivke *imena[1] Matej*, … . Postopek nadaljujemo toliko

časa, da izpišemo še preostal del tabele, ko pogoj ni več izpolnjen (izpisali smo vse vrednosti tabele) končamo.

Poskusimo izpisati peti element tabele *imena*.

```
static void Main(string[] args)
{
   string[] imena={"Jaka", "Matej", "Ana", "Grega", "Sanja"};
   Console.WriteLine(imena[5]);
}
```

Program prevedemo in poženemo: prevajalnik javi napako, saj smo poskušali poklicati element, ki je izven definirane meje. Kot smo že povedali, se veljavni indeksi vedno gibljejo v meji od *0* do *dolzina – 1*, v našem primeru od *0* do *4*.

11.7 PRIMERJANJE TABEL

Oglejmo si še en primer pogoste napake. Denimo, da želimo primerjati dve tabeli. Zanima nas, če so vsi enakoležni elementi enaki. "Naivna" rešitev z *t1 = = t2* nam ne vrne pričakovan rezultat.

```
static void Main(string[] args)
{
    int[] t1 = new int[] { 1, 2, 3 };
    int[] t2 = new int[] { 1, 2, 3 };
    Console.WriteLine(t1 == t2);
}
```

Opis programa.
Čeprav obe tabeli vsebujeta enake elemente, nam izpis (*false*) pove, da tabeli nista enaki. Zakaj? Spomnimo se, da *t1* in *t2* vsebujeta samo naslov, kjer se nahajata tabeli. Ker sta to dve različni tabeli (sicer z enakimi elementi, a vseeno gre za dve tabeli), zato tudi naslova nista enaka. In to nam pove primerjava *t1 == t2*.

Oglejmo si, kako bi pravilno primerjali dve tabeli.

```
1: static void Main(string[] args)
2: {
3:       int[] t1 = new int[] {1, 2, 3};
4:       int[] t2 = new int[] {1, 2, 3};
5:
6:       bool je_enaka = true;
7:
8:       if (t1.Length == t2.Length)
9:       {
10:          for(int i = 0; i < t1.Length; i++)
11:          {
12:              if (t1[i] != t2[i])
```

```
13:                 je_enaka = false;
14:             }
15:         }
16:         else je_enaka = false;
17:
18:         if (je_enaka)
19:             Console.WriteLine("Tabeli sta enaki.");
20:         else Console.WriteLine("Tabeli nista enaki.");
21: }
```

Opis programa.

V 8. vrstici najprej preverimo če sta tabeli enako veliki. Če je ta pogoj izpolnjen, nadaljujemo s primerjavo elementov znotraj tabele. Z zanko *for* se sprehodimo po vseh elementih in na vsakem koraku primerjamo *t1[0] != t2[0], t1[1] != t2[1], ...* Če bi naleteli vsaj na en par neenakih števil, bi spremenljivko *je_enaka* nastavili na *false,* saj tabeli potem nista enaki. V našem primeru pa je ta pogoj vedno neresničen, saj so *1 != 1, 2 != 2, ...* neresnične izjave, zato izpišemo *Tabeli sta enaki.* Če pa naletimo na tabeli, ki nista enako veliki ali pa je pogoj v 12. vrstici resničen vsaj enkrat, se izpiše *Tabeli nista enaki.*

11.8 ZGLEDI

11.8.1 Dnevi v tednu

Deklariraj tabelo sedmih nizov in jo inicializiraj tako, da bo vsebovala dneve v tednu. Tabelo nato še izpiši, vsak element tabele v svojo vrsto.

```
string[] dneviVTednu = new string[7] {"Ponedeljek", "Torek", "Sreda",  "Četrtek", "Petek",
                                      "Sobota", "Nedelja" };
for (int i=0;i<7;i++)
{
    Console.WriteLine(dneviVTednu[i]);
}
```

11.8.2 Mrzli meseci

Preberi povprečne temperature v vseh mesecih leta in potem izpiši mrzle mesece, torej take, ki imajo temperaturo nižjo od povprečne.

```
static void Main(string[] args)
{
   double[] meseci = new double[12];
   double letnoPovp = 0;

   for (int i = 0; i < 12; i++)
   {
      Console.Write("Povprečna temperatura za " + (i + 1) + ". mesec : ");
      meseci[i] = double.Parse(Console.ReadLine());
      letnoPovp = letnoPovp + meseci[i];
```

```
    }

    letnoPovp = Math.Round(letnoPovp / 12, 2);
    Console.WriteLine("Povprečna letna temperatura : " + letnoPovp);
    Console.WriteLine("Mrzli meseci: ");

    for (int i = 0; i < 12; i++)
    {
        if (meseci[i] <= letnoPovp) Console.Write((i + 1) + ". mesec  ");
    }
}
```

11.8.3 Delitev znakov v stavku

Preberi poljuben stavek in izpiši, koliko znakov vsebuje, koliko je v njem samoglasnikov, koliko števk in koliko ostalih znakov

```
static void Main(string[] args)
{
    string samogl = "AEIOU";

    int stSam = 0, stStevk = 0;
    string stavek;

    Console.Write("Vnesi poljuben stavek: ");
    stavek = Console.ReadLine();
    Console.WriteLine();

    for (int i = 0; i < stavek.Length; i++)
    {
        char znak = stavek[i];
        if (samogl.IndexOf(znak) > -1)  //  znak je samoglasnik
            stSam = stSam + 1;
        if ('0' <= znak && znak <= '9')
            stStevk = stStevk + 1;
    }
    Console.WriteLine("\nŠtevilo vseh znakov v stavku : " + stavek.Length);
    Console.WriteLine("Število samoglasnikov       : " + stSam);
    Console.WriteLine("Število cifer               : " + stStevk);
    Console.WriteLine("Število ostalih znakov      : " + (stavek.Length - stSam - stStevk));
}
```

11.8.4 Zbiramo sličice

Začeli smo zbirati sličice. V album moramo nalepiti 250 sličic. Zanima nas, koliko sličic bomo morali kupiti, da bomo napolnili album. Seveda ob nakupu ne vemo, katero sličico bomo dobili. Ker bi to radi vedeli še preden se bomo zares spustili po nakupih, bi radi

polnjenje albuma simulirali s pomočjo računalniškega programa. In če bomo to simulacijo ponovili dovolj mnogokrat, bomo že dobili občutek o številu potrebnih nakupov. Pri tem seveda naredimo nekaj predpostavk:

- hkrati vedno kupimo le eno sličico.
- Vse sličice so enako pogoste. Torej je verjetnost, da bomo kupili i-to sličico ravno 1/250.
- Podvojenih sličic ne menjamo.

Poglejmo, kako bi sestavili program.
Naš album bo tabela. Če bo vrednost ustreznega elementa *false*, to pomeni da sličice še nimamo. Da nam ne bo po vsakem nakupu treba prečesati vsega albuma, da bi ugotovili, ali kakšna sličica še manjka, bomo vodili evidenco o tem, koliko sličic v albumu še manjka. V zanki, ki se bo odvijala toliko časa, dokler število manjkajočih sličic ne bo padlo na nič, bomo kupili sličico (naključno generirali število med 0 in 249). Če je še nimamo, bomo zmanjšali število manjkajočih sličic in si v albumu označili, da sličico imamo.

```
static void Main(string[] args)
{
   int st_slicicvalbumu = 250;
   int st_zapolnjenih = 0;
   int st_kupljenihslic = 0;
   Random bob = new Random();       //  boben naključnih stevil
   bool[] album = new bool[st_slicicvalbumu];       //  album
    //  polnimo album
   while (st_zapolnjenih < st_slicicvalbumu)
   {
      st_kupljenihslic = st_kupljenihslic + 1;
      int st_slikic= bob.Next(st_slicicvalbumu);
       //  jo damo v album
      if (!album[st_slikic])
      {
         album[st_slikic] = true;
         st_zapolnjenih = st_zapolnjenih + 1;
      }
   }
   Console.WriteLine("Da smo napolnili album, smo kupili "+ st_kupljenihslic + " slikic.");
}
```

Opis programa:
V uvodu definiramo tri spremenljivke s katerimi nadzorujemo stanje album. V 9. vrstici začnemo polniti album, za to uporabili zanko *while*. Ob vsakem vstopu v zanko kupimo sličico, ki ji dodelimo naključno mesto v albumu (naključno generirali število med 0 in 249). Preverimo če že imamo to mesto zapolnjeno (*!album[st_slikic]*). Če ne, jo vstavimo v album in povečamo število zapolnjenih mest v albumu. To ponavljamo toliko časa, da je pogoj v zanki resničen. Na koncu še izpišemo, koliko sličic smo morali kupiti, da smo napolnili album.

11.8.5 Najdaljša beseda v nizu

Napišimo program, ki bo našel in izpisal najdaljšo besedo prebranega niza. Predpostavimo, da so besede ločene z enim presledkom.

Da bomo iz niza izluščili besede, si bomo pomagali z metodo *Split*, ki jo najdemo v razredu *string*. Metoda vrne tabelo nizov, v kateri vsak element predstavlja podniz niza, nad katerim smo metodo uporabili. V jeziku C# niz razmejimo na besede z razmejitvenim znakom oziroma znaki, ki jih ločimo z vejico (npr. *Split('/')*). Posamezen razmejitveni znak pišemo v enojnih narekovajih (').

Primer:

Denimo, da imamo

```
string stavek = "a/b/c.";
```

Če uporabimo

```
string[] tab = stavek.Split('/');
```

smo s tem ustvarili novo tabelo velikosti 3. V tabeli so shranjeni podnizi niza stavek, kot jih loči razmejitveni znak /. Prvi element tabele tab je niz "a", drugi element je niz "b" in tretji element niz "c.".

```
public static void Main(string[] args)
{
    // Vnos niza
   Console.Write("Vnesi niz: ");
   string niz = Console.ReadLine();

    // Pretvorba niza v tabelo nizov
   string[] tab = niz.Split(' ');

    // Iskanje najdaljše besede
   string pomozni = "";  // Najdaljša beseda (oz. podniz)
   for (int i = 0; i < tab.Length; i++)
   {
      if (pomozni.Length < tab[i].Length)
      {
         pomozni = tab[i];
      }
   }

    // Izpis najdaljše besede
   Console.WriteLine("Najdaljša beseda: " + pomozni);
}
```

Razlaga
Najprej določimo niz *niz*. Nato s pomočjo klica metode *Split()* določimo tabelo tab. V metodi za razmejitveni znak uporabimo presledek (' '). Če je med besedami niza niz več presledkov, se vsi presledki obravnavajo kot razmejitveni znaki. Nato določimo pomožni niz *pomozni*, ki predstavlja trenutno najdaljšo besedo. Z zanko se sprehodimo po tabeli tab in poiščemo najdaljšo besedo. Na koncu izpišemo najdaljšo besedo niza *niz* oziroma tabele *tab*.
Če za spremenljivko niz določimo prazen niz (v konzoli pri vnosu niza stisnemo le tipko *Enter*), se program izvede in za najdaljši niz izpiše prazen niz. Tabela *tab* je ostala prazna, zato je niz v spremenljivki *pomozni* ostal "".

11.8.6 Uredi elemente po velikosti

Sestavimo program, ki bo uredil vrednosti v tabeli celih števil po naslednjem postopku: poiščimo najmanjši element in ga zamenjajmo s prvim. Nato poiščimo najmanjši element od drugega dalje in ga zamenjajmo z drugim, itn. Tabelo velikosti n preberemo in jo po urejanju izpišemo na ekran.

Sestavimo program po korakih:
Najprej deklariramo spremenljivko *n*, katere prebrano vrednost uporabimo za velikost tabele.

```
Console.Write("Velikost tabele: ");
int n = int.Parse(Console.ReadLine());
```

Nato deklariramo tabelo velikosti *n*.

```
int[] tab = new int[n];
```

Z zanko, ki se bo izvedla *n*-krat, napolnimo tabelo s števili, dobljenimi pri branju iz konzole.

```
Console.WriteLine();
for (int i = 0; i < n; i++)
{
    Console.Write("Vnesi " + (i + 1) + ". element: ");
    tab[i] = int.Parse(Console.ReadLine());
}
```

Nato naredimo zanko, ki se izvede (*n*-1)-krat. Ko nam bo ostal le zadnji element, bo že urejen. V zanki:

- Deklariramo pomožni spremenljivki. V prvi spremenljivki hranimo kandidata za najmanjši element med tistimi z indeksi od indeksa določenega z zanko do konca tabele. V drugi spremenljivki pa hranimo indeks kandidata, shranjenega v prvi spremenljivki.
- Naredimo zanko *for*, ki se sprehodi po tabeli od elementa, katerega indeks je določen s prvo zanko, do zadnjega elementa tabele.
- Če najdemo manjši element, si zapomnimo njegovo vrednost in indeks.
- Izvedemo zamenjavo med trenutnim elementom in najdenim najmanjšim elementom.

```
for (int i = 0; i < n - 1; i++)
{
   int min = tab[i];  //  Kandidat za najmanjši element od indeksa i
    //  do konca tabele
   int indeks = i;  //  Indeks najmanjšega kandidata
   for (int k = i + 1; k < n; k++)
   {
      if (min > tab[k])
      {  //  Poiščimo najmanjši element v tem delu
         min = tab[k];
         indeks = k;
      }
   }
    //  Zamenjava elementov
   int t = tab[i];
   tab[i] = tab[indeks];
   tab[indeks] = t;
}
```

Na koncu z zanko *for*, ki se sprehodi po urejeni tabeli tab, izpišemo elemente in jih pri izpisu ločimo s presledkom.

```
Console.WriteLine();
for (int i = 0; i < n; i++)
{
    Console.Write(tab[i] + " ");
}
 //  Prehod v novo vrstico
Console.WriteLine();
```

Poglejmo še zapis celotnega programa.

```
public static void Main(string[] args)
{
    //  Vnos vrelikosti tabele
   Console.Write("Velikost tabele: ");
   int n = int.Parse(Console.ReadLine());

    //  Deklaracija tabele
   int[] tab = new int[n];

    //  Napolnitev tabele
   Console.WriteLine();
   for (int i = 0; i < n; i++)
   {
      Console.Write("Vnesi " + (i + 1) + ". element: ");
      tab[i] = int.Parse(Console.ReadLine());
   }
```

```
   // Uredimo tabelo
   for (int i = 0; i < n - 1; i++)
   {
      int min = tab[i];  // Kandidat za najmanjši element od indeksa i  do konca tabele
      int indeks = i;  // Indeks najmanjšega kandidata
      for (int k = i + 1; k < n; k++)
      { // Poiščimo najmanjši element v tem delu
         if (min > tab[k])
         {
            min = tab[k];
            indeks = k;
         }
      }
       // Zamenjava elementov
      int t = tab[i];
      tab[i] = tab[indeks];
      tab[indeks] = t;
   }

   // Izpis urejene tabele
   Console.WriteLine();
   for (int i = 0; i < n; i++)
   {
      Console.Write(tab[i] + " ");
   }
   // Prehod v novo vrstico
   Console.WriteLine();
}
```

11.8.7 Manjkajoča števila

Generirajmo tabelo 50 naključnih števil med 0 in 100 (obe meji štejemo zraven). S pomočjo zanke *for* ugotovimo in izpišimo, katerih števil med 0 in 100 ni v tej tabeli. Za kontrolo naredimo izpis elementov tabele.

Sestavimo program po korakih:
Najprej deklariramo dve konstanti. Prvo uporabimo za velikost tabele, drugo pa za zgornjo mejo naključnih števil.

```
const int vel = 50;  // Velikost tabele
const int mejaStevil = 100;  // Zgornja meja naključnih števil
```

Nato deklariramo tabelo, ki ima velikost *vel*.

```
int[] tab = new int[vel];
```

Ustvarimo še generator naključnih števil.

```
Random nak = new Random();  //  Generator naključnih števil
```

Z zanko, ki se izvede *vel*-krat, napolnimo tabelo z naključnimi števili med 0 in 100 (obe meji štejemo zraven):

```
for (int i = 0; i < vel; i++)
{
   tab[i] = nak.Next(mejaStevil + 1);  //  Določitev naključnega števila
}
```

Nato z zanko *for*, ki se sprehodi po elementih tabele *tab*, izpišemo elemente tabele. Elemente tabele ločimo s presledkom.

```
Console.WriteLine("Izpis vsebine tabele naključnih števil: ");
 //  Izpis elementov tabele
for (int i = 0; i < tab.Length; i++)
{
   int el = tab[i];
   Console.Write(el + " ");  //  Izpis elementa, ločenega s presledkom
}
```

Nato še enkrat naredimo zanko, ki se izvede 101-krat. V njej

- Ustvarimo logično spremenljivko za iskanje števila. Začetna vrednost spremenljivke je *false*.
- Naredimo zanko *for*, ki se sprehodi po elementih tabele tab. V zanki
  - če število najdemo
    - vrednost logične spremenljivke nastavimo na *true* in
    - s stavkom *break* prekinemo iskanje.
  - če števila ne najdemo, ga izpišemo. Izpisana števila ločimo s presledkom.

Na koncu gremo še v novo vrstico. Poglejmo sedaj še zapis celotnega programa.

```
public static void Main(string[] args)
{
   const int vel = 50;  //  Velikost tabele
   const int mejaStevil = 100;  //  Zgornja meja naključnih števil
   int[] tab = new int[vel];  //  Dekleracija tabele
   Random nak = new Random();  //  Generator naključnih števil
    //  Polnjenje tabele
   for (int i = 0; i < vel; i++)
   {
      tab[i] = nak.Next(mejaStevil + 1);  //  Določitev naključnega števila
   }
   Console.WriteLine("Izpis vsebine tabele naključnih stevil: ");
    //  Izpis elementov tabele
   for (int i = 0; i < tab.Length; i++)
   {
```

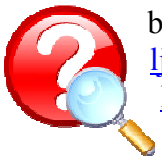
```
            int el = tab[i];
            Console.Write(el + " ");  //  Izpis elementa, ločenega s presledkom
        }
        Console.WriteLine("\n\nŠtevila med 0 in " + mejaStevil +", ki niso v tej tabeli, so: ");
         //  Iskanje števil, ki se ne nahajajo v tabeli
        for (int i = 0; i < mejaStevil + 1; i++)
        {
            bool nasli = false;  //  Spremenljivka za iskanje števil
            for (int j = 0; j < tab.Length; j++)
            {
                int el = tab[j];
                if (el == i)
                {
                    nasli = true;  //  Število smo našli
                    break;  //  Prekinemo iskanje števila
                }
            }
             //  Ali smo število našli, nam pove spremenljivka nasli
            if (!nasli) Console.Write(i + " ");
        }
         //  Nova vrstica
        Console.WriteLine();
    }
```

Če pa si lahko "privoščimo" uporabo dodatne tabele take velikosti, kot je vseh možnih števil (v našem primeru 101), lahko razvijemo tudi boljši in (hitrejši) algoritem.

11.9 POVZETEK

Bistvena prednost uporabe tabel je, da lahko z njihovo pomočjo na enostaven način obdelujemo množico spremenljivk istega tipa, ki so dostopne preko indeksov te tabele. Velikost tabele lahko določimo med samim delovanjem programa, vse tabelarične spremenljivke pa so si enakovredne ne glede na to, ali se nahajajo na začetku ali pa na koncu tabele. V primeru, da pa je število podatkov preveliko in da jih želimo po zaključku programa tudi shraniti za kasnejše obdelave, bomo podatke raje zapisali v datoteke.

Brez samostojnega pisanja programov, ki za rešitev problemov uporabljajo tabele, seveda ne bo šlo. Ustrezne probleme poiščite na primer na http://up.fmf.uni-lj.si/index.html#h2_22, na http://penelope.fmf.uni-lj.si/diri0607/index.php/Kategorija:Tabele, v zbirki Lokar M. in Uranič S., Programiranje 1 - zbirka rešenih nalog v C# in še kje.



Pogosto se pri uporabi tabel uporablja še oblika zanke, ki je nismo spoznali, zanka *foreach*. Ustrezno obravnavo te zanke si lahko ogledate na primer v Petric D., Spoznavanje osnov programskega jezika C# ali pa v Murach J. in Murach M., Murach's C# 2008.

12 VAROVALNI BLOKI IN IZJEME

12.1 UVOD

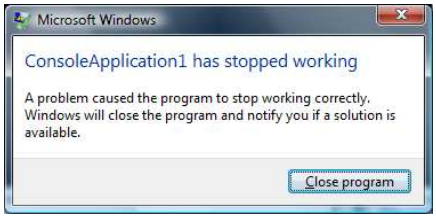
Tako kot vsakdanjem življenju se tudi pri delovanju programov pojavljajo napake. Zelo neprijetno je, če se program sredi delovanja "sesuje" (neha delovati), ker na primer programer ni predvidel, da lahko v določenih izjemnih primerih pride do "čudne" situacije. To se zgodi na primer pri uporabi negativnega indeksa, ali pa v primeru da datoteka, kamor naj bi se zapisali rezultati, ne obstaja, pa tudi ko uporabnik po pomoti vnese število 0, s katerim naj se izvede operacija deljenja in podobno. Dobro je, če imamo možnost, da v primeru napak program ustrezno reagira, pa če drugega ne, izpiše prijazno obvestilo, ne pa da bo program zaradi napake prenehal z delovanjem.

C#, podobno kot drugi sodobni programski jeziki, pozna tako imenovan mehanizem izjem (*exceptions*). Če pri delovanju programa pride do napake, se sproži tako imenovana izjema. To izjemo lahko programsko prestrežemo in napišemo ustrezno kodo, kako naj program nadaljuje v tem primeru.

Poglejmo si primer, Napisali smo program, ki prebere niz in uporabniku omogoči, da vnese indeks znaka, ki ga zanima.

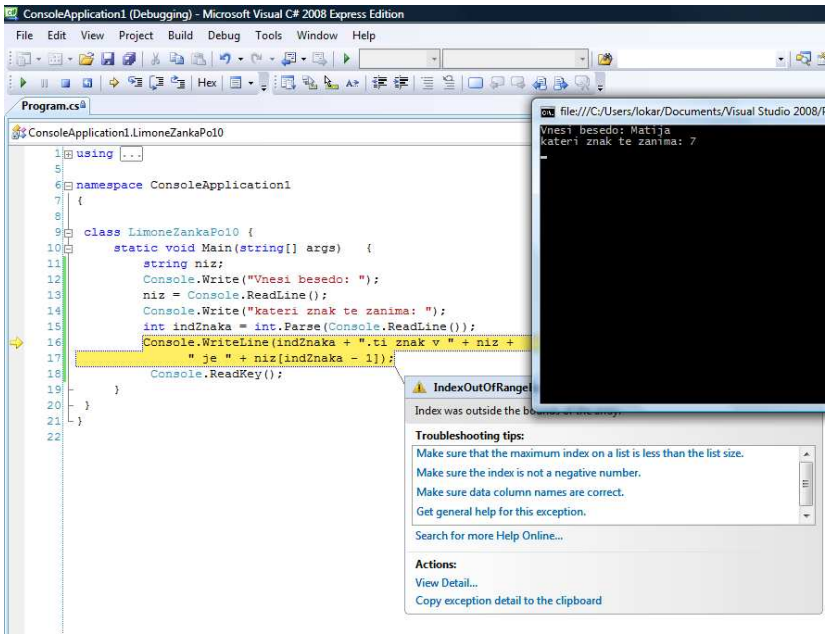
```
static void Main(string[] args)
{
    string niz;
    Console.Write("Vnesi besedo: ");
    niz = Console.ReadLine();
    Console.Write("kateri znak te zanima: ");
    int indZnaka = int.Parse(Console.ReadLine());
    Console.WriteLine(indZnaka + ".ti znak v " + niz +" je " + niz[indZnaka - 1]);
}
```

Če program poženemo in vpišemo neapačen (na primer prevelik) indeks, program neha delovati. Če smo poganjali program neposredno (torej EXE), dobimo le obvestilo operacijskega sistema. V primeru OS Vista je to npr. takole



Slika 21: Napaka zaradi prekoračitve indeksa v OS Vista

Če pa program poženemo znotraj razvojnega okolja Visual C# 2008 Express Edition, nas okolje postavi v "problematično" vrstico in javi, da je prišlo do izjeme *IndexOutOfBounds*.



Slika 22: Pri delovanju programa je prišlo do izjeme *IndexOutOfBounds*

Sedaj bi lahko kodo spremenili tako, da bi preverili ustreznost podatkov in ugotovili ali lahko pride do napake.

```
...
int indZnaka = int.Parse(Console.ReadLine());
if ((0 <= indZnaka) && (indZnaka <= niz.Length()))
{
   Console.WriteLine(indZnaka + ".ti znak v " + niz +" je " + niz[indZnaka - 1]);
}
else
{
  Console.WriteLine(niz + " nima " + indZnaka  + "tega znaka");
}
```

Vendar se na ta način tok programa in koda za obdelavo napak prepletata. Poleg tega lahko včasih pride tudi do napak, ki jih je težko uspešno preprečiti s preverjanjem določenih pogojev ali pa so ti zelo zapleteni. V našem primeru denimo, lahko do napake pride tudi, če uporabnik kot indeks znaka vnese nekaj, kar ni celo število.

Zato je ideja izjem v tem, da dele programa, ki so "kritični" obdamo z varovalnim blokom. Če v tem varovalnem bloku pride do napake, se izvede lovilni del, kjer lahko ob napakah ustrezno reagiramo.

Idejna rešitev za obdelavo izjem je torej v tem, da ločimo kodo, ki predstavlja tok programa in kodo za obdelavo napak. Na ta način postaneta obe kodi lažji za razumevanje, saj se ne prepletata.

12.2 DELITEV VAROVALNIH BLOKOV V C#

Za obdelavo izjem pozna **C#** naslednje bloke:

- blok za obravnavo prekinitev **try…catch …**,
- večkratni varovalni blok **try … catch … catch …**
- brezpogojni varovalni blok **try … finally …** .

Mi si bomo ogledali le prvega.

12.3 BLOK ZA OBRAVNAVO PREKINITEV: TRY … CATCH …

Kodo, ki bi jo sicer napisali v delu programa ali pa npr. v neki metodi, zapišemo v varovalnem bloku **try** (blok je vsak del kode napisane med oklepajema { in }). Drugi del začenja besedica **catch** in v bloku zapišemo enega ali več stavkov za obdelavo izjem oz. napak (t.i. ***catch* handlers**). Če katerikoli stavek znotraj bloka **try** povzroči izjemo (če pride do napake), se normalni tok izvajanja programa prekine (normalni tok izvajanja pomeni, da se posamezni stavki izvajajo od leve proti desni, stavki pa se izvajajo eden za drugim, od vrha do dna), program pa se nadaljuje v bloku **catch**, v katerem pa moramo seveda napako ustrezno obdelati.

Primer:

```
while (napaka)
{
   try
   {
      int levo = int.Parse(Console.ReadLine());
      int desno = int.Parse(Console.ReadLine());      } običajna programska koda
      double rezultat = levo / desno;
      Console.WriteLine(rezultat);
      napaka = false;
   }
   catch
   {
      Console.WriteLine("Ponovno vnesi podatke");     } koda za obdelavo napake
      napaka = true;
   }
}
```

Metoda *int.Parse* skuša napraviti ustrezni pretvorbi iz niza v celo število, pri čemer pa seveda lahko pride do napake (npr. če uporabnik vnese znak, ki je različen od znakov '0' do '9'). Do napake lahko pride tudi v stavku, kjer je operacija za deljenje, saj se lahko zgodi, da je drugi operand enak 0. Varovalni blok take napake prestreže in izvede se stavek (oz. stavki) v bloku **catch**.

Če v stavkih znotraj bloka **try** pride do napake (izjeme), se program na tem mestu ne "sesuje". Ostali stavki znotraj bloka **try** se ne izvedejo. Začnejo pa se izvajati stavki za obdelavo izjem (napak), ki so znotraj bloka **catch**. Če pa do napake v bloku **try** ne pride, se stavki v bloku **catch** NE izvedejo nikoli!

Varovalni blok je torej zgrajen takole:

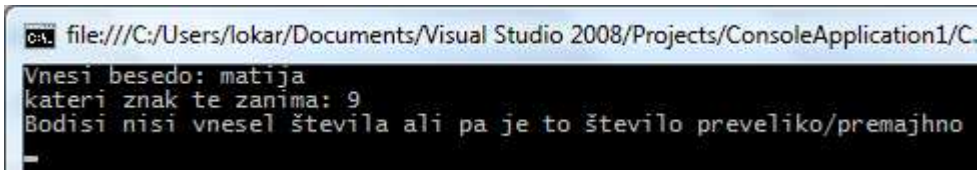
```
try
{
    // stavki, kjer lahko pride do napake
}
catch
{
   // obdelava napake . . .
}
```

Jezik C# ponuja tudi možnost, da tudi ugotovimo, do kakšne izjeme je prišlo in potem reagiramo glede na vrsto izjeme. Izjeme lahko prožimo tudi sami programsko. Vendar se v začetnem spoznavanju jezika C# z vsem tem ne bomo ubadali.

Zaključimo s tem, da zgled, ki smo ga navedli na začetku, damo v varovalni blok.

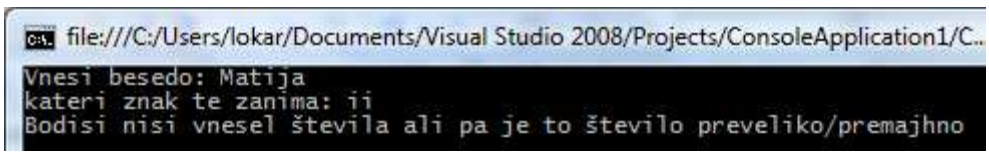
```
static void Main(string[] args)
{
    string niz;
    Console.Write("Vnesi besedo: ");
    niz = Console.ReadLine();
    try
    {
       Console.Write("kateri znak te zanima: ");
       int indZnaka = int.Parse(Console.ReadLine());
       Console.WriteLine(indZnaka + ".ti znak v " + niz + " je " + niz[indZnaka - 1]);
    }
    catch
     Console.WriteLine("Bodisi nisi vnesel števila ali pa je to število preveliko/premajhno");
}
```

Sedaj se naš program uspešno zaključi ne glede na to, ali vnesemo napačen indeks,



Slika 23: Napaka (izjema) zaradi napačnega indeksa

ali pa za indeks ne sploh vnesemo števila.



Slika 24: Do napake (izjeme) je prišlo zato, ker smo vnesli podatek, ki ni število.

12.4 POVZETEK

Mehanizem varovalnih blokov in izjem nam omogoča pisanje programov, ki so veliko bolj »odporni na napake«. V tem razdelku smo si ogledali zgolj najnujnejše. Več o tem si preberite na primer v Uranič S., Microsoft C#.NET, v Joe Mayo, C# Station Tutorial ali pa v C# Practical Learning 3.0.

13 METODE

13.1 UVOD

Vsi naši dosedanji programi so bili večinoma kratki in enostavni, ter zaradi tega pregledni. Pri zahtevnejših projektih pa postanejo programi daljši, saj zajemajo več kot en izračun, več pogojev, več zank. Če so napisani v enem samem kosu postanejo nepregledni. Poleg tega se zaporedja stavkov pri daljših programih lahko tudi večkrat ponovijo. Vzdrževanje in popravljanje takih programov je težavno, časovno zahtevno, verjetnost za napake pa precejšnja.

Vse to so razlogi za pisanje metod. S pomočjo metod programe smiselno razbijemo na pod naloge, vsako pod nalogo pa izdelamo posebej. Te manjše pod naloge imenujemo podprogrami, rešimo jih z lastnimi metodami, zaženemo pa jih v glavnem programu. Program torej razdelimo v več manjših "neodvisnih" postopkov in nato obravnavamo vsak postopek posebej. Metodam v drugih programskih jezikih rečemo tudi funkcije, procedure ali podprogrami.

Na ta način bo naš progam krajši, bolj pregleden, izognili se bomo večkratnemu ponavljanju istih stavkov, pa še popravljanje in dopolnjevanje bo enostavnejše. Bistvo metode je tudi v tem, da ko je enkrat napisana, moramo vedeti le še kako jo lahko uporabimo, pri tem pa nas večinoma ne zanima več, kako je pravzaprav sestavljena. Napisano metodo lahko uporabimo večkrat, seveda pa jo lahko uporabimo tudi v drugih programih.

Metode smo v bistvu že ves čas uporabljali, a se tega mogoče nismo zavedali. Za izpisovanje na zaslon smo uporabljali metodo *Console.WriteLine()*, pri pretvarjanju niza v število smo uporabljali metodo *int.Parse()*, pri računanju kvadratnega korena metodo *Math.Sqrt()* in številne druge. Vse te metode so torej že napisane, vse kar moramo vedeti o njih pa je, kakšen je njihov namen in kako jih uporabiti.

Seveda pa lahko metode pišemo tudi sami. Pravzaprav smo vsaj eno od metod že ves čas pisali – to je bila metoda *Main()*. Ogrodje zanjo nam je zgeneriralo že razvojno okolje, mi pa smo doslej pisali le njeno vsebino oz. kot temu rečemo strokovno – **glavo** metode nam je zgeneriralo razvojno okolje, **telo** metode pa smo napisali sami. Metoda *Main()* predstavlja izhodišče našega programa (t.i. **glavni program**) in ima zaradi tega že vnaprej točno predpisano ime, obliko in namen. Metode, ki se jih bomo naučili pisati, bodo imele prav tako specifičen namen, obliko in ime, vse te lastnosti pa bomo določili mi sami pred in med njihovim pisanjem. Držati pa se moramo pravila, da morajo biti metode, ki jih napišemo, pregledne, če se la da uporabne tudi v drugih programih, biti morajo samozadostne, prav tako jih tudi ne napišemo izrecno za izpisovanje ali branje podatkov, razen če to ni osnovni namen te metode.

13.2 DELITEV METOD

V jeziku C# metode v splošnem delimo na **statične** in **objektne**. V tem poglavju bomo obravnavali le statične metode. Zaenkrat povejmo le, da statične metode kličemo nad razredom, medtem, ko objektne nad objektom. Metode ločimo tudi po načinu dostopa na **javne** (*public*), **privatne** (*private*) in **zaščitene** (*protected*). Ker bomo uporabljali le javne metode, se v način dostopa ne bomo spuščali.

13.3 DEFINICIJA METODE

Metodo v program vpeljemo z definicijo. Definicija metode je sestavljena iz **glave** oziroma **deklaracije** metode in **telesa** metode, napišemo pa jo v celoti pred ali pa **za** glavno metodo *Main* našega programa.
V deklaracijo zapišemo najprej **dostop** do metode (torej ali je metoda javna (*public*), privatna (*private*) ali zaščitena (*protected*)), nato **vrsto** metode, s katero povemo ali je metoda statična (*static*) ali objektna. V tem razdelku bodo vse naše metode vrste *public static*. Sledi ji **tip** rezultata metode. Tip rezultata metode je poljuben podatkovni tip (npr. *int*, *double[]*, *string*) in pomeni tip vrednosti, ki ga metoda vrača. Nato z **imenom metode** povemo kako se imenuje metoda. Velja dogovor, da se imena metod začnejo z veliko črko. Metodam damo smiselna imena, ki povedo, kaj metoda dela. Ime metode je poljubno, ne smemo pa za ime metode uporabiti rezerviranih besed (ime metode **ne sme** biti npr. *string*, *do*, *return*, …). V imenih metod **ne sme** biti presledkov in nekaterih posebnih znakov (npr. znakov '/', ')', '.', …), prav tako pa v imenih metod niso zaželeni šumniki. Za imenom metode zapišemo okrogle oklepaje, ki so obvezni del deklaracije. Znotraj oklepajev zapišemo **argumente metode**, ki jih metoda sprejme (ime argumentov) in kakšnega tipa so. Več argumentov med sabo ločimo z vejico. Če metoda ne sprejme nobenih argumentov, napišemo le prazne okrogle oklepaje *()*. Sledi **telo metode**, to je del, ki je znotraj bloka *{}*. V telo metode zapišemo stavke, ki izvršijo nalogo metode.

Posebne metode so tiste, ki ne vračajo nobenega rezultata, ampak le opravijo določene delo (na primer nekaj izpišejo, narišejo sliko, pošljejo datoteko na tiskalnik, ...). Pri takih metodah kot tip rezultata navedemo tip ***void***. Metode tipa *void* v svojem telesu nimajo stavka *return*.

Javna metoda – metoda je dostopna vsem delom našega programa (do nadaljnjega bodo vse metode javne)

Statična metoda (do nadaljnjega bodo vse metode statične – kaj pravzaprav to pomeni bo jasno šele pri poglavju o razredih in objektih)

tip metode: določa tip rezultata, ki ga metoda vrača.

```
public static   tip_metode   Ime_metode  (parametri metode)   // Glava metode
{                                                              
     ... stavki                                                
     return ... // vračanje vrednosti.                          Telo metode
       // Stavka return ni, če je metoda tipa void             
}                                                              
```

Poglejmo si nekaj primerov deklaracij metod.

Primeri:

```
public static int Beri();
```

Metoda *Beri* je javna (*public*), statična (*static*), vrača rezultat tipa *int* in ne sprejme nobenega argumenta (znotraj oklepajev ni argumentov).

```
public static void Nekaj(string[] tabNiz);
```

Metoda *Nekaj* je javna (*public*), statična (*static*), ne vrača rezultata (tip *void*) in sprejme en argument, ki se imenuje *tabNiz* in je tipa *string[]* (tabela nizov).

```
public static int Max(int a, int b);
```

Metoda *Max* je javna (*public*), statična (*static*), vrača rezultat tipa *int* in sprejme dva argumenta, ki sta celi števili (tip *int*), prvi argument se imenuje *a*, drugi *b*.

```
public double Abs() ;
```

Metoda *Abs* je javna (*public*), objektna (ni določila *static*), vrača rezultat tipa *double* in ne sprejme nobenega argumenta (znotraj oklepajev ni argumentov).

Parametri metod niso obvezni, saj lahko napišemo metodo, ki ne sprejme nobenega parametra, a take metode v splošnem niso preveč uporabne.

13.3.1 Pozdrav uporabniku

Napišimo metodo, ki prebere naše ime in nas pozdravi (npr. za ime "Janez" izpiše *Pozdravljen Janez*).

```
// ker metoda ne bo vrnila ničesar, je tipa void
public static void PozdravUporabniku()
{
    // preberemo podatek
   Console.Write("Vnesi ime: ");
   string ime = Console.ReadLine();
    // in ga izpišemo
   Console.WriteLine("Pozdravljen " + ime + "!");
}
```

Metodo smo napisali, potrebno pa jo še znati uporabiti, oz. jo poklicati. Metode tipa *void* (ki ne vračajo ničesar) uporabimo v **samostojnem stavku** tako, da napišemo njihovo ime, v oklepajih pa še parametre, če seveda metoda parametre potrebuje, sicer pa napišemo le oklepaj in zaklepaj. Zgornjo metodo bi torej poklicali (npr. kjerkoli v glavnem programu *Main*) takole:

```
PozdravUporabniku();
```

13.4 VREDNOST METODE

Če je tip metode različen od *void*, moramo vrednost, ki jo vrne metoda, določiti s stavkom

```
return izraz;
```

Vrednost metode je vrednost izraza, ki je naveden za ključno besedo *return*. V opisu postopka je stavek *return* lahko na več mestih. Kakor hitro se eden izvede, se izvajanje metode konča z vrednostjo, kot jo določa izraz, naveden pri stavku *return*. Če metoda kaj izpisuje ali riše na ekran, to ni rezultat te metode, ampak samo stranski učinek, ki ga ima metoda. Take metode običajno ne vračajo nobenih rezultatov. V tem primeru je rezultat metode tipa *void*.

Primer:

```
public static double Povprecje(double x, double y)
{
   return (x + y) / 2.0;
}
```

Metoda *Povprecje* je javna, statična, vrne rezultat tipa *double* in sprejme dva argumenta, ki sta tipa *double*, prvi se imenuje *x* in drugi *y*.

13.4.1 Vsota lihih števil

Napišimo metodo, ki ne sprejme nobenih parametrov, izračuna in vrne pa vsoto vseh dvomestnih števil, ki so deljiva s 5!

```
public static int VsotaDvomestnih()  // glava funkcije
{
    int vsota = 0;  // začetna vsota je 0
    for (int i = 1; i <= 100; i++)
    {
       if (i % 5 == 0)  // če ostanek pri deljenju deljiv s 5
       vsota = vsota + i;  // potem število prištejemo k vsoti
    }
    // metoda vrne rezultat: vsoto vseh dvomestnih števil, deljivih s 5
   return vsota;
}
```

Še klic metode v glavnem programu: ker metoda *VsotaDvomestnih()* vrača rezultat (tip *int*), jo ne moremo klicati samostojno, ampak v nekem **prireditvenem stavku**, lahko pa tudi kot parameter v drugi metodi.

```
int vsota100 = VsotaDvomestnih(); // klic metode v prireditvenem stavku
```

```
Console.WriteLine(VsotaDvomestnih()); // klic metode kot parametra metode WriteLine
```

Zgornja metoda je sicer povsem legalna, ker pa nima vhodnih parametrov, je uporabna le za točno določen, ozek namen. Če želimo narediti metodo res uporabno, uporabljamo parametre.

13.5 ARGUMENTI METOD − FORMALNI IN DEJANSKI PARAMETRI

V glavi metode lahko uporabimo tudi **argumente**. Pravimo jim tudi parametri metode. Parametre, ki jih napišemo, ko metodo pišemo, imenujemo **formalni** parametri. Ko pa neko metodo pokličemo, formalne parametre nadomestimo z **dejanskimi** parametri.

13.5.1 Iskanje večje vrednosti dveh števil

Napišimo metodo, ki dobi za parametra poljubni celi števili, vrne pa večje izmed teh dveh števil.
Ker bo metoda imela za parametra dve celi števili (tip *int*), je večje izmed teh dveh prav gotovo tudi tipa *int*. Metoda bo torej vrnila celo število, zaradi česar bomo za tip metode uporabili tip *int*.

```
public static int Max(int a, int b)  // metoda ima dva parametra, a in b
{
   if (a > b)   // če a večji od b metoda vrne število a
       return a;
   return b;    // sicer pa metoda vrne število b
}
```

Parametra *a* in *b*, ki smo ju napisali v glavi metode, sta **formalna parametra**. Ko bomo metodo poklicali, pa bomo zapisali **dejanska parametra**. Metodo lahko uporabimo tako, da v glavnem programu najprej preberemo (določimo, zgeneriramo) dve celi števili, nato pa ti dve števili uporabimo za parametra metode *Max*.

```
Console.Write("Prvo število: ");
int prvo = int.Parse(Console.ReadLine());
Console.Write("Drugo število: ");
int drugo = int.Parse(Console.ReadLine());

int vecje = Max(prvo, drugo);  // parametra prvo in drugo sta dejanska parametra
Console.WriteLine("Večje od vnesenih dveh števil je : " + vecje);
```

Metodo *Max* bi seveda lahko poklicali tudi takole:

```
Console.WriteLine("Večje od vnesenih dveh števil je : " + Max(prvo,drugo));
```

Pri klicu metode *Max* je parameter *a* dobil vrednost spremenljivke *prvo*, parameter *b* pa vrednost spremenljivke *drugo*. Formalna parametra *a* in *b*, smo torej pri klicu metode nadomestili z dejanskima parametroma *prvo* in *drugo*. Namesto spremenljivk bi seveda pri klicu metode lahko uporabili tudi poljubni dve števili, ali pa številska izraza, npr.:

```
Console.WriteLine("Večje izmed števil 6 in 11 je : " + Max(6,11);

Console.WriteLine("Večje od vnesenih izrazov je : " + Max(5+2*3,41-3*2));
```

Metodo *Max* pa lahko pa pokličemo tudi takole:

```
Random naklj = new Random();
int prvo = naklj.Next(100);
int drugo = naklj.Next(100);
int tretje = naklj.Next(100);
Console.WriteLine("Največje izmed treh števil je: " + Max(prvo, Max(drugo, tretje)));
```

13.6 ZGLEDI

13.6.1 Število znakov v stavku

Napišimo metodo, ki dobi dva parametra: poljuben stavek in poljuben znak. Metoda naj ugotovi in vrne kolikokrat se v stavku pojavi izbrani znak.

```
// Metoda
static int Kolikokrat(string stavek, char znak)
{
   int skupaj = 0;  // Začetno število znakov je 0
   for( int i=0; i<stavek.Length; i++ )
   {
     if (stavek[i] == znak)  // tekoči znak primerjamo z našim znakom
        skupaj++;
   }
   return skupaj;  // Metoda vrne skupno število najdenih znakov
}

static void Main(string[] args)
{
   Console.Write("Vnesi poljuben stavek: ");
   string stavek = Console.ReadLine();
   Console.Write("Vnesi znak, ki te zanima: ");
   char znak = Convert.ToChar(Console.Read());
   Console.WriteLine("V stavku je " + Kolikokrat(stavek, znak)+" znakov  "+znak+".");
}
```

13.6.2 Povprečje ocen

Napišimo program, ki bo sprejel nekaj 10 ocen, jih "zložil" v tabelo in nam vrnil povprečno, najnižjo in najvišjo oceno.

```
public static void Main(string[] args)
{
   int[] ocene = new int[10];
   for(int stevec = 0; stevec < ocene.Length; stevec++)
   {
      Console.Write("Vnesi " + (stevec + 1) + ". oceno:");
      ocene[stevec] = int.Parse(Console.ReadLine());
```

```
    }
    double povprecje = PovprecjeOcen(ocene);
    Console.WriteLine("Tvoje povprečje je " + povprecje + ".");
    Console.WriteLine("Zaokroženo povp.je " + (int)(povprecje + 0.5) + ".");
    Console.WriteLine("Najnižja ocena je " + MinOcena( ocene));
    Console.WriteLine("Najvišja ocena je " + MaxOcena( ocene));
}

public static int MinOcena(int[] ocene)  //  metoda minOcena
{
    int najnizja = ocene[0];
    for(int i = 1; i < ocene.Length; i++)
    {
        najnizja = Math.Min(najnizja, ocene[i]);
    }
    return najnizja;
}
public static int MaxOcena(int[] ocene)  //  metoda maxOcena
{
    int najvisja = ocene[0];
    for(int i = 1; i < ocene.Length; i++)
    {
        najvisja = Math.Max(najvisja, ocene[i]);
    }
    return najvisja;
}

public static int VsotaOcen(int[] ocene)  // metoda  vsotaOcen
{
    int vsota = 0;
    for(int i = 0; i < ocene.Length; i++)
    {
        vsota = vsota + ocene[i];
    }
    return vsota;
}

public static double PovprecjeOcen(int[] ocene)  //  metoda povprecjeOcen
{
   return (double)vsotaOcen(ocene) / ocene.Length;
}
```

Program je sestavljen iz petih metod. Začnimo pri metodi *MinOcena*, prvo oceno v naši tabeli označimo za najnižjo, postopoma se sprehodimo čez preostal del tabele in na vsakem koraku v spremenljivko *najnizja* shranimo trenutno najnižjo vrednost. Za primerjavo uporabimo metodo *Min()*iz razreda *Math*, ki nam vrne manjše število od obeh, ker je metoda tipa *int* nam vrne vrednost in to je mimimalna ocena. Sledi ji metoda *MaxOcena*, ki je podobna prejšnji

metodi le da ta vrne maksimalno oceno. Naslednja metoda je *VsotaOcen*, ki vrne vsoto vseh ocen. S pomočjo zanke *for* se sprehodimo čez tabelo in seštevamo vrednosti. Metoda *PovprecjeOcen* pa vrne povprečje, ki ga dobimo z preprosto enačbo vsoto ocen deljeno z dolžino tabele ocene.
Še zadnja je metoda *main,* v kateri preberemo podatke, jih zložimo v tabelo in s pomočjo klicev metod, ki smo jih ustvarili znotraj tega programa, izpišemo statistiko naših ocen.

13.6.3 Dopolnjevanje niza

Napišimo metodo *DopolniNiz*, ki sprejme niz *s* in naravno število *n* ter vrne niz dolžine *n*. V primeru, da je dolžina niza s večja od *n*, spustimo zadnjih nekaj znakov. Drugače pa na konec niza *s* dodamo še ustrezno število znakov '+'. Preverimo delovanje metode.

```
public static string DopolniNiz(string niz, int n)
{   //  Metoda DopolniNiz vrne nov niz dolžine n, ki ga dobimo tako, da bodisi skrajšamo
    //  niz niz, ali pa ga dopolnimo z znaki '+'.
   int dolzina = niz.Length;  //  Določimo dolžino niza
   string novNiz = "";  //  Nov niz

   if (dolzina > n)
   {  //  Dolžina niza je večja od n
      novNiz = niz.Substring(0, n);
   }
   else
   {  //  Dolžina niza je manjša ali enaka n
      novNiz += niz;
      for (int i = dolzina; i < n; i++)
      {
         novNiz += '+';  //  Dodamo manjkajoče znake'+'
      }
   }
   return novNiz;  //  Vrnemo nov niz
}
public static void Main(string[] args)
{
    Console.Write("Vnesi niz: ");
    string niz = Console.ReadLine();
    Console.Write("Vnesi n: ");
    int n = int.Parse(Console.ReadLine());
    Console.WriteLine("Nov niz: " + DopolniNiz(niz, n));  //  Klic metode
}
```

Razlaga.
Program začnemo z metodo *DopolniNiz*, ki ji v deklaraciji podamo parametra *niz* in *n*. V metodi določimo dolžino niza *niz*, ki jo shranimo v spremenljivko *dolzina*. Določimo nov niz *novNiz*, ki je na začetku prazen. Preverimo pogoj v pogojnem stavku. Če je izpolnjen (resničen), kličemo metodo *Substring()*, s katero izluščimo podniz dolžine *n*. Rezultat te metode shranimo v spremenljivki *novNiz*. Če pogoj ni izpolnjen (neresničen), potem nizu

novNiz dodamo niz *niz* in ustrezno število znakov '+'. Te nizu dodamo s pomočjo zanke *for*. Na koncu metode s ključno besedo *return* vrnemo niz *novNiz*.
Delovanje metode preverimo v glavni metodi *Main*. V tej metodi preberemo podatka iz konzole, ju shranimo v spremenljivki *niz* in *n* ter s pomočjo klicane metode *DopolniNiz* izpišemo spremenjeni niz.

13.6.4 Papajščina

Sestavimo metodo *public static string Papajscina(string s)*, ki dani niz s pretvori v "papajščino". To pomeni, da za vsak samoglasnikom, ki se pojavi v nizu, postavi črko p in samoglasnik ponovi.

Primer:

Ko se izvedejo ukazi
string s = "Danes je konec šole.";
string papaj = Papajscina(s);
je niz papaj enak "Dapanepes jepe koponepec šopolepe.".

```
// metoda
public static string Papajscina(string s)
{
   string sPapaj = "";
   for (int i=0;i<s.Length;i++)
   {
       // metoda bo delovala tudi če so v stavku velike črke
      char znak=char.ToUpper(s[i]);
      if (znak=='A' || znak=='E' || znak=='I' || znak=='O' || znak=='U')
         sPapaj = sPapaj + s[i] + 'p'+s[i];  // dodamo znak 'p' in še samoglasnik ki se ponovi
      else sPapaj = sPapaj + s[i];
    }
    return sPapaj;  // vračanje papajščine
}

public static void Main(string[] args)
{
  string s = "Danes je konec šole.";
  string papaj = Papajscina(s);
  Console.WriteLine(papaj);
}
```

13.7 POVZETEK

Naučili smo se pisati lastne metode. Prav v vseh programskih jezikih obstaja ogromno število že napisanih metod in preden se odločimo za pisanje nove metode, se raje prepričajmo, če morda taka ali podobna metoda že ne obstaja. Seveda pa bomo slej ko prej naleteli na problem, ko bo pisanje nove metode neizbežno. Naš program bo tako postal bolj pregleden, pa še popravljanje ali pa ažuriranje kadarkoli kasneje bo veliko lažje.

Da bomo utrdili znanje, najprej rešimo kviz, ki je dostopen na http://up.fmf.uni-lj.si/staticne_metode-da2a381e/index.html. Nato pa nas čaka pisanje še nekaj programov, kjer bomo uporabili metode. Primerne naloge najdete v zbirki Lokar M. in Uranič S., Programiranje 1 - zbirka rešenih nalog v C#.

14 OBJEKTNO PROGRAMIRANJE

14.1 UVOD

Pri programiranju se je izkazalo, da je zelo koristno, če podatke in postopke nad njimi združujemo v celote, ki jih imenujemo **objekte**. Programiramo potem tako, da objekte ustvarimo in jim naročamo, da izvedejo določene postopke. Programskim jezikom, ki omogočajo tako programiranje, rečemo, da so objektno (ali z drugim izrazom predmetno) usmerjeni. Objektno usmerjeno programiranje nam omogoča, da na problem gledamo kot na množico objektov, ki med seboj sodelujejo in vplivajo drug na drugega. Na ta način lažje pišemo sodobne programe.

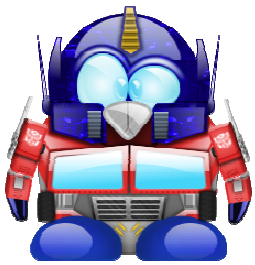
Objektno usmerjenih je danes večina sodobnih programskih jezikov. Pri njihovi uporabi hitro naletimo na pojme, kot so združevanje (**enkapsulacija**), večličnost (**polimorfizem**), dedovanje (**inheritence**), ki so vsaj na prvi pogled zelo zapleteni. Nas podrobnosti ne bodo zanimale in bomo ogledali le osnove.

14.2 OBJEKTNO PROGRAMIRANJE – KAJ JE TO

Pri objektno usmerjenem programiranju se ukvarjamo z … objekti seveda. Objekt je nekakšna črna škatla, ki dobiva in pošilja sporočila. V tej črni škatli (objektu) se skriva tako koda (torej zaporedje programskih stavkov), kot tudi podatki (informacije nad katerimi se izvaja koda). Pri klasičnem programiranju imamo kodo in podatke ločene. Tudi mi smo do sedaj programirali več ali manj na klasičen način. Pisali smo metode, ki smo jim podatke posredovali preko parametrov. Parametri in metode niso bile prav nič tesno povezane. Če smo npr. napisali metodo, ki je na primer poiskala največji element v tabeli števil, tabela in metoda nista bili združeni – tabela nič ni vedela o tem, da naj bi kdo npr. po njej iskal največje število.

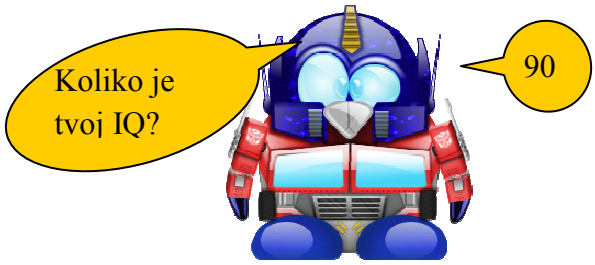
V objektno usmerjenem programiranju (OO) pa sta koda in podatki združeni v nedeljivo celoto – objekt. To prinaša določene prednosti. Ko uporabljamo objekte, nam nikoli ni potrebno pogledati v sam objekt. To je dejansko prvo pravilo OO programiranja – uporabniku ni nikoli potrebno vedeti, kaj se dogaja znotraj objekta. Gre dejansko zato, da nad objektom izvajamo različne metode. In za vsak objekt se točno ve, katere metode zanj obstajajo in kakšne rezultate vračajo. Recimo, da imamo določen objekt z imenom *robotek*. Vemo, da objekte take vrste, kot je *robotek,* lahko povprašamo o tem, koliko je njihov IQ. To storimo tako, da nad njimi izvedemo metodo

robotek.KolikoJeIQ()



Slika 25: Objekt *robotek*
Vir: http://commons.wikimedia.org/wiki/Crystal_Clear

Objekt odgovori s sporočilom (metoda vrne rezultat), ki je neko celo število.



Slika 26: Objekt sporoča
Vir: http://commons.wikimedia.org/wiki/Crystal_Clear

Tisto, kar je pomembno, je to, da nam, kot uporabnikom objekta, ni potrebno vedeti, kako objekt izračuna IQ (je to podatek, ki je zapisan nekje v objektu, je to rezultat nekega preračunavanja …?). Kot uporabnikom nam je važno le, da se z objektom lahko pogovarjamo o njegovem IQ. Vse "umazane podrobnosti" o tem, kaj je znotraj objekta, je prepuščeno tistim, ki napišejo kodo, s pomočjo katere se ustvari objekt.

Seveda se lahko zgodi, da se kasneje ugotovi, da je potrebno "preurediti notranjost objekta". In tu se izkaže prednost koncepta objektnega programiranja. V programih, kjer objekte uporabljamo, nič ne vemo o tem, kaj je "znotraj škatle", kaj se na primer dogaja, ko objekt povprašamo po IQ. Z objektom "komuniciramo" le preko izvajanja metod. Ker pa se klic metode ne spremeni, bodo programi navkljub spremembam v notranjosti objekta še vedno delovali.

Če torej na objekte gledamo kot na črne škatle (zakaj črne – zato, da se ne vidi, kaj je znotraj!) in z njimi ravnamo le preko sporočil (preko klicev metod), naši programi delujejo ne glede na morebitne spremembe v samih objektih.

Omogočanje dostopa do objekta samo preko sporočil in onemogočanje uporabnikom, da vidijo (in brskajo) po podrobnostih, se imenuje skrivanje informacij ali še bolj učeno **enkapsulacija**. In zakaj je to pomembno? Velja namreč, da se programi ves čas spreminjajo. Velika večina programov se dandanes ne napiše na novo, ampak se spremeni obstoječ program. In večina napak izhaja iz tega, da nekdo spremeni določen delček kode, ta pa potem povzroči, da drug del programa ne deluje več. Če pa so tisti delčki varno zapakirani v kapsule, se spremembe znotraj kapsule ostalega sistema prav nič ne tičejo.

Pravzaprav smo se že ves čas ukvarjali z objekti, saj smo v naših programih uporabljali različne objekte, ki so že pripravljeni v standardnih knjižnicah jezika C#. Oglejmo si dva primera objektov iz standardne knjižnice jezika:

- Objekt **System.Console** predstavlja *standardni izhod*. Kadar želimo kaj izpisati na zaslon, pokličemo metodo *Write* na objektu **System.Console**.
- Objekt tipa *Random* predstavlja generator naključnih števil. Metoda *Next(a, b)*, ki jo izvedemo nad objektom tega tipa nam vrne neko naključno celo število med *a* in *b*.

In če se vrnemo na razlago v uvodu – kot uporabniki čisto nič ne vemo (in nas pravzaprav ne zanima), kako metoda *Next* določi naključno število. In tudi, če se bo kasneje "škatla"

Random spremenila in bo metoda *Next* delovala na drug način, to ne bo "prizadelo" programov, kjer smo objekte razreda *Random* uporabljali. Seveda, če bo sporočilni sistem ostal enak. V omenjenem primeru to pomeni, da se bo metoda še vedno imenovala *Next*, da bo imela dva parametra, ki bosta določala meje za naključna števila in da bo odgovor (povratno sporočilo) neko naključno število z želenega intervala.

Naučili smo se torej uporabnosti objektnega programiranja, nismo pa se še naučili izdelave svojih razredov. Seveda nismo omenjeni le na to, da bi le uporabljali te "črne škatle", kot jih pripravi kdo drug (npr. jih dobimo v standardni knjižnici jezika). Objekti so uporabni predvsem zato, ker lahko programer definira nove razrede in objekte, torej sam ustvarja te nove črne škatle, ki jih potem on sam in drugi uporablja.

14.3 RAZRED

Razred je abstraktna definicija objekta (torej opis, kako je naša škatla videti znotraj). Z razredom opišemo, kako je neka vrsta objektov videti. Če na primer sestavljamo razred zajec, opisujemo, katere so tiste lastnosti, ki določajo vse zajce.

Če želimo nek razred uporabiti, mora običajno obstajati vsaj en **primerek** razreda. Primerek nekega razreda imenujemo **objekt**. Ustvarimo ga s ključno besedo **new**.

```
imeRazreda primerek = new imeRazreda();
```

Lastnosti objektov določene vrste so zapisane v razredu (*class*). Ta opisuje katere **podatke** hranimo o objektih te vrste in katere so **metode** oz. odzivi objektov na sporočila. Stanje objekta torej opišemo s spremenljivkami (rečemo jim tudi **polja ali komponente**), njihovo obnašanje oz. odzive pa z metodami.

Oglejmo si primer programa v C#, ki uporablja objekte, ki jih napišemo sami.

```
public class MojR
{
   private string mojNiz;

   public MojR(string nekNiz)
   {
      mojNiz = nekNiz;
   }

   public void Izpisi()
   {
      Console.WriteLine(mojNiz);
   }
}
public static void Main(string[] arg)
{
   MojR prvi;
   prvi= new MojR("Pozdravljen, moj prvi objekt v C#!");
   prvi.Izpisi();
}
```

Definicija razreda

oznaka (ime) objekta

kreiranje objekta

ukaz objektu

Na kratko razložimo, "kaj smo se šli". Pred glavnim programom smo definirali nov razred z imenom *MojR*. Ta je tak, da o vsakem objektu te vrste poznamo nek niz, ki ga hranimo v njem. Vse znanje, ki ga objekti tega razreda imajo je, da se znajo odzvati ne metodo *Izpis()*, ko izpišejo svojo vsebino (torej niz, ki ga hranimo v njem).

Glavni program *Main* (glavna metoda) je tisti del rešitve, ki opravi dejansko neko delo. Najprej naredimo primerek objekta iz razreda *MojR* (prvi) in vanj shranimo niz *"Pozdravljen, moj prvi objekt v C#!"*. Temu objektu nato naročimo, naj izpiše svojo vsebino.

Povejmo še nekaj o drugem načelu združevanja (enkapsulacije), o pojmu **dostopnost**. Potem, ko smo metode in polje združili znotraj razreda, smo pred temeljno odločitvijo, kaj naj bo javno, kaj pa zasebno. Vse, kar smo zapisali med zavita oklepaja v razredu, spada v notranjost razreda. Z besedicami **public, private** in **protected,** ki jih napišemo pred ime metode ali polja, lahko kontroliramo, katere metode in polja bodo dostopna tudi od zunaj:

- Metoda ali polje je privatno (**private**), kadar je dostopno le znotraj razreda.
- Metoda ali polje je javno (**public**), kadar je dostopno tako znotraj, kot tudi izven razreda.
- Metoda ali polje je zaščiteno (**protected**), kadar je vidno le znotraj razreda, ali pa v **podedovanih** (**izpeljanih**) razredih.

Obstajajo tudi druge dostopnosti, a se z njimi ne bomo ukvarjali. Prav tako ne bomo uporabljali zaščite *protected*. Omejili se bomo le na privatni (*private*) in javni dostop (*public*).

14.4 USTVARJANJE OBJEKTOV

V prejšnjem primeru smo iz razreda *MojR* tvorili objekt prvi.

```
MojR prvi;  // prvi je NASLOV objekta
```

Z naslednjim ukazom v pomnilniku naredimo objekt tipa *MojR*, po pravilih, ki so določena z opisom razreda *MojR*. Novo ustvarjeni objekt se nahaja na naslovu *prvi*.

```
prvi = new MojR("Pozdravljen, moj prvi objekt v C#!");
```

Razred je tako v bistvu **šablona**, ki definira spremenljivke in metode skupne vsem objektom iste vrste, objekt pa je **primerek** (srečali boste tudi "čuden" izraz instanca) nekega razreda.

14.4.1 Naslov objekta

Sicer vedno rečemo, da v spremenljivki *prvi* hranimo objekt, a zavedati se moramo, da to ni čisto res. V spremenljivki *prvi* je shranjen **naslov** objekta. Zato po

```
MojR prvi, drugi;
```

nimamo še nobenega objekta. Imamo le dve spremenljivki, v kateri lahko shranimo naslov, kjer bo operator *new* ustvaril nov objekt. Rečemo tudi, da sta spremenljivki *prvi* in *drugi* kazali na objekta tipa *MojR*. Če potem napišemo

```
prvi = new MojR("1.objekt");
```

smo v spremenljivko *prvi* shranili naslov, kjer je novo ustvarjeni objekt. Ustvarimo še en objekt in njegov naslov shranimo v spremenljivko *drugi*

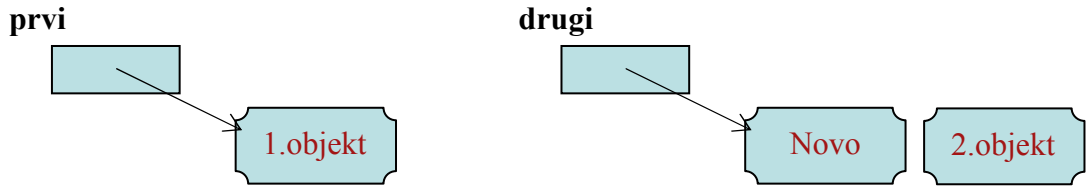
```
drugi = new MojR("2. objekt");
```

Poglejmo, kakšno je sedaj stanje v pomnilniku. S puščico v spremenljivkah *prvi* in *drugi* smo označili ustrezen naslov objektov. Dejansko je na tistem mestu napisano nekaj v stilu *h002a22* (torej naslov mesta v pomnilniku)

In če sedaj napišemo

```
drugi = new MojR("Novo");
```

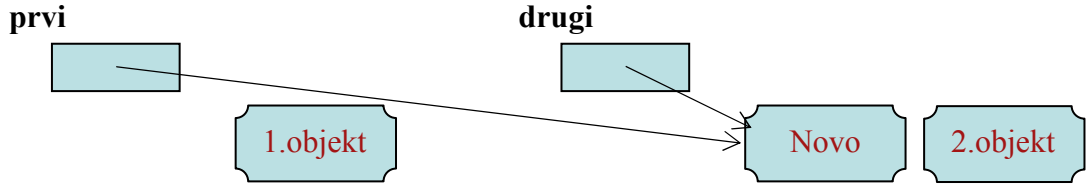
je vse v redu. Le do objekta z vsebino "2. objekt" ne moremo več! Stanje v pomnilniku je



in če naredimo še

```
prvi = drugi;
```

smo s tem izgubili še dostop do objekta, ki smo ga prvega ustvarili in pomnilnik bo videti takole.



Sedaj tako spremenljivka *prvi* in *drugi* vsebujeta naslov istega objekta. Do objektov z vsebino "1. Objekt" in "2. objekt" pa ne more nihče več. Na srečo bo kmalu prišel smetar in ju

pometel proč. Smetar je poseben program v C#, za katerega delovanje nam ni potrebno skrbeti in poskrbi, da se po pomnilniku ne nabere preveč objektov, katerih naslov ne pozna nihče več.

Sicer bomo vedno govorili kot: ... *v objektu mojObjekt imamo niz, nad objektom zajecRjavko izvedemo metodo ...,* a vedeti moramo, da sta v spremenljivkah *mojObjekt* in *zajecRjavko* naslova in ne dejanska objekta.

Kako se torej lotiti načrtovanja rešitve s pomočjo objektnega programiranja?

- Z analizo ugotovimo, kakšne objekte potrebujemo za reševanje.
- Pregledamo, ali že imamo na voljo ustrezen razred (standardne knjižnice, druge knjižnice, naši stari razredi).
- Sestavimo ustrezen razred (določimo polja in metode našega razreda).
- Sestavimo "glavni" program, kjer s pomočjo objektov rešimo problem.

14.5 ZGLEDI

14.5.1 Ulomek

Radi bi napisali program, ki bo prebral polmer kroga in izračunal ter izpisal njegovo ploščino. "Klasično" bi program napisali takole:

```
public static void Main(string[] arg)
{
     //  vnos podatka
    Console.Write("Polmer kroga: ");
    int r = int.Parse(Console.ReadLine());
     //  izračun
    double ploscina = Math.PI * r * r;
     //  izpis
    Console.WriteLine("Ploščina kroga: " + ploscina);
}
```

Sedaj pa to naredimo še "objektno". Očitno bomo potrebovali razred *Krog*. Objekt tipa *Krog* hrani podatke o svojem polmeru. Njegovo znanje pa je, da "zna" izračunati svojo ploščino.

```
class Krog
{
   public double polmer;   // polje razreda Krog
   public double Ploscina()    // metoda razreda krog
   {
       return Math.PI * polmer * polmer;
   }
}
public static void Main(string[] arg)
{
   Krog k = new Krog();  // naredimo nov objekt tipa krog
```

```
    Console.Write("Polmer: ");
    k.polmer = double.Parse(Console.ReadLine());  // polmer preberemo
    Console.WriteLine(k.Ploscina());   // izpis ploščine
}
```

14.5.2 Zgradba

Pogosto razrede uporabimo le zato, da v celoto združimo podatke o neki stvari. Takrat v razred "zapremo" le polja, ki imajo vsa javni dostop. V razredu pa metod ni.

Napišimo razred *Zgradba* z poljema *kvadratura* in *stanovalcev*.

```
class Zgradba // deklaracija razreda Zgradba z dvema javnima poljema.
{
    public int kvadratura;
    public int stanovalcev;
}
```

Če sedaj ustvarimo objekt tipa *Zgradba*

```
   Zgradba hiša = new Zgradba();   //  nov objekt razreda Zgradba
```

do polj *kvadratura* in *stanovalcev* dostopamo z

```
   hiša.stanovalcev = 4;
```

oziroma

```
   hiša.kvadratura = 2500;
```

Če imamo torej nek razred *ImeRazreda* in v njem polje *nekoPolje* in je objekt *mojObjekt* tipa *ImeRazreda*, z

```
   mojObjekt.nekoPolje
```

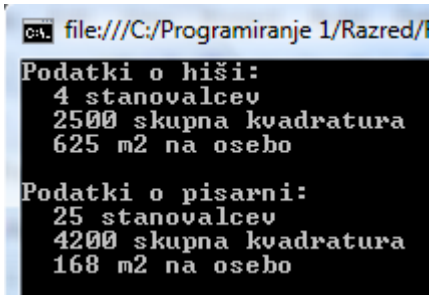
dostopamo do te spremenljivke. Uporabljamo jo enako, kot vse spremenljivke tega tipa (če je *nekoPolje* tipa *double*, z *mojObjekt.nekoPolje* lahko počnemo vse tisto, kar pač lahko počnemo s spremenljivkami tipa *double*).

V metodi *Main* kreirajmo dva objekta in za vsakega posebej ugotovimo, kolikšna stanovanjska površina pride na posameznika.

```
static void Main(string[] args)
{
   Zgradba hiša = new Zgradba();   // nov objekt razreda Zgradba
   Zgradba pisarna = new Zgradba();  // nov objekt razreda Zgradba
   int kvadraturaPP;  //  spremenljivka za izračun kvadrature na osebo
    // določimo začetne vrednosti prvega objekta
   hiša.stanovalcev = 4;
   hiša.kvadratura = 2500;
```

```
    // določimo začetne vrednosti drugega objekta
    pisarna.stanovalcev = 25;
    pisarna.kvadratura = 4200;

    // izračun kvadrature za prvi objekt
    kvadraturaPP = hiša.kvadratura / hiša.stanovalcev;

    Console.WriteLine("Podatki o hiši:\n  " + hiša.stanovalcev + " stanovalcev\n  " +
            hiša.kvadratura + " skupna kvadratura\n  " + kvadraturaPP + " m2 na osebo");

    Console.WriteLine();
    // izračun kvadrature za drugi objekt
    kvadraturaPP = pisarna.kvadratura / pisarna.stanovalcev;

    Console.WriteLine("Podatki o pisarni:\n  " + pisarna.stanovalcev + " stanovalcev\n  " +
        pisarna.kvadratura + " skupna kvadratura\n  " + kvadraturaPP + " m2 na osebo");

}
```

Izpis, ki ga dobimo, je naslednji:



Slika 27: Uporaba razreda *Zgradba*

Razred *Zgradba* je sedaj nov tip podatkov, ki ga pozna C#. Uporabljamo ga tako, kot vse druge, v C# in njegove knjižnice vgrajene tipe. Torej lahko napišemo

```
Zgradba[] ulica;   // ulica bo tabela objektov tipa Zgradba
```

Pri tem pa moramo sedaj paziti na več stvari. Zato bomo o tabelah objektov spregovorili v posebnem razdelku.

14.5.3 Evidenca članov kluba

Napišimo program, ki vodi evidenco o članih športnega kluba. Podatki o članu obsegajo ime, priimek, letnico vpisa v klub in vpisno številke (seveda je to poenostavljen primer). Torej objekt, ki predstavlja člana kluba, vsebuje štiri podatke. Ustrezni razred je:

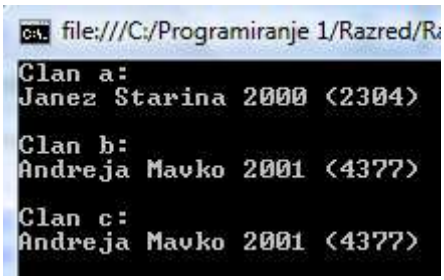
```
public class Clan
{
    public string ime;
    public string priimek;
    public int leto_vpisa;
```

```
    public string vpisna_st;
}
```

S tem smo povedali, da je vsak *objekt* tipa *Clan* sestavljen iz štirih komponent: *ime*, *priimek*, *leto_vpisa* in *vpisna_st*. Če je *a* objekt tipa *Clan*, potem lahko dostopamo do njegove komponente *ime* tako, da napišemo *a.ime*. Tvorimo nekaj objektov:

```
1:      static void Main(string[] args)
2:      {
3:          Clan a = new Clan();
4:          a.ime = "Janez";
5:          a.priimek = "Starina";
6:          a.leto_vpisa = 2000;
7:          a.vpisna_st = "2304";
8:          Clan b = new Clan();
9:          b.ime = "Mojca";
10:         b.priimek = "Mavko";
11:         b.leto_vpisa = 2001;
12:         b.vpisna_st = "4377";
13:         Clan c = b;
14:         c.ime = "Andreja";
15:         Console.WriteLine("Član a:\n" + a.ime + " " + a.priimek +
                              " " + a.leto_vpisa + " (" + a.vpisna_st + ")\n");

16:         Console.WriteLine("Član b:\n" + b.ime + " " + b.priimek +
                              " " + b.leto_vpisa + " (" + b.vpisna_st + ")\n");

17:         Console.WriteLine("Član c:\n" + c.ime + " " + c.priimek +
                              " " + c.leto_vpisa + " (" + c.vpisna_st + ")\n");

18:     }
```

Ko program poženemo, dobimo naslednji izpis:



Slika 28: Objekti tipa *Clan*

Hm, kako to, da sta dva zadnja izpisa enaka? V programu smo naredili dva nova objekta razreda *Clan*, ki sta shranjena v spremenljivkah *a* in *b*. A spomnimo se, da imamo v spremenljivkah *a* in *b* shranjena *naslova* objektov, ki smo ju naredili. Kot vedno, nove objekte naredimo z ukazom *new*. Spremenljivka *c* pa **kaže na isti objekt kot** *b*, se pravi, da se v 13.

vrstici *ni* naredila nova kopija objekta *b*, ampak se spremenljivki *b* in *c sklicujeta* na isti objekt. In zato smo z vrstico 14 vplivali tudi na ime, shranjeno v objektu *b (*bolj točno, na ime, shranjeno v objektu, katerega naslov je shranjen v *b*)!

V vrsticah 4 − 7 smo naredili objekt *a* (naredili smo objekt in nanj pokazali z *a*) in nastavili vrednosti njegovih komponent. Takole nastavljanje je v praksi dokaj neprimerno, ker se zlahka zgodi, da kako komponento pozabimo nastaviti. Zato C# omogoča, da delamo nove objekte na bolj praktičen način s pomočjo **konstruktorjev**.

14.6 KONSTRUKTOR

Ob tvorbi objekta bi radi hkrati nastavili začetno stanje polj. To nam omogočajo konstruktorji. **Konstruktor** je metoda, ki jo pokličemo ob tvorbi objekta z *new*. Je brez tipa rezultata. Ne smemo jih zamenjati z metodami, ki rezultata ne vračajo (te so tipa *void*). Konstruktor tipa rezultata sploh nima, tudi *void* ne. Prav tako smo omejeni pri izbiri imena te metode. Konstruktor mora imeti ime nujno tako, kot je ime razreda. Če konstruktorja ne napišemo, ga "naredi" prevajalnik sam (a metoda ne naredi nič). Vendar pozor: kakor hitro napišemo vsaj en svoj konstruktor, C# ne doda svojega.

Največja omejitev, ki loči konstruktorje od drugih metod je ta, da konstruktorja ne moremo klicati na poljubnem mestu (kot lahko ostale metode). Kličemo ga izključno skupaj z *new*, npr. *new Drzava()*. Uporabljamo jih za vzpostavitev začetnega stanja objekta.

14.7 ZGLEDI

14.7.1 Nepremičnine

Sestavimo razred, ki bo v svoja polja lahko shranil ulico, številko nepremičnine ter vrsto nepremičnine. Ustvarimo poljuben objekt, ga inicializirajmo in ustvarimo izpis, ki naj zgleda približno takole: Na naslovu Cankarjeva ulica 32, Kranj je blok.

```
class Nepremicnina
{
    public string ulica;
    public int stevilka;
    public string vrsta;
     // konstruktor
    public Nepremicnina(string kateraUlica,  int hisnaStevilka, string vrstaNep)
    {
        this.ulica = kateraUlica;
        this.stevilka = hisnaStevilka;
        this.vrsta = vrstaNep;
    }
}

 static void Main(string[] args)
{
     // za kreiranje novega objekta uporabimo konstruktor
```

```
    Nepremicnina nova = new Nepremicnina("Cankarjeva ulica 32, Kranj",1234,"blok");
     // še izpis podatkov o nepremičnini
    Console.WriteLine("Na naslovu " + nova.ulica + " je " + nova.vrsta);
    Console.ReadKey();
}
```

14.7.2 this

V zgornjem primeru smo v konstruktorju uporabili prijem, ki ga doslej še nismo srečali. Napisali smo *this*. Kaj to pomeni?

this označuje objekt, ki ga "obdelujemo". V konstruktorju je to objekt, ki ga ustvarjamo. *this.ulica* se torej nanaša na lastnost/komponento spol objekta, ki se ustvarja (ki ga je ravno naredil *new*).

Denimo, da v nekem programu napišemo

```
Zajec rjavko = new Zajec();
Zajec belko= new Zajec();
```

Pri prvem klicu se ob uporabi konstruktorja *Zajec() this* nanašal na *rjavko*, pri drugem na *belko*.

Na ta način (z *this*) se lahko sklicujemo na objekt vedno, kadar pišemo metodo, ki jo izvajamo nad nekim objektom. Denimo, da smo napisali

```
Random ng = new Random();
Random g2 = new Random();
Console.WriteLine(ng.Next(1, 10));
```

Kako so programerji, ki so sestavljali knjižnico in v njej razred *Random* lahko napisali kodo metode, da se je vedelo, da pri metodi *Next* mislimo na uporabo generatorja *ng* in ne na *g2*?

Denimo, da imamo razred *MojRazred* (v njem pa komponento *starost*) in v njem metodo *MetodaNeka*. V programu, kjer razred *MojRazred* uporabljamo, ustvarimo dva objekta tipa *MojRazred*. Naj bosta to *objA* in *objC*. Kako se znotraj metode *MetodaNeka* sklicati na ta objekt (objekt, nad katerim je izvajana metoda)? Če torej metodo *MetodaNeka* kličemo nad obema objektoma z *objA.MetodaNeka()* oziroma z *objC.MetodaNeka()*, kako v postopku za *metodaNeka* povedati, da gre:

- prvič za objekt z imenom *objA* in
- drugič za objekt z imenom *objC*

Če se moramo v kodi metode *metodaNeka* sklicati recimo na komponento *starost* (jo recimo izpisati na zaslon), kako povedati, da naj se ob klicu *objA.MetodaNeka()* uporabi starost objekta *objA*, ob klicu *objC.MetodaNeka()* pa starost objekta *objC*?

```
Console.WriteLine("Starost je: " + ?????.starost);
```

Ob prvem klicu so ???? *objA*, ob drugem pa *objC*. To "zamenjavo" dosežemo z *this*. Napišemo

```
Console.WriteLine("Starost je: " + this.starost);
```

Ob prvem klicu *this* pomeni *objA*, ob drugem pa *objC*.

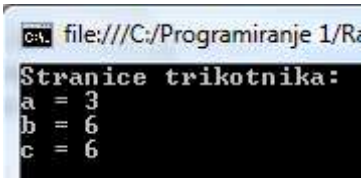
Torej v metodi na tistem mestu, kjer potrebujemo konkretni objekt, nad katerim metodo izvajamo, napišemo *this*.

14.7.3 Trikotnik

Napišimo razred *Trikotnik*, ki bo vseboval konstruktor za generiranje poljubnih stranic trikotnika, pri čemer bomo upoštevali trikotniško pravilo: vsota poljubnih dveh stranic trikotnika mora biti daljša od tretje stranice.

```
class Trikotnik // deklaracija razreda Trikotnik
{
      public int a,b,c;   // stranice trikotnika
       // konstruktor
      public Trikotnik()
      {
          Random naklj = new Random();
           this.a = naklj.Next(1,10);
           this.b = naklj.Next(1,10);
            // tretjo stranico delamo toliko časa, da stranice zadoščajo trikotniškemu pravilu
           while (true)   // neskončna zanka
           {
               this.c = naklj.Next(1,10);
                // sestavljeni pogoj za preverjanje stranic
               if ((a + b) > c && (a + c) > b && (b + c) > a)
                     break;  // če dolžina ustreza, izstopimo iz zanke
           }
      }
}

static void Main(string[] args)
{
    Trikotnik trik1 = new Trikotnik();
    Console.WriteLine("Stranice trikotnika: \na = " + trik1.a + "\nb = " + trik1.b
                              + "\nc = " + trik1.c);
}
```

Izpis:



Slika 29: Razred Trikotnik

V pogojnem stavku

```
if ((a + b) > c && (a + c) > b && (b + c) > a)  // sestavljeni pogoj za preverjanje stranic
```

nismo pisali *this.a*, *this.b* in *this.c*

```
if ((this.a + this.b) > this.c && ...
```

a se prevajalnik ni pritožil. Namreč, če ni možnosti za zamenjavo, *this*. lahko izpustimo. A vsaj začetnikom bo verjetno lažje, če bodo *this*. vsaj v začetku vedno pisali.

Če potrebujemo več načinov, kako nastaviti začetne vrednosti polj nekega objekta, moramo pripraviti več konstruktorjev. A pri tem imamo težavo. Namreč vsi konstruktorji se morajo imenovati tako kot razred. Kako pa bo C# potem vedel, kateri konstruktor mislimo? Če si ogledamo zgled, kjer smo razred *Trikotnik* razširili še z enim konstruktorjem, vidimo, da očitno zadeve gredo. Težav ni, ker C# podpira **preobteževanje** metod, ki jo bomo spoznali v naslednjem razdelku.

```
class Trikotnik // deklaracija razreda Trikotnik
{
      public int a,b,c;   // stranice trikotnika
       // konstruktor
      public Trikotnik()
      {
          Random naklj = new Random();
           this.a = naklj.Next(1,10);
           this.b = naklj.Next(1,10);
            // tretjo stranico delamo toliko časa, da stranice zadoščajo trikotniškemu pravilu
           while (true)   // neskončna zanka
           {
               this.c = naklj.Next(1,10);
               if ((a + b) > c && (a + c) > b &&
                    (b + c) > a)  // sestavljeni pogoj za preverjanje stranic
                  break;  // če dolžina ustreza, izstopimo iz zanke
            }
      }
      public Trikotnik(int a, int b, int c)
      {
           this.a = a;
           this.b = b;
           this.a = c;
      }
}
```

```
static void Main(string[] args)
{
    Trikotnik trik1 = new Trikotnik();
    Trikotnik trik2 = new Trikotnik(3, 4, 5);
    Console.WriteLine("Stranice trikotnika: \na = " + trik1.a + "\nb = " + trik1.b
                      + "\nc = " + trik1.c);
    Console.WriteLine("Stranice trikotnika: \na = " + trik2.a + "\nb = " + trik2.b
                      + "\nc = " + trik2.c);
}
```

V tem zgledu vidimo še primer, ko je *this*. nujno uporabljati. Namreč pri

```
this.a = a;
```

moramo razlikovati med spremenljivko *a*, ki označuje parameter metode (konstruktorja) in med poljem *a*. Z uporabo *this.a* dileme ni. Če pa *this.* ne uporabimo, se vedno uporabi "najbližja" definicija spremenljivke. V našem primeru bi to bil parameter *a*.

14.8 PREOBTEŽENE METODE

Jezik C# podpira tako imenovano preobteževanje (*overloading*) metod. Tako imamo lahko znotraj istega programa več metod z enakim imenom. Metode se morajo razlikovati ne v imenu, ampak podpisu. Lahko si predstavljamo, da je **podpis** metode sestavljen iz imena metode in tipov vseh parametrov. Konstruktor *Denarnica()* ima torej podpis enostavno *Denarnica*, konstruktor *public denarnica(string ime, double znesek)* pa podpis *Denarnica_string_double*. Tip rezultata (*return tip*) **ni** del podpisa!

Preobtežene so lahko tudi “navadne” metode (ne le konstruktorji). Zakaj je to uporabno? Denimo, da imamo metodo *Ploscina*, ki kot parameter dobi objekt iz razreda *Kvadrat*, *Krog* ali *Pravokotnik*. Kako jo začeti?

```
public static double Ploscina(X lik)
```

Ker ne vemo, kaj bi napisali za tip parametra, smo napisali kar *X*. Če preobteževanje ne bi bilo možno, bi morali napisati metodo, ki bi sprejela parameter tipa *X*, kjer je *X* bodisi *Kvadrat*, *Krog* ali *Pravokotnik*. Seveda to ne gre, saj prevajalnik nima načina, da bi zagotovil, da je *X* oznaka bodisi za tip *Kvadrat*, tip *Krog* ali pa tip *Pravokotnik*. Pa tudi če bi šlo – kako bi nadaljevali? V kodi bi morali reči: če je lik tipa *Kvadrat*, uporabi to formulo, če je lik tipa *Krog* spet drugo. S preobteževanjem problem enostavno rešimo. Napišemo tri metode

```
public static double Ploscina(Kvadrat lik) { }

public static double Ploscina(Krog lik) { }

public static double Ploscina(Pravokotnik lik) { }
```

Ker za vsak lik znotraj ustrezne metode točno vemo, kakšen je, tudi ni težav z uporabo pravilne formule. Imamo torej tri metode, z enakim imenom, a različnimi podpisi. Seveda bi lahko problem rešili brez preobteževanja, recimo takole:

```
public static double PloscinaKvadrata(Kvadrat lik) { }

public static double PloscinaKroga(Krog lik) { }

public static double PloscinaPravokotnika(Pravokotnik lik) { }
```

A s stališča uporabnika metod je uporaba preobteženih metod enostavnejša. Če ima nek lik *bla*, ki je bodisi tipa *Kvadrat*, tipa *Krog* ali pa tipa *Pravokotnik*, metodo pokliče z *Ploscina(bla)*, ne da bi mu bilo potrebno razmišljati, kakšno je pravilno ime ustrezne metode ravno za ta lik. Prav tako je enostavneje, če dobimo še četrti tip objekta – v "glavno" kodo ni potrebno posegati – naredimo le novo metodo z istim imenom (*Ploscina*) in s parametrom katerega tip je novi objekt. Klic je še vedno *Ploscina(bla)*!

14.9 OBJEKTNE METODE

Svoj pravi pomen dobi sestavljanje razredov takrat, ko objektu pridružimo še metode, torej znanje nekega objekta. Kot že vemo iz uporabe vgrajenih razredov, metode nad nekim objektom kličemo tako, da navedemo ime objekta, piko in ime metode. Tako če želimo izvesti metodo *WriteLine* nad objektom *Console* napišemo

```
Console.WriteLine("To naj se izpiše");
```

Če želimo izvesti metodo *Equals* nad nizom *besedilo*, napišemo

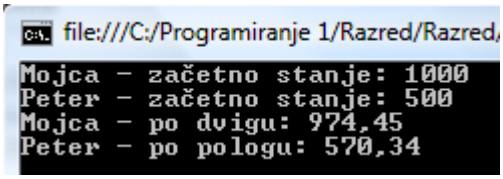
```
besedilo.Equals(primerjava)
```

Primer:

Sestavimo razred *denarnica*, ki bo omogočal naslednje operacije: dvig, vlogo in ugotavljanje stanja. Začetna vrednost se naj postavi s konstruktorjem.

```
public class denarnica
{   // razred ima dve polji: ime osebe in stanje v denarnici
    public string ime;
    public double stanje;

    // konstruktor za nastavljanje začetnih vrednosti
    public denarnica(string ime,double znesek)
    {
       this.ime = ime;
       stanje = znesek;
    }
    public void dvig(double znesek)
    {
       stanje = stanje - znesek;
    }
```

```
    public void polog(double znesek)
    {
       stanje = stanje + znesek;
    }
}

static void Main(string[] args)
{
   // tvorimo dva objekta – uporabimo konstruktor
   denarnica Mojca = new denarnica("Mojca", 1000);
   denarnica Peter = new denarnica("Peter", 500);
   // izpis začetnega stanja v denarnici za oba objekta
   Console.WriteLine("Mojca – začetno stanje: " + Mojca.stanje);
   Console.WriteLine("Peter – začetno stanje: " + Peter.stanje);
   Mojca.dvig(25.55);  // Mojca zapravlja
   Peter.polog(70.34);  // Peter ja zaslužil
   // izpis novega stanja v denarnici za oba objekta
   Console.WriteLine("Mojca – po dvigu: " + Mojca.stanje);
   Console.WriteLine("Peter – po pologu: " + Peter.stanje);
}
```



Slika 30: Objekta razreda *Denarnica*

14.10 ZGLEDI

14.10.1 Gostota prebivalstva

Razred država naj vsebuje polja za osnovne podatke o državi, ter metodo *Gostota* za izračun gostote prebivalstva. Kreirajmo objekt in pokažimo uporabo njegove metode *Gostota*.

```
1:  class Drzava
2:  {
3:        public string ime;
4:        public int povrsina;
5:        public int prebivalci;
6:        public double Gostota() // javna metoda za izračun gostote prebivalstva
7:        {
8:            return prebivalci/povrsina;  // vrnemo celo število!
9:        }
10: }
11: static void Main(string[] args)
12: {
```

```
13:     Drzava d1=new Drzava();
14:     d1.ime = "Slovenija";
15:     d1.povrsina = 20000;
16:     d1.prebivalci = 1980000;
17:     // pokličimo metodo gostota objekta d1
18:     Console.WriteLine("Gostota prebivalstva v " + d1.ime + " je " + d1.Gostota()
19:                                     + " preb/km2");
20: }
```

Razlaga. Najprej ustvarimo razred z imenom *Drzava* (vrstice 1−10). V glavnem programu kreiramo nov primerek (objekt) z imenom *d1* razreda *Drzava* (vrstica 13). Objektu določimo vrednosti (vrstice 14-18) in pokličemo njegovo metodo *Gostota*.

14.10.2 Prodajalec

Prodajalcu napišimo program, ki mu bo pomagal pri vodenju evidence o mesečni in letni prodaji.

```
class Prodajalec
{
   public double[] zneski; // zasebna tabela ki hrani mesečne zneske prodaje
   public Prodajalec()  // konstruktor ki inicializira tabelo prodaje
   {
      zneski = new double[12];
   }
    // metoda za izpis letne prodaje
   public void IzpisiLetnoProdajo()
   {
      Console.WriteLine("Skupna letna prodaja je : " + SkupnaLetnaProdaja() + " EUR");
   }
    // metoda za izračun skupne letne prodaje
   public double SkupnaLetnaProdaja()
   {
      double vsota = 0.0;
      for (int i = 0; i < 12; i++)
         vsota += zneski[i];
      return vsota;
   }
}

static void Main(string[] args)
{
   //  tvorba objekta p tipa Prodajalec, obenem se izvede tudi konstruktor
   Prodajalec p = new Prodajalec();
    //  zanka za vnos prodaje po mesecih
   for (int i = 1; i <= 12; i++)
   {
      Console.Write("Vnesi znesek prodaje za mesec " + i + ": ");
```

```
        p.zneski[i - 1] = Convert.ToDouble(Console.ReadLine());
    }
    p.IzpisiLetnoProdajo(); // objekt p pozna metodo za izpis letne prodaje
}
```

V zgornjem primeru smo v razredu napovedali tabelo *zneski*. Za inicializacijo te tabele smo napisali konstruktor, seveda pa bi lahko tabelo inicializirali že pri deklaraciji s stavkom:

```
public double[] zneski = new double[12]; // vsi elementi tabele dobijo vrednost 0
```

14.11 TABELE OBJEKTOV

Rekli smo že, da smo s tem, ko smo sestavili nov razred, sestavili dejansko nov tip podatkov. Ta je "enakovreden" v C# in njegovim knjižnicam vgrajenim tipom. Torej lahko naredimo tudi tabelo podatkov, kjer so ti podatki objekti.

Denimo, da smo sestavili razred *Zajec*

```
public class Zajec
{
    public string serijska;
    public bool spol;
    public double masa;
}
```

Zaradi enostavnosti ga ne bomo opremili s konstruktorji, saj to nič ne spremeni naše razlage.

Sedaj pišemo program, v katerem bomo potrebovali 100 objektov tipa *Zajec* (denimo, da pišemo program za upravljanje farme zajcev). Ker bomo z vsemi zajci (z vsemi objekti tipa *Zajec*) počeli enake operacije, je smiselno, da uporabimo tabelo.

```
Zajec[] tabZajci;
```

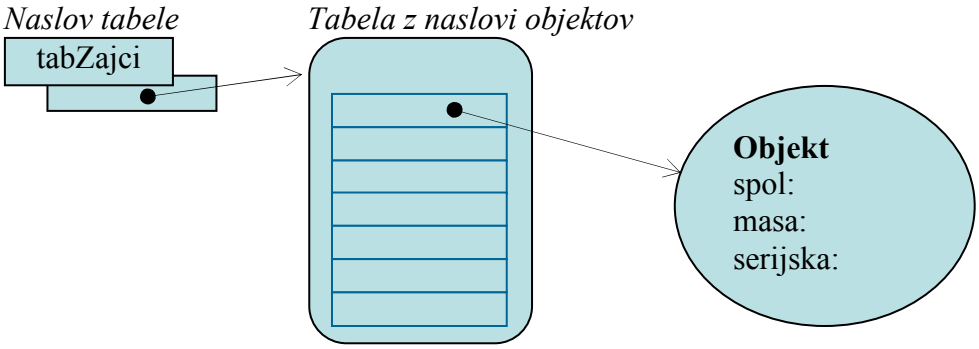
Spemenljivka *tabZajci* nam označuje torej tabelo, v kateri hranimo zajce. A bodimo tokrat še zadnjič pri izražanju zelo natančni. Natančno rečeno nam spremenljivka *tabZajci* označuje **naslov**, kjer bomo ustvarili tabelo, v kateri bomo hranili naslove objektov tipa *Zajec*. Ko torej napišemo

```
tabZajci = new Zajec[250];
```

smo s tem ustvarili tabelo velikosti 250. Kje ta tabela je, smo shranili v spremenljivko *tabZajci*. V tej tabeli lahko hranimo podatek o tem, kje je objekt tipa *Zajec*. V tem trenutku še nimamo nobenega objekta tipa *Zajec*, le prostor za njihove naslove. Šele ko napišemo

```
tabZajci[0] = new Zajec();
```

smo s tem ustvarili novega zajca (nov objekt tipa *Zajec*) in podatke o tem, kje ta novi objekt je (naslov), shranili v spremenljivko *tabZajci[0]*.



Slika 31: Tabela objektov

Seveda pa bomo še vedno govorili, da imamo zajca shranjenega v spremenljivki *tabZ[0]*.

```
public class Zajec
{
  public string serijska;
  public bool spol;
  public double masa;
}

public static void Main(string[] ar)
{
 Zajec[] zajci;
 zajci = new Zajec[250];  //  na farmi imamo do 250 zajecv
 int i = 0;
 int kolikoZajcev = 10;  //  trenutno imamo 10 zajcev
 while (i < kolikoZajcev)
 {
   zajci[i] = new Zajec();  //  "rodil" se je nov zajec
   zajci[i].serijska = "1238-12-“ + i;
   zajci[i].spol = false;
   zajci[i].masa = 0.12;
   i++;
 }
 ...
}
```

14.11.1 Zgoščenka

Pesem na zgoščenki je predstavljena z objektom razreda *Pesem*:

```
public class Pesem
{
   public string naslov;
   public int minute;
   public int sekunde;
```

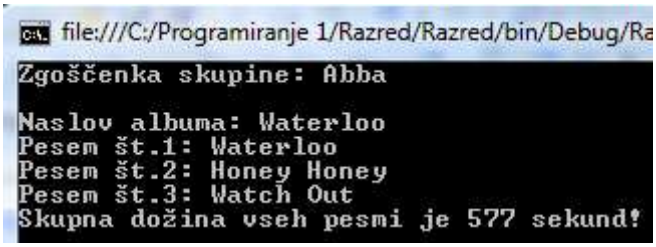
```
   public Pesem(string nasl, int min, int sek)
   {
      naslov = nasl; minute = min; sekunde = sek;
   }
}
```

Objekt *new Pesem("Echoes",15,24)* predstavlja pesem "Echoes", ki traja 15 minut in 24 sekund. Sestavimo razred *Zgoscenka*, ki vsebuje naslov zgoščenke, ime izvajalca in tabelo pesmi na zgoščenki. Razredu *Zgoscenka* bomo dodali še objektno metodo *Dolzina()*, ki vrne skupno dolžino vseh pesmi na zgoščenki, izraženo v sekundah.

```
public class Zgoscenka
{
   public string avtor;
   public string naslov;
   public Pesem[] seznamPesmi;  // Napoved tabele pesmi
    // s konstruktorjem določimo avtorja, naslov in vse pesmi
   public Zgoscenka(string avtor, string naslov,Pesem[]seznamPesmi)
   {
      this.avtor = avtor;
      this.naslov = naslov;
      this.seznamPesmi = new Pesem[seznamPesmi.Length];  // inicializacija tabele pesmi
      for (int i=0;i<seznamPesmi.Length;i++)
      {
          // tabelo pesmi določimo s pomočjo parametra (tabele) seznamPesmi
         this.seznamPesmi[i] = new Pesem(seznamPesmi[i].naslov, seznamPesmi[i].minute,
                                         seznamPesmi[i].sekunde);
      }
   }
   public int Dolzina() // metoda za izračun skupne dolžine vseh skladb
   {
      int skupaj = 0;
      for (int i=0;i<seznamPesmi.Length;i++)
         skupaj=skupaj+seznamPesmi[i].minute*60+seznamPesmi[i].sekunde;
      return skupaj;
   }
}

static void Main(string[] args)
{
    Zgoscenka CD = new Zgoscenka("Abba", "Waterloo",
                                 new Pesem[] { new Pesem("Waterloo", 3, 11),
                                 new Pesem("Honey Honey", 3, 4),
                                 new Pesem("Watch Out", 3, 22)});
     // Izpis albuma
    Console.WriteLine("Zgoščenka skupine: "+CD.avtor+"\n\nNaslov albuma: "+CD.naslov);
    for (int i = 0; i < CD.seznamPesmi.Length; i++)  // izpišemo celoten seznam (tabelo) pesmi
```

```
        Console.WriteLine("Pesem št."+(i+1)+": "+CD.seznamPesmi[i].naslov);
    // pokličemo še metodo Dolzina objekta CD, ki nam vrne skupno dolžino vseh skladb
    Console.WriteLine("Skupna dožina vseh pesmi je "+CD.Dolzina()+" sekund!");
}
```

Izpis:



Slika 32: Razred *Zgoscenka*

14.12 DOSTOP DO STANJ OBJEKTA

Možnost, da neposredno dostopamo do stanj/lastnosti objekta **ni najboljši!** Glavni problem je v tem, da na ta način ne moremo izvajati nobene kontrole nad pravilnostjo podatkov o objektu! Tako lahko za število prebivalcev razreda *Drzava* napišemo npr.

```
d1.prebivalci=-400;
```

Ker ne moremo vedeti, ali so podatki pravilni, so vsi postopki (metode) po nepotrebnem bolj zapleteni, oziroma so naši programi bolj podvrženi napakam. Ideja je v tem, da naj objekt sam poskrbi, da bo v pravilnem stanju. Seveda moramo potem tak neposreden dostop do spremenljivk, ki opisujejo objekt, preprečiti.

Dodatna težava pri neposrednem dostopu do spremenljivk, ki opisujejo lastnosti objekta se pojavi, če moramo kasneje spremeniti način predstavitve podatkov o objektu. S tem bi "podrli" pravilno delovanje vseh starih programov, ki bi temeljili na prejšnji predstavitvi.

Torej bomo, kot smo omenjali že v uvodu, objekt res imeli za "črno škatlo", torej njegove notranje zgradbe ne bomo "pokazali javnosti". Zakaj je to dobro? Če malo pomislimo, uporabnika razreda pravzaprav ne zanima, kako so podatki predstavljeni. Če podamo analogijo z realnim svetom: ko med vožnjo avtomobila prestavimo iz tretje v četrto prestavo, nas ne zanima, kako so prestave realizirane. Zanima nas le to, da se stanje avtomobila (prestava) spremeni. Ko kupimo nov avto nam je načeloma vseeno, če ima ta drugačno vrsto menjalnika, z drugačno tehnologijo. Pomembno nam je le, da se način prestavljanja ni spremenil, da še vedno v takih in drugačnih okoliščinah prestavimo iz tretje v četrto prestavo.

S "skrivanjem podrobnosti" omogočimo, da če pride do spremembe tehnologije se za uporabnika stvar ne spremeni. V našem primeru programiranja v C# bo to pomenilo, da če spremenimo razred (popravimo knjižnico), se za uporabnike razreda ne bo nič spremenilo.

Denimo, da so v C# 3.0 spremenili način hranjenja podatkov za določanje naključnih števil v objektih razreda *Random*. Ker dejansko nimamo vpogleda (neposrednega dostopa) v te

spremenljivke, nas kot uporabnike razreda *Random* ta sprememba nič ne prizadene. Programe še vedno pišemo na enak način, kličemo iste metode. Skratka − ni potrebno spreminjati programov, ki razred *Random* uporabljajo. Še posebej je pomembno, da programi, ki so delovali prej, še vedno delujejo (morda malo bolje, hitreje, a bistveno je, da delujejo enako).

Primer:

Napišimo razred *Avto*, s tremi polji (*znamka*, *letnik* in *registrska*). Začetne vrednosti objektov nastavimo s pomočjo konstruktorja. Polje *letnik* naj bo zasebno, zato napišimo metodo, ki bo letnik spremenila le v primeru, da bo le-ta v smiselnih mejah. Napišimo še metodo za izpis podatkov o določenem objektu.

```
public class Avto
{
   public string znamka;
    // polje letnik  je zasebno, zato ga lahko določimo le s konstruktorjem, spremenimo pa le z
    // metodo SpemeniLetnik
   private int letnik;
   public string registrska;

    // konstruktor
   public Avto(string zn, int leto, string stevilka)
   {
      znamka = zn;
      letnik = leto;
      registerska = stevilka;
   }

   public bool SpremeniLetnik(int letnik)
   {
    if ((2000 <= letnik) && (letnik <= 2020))
    {
       this.letnik = letnik;
       return true;  // leto je smiselno, popravimo stanje objekta in vrnemo true
    }
     return false;  //  leto ni smiselno, zato ne spremenimo nič in vrnemo false
   }
    // metoda za izpis podatkov o določenem objektu
   public override string ToString()
   {
      return ("Znamka: " + znamka + ", Letnik: " + letnik + ", Registrska številka: "
              + registrska);
   }
}
static void Main(string[] args)
{
     // novemu objektu nastavimo začetne vrednosti preko konstruktorja
    Avto novAvto = new Avto("Citroen",1999,"KR V1-02E");
    Console.WriteLine(novAvto.ToString()); // v izpisu bo letnik  1999
```

```
    novAvto.SpremeniLetnik(208); // letnik 208 NI dovoljen, objektu se letnik NE spremeni
    Console.WriteLine(novAvto.ToString());  // v izpisu bo letnik ostal 1999
    novAvto.SpremeniLetnik(2008);  // letnik 2008 JE dovoljen, objektu se letnik spremeni
    Console.WriteLine(novAvto.ToString()); // v izpisu bo letnik 2008
    Console.ReadKey();
}
```

V programu opazimo novo neznano besedo – **override**. Uporabiti jo moramo zaradi "**dedovanja**". Več o tem, kaj dedovanje je, kasneje, a povejmo, da smo z dedovanjem avtomatično pridobili metodo *ToString*. To je razlog, da je ta metoda vedno na voljo v vsakem razredu, tudi če je ne napišemo. Če želimo napisati svojo metodo, ki se imenuje enako kot podedovana metoda, moramo napisati besedico *override*. S tem "povozimo" obstoječo metodo.

Seveda bi lahko napisali tudi neko drugo metodo, na primer *Opis*, ki bi prav tako vrnila niz s smiselnim opisom objekta.

Uporabnikom lahko torej s pomočjo zasebnih polj (*private*) preprečimo, da "kukajo" v zgradbo objektov, ali pa da jim določajo nesmiselne vrednosti. Če bodo želeli dostopati do lastnosti (podatkov) objekta (zvedeti, kakšni podatki se hranijo v objektu) ali pa te podatke spremeniti, bodo morali uporabljati metode. In v teh metodah bo sestavljavec razreda lahko poskrbel, da se s podatki "ne bo kaj čudnega počelo". Potrebujemo torej:

- Metode za dostop do stanj (za dostop do podatkov v objektu)
- Metode za nastavljanje stanj (za spreminjanje podatkov o objektu)

Prvim pogosto rečemo tudi "get" metode (ker se v razredih, ki jih sestavljajo angleško usmerjeni pisci, te metode pogosto pričnejo z *get* (Daj)). Druge pa so t.i. "set" metode (set / nastavi)[3].

Za ponovitev si oglejmo sklop prosojnic, pripravljenih v okviru gradiv projekta Projekt UP - Kako poučevati začetni tečaj programskega jezika, sklop intekativnih gradiv. Dostopne so na naslovu http://up.fmf.uni-lj.si/index.html#h2_25. Nato rešimo še nekaj problemov. Primerni bodo tisti na wikiju Wiki Csharp, dostopnem na naslovu http://penelope.fmf.uni-lj.si/C_sharp/.

[3] Mimogrede, če bi šli v podrobnosti jezika C#, bi videli, da lahko določene komponente proglasimo za lastnosti (*Property*), ki zahtevajo, da jim napišemo metodi z imenom *get* in *set* in potem lahko uporabljamo notacijo *imeObjekta.imeLastnosti*. Na zunaj je videti, kot bi imeli javne spremenljivke. V resnici pa je to le "zamaskiran" klic *get* oziroma *set* metode. Mi bomo lastnosti spustili in bomo pisali "svoje" *get* in *set* metode.

14.13 ZGLED

14.13.1 Razred Tocka

Kreirajmo razred *Tocka*, z dvema zasebnima poljema, koordinatama *x* in *y*. Napišimo tudi dva konstruktorja: privzetega in še enega za nastavljanje vrednosti koordinat. Ker sta koordinati zasebni (nimamo neposrednega dostopa), napišimo še metodi za spreminjanje vrednosti vsake od koordinat. Dodali bomo še metodo za izračun razdalje med dvema točkama.

```
class Tocka
{
   private int x, y;  // koordinati točke sta zasebni polji
   public Tocka()  // osnovni konstruktor
   {
      x = 0;
      y = 0;
   }
   public Tocka(int x, int y) // preobteženi konstruktor
   {
     this.x = x;
     this.y = y;
   }
   public void SpremeniX(int x) // metoda za spreminjanje koordinate x
   {
      this.x = x;
   }
   public void SpremeniY(int y) // metoda za spreminjanje koordinate y
   {
      this.y = y;
   }
   public double RazdaljaOd(Tocka druga) // metoda za izračun razdalje med dvema točkama
   {
      int xRazd = x - druga.x;
      int yRazd = y - druga.y;
      return Math.Sqrt(Math.Pow(xRazd,2) + Math.Pow(yRazd,2));
   }

}
static void Main(string[] args)
{
   Tocka tA = new Tocka();   // Klic konstruktorja brez parametrov
   Tocka tB = new Tocka(6, 8);  //Klic konstruktorja z dvema parametroma
   double razdalja = tA.RazdaljaOd(tB);//klic metode za izračun razdalje A do B
   Console.WriteLine("Razdalja točke A od točke B je enaka " + razdalja + " enot!");
   tB.SpremeniX(4);  // točki B spremenimo prvo koordinato
   tB.SpremeniY(3);  // točki B spremenimo drugo koordinato
   Console.WriteLine("Razdalja točke A do točke B je  " + tA.RazdaljaOd(tB) + " enot!");
   Console.ReadKey();
}
```

14.14 STATIČNE METODE

Za lažje razumevanje pojma **statična metoda**, si oglejmo metodo *Sqrt* razreda *Math*. Če pomislimo na to, kako smo jo v vseh dosedanjih primerih uporabili (poklicali), potem je v teh klicih nekaj čudnega. Metodo *Sqrt* smo namreč vselej poklicali tako, da smo pred njo navedli ime razreda (*Math.Sqrt*) in ne tako, da bi najprej naredili nov objekt tipa *Math*, pa potem nad njim poklicali metodo *Sqrt*. Kako je to možno?

Pogosto se bomo srečali s primeri, ko metode ne bodo pripadale objektom (instancam) nekega razreda. To so uporabne metode, ki so tako pomembne, da so neodvisne od kateregakoli objekta. Metoda *Sqrt* je tipičen primer take metode. Če bila metoda *Sqrt* običajna metoda objekta izpeljanega iz nekega razreda, potem bi za njeno uporabo morali najprej kreirati nov objekt tipa *Math*, npr. takole:

```
Math m = new Math();

double d = m.Sqrt(42.24);
```

Tak način uporabe metode pa bi bil neroden. Vrednost, ki jo v tem primeru želimo izračunati in uporabiti, je namreč neodvisna od objekta. Podobno je pri ostalih metodah tega razreda (npr. *Sin*, *Cos*, *Tan*, *Log*, …). Razred *Math* prav tako vsebuje polje *PI* (iracionalno število Pi), za katerega uporabo bi potemtakem prav tako potrebovali nov objekt. Rešitev je v t.i. **statičnih poljih oz.metodah**.

V C# morajo biti vse metode deklarirane znotraj razreda. Kadar pa je metoda ali pa polje deklarirano kot **statično** (**static**), lahko tako metodo ali pa polje uporabimo tako, da pred imenom polja oz. metode navedemo ime razreda. Metoda *Sqrt* (in seveda tudi druge metode razreda *Math*) je tako znotraj razreda *Math* deklarirana kot statična, nekako takole:

```
class Math

{
   public static double Sqrt(double d)
   {
       . . .
   }
}
```

Zapomnimo pa si, da statično metodo ne kličemo tako kot objektno. Kadar definiramo statično metodo, le-ta **nima** dostopa do kateregakoli polja definiranega za ta razred. Uporablja lahko le polja, ki so označena kot **static** (statična polja). Poleg tega, lahko statična metoda kliče le tiste metode razreda, ki so prav tako označene kot statične metode. Ne-statične metode lahko, kot vemo že od prej, uporabimo le tako, da najprej kreiramo nov objekt.

Primer:

Napišimo razred *Bla* in v njem statično metodo, katere naloga je le ta, da vrne *string*, v katerem so zapisane osnovni podatki o tem razredu

```
class Bla
```

```
{
  public static string Navodila() // Statična metoda
  {
    string stavek="Poklicali ste statično metodo razreda Bla!";
    return stavek;
  }
}
static void Main(string[] args)
{
  // Zato, da pokličemo statično metodo oz. statično polje NE POTREBUJEMO OBJEKTA!!!
  // Primer klica npr. v glavnem programu
  Console.WriteLine(Bla.Navodila()); // Klic statične metode
}
```

14.15 STATIČNA POLJA

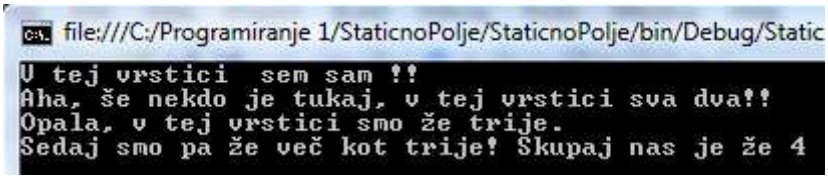
Tako kot obstajajo statične metode, obstajajo tudi statična polja. Včasih je npr. je potrebno, da imajo vsi objekti določenega razreda dostop do istega polja. To lahko dosežemo le tako, da tako polje deklariramo kot statično. Za dostop do statičnega polja ne potrebujemo objektov, saj do takega polja dostopamo preko imena razreda.

Primer:

```
class Nekaj
{
   static int stevilo = 0;   // Statično polje
   public Nekaj()         //  Konstruktor
   {
       // ko se izvede konstruktor (ob kreiranju novega objekta), se statično polje poveča
       // polje stevilo torej šteje živeče objekte
      stevilo++;
   }
   public void Pozdrav()
   {
      if (stevilo==1)
        Console.WriteLine("V tej vrstici  sem sam !!");
      else  if (stevilo==2)
        Console.WriteLine("Aha, še nekdo je tukaj, v tej vrstici sva dva!!");
      else  if (stevilo==3)
        Console.WriteLine("Opala, v tej vrstici smo že trije.");
      else
         Console.WriteLine("Sedaj smo pa že več kot trije! Skupaj nas je že " + stevilo);
   }
}

static void Main(string[] args)
{
   Nekaj a = new Nekaj();  // konstruktor za a (prvi objekt tipa Nekaj)
```

```
    a.Pozdrav();
    Nekaj b = new Nekaj();  // konstruktor za b (drugi objekt tipa Nekaj)
    b.Pozdrav();
    Nekaj c = new Nekaj();  // konstruktor za c (tretji objekt tipa Nekaj)
    c.Pozdrav();
    Nekaj d = new Nekaj();  // konstruktor za d (četrti objekt tipa Nekaj)
    d.Pozdrav();
    Console.ReadKey();
}
```

Izpis:



Slika 33: Štejemo živeče objekte

14.16 DEDOVANJE (INHERITANCE) – IZPELJANI RAZREDI

Dedovanje (*Inheritance*) je ključni koncept objektno orientiranega programiranja. Smisel in pomen dedovanja je v tem, da iz že zgrajenih razredov skušamo zgraditi bolj kompleksne, ki bodo znali narediti nekaj uporabnega. Dedovanje je torej orodje, s katerim se izognemo ponavljanju pri definiranju različnih razredov, ki pa imajo več ali manj značilnosti skupnih. Opredeljuje torej odnos med posameznimi razredi.

Vzemimo pojem sesalec iz biologije. Kot primer za sesalce vzemimo npr. konje in kite. Tako konji kot kiti počnejo vse kar počnejo sesalci nasploh (dihajo zrak, skotijo žive mladiče, …), a prav tako pa imajo nekatere svoje značilnosti (konji imajo npr. štiri noge, ki jih kiti nimajo, imajo kopita, …, obratno pa imajo npr. kiti plavuti, ki pa jih konji nimajo, ..). V jeziku C# bi lahko za ta primer modelirali dva razreda: prvega bi poimenovali *Sesalec*, in drugega *Konj*, in obenem deklarirali, da je *Konj* podeduje (inherits) lastnosti *Sesalca*. Na ta način bi med sesalci in konjem vzpostavili povezavo v tem smislu, da so vsi konji sesalci (obratno pa seveda ne velja!). Podobno lahko deklariramo razred z imenom *Kit*, ki je prav tako podedovan iz razreda *Sesalec*. Lastnosti, kot so npr, *kopita* ali pa *plavuti* pa lahko dodatno postavimo v razred *Konj* oz. razred *Kit*.

14.17 BAZIČNI RAZREDI IN IZPELJANI RAZREDI

Sintaksa, ki jo uporabimo za deklaracijo, s katero želimo povedati, da razred podeduje nek drug razred, je takale:

```
class IzpeljaniRazred : BazičniRazred
{
   . . .
}
```

Izpeljani razred deduje od **bazičnega** razreda. Razred v C# lahko deduje največ en razred in ni možno dedovanje dveh ali več razredov. Seveda pa je lahko razred, ki podeduje nek bazični razred, zopet podedovan v še bolj kompleksen razred.

Primer:

Spoznali smo že razred *Tocka*, s katerim smo predstavili točko v dvodimenzionalnem koordinatnem sistemu

```
class Tocka  // bazični razred
{
  // lastnosti (polja razreda Točka)
   private  int x;
   private  int y;
   public Tocka(int x, int y)   // konstruktor
   {
       // telo konstruktorja
   }
    // telo razreda Tocka
}
```

Sedaj lahko definiramo razred za tridimenzionalno točko z imenom *Tocka3D*, s katerim bomo lahko delali objekte, ki bodo predstavljali točke v tridimenzionalnem koordinatnem sistemu in po potrebi dodamo še dodatne metode:

```
class Tocka3D : Tocka   // razred Tocka3D podeduje razred Tocka
{
   private  int z;   // dodatna lastnosti (polje razreda Točka3D) poleg polj x in y
 // telo razreda Tocka3D – tukaj zapišemo še dodatne metode tega razreda!
}
```

14.18 KLIC KONSTRUKTORJA BAZIČNEGA RAZREDA

Vsi razredi imajo vsaj en konstruktor (če ga ne napišemo sami, nam prevajalnik zgenerira privzeti konstruktor). Izpeljani razred avtomatično vsebuje vsa polja bazičnega razreda, a ta polja je potrebno ob kreiranju novega objekta inicializirati. Zaradi tega mora konstruktor v izpeljanem razredu poklicati konstruktor svojega bazičnega razreda. V ta namen se uporablja rezervirana besedica **base**:

```
class Tocka3D : Tocka   //  // razred Tocka3D podeduje razred Tocka
{
   public Toca3D(int z)
         :base(x,y)   // klic bazičnega konstruktorja Tocka(x,y)
   {
      // telo konstruktorja Tocka3D
   }
    // telo razreda Tocka3D
```

```
}
```

Če bazičnega konstruktorja v izpeljanem razredu ne kličemo eksplicitno (če vrstice *:base(x,y)* ne napišemo), bo prevajalnik avtomatično zgeneriral privzeti konstruktor. Ker pa vsi razredi nimajo privzetega konstruktorja (v veliko primerih napišemo lastni konstruktor), moramo v konstruktorju znotraj izpeljanega razreda obvezno najprej klicati bazični konstruktor. Če klic izpustimo (ali ga pozabimo napisati) bo rezultat prevajanja *compile-time error*:

```
class Tocka3D : Tocka   // razred Tocka3D podeduje razred Tocka
{
   public Tocka3D(int z)
    // NAPAKA - POZABILI smo klicati bazični konstruktor razreda Tocka
    {
        // telo konstruktorja Tocka3D
    }
    // telo razreda Tocka3D
}
```

14.19 NOVE METODE

Razredi lahko vsebujejo več ali manj metod in slej ko prej se lahko zgodi, da se pri dedovanju v izpeljanih razredih ime metode ponovi – v izpeljanem razredu torej napišemo metodo, katere ime, število in tipi parametrov se ujemajo z metodo bazičnega razreda. Pri prevajanju bomo zato o tem dobili ustrezno opozorilo - *warning*. Metoda v izpeljanem razredu namreč v tem primeru prekrije metodo bazičnega razreda. Program se bo sicer prevedel in tudi zagnal, a prevajalnik nas bo opozoril s sporočilom, da smo z metodo v izpeljanem razredu prekrili metodo bazičnega razreda. Opozorilo moramo vzeti resno. Če namreč napišemo razred, ki bo podedoval nek bazični razred, bo uporabnik morda pričakoval, da se bo pri klicu neke metode zagnala metoda bazičnega razreda, v resnici pa se bo zagnala metoda izpeljanega razreda. Problem seveda lahko rešimo tako, da metodo v izpeljanem razredu preimenujemo, še boljša rešitev pa je ta, da v izpeljanem razredu eksplicitno povemo, da gre za NOVO metodo – to storimo tako, da v glavi metode, takoj na začetku, napišemo operator *new*.

```
 // Z operatorjem new napovemo, da ima razred Tocka3D SVOJO LASTNO  metodo Izpis
new public void Izpis()
```

14.20 POVZETEK

V poglavju so zapisane le osnove objektov in dedovanja, ter nekaj osnovnih primerov, ki omogočajo vpogled v enega ključnih pojmov objektno orientiranega programiranja - dedovanje. Zahtevnejšim bralcem, predvsem pa tistim, ki bi želeli o razredih vedeti še veliko več, priporočava študij dodatne literature. Tako je v gradivu Jerše G. in Lokar M., Programiranje II, Objektno programiranje, zbrana vrsta dodatnih zgledov in nalog s tega področja. Za še bolj poglobljen prostop k objektno orientiranem programiranju pa naj bralec poseže na primer po gradivu CSharp Tutorial, dostopnem na naslovu

http://www.java2s.com/Tutorial/CSharp/CatalogCSharp.htm ali pa po knjigi Murach J. in Murach M., Murach's C# 2008.

Seveda pa je za to, da bo tak pristop k programiranju postal domač, potrebno napisati kar največ programov. Zbirka nalog, ki je na voljo, je poleg prej omenjenega gradiva še na primer poglavje Objektno programiranje v Lokar M. in Uranič S., Programiranje 1 - zbirka rešenih nalog v C#, spletna zbirka nalog v wikiju http://penelope.fmf.uni-lj.si/diri0607/index.php/Kategorija:Objektno_programiranje in še kje.

15 DATOTEKE

15.1 UVOD

V vseh dosedanjih programih smo podatke brali s standardnega vhoda (tipkovnica) in jih pisali na standardni izhod (zaslon). Ko se je program zaključil, so bili vsi podatki, ki smo jih v program vnesli, ali pa jih je zgeneriral program sam (npr. naključna števila), izgubljeni. Slej ko prej pa se pri delu s programi pojavi potreba po shranjevanju podatkov, saj jih le na ta način lahko pri ponovnem zagonu programa pridobimo nazaj in nadaljujemo z njihovo obdelavo.

Podatke shranjujemo (zapisujemo) v datoteke, iz datotek pa lahko podatke tudi pridobimo (beremo) nazaj − naučiti se moramo torej kako delamo z datotekami.

15.2 KAJ JE DATOTEKA

V Slovarju slovenskega knjižnega jezika (SSKJ) piše : **datoteka** -e ž (e) elektronsko urejena skupina podatkov pri (elektronskem) računalniku, podatkovna zbirka: vnesti nove podatke v datoteko / datoteka s programom.

Datoteka (ang. *file*) je zaporedje podatkov na računalniku, ki so shranjeni na disku ali kakem drugem pomnilniškem mediju (npr. CD-ROM, disketa, ključek). V datotekah hranimo različne podatke: besedila, preglednice, programe, ipd.

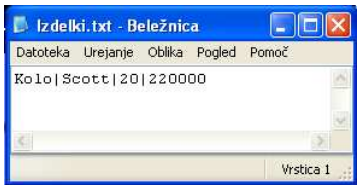
Glede na tip zapisa jih delimo na:

- **tekstovne** datoteke in
- **binarne** datoteke

Tekstovne datoteke vsebujejo nize znakov oziroma zlogov, ki so ljudem berljivi. Poleg teh znakov so v tekstovnih datotekah zapisani določeni t.i. kontrolni znaki. Najpomembnejša tovrstna znaka sta znaka za konec vrstice (*end of line*) in konec datoteke. Najpomembnejše je, da vemo, da so tekstovne datoteke razdeljene na vrstice. Odpremo jih lahko v vsakem urejevalniku (npr. *WordPad*) ali jih izpišemo na zaslon.

V tekstovnih datotekah so tudi vsi številski podatki shranjeni kot zaporedje znakov (števk in morda decimalne pike). Zato jih moramo, če jih želimo uporabiti v aritmetičnih operacijah, spremeniti v številske podatke. V tekstovnih datotekah so torej vsi podatki shranjeni kot tekstovni znaki. Pogosto so posamezni sklopi znakov (besede, polja, …) med seboj ločeni s posebnimi ločili (npr znakom |, ali pa z znakom vejica ipd.), datoteke pa vsebujejo tudi znake *end of line* (znak za konec vrstice). Tekstovne datoteke imajo torej vrstice.

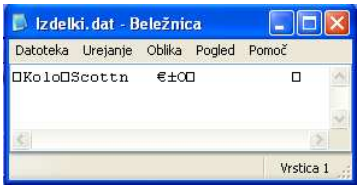
Na sliki je prikazan izgled vsebine tekstovne datoteke odprte z Beležnico (v datoteki je ena sama vrstica).



Slika 34: Tekstovna datoteka odprta z Beležnico

Binarne datoteke vsebujejo podatke zapisane v binarnem zapisu. Zato je vsebina le teh datotek ljudem neberljiva (urejevalniki besedil običajno ne znajo prikazati posebnih znakov, namesto njih prikazujejo "kvadratke").

Tudi podatki v binarnih datotekah lahko vsebujejo znake, tako kot je to v tekstovnih datotekah, a so podatki med seboj ločeni s posebnimi znaki, zaradi česar je vsebina take datoteke, če jo odpremo z nekim urejevalnikom, skoraj neberljiva. Poleg tega binarne datoteke ne vsebujejo vrstic. Na sliki je prikazan izgled vsebine tekstovne datoteke odprte z Beležnico (v datoteki NI vrstic).



Slika 35: Binarna datoteka odprta z beležnico

Ukvarjali se bomo več ali manj le s tekstovnimi datotekami.

15.3 IMENA DATOTEK

Poimenovanje datoteke je odvisno od operacijskega sistema. V tej nalogi se bomo omejili na operacijske sisteme družine Windows. Vsaka datoteka ima ime in je v neki mapi (imeniku). Polno ime datoteke dobimo tako, da zložimo skupaj njeno ime (t.i. kratko ime) in mapo.

Primer:

D:\olimpijada\Peking\atletika.txt

Če povemo z besedami: Ime datoteke je *atletika.txt* na enoti *D*:, v imeniku *Peking*, ki je v podimeniku imenika *olimpijada*.

Imena imenikov so v operacijskem sistemu Windows ločena z znakom \. Kadar želimo, da je znak \ sestavni del niza, moramo napisati kombinacijo \\. Spomnimo se še, da v jeziku C# znak \ v nizu lahko izgubi posebni pomen, kadar pred nizom uporabimo znak @.

15.4 KNJIŽNJICA (IMENSKI PROSTOR)

Za delo z datotekami v jeziku C# so zadolžene metode iz razredov v imenskem prostoru *System.IO*. Zato na začetku vsake datoteke, v kateri je program, ki dela z datotekami, napišemo

```
using System.IO;
```

Navedemo torej uporabo omenjenega imenskega prostora. S tem poenostavimo klice teh metod.

15.5 PODATKOVNI TOKOVI

Kratica *IO* v imenskem prostoru *System.IO* predstavlja vhodne (*input*) in izhodne (*output*) tokove (*streams*). S pomočjo njih lahko opravljamo želene operacije nad datotekami (npr. branje, pisanje). Pisalni ali izhodni tok v jeziku C# predstavimo s spremenljivko tipa *StreamWriter*. Bralni ali vhodni tok pa je tipa *StreamReader*.

V podrobnosti glede tokov se ne bomo spuščali in bomo predstavili zelo poenostavljeno zgodbo. Podatkovne tokove si lahko predstavljamo takole. Lahko si mislimo, da je datoteka v imeniku jezero, reka, ki teče v jezero (ali iz njega), pa podatkovni tok, ki prinaša, oziroma odnaša vodo. Če potrebujemo reko, ki v jezero prinaša vodo, bomo v C# potrebovali spremenljivko tipa *StreamWriter* (pisanje toka), če pa potrebujemo reko, ki vodo (podatke) odnaša, pa *StreamReader* (branje toka).

15.6 TEKSTOVNE DATOTEKE − BRANJE IN PISANJE

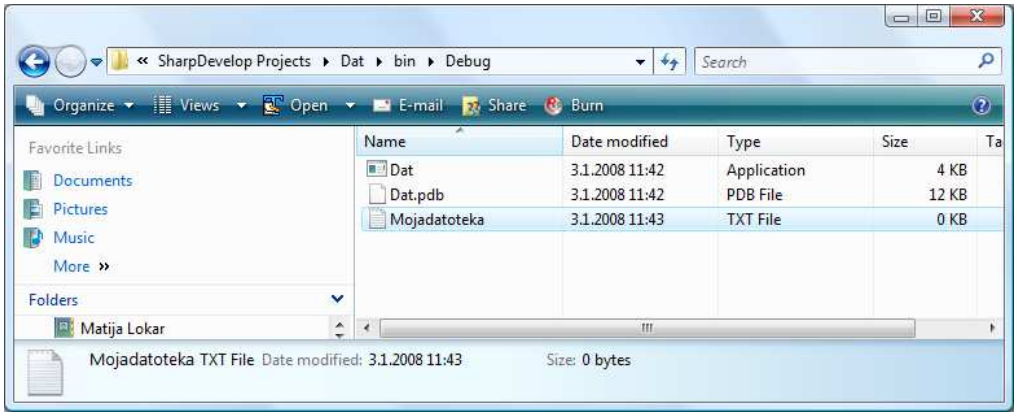
Tekstovne datoteke vsebujejo nize znakov oziroma zlogov, ki so ljudem berljivi. Katere znake vsebujejo, je odvisno od uporabljene kodne tabele.

Poglejmo si najenostavnejši način, kako ustvarimo tekstovno datoteko. "Čarobna" vrstica z ukazom za kreiranje datoteke je

```
File.CreateText("Mojadatoteka.txt");
```

Če pokličemo metodo *File.CreateText("nekNiz")*, ustvarimo datoteko z imenom *nekNiz*. V tem nizu mora seveda biti na pravilen način napisano ime datoteke, kot ga dovoljuje operacijski sistem.

Če poženemo zgornji primer in pokukamo v imenik, kjer ta program je, bomo zagledali (prazno) datoteko.

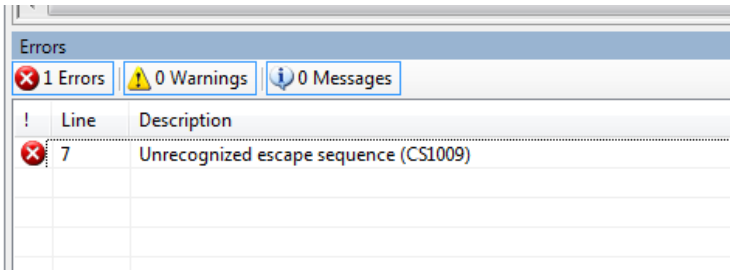


Slika 36: Vsebina mape z novo ustvarjeno tekstovno datoteko *MojaDatoteka.txt*

Če bi želeli datoteko ustvariti v kakšnem drugem imeniku, moramo seveda napisati polno ime datoteke, npr. takole

```
File.CreateText("C:\temp\Mojadatoteka.txt");
```

ne gre, saj se prevajalnik hitro oglasi z



Slika 37: Prevajalnik pri kreiranju nove datoteke javi napako

Spomnimo se, da ima znotraj niza znak '\' poseben pomen – napoveduje, da v nizu prihaja nek poseben znak. Če želimo dobiti \, moramo napisati dva. Torej

```
File.CreateText("C:\\temp\\Mojadatoteka.txt");
```

Sedaj ni težav in v želenem imeniku dobimo ustrezno datoteko. Ker pa je tovrstni zapis s podvojenimi \ malo nepregleden, lahko pred nizom uporabimo znak @. S tem povemo, da se morajo vsi znaki v nizu jemati dobesedno. Zgornji zgled bi torej lahko napisali kot

```
File.CreateText(@"C:\temp\Mojadatoteka.txt");
```

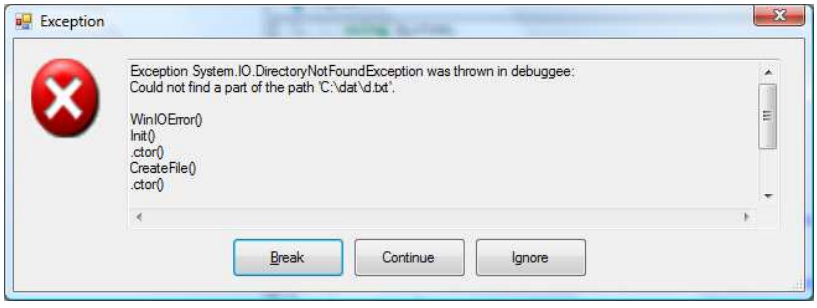
Vendar moramo biti pri uporabi tega ukaza previdni. Namreč, če datoteka že obstaja, jo s tem ukazom "povozimo". Izgubimo torej staro vsebino in ustvarimo novo, prazno datoteko.

Pred ustvarjanjem nove datoteke je zato smiselno, da prej preverimo, če datoteka že obstaja. Ustrezna metoda je

```
File.Exists(ime)
```

ki vrne *true*, če datoteka obstaja in *false* sicer.

Če imenika, kjer želimo narediti datoteko ni, se program "sesuje"



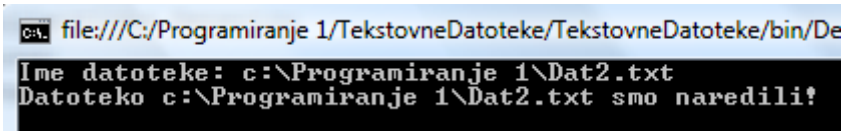
Slika 38: Če imenik ne obstaja, se program sesuje

15.7 ZGLEDI

15.7.1 Ustvari datoteko

Naredimo zgled, ki uporabnika vpraša po imenu datoteke. Če datoteke še ni, jo naredi, drugače pa izpiše ustrezno obvestilo.

```
public static void Main(string[] args)
 {
       Console.Write("Ime datoteke: ");
       string ime = Console.ReadLine();
       if (File.Exists(ime))
       {
            Console.WriteLine("Datoteka " + ime + " že obstaja!");
       }
       else
       {
            File.CreateText(ime);
            Console.WriteLine("Datoteko " + ime + " smo naredili!");
        }
}
```



Slika 39: Ustvarili smo novo datoteko

Iz slike vidimo še eno stvar. Ko tipkamo ime datoteke, \ navajamo običajno (enkratno), saj C# ve, da vnašamo "običajne" znake.

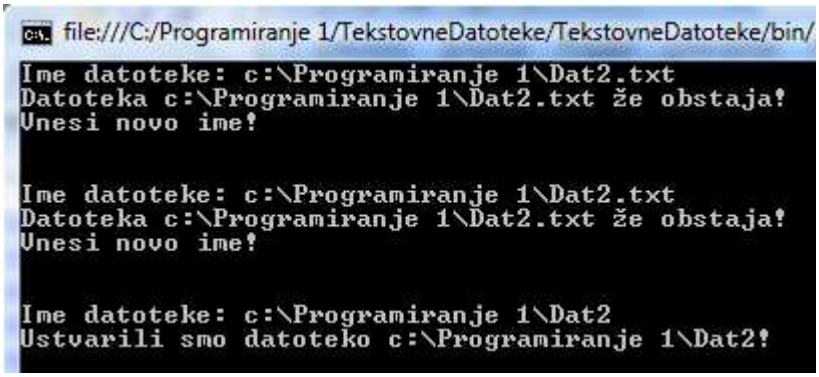
15.7.2 Zagotovo ustvari datoteko

Denimo, da pišemo program, s katerim bomo nadzirali varnostne kamere v našem podjetju. Naš program mora ob vsakem zagonu začeti na novo pisati ustrezno dnevniško datoteko. Ker gre za izjemno skrivno operacijo, ni mogoče, da bi se datoteke ustvarjale kar v nekem fiksnem imeniku. Zato mora ime datoteke vnesti kar operater. Seveda morajo prejšnje datoteke ostati nespremenjene. Datotek se bo hitro nabralo zelo veliko, zato bo operater izgubil pregled nad imeni, ki jih je ustvaril. Zato mu pomagaj in napiši metodo, ki bo uporabnika tako dolgo spraševala po imenu datoteke, dokler ne bo napisal imena, ki zagotovo ni obstoječa datoteka. To ime naj metoda vrne kot rezultat.

Ideja: Metoda se bo začela z *public static string NovoIme()*. V metodi bomo prebrali ime datoteke in to potem ponavljali toliko časa, dokler metoda *File.Exist* ne bo vrnila *false*.

```
public static string NovoIme()
{
   Console.Write("Ime datoteke: ");
   string ime = Console.ReadLine();
```

```
    while (File.Exists(ime))
    {  //  če datoteka že obstaja
       Console.WriteLine("Datoteka " + ime + " že obstaja!");
       Console.WriteLine("Vnesi novo ime!\n\n");
       Console.Write("Ime datoteke: ");
       ime = Console.ReadLine();
    }
    return ime;
}
public static void Main(string[] args)
{
    string imeDatoteke = NovoIme();
    File.CreateText(imeDatoteke);
    Console.WriteLine("Ustvarili smo datoteko " + imeDatoteke + "!");
    Console.ReadLine();
}
```



Slika 40: Ustvarjanje nove datoteke

15.8 PISANJE NA DATOTEKO

Datoteko torej znamo ustvariti. A kako bi nanjo kaj zapisali? Kot vemo, lahko izpisujemo z metodo *WriteLine* (in z *Write*). A zaenkrat znamo pisati le na izhodno konzolo z

```
Console.WriteLine("nekaj izpišimo");
```

Zgodba pri pisanju na datoteko je povsem podobna. Metoda *File.CreateText()*, ko ustvari datoteko, namreč vrne oznako tako imenovanega podatkovnega toka. To oznako shranimo v spremenljivko tipa *StreamWriter*. In če sedaj napišemo

```
oznaka.WriteLine("Mi znamo pisati v datoteko");
```

smo sedaj napisali omenjeno besedilo na datoteko, ki smo jo prej odprli in povezali z oznako.

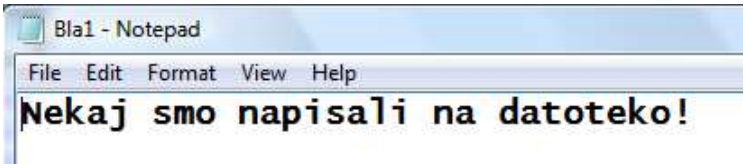
Morebitne nejasnosti bo razjasnil zgled (v zgledu kličemo metodo *NovoIme* iz prejšnjega primera):

```
public static void Main(string[] args)
{
      string imeDatoteke = NovoIme();
      StreamWriter oznaka;
      oznaka = File.CreateText(imeDatoteke);
      oznaka.WriteLine("Nekaj smo napisali na datoteko!");
}
```

Poženimo program in pokukajmo v imenik, kjer je nova datoteka. A glej! Datoteka sicer je tam, a je prazna. Zakaj? Vedno, ko nekaj počnemo z datotekami, jih je potrebno na koncu zapreti. To storimo z metodo *Close*. Če torej program spremenimo v

```
public static void Main(string[] args)
{
   string imeDatoteke = NovoIme();
   StreamWriter oznaka;
   oznaka = File.CreateText(imeDatoteke);
   oznaka.WriteLine("Nekaj smo napisali na datoteko!");
   oznaka.Close();
}
```

in si sedaj ogledamo datoteko, vidimo, da ni več dolga 0 zlogov. Če jo odpremo v najkoristnejšem programu Beležnici:



Slika 41: Vsebina datoteke *Bla1*

bomo na njej našli omenjeni stavek.

Če podatkovnega toka po pisanju podatkov ne zapremo, je možno, da v nastali datoteki manjka del vsebine, oziroma vsebine sploh ni. Namreč, da napisan program pospeši operacije z diskom, se vsebina ne zapiše takoj v datoteko na disku, ampak v vmesni polnilnik. Šele ko je ta poln, se celotna vsebina medpomnilnika zapiše na datoteko. Metoda *Close* v C# poskrbi, da je medpomnilnik izpraznjen tudi, če še ni povsem poln.

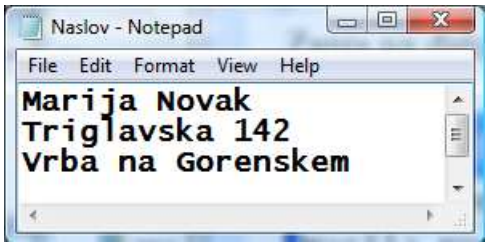
15.9 ZGLEDI

15.9.1 Osebni podatki

Na enoti C v korenskem imeniku ustvarimo mapo *Tekstovna*. V tej pod mapi ustvarimo tekstovno datoteko *Naslov.txt* in vanjo zapišimo svoj naslov. To naredimo le, če datoteke še ni.

```
1:      public static void Main(string[] args)
2:      {
3:          // Pot in ime
4:          string ime = @"C:\Tekstovna\Naslov.txt";

5:          // Preverimo obstoj datoteke
6:          if (File.Exists(ime))
7:          {
8:              Console.WriteLine("Datoteka že obstaja.");
9:              return;
10:         }
11:         // Ustvarimo novo datoteko
12:         StreamWriter dat = File.CreateText(ime);
13:         // Zapišemo osebne podatke
14:         dat.WriteLine("Marija Novak");
15:         dat.WriteLine("Triglavska 142");
16:         dat.WriteLine("Vrba na Gorenskem");
17:         // Zapremo datoteko za pisanje
18:         dat.Close();
19:     }
```

Zapis na datoteki *Naslov.txt*:



Slika 42: Vsebina datoteke *Naslov.txt*

Razlaga. Najprej določimo niz, ki predstavlja polno ime datoteke (vrstica 6). Če bi za niz določili le kratko ime, bi se datoteka ustvarila v mapi, kjer bi izvedli naš program. V vrstici 6 preverimo obstoj datoteke. Če datoteka obstaja, izpišemo obvestilo (vrstica 8) in končamo izvajanje programa (vrstica 9). Nato v vrstici 12 odpremo datoteko za pisanje. S klicem metode *dat.WriteLine()* podatke zapišemo na datoteko (vrstice 14 - 16). Po končanem zapisu datoteko zapremo (vrstica 18). Če tega ne bi storili, datoteka ne bi imela vsebine.

15.9.2 Zapis 100 vrstic

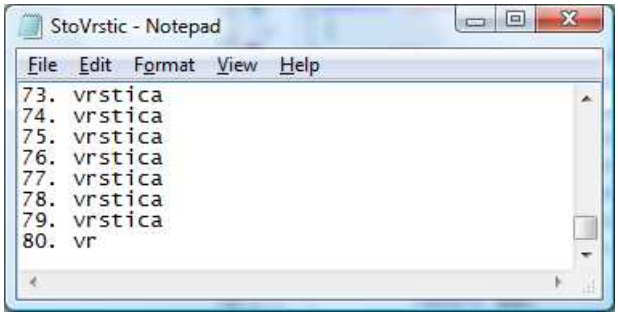
Napišimo sedaj program, ki bo v datoteko *StoVrstic.txt* zapisal 100 vrstic, denimo takih: 1. vrstica, 2. vrstica, ..., 100. vrstica.

Naloga je enostavna:

Ustvarimo datoteko *StoVrstic.txt* in jo povežemo z pisalnim podatkovnim tokom. V zanki izpišemo števec in besedilo ". vrstica". Nato ... . Nič! To je vse.

```
public static void Main(string[] args)
{
   StreamWriter oznaka;
   oznaka = File.CreateText(@"c:\temp\StoVrstic.txt");
   for (int i = 1; i <= 100; i++)
   {
      oznaka.WriteLine(i + ". vrstica");
   }
}
```

Poglejmo datoteko. Na začetku je vse lepo in prav, a kaj pa je s koncem:

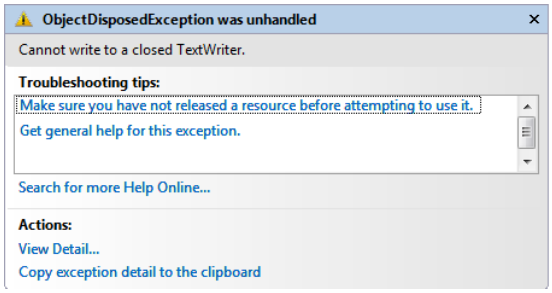


Slika 43: Vsebina datoteke *StoVrstic*

20 vrstic manjka! Razlog je v tem, da smo pozabili na *Close*. Če torej dodamo vrstico 7

```
public static void Main(string[] args)
{
   StreamWriter oznaka;
   oznaka = File.CreateText(@"c:\temp\StoVrstic.txt");
   for (int i = 1; i <= 100; i++)
   {
      oznaka.WriteLine(i + ". vrstica");
      oznaka.Close();
   }
}
```

bo datoteka *StoVrstic.txt* taka, kot pričakujemo. Opa, ne le, da datoteka nima pričakovane vsebine, celo program se je "sesul"



Slika 44: Napaka pri pisanju v zaprto datoteko

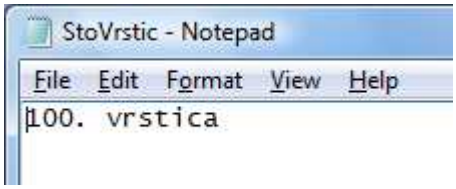
Razlog nam lepo pojasni obvestilno okno. Piše namreč:

Cannot write to a closed TextWriter.

Datoteko smo torej zaprli na napačnem mestu, v zanki, ko bi morala biti še odprta. Popravimo program

```
public static void Main(string[] args)
{
   StreamWriter oznaka;
   for (int i = 1; i <= 100; i++)
   {
      oznaka = File.CreateText(@"c:\temp\StoVrstic.txt");
      oznaka.WriteLine(i + ". vrstica");
      oznaka.Close();
   }
}
```

Tako. Program sedaj deluje lepo in prav. A ko odpremo datoteko *StoVrstic.txt* nas čaka presenečenje:



Slika 45: V datoteki je le zadnja vrstica

V datoteki je le zadnja vrstica. Seveda – vsako odpiranje datoteke pobriše staro vsebino. In zato smo sproti pobrisali to, kar smo na prejšnjem koraku napisali. No, ko program spremenimo v

```
public static void Main(string[] args)
{
   StreamWriter oznaka;
   oznaka = File.CreateText(@"c:\temp\StoVrstic.txt");
   for (int i = 1; i <= 100; i++)
   {
         oznaka.WriteLine(i + ". vrstica");
   }
   oznaka.Close();
}
```

je datoteka končno taka, kot smo pričakovali.

Bralcem se za "neumne" napake seveda opravičujeva. A dejstvo je, da so prav take napake zelo pogoste v začetniških programih. Zato si jih je dobro ogledati in razumeti, zakaj se zadeva obnaša na tak način. Tako se bomo lažje izognili napakam v naših programih.

Seveda na tekstovno datoteko lahko pišemo tudi s pomočjo metode *Write*. V ta namen si oglejmo še en zgled.

15.9.3 Datoteka naključnih števil

Sestavimo metodo, ki ustvari datoteko *NakStevX.dat* z n naključnimi števili med *a* in *b*. *X* naj bo prvo "prosto" naravno število. Če torej obstajajo datoteke *NakStev1.dat*, *NakStev2.dat* in *NakStev3.dat*, naj metoda ustvari datoteko *NakStev4.dat*. Števila naj bodo levo poravnana v vsaki vrsti, torej npr. kot:

12 134 23 22 78

167 12 1 134 45

13 9

Ideja: Naša metoda bo imela 4 parametre:

- *n*: koliko števil bo v datoteki
- *spMeja*, *zgMeja*: meji za nakključna števila
- *kolikoVVrsti*: koliko števil je v vrsti

Rezultat metode bo *void*, saj bo metoda imela le učinek (ustvarjeno datoteko).

```
public static void UstvariDat(int n, int spMeja, int zgMeja, int kolikoVVrsti)
{ . . . }
```

Najprej bomo z zanko, ki bo zaporedoma preverjala, če datoteke z imeni *NakStevX.dat* obstajajo., poskrbeli, da bomo ustvarili primerno datoteko:

```
string osnovaImena = @"C:\temp\NakStev";
int katera = 1;
string ime = osnovaImena + katera + ".dat";
while (File.Exists(ime))
{          //  Če datoteka že obstaja
           katera++;
           ime = osnovaImena + katera + ".dat";
}
```

Nato bomo datoteko ustvarili in povezali s tokom za pisanje.

```
StreamWriter oznaka;
oznaka = File.CreateText(ime);
```

Ustvarili bomo še generator naključnih števil, nato pa naredimo zanko, ki se bo izvedla n-krat. V njej ustvarimo naključno število in ga zapišemo na datoteko. S tabulatorjem "\t" poskrbimo za poravnavo. Če smo izpisali že večkratnik *kolikoVVrsti* števil, gremo v novo vrsto

```
int nakStev = genNak.Next(spMeja, zgMeja);
pisiDat.Write("\t " + nakStev);
 //  s tabulatorji poskrbimo za poravnavo
stevec++;  //  zapisali smo še eno število
if (stevec % kolikoVVrsti == 0)
{
   pisiDat.WriteLine();  //  zaključimo vrstico
}
```

Na koncu le še zaključimo vse skupaj, po potrebi gremo še v novo vrsto in zapremo podatkovni tok

```
if (stevec % kolikoVVrsti != 0)
{
    //  če prej nismo ravno končali z vrstico
   pisiDat.WriteLine();  //  zaključimo vrstico
}
pisiDat.Close();
```

Poglejmo sedaj še celotni program

```
public static void UstvariDat(int n, int spMeja, int zgMeja, int kolikoVVrsti)
{
        string osnovaImena = @"C:\temp\NakStev";
        int katera = 1;
        string ime = osnovaImena + katera + ".dat";
        while (File.Exists(ime)) {  //  če datoteka že obstaja
                katera++;
                ime = osnovaImena + katera + ".dat";
        }
         //  našli smo "prosto" ime
        StreamWriter pisiDat;
        pisiDat = File.CreateText(ime);
        Random genNak = new Random();
        int stevec = 0;
        while (stevec < n) {
                int nakStev = genNak.Next(spMeja, zgMeja);
                pisiDat.Write("\t " + nakStev);
                //  s tabulatorji poskrbimo za poravnavo
                stevec++;  //  zapisali smo še eno število
                if (stevec % kolikoVVrsti == 0)
                {
                        pisiDat.WriteLine();  //  zaključimo vrstico
                }
        }
        if (stevec % kolikoVVrsti != 0) {
        //  če prej nismo ravno končali z vrstico
```

```
                pisiDat.WriteLine();  //  zaključimo vrstico
        }
        pisiDat.Close();
}

public static void Main(string[] args)
{
        UstvariDat(32, 1, 1000, 5);
}
```

15.10 BRANJE TEKSTOVNIH DATOTEK

Poglejmo si sedaj, kako bi prebrali tisto, kar smo v prejšnjem razdelku napisali na datoteko.

15.10.1 Branje po vrsticah

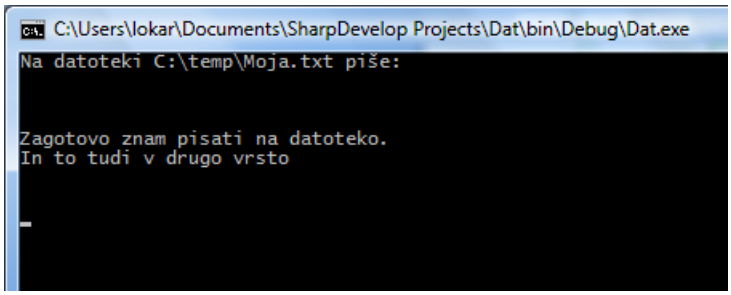
S tekstovnih datotek najlažje beremo tako, da preberemo kar celo vrstico hkrati. Poglejmo si naslednjo metodo

```
1:    public static void IzpisDatoteke(string ime)
2:    {
3:        Console.WriteLine("Na datoteki " + ime + " piše: \n\n\n");
4:        //  odpremo datoteko za
5:        StreamReader izhTok;
6:        izhTok = File.OpenText(ime);
7:        //  preberemo prvo vrstico in jo izpišemo na zaslon
8:        Console.WriteLine(izhTok.ReadLine());
9:        //  preberi z datoteke naslednji dve vrstici in ju izpiši na zaslon
10:       Console.WriteLine(izhTok.ReadLine());
11:       Console.WriteLine(izhTok.ReadLine());
12:       //  preberi preostale vrstice
13:       Console.WriteLine(izhTok.ReadToEnd());
14:       izhTok.Close(); //  zaprimo tok
15:   }
```

Razlaga. V 3. vrstici najprej na zaslon izpišemo obvestilo in spustimo tri prazne vrstice. Nato v vrstici 5 določimo spremenljivko, ki bo označevala izhodni tok. V vrstici 6 z metodo *OpenText* izhodni tok povežemo z datoteko z imenom *ime*. Kot vidimo, podatke beremo z *ReadLine()*. S tem preberemo kompletno vrstico. Obstaja tudi metoda *ReadToEnd()*, ki prebere vse vrstice v datoteki od trenutne vrstice dalje pa do konca. Tudi datoteke s katerih beremo se spodobi zapreti, čeprav pozabljeni *Close* tu ne bo tako problematičen kot pri datotekah na katere pišemo.

15.10.2 Branje datoteke z dvema vrsticama

Z metodo *IzpisDatoteke* preberimo datoteko, ki ima le dve vrstici.

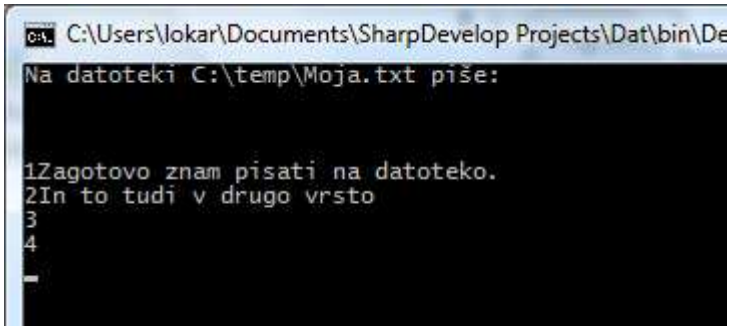


Slika 46: Izpis datoteke

Kot vidimo, smo izpisali štiri vrstice! Kako to? Če si z Beležnico ogledamo datoteko Moja.txt vidimo, da ima le dve vrstici. Od kje potem preostali dve? Če popravimo kodo metode tako, da se spredaj izpiše tudi ustrezna številka

```
// preberemo prvo vrstico in jo izpišemo na zaslon
Console.WriteLine(1 + izhTok.ReadLine());
// preberi z datoteke naslednji dve vrstici in ju izpiši na zaslon
Console.WriteLine(2 + izhTok.ReadLine());
Console.WriteLine(3 + izhTok.ReadLine());
// preberi preostale vrstice
Console.WriteLine(4 + izhTok.ReadToEnd());
```

na zaslon dobimo



Slika 47: Ponovni izpis

Namreč, če z metodo *ReadLine()* beremo potem, ko datoteke ni več (neobstoječe vrstice), metoda vrne kot rezultat neobstoječ niz (vrednost *null*). Če tak neobstoječ niz izpišemo, se izpiše kot prazen niz (""). Zato tretji klic izpisa *Console.WriteLine(3 + izhTok.ReadLine());* izpiše 3 in prazen niz. Prav tako tudi metoda *ReadToEnd()* vrne neobstoječ niz, če vrstic ni več.

15.11 BRANJE PO ZNAKIH

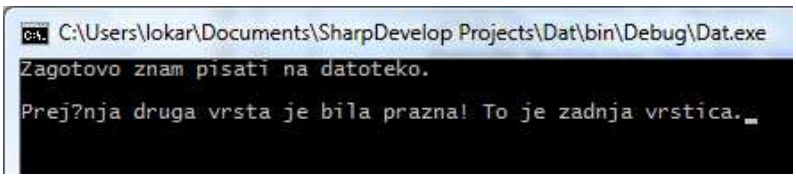
Tekstovne datoteke lahko beremo tudi znak po znak. Poglejmo si kar metodo, ki datoteko znak po znak prepiše na zaslon.

```
1:      public static void IzpisDatotekePoZnakih(string ime)
```

```
2:      {
3:          StreamReader s = File.OpenText(ime);
4:          int beri;
5:          beri = s.Read();
6:          while (beri != -1)  //  konec datoteke
7:          {
8:              Console.Write((char)beri);  //  izpisujemo ustrezne znake
9:              beri = s.Read();
10:         }
11:         s.Close();
12:     }
```

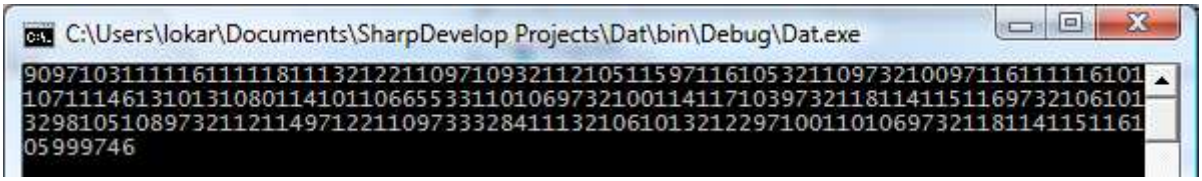
Ko beremo datoteko po znakih uporabljamo metodo *Read*. Ta nam vrne kodo znaka, ki ga preberemo. Ko bomo naleteli na konec datoteke, bo prebrana koda -1, ki jo uporabimo za to, da vemo, kdaj smo pri koncu.

Seveda ga moramo pri izpisu lepo pretvoriti v znak, saj bi drugače namesto



Slika 48: Pravilen izpis tekstovne datoteke

dobili (če bi bila 8. vrstica torej *Console.Write(beri);*)



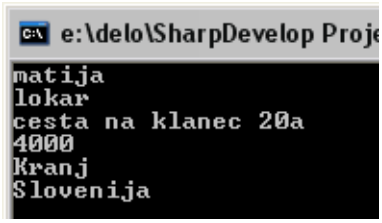
Slika 49: Kodiran izpis tekstovne datoteke

Če verjamete ali ne, gre za isto datoteko, le da so druga poleg druge izpisane kode znakov namesto njihovih grafičnih podob.

V C# imamo na voljo metodo *Split*, ki zna niz razdeliti na posamezne dele. Spomnimo se na poglavje o nizih (*string*). Ta del kode bo povedal vse:

```
string info = "matija;lokar;cesta na klanec 20a;4000;Kranj;Slovenija";
string[] tabInfo;
 // določimo znake, ki predstavljajo ločila med podatki
char[] locila = { ';' };  // za ločilo smo vzeli le ;
tabInfo = info.Split(locila);
for (int x = 0; x < tabInfo.Length; x++)
   Console.WriteLine(tabInfo[x]);
```

Če izvedemo to kodo, dobimo 6 vrstic, saj je v nizu pet ločil *';'*.

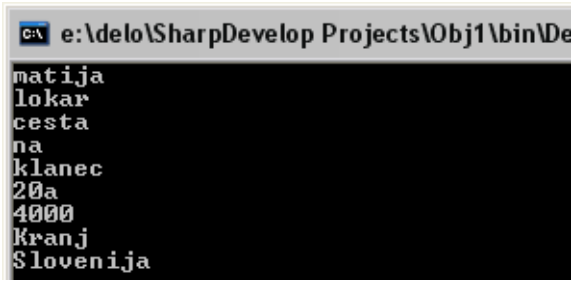


Slika 50:Uporaba metode Split (eno ločilo)

Če pa bi med ločila dodali še npr. presledek

```
char[] locila = { ';', ' ' };  //  ločili sta presledek in ;
```

bi dobili dodatne tri vrstice



Slika 51: Uporaba metode Split (dve ločili)

Seveda pa moramo paziti. Če npr. niz spremenimo v (med *matija* in *;* smo dodali presledek, prav tako smo med *cesta* in *na* dodali dodatni presledek)

```
string info = "matija ;lokar;cesta  na klanec 20a;4000;Kranj;Slovenija";
```

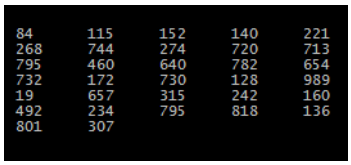
bomo dobili še dve vrstici

Slika 52: Težave pri uporabi metode *Split*

Poznati je potrebno strukturo datoteke, da znamo določiti razmenitveni znak (ločila)! V našem primeru bosta to presledek in tabulatorski znak ('\t'), ker bo datoteka tista, ki jo ustvari metoda *UstvariDat*, ki smo jo napisali prej. Ker pa je med števili lahko več teh ločil, bomo prazne nize prezrli!

15.12 IZ DATOTEKE V TABELO

Kako iz datoteke, kjer je v vrstici več števil,



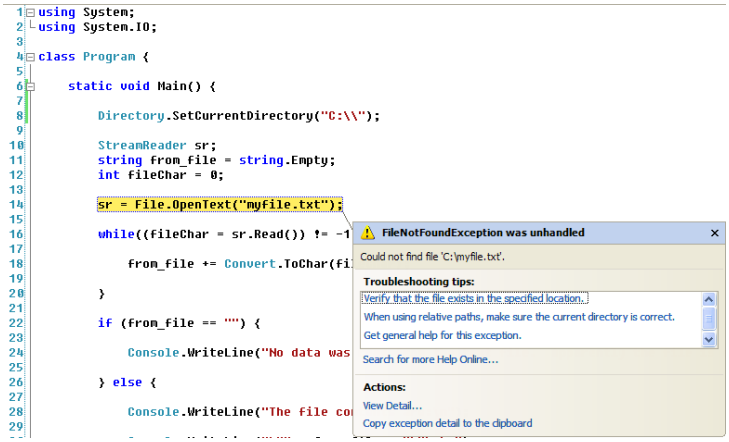
Slika 53: Izgled datoteke

dobiti posamezna števila. Predpostavimo, da v datoteki ni več kot 100 števil.

```
public static int[] IzDatStevilTabela(string vhod)
{
   StreamReader beri = File.OpenText(vhod);
   int[] tabela = new int[100];
   int koliko = 0;
   string vrst = beri.ReadLine();
   while (vrst != null)
   {
       //  vrstico moramo predelati v tabelo posameznih števil
      string[] tabInfo;
       // določimo znake, ki predstavljajo ločila med podatki
      char[] locila = { ' ', '\t' };  //  ločilo je presledek in tabulator
      tabInfo = vrst.Split(locila);
      for (int x = 0; x < tabInfo.Length; x++)
      {
         if (tabInfo[x] != "")
         {  //  če nismo dobili le prazni niz
            tabela[koliko] = int.Parse(tabInfo[x]);
            koliko++;
         }
      }
      vrst = beri.ReadLine();
   }
   beri.Close();
   return tabela;
}
```

15.13 DATOTEKE NI

Najbolj pogosto napako pri branju s tekstovnih datotek prikazuje slika



Slika 54: Datoteka ne obstaja

Gre za to, da smo z *OpenText* odpirali datoteko, ki je ni! A to že znamo preprečiti. Spomnimo se metode *File.Exists(ime)*,ki smo jo v razdelku o pisanju na datoteke uporabili zato, da nismo slučajno ustvarili datoteke z enakim imenom, kot jo ima že obstoječa datoteka (ker bi staro vsebino s tem izgubili). Torej je smiselno, da pred odpiranjem datoteke to preverimo. Popravimo eno od prejšnjih metod

```
public static void IzpisDatoteke4(string ime)
{
   if (!File.Exists(ime))
   {
      Console.WriteLine("Datoteka " + ime + " ne obstaja!");
   }
   else
   {
      StreamReader izhTok;
      izhTok = File.OpenText(ime);
       //  preberemo prvo vrstico in jo izpišemo na zaslon
      string vrstica = izhTok.ReadLine();
      int stevec = 1;
      while (vrstica != "")
      {
         Console.WriteLine(stevec + ": " + vrstica);
         vrstica = izhTok.ReadLine();
         stevec++;
      }
       //  zaprimo tok
      izhTok.Close();
   }
}
```

15.14 PONOVITEV

Ponovimo pomembne metode, ki jih srečamo pri delu s tekstovnimi datotekami.

Tabela 10: Razreda in osnovne metode za delo z datotekami

Razred	**Pomen**
StreamReader	Tip spremenljivke za branje toka.
StreamWriter	Tip spremenljivke, ki ga uporabimo pri spremenljivkah, v katerih je shranjena oznaka podatkovnega toka.
Metoda	**Pomen**
File.Exists()	Metoda, ki jo uporabimo zato, da preverimo obstoj datotek.
File.OpenText()	Metoda, ki izhodni tok poveže z datoteko, s katero delamo.
File.CreateText()	Metoda, ki ustvari datoteko in vrne oznako podatkovnega toka na katerega pišemo.
ReadLine()	Metoda, ki prebere celo vrstico v datoteki.
ReadToEnd()	Metoda, ki prebere vse vrstice v datoteki od trenutne vrstice pa do konca.
Read()	Metoda, ki nam vrne kodo znaka, ki ga preberemo iz datoteke.
WriteLine()	Metoda, ki zapiše dani niz na datoteko, po zapisu se premaknemo v novo vrsto.
Write()	Metodi, ki zapiše dani niz na datoteko
Close():	Metoda, s katero zapremo datoteko.

15.15 ZGLEDI

15.15.1 Kopija datoteke (po vrsticah)

Sestavimo metodo, ki bo naredila kopijo datoteke. Najprej bomo odprli dve datoteki. Prvo za branje (*File.OpenText*), drugo za pisanje *File.CreateText*), potem bomo brali s prve datoteke vrstico po vrstico (metoda *ReadLine*) in jih sproti zapisovali na izhodno datoteko. To bomo počeli tako dolgo, dokler prebrani niz ne bo *null*.

```
public static void Kopija(string vhod, string izhod)
{
   StreamReader beri = File.OpenText(vhod);
   StreamWriter pisi = File.CreateText(izhod);
   string vrst = beri.ReadLine();
   while (vrst != null)
   {
      Console.WriteLine(vrst);
```

```
        pisi.WriteLine(vrst);
        vrst = beri.ReadLine();
    }
    beri.Close();
    pisi.Close();
}
```

15.15.2 Kopija datoteke (po znakih)

Naredimo sedaj enako, le da tokrat kopirajmo datoteko znak po znak. Postopek bo enak, le da bomo brali z *Read* in konec preverjali z kodo znaka -1. Pravzaprav, če dobro premislimo – metoda bo v bistvu kombinacija prejšnje metode in metode *izpisDatotekePoZnakih*, kjer smo datoteko po znakih izpisali na zaslon. Sedaj bomo počeli enako, le da bomo namesto na zaslon pisali na datoteko.

```
public static void KopijaDatotekePoZnakih(string vhod, string izhod)
{
    StreamReader beri = File.OpenText(vhod);
    StreamWriter pisi = File.CreateText(izhod);
    int prebranZnak;
    prebranZnak = beri.Read();
    while (prebranZnak != -1)  //  konec datoteke
    {
        pisi.Write((char)prebranZnak);  //  izpisujemo ustrezne znake
        prebranZnak = beri.Read();
    }
    beri.Close();
    pisi.Close();
}
```

15.15.3 Datoteka s števili (vsako število v svoji vrsti), ki jih preberemo v tabelo

Dana je datoteka, na kateri so zapisana cela števila. Vemo, da na datoteki ni več kot 100 števil. Vsako število je v svoji vrstici. Sestavimo metodo, ki za dano ime datoteke vrne tabelo števil.

```
public static int[] IzDatTabela(string vhod)
{
    StreamReader beri = File.OpenText(vhod);
    int[] tabela = new int[100];
    int koliko = 0;
    string vrst = beri.ReadLine(); // preberemo celo vrstico s številom
    while (vrst != null)
    {
        tabela[koliko] = int.Parse(vrst);  // prebrano vrstico spremenimo v int in damo v tabelo
        koliko++;   // povečamo indeks v tabeli
        vrst = beri.ReadLine();
    }
```

```
   beri.Close();
   return tabela;
}
```

Kaj pa če ne vemo koliko je števil v datoteki: v tem primeru najprej izvedemo en prehod preko datoteke in preštejemo število vrstic. Nato kreiramo tabelo, ki ima takšno dimenzijo, kot je vrstic v datoteki.

15.15.4 Zapis tabele na datoteko

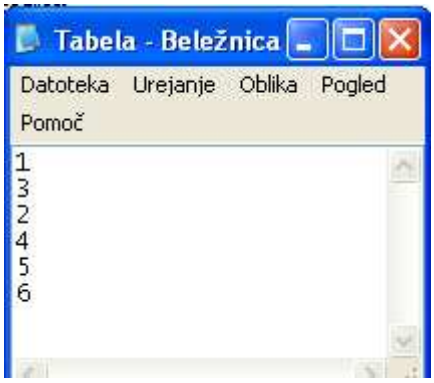
Sestavimo metodo, ki tabelo celih števil prepiše v datoteko, določeno z nizom, ki je tudi parameter metode. Vsak element tabele naj bo zapisana v svoji vrstici.

Primer tabele:

[1 3 2 4 5 6]

```
public static void IzpisovanjeNaDatoteko(int[] m, string ime)
{
    //  preverimo, če že obstaja datoteka z enakim imenom
   if (File.Exists(ime))
   {
      Console.WriteLine("Datoteka s takšnim imenom že obstaja");
   }
   else
   {
       //  odpremo datoteko v katero bomo zapisovali podatke
      StreamWriter oznaka = File.CreateText(ime);
       //  sprehodimo se čez tabelo
      for (int i = 0; i < m.Length; i++)
      {
          // posamezni element tabele zapišemo v datoteko in to v svojo vrsto
         oznaka.WriteLine(m[i]);
      }
       //  zapremo datoteko po uporabi
      oznaka.Close();
   }
}

public static void Main(string[] args)
{
  int [] tabela={1,2,3,4,5,6};    // deklaracija in inicializacija tabele
  IzpisovanjeNaDatoteko(tabela,@"c:\temp\Tabela.txt");     //  klic metode
}
```

Prikaz podatkov iz datoteke *Tabela.txt*:



Slika 55: Tabelo smo prepisali v datoteko

15.15.5 Prepis vsebine datoteke v tabelo

Napisano imamo datoteko z imenom *Tabela.txt*. Ali sedaj znamo zapisati te iste podatke iz datoteke nazaj v tabelo? Sicer bi šlo tako, kot smo si ogledali v prejšnjem zgledu. Najprej napišemo metodo, ki vrne število vrstic naše datoteke. To metodo potem uporabimo zato, da ustvarimo primerno veliko tabelo in se potem lotimo branja. Toda ta način ni dober za primere, ki imajo po več sto vrstic, saj bi bil porabljen čas prevelik. Boljši način je, da na kaj takega, kot je naknadno branje, pomislimo vnaprej in spremenimo format zapisa. V prvo vrstico datoteke zapišemo velikost tabele. V ostalih vrsticah pa zapišemo elemente tabele tako, kot smo jih v prvem primeru.

Popravljena metoda:

```
public static void IzpisovanjeNaDatoteko(int[] m, string ime)
{
   if (File.Exists(ime))
   {
      Console.WriteLine("Datoteka s takšnim imenom že obstaja");
   }
   else
   {
      StreamWriter oznaka = File.CreateText(ime);
       //  V prvo vrstico datoteke vpišemo velikost tabele
      oznaka.WriteLine(m.Length);
      for (int i = 0; i < m.Length; i++)
      {
         oznaka.WriteLine(m[i]);
      }
      oznaka.Close();
   }
}
```

15.15.6 Odstranitev števk

Sestavimo metodo *PrepisiBrez*, ki sprejme imeni vhodne in izhodne datoteke. Metoda na izhodno datoteko prepiše tiste znake iz vhodne datoteke, ki niso števke. Predpostavimo, da sta

imeni datotek ustrezni (torej, da datoteka za branje obstaja, datoteka za pisanje pa ne (oziroma da njeno morebitno prejšnjo vsebino lahko izgubimo)).

Najprej bomo odprli ustrezni datoteki. Prvo datoteko odpremo za branje in drugo za pisanje. Brali bomo posamezne znake iz vhodne datoteke. Če prebran znak ne bo številka, ga bomo prepisali v izhodno datoteko. To bomo počeli toliko časa, dokler ne bomo prebrali znaka s kodo -1 (torej znak EOF).

```
public static void PrepisiBrez(string imeVhod, string imeIzhod)
{
   StreamReader datZaBranje;  //  Ustvari podatkovni tok za branje na datoteki
   datZaBranje = File.OpenText(imeVhod);
   StreamWriter datZaPisanje;  //  Ustvari tok za pisanje na datoteko
   datZaPisanje = File.CreateText(imeIzhod);

    //  Prepis znakov iz ene datoteke na drugo datoteko
   int znak = datZaBranje.Read();  //  Prebere en znak
   while (znak != -1)
   {
       //  Primerjamo ali je prebrani znak številka
      if (!('0' <= znak && znak <= '9'))
               datZaPisanje.Write("" + (char)znak);  //  Če ni številka, zapišemo znak
      znak = datZaBranje.Read();
   }
    //  Zapremo tokova datotek
   datZaBranje.Close();
   datZaPisanje.Close();
}
```

15.15.7 Primerjava vsebine dveh datotek

Napišimo metodo *Primerjaj*, ki sprejme imeni dveh datotek in primerja njuno vsebino. Če sta vsebini datotek enaki, metoda vrne vrednost *true*, sicer vrne *false*. Predpostavimo, da sta imeni datotek ustrezni (torej, da datoteki za branje obstajata).

```
1:      public static bool Primerjava(string ime1, string ime2)
2:      {
3:              //  Odpremo obe datoteki za branje
4:              StreamReader dat1 = File.OpenText(ime1);
5:              StreamReader dat2 = File.OpenText(ime2);
6:              //  Preberemo vsebino prve in druge datoteke
7:              string beri1 = dat1.ReadToEnd();
8:              string beri2 = dat2.ReadToEnd();
9:              //  Zapremo obe datoteki za branje
10:             dat1.Close();
11:             dat2.Close();
12:             //  Primerjamo vsebini
13:             return (beri1.Equals(beri2));
14:     }
```

Razlaga. Najprej odpremo datoteki (vrstici 4 in 5). V niz *beri1* s pomočjo metode *ReadToEnd()* preberemo celotno vsebino datoteke *dat1* (vrstica 7). V niz *beri2* shranimo celotno vsebino datoteke *dat2* (vrstica 8). Zapremo datoteki (vrstici 10 in 11). Na koncu vrnemo rezultat, ki ga dobimo pri primerjavi niza *niz1* z nizom *niz2*.

15.15.8 Zamenjava

Napišimo program, ki preko tipkovnice prebere ime vhodne datoteke in ime izhodne datoteke. Nato vhodno datoteko prepiše na izhodno, pri čemer vse znake 'a' nadomesti z nizom "apa".

Ideja programa: Preberemo ime vhodne in izhodne datoteke in preverimo obstoj datotek. Nato odpremo vhodno datoteko za branje in izhodno datoteko za pisanje. V zanki izpisujemo znake iz vhodne datoteko v izhodno datoteko. Če je znak 'a', ga nadomestimo z "apa", vhodno in izhodno datoteko zapremo.

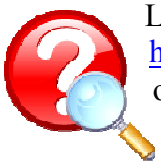
```
1:    public static void Main(string[] args)
2:    {
3:        // Vnos imen datotek
4:        Console.Write("Vnesi ime vhodne datoteke: ");
5:        string vhod = Console.ReadLine();
6:        Console.Write("Vnesi ime izhodne datoteke: ");
7:        string izhod = Console.ReadLine();
8:        // Preverjanje obstoja: vhodna mora obstajati, izhodna pa ne
9:        if (!File.Exists(vhod) || File.Exists(izhod))
10:       {
11:           Console.WriteLine("Napaka v imenu datotek.");
12:           return;
13:       }
14:       // Odprtje datotek
15:       StreamReader branje = File.OpenText(vhod);
16:       StreamWriter pisanje = File.CreateText(izhod);
17:       // Prenos podatkov
18:       while (branje.Peek() != -1)
19:       { // Do konca datoteke
20:           char znak = (char)branje.Read();
21:           if (znak == 'a')
22:           {
23:               pisanje.Write("ap"); // Zadnji 'a' bo dodal naslednji stavek
24:           }
25:           pisanje.Write(znak);
26:       }
27:       // Zapremo datoteki
28:       branje.Close();
29:       pisanje.Close();
30:   }
```

Razlaga. Preberemo ime vhodne in izhodne datoteke (vrstici 4 – 7). Nato preverimo obstoj datotek (8). Če vhodna datoteka ne obstaja ali izhodna datoteka obstaja, izpišemo obvestilo (vrstica 11) ter prekinemo izvajanje metode (12). Datoteki odpremo za branje oziroma pisanje (15 in 16). Nato z zanko *while* beremo posamezne znake iz vhodne datoteke in jih prepisujemo na izhodno datoteko (18 - 26). Za pogoj smo uporabili metodo *Peek*, ki vrne vrednost -1 le v primeru, da smo na koncu datoteke. Če je prebran znak 'a' (21), pred njim na izhodno datoteko zapišemo še niz "ap" (23). Znak 'a' iz vhodne datoteke na ta način zapišemo na izhodno datoteko z nizom "apa". Zanko končamo, ko preberemo kodo oznake EOF. Po koncu prepisovanja znakov datoteki zapremo (28 in 29).

15.16 POVZETEK

Datoteke bomo v svojem programu uporabili tedaj, kadar obdelujemo veliko količino podatkov oz. kadar hočemo, da se podatki ohranijo tudi po zaključku izvajanja programa. Razložila sva le osnovne operacije s tekstovnimi datotekami, ki jih uporabljamo za shranjevanje teksta. Dostop do podatkov je tu sekvenčen. Kadar pa bomo želeli v datoteko shraniti velike količine podatkov, ki jih želimo kasneje tudi obdelovati, bomo raje uporabili binarne datoteke. Podatki, ki jih pišemo na binarno datoteko, se ne pretvarjajo v znake, ampak se zlogi zapišejo tako, kot so shranjeni v pomnilniku. Vendar pa je delo s takimi datotekami zahtevnejše in presega namen tega gradiva.

Za utrjevanje znanja je potrebno napisati čim več programov, ki rešujejo probleme v povezavi z delom z datotekami. Vrsto rešenih in tudi nerešenih nalog najdemo na primer v Jerše G. in Lokar M., Programiranje II, Rekurzija in datoteke, wikiju Csharp na naslovu http://penelope.fmf.uni-lj.si/C_sharp/index.php/Datoteke, v Petric D., Spoznavanje osnov programskega jezika C# in v Uranič S., Microsoft C#.NET.



16 ZAKLJUČEK

V samem besedilu sva poskušala prikazati tiste osnovne značilnosti jezika C#, ki naj bi jih spoznali študenti predmeta Programiranje 1. Glede na število ur namenjenih predmetu je snovi kvečjemu preveč, čeprav sva določene teme obdelala zelo na kratko (na primer teme varovalni bloki, dedovanje, podatkovni tipi, podatkovni tokovi ...), druge pa spet zavestno spustila (uporabniški vmesnik in dogodkovno programiranje, polimorfizem, vmesniki, lastnosti v razredu, binarne datoteke ...).

Večkrat sva napisala manj razlage, kot bi si jo želela. Prav tako bi bilo glede na izkušnje pri poučevanju dobrodošlo še večje število zgledov. Pravila projekta Impletum pač postavljajo določene omejitve glede dolžine gradiva. Zato posebej tiste, ki jim bo zgledov in razlage primanjkovalo, vabiva, da si ogledajo tudi gradiva v seznamu literature, še posebej pa gradiva, nastala v oviru projekta "UP ali Učenje programiranja - Kako poučevati začetni tečaj programskega jezika", dosegljiva na naslovu http://up.fmf.uni-lj.si.

17 SEZNAM LITERATURE IN VIROV

- C# Practical Learning 3.0, (online). 2008. (citirano 1.12.2008). Dostopno na naslovu http://www.functionx.com/csharp/
- C# Station, (online). 2008 (citirano 1.12.2008). Dostopno na naslovu http://www.csharp-station.com/Tutorial.aspx
- Coelho, E., Crystal Clear Icons, (online). 2008 (citirano 1.12.2008). Dostopno na naslovu: http://commons.wikimedia.org/wiki/Crystal_Clear
- CSharp Tutorial, (online). 2008. (citirano 2. 12. 2008). Dostopno na naslovu http://www.java2s.com/Tutorial/CSharp/CatalogCSharp.htm
- FunctionX Inc, (online). 2008. (citirano 1.12.2008). Dostopno na naslovu http://www.functionx.com/csharp/
- http://lisa.uni-mb.si/osebje/bonacic/predmeti/orodja_za_razvoj_aplikacij_in_vs3/index.htm
- Jerše G. in Lokar M., Programiranje II, Objektno programiranje, B2, 2008
- Jerše G. in Lokar M., Programiranje II, Rekurzija in datoteke, B2, 2008
- Joe Mayo, C# Station Tutorial, (online). 2008 (citirano 1.12.2008). Dostopno na naslovu http://www.csharp-station.com/Tutorial.aspx
- Kerčmar N., Prvi koraki v Javi, diplomska naloga, UL FMF, 2006
- Lokar M., Osnove programiranja : programiranje – zakaj in vsaj kaj, Zavod za šolstvo, Ljubljana, 2005
- Lokar M., Wiki C#, (online). 2008 (citirano 2.12.2008). Dostopno na naslovu http://penelope.fmf.uni-lj.si/C_sharp
- Lokar M. et. al., Projekt UP - Kako poučevati začetni tečaj programskega jezika, sklop interaktivnih gradiv, (online). 2008. (citirano 1. 12. 2008). Dostopno na naslovu: http://up.fmf.uni-lj.si/
- Lokar M. in Uranič S., Programiranje 1 - zbirka rešenih nalog v C#, v pripravi
- Murach J. in Murach M., Murach's C# 2008, London, 2008
- Petric D., Spoznavanje osnov programskega jezika C#, diplomska naloga, UL FMF, 2008
- Sharp J., Microsoft Visual C# 2005 Step by Step, Washington, 2005
- SharpDevelop, razvojno okolje za C#, (online). 2008. (citirano 7. 12. 2008). Dostopno na naslovu http: // www.icsharpcode.net/OpenSource/SD/.
- Uranič S., Microsoft C#.NET, (online). Kranj, 2008. (citirano 10. 10. 2008). Dostopno na naslovu *http://uranic.tsckr.si/C%23/C%23.pdf*
- WIKI DIRI 06/07, (online). 2008. (citirano 10. 10. 2008). Dostopno na naslovu: http://penelope.fmf.uni-lj.si/diri0607/index.php/Kategorija:Naloge
- Wikimedia Foundation, (online). 2008 (citirano 1.12.2008). Dostopno na naslovu: http://commons.wikimedia.org/wiki/Crystal_Clear