C# .NET

PROGRAMIRANJE ZA VSŠ

Izbor gradiv, avtorja posameznih poglavij: Srečo Uranič in Matija Lokar

Kaj je C# in .NET Framework

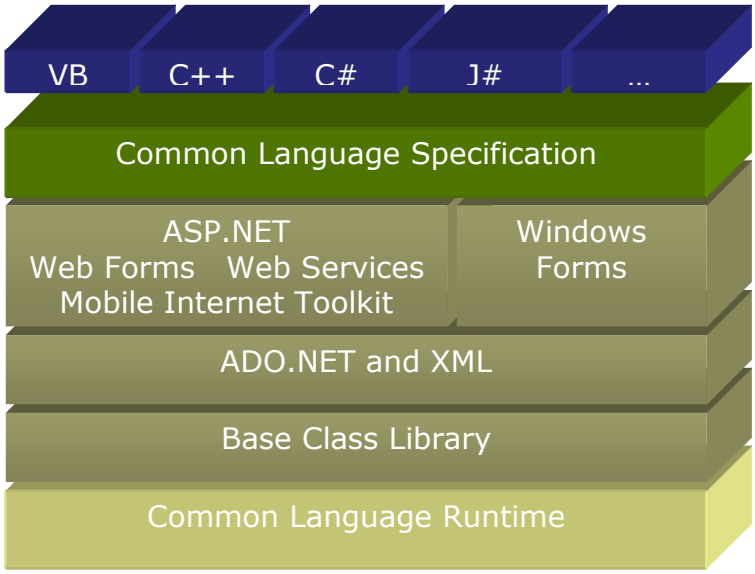
C# je Microsoftov objektno orientiran programski jezik. Jezik je nekakšna kombinacija programske moči jezika C++ z dodanimi dobrotami iz Visual Basica in Jave. Čeprav C# temelji na C++, pa vsebuje številne sestavine značilne za Javo. Razvijalcem Visual C# omogoča razvoj visoko prenosljivih aplikacij.

C# je bil oblikovan za delo z Microsoftovo .NET platformo (.NET Framework). .NET Framework je platforma/knjižnica, ki predstavlja ogrodje za vse .NET orientirana programska orodja in aplikacije za osebne računalnike, dlančnike, pametne telefone, razne vgrajene sisteme,…) Vsebuje pester nabor jezikov (C++, C#, Visual Basic, VBScript, J#, JScript…), omogoča hiter in predvsem lažji razvoj kot tudi izvajanje tako obsežnih in zahtevnih, kot tudi majhnih in enostavnih projektov. Omogoča tudi kreiranje projektov, sestavljenih iz posameznih modulov, zgrajenih z različnimi programskimi jeziki. Obenem omogoča kar najbolj možno prenosljivost programske kode. Vsebuje tudi kar se da optimizirano vgrajeno kodo, standardizirano z najnovejšo tehnologijo kot npr. XML in SOAP.

Razvojno okolje VISUAL Studio .NET omogoča razvijalcem izbor enega od štirih orodij (programskih jezikov) za razvoj tako konzolnih kot tudi vizuelnih aplikacij: **Visual C#**, **Visual C++**, **Visual Basic** in **Visual J#**. Pri instalaciji se lahko odločimo, da instaliramo vse programske jezike, ali pa izberemo (odkljukamo) le tistega, ki ga bomo uporabljali, oz. s katerim bomo razvijal naše aplikacije.

Razvojno okolje **VISUAL Studio** nastopa v več različicah, odvisno od namena uporabe: najbolj znane so različice **Express Edition**, **Standard Edition** in **Professonal Edition**. Verzija **Express Edition** je brezplačna, naložimo pa si jo lahko kar preko spleta. Razlaga in vaje v tej literaturi bodo temeljili večinoma na različici **Express Edition**, nekatere pa tudi na **Standard Edition**.

Zgradba .NET



Izdelava konzolnih aplikacij v VISUAL C# . NET

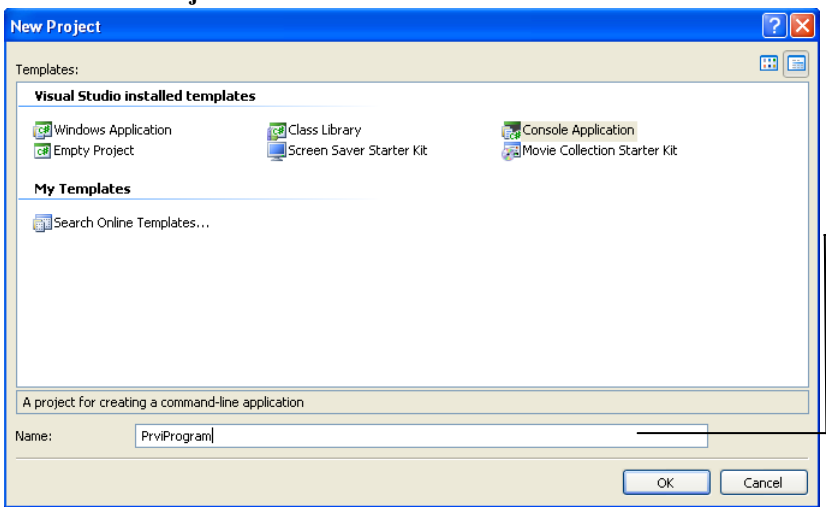
Začetek programiranja v okolju Visual C# - prva konzolna aplikacija

Pri kreiranju novega projekta je majhna razlika glede na to, katero različico razvojnega okolja uporabljamo. Kasneje razlik skoraj ni, še posebej ne pri razvoju konzolnih aplikacij.

Začenjanje novega projekta v različici Express Edition

Za začetek novega projekta v okolju **Visual C# Express Edition** odprimo meni

File -> New Project…

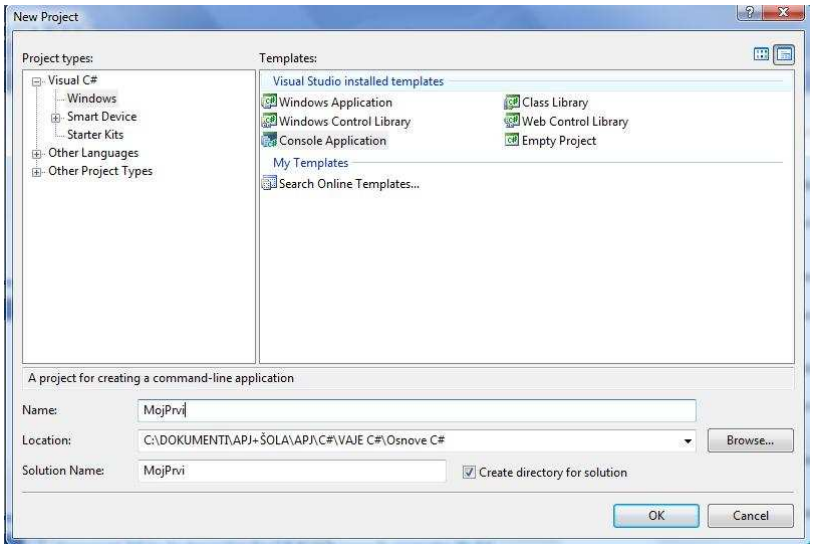


Odpre se novo okno **New Project** v katerem pod **Templates** (Vzorci) izberemo ikono **Console Application**. Na dnu okna napišimo še ime naše prve konzolne aplikacije, npr. *Prvi Program*.

S klikom na gumb OK se ustvari **nov projekt**.

Začenjanje novega projekta v različici Standard Edition

Za začetek novega projekta v okolju **Visual C# Standard Edtion** odprimo meni **File -> New Project…**

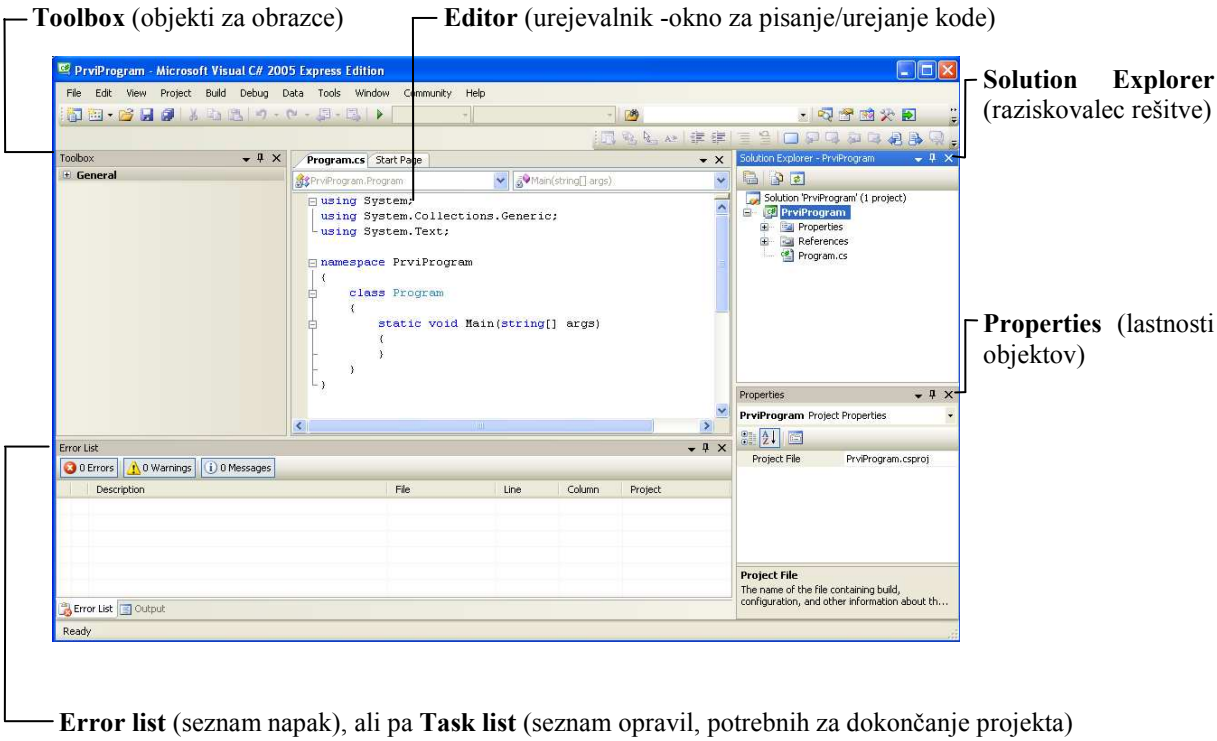


Odpre se novo okno **New Project** v katerem pod **Templates** (Vzorci) izberemo ikono **Console Application**. Na dnu okna napišimo še ime naše prve konzolne aplikacije (**Name**), npr. *Moj Prvi*, določimo mapo v kateri bo shranjen naš projekt (**Location**) in še zapišemo ime naše rešitve (**Solution Name** - priporočljivo je, da sta vsaj v začetnih projektih imeni programa in rešitve enaka).

S klikom na gumb OK se ustvari **nov projekt**.

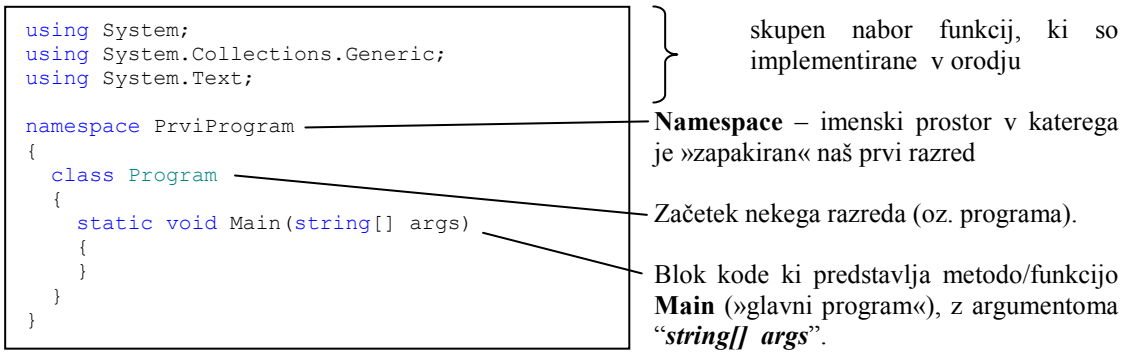
Pisanje programske kode z uporabo IntelliSense tehnologije

Ko smo ustvarili nov projekt, se odprejo **osnovna** okna za urejanje naše prve konzolne aplikacije.



Kratek opis osnovnih oken razvojnega okolja

- **Solution Explorer** (raziskovalec rešitve): V obliki drevesne strukture nam prikazuje in omogoča dostop do vseh datotek znotraj našega projekta. Datoteke lahko dodajamo, jih spreminjamo in brišemo. Okno je dostopno tudi preko tipkovnice, s kombinacijo tipk **Ctrl+Alt+L**. Na dnu tega okna sta dva jezička: **Solution Explorer** in **ClassView**. Okno **Class View** prikazuje hierarhijo razredov znotraj rešitve. Dostopno je tudi preko tipkovnice, s kombinacijo tipk **Ctrl+Alt+L.** Okno je bolj uporabno kot **Solution Explorer** predvsem pri velikih projektih.
- **Toolbox** (objekti za obrazce): Vsebuje objekte (kontrole) za izdelavo **Windows Forms** (obrazci) ali **Web Forms** aplikacije. Pri izdelave konzolne aplikacije je seveda okno prazno, saj v tem primeru ne delamo z že pripravljenimi objekti. Okno je dostopno tudi preko tipkovnice, s kombinacijo tipk **Ctrl+Alt+X**.
- **Properties** (lastnosti objektov): V tem oknu spreminjamo in nastavljamo lastnosti in dogodke objektom pri izdelavi **Windows Forms** ali **Web Forms** aplikacije. Pri izdelave konzolne aplikacije je seveda okno prazno, saj v tem primeru ne delamo z že pripravljenimi objekti. Okno je dostopno tudi preko tipkovnice, s pomočjo tipke **F4**.
- **Error list** (seznam napak), ali pa **Task list** (seznam potrebnih opravil, ki so potrebna za dokončanje projekta). Vrsta prikazanega okna je odvisna od izbire v meniju **View** (lahko je **Error List**, **TaskList**, Output, …).
- **Editor** (urejevalnik -okno za pisanje/urejanje kode): Okno je namenjeno za pisanje ter urejanje programske kode. Omogoča tudi skrivanje in prikazovanje delov kode (**outlining**), ki smiselno tvorijo neko celoto (+ in – na levi strani urejevalnika). V primeru da je del kode skrit (da levi strani znak -) lahko (ne da bi skriti del odprli s klikom na znak -) vidimo vsebino tudi tako, da se z miško premaknemo čez povzetek skritega dela besedila na desni strani vrstice.



- Opomba: **C#** je **case sensitive** jezik, zato mora biti metoda **Main** napisana z veliko začetnico, sicer bo prevajalnik javil napako.

V **Visual C#** lahko v fazi načrtovanja (razvoja) programa ustvarjamo t.i. **regije** (začetek regije **#region** konec **#endregion**). Del kode, zapisane med ti dve besedici lahko kadarkoli skrijemo in zopet prikažemo, s tem pa pridobimo na preglednosti izvornega programa.

Ko smo projekt ustvarili, nam je **C#** ustvaril datoteko **Program.cs**, ki definira razred imenovan **Program**, le-ta pa vsebuje metodo imenovano **Main**. **Vse metode našega programa morajo biti definirane znotraj razreda imenovanega Program.** Metoda **Main** je posebna, ker je vstopna točka programa, zato **mora biti statična** (besedica **static mora** biti zapisana pred imenom metode). Kaj pa pravzaprav pomeni, da je neka metoda statična, bomo spoznali precej kasneje, pri poglavju o razredih.

Programsko kodo naše prve konzolne aplikacije bomo napisali v metodo **Main**, zgleda pa takole:

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

Console je **razred**, ki je definiran v imenskem prostoru (knjižnici) **System**. Pojem **imenski prostor** je razložen v zadnjem delu tega poglavja, na tem mestu pa moramo napisati nekaj o pojmu **razred**. Pojem **razred** in z njim povezan pojem **objekt**, sta pravzaprav osnova objektnega programiranja. Poenostavljeno rečeno je razred neka kompleksna podatkovna struktura, ki vsebuje množico lastnosti in metod, ki operirajo nad temi lastnostmi oz. opravljajo kako drugo nalogo. Razred je abstrakten pojem, v programih pa jih uporabljamo tako, da iz njih izpeljemo objekte, to je predstavnike razredov. O tem, kako napišemo svoj razred, kako mu določimo lastnosti in metode, ter še veliko drugega kar se tiče razredov, bomo spoznali veliko kasneje. Zaenkrat pa bomo razrede obravnavali tako, da bomo spoznali njihova imena, kako naredimo predstavnika (primerek, instanco) tega razreda (oz. objekt), čemu je kakšen razred namenjen in kako preko objektov uporabljamo njegove lastnosti in metode. Zapomnimo si le to, da do lastnosti in metod razredov (oz. lastnosti in metod objektov) pridemo tako, da za imenom razreda (objekta) zapišemo piko, temu pa sledi ime lastnosti oz. metode.

Ker smo knjižnico **Console** navedli na začetku naše prve konzolne aplikacije (stavek **using System**), se lahko na ta razred sklicujemo samo z imenom. V primeru, da pa na začetku programa ne bi bilo vrstice **using System**, bi do razreda prišli preko knjižnice **System** (torej **System.Console…).**

Razred **Console** je prvi izmed množice razredov, ki jih bomo uporabljali. Zaenkrat moramo o njem vedeti le to, da ta razred vsebuje metode, ki lahko berejo posamezne znake ali pa zaporedja znakov (stavke oz. vrstice) preko konzole (najpogosteje tipkovnice). Vsebuje tudi metode za pretvarjanje podatkov (npr. števila v zaporedja znakov-**stringe** in obratno), metode za formatiranje spremenljivk, ter za formatiran izpis npr. na ekran ali pa v datoteko.

Ko za besedo **Console** napišemo piko, se nam odpre t.i. **IntelliSense** meni (kontekstno občutljiva pomoč). Le-ta nam prikaže imena vseh članov in metod (**atributov**) razreda **Console** (v splošnem pa imena razreda, ki smo ga napisali pred znakom pika). Na levi strani vsakega atributa je ikona, ki označuje tip določenega atributa.

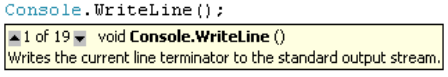
Ikona	Pomen
	method – metoda
	property – lastnost
	class – razred
	struct – struktura
	enum - naštevanje
	interface - vmesnik
{ }	namespace – imenski prostor
	event - dogodek
	delagate - delegat

V **IntelliSense** meniju poiščimo metodo **WriteLine** in pritisnimo <**Enter**> ali dvakrat kliknimo z miško na metodo - metoda je s tem dodana programski kodi. Za tem napišimo oklepaj in zaklepaj, znotraj oklepaja pa v narekovajih napišimo besedilo, ki nam ga bo program izpisal na zaslon - v našem primeru "Pozdravljen svet!" . Za oklepajem dodajmo še podpičje, saj s tem zaključimo ta ukaz (dvopičje je znak za konec stavka).

> ❖ Opomba: Kadar je na vrhu okna (zavihka oz. okna urejevalnika) v katerem je naša koda znak 
>
> zvezdica (*), to pomeni, da je bila v naši datoteki narejena vsaj ena sprememba, potem, ko smo jo nazadnje shranili. Ročno shranjevanje ni potrebno, saj se shranjevanje izvede avtomatsko takoj ko poženemo ukaz za prevajanje. Seveda pa je ročno shranjevanje priporočljivo takrat, kadar smo zapisali veliko vrstic kode, pa prevajanja še ne želimo pognati.

V splošnem nam **IntelliSense** nudi več vrst pomoči

- **List Members** : kombinacija tipk **Ctrl-J** nam prikaže seznam vseh članov (**members**) nekega atributa, nad katerim se nahaja kazalnik miške.
- **Parameter info** (informacije o parametrih neke metode): z miško kliknimo med oklepaja poljubne metode, in uporabimo kombinacijo tipk **Ctrl-Shift-Space**. V okvirju (seveda če le ta za to metodo sploh obstaja) se prikažejo vse možne variante izbrane metode. V levem zgornjem kotu imamo podatek o tem, koliko različnih vrst te metode obstaja (**preobložene metode!!!**). S klikom na trikotnika se lahko sprehodimo po teh metodah, obenem pa so prikazani podatki o številu in tipu parametrov, v spodnjem delu okna pa je še razlaga metode.



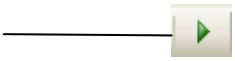
- **Quick Info** (hitra pomoč): S kombinacijo tipk **Ctrl-I** se nam namreč ob kazalniku miške prikaže sličica daljnogleda; če sedaj miško zapeljemo čez poljuben identifikator v urejevalniškem oknu, se v majhnem okencu prikazujejo osnovne informacije o tem identifikatorju.
- **Complete Word** (dokončaj besedo): S kombinacijo tipk **Ctrl-Space** dosežemo, da se neka beseda, ki jo pišemo, avtomatsko izpiše do konca, v primeru, da pa je možnih besed več, pa nam **C#** v okvirčku, v obliki **PopUp** menija, ponudi seznam vseh možnih besed (to je seznam vseh možnih metod, dogodkov, lastnosti, …). Zraven vsakega od njih je ustrezna ikona, ki označuje za kakšno besedo gre (ali je to metoda, lastnost, dogodek,…). Pomen ikon je pojasnjen v posebni tabeli na prejšnji strani.

Prevajanje in zagon projekta

Naš prvi projekt je sedaj pripravljen za prevajanje. Na voljo imamo več možnosti

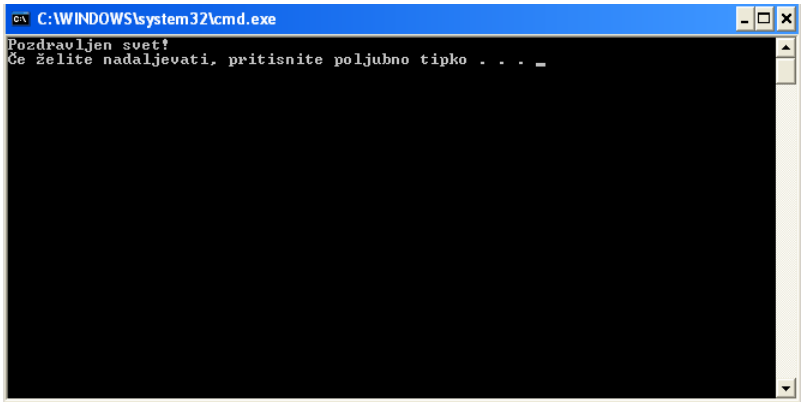
- Če želimo naš projekt le prevesti, ne pa tudi zagnati, potem v meniju **Build** kliknemo opcijo **Build Solution**. V primeru, da smo naredilo kako napako, nam **C#** v oknu **ErrorList** napiše vse potrebne informacije o številu in vrsti napak, pa tudi natančno informacijo o vrstici in stolpcu kjer je napaka;

- Prevajanje in obenem zagon projekta poženemo iz menija **Debug**, opcija **Start Debugging** (tipka **F5**) ali pa **Start Without Debugging** (**Ctrl-F5**). Razlika med načinoma je v tem, da se bo v drugem primeru okno, v katerem se bo izvedla naša aplikacija takoj zaprlo in si eventuelnih rezultatov(v našem primeri izpisa na zaslonu) ne bomo mogli ogledati;

- Kliknimo gumb **Start Debugging** v orodjarni 

- O ostalih opcijah (**Rebuild Solution**, **Step info**, **Step Over**) bo več zapisanega v nadaljevanju.

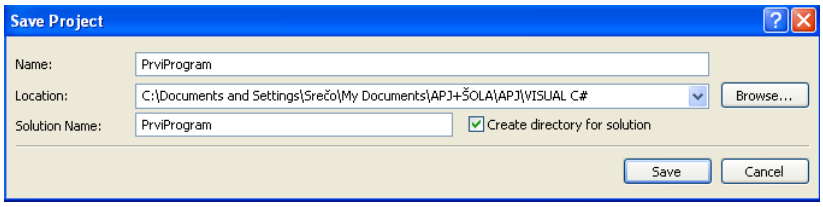
Naš program poženemo torej z opcijo **Ctrl-F5** (oziroma v meniju **Debug** kliknimo **Start Without Debugging**). Če je program brez napak, se odpre novo okno, v katerem steče naš program in se v njem prikažejo rezultati našega programa. Konzolno okno je privzeto črno, napis na oknu prikazuje mapo in ime programa, barva znakov v oknu pa je bela.



Prepričajmo se, da je prikazano okno aktivno (oz. da ima **focus**) in pritisnimo poljubno tipko. Okno se zapre in vrnemo se v okolje **Visual C#**.

Naš projekt se je ob prvem prevajanju tudi v celoti shranil, vendar POZOR! Običajno želimo projekt shraniti v točno določeno mapo, zaradi česar običajno projekt pred prvim zagonom shranimo tja, kamor želimo sami in ne tja kamor so nastavljene privzete nastavitve. To storimo tako, da pred prvim prevajanjem v meniju **File** kliknemo na ikono **Save All** (ali pa kar na ikono **Save All** v orodjarni).

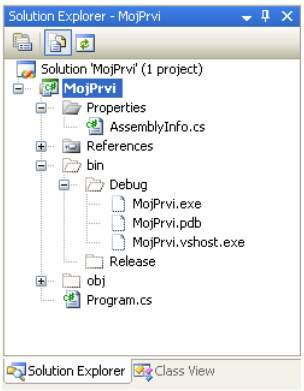
Če uporabljamo različico **Express Edition**, se prikaže okno za nastavitev shranjevanja, v katerem nastavimo (zapišemo) vse potrebne podatke o tem, kam in pod kakšnim imenom želimo naš projekt shraniti:



Ime našega programa lahko sedaj poljubno spremenimo (vrstica **Name**), izberemo mapo v kateri želimo imeti naš projekt (**Location**). Če mapa še ne obstaja, jo lahko ob kliku na gumb **Browse** skreiramo. V vrstici **Solution Name** imamo možnost, da se ime programa razlikuje od same rešitve, a priporočljivo je, da sta ime programa in konkretna rešitev poimenovana enako. Dodana je še možnost kreiranja novega imenika za naš program (znotraj izbranega imenika/mape v vrstici **Location**). Priporočljivo pa je, da za vsak nov projekt okolje **Visual C#** samo kreira svoj imenik oz. mapo, zato da imamo vse potrebne datoteke, ki tvorijo projekt, v svoji mapi.

Pri shranjevanju v različici Standard zgornjega okna ni, saj smo vse potrebne nastavitve naredili že pri odpiranju projekta.

Preglejmo še katere datoteke so nastale ob kreiranju naše prve konzolne aplikacije. V **Solution Explorerju** kliknimo ikono **Show All Files**. Po imenom mape našega projekta (v našem primeru **MojPrvi**) se prikažejo še mape **Properties**, **bin** in **obj**. Te mape se kreirajo takoj, ko poženemo prevajanje programa in vsebujejo izvršilno (**executable**) verzijo našega programa, ter še nekaj datotek, ki so potrebne za razhroščevanje (**debugging**) programa. Če v **Solution Explorerju** kliknimo znak + pred mapo **bin**, se prikaže še vsebovana podmapa **Debug**. Kliknimo še na + pred mapo **Debug** in prikažeta se datoteki **MojPrvi.exe** in **MojPrvi.pdb**.



- Opomba: Vsak izvorni program (**source code**) lahko prevedemo v izvršilno verzijo (**executable code**) tudi neposredno iz komandne vrstice (konzole) z uporabo **C# csc** prevajalnika.

Samo zaradi tega, da se navadimo na to, da imajo razredi v splošnem veliko lastnosti in metod, ki so nam takoj na voljo in jih torej lahko uporabimo v naših aplikacijah, je tule tabela **nekaterih** lastnosti in metod razreda **Console**, ki jih lahko preizkusimo v naslednji vaji.

Lastnost	**R a z l a g a**
BackgroundColor	Pridobivanje oz. nastavljanje barve ozadja konzole.
ForegroundColor	Pridobivanje oz. nastavljanje barve znakov (fontov).
Title	Besedilo, ki je izpisano na vrhu konzolnega okna.
Metode	**R a z l a g a**
Beep	Zvočni signal.
Clear	Brisanje vsebine konzolnega okna.
Read	Branje naslednjega znaka z vhodnega toka, npr,. tipkovnice.
ReadKey	Branje znaka ob uporabnikovem pritisku tipke oz. kombinacije tipk.
ReadLine	Branje vrstice teksta (zaporedja znakov do pritiska tipke <Enter>) z vhodnega toka.
ToString	Pretvorba nekega objekta (npr celega števila) v zaporedje znakov (string).
Write	Pisanje besedila na standardni izhodni tok, npr. na ekran
WriteLine	Pisanje besedila na standardni izhodni tok, npr. na ekran, ki mu sledi še oznaka za konec vrstice.

Nekaj primerov uporabe lastnosti in metod razreda **Console**:

```
//barva ozadja bo bela
Console.BackgroundColor = ConsoleColor.White;
//celo okno bo belo
Console.Clear();
//barva pisave bo črna
Console.ForegroundColor = ConsoleColor.Black;
//napis na oknu
Console.Title = "POZDRAVNO OKNO";
//zvočni signal, s frekvenco 1000 HZ, ki traja 500 milisekund
Console.Beep(1000,500);
```

```
//Pozdravno sporočilo
Console.Write("Dober dan!\n\nVnesi svoje ime in priimek: ");
//zahtevamo vnos podatkov preko tipkovnice. Vnos zaključimo z Enter
string stavek=Console.ReadLine();
Console.WriteLine("\n\n\nPozdravljen " + stavek+"!\n\n\n\n");
```

Še razlaga metode **Console.Readkey()**:

```
Console.Write("Pritisni poljubno tipko ali kombinacijo tipk: ");
//Funkcija ReadKey vrne vrednost tipa ConsoleKeyInfo
ConsoleKeyInfo k;
//Preko lastnosti KeyChar spremenljivke tipa ConsoleKeyInfo dobimo oznako tipke oz. oznako
//kombinacije tipk
k = Console.ReadKey();
Console.WriteLine();
Console.WriteLine("Pritisnil si tipko "+k.KeyChar);
```

Napake pri prevajanju (Compile Time Error) in napake pri izvajanju (Run Time Error)

Pri pisanju kode se seveda lahko zgodi, da naredimo kakšno napako. V svetu programiranja napakam v programu rečemo tudi **hrošči** (**bugs**). Na srečo ima okolje **Visual C#** sposobnost, da nekatere vrste napak odkrije.

Poznamo tri vrste napak:

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

Prvi dve vrsti napak prevajalnik oziroma okolje odkrije in nam pomaga pri njihovem odpravljanju. Te vrste napak zaradi tega navadno niso problematične, saj jih zaradi pomoči prevajalnika popravimo sorazmerno enostavno in hitro. Bolj problematične so tretje vrste napak – te napake so pomenske, njihovo odpravljanje pa zaradi tega veliko težje in dolgotrajnejše. **Visual C#** pa ima tudi za odpravljanje logičnih oziroma pomenskih napak na voljo posebno orodje, ki pa ga bomo spoznali v drugem delu te literature. To je t.i. **razhroščevalnik** oz. **Debugger**.

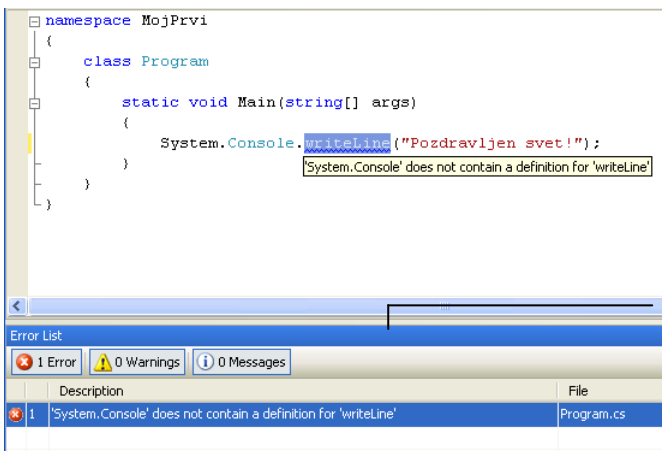
Na primeru naše prve konzolne aplikacije si poglejmo prikaz in odpravljanje prvih dveh vrst napak. Recimo, da smo se pri pisanju programske kode zmotili in namesto pravilnega zapisa

```
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 le-ta 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.



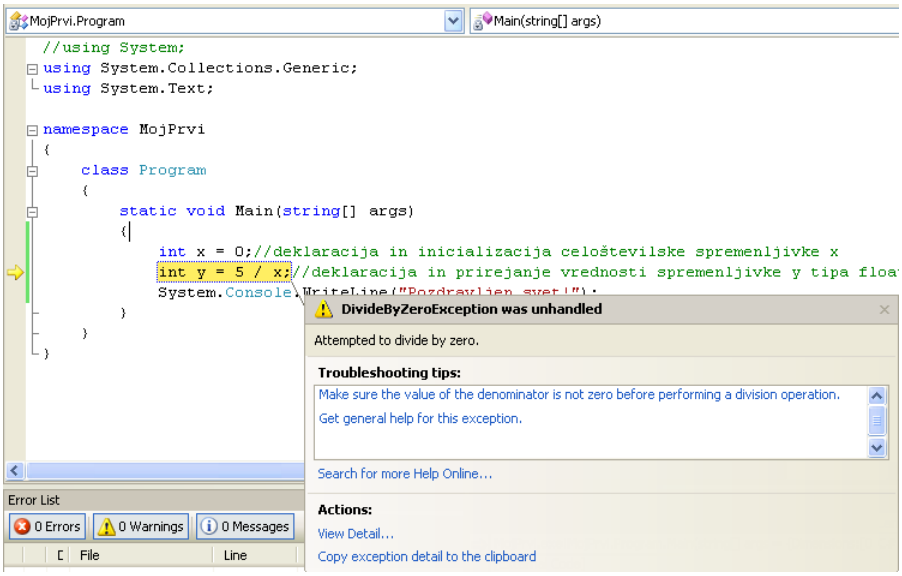
Poleg tega nam **Visual C#** podčrta del kode, kjer se napaka nahaja. Če se z miško premaknemo nad podčrtani del kode, kjer je napaka, nam **Visual C#** v okvirčku pod to besedo izpiše za kakšno napako gre.

Napako popravimo in sprožimo novo prevajanje. V kolikor napak ni več, se bo naš program prevedel in zagnal, sicer pa v oknu **Error List** dobimo nov seznam napak. Postopek ponavljamo, dokler ne odpravimo vseh napak.

Zgodi pa se, da se naš program prevede, a kljub temu pride do napake pri samem izvajanju programa. Kot primer take napake poglejmo klasično napako pri deljenju 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 celoštevilčne spremenljivke y
System.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 z obvestilom o napaki.



Taki napaki pravimo napaka med izvajanjem programa (**Run Time Error)**.

Program moramo seveda popraviti tako, da se izognemo takim operacijam (npr. z **if** stavkom v katerem preverimo vrednost delitelja).

Imenski prostori

Majhni programčki oz. programi postajajo s časoma vedno večji in pojavita se dva problema. Prvi je ta, da so večji programi težje obvladljivi in razumljivi kot pa manjši programi. Drugi problem pa je v tem, da več kode navadno pomeni več novih imen, več spremenljivk, funkcij, razredov, … S tem, ko se število teh atributov veča, pa slej ko prej naletimo na problem podvajanja imen, kar pa seveda ni dovoljeno. Program ne dela, potrebno je spreminjanje imen, a to opravilo je pri obsežnih programih lahko mukotrpno ali celo neizvedljivo. Problem podvajanja v sodobnih programskih jezikih rešujejo t.i. imenski prostori (**namespaces**), v katere »zapakiramo« našo kodo. Poskrbeti moramo le za to, da je ime našega imenskega prostora unikatno. Imenski prostor je nabor imen, spremenljivk, razredov, …, ki logično pripadajo neki celoti, v kateri velja pravilo neponovljivosti.

Če želimo npr. narediti nov razred z imenom ***MojPrviPozdrav***, je bolje da naredimo razred poimenovan ***Pozdrav*** znotraj imenskega prostora ***MojPrvi*** takole:

```
namespace MojPrvi
{
```

```
    class Pozdrav
    {
      ...
    }
}
```

Na razred ***Pozdrav*** se sedaj lahko sklicujemo kot ***MojPrvi.Pozdrav***. V primeru, da bo razred z enakim imenom kreiral še kdo drug v okviru njegovega imenskega prostora, ter ga instaliral na naš računalnik, bo naša aplikacija še vedno delala brez problemov. Priporočilo .NET platforme je, da vse razrede definiramo v imenskih prostorih, .NET razvojno okolje pa to priporočilo nadgradi s tem, da za ime projekta prevzame ime zunanjega imenskega prostora.

> ❖ Opomba: Izogibajmo se podvajanju imen imenskih prostorov in razredov. Z drugimi besedami, ne kreirajmo razredov z enakim imenom kot imenski prostor. Imenski prostor lahko vsebuje večje število razredov.

Vsak razred živi v svojem imenskem prostoru. Razred **Console** npr. živi znotraj imenskega prostora **System**. To pomeni, da je njegovo polno ime **System.Console**. Da pa nam polnega imena ni potrebno pisati vsakič znova, lahko ta problem rešimo z napovedjo

```
using System;
```

Using stavke lahko napišemo na začetku našega programa, ali pa kot prve stavke znotraj našega imenskega prostora. Z uporabo teh stavkov je poimenovanje razredov bistveno krajše, koda pa bolj pregledna. Takole bi izgledal naš prvi program, če **using** stavka ne bi uporabili (v spodnjem primeru je stavek **using System** označen kot komentar).

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

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

C# nam nudi možnost pisanja lastnih imenskih prostorov – to storimo z uporabo rezervirane besede **namespace**, Na spremenljivke, objekte ali pa funkcije iz tega imenskega prostora se v tem primeru sklicujemo preko imena tega imenskega prostora in operatorja pika.

Primer:

```
namespace MojImenskiProstor //začetek novega imenskega prostora
{
   public class Test
   {
      public static int funkcija()
      {
          for (int i = 0; i < 10; i++)
          {
              Console.WriteLine("i: {0}", i);
          }
          return 0;
      }
    }
}
//vstopna točka programa
```

```
class Program
{
   static void Main(string[] args)
   {
       MojImenskiProstor.Test.funkcija();//klic funkcije imenskega prostora MojImenskiProstor
   }
}
```

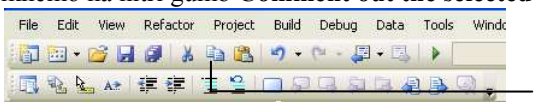
Komentarji

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

```
//Tole je enovrstični komentar;

/*

  Tole pa je večvrstični komentar!

*/
```

Visual C# pa nam ponuja še eno zelo priročno možnost, da nek del teksta, ki je že napisan, označimo kot komentar (in ga s tem npr. začasno izključimo iz prevajanja). To nam omogoča hitri gumb **Comment out the selected lines**. Tekst, ki ga želimo zakomentirati (označiti kot komentar) najprej označimo z miško, nato pa kliknemo na hitri gumb **Comment out the selected lines**, ali pa pritisnemo kombinacijo tipk **Crtl+E, C**.



Gumb **Comment out the selected lines**

Podobno lahko delu teksta, ki je označen kot komentar, oznake za komentar umaknemo. To storimo tako, da ta del teksta najprej označimo z miško, nato pa kliknemo na hitri gumb **Uncomment the selected lines**, oziroma pritisnemo kombinacijo tipk **Crtl+E, U**.

Podatkovni tipi v C#

Podatkovni tipi, ki jih uporabljamo v C# temeljijo na skupnem sistemu tipov (**Common Type System - CTS**).

Podatkovne tipe v C# lahko delimo na

- **Vgrajene** podatkovne tipe, ki jih že poznamo. To so podatkovni tipi **int, float** in **char**;
- **Uporabniško definirane** podatkovne tipe (teh še ne poznamo), kot sta npr **struct** in **class**.

Prav tako pa lahko podatkovne tipe delimo v

- **Vrednostne** podatkovne tipe – spremenljivke teh tipov shranjujejo vrednosti;
- **Referenčne** podatkovne tipe – spremenljivke teh tipov hranijo le referenco (oz. naslov) dejanskih podatkov.

Vrednostni podatkovni tipi

Vrednostne podatkovne tipe delimo na:

- **Strukturni tipi**
  - **Numerični tipi** – Numerične tipe že poznamo in jih delimo še na tri podskupine
    - **Celoštevilčni oz. integralni podatkovni tipi**

Tip	Razpon	Razlaga
sbyte	-128 do 127	Predznačeno 8 bitno celo število.
byte	0 do 255	Nepredznačeno 8 bitno celo število.
char	U+0000 do U+ffff	Unicode 16 bitni znak.
short	-32.768 do 32.767	Predznačeno 16 bitno celo število.
ushort	0 do 65.535	Nepredznačeno 16 bitno celo število.
int	-2.147.483.648 do 2.147.483.647	Predznačeno 32 bitno celo število.
uint	0 do 4.294.967.295	Nepredznačeno 32 bitno celo število
long	-9,223.372.036,854.775.808 do 9.223.372.036.854.775.807	Predznačeno 64 bitno celo število
ulong	0 to 18,446,744,073,709,551,615	Nepredznačeno 64 bitno celo število

Nekaj primerov:

```
int i = 100;
```

```
long velikoStevilo = 1000000000;

char char1 = 'Z';        // znak
char char2 = '\x0058';   // Hexadecimalen zapis
char char3 = (char)88;   // Konverzija celega števila v znak
char char4 = '\u0058';   // Unicode
```

- **Realni podatkovni tipi**

Tip	Razpon	Natančnost
float	±1.5e−45 to ±3.4e38	7 cifer
double	±5.0e−324 to ±1.7e308	15-16 cifer

Za inicializacijo spremenljivke tipa float moramo realnemu številu dodati še pripono f ali F.

❖ POZOR: Decimalno ločilo pri inicializaciji spremenljivke tipa float (pa tudi double in decimal) je **decimalna pika**!

Nekaj primerov:

```
float x = 124.80f;
float xx=50.333F;
//cela števila se avtomatsko konvertirajo v realna, oznaka f zato ni
//potrebna
float y = 500;
double z = 3900.89;
```

- **Decimalni podatkovni tip**

Tip	Razpon	Natančnost
decimal	±1.0 × 10e−28 to ±7.9 × 10e28	28-29 cifer

Če hočemo, da bo neka numerična vrednost obravnavana kot decimalna, ji moramo dodati še oznako **m** ali **M**.

Nekaj primerov:

```
decimal znesek=450.60m; //oznaka m ali M obvezna
decimal cena=345.78M;

//cela števila se avtomatsko konvertirajo v decimalna, oznaka m zato ni
//potrebna
decimal visina = 5600;
```

- Logični tip

  Spremenljivka tipa bool lahko zavzame le dve vrednosti: **true** ali **false**.

```
bool logicna=true;
```

- Uporabniško definirani strukturni tipi

- Naštevni tipi: Naštevnih tipov še ne poznamo, spoznali jih bomo v nadaljevanju.

Najpomembnejše metode posameznih podatkovnih tipov

Tip char	Razlaga
IsDigit	Metoda vrne true, če znak predstavlja cifro
IsLetter	Metoda vrne true, če znak predstavlja črko
IsLetterOrDigit	Metoda vrne true, če znak predstavlja črko ali cifro
IsLower	Metoda vrne true, če znak predstavlja malo črko abecede
IsNumber	Metoda vrne true, če znak predstavlja cifro
IsUpper	Metoda vrne true, če znak predstavlja veliko črko abecede
Parse	Pretvarjanje stringa v znak
ToLower	Pretvarjanje črke v malo črko
ToString	Pretvarjanje znaka v string
ToUpper	Pretvarjanje črke v veliko črko

O izpisovanju besedila in podatkov na zaslon bo sicer več govora v naslednjih poglavjih. Da pa si bomo rezultate primerov, ki sledijo, lahko ogledali na zaslonu, se že sedaj naučimo uporabljati metodo **Console.WriteLine()**) za izpisovanje podatkov na konzolo - zaslon. Z metodo **WriteLine** namreč lahko izpišemo poljubno število parametrov (besedilo, spremenljivke, konstante) hkrati. Najenostavnejša uporaba te metode za izpis enega podatka (podatek je lahko poljubno besedilo, ki pa mora biti v dvojnih narekovajih, podatek je lahko poljubna spremenljivka ali pa konstanta) je takale:

```
Console.WriteLine(podatki);
```

V kolikor pa želimo hkrati izpisati več podatkov lahko metodo **WriteLine** uporabimo kar takole:

```
Console.WriteLine(podatek1+podatek2+podatek3+ …);
```

V metodi **WriteLine** torej operator '+' uporabljamo za sestavljanje izpisa. Pri tem ni prav nič važno, kakšnega podatkovnega tipa so posamezni podatki, saj metoda sama poskrbi za avtomatsko konverzijo v izpisovanju!

Nekaj primerov:

```
char ch = '8';
Console.WriteLine(Char.IsDigit(ch));          //izpis True

char ch1 = '8';
Console.WriteLine(Char.IsLetter(ch1));        //izpis False

char ch2 = 'A';
Console.WriteLine(Char.IsLetterOrDigit(ch2)); //izpis False

char ch3 = 'a';
Console.WriteLine(Char.IsLower(ch3));         //Izpis: True

Console.WriteLine(Char.IsNumber('8'));        //Izpis: True

char ch4 = 'A';
Console.WriteLine(Char.IsUpper(ch4));         //izpis True

Console.WriteLine(Char.Parse("A"));           // Izpis: 'A'

char znak = Char.ToLower('A');
Console.WriteLine(znak);  //izpis a
```

```
Console.WriteLine(Char.ToLower('A'));          //izpis 'a'

char znak1 = 'a';
Console.WriteLine(znak1.ToString());          //Izpis : "a"

char zn = Char.ToUpper('x');
Console.WriteLine(zn);                         //izpis 'X'
```

Tip int	Razlaga
Parse	Pretvarjanje stringa v celo število
ToString	Pretvarjanje celega števila v string

Nekaj primerov:

```
int cifra=int.Parse("100");
Console.WriteLine(cifra);   //izpis 100

int celo=550;
string sCelo=celo.ToString();
```

Tip float	Razlaga
Parse	Pretvarjanje stringa v celo število
ToString	Pretvarjanje celega števila v string

Nekaj primerov:

```
float stevilo= float.Parse("100,67");
Console.WriteLine(stevilo);   //izpis 100,67
float znesek = 550.98f;
string sZnesek = znesek.ToString();
Console.WriteLine(sZnesek);   //izpis 550,98
```

Tip double	Razlaga
Parse	Pretvarjanje stringa v celo število
ToString	Pretvarjanje celega števila v string

Nekaj primerov:

```
double stevilo= double.Parse("100,67");
Console.WriteLine(stevilo);   //izpis 100,67
double znesek = 550.98;
string sZnesek = znesek.ToString();
Console.WriteLine(sZnesek);   //izpis 550,98
```

Tip decimal	Razlaga
Parse	Pretvarjanje stringa v decimalno število
Round	Zaokroževanje na najbližje celo število

ToByte	Pretvarjanje decimalnega števila v 8 bitno nepredznačeno celo število
ToDouble	Pretvarjanje decimalnega števila v število tipa double
ToInt16	Pretvarjanje decimalnega števila v 16 bitno predznačeno celo število
ToInt32	Pretvarjanje decimalnega števila v 32 bitno predznačeno celo število
ToInt64	Pretvarjanje decimalnega števila v 64 bitno predznačeno celo število
ToString	Pretvarjanje decimalnega števila v string
Truncate	Pretvarjanje decimalnega števila v celo število – odrežemo decimalke

Nekaj primerov:

```
decimal decimalnoStevilo = Decimal.Parse("1234,98");
Console.WriteLine(decimalnoStevilo);              //izpis 1234,98
//POZOR
decimal Stevilo = Decimal.Parse("1234.98");
Console.WriteLine(Stevilo);                       //izpis 123498

decimal znesek = 900.67m;
decimal zaokrozeno = decimal.Round(znesek);
Console.WriteLine(zaokrozeno);// izpis 901

decimal cena = 500.45m;
cena = decimal.Truncate(cena);
Console.WriteLine(cena);// izpis 500
```

Velikost pomnilnika v bytih, ki ga zasedajo spremenljivke vrednostnih podatkovnih tipov dobimo s pomočjo rezervirane besede **sizeof**.

```
Console.WriteLine("Velikost pomnilnika v bytih za vrednostne podatkovne tipe:\n");
Console.WriteLine("byte: " + sizeof(byte));//1
Console.WriteLine("sbyte: " + sizeof(sbyte));//1

Console.WriteLine("bool: " + sizeof(bool));//2
Console.WriteLine("short: "+sizeof(short));//2
Console.WriteLine("ushort: " + sizeof(ushort));//2
Console.WriteLine("char : "+sizeof(char));//2

Console.WriteLine("int  : "+sizeof(int));//4
Console.WriteLine("uint  : " + sizeof(uint));//4
Console.WriteLine("float : " + sizeof(float));//4

Console.WriteLine("long : "+sizeof(long));//8
Console.WriteLine("ulong : " + sizeof(ulong));//8
Console.WriteLine("double : " + sizeof(double));//8
Console.WriteLine("decimal : " + sizeof(decimal));//16
```

Konverzija tipov

Osnovni tipi so lahko prosto pomešani v izrazih, saj je poskrbljeno, da se opravljajo avtomatske pretvorbe (konverzije), ki ohranjajo informacijo, če je le to mogoče. Pogosto lahko spremenimo tip iz nižjega (zasede manj bitov) v višjega, problemi pa nastopijo, če gremo v obratno smer, saj v tem primeru izgubimo informacijo.

```
byte b = 3;
Console.WriteLine("byte: " + b);//izpis:3
int st = b;
Console.WriteLine("st: " + st);//avtomatska konverzija, izpis:3
```

```
char znak = 'A';
Console.WriteLine("znak: " + znak);//izpis:A
int stevilo = znak;
Console.WriteLine("stevilo: " + stevilo);//avtomatska konverzija, izpis:65

int i = 10;
Console.WriteLine("i : " + i); //izpis:10
float x = i;//avtomatska konverzija iz int v float
Console.WriteLine("x : " + x); //izpis:10

double razdalja = x; //avtomatska konverzija iz float v double
Console.WriteLine("razdalja : " + razdalja); //izpis:10
```

Eksplicitne konverzije (casting)

Marsikdaj želimo napraviti konverzijo, ki sicer ni razvidna oz. jo prevajalnik ne bi opravil. V tem primeru govorimo o **eksplicitni** konverziji. To storimo tako, da pred spremenljivko v oklepaju povemo nov tip.

```
int i1=25;
int i2=10;

float f1 =  i1 / i2;
//ker sta i1 in i2 celi števili se operator "/" obnaša kot operator za celoštevilčno deljenje
Console.WriteLine("f1 : " + f1); //rezultat 2

//če poskrbimo za eksplicitno konverzijo int v float se operator "/" obnaša kot operator za
deljenje realnih števil
f1 = (float) i1 / (float) i2;    //eksplicitna konverzija iz int v float
Console.WriteLine("f1 : " + f1); //rezultat 2,5

double cena=1240.40;
decimal znesek = (decimal)cena; //potrebna eksplicitna konverzija double v decimal
Console.WriteLine("znesek : " + znesek);
```

Vaja:

```
//S pomočjo eksplicitne konverzije tipov lahko zamenjamo vrednosti dveh spremenljivk tipa
char, brez uporabe pomožne spremenljivke!
char st1='A';
char st2='B';
Console.WriteLine("PRED ZAMENJAVO : st1 = "+st1+",  st2 = "+st2);

//izpišimo konvertirane vrednosti še pred konverzijo:
Console.WriteLine("(int)st1 = "+(int)st1+" (int)st2 = "+(int)st2);//izpis int(st1)=65
                                                                  //int(st2)=66
st1=(char)((int)st1+(int)st2);
st2=(char)((int)st1-(int)st2);
st1=(char)((int)st1-(int)st2);
Console.WriteLine("PO ZAMENJAVI : st1 = "+st1+",  st2 = "+st2);
```

Osnovni operatorji v C#

V naslednji tabeli so zbrani osnovni operatorji jezika C#. Zanje velja enaka prioriteta kot je to v matematiki, seveda pa lahko njihov vrstni red določimo s pomočjo okroglih oklepajev (tako kot v matematiki).

Operator	Razlaga

+	seštevanje
-	odštevanje
*	množenje
/	deljenje
%	modulo (ostanek pri celoštevilskem deljenju)
++	inkrement (predinkrement ali poinkrement)
--	dekrement (preddekrement ali podekrement)
?:	pogojni operator
=	prirejanje
+=	seštevanje in prirejanje
-=	odštevanje in prirejanje
*=	množenje in prirejanje
/=	Deljenje in prirejanje

Primeri:

```
int st = 10;
int st1 = 3;
Console.WriteLine(st/st1);//celoštevilsko deljenje, rezultat je 3
Console.WriteLine(st % st1);//ostanek pri celoštevilskem deljenju, rezultat je 1

int i = 1;
i++;  //poinkrement, enakovreden zapis kot i=i+1; i torej dobi vrednost 2

int j = 1; int j1 = 10;
int k = j++ + j1++;  //poinkrement: najprej se izračuna vsota, nato se vrednost
                     //spremenljivkama j in st poveča za 1
Console.WriteLine(k);//izpis 11

int n = 1; int n1 = 10;
int m = ++n + ++n1;  //predinkrement: najprej se vrednost spremenljivkama n in n1 poveča za 1,
                     //nato se izračuna vsota
Console.WriteLine(m);//izpis 13

int stevilo = 10;
stevilo += 5;//enakovreden zapis kot stevilo=stevilo+5;
Console.WriteLine(stevilo);//izpis 15
```

Naloge:

- Napiši program ki na osnovi danih robov kvadra izračuna in izpiše njegovo površino in prostornino.
- Dani sta poljubni celi števili *stevilo1* in *stevilo2*. Ugotovi in izpiši rezultat celoštevilskega deljenja teh dveh števil, ostanek pri celoštevilskem deljenju, rezultat pravega deljenja teh dveh števil, ter celo število, ki ga dobiš, če rezultatu pravega deljenja teh dveh števil odrežeš vse decimalke.
- Dane so deklaracije:

```
int n = 5;
int m = 10;
int k = 0;
```

```
int i = 7;
```

- Kakšno vrednost dobijo spremenljivke *stevilo1*, *stevilo2*, *stevilo3* in *stevilo4* po izvedbi naslednjih stavkov:

```
int stevilo1 = 2 * n++ + jm++;
int stevilo2 = --m + ++k - ++n;

k += 5;
n *= 2;
m /= 3;
i %= 4;

int stevilo3 = n++ -2 / k++ - m + i++;
int stevilo4 = 2 * (n - ++m) + 2 * (i + --j);
```

- Dano je poljubno celo število st. Ugotovi in izpiši njegove enice.

- Dane so deklaracije:

```
double x,y = 2;
int z = 3;
int a = 10;
bool c= true;
```

Kateri od izrazov NI pravilen?

```
x= y + z;
z= x - y;
c= true;
int b= z = a;
```

- Dane so deklaracije – podatki o začetnem in končnem času nekega tekmovanja. Ugotovi porabljeni čas!

```
int uraStart = 5;
int minutaStart = 45;
int sekundaStart = 33;

int uraCilj = 18;
int minutaCilj = 1;
int sekundaCilj = 11;
```

- Dana sta kota α in β nekega trikotnika, podana v stopinjah, minutah in sekundah. Izračunaj tretji kot (γ), če veš, da vsi trije koti skupaj merijo 180 stopinj!

- Dana je celoštevilska spremenljivka *stirimestno*, njena vrednost pa je neko celo štirimestno število, npr.: `int stirimestno = 1234;` Zamenjaj prvo in zadnjo cifro, za drugo in tretjo cifro pa izračunaj tretjo potenco in iz tako dobljenega števila ohrani samo enice. Primer: iz števila 1234 ustvari 4871 (1 in 4 smo zamenjali, tretja potenca druge cifre 2 je enaka 8, tretja potenca tretje cifre 3 je enaka 27, ohranimo pa le enice, to je 7).

Referenčni podatkovni tipi

Spremenljivke referenčnih podatkovnih tipov hranijo referenco na ustrezen podatek in ne podatka samega. Tega se pri delu sploh ne zavedamo, zaradi česar se uporaba teh spremenljivk ne razlikuje od spremenljivk vrednostnih tipov.

Referenčne podatkovne tipe bomo obravnavali precej kasneje, zaenkrat jih le naštejmo:

- Razred – **class**
- Vmesnik – **interface**

- Delegati – **delegate**

Med referenčne tipe pa štejemo tudi vgrajene referenčne tipe: to pa so tip **string** in tip **object**.

Podatkovni tip **string** (niz) označuje zaporedje znakov. Spremenljivko tega tipa v **C#** lahko inicializiramo tudi takole:

```
string var5;
var5 = "Lokomotiva";
```

Najpomembnejše lastnosti in metode razreda string za delo z zaporedji znakov (stringi)

Indeks	Razlaga
[indeks]	Dostop do znaka na določeni poziciji.
Lastnost	**Razlaga**
Length	Število znakov stringu.
Metoda	**Razlaga**
StartsWith(string)	Vrne logično vrednost, ki označuje, ali se nek string začenja z navedenim stringom.
EndsWith(string)	Vrne logično vrednost, ki označuje, ali se nek string končuje z navedenim stringom.
IndexOf(string[,začetni indeks])	Vrne celo število ki predstavlja pozicijo (indeks) prve pojavitve navedenega stringa v nekem stringu od začetnega indeksa naprej. Če začetni indeks ni naveden, se iskanje začne na začetku stringa. Če navedeni string ni najden, je vrnjena vrednost -1.
Insert(začetni indeks, string)	Vrne string v katerega je na navedeno mesto (začetni indeks) vrinjen navedeni string.
PadLeft(skupna_dolžina)	Vrne string, 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 string, 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 string iz katerega je odstranjeno **N** znakov od pozicije **začetni_indeks** naprej.
Replace(stariString, noviString)	Vrne string, v katerem so vse pojavitve stringa **stariString** zamenjane s stringom **noviString**.
Substring(začetni_indeks[,dolžina])	Vrne del stringa, ki se začne na navedeni poziciji in ima navedeno dolžino. Če dolžina ni navedena metoda vrne vse znake do konca stringa.
ToLower()	Vrne string v katerem so vsi znaki zamenjani z malimi znaki.
ToUpper()	Vrne string v katerem so vsi znaki zamenjani z velikimi znaki.

Trim()	Vrne string iz katerega so odstranjeni vsi vodilni in končni presledki.
Split(razmejitveni_znak)	Vrne tabelo stringov v katerem so vsi elementi deli tega stringa (besede) med seboj razmejeni z znakom **razmejitveni_znak**.

Privzeta »vrednost« referenčnih tipov spremenljivk (v primeru, da spremenljivke še niso inicializirane) je `null` - *null* NI neka vrednost, ampak je **stanje**, ki nam pove, da neka spremenljivka (ali pa objekt) nima nobene vrednosti, oz. da je vrednost neznana, neobstoječa. To ne pomeni, da ima spremenljivka vrednost 0, ali pa da je prazna, oz. v primeru stringov da gre za t.i. prazen string, pa tudi obnaša se ne tako kot katerakoli izmed naštetih vrednosti.
Za razliko od referenčnih spremenljivk, pa vrednostne spremenljivke **ne morejo** nikoli imeti »vrednost« *null*.

Nekaj primerov:

```
//dostop do posameznega znaka v stringu
string znaki = "abcdefg";
char a=znaki[0];     // 'a'
char b=znaki[1];     // 'b'

string beseda="Danes je lep dan!";
//sestavimo nov niz iz nekaterih znakov niza beseda
string znakiInPresledki = beseda[0]+ beseda[3]+ beseda[4]+ beseda[14];
//znakiInPresledki dobi vrednost "Desa"

//Uporaba metod StartsWith in EndsWith
bool zacneZabc = znaki.StartsWith("abc");   // true
bool koncaZabc = znaki.EndsWith("abc");     // false

//Uporaba metode IndexOf
string sola = "Tehniški Šolski Center Kranj";
int index1 = sola.IndexOf(" ");        // 8, ker se string " " pojavi prvič na osmem mestu
int index2 = sola.IndexOf(' ');        // 8, ker se znak ' ' pojavi prvič na osmem mestu
int index3 = sola.IndexOf("Center");   // 16, ker se string "Center" pojavi prvič na 16 mestu
int index4 = sola.LastIndexOf(" ");    // 22, ker se string " " pojavi zadnjič na 22 mestu

//Uporaba metod Remove, Insert in Replace
sola = sola.Remove(0, 9);                       // Šolski center Kranj
sola = sola.Insert(sola.Length, ", Slovenija"); // Šolski center Kranj, Slovenija
sola = sola.Replace("Slovenija", "4000 Kranj"); // Šolski center Kranj, 4000 Kranj

//Uporaba metod Substring, ToUpper in ToLower
string ime = "aNJA";
string prvaCrka = ime.Substring(0, 1).ToUpper(); // A
string drugeCrke = ime.Substring(1).ToLower();  // nja
ime = prvaCrka + drugeCrke;                     // Anja

//Kopiranje enga stringa v drug string
string s1 = "abc";
string s2 = s1;       // string s2 dobi vrednost strinfa s1, torej s2 postane "abc"
s2 = "def";           // string s2 postane "def", string s1 pa se ne spremeni
string s3 = s1 + s2;  // strig s3 dobi vrednost "abcdef"

//Uporaba metode Substring
string polnoIme = "  Edward C Koop    ";     // "  Edward C Koop    "
polnoIme = polnoIme.Trim();                  // "Edward C Koop"
int prvipresledek = polnoIme.IndexOf(" ");   // 6
string lastnoIme = polnoIme.Substring(0,prvipresledek); // "Edward"

//Uporaba razmejitvenih znakov v stringu
string naslov = " |34 Kališka ulica|Slovenija|Kranj|4000 ";
naslov = naslov.Trim();        // "|34 Kališka ulica|Slovenija|Kranj|4000"
naslov.Remove(0,1);            //iz naslova odstranimo prvi znak
string naslov1= naslov.Remove(naslov.Length-1,1);// iz naslova odstranimo zadnji znak

int indeksUlice = naslov.IndexOf("|") + 1;                                    // 1
int indeksDrzave = naslov.IndexOf("|", indeksUlice) + 1;                      // 18
```

```
int indeksKraja = naslov.IndexOf("|", indeksDrzave) + 1;                    // 28
int indeksPoste = naslov.IndexOf("|", indeksKraja) + 1;                     // 34

string ulica = naslov.Substring(0, indeksDrzave-1);                      // 34 Kališka ulica
string mesto = naslov.Substring(indeksKraja, indeksPoste - indeksKraja - 1);  // Kranj
string posta = naslov.Substring(indeksPoste);                               // 4000

//Uporaba metode Split
string pisatelj = " Edward C Koop  ";   // " Edward C Koop  "
pisatelj = pisatelj.Trim();             // "Edward C Koop"
string[] imena = pisatelj.Split(' ');
string imePisatelja = imena[0];         // "Edward"
string imePisatelja1 = imena[1];        // "C"
string imePisatelja2 = imena[2];        // "Koop"

//Uporaba metode Insert
string telefonskaStevilka = "041827919";                  // "041827919"
telefonskaStevilka = telefonskaStevilka.Insert(3, "-");   // "041-827919"
telefonskaStevilka = telefonskaStevilka.Insert(7, "-");   // "041-827-919"

//Uporaba metode Replace
string datum = "21-08-2007";                  // "21-08-2007"
datum = datum.Replace("-", "/");
```

Naloge:

- Dana je deklaracija `string stavek = "moj stavek";` S pomočjo lastnosti **Length** ter metode **ToUpper** spremeni prvo in zadnjo črko tega stringa v veliko črko. Prvo in zadnjo črko tega stringa spremeni v veliko črko.

- Ugotovi na katerem mestu stringa *stavek* se nahaja prva pojavitev veznika in (uporabi metodo **IndexOf**)?

- Iz stringa *stavek* odstrani vse znake od 10. znaka naprej! Uporabi metodo **Remove**!

- Dana je spremenljivka *stavek* tipa string, spremenljivka ima že neko vrednost (vsebuje več kot 10 znakov).
  - Odstrani vse vodilne in končne presledke;
  - Vse črke v stringu *stavek* spremeni v velike črke angleške abecede;
  - Ugotovi in izpiši prvi in zadnji znak tega stringa;
  - Ugotovi in izpiši koliko znakov je v tem stringu;
  - V spremenljivko *beseda* (string) zapiši prvih pet znakov stringa *stavek*;
  - Zadnje tri znake tega stringa nadomesti s pikicami;
  - Na sredino stringa *stavek* vrini tri podčrtaje;
  - Iz stringa *stavek* odstrani vse znake od šestega znaka naprej.

- V stringu ***stavek*** vse številke nadomesti z besednim opisom (številko 1 zamenjaj z ena, številko 2 z dva, …)

- Dane so trije stringi *st1*, *st2* in *st3*, ki so že inicializirani. Deklariraj string *st4*, ki naj bo sestavljen iz zadnjih znakov stringov *st1*, *st2* in *st3*!

Pretvarjanje osnovnih podatkovnih tipov

Pri delu s spremenljivkami se velikokrat srečamo s potrebo po pretvarjanju osnovnih podatkovnih tipov (npr. zaporedja znakov v celo število ali realno število, znaka v cifro, celo število v string, …). Najpogosteje za pretvarjanje uporabljamo razred **Convert**, ki vsebuje celo vrsto metod za pretvarjanje. Omenimo le nekaj najpomembnejših:

- **ToInt32** – pretvorba določene vrednosti v 32 bitno predznačeno (**signed**) celo število
- **ToInt64** – pretvorba določene vrednosti v 64 bitno predznačeno (**signed**) celo število
- **ToByte** – pretvorba v 8-bitno nepredznačeno celo število
- **ToDouble** – pretvorba določene vrednosti v vrednost tipa **double**
- **ToDecimal** – pretvorba v decimalno število
- **ToSingle** – pretvorba v spremenljivko tipa **float**
- **ToChar** – pretvorba v znakovno vrednost
- **ToBoolean** – pretvorba v logično vrednost
- **ToString** – pretvorba v zaporedje znakov - **string**

Primeri uporabe:

```
int celo_stevilo = Convert.ToInt32("12345");
long veliko_celo = Convert.ToInt64("123456789");
float realno = Convert.ToSingle("12.89");//namesto decimalne pike lahko pišemo vejico!!!
double veliko_stevilo = Convert.ToDouble("250000.89");
char znak = Convert.ToChar(65);
bool logicna = Convert.ToBoolean(1);
double vsota = 100.22 + 200.33;
string stavek = Convert.ToString(vsota);
```

Standardne oznake/kode za formatiranje števil

V naslednji tabeli so prikazane standardne kode (oznake - šifre), ki jih uporabimo za formatiranje števil, kadar jih želimo z metodo **ToString** spremeniti v zaporedje znakov – string. Katerokoli od teh kod (znakov) lahko uporabimo znotraj oklepajev metode ToString, obvezno pa jih moramo zapisati med dvojna narekovaja.

Oznaka	Format	Opis
C ali c	Denarna valuta	Formatiranje števila kot denarno valuto z določenim številom decimalnih mest.
P ali p	Procent	Formatiranje števila kot procent, z določenim številom decimalnih mest.
N ali n	Številka	Formatiranje števila z ločilom za tisočice in določenim število decimalnih mest.
F ali f	Float	Formatiranje števila kot decimalnega števila, z določenim število decimalnih mest.
D ali d	Cifre	Formatiranje celega števila z določenim število cifer.
E ali e	Eksponent	Formatiranje števila v eksponentni obliki z določenim število decimalnih mest.
G ali g	Splošno	Formatiranje števila v decimalni ali eksponentni obliki odvisno od tega, katera oblika je bolj primerna.

Nekaj primerov:

```
decimal znesek=1547.20m;
string  noviznesek=znesek.ToString("c"); //Rezultat: €1.547,20

decimal obresti=0.023m;
string obrestnamera=obresti.ToString("p1");//v stringu bo zapisano eno decimalno mesto,
                                           //Rezultat: 2,3%

float vrednost = 15000;
string novavrednost = vrednost.ToString("n0");//v stringu bo zapisano 0 decimalnih mest
                                              //Rezultat: 15.000

decimal skupaj=432818.678m;
```

```
string vseskupaj = skupaj.ToString("f2");//v stringu bosta zapisani 2 decimalni mesti, število
                                         //bo zaokroženo. Rezultat: 432818,68
decimal cena = 145345.23m;
string novacena = vrednost.ToString("n2");//izpis bo formatiran na tisočice, v stringu bosta
                                          //zapisani 2 decimalni mesti. Rezultat: 145.345,23
```

Pri formatiranju števil si lahko pomagamo tudi z metodo **Format** razred **String** (pozor, **String** z veliko začetnico!!!).

Nekaj primerov;

```
//Z metodo Format lahko formatiramo lahko več števil hkrati. Zapis 0:c pomeni, da se bo prvo
število (oznaka 0) formatiralo s šifro c
string novacena = String.Format("{0:c}", 1547.2m);    //Rezultat 1.547,20 €

//Zapis 0:c pomeni, da se bo prvo število (oznaka 0) formatiralo s šifro c, zapis 1:p pa
//pomeni, da se bo drugoo število (oznaka 1) formatiralo s šifro p - torej kot procent
//Rezultat naslednjega formata: Znesek: 1.547,20 €,  Procent: 12,00%
string novacena = String.Format("Znesek: {0:c},  Procent: {1:p}", 1547.2m,0.12);

string obrestnamera = String.Format("{0:p1}", .023m); //Rezultat 2,3%
string novavrednost=String.Format("{0:n0}",15000);    //Rezultat 15.000
string vseskupaj = String.Format("{0:f3}", 432.8175); //Rezultat 432,818

//format {0,30} pomeni desno poravnavo teksta, format {0,-30} pa levo poravnavo
string stavek=string.Format("{0,30}{1,9:f1}",beseda,hitrost); //leva poravnava
string stavek1 = string.Format("{0,-30}{1,-9:f1}", beseda, hitrost); //desna poravnava
```

Običajne (custom) oznake/kode za formatiranje števil

Če nam standardno formatiranje števil ne zadošča za oblikovanje formata, ki ga želimo, lahko kreiramo svoj lasten format. Pri takem formatiranju uporabimo običajne (**custom**) šifre za formatiranje.

Tako v metodi **ToString**, kot v metodi **Format** lahko z uporabo treh sekcij, ki so med seboj ločene z znakom podpičje določimo, kako naj bo formatirano pozitivno število, kako negativno število in kako naj bo formatirano število 0.

Seznam običajnih oznak (kod) za formatiranje števil;

Oznaka	Pomen
0	Mesto rezervirano za cifro 0.
#	Mesto rezervirano za cifro.
.	Decimalna pika.
,	Decimalno ločilo.
%	Mesto rezervirano za znak procent.
;	Ločilo posamezne sekcije.

```
decimal znesek = -1547.20m; //negativno število

string znesekS=znesek.ToString("€#,##0.00"); //Rezultat -€1.547,20
string znesekS1=znesek.ToString(" €#,##0.00"); //Rezultat - €1.547,20

znesek = 1547.20m; //pozitivno število
```

```
//PRIKAZ SEKCIJ v formatiranju
//če je število, ki ga formatiramo poztivno, se upošteva prvi format (prva sekcija), če je
//negativno druga sekcija, cicer pa tretja sekcija. Ločilo za sekcije je znak ";";

string znesekS1 = znesek.ToString("€#,##0.00;(-#,##0.00);Nič");//Rezultat: €1.547,20
string znesekS2 = String.Format("{0:€#,##0.00;(-#,##0.00);Nič}",znesek); //Rezultat: €1.547,20

decimal znesek1 = -1547.20m; //negativno število

string znesekS3 = znesek1.ToString("€#,##0.00;(-#,##0.00);Nič");//Rezultat: (-1.547,20)
string znesekS4 = String.Format("{0:€#,##0.00;(-#,##0.00);Nič}", znesek1);
//Rezultat: (-1.547,20)

//Rezultat naslednjega formata: Znesek: €1.547,20,  Procent: 12,00%
string novacena1 = String.Format("Znesek: {0:€#,##0.00},  Procent: {1:p}", 1547.2m, 0.12);

decimal znesek2 = 0;
string znesekS5 = znesek2.ToString("€#,##0.00;(#,##0.00);Nič"); //Rezultat: Nič
```

Naloge:

- Dane so deklaracije

  ```
  string ulica1 = "Trubarjeva ulica";
  string ulica2 = "Mali trg";
  int st1 = 12,st2 = 2390;
  ```

  Formatiraj stringa *naslov1* in *naslov2* tako , da bo ulica formatirana na 30 mest, hišna številka pa na 5 mest! Ulica naj bo poravnana levo, hišna številka pa desno!

- Dani sta deklaraciji:

  ```
  string stevilo1 = "200";
  string stevilo2 = "300";
  ```

  Oba podatka pretvori v celoštevilski vrednosti in izračunaj njuno vsoto

- Dane so spremenljivke:

  ```
  string ime="Andrej";
  int starost=20;
  double velikost=178.45;
  ```

  Formatiraj izpis tako, da dobiš na zaslonu takle izpis:

  ```
  Naziv         starost          velikost
  ---------------------------------------------
  Andrej            20            178.45
  ```

- Kakšno vrednost dobi spremenljivka *stavek* po izvedbi naslednjih stavkov?

  ```
  string jezero = "Bohinjsko jezero";
  double razdalja=178.45124;
  stavek = string.Format("{0,-30}{1,9:f2}", jezero, razdalja);
  ```

- Kakšno vrednost dobita spremenljivka *znesek1* in *znesek2* po izvedbi naslednjih stavkov?

  ```
  decimal znesek = -1547.20m; //pozitivno število

  string znesek1 = znesek.ToString("#,##0.00");//Rezultat: €1.547,20
  string znesek2 = znesek.ToString("#,##0.00;(-#,##0.00)");
  ```

- Kakšno vrednost dobi spremenljivka *obvestilo* po izvedbi naslednjih stavkov?

```
int st = 15;
int st1 = 6;
decimal procent = .15m;
string obvestilo = String.Format("Od {0,3} izdelkov se jih je kar {1,3} podražilo za
                                  več kot {2:p1}!",st,st1,procent);
```

Vhodni in izhodni stavki (branje in izpis podatkov)

Branje (**Console.ReadLine()**) in izpisovanje (**Console.WriteLine()**) podatkov je v C# izvedeno z branjem in zajemanjem nizov – stringov. Že v prejšnjih primerih smo prikazali nekaj preprostih izhodnih stavkov – izpisovanje rezultatov na konzolo (ekran).

Primer:

```
//izpis teksta, na koncu pa se doda oznaka za prehod v novo vrstico
Console.WriteLine("Pozdravljen C#");
Console.Write("Pozdravljen C#"); //izpis brez dodanega znaka za prehod v novo vrstico
```

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 metode razred **Convert**. Največkrat bomo uporabljali slednje.

Primer:

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

//če želimo prebrati spremenljivko tipa int, se moramo zavedati, da metoda ReadLine vrne
//string, ki ga moramo spremeniti v int
Console.Write("Vnesi celo število: ");
int stevilo = Convert.ToInt32(Console.ReadLine());

//ali pa

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

Z metodo **WriteLine** lahko izpisujemo tudi več parametrov (spremenljivk, konstant) hkrati. V tem primeru moramo znotraj prvega parametra (ki je neko zaporedje znakov – **string**) za vsak dodatni parameter (spremenljivko) napovedati njegov položaj znotraj izpisa in njegovo zaporedno številko (to storimo s parom znakov **{}**, znotraj oklepajev pa zaporedna številka parametra).

Primeri:

```
decimal znesek = 1297.86m;
Console.WriteLine("Izpis decimalnega števila: {0}", znesek);              //Izpis: 1297,86

decimal procent = 14.5m;
//Rezultat deljenja ni zaokrožen
Console.WriteLine("14.5% od 1297.86 = {0}",(znesek / 100) * procent);
//Izpis: 14.5% od 1297.86 = 188,18970

//Naslednji rezultat v izpisu je zaokrožen na 2 decimalki
Console.WriteLine("14.5% od 1297.86 = {0:F2}",(znesek / 100) * procent);//Izpis na 2 decimalki


//Rezultat zaokrožimo s pomočjo oznake za format
Console.WriteLine("14.5% od 1297.86 = {0:F2}",(znesek / 100) * procent);
//Izpis: 14.5% od 1297.86 = 188,18970

//Rezultat zaokrožimo s pomočjo metode Round razreda Math
decimal rezultat=Math.Round((znesek / 100) * procent,2);
Console.WriteLine("14.5 % od 1297.86 = {0}", rezultat);
//Izpis: 14.5% od 1297.86 = 188,18970

//ali pa kat takole
```

```
Console.WriteLine("14.5% od 1297.86 = {0}", Math.Round((znesek/100) * procent,2));
//Izpis: 14.5% od 1297.86 = 188,18970

decimal znesek = 100.55m;
decimal znesek1 = 166.5m;
decimal znesek2 = 1600.67m;
//izpis formatiran na 15 mest za vsako spremenljivko
Console.WriteLine("{0,15}{1,15}{2,15}",znesek,znesek1,znesek2);
```

V enem od zgornjih primerov smo uporabili metodo razreda **Math**. Razred **Math** vsebuje številne metode za uporabo matematičnih funkcij. Nekatere od njih so zbrane v spodnji tabeli:

Lastnost	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.
Ceiling	Najmanjše celo število večje ali enako od števila, ki nastopa kot argument metode.
Cos	Kosinus kota.
Exp	Eksponentna funkcija (eksponent je iracionalno število e).
Floor	Največje celo število manjše ali enako od števila, ki nastopa kot argument metode.
Log	Logaritem.
Log10	Desetiški logaritem.
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:

```
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

decimal stevilo = 6.45327m;
decimal cstevilo = Math.Ceiling(stevilo);
Console.WriteLine(cstevilo);  //izpis 7

double gama = Math.PI;
double cosgama = Math.Cos(gama);
Console.WriteLine(cosgama);  //izpis -1

decimal stevilo1 = 6.45327m;
decimal cstevilo1 = Math.Floor(stevilo1);
Console.WriteLine(cstevilo1);  //izpis 6

double st = 100;
double log10st = Math.Log10(st);
Console.WriteLine(log10st);  //izpis 2

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);
Console.WriteLine(potenca);  //izpis 125

decimal decimalno = 245.67843m;
Console.WriteLine(Math.Round(decimalno)); //izpis 246
int celo = Convert.ToInt32(Math.Round(decimalno));
Console.WriteLine(celo);  //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

decimal velikost = 165.45m;
int vel = Convert.ToInt32(Math.Truncate(velikost));
Console.WriteLine(vel); //izpis 165
```

Naloge:

- Dana je poljubna spremenljivka tipa decimal, ki je že inicializirana
  - kolikšen je njen celi del
  - kolikšen je njen decimalni del
  - zaokroži jo na dve decimalki natančno
  - izračunaj in izpiši njen kvadratni koren
  - izračunaj in izpiši četrto potenco tega števila
- Dani sta kateti pravokotnega trikotnika. Izračunaj hipotenuzo, in notranje kote tega trikotnika.
- Zapiši v C# naslednji matematični izraz

$$x = \frac{(2a-b)^2 + \sqrt{a3 + b2}}{3xy^3}$$

- Premer kroga znaša 10 cm. Ugotovi obseg in ploščino kroga. Kakšna tetiva pripada središčnemu kotu 33 stopinj? Rezultat zaokroži na dve decimalki natančno!

- V pravokotnem koordinatnem sistemu je točka A(5,12). Ugotovi njeno razdaljo od koordinatnega izhodišča! Rezultat zaokroži na tri decimalke!

- Dani sta stranici romboida in njun vmesni kot podan v stopinjah. Izračunaj obseg, ploščino ter obe diagonali tega romboida.

- Napiši program, ki bo za poljubno količino mleka, podanega v litrih (1 liter je enak enemu kubičnemu decimetru) izračunal in izpisal količino potrebne embalaže oblike tetrapak (dimenzije ene embalaže so: dolžina 1 dm, širina 0.5 dm in višina 2 dm !). Izpis ustrezno oblikuj(formatiraj)!

- Dana je premica $y = 8x + 16$. Ugotovi in izpiši:

  - pod kakšnim kotom seka abscisno os
  - koordinati presečišč z obema osema
  - ploščino pravokotnega trikotnika, ki jo tvori premica z obema osema

Krmilni stavki v C#

Stavek if

Osnovni krmilni stavek v **C#** je **if** stavek. Uporabljamo ga tedaj, ko želimo programski tok razvejiti na dve veji. Poznamo tri oblike tega stavka:

- pogojna izvršitev,
- razvejitev in
- gnezdena oblika.

Pogojna izvršitev

```
if (pogoj P) // pogoj mora biti vedno v oklepajih
{
   Stavek1; //poljuben stavek
   Stavek2; //poljuben stavek
}
```

Stavek **S** se izvede v primeru, da je pogoj **P** izpolnjen, sicer pa se program nadaljuje za **if** stavkom. V primeru, da je znotraj if stavka en sam stavek, lahko zavite oklepaje izpustimo:

```
if (pogoj P) // pogoj mora biti vedno v oklepajih
   Stavek1; //poljuben stavek
```

Razvejitev

```
if (pogoj P)
{
   stavek S1;  //poljuben stavek
}
else
{
   Stavek S2;  //poljuben stavek
}
```

Stavek **S1** se izvede v primeru, da je pogoj **P** izpolnjen, sicer pa se izvede stavek **S2**.

V pogojih lahko uporabljamo relacijske operatorje:

Relacijski Operator	Pomen
<	Manjše.
>	Večje.
==	Enako.

<=	Manjše ali enako.
>=	Večje ali enako.
!=	Različno.

Pogoji so lahko tudi sestavljeni. Sestavljamo jih lahko s pomočjo logičnih operatorjev.

Logični Operator	Pomen
&&	Logični in (**and**).
||	Logični ali (**or**).
!	Negacija (**not**).

Če je združenih več pogojev lahko uporabimo oklepaje. Vrstni red operacij je takšen, kot to velja pri matematiki in ga določajo oklepaji.

Gnezdeni if stavek

Gnezdeni **if** stavek je **if** stavek znotraj **if** stavka. Pri takih stavkih moramo biti pozorni na to, v katerem primeru se izvede **else** veja takega gnezdenega stavka. Pomagamo si seveda z oklepaji **{}**.

```
int stevilo = Convert.ToInt32(Console.ReadLine());
if (stevilo > 10)
{
    if (stevilo % 2 == 0)
       Console.WriteLine("Sodo število večje od 10");
    else
       Console.WriteLine("Liho število večje od 10");
}
else
{
    if (stevilo % 2 == 0)
       Console.WriteLine("Sodo število manjše od 10");
    else
       Console.WriteLine("Liho število manjše od 10");
}
```

Vaja:

```
/* Preberi dvomestno število in izpiši njegove desetice in enice. Če vnešeno število
ni dvomestno, naj program izpiše obvestilo*/

Console.Write("Vnesi število : ");
int stevilo=Convert.ToInt32(Console.ReadLine());
if (stevilo>=10 && stevilo <=100)
{
   Console.WriteLine("desetice : {0}",stevilo/10);
   Console.WriteLine("enice : {0}",stevilo %10); // % = ostanek pri celoštevilčnem deljenju
}
Else
 Console.WriteLine("Število ni v dogovorjenih mejah!");
```

Vaja:

```
/*Program naj zahteva vnos starosti neke osebe, nato pa izpiše za kakšno vrsto osebe gre in
sicer
```

```
           do 2 leti  Dojenček
           3-10 let - Mladoletnik
           11-19 let - Najstnik
           20 in več - Odrasla oseba*/

Console.Write("Starost osebe v letih: ");
int starost=Convert.ToInt32(Console.ReadLine());
if (starost<=2)
   Console.WriteLine("Dojenček!");
else if (starost>=3 && starost<=10)
   Console.WriteLine("Mladoletnik!");
else if (starost>=11 && starost<=19)
   Console.WriteLine("Najstnik!");
else Console.WriteLine("Odrasla oseba!");
```

Vaja:

```
//REŠEVANJE TRIKOTNIKA

Console.Write("Vnesi stranico a : ");
double a = Convert.ToDouble(Console.ReadLine());
Console.Write("Vnesi stranico b : ");
double b = Convert.ToDouble(Console.ReadLine());
Console.Write("Vnesi stranico c : ");
double c = Convert.ToDouble(Console.ReadLine());

if (a + b < c || a + c < b || b + c < a)  // logicni ali zapisemo v C# takole: ||
    Console.WriteLine("Tak trikotnik ne obstaja!");
else
{
    double s = (a + b + c) / 2;
    double plosc = Math.Sqrt(s * (s - a) * (s - b) * (s - c));
    double vc = 2 * plosc / c;
    double va = 2 * plosc / a;
    double vb = 2 * plosc / b;
    //funkcija asin je obratna/inverzna funkcija funkcije sinus
    double alfa = Math.Asin(vc / b) * 180 / Math.PI;
    double beta = Math.Asin(vc / a) * 180 / Math.PI;
    double gama = 180 - alfa - beta;
    double R = a / (2 * Math.Asin(vc / b));
    double r = a * b * c / (4 * plosc);
    Console.WriteLine("Obseg trikotnika       : {0:F}", 2 * s);
    Console.WriteLine("Ploščina trikotnika    : {0:F} ",plosc);
    Console.WriteLine("Višina na stranico c   : {0:F}",vc);
    Console.WriteLine("Višina na stranico a   : {0:F} ", va);
    Console.WriteLine("Višina na stranico b   : {0:F}", vb);
    Console.WriteLine("Kot alfa               : {0:F}", alfa);
    Console.WriteLine("Kot beta               : {0:F}", beta);
    Console.WriteLine("Kot gama               : {0:F} ", gama);
    Console.WriteLine("Polmer včrtanega kroga : {0:F} ", R);
    Console.WriteLine("Polmer očrtanega kroga : {0:F} ", r);
}
```

Naloge:

- Napiši program, ki zahteva vnos stranic trikotnika in ugotovi, ali tak trikotnik sploh obstaja, ali je trikotnik pravokoten, ali je trikotnik enakokrak in ali je trikotnik mogoče enakostraničen.
- Napišim program, ki bo pomagal ugotoviti, ali bo na določeno leto prestopno ali ne. Leto je prestopno, kadar je deljivo s 4 in ne s 100 ali kadar je deljivo s 400.
- Ugotovi pravilnost oz. nepravilnost naslednjega pogoja, če je x = 5

  ((3 > x) || (5 <= x)) && (x! = 8)

- Kakšna je vrednost spremenljivke N po izvedbi naslednjega **if** stavka?

```
int N = 1;
bool B = true;
if ((N < 5) && B)
{
N = N + 1;
}
else
{
N = 0;
}
```

- Preberi poljubno celo število. Ugotovi in izpiši, ali je sodo ali liho!

- Šola organizira izlet za učence na šoli. Cena izleta je 450 EUR, če se prijavi do vključno 30 učencev. Vsaka naslednja prijava zniža ceno na posameznika za 1%. Če se prijavi manj kot 20 učencev izlet ne bo organiziran. Napiši program, ki bo zahteval vnos števila učencev in nato izračunal in izpisal ceno izleta na posameznega učenca, oziroma, da je premalo prijav.

- Sestavi program, ki ugotovi, ali je poljubno trimestno število deljivo s svojo srednjo cifro.

- Na transakcijskem računu imaš določen znesek, dovoljen limit je 500 EUR. Ker ti je zmanjkalo denarja, želiš dvigniti določen znesek. Če limit ni presežen, bankomat izplača izbrani znesek. Napiši 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 naj program prebere.

- Indeks staranja prebivalstva izračunamo tako, da število prebivalcev starejših od 60 let delimo s številom mladih do 20 in količnik pomnožimo s 100. Na podlagi rezultatov lahko ugotovimo tip starostne strukture. Napiši program, ki bo prebral število mladih do 20 let in starejših od 60 let. Na podlagi teh podatkov izračunaj indeks staranja in s pomočjo pogojnih stavkov določi starostni tip prebivalstva ter ga izpiši.

- Napiši program, ki poišče rešitev kvadratne enačbe.

- Realiziraj naslednjo funkcijo :

$$f(x) = \begin{cases} -3 & ; x < -3 \\ -3*x+2 & ; -3 <= x < 0 \\ 7 & ; x = 0 \\ 3*x - 2 & ; 0 < x < 4 \\ 3 & ; x >= 4 \end{cases}$$

Stavek switch

V primeru, ko želimo program razvejiti na več vej, uporabimo **switch** stavek.

Sintaksa:

```
switch (spremenljivka)
{
   case vrednost1: stavek1; break;
   case vrednost2: stavek2; break;
   case vrednost3: stavek3; break;
   .
   .
   case vrednostN: stavekN; break;

   default:
      stavki;
}
```

Stavki, ki so zapisani v veji **default** se izvedejo le v primeru, da ni bil izveden noben **case** stavek. Za razliko od **switch** stavka v **C++**, v **C#** ni dovoljen impliciten prehod iz ene **case** veje v drugo, razen če je veja prazna.

Primer:

```
//ker smo v prvi veji pozabili break stavek, pride pri prevajanju do napake
int n = 2;
switch (n)
{
    case 1: Console.WriteLine("Vrednost n je enaka  1!"); //NAPAKA - manjka break stavek!!!
    case 3:
       Console.WriteLine("Vrednost n je enaka 3.");
       break;
    default:
       Console.WriteLine("Število n je različno od 1 ali 3!");
    break;
}
```

V primeru, da so posamezne veje **switch** stavka prazne, se izvedejo stavki v prvi veji, ki vsebuje stavek **break**.

```
//ker sta prvi dve veji prazni, dobimo na ekrau izpis:  Vrednost n je enaka  1, 2, ali 3.
int n = 2;
switch (n)
{
    case 1:
    case 2:
    case 3:
       Console.WriteLine("Vrednost n je enaka  1, 2, ali 3.");
       break;
    default:
       Console.WriteLine("Število n je različno od 1, 2 ali 3!");
       break; //break na tem mestu OBVEZEN
}
```

V **switch** stavku lahko testiramo tudi poljuben string, npr.:

```
string beseda = Console.ReadLine();
switch (beseda)
{
    case "Miza": Console.WriteLine(beseda);//če je vnesena beseda enaka besedi "Miza" sledi
                                           //izpis!
                 break;
```

```
    . . .

}
```

Vaja:

```
/* Dan je poljuben datum. Izračunali bi radi, na kateri dan v tednu pade. Čee gre za datum v
obdobju do leta 4000, laho uporabimo  t.i. Zellerjevo formulo :
    x:= [( 13*mes - 1)/5 ] + dan + [ 5*leto/4 ] + [ 21*stol/4 ]
    kjer je:
    [ a ]    celi del števila a.
    dan  ... dan v mesecu
    mes  ... mesec - 2, če mesec ni januar ali februar in
    mesec + 10, če je januar ali februar;  v tem primeru moramo leto zmanjšati za 1
    leto  ... zadnji dve cifri leta
    stol  ... prvi dve cifri leta

    Dan v tednu nam da ostanek pri deljenju s 7 :

        0 : nedelja
        1 : ponedeljek
        2 : torek
        3 : sreda
        4 : četrtek
        5 : petek
        6 : sobota*/

int dan,mesec,mes,letnik,leto,stol,x;

//preberemo datum
Console.Write("Dan : ");
dan=int.Parse(Console.ReadLine());
Console.Write("Mesec : ");
mesec=int.Parse(Console.ReadLine());
Console.Write("Leto : ");
letnik=int.Parse(Console.ReadLine());
Console.WriteLine("Datum: "+dan+"."+mesec+"."+letnik);

//določimo mes, leto in stol po Zellerjevem pravilu
if (mesec > 2)
  mes= mesec - 2;
else
{
  mes= mesec + 10;
  leto= letnik - 1;
}
leto= letnik % 100;
stol= letnik / 100;
//uporabimo Zellerjevo formulo
x= ( 13 * mes -1 ) / 5 + dan + ( 5 * leto ) / 4 + ( 21 * stol ) / 4;
x= x % 7;
Console.Write("Dan: ");
//iz ostanka pri deljenju s 7 ugotovimo za kateri dan v tednu gre
switch (x % 7 )
{
  case 0:Console.WriteLine("Nedelja");
         break;
  case 1:Console.WriteLine("Ponedeljek");
         break;
  case 2:Console.WriteLine("Torek");
         break;
  case 3:Console.WriteLine("Sreda");
         break;
  case 4:Console.WriteLine("Četrtek");
         break;
  case 5:Console.WriteLine("Petek");
         break;
  default:Console.WriteLine("Sobota");
         break;
}
Console.ReadKey();
```

Vaja:

```
//Program namenejn kavnemu avtomatu.

Console.WriteLine("Vrsta kave: 1=Mala 2=Srednja 3=Velika");
Console.Write("Vnesi ustrezno številko: ");
string s = Console.ReadLine();
int n = Convert.ToInt16(s);
int cost = 0;
switch (n)
{
    case 1:
        cost += 50;
        break;
    case 2:
        cost += 25;
        goto case 1;
    case 3:
        cost += 50;
        goto case 1;
    default:
        Console.WriteLine("Napačen vnos. Izberi  1, 2, al 3.");
        break;
}
if (cost != 0)
{
    Console.WriteLine("Vstavite prosim {0} centov.", cost);
}
Console.WriteLine("Hvala!");
```

Naloge:

- Program naj prebere poljubno cifro in jo izpiše z besedo. V primeru napačnega vnosa naj se izpiše ustrezno obvestilo.

- Napiši program, ki zna računati obseg in ploščino geometrijskih likov. Uporabniku ponudi nabor geometrijskih likov (1 - kvadrat, 2 - pravokotnik, 3 - pravokotni trikotnik in 4 - krog). S pomočjo **switch** stavka ugotovi uporabnikovo izbiro, nato pa v odvisnosti od izbranega lika zahtevaj vnos ustreznih podatkov.

- Z uporabo **switch** stavka kreiraj izpis:

  ocena 1 pomeni nezadostno
  ocena 2 pomeni zadostno
  ocena 3 pomeni dobro
  ocena 4 pomeni prav dobro
  ocena 5 pomeni odlično

- Preberi poljuben stavek dolžine pet znakov. Kreiraj nov stavek *sifriranstavek*, v katerem boš s pomočjo **switch** stavka vse samoglasnike nadomestil z njihovo ASCII kodo!

- Preberi poljuben stavek, nato pa s pomočjo **switch** stavka ugotovi, ali je prvi znak tega stavka samoglasnik, številka, ali pa kaj tretjega!

- Napiši program, ki pretvarja poljuben znesek podan v EUR v drugo valuto, npr. USD in obratno. Na ekranu naj bo sporočilo:

  Pretvarjanje v drugo valuto:
  A – EUR v USD, B – USD v EUR
  Izberi: _

  Po uporabnikovi odločitvi zahtevaj vnos zneska za menjavo in ustrezen menjalniški tečaj, nato pa s pomočjo **switch** stavka (upoštevaj vnos velikih in malih črk!) izračunaj in izpiši rezultat.

- Preberi poljuben stavek, nato pa s pomočjo **switch** stavka ugotovi, koliko je v tem stavku pik, vejic in dvopičij!

- Naslednje zaporedje stavkov spremeni tako, da namesto **if** stavka uporabiš **switch** stavek: (spremenljivka a je tipa **int**) :

```
if  (a == 3)
   Console.WriteLine("a =  3");
else if (a>5 && a<10)
   Console.WriteLine ("a med 5 in 10");
else Console.WriteLine("a je napačen");
```

- Napiši program, ki bo prebral poljubno celo število in ugotovil ter izpisal koliko cifer 1 2 3 4 vsebuje.

Zanke

Zanke so 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.

Zanka For

Zanka **for** je najbolj poznana in največkrat uporabljena zanka v vseh programskih jezikih . Najpogosteje se pri krmiljenju zanke uporablja spremenljivka v obliki števca, katerega vrednost se po korakih povečuje ali zmanjšuje. Spremenljivka, ki se uporablja kot števec mora biti števna (celo število, znak, naštevni tip, …)

Struktura zanke **for** :

```
for (inicializacja; test; korak)
{
    stavek;
    stavek;
   // …
}
```

ali :

```
for ( inicializacija; test ; korak )
  stavek;
```

inicializacija: določimo začetno vrednost števca (npr int i = 0) ali števcev, če jih je več;
test: določimo pogoj, ki naj se testira pri posameznem prehodu zanke;
korak: določimo, kako naj se povečuje ali zmanjšuje števec (oz. števci, če jih je več)

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

Vaja:

```
//Izpis tabele kvadratov števil od 1 do 20. Izpis je formatiran na 8 mest, desna poravnava.
Console.WriteLine("Število     Kvadrat");
for (int i = 1; i <= 20; i++)    //i++ je krajša oblika zapisa    i=i+1;
   Console.WriteLine("{0,8}   {1,8}",i,Math.Pow(i, 2));
```

Vaja:

```
//Program ki poišče vsa naravna števila med 1 in 10000, ki so enaka vsoti kubov svojih števk.
//Eno od števil, ki ima zahtevano lastnost je npr. 153: 153 = 13 + 53 + 33.
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;
        vk = Math.Pow(i, 3) + Math.Pow(j, 3) + Math.Pow(k, 3) + Math.Pow(l, 3);
        if (st==vk)
              Console.Write(st+",    ");
      }
```

Vaja:

```
//Primer for zanke, ki vsebuje dva števca: eden se povečuje, drugi zmanjšuje.
Console.Write("Vnesi mejo: ");
int konec = Convert.ToInt32(Console.ReadLine());
for (int gor = 1, dol = konec; (gor <= konec) && (dol >= 1); gor++, dol--)
   Console.WriteLine("{0,5}  {1,5}",gor,dol);
```

Vaja:

```
/*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
*/

int i,j;
Console.Write("Velikost vzorca (1 - 9): ");
int n = Convert.ToInt32(Console.ReadLine());
for(i=n; i>0; i--) {
  for(j=n; j>=i; j--)
     Console.Write(i);
  Console.WriteLine();
}
```

```
/*Napišite program, ki s tipkovnice prebere dve števili, N in R, ter izračuna vsoto vseh
tistih naravnih števil, manjših ali enakih N, ki so deljiva z R.*/

int vs = 0;
Console.Write("Vnesi naravno število N: ");
int N = Convert.ToInt32(Console.ReadLine());
Console.Write("Vnesi delitelj R: ");
int R = Convert.ToInt32(Console.ReadLine());
for (; N >= R; N--)
{
   if ((N % R) == 0) vs = vs + N;
}
Console.WriteLine("\nVsota je {0}", vs);
```

Naloge:

- Kolikšna je vsota vseh naravnih števil med 1 in 1000?
- Tabeliraj linearno funkcijo f(x) = 4x + 3 na intervalu [-5 , 5] s korakom 1!
- Seštej 100 členov vrste: **vrsta = 1 + 1/2+ 1/3 +1/4 + ..**
- Preberi poljubno celo število in izpiši njegov poštevanko!
- Preberi poljubno celo število in ga izpiši navpično!
- Preberi poljuben stavek in ga izpiši navpično in nato še diagonalno (vsaka črka v svoji vrstici, za en znak bolj v desno!.

- Preberi poljubno celo število in ga izpiši z besedami. Celo število **235** tako izpiši kot **dva tri pet**.
- Napiši program, ki vse šumnike *š*, *č* in *ž* nekega stavka nadomesti z znaki *s*, *c* in *z*!
- Napiši program, ki bo izpisal vse kvadrate naravnih števil do n (n preberemo), ki se naprej in nazaj berejo enako. (npr 141, 232, 181, …).
- Napiši program, ki bo prebral dve celi števili, ugotovil, katero je večje in izpisal:
  - vsa števila med najmanjšim in največjim številom.
  - vsa števila med najmanjšim in največjim številom, ki delijo največje število.
  - vsa števila med najmanjšim in največjim številom, ki so soda in delijo največje število.
  - vsa števila med najmanjšim in največjim številom, ki so soda ali delijo največje število.

While zanka

Zanko **while** (ta tudi zanko **do while**) uporabljamo, kadar število ponavljanj zanke ni vnaprej znano, saj je število ponavljanj zanke odvisno od nekega pogoja. Najpomembnejše značilnosti **while** zanke so

- pogoj je testiran 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 testiramo je lahko sestavljen.

Struktura **while** zanke:

```
while (pogoj)
{
    Stavki  //eden ali več poljubnih stavkov
}
```

Vaja:

```
/*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.*/

int n=1,vsota=0;
Console.WriteLine("Zaporedje ima splosni clen a[n]=(n+1)*(n-1); n =1..N!\n");
Console.Write("Vpisi stevilo clenov zaporedja: ");
int N=Convert.ToInt32(Console.ReadLine());
while (n <= N) // tekoci člen <= n-temu clenu
{
    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 "+N+" clenov tega zaporedja je "+ vsota);
```

Vaja:

```
/*Napiši program, ki iz prebranega celega števila naredi novo število, v katerem so le sode
          cifredanega števila. Če so vse cifre danega števila lihe, je novo število enako 0.
*/

int novostevilo=0,cifra,faktor=1;
Console.Write("Vnesi polubno celo število: ");
```

```
int stevilo=Convert.ToInt32(Console.ReadLine());
while (stevilo>0){
    cifra=stevilo%10;
    if (cifra%2==0)
    {
        novostevilo=novostevilo+cifra*faktor;
        faktor=faktor*10;
    }
    stevilo=stevilo/10;
}
Console.WriteLine("\nNovo stevilo je {0}\n",novostevilo);
```

Vaja:

```
/*izpis vseh stirimestnih stevil, pri katerih je vsota zadnjih dveh cifer enaka vsoti prvih
dveh izpis po 10 v vrsti. Koliko je takih števil? */

int stevilo = 1000, e, d, s, t, vvrsti = 0, vseh = 0;
Console.Clear();
while (stevilo < 9999)
{
    e = stevilo % 10;    //enice
    d = (stevilo % 100) / 10;  //desetice
    s = (stevilo % 1000) / 100;  //stotice
    t = (stevilo % 10000) / 1000;  //tisočice
    if (e + d == s + t)
    {
        Console.Write("{0}, ", stevilo);
        vvrsti = vvrsti + 1;
        vseh = vseh + 1;
        if (vvrsti % 10 == 0) Console.WriteLine();
    }
    stevilo = stevilo + 1;
}
Console.WriteLine("\n\nVseh takih števil je {0}.\n", vseh);
```

Naloge:

- Preberi poljuben string. S pomočjo while zanke ugotovi in izpiši, koliko presledkov vsebuje.
- Preberi kemijsko formulo in jo izpiši tako, da so številke vrstico nižje. Če npr. prebereš formulo H2SO4, jo izpiši kot

  H SO
  2 4

- Dana sta dva stringa, *ime* in *priimek*. Sestavi nov string *sifra* tako, da najprej obrneš ime in priimek, nato pa izmenično iz zaporednih črk obrnjenega imena in priimka sestaviš nov string. Iz stringov *ime* in *priimek* vzemi le toliko črk, kot znaša dolžina krajšega od obeh stringov.
- Sestavi program *Kopije*, ki prebere niz *beseda* in pozitivno celo število ***k*** ter izpiše niz, ki je sestavljen iz ***k*** kopij niza *stavek*.
- Sestavi program, ki bo prebral naravno število in izračunal produkt njegovih neničelnih števk. Primer: za število 2304701 program vrne 168.
- Preberi poljubno celo število manjše od 100, računalnik pa naj ga ugane. Seveda predpostaviš, da računalnik števila ne pozna in da ga ugiba. Ugotovi in izpiši koliko poskusov bo računalnik potreboval za to, da ugotovil pravo število. Kakšno taktiko pa bo računalnik ubral, je odvisno od tebe. Nekaj primerov:
  - naključno ugiba števila, dokler ne ugane pravega. Pri tem nič ne upošteva podatka o tem, ali je izžrebano število od iskanega manjše ali večje.

- gre po vrsti od 1 do 100. Pri tem nič ne upošteva podatka o tem, ali je izžrebano število od iskanega manjše ali večje.
- naključno ugiba števila, dokler ne ugane pravega. Pri tem upošteva podatek ali je naključno število preveliko ali premajhno število tako, da zoži interval, iz katerega žreba števila.

- Napiši program ki naj izpiše vsa trimestna števila, katerih vsota števk je 20.

- Oče bi si rad kupil avtomobil, zato se je odločil 1 mesec varčevati na prav poseben način. Prvi dan bo dal na stran 1 tolar, drugi dan 2 tolarja, tretji dan 4 tolarje itd., vsak dan torej dvakrat toliko kot prejšnji dan. Program naj izračuna, po koliko dnevih si lahko oče kupi avtomobil. Ceno avtomobila vnesemo sami.

- Leta 2000 je bila dolžina kapnika 3 mm, nato pa se vsakih 10 let poveča za 6 mm. Napiši program, s katerim boš ugotovil in nato izpisal, kolikšna bo višina kapnika leta 2020 in katerega leta bo dosegel višino 1,5 m ?

- Napiši program, ki izpiše vsa naravna števila med 1 in 3000, ki so deljiva s 7 in niso liha!

Do while zanka

Zanko **do while** (tako kot zanko **while**) uporabljamo, kadar število ponavljanj zanke ni vnaprej znano, saj je število ponavljanj zanke odvisno od nekega pogoja. Najpomembnejše značilnosti **do while** zanke so

- pogoj je testiran na koncu zanke,
- zanka se izvaja, dokler je pogoj izpolnjen,
- zanka se v vsakem primeru izvede vsaj enkrat, četudi pogoj ni izpolnjen že na začetku,
- pogoj, ki ga testiramo je lahko sestavljen.

Struktura **do while** zanke:

```
do
{
    Stavki  //eden ali več poljubnih stavkov
}
while (pogoj)
```

Vaja:

```
/*Program, ki iz vnesenih znakov sestavi številko */
Console.Write("Vnesi zaporednje znakov in pritisni Enter: ");
char znak;
int stevilo = 0;
int cifra;
int faktor = 1;

do
{
    znak = Convert.ToChar(Console.Read());   //prestrežemo znak s tipkovnice
    if ((znak >= '0') && (znak <= '9'))  //vneseni znak predstavlja cifro
    {
       cifra = (int)(znak - 48); //znak spremenimo v cifro glede na ASCII tabelo znakov
       Console.Write(cifra);  //izpišemo vneseno cifro na ekran, da jo bo uporabnik videl
       stevilo = stevilo * faktor + cifra; //cifre spreminjamo/dodajamo skupnemu številu
       faktor = 10;
    }
}
while (znak != (char)13);  //vnos se zaključi s pritiskom tipke <Enter>

Console.WriteLine("\nVneseno stevilo : {0}.", stevilo);
```

Vaja:

```
/*Program nariše trikotnik iz samih zvezdic */

int velikost;
do
{
    Console.Clear();
    Console.Write("Vnesi velikost trikotnika ( 1 - 24 ) : ");
    velikost = Convert.ToInt32(Console.ReadLine());
}
while (velikost < 1 || velikost > 24); //vneseni podatek mora biti med 1 in 24

int zvezde = 1;
while (velikost > 0)
{
    int p = velikost;
    while (p != 0)
    {
       Console.Write(" ");
       p = p - 1;
    }
    int z = zvezde;
    while (z != 0)
    {
       Console.Write("*");
       z = z - 1;
    }
    Console.WriteLine();
    velikost = velikost - 1;
    zvezde = zvezde + 2;
}
```

Vaja:

```
/*Napiši program, ki izračunannalednjo vsoto za poljubno število sumandov!
              2    2    2    2
             -- + -- + -- + -- + ...
              2    3    4    5
  Izpisuj vmesne vsote!!!   */

int clenov;
do
{
     Console.Clear();
     Console.Write("Vnesi število sumandov ( 1 ali več) : ");
     clenov = Convert.ToInt32(Console.ReadLine());
}
while (clenov < 1); //vneseni podatek mora biti več ali enako 1
float suma = 0;
int n = 0;
while (n < clenov)
{
     suma = suma + (float)2 / (2 + n);
     Console.WriteLine("\n{0} .  {1}", n + 1, suma); //izpisujem vmesne rezultate
     n++;
}
Console.WriteLine("\nVsota {0}. členov tega zaporedja je {1}", clenov, suma);
```

Vaja:

```
/*Napiši program, ki na zaslon izpiše prvih 50 vrednosti zaporedja, podanega kot f(n) = f(n-1)
+ f(n-2) (n je naravno število n=1,2,3,...      Pri tem velja   f(1) = 1   in   f(2) = 1.
V vsaki vrstici naj b po izpisanih po 5 števil, izpis na 15 mest, desna poravnava  */

int n = 3;
long f, f1 = 1, f2 = 1;
Console.WriteLine("Vrednosti funkcije f(n) = f(n-1) + f(n-2) na intervalu od 1 do 40\n\n");
Console.Write("{0,10}{0,10}", f1, f2);
do
{
    f = f1 + f2;
    f2 = f1;
    f1 = f;
    Console.Write("{0,10}", f);
```

```
    n++;
    if (n % 5 == 1) Console.WriteLine(); //zato, ker se zanka začne z n=3
} while (n <= 40);
```

Naloge:

- S pomočjo **do while** zanke beri znake toliko časa, da je vneseni znak enak pika (.). Prebrane znake nato izpiši v obliki stavka.
- Preberi poljubno celo število in ga pretvori v dvojiški sestav. Pretvorjeno število nato izpiši!
- Poišči največji skupni delitelj dveh celih števil
- Napiši program, ki bo izpisal vse kvadrate naravnih števil do n (n preberemo), ki se naprej in nazaj berejo enako (npr 121, 676, 484, …). Uporabi **do while** zanko.
- Sestavi preprost program, ki bo kodiral in dekodiral kratka telefonska sporočila (sms-e). Program mora v obrniti vrstni red besed v sporočilu in vrstni red črk v besedi.
- Sestavi program, ki prebere n decimalnih števil (tudi n je podatek). Izpiši, koliko od teh števil je manjših od 10, med 10 (vključno) in 100 (vključno) in koliko večjih od 100.
- Preberi naravno število n. Če je liho, izračunaj 3n+1, sicer pa n/2.Isto naredi z novim številom in postopek ponavljaj, dokler ne dobiš 1. Izpiši število korakov postopka!
- Nek izdelek naj bi se vsak teden podražil za 2%. Da bi državljani razumeli, kaj to pomeni, napiši program, ki prebere začetno in končno ceno tega izdelka in izpiše, v kolikšnem času (število tednov) bo cena dosegla ali presegla to končno vrednost.
- Sestavi program, ki prebere velikost paralelograma in ga izpiše na zaslon. Izpis paralelograma za velikost 5 naj bo takle

```
*****
*****
*****
*****
*****
```

- Največ kolikokrat je potrebno neko število (vneseš ga preko tipkovnice) pomnožiti z 1.1, da bo rezultat še vedno manjši kot 100?

Stavka break in continue

Stavek **break** uporabljamo tudi v zankah: povzroči izstop iz (najbolj notranje) zanke tipa **for**, **while** ali **do while**.

Stavek **continue** pa ima nasprotno vlogo in se lahko pojavi samo v zankah. Pri zanki **for** izvede spremembo spremenljivk in skoči na začetek zanke. Pri ostalih dveh zankah pa skoči na pogoj zanke, ter sproži ponovno preverjanje pogoja. Če je ta še izpolnjen se izvajanje zanke nadaljuje, sicer pa se zanka zaključi.

Vaja:

```
double stevilo;
while (true)
{
    Console.Write("Vnesi poljubno število :");
    stevilo=Convert.ToDouble(Console.ReadLine());
```

```
    if (stevilo == 0.0)
          continue; //nazaj na začetek zanke
    Console.WriteLine("  Obratna vrednost števila "+stevilo+" je " + 1 / stevilo);
    break; //izstop iz zanke
}
```

Naloge:

- Preberi 5 stavkov. Če je v stavku manj kot 5 znakov, ga ne upoštevaj (stavek **continue**). Na koncu ugotovi in izpiši najdaljši stavek.

- Radi bi tabelirali funkcijo y = 10 sin(5x) na intervalu od 0 do .. Želimo izpisati le tiste točke, ki so po ordinatni vrednosti za 1 večje od sosednje točke. (nasvet: izberemo majhen korak in preračunamo vrednost funkcije. Če je izračunana vrednost premajhna glede na prejšnjo izračunano vrednost, s pomočjo ukaza **continue** ponovno izračunamo y v naslednji točki.

- Sestavi program, ki bo izpisal prvih n decimalk kvocienta a/b. Delaj samo s celimi števili.
  Primer:
  Vneseno število a: 10
  Vneseno število b: 761
  Vneseno število n : 40
  Rešitev: 0.0131406044678055190538764783180026281208

- Program, ki prebere naravno število in izpiše vse kvadrate do vključno prebranega.

- Napiši program, ki meče dve kocki toliko časa, da na obeh padeta šestici. Vsako kocko predstavljajo naključna števila med 1 in 6. Program naj tudi šteje, kolikokrat smo vrgli kocki, da smo dobili dvojno šestico.

- Sod drži 780 litrov. Začel je puščati, ker ga kletar ni popravil. Na minuto je izteklo 6 litrov vina. To je opazil šele čez pol ure. Koliko vina je še ostalo v sodu? Napiši še program, ki za vsako velikost soda izračuna in izpiše, koliko tekočine je izteklo iz njega, če je pogoj nespremenjen, torej izteče 6 litrov na minuto.

- Sestavi enostavni program, ki bo prebral vnešeno število ter izpisoval vsako drugo celo število do izbranega števila.

- Napiši program, ki najprej prebere naravni števili **n** in **m**, nato generira naključna števila med 1 in **m** toliko časa, da se pojavi število **n** in jih izpiše. Izpiše naj tudi število poskusov. Če vpisani podatki onemogočajo rešitev (če je **n > m**), naj se izpiše, da problem ni rešljiv.

- Napiši program, v katerega vnašaš števila in ti na koncu izpiše največjo in najmanšo vnešeno številko. Vnos se zaključi, ko uporabnik vnese število 0!

- Izpiši tista števila med celima številoma a in b (podatka, ki ju prebereš), ki so deljiva z vsoto svojih števk. Npr. 12 je že deljivo z vsoto števk (3). Prav tako 1101. Če sta torej podatka 8 in 14, naj program izpiše: Med 8 in 14 so z vsoto svojih števk deljiva naslednja števila: 8, 9, 10, 12.

Operatorji nad biti v C#

Operacije nad biti: OR(|), XOR(^), AND(&) in NOT(~)

C# pozna naslednje operatorje, ki delujejo nad biti:

- binarni OR(|) operator
- binarni AND(&) operator
- XOR (^) operator
- Not (~) operator

Binarni OR(|) operator

Binarni operator | (logični bitni OR) je namenjen številnim podatkovnim tipom in logičnim tipom. Pri številnih podatkovnih tipih je rezultat *false* samo v primeru, **da sta oba operatorja** *false*.

Primer:

```
byte a = 7;
byte b = 9;
int logicniOr = a | b;
Console.WriteLine(a+" | "+ b+" = "+ logicniOr); //Izpis: 7 | 9 = 15
```

Da bi razumeli, kako pridemo do takega rezultata, se spomnimo, da so podatki v pomnilniku zapisani v binarni obliki. Število 1 nam predstavlja vrednost *true*, število 0 pa vrednost *false*. Tule je bazična tabela, ki predstavlja binarni zapis potence števila 2 od 1 (2^0) do 128 (2^7). Ker je *byte* sestavljen iz 8 *bitov*, potrebujemo tabelo 8 elementov:

128	64	32	16	8	4	2	1	Rezultat
0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	1	0	2
0	0	0	0	0	1	0	0	4
0	0	0	0	1	0	0	0	8
0	0	0	1	0	0	0	0	16
0	0	1	0	0	0	0	0	32
0	1	0	0	0	0	0	0	64
1	0	0	0	0	0	0	0	128

Če pogledamo tabelo bolj podrobno vidimo, kako se vsako naslednjo vrstico (vsako višjo potenco števila 2) enka premika po tabeli za eno mesto bolj v levo (»**bit shift**«).

Poglejmo sedaj, kako bi v binarni tabeli predstavili števili 7 in 9.

128	64	32	16	8	4	2	1	Izračun	Rezultat
0	0	0	0	0	1	1	1	(4+2+1)	7
0	0	0	0	1	0	0	1	(8+1)	9

Bitni **or** (|) deluje nad posameznimi biti. Ker v našem primeru noben par bitov ni hkrati *false* (vselej je vsaj eden enak 1, torej *true*) je rezultat takle

0	0	0	0	1	1	1	1	**(8+4+2+1)**	**15**

Vaja:

```
//številom med 0 in 10 zamenjajmo prvi bit (0 v 1, oz 1 v 0)

ushort stevilo;
ushort i;
for (i = 0; i <= 10; i++)
{
     stevilo = i;
     Console.WriteLine("Število: " + stevilo);
     stevilo = (ushort)(stevilo | 1); // stevilo | 0000 0001
     Console.WriteLine("Število potem, ko smo zamenjali prvi bit: "+ stevilo + "\n");
}
// Izpis:
Število: 0
Število potem, ko smo zamenjali prvi bit: 1
Število: 1
Število potem, ko smo zamenjali prvi bit: 1
. . .
Število: 10
Število potem, ko smo zamenjali prvi bit: 11
```

Binarni AND(&) operator

Binarni operator & (logični bitni AND) je namenjen številnim podatkovnim tipom in logičnim tipom. Pri številnih podatkovnih tipih je rezultat *true* samo v primeru, **da sta oba operatorja** *true*.

Primer:

```
byte a = 7;
byte b = 9;
int logicniIn = a & b;
Console.WriteLine(a+" & "+ b+" = "+ logicniIn); //Izpis: 7 & 9 = 1
```

Da bi razumeli, kako pridemo do takega rezultata, poglejmo še enkrat v tabelo. Ker bitni **in** (**&**) deluje nad posameznimi biti, v našem primeru pa je le en par bitov hkrati *true* (samo enkrat je par enak 1, torej *true*) je rezultat takle

128	64	32	16	8	4	2	1	Izračun	Rezultat
0	0	0	0	0	1	1	1	(4+2+1)	7
0	0	0	0	1	0	0	1	(8+1)	9
0	0	0	0	0	0	0	1	(1)	1

Vaja:

```
//S pomočjo operatorja & ugotovi, ali je neko število sodo ali liho

byte num;
num = 21;

if ((num & 1) == 1)
    Console.WriteLine(num + " je liho število");

Random naklj = new Random();
```

```
byte stevilo = (byte)naklj.Next(0, 100);//naključno celo število med 0 in 1000

if ((stevilo & 1) == 1)
    Console.WriteLine(stevilo + " je liho število");
else
    Console.WriteLine(stevilo + " je sodo število");
```

Bitni Xor (^) operator

Binarni operator & (logični bitni ekskluzivni OR) je namenjen številnim podatkovnim tipom in logičnim tipom. Pri številnih podatkovnih tipih je rezultat *true* samo v primeru, **da je natanko eden od operatorjev** *true*.

Primer:

```
byte a = 7;
byte b = 9;
int logicniXor = a ^ b;
Console.WriteLine(a+" ^ "+ b+" = "+ logicniXor); //Izpis: 7 & 9 = 14
```

Da bi razumeli, kako pridemo do takega rezultata, poglejmo še enkrat v tabelo. Ker bitni **ekskluzivni ali** (^) deluje nad posameznimi biti, v našem primeru pa je v dveh primerih natanko en od operatorjev enak *true* (dvakrat je natanko eden od parov enak 1, torej *true*) je rezultat takle

128	64	32	16	8	4	2	1	Izračun	Rezultat
0	0	0	0	0	1	1	1	(4+2+1)	7
0	0	0	0	1	0	0	1	(8+1)	9
0	0	0	0	0	0	0	1	(8+4+2)	14

Še en primer uporabe bitnega **XOR** operatorja nad spremenljivkami tipa *char*:

```
char ch1 = 'A';
char ch2 = 'B';
char ch3 = 'C';
int key = 88;

Console.WriteLine("Originalno sporočilo: " + ch1 + ch2 + ch3);//Izpis ABC

// kodirajmo sporočilo
ch1 = (char)(ch1 ^ key);
ch2 = (char)(ch2 ^ key);
ch3 = (char)(ch3 ^ key);

Console.WriteLine("Kodirano sporočilo: " + ch1 + ch2 + ch3);//Izpis ↓→←  (ASCII 25,26 in 27)

// dekodirajmo sporočilo, ki bo enako originalnemu!!!
ch1 = (char)(ch1 ^ key);
ch2 = (char)(ch2 ^ key);
ch3 = (char)(ch3 ^ key);

Console.WriteLine("Dekodirano sporočilo: " + ch1 + ch2 + ch3);//Izpis ABC
```

Bitni not (~) operator

Operator ~ (Bitni NOT) predstavlja komplement operanda, kar v praksi pomeni nasprotno vrednost vsakega bita posebej. Uporabljamo ga lahko nad spremenljivkami tipov *int*, *uint*, *long*, in *ulong*. Operator ~ je tudi **unarni** operator, tako da mu ni potrebno posredovati vrednosti (npr. *~n* ali pa *~a+b*).

Ker binarni operator ~ vrača komplement vsakemu bitu, je rezultat lahko pozitiven ali pa negativen (**negativen seveda le v primeru, da smo uporabili spremenljivko predznačenega tipa** – torej spremenljivko, ki lahko hrani negativno vrednost)

Primer:

```
byte a = 7;
byte b = 9;
int NOTMINUSa = ~-a;//bitni not negativnega števila a
int NOTa = ~a;
int rez=~a+b;

Console.WriteLine(" ~-" + a + " = " + NOTMINUSa);//Izpis: ~ -7 = 6
Console.WriteLine(" ~ " + a + " = " + NOTa);//Izpis: ~ 7 = -8
Console.WriteLine(" ~ " + a + " + " + b + " = " + rez);//Izpis: ~ a + b = 1
```

Poglejmo še, kako pridemo do takih rezultatov.

128	64	32	16	8	4	2	1	Izračun	Rezultat
0	0	0	0	0	1	1	1	(4+2+1)	7
1	1	1	1	1	0	0	0	**~a = -128+64+32+16+8+1**	-8

Pri spremenljivkah predznačenih tipov, je skrajni levi bit namenjen hranjenju predznaka (1= negativen predznak, 0 je pozitiven predznak). **Negativna števila pa so v C# interno predstavljena v dvojiškem komplementarnem formatu: vsak bit dobi najprej nasprotno vrednost, nato pa dodamo še 1.**

128	64	32	16	8	4	2	1	Izračun	Rezultat
0	0	0	0	0	1	1	1	(4+2+1)	7
1	1	1	1	1	0	0	0	**-a= (-128 +64+32+16+8) +1**	-7
0	0	0	0	0	1	1	1	**~-a = 4+2+1**	-8

Vaja:

```
//Program izpiše posamezne bite števila (oz. število izpišemo v binarnem zapisu), nato pa
//to število obrne z operatorjem ~ in ga ponovno izpiše!

sbyte b = 104;
//ispis posameznih bitov števila b
for(int t=128; t > 0; t = t/2)
{
   if((b & t) != 0)
     Console.Write("1 ");
   else
     Console.Write("0 ");
}
Console.WriteLine();

//obrnimo posamezne bite
b = (sbyte) ~b;
//ispis posameznih bitov števila ~b
for (int t = 128; t > 0; t = t / 2)
{
   if ((b & t) != 0)
     Console.Write("1 ");
   else
     Console.Write("0 ");
}
//Izpis:
0 1 1 0 1 0 0 0
```

```
1 0 0 1 0 1 1 1
```

Shift operatorja levi Shift (<<) in desni Shift (>>)

***Levi Shift* operator (<<)** premakne prvi operand v levo za tolikšno število bitov, kot jih navedemo v drugem operandu. Zaradi tega mora biti drugi operand obvezno tipa *int*.

```
int n = 1;
long lg = 1;
//izpis v desetiškem formatu
Console.WriteLine(n << 1);   //izpis: 2
Console.WriteLine(n << 33);  //izpis: 2
Console.WriteLine(lg << 33); //izpis: 8589934592
//POZOR: i<<1 in i<<33 nam data enak rezultat, ker imata 1 and 33 enakih spodnjih pet bitov.
//izpis v heksadecimalnem (šestnajstiškem) formatu
Console.WriteLine("0x{0:x}", n << 1);   //izpis: 0x2
Console.WriteLine("0x{0:x}", n << 33);  //izpis: 0x2
Console.WriteLine("0x{0:x}", lg << 33); //izpis: 0x200000000
```

***Desni Shift* operator (>>)** premakne prvi operand v desno za tolikšno število bitov, kot jih navedemo v drugem operandu. Zaradi tega mora biti drugi operand obvezno tipa *int*.

```
//DESNI Shift operator
int st = 5;//5= 00000101
Console.WriteLine(st >> 1);//Izpis 2, ker je st>>1=00000010=2

st = 16;//16= 00010000
Console.WriteLine(st >> 4);//Izpis 1, ker je st>>4=00000001=1
```

Vaja:

```
//Primer uporabe Shift operatorjev << in >>
Console.WriteLine("Operator <<");
int val = 1;
for (int i = 0; i < 8; i++)
{
   for (int t = 128; t > 0; t = t / 2)
   {
       if ((val & t) != 0) Console.Write("1 ");
       if ((val & t) == 0) Console.Write("0 ");
   }
   Console.WriteLine();
   val = val << 1; // levi shift
}
            /*Izpis:
            0 0 0 0 0 0 0 1
            0 0 0 0 0 0 1 0
            0 0 0 0 0 1 0 0
            0 0 0 0 1 0 0 0
            0 0 0 1 0 0 0 0
            0 0 1 0 0 0 0 0
            0 1 0 0 0 0 0 0
            1 0 0 0 0 0 0 0*/

Console.WriteLine();

Console.WriteLine("Operator <<");
val = 128;

for (int i = 0; i < 8; i++)
{
   for (int t = 128; t > 0; t = t / 2)
   {
       if ((val & t) != 0) Console.Write("1 ");
       if ((val & t) == 0) Console.Write("0 ");
```

```
   }
   Console.WriteLine();
   val = val >> 1; // desni shift
}
            /*Izpis:
            1 0 0 0 0 0 0 0
            0 1 0 0 0 0 0 0
            0 0 1 0 0 0 0 0
            0 0 0 1 0 0 0 0
            0 0 0 0 1 0 0 0
            0 0 0 0 0 1 0 0
            0 0 0 0 0 0 1 0
            0 0 0 1 0 0 0 1
           */
```

Metode (funkcije)

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 se 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 podnaloge, vsako podnalogo pa izdelamo posebej. Te manjše podnaloge 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 prav izrecni namen te metode.

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.

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 malo črko. Metodam damo smiselna imena, ki povedo, kaj metoda dela. Ime metode je poljubno, ne smemo pa za ime metode pa 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 zaželjeni 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 so metode, 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 delm našega programa (do nadalnjega bodo vse metode javne)

Statična metoda (do nadanjega 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.

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 izpisemo
    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();
```

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*.

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 vsota() //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 `vsota()` vrača rezultat (tip *int*), jo ne moremo klicati samostojno, ampak v nekem **prireditvenem stavku**, lahko pa tudi kot parameter v drugi metodi.

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

```
Console.WriteLine(vsota());//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.

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.

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 stevilo a
      return a;
    return b;   //sicer pa metoda vrne stevilo 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);
```

Funkcijo `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 naključnih števil je: "+
                  max(prvo, max(drugo, tretje)));
```

Zgledi

Trikotnik

Napišimo metodo, dobi za parameter polubno celo število n in ki na zaslon nariše trikotnik, sestavljen iz samih zvezdic. Velikost trikotnika določa nenegativno celo število n, ki je podano kot argument te metode.

```
Primer:  n=0  *     n=1  *     n=2  *     n=3  *
                        ***        ***        ***
                                  *****      *****
                                            *******
```

```
//Metoda
static void trikotnik(int n)
{
    int i, j;
    for( i=0; i<n+1; i++ )
    {
       for( j=n-i; j>0; j-- )      //Najprej ustrezno število presledkov
           Console.Write(" ");
       for( j=0; j<(2*i+1); j++ ) //Rišemo zvezdice
           Console.Write("*");
       Console.WriteLine();        //Skok v novo vrstico
    }
}
//Glavi program
static void Main(string[] args)
{
    Console.Write("Velikost trikotnika: ");//Velikost trikotnika preberemo
    int velikost = Convert.ToInt32(Console.ReadLine());
    trikotnik(velikost); //klic metode
}
```

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
}
//Glavni program
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+".");
}
```

Število Pi

Število PI lahko izračunamo tudi kot vsoto vrste 4 - 4/3 + 4/5 - 4/7 + 4/9 - ... Napiši metodo, ki ugotovi in vrne, koliko členov tega zaporedja moramo sešteti, da se bo tako dobljena vsota ujemala s konstanto Math.PI do npr. vključno devete decimalke. Metoda naj ima za parameter število ujemajočih se decimalk. Rešitev za npr. 9 členov: 1096634169

```
//Metoda
static long stClenov(int clenov)
{
    double pi=4;   //definiramo in inicializiramo začetno vrednost za pi
    double clen;   //tekoči člen zaporedja
    long i=1;      //definicija in inicializija števca členov
    while (Math.Round(pi,clenov)!=Math.Round(Math.PI,clenov))
    {
       clen=4.00/(i*2+1);  //izracun tekocega clena
       if(i%2!=0)
           pi-=clen; //lihe člene odštevamo, krajši zapis za pi=pi-clen
       else
           pi+=clen; //sode člene prištevamo, krajši zapis za pi=pi+clen
       i++;
    }
    return i;
}

//Glavni program
static void Main(string[] args)
{
```

```
    Console.WriteLine("Računam koliko členov zaporedja 4 - 4/3 + 4/5 -
           4/7 + 4/9 - ... je potrebno\nsešteti, da bo tako dobljena vsota
           enaka konstanti Math.PI!");
    Console.WriteLine("\nTrenutek ......\n");

    //ujemanje na 9 decimalk
    Console.WriteLine("Število členov: " + stClenov(9));
}
```

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());
   } // for
   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));
} // main

public static int minOcena(int[] ocene)
{
   int najnizja = ocene[0];

   for(int i = 1; i < ocene.Length; i++)
   {
       najnizja = Math.Min(najnizja, ocene[i]);
   } // for
   return najnizja;
} // minOcena

public static int maxOcena(int[] ocene)
{
   int najvisja = ocene[0];

   for(int i = 1; i < ocene.Length; i++)
   {
       najvisja = Math.Max(najvisja, ocene[i]);
   } // for
   return najvisja;
} // maxOcena

public static int vsotaOcen(int[] ocene)
{
   int vsota = 0;

   for(int i = 0; i < ocene.Length; i++)
   {
```

```
        vsota = vsota + ocene[i];
    } // for
    return vsota;
} // vsotaOcen

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

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.

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
    // Vrni 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)
{ // Glavna metoda
    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.

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 staku 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 se ponovi
        else sPapaj = sPapaj + s[i];
     }
     return sPapaj; //vračanje papajščine
}
//glavni program
public static void Main(string[] args)
{
   string s = "Danes je konec šole.";
   string papaj = papajscina(s);
   Console.WriteLine(papaj);
}
```

Naloge za utrjevanje znanja iz metod

- Izmed spodnjih imen metod izberi tisto, ki se začne izvajati ob zagonu programa z imenom Test.
  - `static void Test(string[] vhod)`
  - `static void Main()`
  - `static void Test()`
  - `static void Main(string[] args)`

- Kakšen je izpis naslednje metode, če jo pokličemo s stavkom *vraca(10)*? Najprej reši "peš" in šele potem preizkusi z računalnikom.

```
public static void vraca(int n)
{
    int i = 1;
    while (i <= n)
    {
       if (i % 2 != 0) Console.Write(i);
       else Console.Write('#');
       i++;
    }
    Console.WriteLine('#');
}
```

- Dana je metoda

```
public static int Vsota(int n)
{
    int r = 1;
    int i = 1;
    while (i < n)
    {
        i = i + 1;
        r = i * r;
    }
    return r;
}
```

  - Kako je ime metodi?
  - Kakšen je tip rezultata, ki ga vrača metoda?
  - Koliko argumentov sprejme metoda?
  - Kakšni so tipi in imena argumentov?
  - Kaj metoda počne? Ali lahko predlagaš boljše ime za metodo?
  - Napiši program, ki demonstrira delovanje metode

- Napišite metodo, ki bo izpisala vsa števila med 0 in 1000, katerih vsota števk je enaka številu, ki nastopa kot parameter te metode.

- Denimo, da ste stari 16 let in imate zelo sitne starše, ki želijo brati vaša sporočila. Zato se s prijateljem dogovorita, da si bosta sporočila pošiljala zakodirana. Napisati morate metodo, ki bo sporočilo zakodirala. Zakodirano sporočilo naj bo sestavljeno najprej iz znakov, ki so bili v originalnem sporočilu na sodih mestih in nato iz znakov, ki so bili v originalnem sporočilu na lihih mestih.
  Primer:
  Originalno sporočilo Greva na pijaco?, se zakodira v sporočilo Gean iaorv apjc?.

- Za lažje sporazumevanje s prijateljem sestavite še metodo, ki dekodira sporočilo, kodiramo po metodi prejšnje naloge.

- Kaj se izpiše na zaslon po zagonu metode *preveri(tab)*. ? Najprej reši "peš" in šele potem preizkusi z računalnikom.

```
static void Main(string[] args)
{
    int[] tab = { 1, 2, 4, 3, 0, 100, 1 };
    preveri(tab);

    Console.ReadKey();
}
```

```
public static void preveri(int[] tab)
{
    int i = 0;
    while (i < tab.Length)
    {
        if (tab[i] == 0)
        {
            Console.WriteLine('*');
            break;
        }
        Console.Write(tab[i] + ",");
        i++;
    }
}
```

- Sestavite metodo Razlika, ki sprejme tabelo celih števil in vrne razliko med največjim in najmanjšim elementom v tabeli.
  Primer:
  Ko se izvedejo ukazi
  int[] tab = new []{5,1,2,4,6,8};
  int raz = Razlika(tab);
  ima raz vrednost 7.

- Sestavite metodo, ki kot parameter dobi tabelo in permutacijo (permutacija 1 3 2 4 pove, da prvi element ostane tam, kjer je, novi drugi element je stari tretji, tretji je stari drugi, četrti pa ostane tam kjer je). Vrnemo novo tabelo, ki ima elemente premešane tako, kot zahteva permutacija.

- Napišite metodo, ki kot parameter dobi dve tabeli celih števil. Vrne naj novo tabelo, v kateri so tiste vrednosti, ki se pojavljajo v prvi tabeli in ne v drugi! Vemo, da so vse vrednosti v obeh tabelah med seboj različne (torej se v isti tabeli število ne ponovi).
  Primera:
  Če so v prvi tabeli podatki 4, 5, 6 in v drugi 5, 6, 4, naj metoda vrne prazno tabelo.
  Če so v prvi tabeli podatki 4, 5, 6 in v drugi 1, 2, 4, naj metoda vrne tabelo 5 in 6.

- Sestavite 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 sole.";
  string papaj = Papajscina(s);
  je niz papaj enak "Dapanepes jepe koponepec sopolepe.".

- Napiši metodo sekunde, ki sprejme tri cela števila : ure, minute in sekunde. Metoda naj izračuna, koliko je to sekund in vrne rezultat.

- Napiši metodo, ki izpiše vsa števil med 0 in 1000, katerih vsota števk je enaka številu, ki nastopa kot parameter te metode.

- Napiši metodo, ki dobi za parameter poljubno decimalno število in ki izračuna in vrne vsoto vseh cifer v tem številu!

- Sestavi metodo kopije(String s, int k), ki sprejme niz s in pozitivno celo število k ter vrne niz, ki je sestavljen iz k kopij niza s.

- Napiši metodi zakodiraj in odkodiraj, ki sprejmeta niz in vrneta zakodiran oz. odkodiran niz. Metoda zakodiraj naj vsako črko v tekstu zamenja s črko, ki v (angleški) abecedi leži 5 črk za originalno. Menjava je seveda ciklična, tako da črko a zamenjata s črko e, črko z pa s črko d. Upoštevaj, da so črke lahko male ali pa velike. Seveda naj metoda ločil in ostalih znakov ne kodira!

- Napiši metodo, ki sestavi posebno enostavno križanko in jo izpiše. To pomeni, da uporabnik vnese prvo besedo s poljubnim številom črk in jo napiše navpično navzol. Potem ga program za vsako črko te besede vpraša za besedo, ki se začne na to črko in je dolga 4 črke in jo zapiše vodoravno od tiste črke.

- Napiši metodo donos, ki ima tri parametre: cas, znesek in obresti. Cas: koliko let se denar obrestuje. Znesek: koliko denarja položimo na bančni račun. Obresti: koliko obresti (v %)se pripiše glavnici vsako leto. Uporabi obrestno obrestni račun in metodo preizkusi.

- Sestavi metodo s pomočjo katere boš izračunal produkt vseh sodih števil med dvema danima pozitivnima celima številoma.

- Napiši metodo, ki sprejme pozitivno celo število n in na zaslon izpiše naslednji vzorec: najprej se n-krat izpiše .(pika) in nato n-krat znak #. Prikazan je primer za n = 5:

  .....#####

- Sestavi metodo, ki bo izračunala vsoto notranjih kotov večkotnika. Namig: vsota notranjih kotov n - kotnika se izračuna po formuli: (n−2)·180.

Prenos parametrov po referenci

V vseh dosedanjih primerih so bili parametri, ki smo jih posredovali metodam, posredovani na privzeti način. Takemu načinu pravimo **prenos parametrov po vrednosti**. To pomeni, da je bila vrednost vsake spremenljivke posredovana ustreznemu parametru v metodi, ki je tako v resnici delala s kopijo originalne spremenljivke. Zaradi tega sprememba vrednosti parametra v metodi ni vplivala na velikost spremenljivke, ki smo jo navedli pri klicu metode.

Parametre pa lahko pri klicu metode kličemo tudi po **referenci**. V tem primeru dobi metoda le referenco na ustrezno spremenljivko, kar dejansko pomeni, da vsaka sprememba tega parametra v metodi, pomeni spremembo vrednosti spremenljivke, ki smo jo uporabili pri klicu te metode. Klic po referenci dosežemo s pomočjo rezervirane besede **ref**. Vendar pozor: besedico **ref** moramo napisati tako pred tipom parametra v glavi metode, kot tudi pred imenom spremenljivke pri klicu metode.

```
//Metoda
static void metoda(ref string s) //parameter s je posredovan po referenci
{
    s = "Spremenjen!";
}

//Glavni program
static void Main()
{
    string stavek = "Originalen stavek";
    metoda(ref stavek); //parameter stavek je klican po referenci
   // stavek je sedaj spremenjen
}
```

Metoda za zamenjavo vrednosti dveh spremenljivk

Napišimo metodo, ki naj dobi za parametra dve spremenljivki *st1* in *st2*, metoda pa naj zamenja vrednosti teh dveh spremenljivk.

```
//Metoda
static void zamenjaj(ref int st1, ref int st2)//parametra podana po
referenci
{
    int zacasna = st1;
    st1 = st2;
    st2 = zacasna;
}

//Glavni program
static void Main(string[] args)
{
    int stevilo1 = 200;
    int stevilo2 = 500;
    Console.WriteLine("Število 1: "+stevilo1+", število 2: " + stevilo2);
    zamenjaj(ref stevilo1, ref stevilo2); //klic parametrov po referenci
    Console.WriteLine("Število 1: "+stevilo1+", število 2: " + stevilo2);
}
```

Jezik C# pozna še en način klica parametrov po referenci, to je klic s pomočjo rezervirane besede **out**. Besedica *out* je sicer podobna besedici *ref*, razlika pa je v tem, da klic s pomočjo besedice *ref* zahteva, da je spremenljivka, ki nastopa kot parameter metode, pred klicem že inicializirana. Tudi pri klicu s pomočjo besedice *ref* velja, da jo moramo napisati tako pred tipom parametra v glavi metode, kot tudi pred imenom spremenljivke pri klicu metode.

Primera:

```
//Metoda
static void metoda(out int i)
{
    i = 44;
}
static void Main()
{
    int stevilo;  //spremenljivka še ni inicializirana
    metoda(out stevilo);
     //vrednost spremenljivke je 44
}
```

```
//Klic po referenci - out
static void methoda1(out int i, out string s1, out string s2)
{
    i = 44;
    s1 = "Danes je lep dan!";
    s2 = null;
}

//Glavni program
static void Main()
{
```

```
    int stevilo;
    string str1, str2;
    metoda1(out stevilo, out str1, out str2);
    //spremenljivke stevilo, str1 in str2 so dobile vrednosti v metodi
}
```

Zgled

Obrni niz

Napišimo metodo, ki obrne niz znakov (npr »abeceda« pretvori v »adeceba«).

```
//Metoda: parameter stavek je klican po referenci
static void obrni(ref string stavek)
{
    string pomozni = "";
    for (int i = stavek.Length - 1; i >= 0; i--)
        pomozni = pomozni + stavek[i];
    stavek = pomozni;
}

//Glavni program
static void Main(string[] args)
{
    Console.Write("Stavek: ");
    string stavek = Console.ReadLine();
    Console.WriteLine("\n\nOriginalni stavek: \n\n" + stavek);
    obrni(ref stavek); //Parameter klican po referenci
    Console.WriteLine("\n\nObrnjeni stavek: \n\n"+stavek+"\n\n");
}
```

Kreiranje metode s pomočjo čarovnika

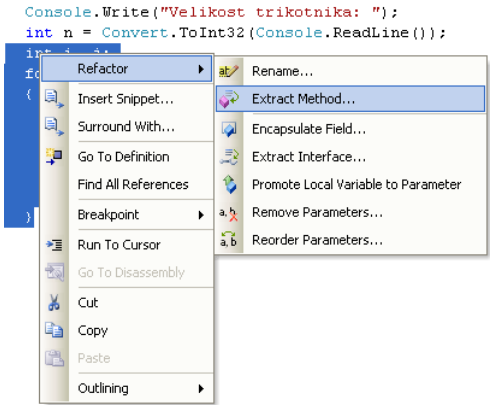
Bolj za informacijo, kot pa za neko resno uporabo si poglejmo še kako nam C# omogoča pisanje metod tudi s pomočjo čarovnika. Kot primer vzemimo naslednji program:

```
static void Main(string[] args)
{
    Console.Write("Velikost trikotnika: ");
    int n = Convert.ToInt32(Console.ReadLine());

    int i, j;
    for (i = 0; i < (n + 1); i++)
    {
        for (j = n - i; j > 0; j--)
            Console.Write(" ");

        for (j = 0; j < (2 * i + 1); j++)
            Console.Write("*");
        Console.WriteLine();
```

Tole kodo v glavnem programu bi radi nadomestili s funkcijo

```
    }
}
```

Kodo, ki bi jo radi s pomočjo čarovnika zapisali v novo funkcijo, najprej označimo. Nato kliknemo desni miškin gumb in v oknu, ki se odpre izberemo opcijo Rafactor in nato Extract Method.. V oknu, ki se odpre, nato še zapišemo ime funkcije (npr. trikotnik) in ime potrdimo s klikom na gumb OK. Čarovnik nam nato sam zgenerira ustrezno funkcijo.



V našem primeru bo rezultat takle:

```
//Glavi program
static void Main(string[] args)
{
     Console.Write("Velikost trikotnika: ");
     int n = Convert.ToInt32(Console.ReadLine());
     trikotnik(n);  //KLIC nove metode
}

private static void trikotnik(int n)
{
   int i, j;
   for (i = 0; i < (n + 1); i++)
   {
       for (j = n - i; j > 0; j--)
         Console.Write(" ");
       for (j = 0; j < (2 * i + 1); j++)
         Console.Write("*");
       Console.WriteLine();
   }
}
```

NOVA metoda – zgeneriral jo je čarovnik

Znaki (tip char)

V spremenljivki lahko hranimo tudi en znak. V ta namen uporabljamo podatkovni tip **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);
} // main
```

Program prevedemo in poženemo:

```
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();
} // main
```

Program prevedemo in poženemo:

```
194
```

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 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';
```

Abeceda

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:         }  // for
12:    }
```

Program prevedemo in poženemo:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
z y x w v u t s r q p o n m l k j i h g f e d c b a
```

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 ASCII[1] standardu, 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.

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);
```

1 V računalništvu oznaka za splošno uveljavljeno 8-bitno kodo za kodiranje običajnih črk in znakov, s katerimi se zapisuje besedila.

```
    Console.WriteLine(znak_a == 'a');
} // main
```

Program prevedemo in poženemo:

```
False
True
```

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.

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.

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).

```
C2:  static void Main(string[] args)
C3:  {    // Ustvarimo generator naključnih števil
C4:      Random geslo = new Random();
C5:      // Določimo dve števki
C6:      int stev1 = geslo.Next(10);
C7:      int stev2 = geslo.Next(10);
C8:      // Določimo tri naključne črke (velike tiskane)
C9:      char c1 = (char)geslo.Next('A', 'Z' + 1);
C10:     char c2 = (char)geslo.Next('A', 'Z' + 1);
C11:     char c3 = (char)geslo.Next('A', 'Z' + 1);
C12:     // Določimo eno naključno števko
C13:     int stev3 = geslo.Next(10);
C14:
C15:     // Določimo naključno tromestno število
C16:     int s = geslo.Next(100, 1000);
C17:     // Izpis gesla
C18:     Console.WriteLine("Geslo: "+stev1+stev2 +c1 +c2 +c3+stev3+ s);
C19:     Console.ReadKey();
C20: }
C21:
C22:
```

Zapis na zaslonu:

```
Geslo: 99OSF8654
```

Razlaga

Najprej ustvarimo generator naključnih števil. Nato generiramo posamezne elemente gesla (C7 - C18). 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 (C20).

Naloge (tip char)

- Napišite program, ki bere posamezne znake in jih kodira v števila na naslednji način:
  - Ne loči velike in male črke,
  - črke od a do e (A do E) kodira po vrsti s števili od 1 do 5,
  - številke 0 do 9 kodira po vrsti s števili od 6 do 15,
  - črke od f do z (F do Z) kodira s števili od 16 do 36 v obratnem vrstnem redu,
  - vse ostale znake zakodira s številom 0.

Program naj za vsak posamezen znak izpiše na zaslon znak = koda. Branje se konča, če je prebran znak *.
Namig: Za pretvarjanje iz tipa string v tip char si pomagajte z metodo char.Parse(niz).

Primer kodiranja:

Vnesi znak: a
a = 1

Vnesi znak: Z
Z = 16

Nizi (tip string)

Pogosteje kot posamezne znake uporabljamo nize znakov (ang. string). 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();
}
```

Program prevedemo in poženemo:

```
Dobro jutro.
```

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

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

Dostop do znakov v nizu

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

Dolžina niza

Da zvemo dolžino niza, uporabljamo lastnost Length.

Primer:

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

Vrednost spremenljivke dolzina je 11.

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

Zgledi

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("Vnešeni niz je: " + beri);
6:
7:         for(int i = beri.Length - 1; i >= 0; i--)
8:         {
9:             Console.Write (beri[i]);
10:        } // for
11:        Console.WriteLine ("\n");
12:    } // main
```

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.

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("Vnešeni niz je: " + beri);
      int i = 0;
      string obrnjeniNiz = "";
      while (i < beri.Length)
      {
           obrnjeniNiz = beri[i] + obrnjeniNiz;
           i++;
      }
      Console.WriteLine("\nObrnjeni niz: " + obrnjeniNiz);
}
```

Preštej števke v nizu

Ugotovimo, koliko števk vsebuje poljuben niz:

- Preberemo niz
- Pregledamo vsak znak
  - `znak = niz[i]; // tekoči znak`
- Če je števka, povečamo števec za 1
  - `if (('0' <= znak) && (znak <= '9'))`
    `{ // če je števka`
    `koliko_stevk++; // povečanje za 1`
    `}`
- Izpišemo rezultat

```
i = 0;
while (i < dol_niza) {
    znak = niz[i]; // tekoči znak
    if (('0' <= znak) && (znak <= '9'))  {
        // če je stevka
```

```
            koliko_stevk = koliko_stevk + 1;
        }
        i = i + 1;
    }
    rezultat = "V nizu " + niz + " je " + koliko_stevk + " števk.";
```

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);
} // main
```

Program prevedemo in poženemo:

```
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);
} // main
```

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.

- `String.Compare("matija","mojca") // vrne negativno število`
- `bool jeManj = String.Compare(n1,n2) < 0;`

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();
} // main
```

Program prevedemo in poženemo:

```
-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)
```

Prejšne 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.

Metode nad 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)
```

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, 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(int,int)	Metoda vrne podniz danega niza. Prvi argument pomeni začetni položaj (indeks) v nizu. Drugi argument v javi pomeni položaj (indeks) za zadnjim znakom podniza, medtem pa v C# 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.

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 ("").

```
C1:  static void Main(string[] args)
C2:  {
C3:       // Vnos niza
C4:       Console.Write("Vnesi niz: ");
C5:       string niz = Console.ReadLine();
C6:       // Pomožne spremenljivke
C7:       string samoglasniki = "aAeEiIoOuU"; // Niz samoglasnikov
C8:       string novNiz = ""; // Nov niz
C9:       int dolzina = niz.Length; // Dolžina prebranega niza
C10:      // Zamenjava samoglasnikov z zvezdico
C11:      for (int i = 0; i < dolzina; i++)
C12:      {
C13:          char znak = niz[i];
C14:          if (samoglasniki.IndexOf(znak) != -1)
C15:          {  // Znak je samoglasnik
```

```
C16:                novNiz = novNiz + '*'; // Samoglasnik zamenjamo z *
C17:            }
C18:            else
C19:            {
C20:                novNiz = novNiz + znak;//Ostali znaki ostanejo nespr.
C21:            }
C22:        }
C23:        // Izpis novega niza
C24:        Console.WriteLine("Spremenjen niz: " + novNiz);
C25:    }
```

Zapis na zaslonu:

```
Vnesi niz: Lepa Vida kolo vodi.
Spremenjen niz: L*p* V*d* k*l* v*d*.
```

Razlaga. Najprej preberemo niz (C5), nato deklariramo niz samoglasnikov (C7) in prazen niz (C8). Potem določimo dolžino niza niz (C9). Z zanko se sprehodimo preko vseh znakov v nizu. Znotraj zanke deklariramo spremenljivko znak in ji priredimo trenutni znak niza niz. V vrstico C14 preverimo, če je znak, katerega koda je shranjena v spremenljivki znak, podniz niza samoglasniki. Če je pogoj izpolnjen, se nizu novNiz doda znak '*' (C16), sicer pa se mu doda trenutno preverjeni znak (C20). Na koncu izvedemo izpis niza novNiz.

Galci in Rimljani

Verjetno je vsak med nami že kdaj prebral kak strip o Asterixu in Obelixu. Zato vemo, da se imena vseh Galcev zaključijo z "ix", na primer Asterix, Filix, Obelix, Dogmatix, itn. Napišimo program, ki bo prebral ime, nato pa bo izpisal "Galec!", če gre za galsko ime, v nasprotnem primeru pa bo izpisal "Rimljan!". Predpostavimo, da so vnesena imena dolga vsaj tri znake. Program zapišimo le v jeziku C#.

```
static void Main(string[] args)
{
     // Preberemo ime
     Console.Write("Vnesi ime: ");
     string ime = Console.ReadLine();
    // Podniz
    string podniz = ime.Substring(ime.Length - 2);
    podniz = podniz.ToLower(); // Da ne bo težav z velikimi črkami

    // Glede na ime izpišimo ustrezen tekst
    if(podniz.Equals("ix"))
    {
         Console.WriteLine("Galec!");
    }
    else
    {
         Console.WriteLine("Rimljan!");
    }
}
```

Kot smo povedali, metodo Equals() lahko nadomestimo z operatorjem enakosti (==). Zgornji program bi torej lahko zapisali tudi na naslednji način.

```
static void Main(string[] args)
{
    // Preberemo ime
    Console.Write("Vnesi ime: ");
    string ime = Console.ReadLine();
```

```
    // Podniz
    string podniz = ime.Substring(ime.Length - 2);
    podniz = podniz.ToLower(); // Da ne bo težav z velikimi črkami

    // Glede na ime izpišimo ustrezen tekst
    if (podniz == "ix")
    {
        Console.WriteLine("Galec!");
    }
    else
    {
        Console.WriteLine("Rimljan!");
    }
}
```

Zapis na zaslonu:

```
Vnesi ime: Filix
Galec!
```

Razlaga. Na začetku preko konzolnega okna preberemo ime osebe in ga shranimo v spremenljivki niz (C6). Potem določimo podniz iz zadnjih dveh znakov niza niz (C9). V vrstici C10 pretvorimo podniz podniz v male tiskane črke. Nato preverimo, če je podniz podniz enak nizu "ix" (C13). Če to drži, izpišemo "Galec!", sicer izpišemo "Rimljan!".

Razporeditev besed po abecedi

Napišimo program, ki bo prebral tri besede in jih izpisal po abecednem vrstnem redu. Na primer za prebrane besede buda, vescine in tempelj, naj program izpiše buda tempelj vescine. Predpostavimo, da so prebrane besede zapisane z malimi tiskanimi črkami.

Program sestavimo po korakih:
Najprej preko konzolnega okna določimo tri besede.

```
Console.Write("Vnesi prvo besedo: ");
string beseda1 = Console.ReadLine();
Console.Write("Vnesi drugo besedo: ");
string beseda2 = Console.ReadLine();
Console.Write("Vnesi tretjo besedo: ");
string beseda3 = Console.ReadLine();
```

Nato preverimo, če je beseda beseda3 abecedno pred besedo beseda1.

- Če je pogoj resničen, zamenjamo vrstni red besed beseda3 in beseda1.

```
if (beseda1.CompareTo(beseda3) > 0)
{
    string pom = beseda1;
    beseda1 = beseda3;
    beseda3 = pom;
}
```

V naslednjem koraku preverimo, če je beseda beseda2 abecedno pred besedo beseda1.

- Če je pogoj resničen, zamenjamo vrstni red besed beseda2 in beseda1.

```
if (beseda1.CompareTo(beseda2) > 0)
{
    string pom = beseda1;
    beseda1 = beseda2;
    beseda2 = pom;
```

```
}
```

Za konec preverimo še, če je beseda beseda3 abecedno pred besedo beseda2.

- Če je pogoj resničen, zamenjamo vrstni red besed beseda3 in beseda2.

```
if (beseda2.CompareTo(beseda3) > 0)
{
    string pom = beseda2;
    beseda2 = beseda3;
    beseda3 = pom;
}
```

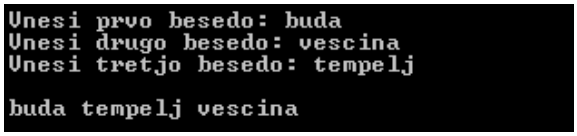
Besede so abecedno urejene, zato jih izpišemo. Ločimo jih s presledkom.

```
Console.WriteLine("\n" + beseda1 + " " + beseda2 + " " + beseda3);
```

Združimo posamezne korake v celoten program.

```
static void Main(string[] args)
{
    // Vnesene besede
    Console.Write("Vnesi prvo besedo: ");
    string beseda1 = Console.ReadLine();
    Console.Write("Vnesi drugo besedo: ");
    string beseda2 = Console.ReadLine();
    Console.Write("Vnesi tretjo besedo: ");
    string beseda3 = Console.ReadLine();

    // beseda3 je abecedno pred beseda1
    if(beseda1.CompareTo(beseda3) > 0)
    {
        string pom = beseda1;
        beseda1 = beseda3;
        beseda3 = pom;
    }

    // beseda2 je abecedno pred beseda1
    if(beseda1.CompareTo(beseda2) > 0)
    {
       string pom = beseda1;
       beseda1 = beseda2;
       beseda2 = pom;
    }

    // beseda3 je abecedno pred beseda2
    if(beseda2.CompareTo(beseda3) > 0)
    {
       string pom = beseda2;
       beseda2 = beseda3;
       beseda3 = pom;
    }

    // Izpis po abecedi
    Console.WriteLine("\n" + beseda1 + " " + beseda2 + " " + beseda3);
}
```

Zapis na zaslonu:



Še nekaj metod

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

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.

Nekaj primerov:

```
//dostop do posameznega znaka v nizu
string znaki = "abcdefg";
char a=znaki[0];  // 'a'
char b=znaki[1];  // 'b'

string beseda="Danes je lep dan!";
//sestavimo nov niz iz nekaterih znakov niza beseda
string znakiInPresledki = beseda[0]+ beseda[3]+ beseda[4]+ beseda[14];
//znakiInPresledki dobi vrednost "Desa"

//Uporaba metod StartsWith in EndsWith
bool zacneZabc = znaki.StartsWith("abc"); // true
bool koncaZabc = znaki.EndsWith("abc");   // false

//Uporaba metode IndexOf
string sola = "Tehniški Šolski Center Kranj";
int index1 = sola.IndexOf(" ");// 8, ker se string " " pojavi prvič na
                               // osmem mestu
int index2 = sola.IndexOf(' '); // 8, ker se znak ' ' pojavi prvič na osmem
                                // mestu
int index3 = sola.IndexOf("Center");// 16, ker se string "Center" pojavi
                                   // prvič na 16 mestu
int index4 = sola.LastIndexOf(" ");// 22, ker se string " " pojavi zadnjič
                                   // na 22 mestu

//Uporaba metod Remove, Insert in Replace
sola = sola.Remove(0, 9);  // Šolski center Kranj
sola = sola.Insert(sola.Length, ", Slovenija");// Šolski center Kranj
                                               //, Slovenija
sola = sola.Replace("Slovenija", "4000 Kranj");// Šolski center Kranj, 4000
                                               // Kranj

//Uporaba metod Substring, ToUpper in ToLower
string ime = "aNJA";
string prvaCrka = ime.Substring(0, 1).ToUpper(); // A
string drugeCrke = ime.Substring(1).ToLower();  // nja
ime = prvaCrka + drugeCrke;                      // Anja

//Kopiranje enga stringa v drug string
string s1 = "abc";
string s2 = s1; // string s2 dobi vrednost s1, torej s2 postane "abc"
s2 = "def";     // string s2 postane "def", string s1 pa se ne spremeni
string s3 = s1 + s2;  // strig s3 dobi vrednost "abcdef"
```

```
//Uporaba metode Substring
string polnoIme = "  Edward C Koop     ";      // "  Edward C Koop     "
polnoIme = polnoIme.Trim();                    // "Edward C Koop"
int prvipresledek = polnoIme.IndexOf(" ");    // 6
string lastnoIme = polnoIme.Substring(0,prvipresledek); // "Edward"

string naslov = " |34 Kališka ulica|Slovenija|Kranj|4000 ";
naslov = naslov.Trim();     // "|34 Kališka ulica|Slovenija|Kranj|4000"
naslov.Remove(0,1);         //iz naslova odstranimo prvi znak
string naslov1= naslov.Remove(naslov.Length-1,1);// iz naslova odstranimo
                                                 // zadnji znak

int indeksUlice = naslov.IndexOf("|") + 1;  // 1
int indeksDrzave = naslov.IndexOf("|", indeksUlice) + 1;  // 18
int indeksKraja = naslov.IndexOf("|", indeksDrzave) + 1;  // 28
int indeksPoste = naslov.IndexOf("|", indeksKraja) + 1;   // 34

string ulica = naslov.Substring(0, indeksDrzave-1); // 34 Kališka ulica
string mesto = naslov.Substring(indeksKraja,indeksPoste - indeksKraja - 1);
// Kranj
string posta = naslov.Substring(indeksPoste); // 4000

//Uporaba metode Insert
string telefonskaStevilka = "041827919";                    // "041827919"
telefonskaStevilka = telefonskaStevilka.Insert(3, "-");    // "041-827919"
telefonskaStevilka = telefonskaStevilka.Insert(7, "-");    // "041-827-919"

//Uporaba metode Replace
string datum = "21-08-2007";     // "21-08-2007"
datum = datum.Replace("-", "/");
```

Naloge za utrjevanje znanja iz nizov

Kaj izpišejo spodnji deli programov :

```
string niz = "Tantadruj";
char y;
y = niz[5];
Console.WriteLine("V nizu je crka " + y);
```

```
string niz1 = "Tantadruj";
int n = niz1.Length;
string niz2 = "";
int i = 0;
while(i < n) {
    char x = niz1[i];
    niz2 = x + niz2;
    i = i + 1;
}
Console.WriteLine(niz2);
```

```
string niz1 = "Tantadruj";
int n = niz1.Length;
string niz2 = "";
```

```
int i = 3;
while(i < n) {
    niz2 = niz2 + niz1[i];
    i = i + 1;
}
niz2 = niz2 + niz1[0] + niz1[1] + niz1[2];
Console.WriteLine(niz2);
```

_____

- Kaj izpiše naslednja metoda:

```
public static void Izpis (){
  string a = "moje bojno polje";
  System.Console.Write(a.IndexOf("oj"));
}
```

a) 6
b) 7
c) 1
d) 2
e) nič od navedenega, ampak _____

- Dani niz izpiši obrnjeno.

Primera:

-

"1. januar 2008"
Niz 1. januar 2008 izpisan obrnjeno je 8002 raunaj .1

"Lahek izpit"
Niz Lahek izpit izpisan obrnjeno je tipzi kehaL

- Na svetovnem spletu uporabljamo več kodnih tabel znakov, od katerih samo nekatere vsebujejo šumevce (č, š, ž). Da se šumevci ne pretvorijo v nerazumljive znake, jih pretvorimo v ustrezne sičnike (c, s, z) in jih take tudi izpišimo. Napišite program, ki prebere vrstico besedila s šumevci in na zaslon izpiše spremenjeno besedilo.

- Sestavite program, ki prebere niz in ga izpiše tako, da med znake besede vrine presledke.

Primer: Prebran niz je Lepa Anka kolo vodi.
L e p a A n k a k o l o v o d i .

- Napišite program Okvir, ki bo zahteval vnos niza. Nato ga bo izpisal na zaslon v "okvirju".

Primer:

Vnesi niz: `Mladost je norost, skače čez jarke, kjer je most.`

```
*-----------------------------------------------*
|Mladost je norost, skače čez jarke, kjer je most.|
*-----------------------------------------------*
```

- Sestavite program, ki bo zahteval vnos besede. Nato to besedo izpiše na zaslon tako, da bo beseda izpisana v obliki trikotnika.

Primer: Vnesena beseda je JEZIK

```
JEZIK
JEZI
```

```
JEZ
JE
J
```

- Napiši program, ki bo prebral niz in preveril prvi znak. Če prvi znak ni črka, naj izpiše "Niz se ne začne s črko." Če je prvi znak velika črka, naj izpiše "Niz niz ima veliko začetnico.". Če je prvi znak mala črka, naj niz spremeni, tako da bo prvi znak velika začetnica in izpiše: "Vneseni niz niz ni imel velike začetnice. Po popravku zgleda tako: popravljen niz niz.".

- Dana je deklaracija string stavek = "moj stavek"; S pomočjo lastnosti Length ter metode ToUpper spremeni prvo in zadnjo črko tega niza v veliko črko.

- Ugotovi na katerem mestu spremenljivke stavek tipa string se prvič pojavi veznik in (uporabi metodo IndexOf)?

- Iz spremenljivke stavek tipa string odstrani vse znake od 10. znaka naprej! Uporabi metodo Remove!

- Dana je spremenljivka stavek tipa string, spremenljivka ima že neko vrednost (vsebuje več kot 10 znakov).
  - Odstrani vse vodilne in končne presledke;
  - Vse črke v stringu stavek spremeni v velike črke angleške abecede;
  - Ugotovi in izpiši prvi in zadnji znak tega stringa;
  - Ugotovi in izpiši koliko znakov je v tem stringu;
  - V spremenljivko beseda (string) zapiši prvih pet znakov stringa stavek;
  - Zadnje tri znake tega stringa nadomesti s pikicami;
  - Na sredino stringa stavek vrini tri podčrtaje;
  - Iz stringa stavek odstrani vse znake od šestega znaka naprej.

- V nizu stavek vse številke nadomesti z besednim opisom (številko 1 zamenjaj z ena, številko 2 z dva, …)

- Dane so tri stpremenljivke tipa string st1, st2 in st3, ki so že inicializirani. Deklariraj spremenljivko st4 tipa string, katere vrednost naj bo sestavljena iz zadnjih znakov nizov v st1, st2 in st3!

- Dan je zelo dolg niz, sestavljen iz številk. Ugotovi, katerih številk je v nizu največ.

- Izpiši najprej vse samoglasnike poljubnega stavka, nato ta stavek izpiši navpično, vsak znak v svojo vrsto!

- Napiši program, ki prebere niz in vsak znak niza razmnoži tolikokrat kot mu naroči uporabnik z vnesenim številskim parametrom. Primer izpisa: Uporabnik je vnesel niz AVTO in zahteval, da se vsak znak ponovi trikrat.
  AAAVVVTTTOOO

- Napiši program, ki bo uporabniku omogočal vpis poljubnega stavka. Iz vpisanega stavka program nato izloči samoglasnike. Namesto ostalih znakov pa izpiše podčrtaj (_).

- Preberi poljuben stavek in ga izpiši navpično in nato še diagonalno (vsaka črka v svoji vrstici, za en znak bolj v desno!

- Napiši program, ki vse šumnike š, č in ž nekega stavka nadomesti z znaki s, c in z!

- Preberi poljuben string. Ugotovi in izpiši, koliko presledkov vsebuje.

- Preberi kemijsko formulo in jo izpiši tako, da so številke vrstico nižje. Če npr. prebereš formulo H2SO4, jo izpiši kot
  H SO
  2 4

- Dana sta dva niza, ime in priimek. Sestavi nov niz sifra tako, da najprej obrneš ime in priimek, nato pa izmenično iz zaporednih črk obrnjenega imena in priimka sestaviš nov niz. Iz nizov ime in priimek vzemi le toliko črk, kot znaša dolžina krajšega od obeh nizov.

- Beri znake toliko časa, da je vneseni znak enak pika (.). Prebrane znake nato izpiši v obliki stavka.

- Sestavi preprost program, ki bo kodiral in dekodiral kratka telefonska sporočila (sms-e). Program mora v obrniti vrstni red besed v sporočilu in vrstni red črk v besedi.

- Preberi 5 stavkov. Če je v stavku manj kot 5 znakov, ga ne upoštevaj (stavek continue). Na koncu ugotovi in izpiši najdaljši stavek.

Tabele

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 imajo vse enako ime (npr. tabela), razlikujejo pa se po **indeksu**, zaradi česar vsako od njih obravnavamo kot samostojno spremenljivko.

Preberi 5 števil in jih izpiši v obratnem vrstnem redu

Denimo, da 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 prazzaprav 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". No razlog je v tem, da so tisti naši xi deklarirani v prvi zanki in jih "drugje ni". Torej jih moramo deklarirati zunaj.

Spremenimo v

```
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 vnešeno število?

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).
Kako torej napišemo indekse? 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 spremenljike x1, x2, ... ne ve pa, da je x ime tabele, kjer bi radi imeli 5 indeksov (torej prostor za 5 števil).

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:

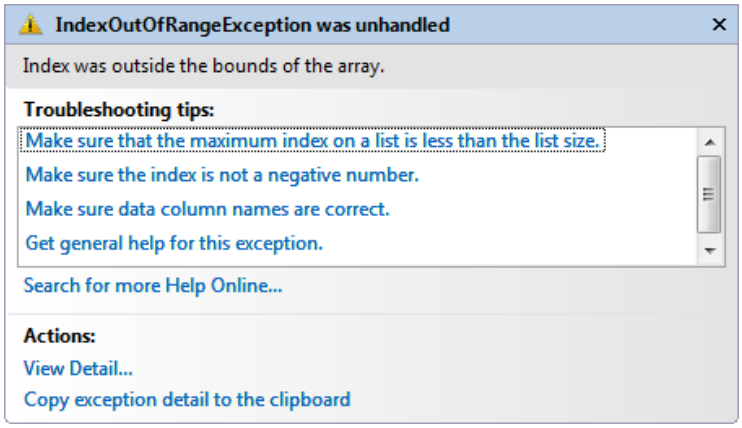
```
// tabela 10 celih števil
int[] stevila = new int[10];
// tabela 3 realnih števil
double[] cena = new double[3];
// tabela 4 znakov
char[] tabelaZnakov = new char[4];
// tabela 500 nizov
string[] ime = new string[500];
```

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



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 500
```

Izgled tabele po izvedbi zgornjih stavkov je torej takle:

100	80	300	400	380

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 željeni 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: ");
int dimenzija = int.Parse(Console.ReadLine());
    //Dimenzijo tabele določi uporabnik med izvajanjem!!!

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 konstanton. 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, omenjeno "kolobocijo" zgoraj 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*.

Indeksi v nizih in tabelah

Izraz, s katerim dostopamo do i-tega elementa tabele (imeTabele[i]), se v jeziku C# po videzu ujema z izrazom za dostopanje do i-tega znaka v nizu (imeNiza[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("Dolzina tabele je " + imena.Length + ".");
```

```
5:     Console.WriteLine("Ime na sredini tabele je  " +
6:                       imena[imena.Length / 2] + ".");
7: } // main
```

Program prevedemo in poženemo:

```
Dolzina tabele je 5.
Ime na sredini tabele je  Ana.
```

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:

indeks	*0*	*1*	*2*	*3*	*4*
vrednost	Jan	Matej	Ana	Grega	Sanja

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 " +
5:                      stevila.Length + ".");
6:  }
```

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("Dolzina tabele je " +stevila.Length + ".");
5:  }
```

Program prevedemo in poženemo:

```
Dolzina tabele je 3.
```

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);
} // main
```

Program prevedemo in poženemo:

```
System.String[]
```

Dobimo čuden izpis. 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]);
} // main
```

Program prevedemo in poženemo:

```
Jaka
Matej
Ana
Grega
Sanja
_
```

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]);
} // main
```

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*.

Primerjanje tabel

Oglejmo si še en primer pogoste napake. Denimo, da želimo primerjati dve tabeli. Zanima nas, če so vsi istoležni 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);
} //main
```

Program prevedemo in poženemo:

```
False
```

Opis programa.
Čeprav obe tabeli vsebujeta enake elemente, nam izpis 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: } //main
```

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.*

Zgledi

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]);
}
```

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  ");
    }
} //main
```

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));
} //main
```

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;
    // boben nak. stevili
    Random bob = new Random();
    // album
    bool[] album = new bool[st_slicicvalbumu];
    // 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.");
}
```

Program prevedemo in poženemo:

```
Da smo napolnili album, smo kupili 2265 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.

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 ('). V javi za razmejitev niza ne uporabljamo razmejitvenih znakov, temveč uporabimo razmejitveni niz (npr. split("/")), ki ga pišemo v navednicah ("). Metodo Split kot vedno v javi pišemo z malo začetnico.

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.".

```
C1:  public static void Main(string[] args)
C2:  {
C3:      // Vnos niza
C4:      Console.Write("Vnesi niz: ");
C5:      string niz = Console.ReadLine();
C6:
C7:      // Pretvorba niza v tabelo nizov
C8:      string[] tab = niz.Split(' ');
C9:
C10:     // Iskanje najdaljše besede
C11:     string pomozni = ""; // Najdaljša beseda (oz. podniz)
C12:     for (int i = 0; i < tab.Length; i++)
C13:     {
C14:         if (pomozni.Length < tab[i].Length)
```

```
C15:          {
C16:              pomozni = tab[i];
C17:          }
C18:      }
C19:
C20:      // Izpis najdaljše besede
C21:      Console.WriteLine("Najdaljsa beseda: " + pomozni);
C22:  }
```

Zapis na zaslonu:

```
Vnesi niz: Lep poletni dan.
Najdaljsa beseda: poletni
```

Razlaga

Najprej določimo niz niz (C5). Nato s pomočjo klica metode Split() določimo tabelo tab (C8). 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 (C11). Z zanko se sprehodimo po tabeli tab in poiščemo najdaljšo besedo. V vrstici C21 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 "".

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();
}
```

Zapis na zaslonu:

```
Velikost tabele: 5

Vnesi 1. element: 6
Vnesi 2. element: 7
Vnesi 3. element: 3
Vnesi 4. element: 9
Vnesi 5. element: 1

1 3 6 7 9
```

Vsota elementov

Predpostavimo, da imamo tabelo z naslednjimi elementi: 2, 5, 7, 1, 6, 10, 3, 8, 0 in 11. Napišimo program, ki bo določil in izpisal vsoto vseh elementov.

```
1:  public static void Main(string[] args)
2:  {
3:      // Deklariramo tabelo in jo napolnimo z elementi
4:      int[] tab = { 2, 5, 7, 1, 6, 10, 3, 8, 0, 11 };
5:
6:      // Vsota elementov
7:      int vsota = 0;
8:
9:      // Ugotovitev vsote elementov tabele
10:     for (int i = 0; i < tab.Length; i++)
11:      {
12:       int element = tab[i];
13:          vsota = vsota + element;
14:      }
15:
16:      // Izpis vsote elementov tabele
17:      Console.WriteLine("Vsota elementov je " + vsota + ".");
18:  }
```

Zapis na zaslonu:

```
Vsota elementov je 53.
```

Razlaga. Najprej deklarirajmo tabelo tab in jo napolnimo z elementi. Zatem deklariramo spremenljivko vsota in ji priredimo začetno vrednost 0.
V vrstici 9 uporabimo zanko for, ki ponavlja stavka v vrsticah 12 in 13 toliko časa, niso uporabljeni vsi elementi tabele tab. V vrstici 13 seštevamo elemente tabele in njihovo vsoto hranimo v spremenljivki vsota. Po koncu zanke izpišemo skupno vsoto vseh elementov (17).

Manjkajoča števila

Generirajmo tabelo 50 naključnih števil med 0 in 100 (obe meji štejemo zraven). S pomočjo zanke foreach 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 bo 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 nakljucnih stevil: ");
// 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 foreach, 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.

```
Console.WriteLine("\n\nStevila 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
    foreach (int el in tab)
    {
      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 + " ");
}
```

Na koncu gremo še v novo vrstico.

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

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 nakljucnih 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\nStevila 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 našli
        if (!nasli) Console.Write(i + " ");
    }

    // Nova vrstica
    Console.WriteLine();
}
```

V napisanem programu smo uporabili algoritem, katerega časovna zahtevnost je O (velikost tabele x število vseh števil). Če pa si lahko "privoščimo" uporabo dodatne tabele take velikosti, kot je vseh možnih števil (v našem primeru 101), lahko razvijemo boljši (hitrejši) algoritem.

Za izboljšavo napisanega programa bomo uporabili pomožno tabelo, ki bo nadomestila tisti del "starega" programa, kjer smo uporabili gnezdeno zanko. V tabeli bomo vsem elementom, katerih indeksi so enaki izbranim številom, nastavili vrednosti na *true*. Po nastavitvi vrednosti bomo poiskali vse tiste elemente, ki so ohranili vrednost false. Kadar bomo našli tak element, bomo izpisali njegov indeks. Le ta bo neizbrano število.

Najprej iz programa Stevila.cs uporabimo vrstice

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

Določimo tabelo logičnih vrednosti, katere velikost je mejaStevil + 1 (vsa števila med 0 in 100, vključno z mejama).

```
bool[] pom = new bool[mejaStevil + 1]; // Pomožna tabela
```

Ob deklaraciji se vsi elementi tabele avtomatsko postavijo na false. S tem smo predpostavili, da podatek i ni v tabeli tab.

Nato naredimo zanko foreach, s katero v tabeli pom na true nastavimo tiste elemente, ki imajo indekse, ki so v tabeli tab.

```
// Označimo elemente, ki so v tabeli tab
foreach (int el in tab)
{
    pom[el] = true; // Vrednost elementa nastavimo na true
}
```

Sedaj moramo poskrbeti le še za izpis.
Zato naredimo zanko for, ki se sprehodi po tabeli pom.
Če je vrednost trenutnega elementa false, izpišemo njegov indeks. Indekse ločimo s presledkom.

```
Console.WriteLine("\n\nStevila med 0 in " + mejaStevil +
                  ", ki niso v tej tabeli, so: ");
// Iskanje števil, ki se ne nahajajo v tabeli
for (int i = 0; i < pom.Length; i++)
{
    if (pom[i] == false)
        Console.Write(i + " ");
```

```
}
// Nova vrstica
Console.WriteLine();
```

Poglejmo sedaj še zapis izboljšanega 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 nakljucnih stevil: ");
    // Izpis elementov tabele tab
    foreach (int el in tab)
    {
        Console.Write(el + " "); // Izpis elementa, ločenega s presledkom
    }

    bool[] pom = new bool[mejaStevil + 1]; // Pomožna tabela

    // Označimo elemente, ki so v tabeli tab
    foreach (int el in tab)
    {
        pom[el] = true; // Vrednost elementa nastavimo na true
    }

    Console.WriteLine("\n\nStevila med 0 in " + mejaStevil +
     ", ki niso v tej tabeli, so: ");
    // Iskanje števil, ki se ne nahajajo v tabeli
    for (int i = 0; i < pom.Length; i++)
    {
        if (pom[i] == false)
            Console.Write(i + " ");
    }
    // Nova vrstica
    Console.WriteLine();
}
```

Zapis na zaslonu:

```
Izpis vsebine tabele nakljucnih stevil:
2 43 5 85 17 32 97 72 72 23 18 95 4 86 42 51 1 9 14 57 79 99 37 85 72 63 17 67 43 92 47 57
 9 79 21 77 8 62 6 36 18 75 50 6 67 81 24 54 11 72

Stevila med 0 in 100, ki niso v tej tabeli, so:
0 3 7 10 12 13 15 16 19 20 22 25 26 27 28 29 30 31 33 34 35 38 39 40 41 44 45 46 48 49 52
53 55 56 58 59 60 61 64 65 66 68 69 70 71 73 74 76 78 80 82 83 84 87 88 89 90 91 93 94 96
98 100
```

Naloge za utrjevanje znanja iz tabel

- Napišite program, ki prebere tri cela števila n, a in b ter tabelo velikosti n napolni z naključnimi števili med a in b. Tabelo naj potem program izpiše tako, da v vrstici izpiše po 5 števil.

- Sestavite program, ki bo prebral podatke v tabelo celih števil, nato pa preveril, ali tabela predstavlja permutacijo števil od 1 do n, kjer je n dolžina tabele.
  Namig: Tabela dolžine n predstavlja permutacijo, če v njej vsako naravno število od 1 do n nastopa natanko enkrat.

- Dana je tabela [1, 6, 4 ,2, 8, 3, 7]. Uredite jo po naslednjem postopku: Poiščite največji element in ga zamenjajte z zadnjim. Nato poiščite največji element od prvega do predzadnjega in ga zamenjajte s predzadnjim …

- Generirajte 1000 naključnih števil med 0 in 10 in preštejete, kolikokrat se vsak element pojavi.

- V tabelo dolžine n zapišite naključne male črke angleške abecede. Sestavite novo tabelo, v kateri se vsak element prvotne tabele ponovi natanko dvakrat. Novo tabelo izpišite na zaslon.
  Primer:
  Če je prvotna tabela ['a', 'n', 'c'], je nova tabela ['a', 'a', 'a', 'n', 'n', 'n', 'c', 'c','c'].

- V tabelo dolžine 30 zapišite naključna števila od 0 do 20 in jih izpišite kot množico: vsako samo enkrat.

- Sestavite program, ki napolni tabelo velikost n z naključnimi nenegativnimi celimi števili. Prepišite te elemente v novo tabelo tako, da je i-ti element razlika med vsoto vseh elementov prvotne tabele in i-tim elementom prvotne tabele. Izpišite obe tabeli tako, da v vrsti izpišete po 10 elementov.

Dvodimenzionalne in večdimenzionalne tabele

Dvodimenzionalne tabele vsebujejo vrstice in stolpce.

Splošna deklaracija dvodimenzionalne tabele:

```
Podatkovni_tip[ , ] ime_tabele = new Podatkovni_tip [vrstic, stolpcev];
```

Tudi dvodimenzionalno tabelo lahko deklariramo na tri načine:

- Najprej deklariramo tabelo (npr z imenom **tabela)** , kasneje pa ji določimo še velikost:

```
int[,] tabela;  //deklaracija tabelarične spremenljivke
tabela = new int[100 , 200]; //dvodimenzionalna tabela 100 vrstic in
200 stolpcev
```

- Ob deklaraciji tabelarične spremenljivke z operatorjem **new** takoj zasežemo pomnilnik:

```
int[ , ] tabela = new int[100 , 200];
```

- Tabelo najprej deklariramo, z operatorjem **new** zasežemo pomnilnik, nato pa jo še inicializiramo

```
int[ , ] tabela = new int[ 2, 3]  { { 1, 1 ,1 },{2, 2, 2 }};
```

Tudi v prvih dveh primerih se spremenljivke, samodejno inicializirajo (v našem primeru ima vseh 100 x 200 (to je 20.000) elementov tabele vrednost 0.

Primer:

Deklariraj dvodimenzionalno tabelo 10 x 10 celih števil in jo inicializiraj tako, da bodo po diagonali same enice, nad diagonalo same dvojke, pod diagonalo pa ničle.

```
int[,] tabela=new int[10,10];  //deklaracija tabelarične spremenljivke
//ob  deklaraciji  se  je  avtomatični  izvedla  še  inicializacija:  vse
tabelarične spremenljivke so //dobile vrednost 0

for (int i=0;i<10;i++)
{
    for (int j = 0; j < 10; j++)
    {
        if (i == j)
             tabela[i, j] = 1;
        else if (i<j)
             tabela[i, j] = 2;
    }
}
//tabelo še izpišimo v primerni obliki
for (int i = 0; i < 10; i++)
{
     for (int j = 0; j < 10; j++)
         Console.Write(tabela[i, j] + "  ");
     Console.WriteLine();
}
```

Poštevanka

Kreiraj dvodimenzionalno tabelo 10 x 10 celih števil. V posamezni vrstici naj bo poštevanka števil med 1 in 10. Tabelo na primeren način tudi izpiši.

```
int [,] tabela=new int[10,10];
for (int i=0;i<10;i++)
    for (int j=0;j<10;j++)
        tabela[i,j]=(i+1)*(j+1);

//Še izpis tabele
for (int i=0;i<10;i++)
{
    for (int j=0;j<10;j++)
        Console.Write(tabela[i,j] + "\t");
    Console.WriteLine();
}
```

Največja vrednost v tabeli

Kreiraj naključne elemente kvadratne matrike (4x4) celih števil med 0 in 10 in nato izpiši največjo vrednost v tabeli. Ugotovi in izpiši, na katerih mestih se ta največji element pojavi v tabeli (indeksi!).

```
int[,] mat = new int[4, 4];
int i, j, max = 0;

//generator naključnih vrednosti
Random Nakljucno = new Random();
for (i = 0; i < 4; i++)
{
    for (j = 0; j < 4; j++)
    {
         mat[i, j] = Nakljucno.Next(10);
         if ((i == 0) && (j == 0)) max = mat[i, j]; //iskanje najvecjega
         if (max < mat[i, j]) max = mat[i, j];
    }
}

Console.WriteLine("TABELA 4 x 4\n");
for (i = 0; i < 4; i++)
{ //izpis matrike
    for (j = 0; j < 4; j++)
    Console.Write(mat[i, j] + " ");
    Console.WriteLine();
}
Console.WriteLine("\nNajvecji element: " + max + "\n");
Console.WriteLine("Indeksi najvecjega elementa: ");
for (i = 0; i < 4; i++)  //izpis pozicij/indeksov najvecjega elementa
    for (j = 0; j < 4; j++)
        if (mat[i, j] == max) Console.WriteLine("[" + i + "," +  j + "]",
i, j);
```

Naloge

- Ustvari naključno tabelo 100 celih števil z vrednostmi med 10 in 100 in izpiši le tiste člene, ki so večji od zadnjega elementa v tej tabeli. Program dopolni tako, da ustvariš naključno dolgo tabelo (a ne krajšo od 10 in ne daljšo od 100 elementov).

- Zgeneriraj tabelo naključnih naravnih števil in izpiši vse elemente, ki so deljivi s številom, ki je na sredini tabele. Indeks elementa na sredini tabele je definiran kot (dolzina tabele) / 2.

- Sestavi program, ki bo uporabniku zastavil 10 računov za množenje celih števil. Eden izmed faktorjev naj bo naključno trimestno celo število, drugi izmed faktorjev pa mora biti naključno enomestno število. Števila in uporabnikove rezultate shranjuj v ustrezno tabelo. Tabelo NA KONCU obdelaj (preveri rezultate) in izpiši število pravilnih in število nepravilnih odgovorov.

- Napiši program, ki prebere tri cela števila: dimenzija, spMeja in zgMeja. Program naj nato tabelo velikosti dimenzija napolni z naključnimi števili med spMeja in zgMeja.

- Beri besede in jih shranjuj v tabelo dimenzije do 100 elementov. Ugotovi in izpiši najdaljšo besedo v tej tabeli. Napiši funkcijo, ki dobi za parameter to tabelo in poljubno celo število N in ki vrne podatek o tem, koliko besed v tej tabeli vsebuje več kot N znakov!

- Napiši program, ki bo izpisal dvodimenzionalno tabelo naključnih celih števil med 0 in 100 (dimenziji izbere uporabnik programa), poiskal njen največji element ter izračunal, za koliko se največji element razlikuje od povprečne vrednosti vseh elementov.

- Napiši program, ki prešteje število sodih in lihih elementov v naključno generirani tabeli celih števil. Nato iz začetne tabele sestavi dve tabeli. V prvi so samo soda števila, v drugi pa samo liha.

- Sestavi tabelo naključnih celih števil med 1 in 100. Prepiši jih v novo tabelo tako, da bodo v novi tabeli najprej elementi, ki imajo v prvi tabeli indekse 0, 2, 4, ..., potem pa še elementi z lihimi indeksi. Primer: Če ime prvotna tabela elemente 2, 4, 23, 5, 45, 6, 8 so v novi tabeli elementi razporejeni kot 2, 23, 45, 8, 4, 5, 6. 3. Izpiši obe tabeli po 10 v vrsto. Za realizacijo naloge napiši metode: generiraj - ustvari tabelo, prelozi - preloži v novo tabelo na zahtevan način in izpisi - v vsaki vrsti se izpiše 10 elementov

- Dan je zelo dolg niz, sestavljen iz števk. Ugotovi, katerih števk je v nizu največ.

- Dana je dvodimenzionalna tabela 10 x 10 celih števil. Napiši funkcijo, ki dobi to tabelo za parameter in ki vrne vsoto vseh sodih števil iz te tabele. Za obdelavo tabele uporabi zanko foreach.

- Dana je enodimenzionalna tabela 100 znakov. Ugotovi, ali se v tej tabeli nahaja določen znak.

- Dana je premica y = 2x+3. Napiši program, ki bo najprej zgeneriral tabelo TOCKE 10 naključnih celih števil med -20 in + 20 (POZOR! – števila morajo biti različna) nato pa izpisal koordinate desetih točk, ki ležijo na dani premici, pri čemer boš za prve koordinate vzel vrednosti iz tabele TOCKE.

- Ustvari naključno tabelo 100 celih števil z vrednostmi med 50 in 100 in nato izpiši le tiste člene, ki so večji od zadnjega. Program dopolni še tako, da ustvariš naključno dolgo tabelo (a ne krajšo od 10 in ne daljšo od 1000 elementov).

- Napiši program, ki prebere določeno število nizov in jih izpiše v obratnem vrstnem redu. Podatek o tem, koliko nizov bo vnesenih, prebereš na začetku programa. Pomagaj si s tabelo.

- Najprej smiselno (na dveh mestih) dopolni program

```
public class TabelaImen
{
  public static void Main (string[] args)
  {
```

```
    string[] ________= {"Mojca", "Katja", "Gašper", "Gabriela", "Tine"};

    System.Console.WriteLine("Dolžina tabele je " + _____________ + ".");
    System.Console.WriteLine("Ime v tabeli je " + imena[3] + ".");
   }
}
```

Nato odgovori: Kaj se izpiše, ko program prevedemo in poženemo?

- Dana sta ukaza

```
string[] stavek = new string[10];
stavek[0] = "Konec se bliza";
```

Kateri ukaz moramo uporabiti, če želimo izpisati dolžino niza "Konec se bliza"? Obkroži vse pravilne odgovore!

```
a) System.Console.WriteLine(stavek.Length());
b) System.Console.WriteLine(stavek[0].Length);
c) System.Console.WriteLine(stavek.Length);
d) System.Console.WriteLine(stavek[0].Length());
```

- Napiši program, ki prebere dve tabeli celih števil. Izpiše naj tiste vrednosti, ki se pojavijo v drugi tabeli in ne v prvi! Vemo, da so vse vrednosti v obeh tabelah med seboj različne (torej se v isti tabeli število nikoli ne ponovi)

  Primeri:

  - Če so v prvi tabeli podatki 3, 4, 2, 1 in v drugi 3, 4, 6, 5, naj program izpiše podatka 3 in 4.
  - Če so v prvi tabeli podatki 3, 41, 2, 11 in v drugi 3, 11, 41, 2 naj program ne izpiše nič.
  - Če so v prvi tabeli podatki 3, 41, 2, 11 in v drugi 100, 17, 11, 13, 41, 2 naj program izpiše 100, 17 in 13.

- Kaj izpiše spodnji del programa:

```
string[] niz = {"Tantadruj"};
int x;
x = niz.Length;
System.Console.WriteLine(x);
```

Razred Random

V programih se večkrat pojavi potreba po generiranju oz. uporabi naključnih števil. V **C#** je naključnim številom namenjen razred **Random**. Njegove metode omogočajo generiranje naključnih celih števil v poljubnem obsegu in iz poljubnega intervala, pa tudi generiranje naključnih števil tipa **double**.

Če hočemo delati z naključnimi števili, moramo najprej s pomočjo rezervirane besede **new** ustvariti nov **objekt** za generiranje naključnih števil:

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

Objekt **nakljucno** sedaj lahko uporabimo za generiranje naključnih števil. Za samo generiranje imamo na voljo dve metodi razreda **Random:**

Metoda	Razlaga
Next	Naključno celo število
NextDouble	Naključno število iz intervala 0.0 in 1.0

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; //naklučno število tipa double na intervalu od
                                               //0.0 do 1000
```

Vaja:

```
//Generirajmo nekaj naključnih celih in naključnih realnih števil
Random nakljucno = new Random();

//Generiramo 10 naključnih celih števil, večjih od 0 in MANJŠIH od 5
for (int j = 0; j < 10; j++)
     Console.Write(" {0,5} ", nakljucno.Next(5)); //izpis formatiran na 3 mesta
Console.WriteLine();

//Generirajmo 6 naključnih realnih števil
for (int j = 0; j < 6; j++)
    Console.Write(" {0,5:F3} ", nakljucno.NextDouble());//izpis na 5 mest, formatiran na 3
                                                        //decimalke
```

Vaja:

```
/*Kreiraj naključne elemente kvadratne matrike (4x4) celih števil med 0 in 10 in nato izpiši
največjo vrednost v tabeli. Ugotovi in izpiši, na katerih mestih se ta največji element pojavi
v tabeli (indeksi!).*/

int[,] mat = new int[4, 4];
int i, j, max = 0;

//generator naključnih vrednosti
Random Nakljucno = new Random();
for (i = 0; i < 4; i++)
{
    for (j = 0; j < 4; j++)
    {
         mat[i, j] = Nakljucno.Next(10);
         if ((i == 0) && (j == 0)) max = mat[i, j]; //iskanje najvecjega
         if (max < mat[i, j]) max = mat[i, j];
    }
}

Console.WriteLine("TABELA 4 x 4\n");
for (i = 0; i < 4; i++)
{ //izpis matrike
    for (j = 0; j < 4; j++)
    Console.Write(mat[i, j] + " ");
    Console.WriteLine();
}
Console.WriteLine("\nNajvecji element: {0}\n", max);
Console.WriteLine("Indeksi najvecjega elementa: ");
for (i = 0; i < 4; i++)  //izpis pozicij/indeksov najvecjega elementa
    for (j = 0; j < 4; j++)
        if (mat[i, j] == max) Console.WriteLine("[ {0} , {1} ]", i, j);
```

Vaja:

```
/*Deklariraj dvodimenzionalno tabelo celih števil dimenzije 20x20 in jo napolni po pravilu: na
diagonali naj bodo zaporedoma mnogokratniki števila 2 (2,4,6,…), pod diagonalo naj bodo taka
števila kot so na diagonali, nad diagonalo pa naj bodo naključna naravna števila med -5 in +5!

Napiši še funkcijo, ki dobi za parameter poljubno število  N in ki vrne podatek o tem, koliko
števil v zgornji tabeli je enakih N*/

//funkcija ima dva za drugi parameter dvodimenzionalno tabelo
static int enakihN(int N,int [,] stevila)
{
   int skupaj = 0;
   for (int i = 0; i < 20; i++)
      for (int j = 0; j < 20; j++)
         if (stevila[i,j] == N) skupaj++;
   return skupaj;
}

//glavni program
static void Main(string[] args)
{
     Random naklj = new Random();
     int[,] stevila = new int[20, 20];
     for (int i = 0; i < 20; i++)
         for (int j = 0; j < 20; j++)
             if (i == j) stevila[i, j] = (i + 1) * 2;
             else if (i > j) stevila[i,j] = stevila[j,j];
             else stevila[i, j] = naklj.Next(6) - 5;
     for (int i = 0; i < 20; i++)
     {
         for (int j = 0; j < 20; j++)
             Console.Write("{0,3}", stevila[i,j]); //izpis formatiran na 3 mesta
         Console.WriteLine();
     }
     Console.WriteLine("\n\nV tej tabeli je {0} števil ekakih 6!", enakihN(6,stevila));
}
```

Če tabelo ob deklaraciji tudi inicializiramo, je možna tudi deklaracija na krajši način:

```
Podatkovni_tip[] ime_tabele = {elementi tabele ločeni z vejico};
```

Primer:

```
int[] tab = { 0,1,2,3,4,5};//deklaracija in inicializacija enodimenzionalne tabele 6 elementov
```

Pri tabelaričnih spremenljivkah oz. pri tabelah obstajata dve metodi, s pomočjo katerih lahko ugotovimo število elementov v tabeli in dimenzijo tabele:

Metoda	Razlaga
Length	Število elementov v tabeli
Rank	Dimenzija tabele

Primer:

```
Random naklj = new Random();
int dimenzija = naklj.Next(10000)+1; //dimenzija tabele je naključno število med 1 in 10000

int[] tabela = new int[dimenzija];

for (int i = 0; i < dimenzija; i++)
   tabela[i] = naklj.Next(1000);//v tabeli so naključna števila med 0 in 1000

Console.WriteLine("V tej tabeli je " + tabela.Length + " elementov, dimenzija tabele pa je " +
        tabela.Rank);
```

Naloge:

- Iz treh naključnih števil med 0 in 9 sestavi naključno tromestno število med 0 in 999!
- Ustvari naključno tabelo 100 celih števil z vrednostmi med 10 in 100 in izpiši le tiste člene, ki so večji od zadnjega elementa v tej tabeli. Program dopolni tako, da ustvariš naključno dolgo tabelo (a ne krajšo od 10 in ne daljšo od 100 elementov).
- Zgeneriraj tabelo naključnih naravnih števil in izpiši vse elemente, ki so deljivi s številom, ki je na sredini tabele. Indeks elementa na sredini tabele je definiran kot (dolzina tabele) / 2.
- Sestavi program, ki bo uporabniku zastavil 10 računov za množenje celih števil. Eden izmed faktorjev naj bo naključno trimestno celo število, drugi izmed faktorjev pa mora biti naključno enomestno število. Števila in uporabnikove rezultate shranjuj v ustrezno tabelo. Tabelo NA KONCU obdelaj (preveri rezultate) in izpiši število pravilnih in število nepravilnih odgovorov.
- Napiši program, ki prebere tri cela števila: *dimenzija*, *spMeja* in *zgMeja*. Program naj nato tabelo velikosti *dimenzija* napolni z naključnimi števili med *spMeja* in *zgMeja*.
- Beri besede in jih shranjuj v tabelo dimenzije do 100 elementov. Ugotovi in izpiši najdaljšo besedo v tej tabeli. Napiši funkcijo, ki dobi za parameter to tabelo in poljubno celo število *N* in ki vrne podatek o tem, koliko besed v tej tabeli vsebuje več kot N znakov!
- Napiši program, ki bo izpisal dvodimenzionalno tabelo naključnih celih števil med 0 in 100 (dimenziji izbere uporabnik programa), poiskal njen največji element ter izračunal, za koliko se največji element razlikuje od povprečne vrednosti vseh elementov.
- Napiši program, ki prešteje število sodih in lihih elementov v naključno generirani tabeli celih števil. Nato iz začetne tabele sestavi dve tabeli. V prvi so samo soda števila, v drugi pa samo liha.

- Sestavi tabelo naključnih celih števil med 1 in 100. Prepiši jih v novo tabelo tako, da bodo v novi tabeli najprej elementi, ki imajo v prvi tabeli indekse 0, 2, 4, ..., potem pa še elementi z lihimi indeksi. Primer: Če ime prvotna tabela elemente 2, 4, 23, 5, 45, 6, 8 so v novi tabeli elementi razporejeni kot 2, 23, 45, 8, 4, 5, 6. 3. Izpiši obe tabeli po 10 v vrsto. Za realizacijo naloge napiši metode: **generiraj** - ustvari tabelo, **prelozi** - preloži v novo tabelo na zahtevan način in **izpisi** - v vsaki vrsti se izpiše 10 elementov

- Dan je zelo dolg niz, sestavljen iz števk. Ugotovi, katerih števk je v nizu največ.

Garbage Collector

Novo **instanco** (nov objekt) nekega razreda kreiramo s pomočjo rezervirane besede **new** (to smo spoznali že pri učenju osnov jezika **C#** oz. **C++** - poglavje razredi in objekti). Za sproščanje pomnilnika, ki ga zasedejo objekti kreirani z operatorjem **new** nam v okolju **Visual Studio .NET** ni potrebno skrbeti. **.NET Platforma** (**Framework**) uporablja za ta namen proces imenovan **garbage collector**, ki avtomatsko sprošča objekte, ki niso več v uporabi in s tem skrbi za upravljanje s pomnilnikom. V večini primerov se ta proces izvede avtomatično in se ga zelo redko sploh zavedamo. Ko sistem **garbage collector** zazna, da sistemu začne primanjkovati pomnilnika, oz. da je nezaposlen, sprosti pomnilnik vseh tistih objektov, ki niso več v funkciji. Preden jih počisti iz pomnilnika pa poskrbi, da se izvede metoda **Finalize** za vsak tak objekt, pa četudi je metoda privzeta, torej prazna (nismo napisali svoje kode, kaj naj se zgodi ob uničenju določenega objekta).

Zanka foreach

Zanka **foreach** je zanka, v kateri se ponavlja množica stavkov za vsak element neke tabele ali množice objektov. Zanke ne moremo uporabiti za spremembo elementov tabele, po kateri se sprehajamo, oz. za spremembo objektov. Na začetku **foreach** zanke je deklarirana spremenljivka poljubnega tipa, ki avtomatično pridobi vrednost posameznega elementa neke tabele, zaradi česar ima ta oblika zanke prednost pred klasičnim **for** stavkom za obdelavo neke tabele. Uporabimo jo lahko le za obdelavo cele tabele, ne pa tudi za obdelavo le njenega dela. Iteracija poteka vedno od indeksa **0** do indeksa **Length -1**, iteracija v obratni smeri pa **NI** možna.

Splošna oblika **foreach** zanke:

```
foreach ( tip spremenljivka in tabela)
{
  ..stavki..  //poljuben stavek ali več stavkov
}
```

Primer:

```
string znaki = "abcdefg";

//foreach zanka za dostop do vseh znakov v stringu
string znakiSPresledki = "";

foreach (char znak in znaki)
    znakiSPresledki += znak + " ";
//znakiSPresledki dobi vrednost "a b c d e f g "
```

Še en primer:

```
//deklaracija in inicializija enodimenzionalne tabele celih števil
int[] tab = { 0,1,2,3,4,5,6,7,8,9,10 };

Console.WriteLine("\nTabela tab vsebuje "+tab.Length+" elementov. \nTa tabela je "+tab.Rank+".
dimenzionalna");

Console.WriteLine(vsebina tabele: ");

foreach(int x in  tab)
    Console.Write(x+", "); //Izpis vsebine tabele
```

Rezervirana beseda *params*

C# omogoča kreiranje metode, ki ji lahko za parameter posredujemo neko tabelo, ki jo potem lahko v metodi obdelamo s pomočjo zanke ***foreach***. Rezervirana beseda ***params*** omogoča, da je število takih elementov v fazi načrtovanja metode še nedoločeno.

Primer:

```
//v glavi metode uporabimo rezervirano besedo params
public static void obdelajTabelo(params int[] poljubnaTabelaCelihStevil)
{
    int vsota = 0;
    foreach (int i in poljubnaTabelaCelihStevil)
    {
       vsota = vsota + i;
    }
}
```

```
//klic metode obdelajTabelo - metodi posredujemo poljubno število parametrov
obdelajTabelo(7, 8, 9, 10);

//deklaracija in inicilaizacija tabele 5 celih števil
int[] tabelaCelihStevil = new int[5] { 1, 2, 3, 4, 5 };
//klic metode obdelajTabelo - metodi posredujemo tabelo celih števil
obdelajTabelo(tabelaCelihStevil);
```

Vaja:

```
/*Deklariraj enodimenzionalno tabelo 10 decimalnih števil in jo napolni z naključnimi
decimalnimi števili med 0 in 1, zaokroženimi na 3 decimalke. Vsebino tabele nato izpiši s
pomočjo zanke foreach!*/

decimal[] stevila = new decimal[10];
string stringstevilo = "";
Random naklj = new Random();

for (int i = 0; i < 10; i++)
{
    //generiranje realnega števila med 0 in 1, zaokroženega na 3 decimalke
    stevila[i] = Math.Round((decimal)naklj.NextDouble(), 3);
}
Console.WriteLine("Tabela 10 naklj. realnih števil. Za izpis uporabimo foreach zanko)\n\n");

foreach (decimal stevilo in stevila)
{
    stringstevilo += stevilo.ToString() + "   ";
}
Console.WriteLine("Vsebina tabele:\n"+stringstevilo);
```

Vaja:

```
/*Deklariraj tabelo 7 stringov in jo ob deklaraciji inicializiraj tako, da bodo v njej dnevi v
tednu. Napiši metodo izpisiTabelo(dneviVTednu), ki dobi za parameter to tabelo in ki s pomočjo
zanke foreach to tabelo izpiše*/

//Metoda
static void izpisiTabelo(string[] tabela)
{
    foreach (string dan in tabela)
        Console.Write( dan+", ");

    Console.WriteLine();
}

//Glavni program
static void Main(string[] args)
{
     // Deklaracija in inicilaizacija tabele
     string[] dneviVTednu = new string[] { "Ned", "Pon", "Tor", "Sre", "Čet", "Pet", "Sob" };

     //Funkcija dobi za parameter tabelo:
     izpisiTabelo(dneviVTednu);
}
```

Vaja:

```
/*Generiraj dvodimenzionalno tabelo 3 x 5 celih števil in jo napolni za naključnimi celimi
števili med 0 in 100. Vsebino tabele nato izpiši s pomočjo zanke foreach. Ugotovi in izpiši
vsoto vseh elementov.*/

int sum = 0;
int[,] tab = new int[3, 5]; //Dvodimenzionalna tabela

Random naklj = new Random();

//inicializacija tabele
for (int i = 0; i < 3; i++)
```

```
    for (int j = 0; j < 5; j++)
        tab[i, j] = naklj.Next(100); //Naključna števila med 0 in vključno 99

Console.WriteLine("Izpis vseh elementov dvodimenzionalne tabele s pomočjo zanke foreach!\n");
// zanka foreach za izpis elementov dvodim. tabele
foreach (int stevilo in tab)
{
    Console.Write(stevilo+" ");
    sum += stevilo; //števila sproti seštevamo
}
Console.WriteLine("\n\nSkupna vsota elementov: " + sum+"\n\n");
```

Vaja:

```
/*generiraj tabelo 50 naključnih celih med vključno 0 in 100. S pomočjo foreach zanke
ugotovi in izpiši, katerih števil med 0 in 100 NI v tej tabeli*/

int[] stevila = new int[50];
Random naklj = new Random();

for (int i = 0; i < 50; i++)
     stevila[i] = naklj.Next(101); //naključna števila med 0 in 100

Console.WriteLine("Izpis vsebine tabele naključnih števil: \n(v vsaki vrstici je 10
           števil)\n");
for (int i = 0; i < 50; i++)
{
     Console.Write("{0,3} ",stevila[i]);
     if ((i+1) % 10 == 0) //po 10 števil v vsako vrstico izpisa
     Console.WriteLine();
}

Console.WriteLine("\n\nštevila med 0 in 100, ki jih NI v tej tabeli so:\n");
for (int st=0;st<=50; st++)
{
     bool found = false;  //logična spremenljivka za iskanje števila
     foreach (int x in stevila)
     {
        if (x == st)
        {
           found = true;  //število najdeno
           break;         //takojšnja prekinitev zanke foreach
        }
     }

     if (!found) //če števila v tabeli ni, ga izpišemo
         Console.Write(st+" ");
}
Console.WriteLine();
```

Naloge:

- S pomočjo **foreach** zanke izpiši najprej vse samoglasnike poljubnega stringa, nato ta stavek izpiši navpično, vsak znak v svojo vrsto!
- Dana je dvodimenzionalna tabela 10 x 10 celih števil. Napiši funkcijo, ki dobi to tabelo za parameter in ki vrne vsoto vseh sodih števil iz te tabele. Za obdelavo tabele uporabi zanko **foreach**.
- Dana je enodimenzionalna tabela 100 znakov. S pomočjo zanke **foreach** ugotovi, ali se v tej tabeli nahaja določen znak.
- Dana je premica y = 2x+3. Napiši program, ki bo najprej zgeneriral tabelo TOCKE 10 naključnih celih števil med -20 in + 20 (POZOR! – števila morajo biti različna) nato pa izpisal koordinate desetih točk, ki ležijo na dani premici, pri čemer boš za prve koordinate vzel vrednosti iz tabele TOCKE.

- Ustvari naključno tabelo 100 celih števil z vrednostmi med 50 in 100 in nato izpiši le tiste člene, ki so večji od zadnjega. Program dopolni še tako, da ustvariš naključno dolgo tabelo (a ne krajšo od 10 in ne daljšo od 1000 elementov).

- Napiši program, ki prebere niz in vsak znak niza razmnoži tolikokrat kot mu naroči uporabnik z vnesenim številskim parametrom. Primer izpisa: Uporabnik je vnesel niz AVTO in zahteval, da se vsak znak ponovi trikrat. Za obdelavo niza uporabi foreach zanko.

  AAAVVVTTTOOO

- Sestavi program, ki prebere celo število. Izpiše naj vsoto vseh števil od 1 do prebranega števila. Izpiše naj tudi vmesne vsote, vsako v svojo vrstico, na koncu pa še končno, skupno vsoto.

  Primer: Če je izbrano število 5, naj program izpiše :

  1
  3
  6
  10
  15

- Sestavi program, s katerim prebereš neko naravno število in ugotoviš, katero naravno število moraš povečat za 9-krat, da dobiš število, ki si ga izbral. Primer: Katero je število, ki ga moraš povečati za 9-krat, da dobiš število 810. To je število 90. Če tako število ne obstaja, naj nas program o tem obvesti!

- Napiši program, ki bo uporabniku omogočal vpis poljubnega stavka. Iz vpisanega stavka program nato izloči samoglasnike. Namesto ostalih znakov pa izpiše podčrtaj (_).
- Napiši program, ki prebere določeno število nizov in jih izpiše v obratnem vrstnem redu. Podatek o tem, koliko nizov bo vnesenih, prebereš na začetku programa. Pomagaj si s tabelo.

Strukture

Struktura je sestavljeni podatkovni tip, sestavljen največkrat iz enostavnih podatkovnih tipov. V strukturo združimo podatke različnih tipov, ki skupaj tvorijo neko celoto (npr. podatki o določeni osebi, podatki o določenem artiklu, podatki o avtomobilu, …).

Strukture so spremenljivke vrednostnega tipa, kar pomeni, da novo spremenljivko tipa struktura deklariramo tako kot običajno vrednostno spremenljivko (brez uporabe operatorja **new**). To pa hkrati pomeni, da ko neko strukturo deklariramo, **njene komponente še nimajo začetne vrednosti** (niso inicializirane).

Splošna deklaracija strukture :

```
struct <ime strukture>
{
    <public><tip1> <ime prvega člana>;
    <public><tip2> <ime drugega člana>;
    // ….
};  //Podpičje za zaključek strukture ni obvezno
```

Struktura se začne z rezervirano besedo **struct**, ki ji sledi ime strukture, nato pa v zavitih oklepajih naštejemo člane strukture. Besedica **public** naj nam zaenkrat pomeni, da so člani te strukture javni in imamo do njih neomejen dostop.

Primer:

```
struct obcan   //deklaracija MORA biti pred glavnim programom
{
   public string ime;
   public string priimek;
   public string EMSO;
   public long davcna;
};
obcan o; //deklaracija spremenljivke obc tipa obcan
//POZOR: elementi strukture obcan še NISO inicializirani, nimajo začetne vrednosti
```

Do elementov strukture dostopamo s pomočjo operatorja pika ('.'). Nad spremenljivkami izpeljanimi iz neke strukture lahko izvajamo vse operacije, ki jih poznamo nad osnovnimi podatkovnimi tipi. **Pomembno pa je, da strukturo deklariramo pred glavnim programom**.

Primer:

```
struct obcan
{
    public string ime;
    public string priimek;
    public string EMSO;
    public long davcna;
};

static void Main(string[] args)
{
    obcan o;
    Console.Write("Vpiši ime            : ");
    o.ime=Console.ReadLine();
```

```
    Console.Write("      priimek         : ");
    o.priimek=Console.ReadLine();
    Console.Write("      EMŠO            : ");
    o.EMSO=Console.ReadLine();
    Console.Write("      davčna številka : ");
    o.davcna=Convert.ToInt64(Console.ReadLine());
    Console.WriteLine("\nPodatki o občanu : \n");
    Console.WriteLine("    Priimek in ime : "+o.priimek+" "+o.ime);
    Console.WriteLine("    EMŠO           : "+o.EMSO);
    Console.WriteLine("    Davčna številka: "+o.davcna);
    Console.WriteLine();
}
```

Vaja:

```
/*Deklariraj strukturo ulomek z dvema komponentama: števec in imenovalec ulomka. Napiši
funkcijo za vnos podatkov za ta dva ulomka. Ulomka nato seštej in rezultat zapiši v tretji
ulomek, ki ga nato v primerni obliki tudi izpiši. Napiši tudi metodo za krajšanje ulomka*/

struct ulomek
{
    public int stevec;
    public int imenovalec;
};

static void vnos(out ulomek u) //metoda za vnos števca in imenovalca ulomka
{
    Console.WriteLine("Vnesi števec in menovalec ulomka(Imenovalec mora biti različen od 0)");
    Console.Write("Števec: ");
    u.stevec=Convert.ToInt32(Console.ReadLine());
    do
    {
        Console.Write("Imenovalec: ");
        u.imenovalec = Convert.ToInt32(Console.ReadLine());
    }
    while (u.imenovalec == 0);
}

static void izpis(ulomek u) //metoda za izpis ulomka
{
    Console.WriteLine(u.stevec + " / " + u.imenovalec);
}

static void krajsaj(ref ulomek u)//Metoda za krajšanje ulomka. Parameter klican po referenci
{
    for (int i=2;i<=u.stevec;i++)
      if (u.stevec % i == 0 && u.imenovalec % i == 0)
      {
         do  //ker obstaja možnost, da lahko ulomek z istim številom krajšamo večkrat!!!
         {
             u.stevec = u.stevec / i;
             u.imenovalec=u.imenovalec/i;
         }
         while (u.stevec % i == 0 && u.imenovalec % i == 0);
      }
}

static void Main(string[] args)
{
    ulomek u1, u2;//dva nova ulomka
    vnos(out u1); //klic funkcije za vnos števca in menovalca ulomka u1 (klic je po referenci
                  //out, ker ulomka še nista inicializirana)
    vnos(out u2); //klic funkcije za vnos števca in menovalca ulomka u2 (klic je po referenci
                  //out, ker ulomka še nista inicializirana)
    //seštejmo oba ulomka
    ulomek u3;
    u3.imenovalec=u1.imenovalec*u2.imenovalec;
    u3.stevec = u1.stevec * u2.imenovalec + u2.stevec * u1.imenovalec;
    Console.Write("\nVsota dveh ulomkov \n"+ u1.stevec + " / " + u1.imenovalec + " + " +
                   u2.stevec + " / " + u2.imenovalec + " = ");
    izpis(u3); //klic metode za izpis ulomka u3
    //okrajšajmo ulomek u3
    krajsaj(ref u3); //klic metode za krajšanje ulomka - parameter klican po referenci ref,
                     //ker je ulomek u3 ŽE inicializiran
```

```
            Console.WriteLine("Okrajšana vsota: " + u3.stevec + " / " + u3.imenovalec);
}
```

Vaja:

```
/*Napišimo program za vodenje evidence podrtih kegljev za dva tekmovalca. Kreiraj strukturo
tekmovalec, ki ima dve komponenti: naziv tekmovalca in tabelo v katero boš vpisoval posamezne
mete. Preberi imeni tekmovalcev, nato pa še posamezne mete posameznega tekmovalca (po 10
metov). Na koncu ugotovi zmagovalca in napiši metodo, ki vrne povprečno število podrtih
kegljev izbranega tekmovalca.*/

struct tekmovalec
{
    public String naziv;
    public int [] keglji;  //napoved tabele: DIMENZIJA tabele še NI določena!!!
}

static double povp(tekmovalec T,int N)
{
    int suma = 0;
    for (int i = 0; i < N; i++) //seštejemo mete temovalca
        suma += T.keglji[i];
    return (Math.Round((double)suma / N,2));
}

static void Main(string[] args)
{
    tekmovalec A,B;  //dve vrednostni spremenljivki tipa struktura
    int N = 2;//število metov posameznega tekmovalca
    A.keglji=new int[N];   //na kopici ustvarimo tabelo za vnos podatkov o podrtih kegljih
                           //prvega tekmovalca
    B.keglji = new int[N]; //na kopici ustvarimo tabelo za vnos podatkov o podrtih kegljih
                           //drugega tekmovalca
    Console.Write("Naziv prvega tekmovalca: ");
    A.naziv=Console.ReadLine();
    Console.Write("Naziv drugega tekmovalca: ");
    B.naziv=Console.ReadLine();
    Console.WriteLine("Vnos posameznih metov obeh tekmovalcev");
    Console.WriteLine("---------------------------------------");
    Console.WriteLine("{0,-12}              {1,-12}", A.naziv, B.naziv);
    Console.WriteLine("------------              ------------");
    for (int i = 0; i < N; i++)
    {
        A.keglji[i] = Convert.ToInt32(Console.ReadLine());
        /*metoda SetCursorPosition postavi kurzor na pravilno pozicijo za vnos podatkov.
         Metoda ima dva parametra -  oddaljenost od levega roba in oddaljenost od zgornjega
         roba (merjeno v znakih) */
        Console.SetCursorPosition(26, 6 + i);
        B.keglji[i] = Convert.ToInt32(Console.ReadLine());
    }
    int sumaA = 0, sumaB = 0;
    for (int i=0;i<N;i++) //seštejemo mete posameznih temovalcev
    {
         sumaA += A.keglji[i];
         sumaB += B.keglji[i];
    }

    if (sumaA > sumaB)
        Console.WriteLine("Zmagovalec je " + A.naziv + ". Skupaj je podrl " + sumaA + "
                          kegljev!");
    else if (sumaA==sumaB)
        Console.WriteLine("Tekmovalca sta izenačena. Oba sta podrla po " + sumaA + "
                          kegljev!");
    else
        Console.WriteLine("Zmagovalec je " + B.naziv + ". Skupaj je podrl " + sumaB + "
                          kegljev!");

    Console.WriteLine("Povprečno število metov prvega tekmovalca je
                      "+povp(A,A.keglji.Length));
    Console.WriteLine("Povprečno število metov drugega tekmovalca je
                      "+povp(B,B.keglji.Length));
}
```

Spremenljivko tipa struktura (objekt) pa lahko ustvarimo tido na kopici, torej z uporabo operatorja new:

```
struct kontinent
{
   public string naziv;
   public long stprebivalcev;
};

static void Main(string[] args)
{
   kontinent k1;//k1 je vrednostna spremenljivka tipa kontinent
   kontinent k2 = new kontinent();//k2 smo ustvarili na kopici-referenčna spremenlj. (objekt)

   //od tu dalje delamo z obema spremenljivkama enako
   k1.naziv = "Evropa";
   k2.naziv = "Azija";
   Console.Write(k1.naziv+" - število prebivalcev: ");
   k1.stprebivalcev = Convert.ToInt64(Console.ReadLine());
   Console.Write(k2.naziv+" - število prebivalcev: ");
   k2.stprebivalcev = Convert.ToInt64(Console.ReadLine());
}
```

Naloge:

- Napiši program, ki v strukturo podatki prebere podatke o šoli (v strukturi je ime šole, naslov, poštna številka kraj in telefon). Vpisane podatke nato izpiše v obliki: "Hodim v šolo ..., ki je na naslovu: ... Telefonska številka je ..., poštna ..., kraj …!

- Kreiraj strukturo *pravokotnik* z dvema elementoma – dolžino in višino pravokotnika. Ustvari vrednostno spremenljivko *P1* tipa pravokotnik in referenčno spremenljivko objekt) *P2* izpeljano iz strukture pravokotnik. Napiši funkcijo za vnos podatkov v spremenljivki *P1* oz. *P2*. Izračunaj in izpiši obseg in ploščino obeh pravokotnikov!

- Kreiraj strukturo *spricevalo* s tremi komponentami: ime in priimek (string), predmetnik (enodimenzionalna tabela 10 stringov, v katere bomo shranjevali imena predmetov), ter ocene (enodimenzionalna tabela 10 celih števil - za toliko ocen, kot je bilo predmetov). Napiši funkcijo
  - za vnos podatkov v tako strukturo;
  - za izpis spričevala. Na koncu izpiši še uspeh(5-odl, 4-pdb, 3-db, 2-zd, 1-nzd);
  - ki izračuna in vrne povprečno oceno.

Tabele struktur

Struktura lahko nastopa tudi kot tabelarična spremenljivka. Do komponent posamezne spremenljivke dostopamo preko indeksa tabelaričnega elementa in operatorja pika. Tabela struktur tako predstavlja zelo kompaktno podatkovno strukturo in ohranja vse prednosti in jih prinašajo operacije nad tabelami in strukturami.

Primer:

```
struct tekoci    //deklaracija strukture MORA biti pred glavnim programom
{
    public int zapst;
    public string naziv;
    public decimal znesek;
};

static void Main(string[] args)
{
     tekoci[] racuni = new tekoci[10];  //tabela 10 struktur

     Console.WriteLine("Vnos podatkov v tabelo: \n");
     for (int i = 0; i < 10; i++)
     {
```

```
            Console.WriteLine("\nKomitent številka " + (i + 1));
            racuni[i].zapst=i+1;
            Console.Write("Naziv: ");
            racuni[i].naziv = Console.ReadLine();
            Console.Write("Znesek: ");
            racuni[i].znesek = Convert.ToDecimal(Console.ReadLine());
        }
}
```

Vaja:

```
/*Deklariraj strukturo slaščica z dvema komponentama: ime slaščice in cena. Napiši zanko za
vnos podatkov v tabelo N takih struktur. Napiši metodi za iskanje in izpis najdražje slaščice
v tej tabeli in metodo, ki vrne število slaščic dražjih od nekega poljubnega zneska*/

struct slascica  //deklaracija strukture slascice
{
    public string ime;
    public double cena;
};


//metoda, ki poišče in izpiše ime in ceno najdražje slaščice
static void najdrazja(slascica[] slascicarna)
{
    int naj = 0; //predpostavimo, da je najdražja slaščica v tabeli tista, ki ima indeks 0
    for (int i = 1; i < slascicarna.Length; i++)
    {
        if (slascicarna[i].cena > slascicarna[naj].cena)
        naj = i;
    }
Console.WriteLine("\nNajdrazja slascica v tabeli je " + slascicarna[naj].ime + ",  njena cena
                  pa je " + slascicarna[naj].cena);
}

//metoda, ki vrne število slaščic, ki so dražje od 200 sit!
static int drazje(double znesek, slascica[] slascicarna)
{
    int stevilo = 0; //začetna vrednost števila slaščic dražjih od 200 SIT
    for (int i = 1; i < slascicarna.Length; i++)
    {
       if (slascicarna[i].cena > znesek)
          stevilo++;
    }
    return stevilo;
}

static void Main(string[] args)
{
    int N = 5;
    slascica[] slascicarna = new slascica[N];  //tabela 50 slaščic
    //vnos podatkov tabelo
    for (int i = 0; i < N; i++)
    {
        Console.Write("Slaščica: ");
        slascicarna[i].ime = Console.ReadLine();
        Console.Write("Cena: ");
        slascicarna[i].cena = Convert.ToDouble(Console.ReadLine());
    }
    //funkcija, ki poišče in izpiše ime najdražje slaščice v tabeli
    najdrazja(slascicarna);
    //klic funkcije, ki vrne število slaščic, ki so dražje od 200 SIT
    Console.WriteLine("\nŠtev. slaščic, ki so dražje od 200 SIT je " +
                      drazje(200.00,slascicarna));
}
```

Vaja:

```
/*Kreiraj strukturo tocka2D z dvema komponentama - koordinatama tipa int. Napiši funkcije za
vnos podatkov, izračun razdalje posamezne točke od koordinatnega izhodišča in funkcijo za
izračun razdalje med točkama!*/
```

```
struct tocka2D {
    public int x,y;
};

//parameter je klican po referenci, ker pa še ni inicializiran uporabimo operator out
static void vnos(out tocka2D T)
{
    Console.Write("Prva koordinata:  ");
    T.x=Convert.ToInt32(Console.ReadLine());
    Console.Write("Druga koordinata: ");
    T.y=Convert.ToInt32(Console.ReadLine());
}

static double razdalja (tocka2D T)
{
    double pomozna = Math.Sqrt(Math.Pow(T.x, 2) + Math.Pow(T.y, 2));
    return (Math.Round(pomozna,2));
}

static double razdalja(tocka2D T1, tocka2D T2)
{
    double pomozna = Math.Sqrt(Math.Pow(T1.x - T2.x, 2) + Math.Pow(T1.y - T2.y, 2));
    return (Math.Round(pomozna,2));
}

static void najboljoddaljena(tocka2D[] tabela)
{
    int naj = 0; //spremenljivka 'naj' predstavlja INDEKS najbolj oddaljene točke; na začetku
                 določimo da je to kar prva točka v tabeli
    for (int i = 1; i < 100; i++)
    {
       if(razdalja(tabela[i])>razdalja(tabela[naj]))
          naj=i; //če smo našli bolj oddaljeno točko, si zapomnimo njen indeks v tabeli
    }
    Console.WriteLine("Koordinati točke, ki je najbolj oddaljena od izhodišča: ( tabela[naj].x
             +" , "+tabela[naj].x+" )");
}

static void Main(string[] args)
{
    tocka2D A,B,C;
    Console.WriteLine("Vnos koordinat tocke A");
    vnos(out A); //ker struktura A še ni inicializirana, smo za klic po referenci uporabili
                    operator out!!!
    Console.WriteLine("Vnos koordinat tocke B");
    vnos(out B); //ker struktura B še ni inicializirana, smo za klic po referenci uporabili
                    operator out!!!

    Console.WriteLine("Tocka A je od koord. izhodišča oddaljena za "+razdalja(A)+" enot");
    Console.WriteLine("Razdalja med točkama A in B je "+razdalja(A,B)+" enot!");

    /*VAJE:
      Preberi podatke še za točko C! Kolikšen je obseg trikotnika ABC?
      Katera od tock A, B in C je najbolj oddaljena od koord.izhodišča?
      Napiši funkcijo HERON, ki ima za parametre tri točke, vrne pa ploščino trikotnika ki ga
      tvorijo; klic funkcije npr: double P=HERON(A,B,C);
      Heronova formula p=sqrt(s*(s-a)*(s-b)*(s-c)), s=(a+b+c)/2  */

     /*Deklarirajmo še tabelo 100 naključnih točk (koordinati x in y sta naključni števili med
      0 in 100)in s pomočjo funkcije ugotovimo in izpišimo koordinate tiste točke, ki je
      najdalj od izhodišča */

     tocka2D[] tabela = new tocka2D[100];//deklaracija tabele 100 točk tipa tocka2D
     Random naklj = new Random();
     for (int i = 0; i < 100; i++)  //določimo naključne koordinate vsem 100 točkam
     {
        tabela[i].x = naklj.Next(100);
        tabela[i].y = naklj.Next(100);
     }
     najboljoddaljena(tabela);  //klic funkcije, ki ugotovi in izpiše koordinati točke, ki je
                                  najbolj oddaljena od koordinatnega izhodišča
}
```

Naloge:

- Kreiraj strukturo **tocka**, ki predstavlja točko v dvodimenzionalnem koordinatnem sistemu (elementa strukture sta koordinati, obe sta celoštevilskega tipa). Nato deklariraj tabelo 100 struktur tipa **tocka** in jo inicializiraj tako, da bodo koordinate točk naključna cela števila med -100 in +100. Napiši funkcijo, ki dobi za parameter to tabelo in ki
  - ugotovi in v primerni obliki izpiše koordinati točke, ki je najbolj oddaljena od koordinatnega izhodišča (razdalja točke od izhodišča je kvadratni koren iz vsote kvadratov obeh koordinat!);
  - vrne število točk, ki ležijo v prvem kvadrantu (obe koordinati >0).

- Napiši program, s katerim bi za 20 trgovin (tabela trgovin!) vnesel podatke o nazivu trgovine in ceni za 1 kg kruha. Napiši funkcijo
  - ki dobi za parameter to tabelo, vrne pa naziv trgovine z najdražjim kruhom;
  - ki dobi za parameter to tabelo, vrne pa povprečno ceno kruha.

- Deklariraj strukturo, ki naj predstavlja nek ulomek. Nato deklariraj enodimenzionalno tabelo 100 takih struktur. Napiši
  - Stavke, ki vsem ulomkom v tej tabeli priredijo naključne vrednosti števcev in imenovalcev (vrednosti naj bodo med 1 in 10);
  - funkcijo, ki dobi kot argument dva indeksa te tabele, vrne pa vsoto ulomkov na ustreznih mestih tabele (vrednost, ki jo funkcija vrne naj bo tipa **double**);
  - funkcijo, ki dobi za parameter to tabelo, vrne pa največji ulomek te tabele (funkcija torej vrne strukturo!).

- Deklariraj strukturo, s katero boš opisal podatke o nekem rabljenem vozilu (tip vozila, prevoženi km in cena). Kreiraj tabelo 10 takih struktur. Napiši program, ki bo v obliki menija nudil naslednje opcije:
  - Vnos novega vozila v seznam;
  - Izpis vseh podatkov o vozilih vozil dražjih od 10.000 EUR na zaslon.

- Za zaposlenega potrebujejo v podjetju naslednje podatke:

  ime, *priimek*, *šifra* (celo število) in *osebne dohodke za zadnjih 12 mesecev* (tabela 12 realnih števil)

  Napiši:
  - deklaracijo za tako strukturo;
  - deklaracijo spremenljivk *zaposl1* in *zaposl2* za tako strukturo;
  - sklic na plačo z indeksom nič spremenljivke *zaposl1*;
  - deklaracijo za dinamično ustvarjen objekt strukture (ustvarjen na kopici);
  - sklic na priimek dinamično ustvarjene strukture:
  - ukaz za sprostitev pomnilnika, zasedenega z dinamično ustvarjenim objektom strukture;
  - deklaracijo tabele 10 primerkov struktur zgornjega tipa;
  - sklic na ime primerka strukture z indeksom 5.

- Dijak obiskuje 5 predmetov. Šolsko leto ima tri ocenjevalna obdobja. V vsakem ocenjevalnem obdobju dobi dijak pri vsakem predmetu dve oceni. Deklariraj ustrezno podatkovno strukturo in napiši podprogram, za vnos in izpis vseh ocen za enega dijaka!

Gnezdene strukture

Strukture so lahko tudi gnezdene (struktura v strukturi). Najprej deklariramo strukturo, ki bo vgnezdena v drugi strukturi, nato pa še samo strukturo. Do članov strukture dostopamo z operatorjem pika.

Splošna deklaracija gnezdene strukture :

```
struct <ime gnezdene strukture>  //deklaracija strukture, ki bo vgnezdena v
```

```
{                                    //drugi strukturi
    <public><tip1> <ime prvega člana>;
    <public><tip2> <ime drugega člana>;
    // ….
};
```

```
struct <ime strukture>                    //struktura
{
    <public><tip1> <ime prvega člana>;
    <public><tip2> <ime drugega člana>;
    <gnezdena struktura> <ime člana>    //gnezdena struktura
    // ….
};
```

Primer:

```
struct rojen   //struktura, ki bo vgnezdena
{
    public int dan,mesec,leto;
};

struct oseba
{
    public string priimek,ime,naslov;
    public rojen rojstvo;  //gnezdena struktura
};

static void vnos(out oseba os)//metoda za vnos/branje podatkov v strukturo
{
    Console.Write("Priimek: ");
    os.priimek=Console.ReadLine();
    Console.Write("Ime: ");
    os.ime=Console.ReadLine();
    Console.Write("Naslov: ");
    os.naslov=Console.ReadLine();
    Console.Write("Podatki o rojstvu -   Dan: ");
    os.rojstvo.dan=Convert.ToInt32(Console.ReadLine());
    Console.Write("                    Mesec: ");
    os.rojstvo.mesec=Convert.ToInt32(Console.ReadLine());
    Console.Write("                     Leto: ");
    os.rojstvo.leto=Convert.ToInt32(Console.ReadLine());
}

static void Main(string[] args)
{
    oseba nekdo, Mary, Angelika;  //tri spremenljivke tipa oseba (tri nove strukture)
    //podatke za osebo nekdo določimo npr. kar takole:
    nekdo.priimek = "Andersen";
    nekdo.ime = "Tim";
    nekdo.naslov = "Beverly Hills 123";
    nekdo.rojstvo.dan = 12;
    nekdo.rojstvo.mesec = 6;
    nekdo.rojstvo.leto = 2000;

   vnos(out Mary);//podatke za osebo Mary preberemo s pomočjo funkcije
   //Napiši metodo za izpis podatkov o neki spremenljivki tipa oseba (klic npr : izpis(nekdo)
   //Napiši funkcijo, ki dobi za parameter dve osebi in ki vrne starejšo od obeh oseb
}
```

Vaja:

```
/*Deklariraj strukturo dijak, nato pa strukturo tecaj, v kateri je vgnezdena tabela struktur
tipa dijak. Preberi podatke, nato pa izračunaj povprečno starost vseh dijakov!*/

struct dijak              //struktura, ki bo vgnezdena v drugo strukturo
{
   public string ime;
   public string priimek;
```

```
   public int dan;
   public int mesec;
   public int leto;
};

struct krozek
{
   public int Id;
   public string naziv;
   public dijak [] Dijaki;  //tabela dijakov - vsak dijak je gnezdena struktura
};

const int N=10;   //definicija celoštevilske konstante (dimenzija tabele)

static void Main(string[] args)
{
   krozek T;                  //nova spremenljivka tipa krozek (struktura)
   Console.WriteLine("Vnesi podatke o tečaju: ");
   Console.Write("Številka tečaja: ");
   T.Id=Convert.ToInt32(Console.ReadLine());
   Console.Write("Naziv tečaja: ");
   T.naziv=Console.ReadLine();
   Console.WriteLine("Podatki o udeležencih tečaja: ");
   T.Dijaki = new dijak[N];   //inicializacija tabele
   for (int i = 0; i < N; i++)
   {
      Console.WriteLine("\nDijak št. "+(i + 1)+". :");
      Console.Write("          ime: ");
      T.Dijaki[i].ime=Console.ReadLine();
      Console.Write("      priimek: ");
      T.Dijaki[i].priimek = Console.ReadLine();
      Console.Write("  dan rojstva: ");
      T.Dijaki[i].dan = Convert.ToInt32(Console.ReadLine());
      Console.Write("mesec rojstva: ");
      T.Dijaki[i].mesec=Convert.ToInt32(Console.ReadLine());
      Console.Write(" leto rojstva: ");
      T.Dijaki[i].leto = Convert.ToInt32(Console.ReadLine());
    }
    seznam(T,1990);//Funkcija, naj izpiše seznam dijakov rojenih po letu 1990
    //Izračunajmo skupno starost vseh dijakov glede na današnji datum
    DateTime d; //struktura DateTime je namenjena deli z datumi in časi
    d = DateTime.Now;  //Now je lastnost, ki v spremenljivko tipa DateTime vrne trenutni datum
    int starost = 0;
    for (int i = 0; i < N; i++)
       starost = starost + d.Year - T.Dijaki[i].leto;
    Console.WriteLine("\nSkupna starost vseh tečajnikov je "+starost/N +" let.");
}

//Funkcija, ki izpiše seznam dijakov rojenih po letu letnik
static void seznam(krozek T, int letnik) {
    Console.WriteLine("Seznam dijakov rojenih po letu " + letnik + ": \n");
    for (int i = 0; i < N; i++)
       if (T.Dijaki[i].leto > letnik)
          Console.WriteLine(T.Dijaki[i].ime + " " + T.Dijaki[i].priimek);
}
```

Vaja:

```
/*Kreiraj telefonski imenik: sestavlja ga tabela oseb. Oseba je struktura s podatki o določeni
osebi, ter z gnezdeno strukturo, ki predstavlja podatek o področni kodi ter telefonski
številki!*/

struct telefon
{
    public int  pkoda; //področna koda
    public long stevilka;
};

struct oseba
{
    public string ime;
    public telefon TEL; //gnezdena struktura
```

```
};

const int ST=2; //število oseb v imeniku

static void vnos(out oseba OS)
{
   Console.Write("\nIme: ");
   OS.ime=Console.ReadLine();
   Console.Write("TELEFON - Podrocna koda: ");
   OS.TEL.pkoda=Convert.ToInt32(Console.ReadLine());
   Console.Write("          Številka     : ");
   OS.TEL.stevilka=Convert.ToInt32(Console.ReadLine());
}

/*funkcija OBDELAJ ugotovi in izpiše koliko oseb je iz omrežne skupine OR*/
static void OBDELAJ(oseba [] IM,int OMR)
{
   int  skupaj=0;
   for (int i=0;i<ST;i++)
   {
       if (IM[i].TEL.pkoda==OMR) skupaj++;
   }
   Console.WriteLine("\nŠtevilo oseb iz omrežne skupine "+OMR+": "+skupaj);
}

static void Main(string[] args)
{
   oseba[] Imenik = new oseba[ST];
   for (int i = 0; i < ST; i++)   //Vnos podatkov v tabelo
   {
       vnos(out Imenik[i]); //funkcija ima za parameter strukturo (to je element tabele)
   }
   OBDELAJ(Imenik,4);//funkcija ugotovi, koliko oseb je iz omrežne skupine 4
}
```

Naloge:

- Kreiraj telefonski imenik: sestavlja ga tabela oseb (velikost tabele določi sam). Oseba je struktura s podatki o določeni osebi, ter z gnezdeno strukturo, ki predstavlja podatek o področni kodi (omrežni skupini) ter telefonski številki! Napiši funkcijo, ki izpiše seznam vseh oseb iz določene omrežne skupine!

- Kreiraj strukturo *datum*, ki naj vsebuje komponente *dan*, *mesec* in *leto*. Strukturo *datum* nato uporabi v strukturi *oseba*, ki naj vsebuje *ime*, *priimek*, *datum-rojstva*, *datum-prihoda*, *datum-odhoda* in *število-dni*. Nato deklariraj spremenljivko *o1* tipa *oseba*.

    - in napiši funkcijo za vnos podatkov v strukturo *oseba*;
    - napiši stavke za izračun razlike med datumom prihoda in datumom odhoda osebe o1.

Konstruktor

Omenili smo že, da so strukture spremenljivke vrednostnega tipa, kar pomeni, da ob deklaraciji nove spremenljivke tega tipa njene komponente še nimajo začetne vrednosti.

Primer:

```
struct tocka3D {
    public int x,y,z;
};

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

```
    Tocka3D A; //točka A še nima inicializiranih komponent/koordinat

    Console.WriteLine(A.x + "  " + A.y + "  " + A.z); /*Compile time error,
                                        ker komponente točke A še niso inicializirane!!!*/
}
```

Spremenljivka A je spremenljivka vrednostnega tipa, shranjena na skladu in ob deklaraciji še ni inicializirana. Za inicializacijo komponent seveda lahko poskrbimo sami, npr. takole:

```
tocka3D A; //deklaracija nove vrednostne spremenljivke A, ki je tipa tocka3D (struktura)
A.x = 2;   //inicializacija prve komponente točke A
A.y = 5;   //inicializacija druge komponente točke A
A.z = 4;    //inicializacija tretje komponente točke A

Console.WriteLine(A.x + "  " + A.y + "  " + A.z); //Izpis 2  5  4
```

Tak način je seveda čisto legalen, a zamuden če je takih spremenljivk veliko. Obstaja pa še možnost, da spremenljivko tipa struktura ustvarimo na kopici, seveda s pomočjo operatorja **new**. V tem primeru gre za referenčno spremenljivko, kar pa pomeni, da je taka spremenljivka ob deklaraciji že inicializirana:

```
tocka3D B = new tocka3D();//deklaracija nove REFERENČNE spremenljivke za strukturo tocka3D
/*POZOR: ker je B referenčna spremenljivka (ustvarili smo jo s pomočjo operatorja new), so
komponente točke B ŽE INICIALIZIRANE, imajo torej vrednost 0*/

Console.WriteLine(B.x + "  " + B.y + "  " + B.z); //Izpis 0  0  0
```

Vrednost, ki so jo dobile komponente spremenljivke B, kreirane s pomočjo operatorja **new**, so seveda enake 0 (privzeta začetna vrednost). V kolikor pa bi želeli imeti drugačne začetne vrednosti, lahko to storimo s pomočjo posebne metode znotraj strukture – ta metoda se imenuje **konstruktor**. Konstruktor je metoda znotraj strukture, ki ima enako ime kot struktura, njeni parametri pa so komponente te strukture (OBVEZNO morajo kot parametri nastopati VSE vrednostne komponente strukture). Struktura torej ne more vsebovati konstruktorja brez parametrov. (V poglavju razredi in objekti bomo spoznali bolj kompleksen podatkovni tip **class (razred)**, ki pa za razliko od strukture lahko vsebuje tudi konstruktorje brez parametrov!!!).

```
struct tocka3D
{
    public int x,y,z;

    //KONSTRUKTOR - poskrbi za inicializacijo komponent strukture
    public tocka3D(int koordX,int koordY, int koordZ)
    {
         x = koordX;
         y = koordY;
         z = koordZ;
    }
};

static void Main(string[] args)
{
    tocka3D B = new tocka3D(); //Privzeta deklaracija nove spremenljivke tipa tocka3D
    Console.WriteLine(B.x + "  " + B.y + "  " + B.z); //izpis 0  0  0

    tocka3D C = new tocka3D(3,6,9); //Ob deklaraciji točke C se izvede tudi konstruktor
    Console.WriteLine(C.x + "  " + C.y + "  " + C.z); //izpis 3  6  9
}
```

Iz primera je razvidno, da se konstruktor izvede le v primeru, ko pri kreiranju nove referenčne spremenljivke v oklepaju navedemo tudi vrednosti posameznih komponent. **OBVEZNO** pa moramo navesti želene vrednosti **vseh** komponent strukture – izjema so le tabelarične komponente strukture, ki jih v glavi konstruktorja ne navajamo.

Primer:

Kreirajmo novo referenčno spremenljivko A1 (točko) tipa **tocka3D**, ki bo že ob deklaraciji imela naključne koordinate med 0 in 10.

```
struct tocka3D
{
    public int x,y,z;
    //konstruktor, ki poskrbi za inicializacijo komponent strukture
    public tocka3D(int koordX,int koordY, int koordZ)
    {
         x = koordX;
         y = koordY;
         z = koordZ;
    }
};

static void Main(string[] args)
{
   Random naklj = new Random();
   //novi referenčni spremenljivki A1 posredujemo za paramere 3 naključna števila med 0 in 11
   tocka3D A1 = new tocka3D(naklj.Next(11), naklj.Next(11), naklj.Next(11));
   Console.WriteLine(A1.x + "  " + A1.y + "  " + A1.z); //izpis naključnih koordinat točke A1
}
```

Vaja:

```
/*Deklarirajmo strukturo, ki naj predstavlja kompleksno število. Struktura naj ima svoj
konstruktor. Napišimo še metodo za izpis kompleksnega števila, nato pa kreirajmo tabelo 10
kompleksnih števil in jo inicializirajmo z naključnimi vrednostmi obeh komponent.*/
struct Complex
{
    public Complex(float real, float imag)  //Konstruktor
    {
        realna = real;
        imaginarna = imag;
    }
    public float realna;
    public float imaginarna;
    public string ToString(Complex C)
    {
        return (C.realna + " + ( " + C.imaginarna + " * i )");
     }
};

public static void Main()
{
    Complex[] tabK = new Complex[10]; //tabela 10 kompleksnih števil
    Random naklj = new Random();
    for (int i = 0; i < tabK.Length; i++)
    {
       tabK[i] = new Complex(naklj.Next(-10, +10), naklj.Next(-10, +10));
       Console.WriteLine(i+". ti element "+ ToString(tabK[i]));
    }
    /*Napiši metodo za seštevanje dvek kompleksnih števil. Metoda ima za parametra
      kompleksni števili in tudi vrne kompleksno število.
      Napiši metodo za množenje dveh kompleksnih števil.  Metoda ima za parametra
      kompleksni števili in tudi vrne kompleksno število.
      Napiši metodo za izračun absolutne vrednosti kompleksnega števila.  Metoda ima za
      parameter kompleksno število in vrne absolutno vrednost.*/
}
```

Vaja:

```
/* Za vodenje evidence o padavinah v zadnjem letu potrebujemo naslednje podatke: ime kraja in
podatke o količini padavin v zadnjih  12 mesecih (12 števil v polju/tabeli). Kreiraj ustrezno
podatkovno strukturo (ime strukture naj bo padavine).
Napiši funkcijo za vnos podatkov o padavinah za vseh 12 mesecev.
Napiši funkcijo povp(kraj), ki vrne povprečno mesečno količino padavin ustreznega kraja!*/

//deklaracija strukture padavine
struct padavine
{
    public padavine(string ime) //konstruktor
```

```
    {
         ime kraja = ime;
         kolicina = new double[12]; //inicializacija tabele padavin
    }
    public string ime_kraja;
    public double[] kolicina; //deklaracija tabele padavin - inicializira jo konstruktur
};

static void vnos(padavine kraj) //metoda za vnos količine padavin kraja
{
    Console.WriteLine("\nVnos mesečne količine padavin za kraj: " + kraj.ime_kraja);
    for (int i = 0; i < 12; i++)
    {
        Console.Write("Količina padavin v " + (i + 1) + ". mesecu: ");
        kraj.kolicina[i] = Convert.ToDouble(Console.ReadLine());
    }
}

static double povp(padavine kraj) //metoda za izračun povprečne količ.padavin določenega kraja
{
    double vsota = 0;
    for (int i = 0; i < 12; i++)
       vsota += kraj.kolicina[i];
    return (Math.Round(vsota / 12, 2)); //rezultat zaokrožimo na dve decimalki
}

//glavni program
static void Main(string[] args)
{
    padavine kraj1=new padavine("Kranj"); //struktura kraj1 je ustvarjena na kopici (zaradi
                                          //operatorja new)
    vnos(kraj1); //klic metode za vnos padavin kraja kraj1

    padavine kraj2 = new padavine("Ljubljana");//pa še ena struktura ustvarjena na kopici
    vnos(kraj2);

    Console.WriteLine("Povprečna količina padavin v mestu "+kraj1.ime kraja+" je bila "
          +povp(kraj1));

    //pa še deklaracija spremenljivke vrednostnega tipa izpeljane iz strukture
    padavine kraj;
    Console.Write("Naziv kraja: ");
    kraj.ime kraja = Console.ReadLine();//preberemo/vnesemo ime kraja
    Console.WriteLine("\nVnos mesečne količine padavin za kraj: " + kraj.ime kraja);
    kraj.kolicina = new double[12]; //inicializacija tabele padavin za določen kraj
    for (int i = 0; i < 12; i++)
    {
        Console.Write("Količina padavin v " + (i + 1) + ". mesecu: ");
        kraj.kolicina[i] = Convert.ToDouble(Console.ReadLine());
    }
}
```

Naštevni tipi - enum

Naštevni tip (**enum**) je tip, v katerem je zbrana množica poimenovanih celoštevilskih konstant. Uporabljamo ga torej za imenovane vrednosti, ki so znane med prevajanjem. Temelji na celoštevilčnem osnovnem tipu.

Naštevanje (enumeracija) je bistvu alternativa konstantam. Naštevni tipi so zato vrednostni podatkovni tip, ki ga sestavlja množica konstant.

Recimo, da želimo deklarirati in inicializirati dve konstanti:

```
const int TockaZmrzovanja = 0;
const int TockaVrenja = 100;
```

Tema dvema konstantama bi radi kasneje dodali še tri, npr.:

```
const int TemperaturaZraka = 30;
const int TemperaturaVode = 25;
const int TemperaturaHladilnika = 4;
```

Tak način deklaracije in inicializacije konstant je sicer legalen, a med temi konstantami ni nobene logične povezave, čeprav vse predstavljajo podatke o temperaturah. S pomočjo naštevanje bi bila deklaracija naslednja:

```
enum Temperature
{
   TockaZmrzovanja = 0,
   TemperaturaHladilnika = 4,
   TemperaturaVode = 25,
   TemperaturaZraka = 30,
   TockaVrenja = 100,
}
```

Na posamezno konstanto se sklicujemo preko imena naštevnega tipa, operatorjem pika in imenom konstante, npr.;

```
Console.WriteLine("Voda zavre pri {0} stopinj Celzija!",Temperature.TockaVrenja);
//izpis: Voda zavre pri TockaVrenja stopinj Celzija!

Console.WriteLine("Voda zavre pri {0} stopinj Celzija!", (int)Temperature.TockaVrenja);
//izpis: Voda zavre pri 100 stopinj Celzija!
```

Privzeto so vrednosti konstant tipa **int**, lahko pa določimo tudi katerikoli drug števni tip: **byte**, **sbyte**, **short**, **ushort**, **uint**, **long** ali **ulong** (tipa **char** in **bool** nista dovoljena!). npr.:

```
enum Velikost:uint
{
   majhen = 1,
   srednji = 2,
   velik = 3,
}
```

Prva konstanta naštevnega tipa ima privzeto vrednost 0 (razen, če ne deklariramo druge vrednosti), vrednost vsake naslednje konstante pa je za eno večja.

Deklaracija in inicializacija naštevnega tipa

```
enum <ime> { vrednost1, vrednost2,  … };
```

Primer:

Deklarirajmo naštevni tip **Dan** s sedmimi konstantami, ki predstavljajo dneve v tednu:

```
enum Dan {Pon, Tor, Sre, Cet, Pet, Sob, Ned};
```

```
//Primer uporabe v glavnem programu
int x = (int)Dan.Ned;
int y = (int)Dan.Pet;
Console.WriteLine("Nedelja = " +x);
Console.WriteLine("Petek = "+ y);

Dan d=Dan.Sre;  //deklaracija in inicializacija nove spremenljivke d, ki je tipa Dan
```

Ker nismo določili drugače, ima konstanta **Pon** vrednost 0, konstanta **Tor** vrednost 1, itd.

Še en primer:

Deklarirajmo naštevni tip s katerim bomo deklarirali in inicializirali množico vseh ocen:

```
enum Ocene {Negativno = 1, Zadostno, Dobro, PravDobro, Odlično};
```

Prvi konstanti (**Negativno**) smo priredili vrednost, zato ima konstanta **Zadostno** vrednost 2, ostalim pa se vrednost povečuje za 1.

Vaja:

```
/*Deklarirajmo naštevni tip Dan, s konstantami, ki predstavljajo dneve v tednu.  */

enum Dan {Pon, Tor, Sre, Cet, Pet, Sob, Ned};

static string izpis(Dan d) //funkcija vrne polno ime dneva
{
    if (d == Dan.Pon) return "Ponedeljek";
    else if (d == Dan.Tor) return "Torek";
    else if (d == Dan.Sre) return "Sreda";
    else if (d == Dan.Cet) return "Četrtek";
    else if (d == Dan.Pet) return "Petek";
    else if (d == Dan.Sob) return "Sobota";
    else return "Nedelja";
}

static void Main(string[] args)
{
    Dan d;
    Console.WriteLine("Dnevi v tednu so: ");
    for (d = Dan.Pon; d <= Dan.Ned; d++) //operator ++ lahko uporabimo tudi za spremenljivko
                                         //naštevnega tipa
    Console.Write(izpis(d));
    //Elemente naštevnega tipa lahko izpišemo tudi takole
    for (d = Dan.Pon; d <= Dan.Ned; d++)
       Console.Write("Dan "+d+" ima vrednost "+(int)d+"\n");
}
```

Vaja:

```
/*Deklarirajmo naštevni tip Vrata s štirmi konstantami in vsem konstante hkrati tudi
inicializirajmo.*/

enum Vrata { Motor = 0, SportniAvto = 2, Limuzina = 4, HatcBack = 5 };

public static void Main()
{
    Vrata mojAvto = Vrata.SportniAvto;
    Vrata tvojAvto = Vrata.Motor;
    Vrata sosedovAvto = Vrata.Limuzina;

    Console.WriteLine("Ali ima "+ mojAvto+" več vrat kot "+ tvojAvto+("?");
```

```
    //Izpis: Ali ima SportniAvto več vrat kot Motor?
    if (mojAvto.CompareTo(tvojAvto) > 0)
       Console.WriteLine("Da\n");
    else Console.WriteLine("Ne\n");

    Console.WriteLine("Ali ima " + mojAvto + " več vrat kot " + sosedovAvto);
    //Izpis: Ali ima SportniAvto več vrat kot Limuzina?
    Console.WriteLine("{0}", mojAvto.CompareTo(sosedovAvto) > 0 ? "Da" : "Ne");
}
```

Vaja:

```
//Še en primer naštevnega tipa

enum Barve { Rdeča, Zelena, Modra, Rumena };

public static void Main()
{
    Barve mojaBarva = Barve.Modra;

    Console.WriteLine("Moja priljubljena barva je " + mojaBarva);
    Console.WriteLine("Vrednost moje priljubljene barve je "+ (int) mojaBarva);

    //ali

    Console.WriteLine("Vrednost moje priljubljene barve je " + Enum.Format(typeof(Barve),
                      mojaBarva, "d"));

    Console.WriteLine("Heksadecimalna vrednost moje priljubljene barve pa je "
                      +Enum.Format(typeof(Barve), mojaBarva, "x"));

    //metoda GetName vrne ime konstante določenega naštevnega tipa
    Console.WriteLine("Četrta vrednost barv v naštevnem tipu je "+ Enum.GetName(typeof(Barve),
                      3));  //izpis Rumena

    foreach (int i in Enum.GetValues(typeof(Barve)))
-        Console.Write(i+"  "); //Izpis: 0 1 2 3 4
    Console.WriteLine();
    Barve barva;
    for (barva=Barve.Rdeča;barva<=Barve.Rumena;barva++)
        Console.Write(barva + "  "); //Izpis: Rdeča Zelena Modra Rumena

    //ali pa

    foreach (string s in Enum.GetNames(typeof(Barve)))
        Console.Write(s); //Izpis: Rdeča Zelena Modra Rumena
}
```

Vaja:

```
/*Deklarirajmo naštevni tip ZaposleniTip z nekah elementi, nato pa strukturo Zaposleni, ki
vsebuje podatek o tipu zaposlenega, njegovem imenu in oddelku. Napišimo tudi ustrezen
konstruktor, nato pa prikažimo primer deklaracije spremenljivke vrednostnega tipa in
spremenljivke ustvarjene na kopici*/

enum ZaposleniTip : byte
{
   Menedžer,
   Razvijalec,
   Administrator,
   Programer
}

struct Zaposleni
{
   public ZaposleniTip naziv;
   public string ime;
   public short oddelek;

   public Zaposleni(ZaposleniTip ZTip,string Zime,short Zodd)//konstruktor
   {
      naziv = ZTip;
      ime = Zime;
```

```
      oddelek = Zodd;
   }
}

public static void Main(string[] args)
{
   Zaposleni fred;//fred je vrednostna spremenljivka na skladu (konstruktor se ni izvedel)
   fred.oddelek = 40;
   fred.ime = "Fred";
   fred.naziv = ZaposleniTip.Razvijalec;

   Zaposleni mary = new Zaposleni(ZaposleniTip.Programer, "Mary", 11);//mary je referenčna
                    //spremenljivka, ustvarjena na kopici
   Zaposleni tom = new Zaposleni();//izvede se privzeti konstruktor: numerična polja dobijo
                                   //vrednost 0, polja tipa string pa so prazna
}
```

Vaja:

```
/*Definiraj naštevni tip Izobrazba, ki ima elemente (osnovna, poklicna, srednja, višja,
visoka, magisterij, doktorat). Naštevni tip naj se prične s števko 3. Deklariraj še strukturo
za delavca z elementi: ime, priimek, končana šola. Deklariraj spremenljivko za tako strukturo,
vnesi podatke in jih nato izpiši.(Navodilo: za delavca vnesemo stopnjo izobrazbe s števko
izpišemo pa element naštevnega tipa). */

public enum Izobrazba {osnovna = 3, poklicna,srednja, višja, visoka, magisterij, doktorat}

public struct Tdelavec
{
    public string ime;
    public string priimek;
    public Izobrazba KoncanaSola;
    public Tdelavec(string ime, string priimek, Izobrazba KoncanaSola)   //konstruktor
    {
        this.ime = ime;
        this.priimek = priimek;
        this.KoncanaSola = KoncanaSola;
    }
}

static void Main(string[] args)
{
    Tdelavec delavec;
    Console.WriteLine("Podatki o delavcu");
    Console.Write("  Ime: ");
    delavec.ime = Console.ReadLine();

    Console.Write("\n  Priimek: ");
    delavec.priimek = Console.ReadLine();
    Console.Write("\n  Izobrazba(3,4,5,6,7,8,9): ");

    //eksplicitna konverzija iz int v VrstaIzobrazevanja
    delavec.KoncanaSola = (Izobrazba)Convert.ToInt32(Console.ReadLine());
    Console.WriteLine(delavec.ime);
    Console.WriteLine(delavec.priimek);
    Console.WriteLine("Izobrazba: "+delavec.KoncanaSola);
}
```

Naloge:

- Deklariraj naštevni tip **Glasnost** s tremi elementi (**Tiho**, **Srednje**, **Glasno**). Element **Tiho** naj ima privzeto vrednost 1. Napiši funkcijo za uporabnikov vnos nastavitve glasnosti. Napiši še funkcijo za poimenski in vrednostni izpis elementov naštevnega tipa **Glasnost**.

- Kreiraj strukturo *CD* z elementi naslov, izvajalec, zvrst, založba, letnica in cena *CD*-ja. Element zvrst je naštevni tip, katerega vrednosti določi sam (npr. pop, rock, klasika, …) Napiši funkcijo vnos, ki prebere podatke o *CD*-ju. Vpisane podatke izpiši s pomočjo funkcije izpis. Funkcija vnos ima za parameter referenco na strukturo (klic po referenci), funkcija izpis pa strukturo (prenos po vrednosti). Uporabnik naj vpiše podatke za dva *CD*-ja, ki ju potem izpiši po abecedi po naslovih.

- Deklariraj naštevni tip *vrstaRože* (vrtnica=1, lilija, dalija, nagelj, iris, mešano..), ter naštevni tip *barvaRože* (rdeča=1, bela, rumena, vojoličasta, oranžna, modra, zelena …), nato pa kreiraj strukturo *roža* s komponentami *tip* (*vrstaRože*), *barva* (*barvaRože*), komadi (*int*) ter *cena* (*decimal*);

    - napiši konstruktor, ki določi tip ter barvo rože, za začetno ceno pa določi 0;
    - napiši program, ki od uporabnika zahteva izbiro rože, ter barve;
    - glede na izbiro rože ter barve deklariraj ustrezen objekt *šopek* tipa *enostavenŠopek*, vnesi še število rož ter ustrezno ceno za komad, nato pa izpiši ceno šopka;
    - deklariraj še netipizirano zbirko *šopek*, v kateri boš lahko dodal poljubno število objektov tipa *enostavenŠopek*. Vsebino zbirke *šopek* na koncu izpiši in obenem izračunaj ter izpiši njegovo ceno!

Rekurzija

Naloge v praksi ponavadi zajemajo več kot en izračun, več pogojev, več zank. Zato jih smiselno razgradimo na podnaloge, rešimo vsako posebej in jih na koncu združimo skupaj. Podnaloge imenujemo podprogrami, rešimo jih z lastnimi metodami. Zaženemo pa jih v glavnem programu.
Zaenkrat imamo osnovno vedenje o metodah. Vse naše metode so statične, torej opremljene z besedico static. Vemo, kako takšne metode napišemo, kako jih kličemo …

Tokrat si bomo najprej ogledali močan prijem v programiranju – rekurzivne metode.

Rekurzivni klic metode je močno programersko orodje. Velikokrat se s pomočjo rekurzije algoritmi lažje in pregledneje izrazijo. O rekurzivnem klicu oz. rekurzivnih metodah govorimo takrat, ko **metoda kliče samo sebe**.

Kaj je to rekurzija

Vrsto let nazaj je bila zelo priljubljena pesmica, ki je zlasti prišla prav, če je učiteljica naročila, da se moraš naučiti poljubno pesmico, ki mora imeti vsaj 50 vrstic. Pesmica je šla takole

O možu in psu

Živel je mož, imel je psa, lepo ga je učil.
Nekoč ukradel mu je kos mesa, zato ga je ubil.
Postavil mu je spomenik in nanj napisal:
    Živel je mož, imel je psa, lepo ga je učil.
    Nekoč ukradel mu je kos mesa, zato ga je ubil.
    Postavil mu je spomenik in nanj napisal:
        Živel je mož, imel je psa, lepo ga je učil.
        Nekoč ukradel mu je kos mesa, zato ga je ubil.
        Postavil mu je spomenik in nanj napisal:
            Živel je mož, imel je psa, lepo ga je učil.
            Nekoč ukradel mu je kos mesa, zato ga je ubil.
            Postavil mu je spomenik in nanj napisal:
                Živel je mož, imel je psa, lepo ga je učil.
                Nekoč ukradel ...

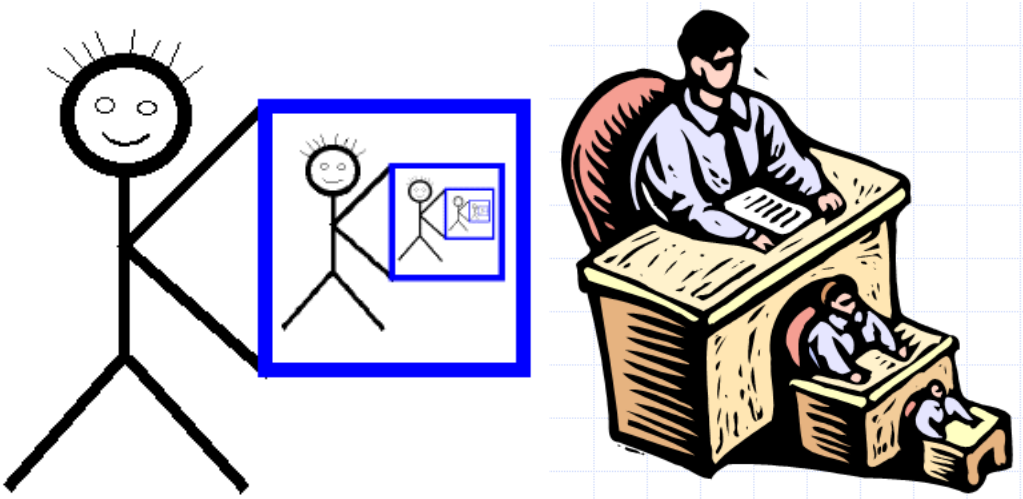
Tudi za dolge spise s predpisanim številom besed obstaja recept

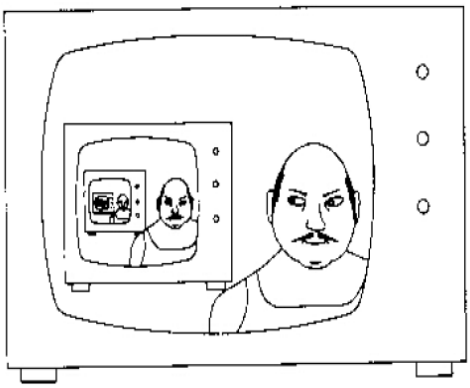
Kapitanova zgodba

Bila je temna, nevihtna noč ... Ladjo so valovi premetavali naprej in nazaj, veter je zavijal med jadri in dež se je zlival na palubo. Posadka je bila zbrana ob petrolejki. Vsi so se zavijali v odeje in trepetali, ko je kapitan pričel pripovedovati zgodbo:
"Bila je temna, nevihtna noč ... Ladjo so valovi premetavali …"

Kaj je skupnega tema dvema "literarnima deloma". Oba sta pravzaprav določena sama s sabo. Če bi ju želeli opisati na kratko bi reki:
Pesem o možu in psu govori o možu, ki je psu postavil spomenik in nanj napisal Pesem o možu. Kapitanova zgodba govori o kapitanu, ki svojim možem pripoveduje Kapitanovo zgodbo.
Tudi na likovnem področju hitro najdemo primere tovrstnih zgradb. Na sliki je le nekaj naključno izbranih primerkov iz različnih virov.

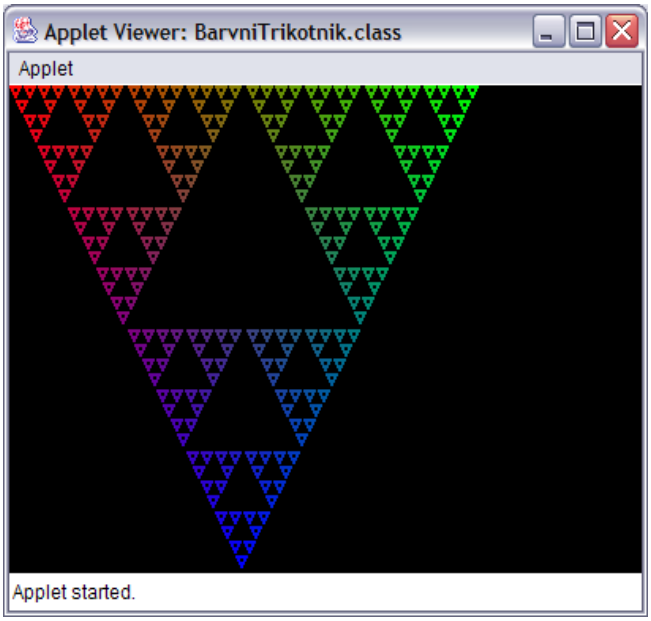
Slika v sliki



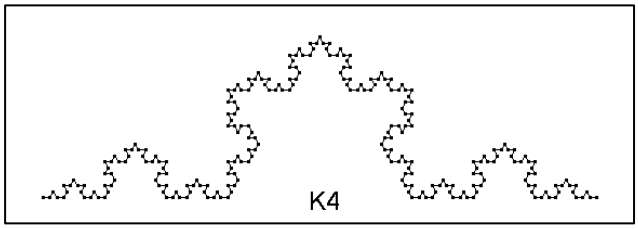


Problemi

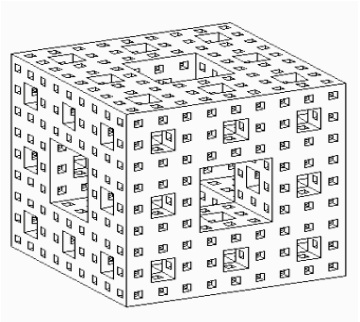
Oglejmo si nekaj različnih problemov. Prvi zahteva, da narišemo trikotnik Sierpinskega, katerega barvna slika je



Spet naslednji zahteva, da izračunamo dolžino tako imenovane Kochove črte stopnje 4 (ali več). Te Kochove črte pogosto uporabljajo pri modeliranju robov objektov v naravi (računalniške igrice, simulacije …)



Spet tretji problem (najdemo ga npr. v gradbeništvu, arhitekturi, pri konstrukciji različnih vpojnih materialov …) zahteva izračun volumna t.i. Mengerjeve spužve - kocke, preluknjane po določenem postopku n-krat



Naštejmo še nekaj problemov:

- Poišči največje in najmanjše število v tabeli števil
- Uredi podatke po velikosti.
- Izračunaj produkt naravnih števil od 1 do n.
- Izračunaj y^n s čim manj množenji.

Če je naša naloga sestaviti navodila (postopek), s katerim bi problem rešili, imajo vsi ti različni problemi, navkljub različnosti skupni prijem, ki mu rečemo **rekurzija**. Kaj pa ta beseda sploh pomeni? V Slovarju slovenskega knjižnega jezika SSKJ ni besede rekurzija, pojavlja pa se beseda rekurz -a m (u) knjiž. vrnitev (na kako stvar, dejstvo): rekurz na že omenjana dognanja ni potreben / v njegovih romanih so pogosti rekurzi v preteklost (tudi v filmih je tega precej) in rekurirati -am dov. in nedov. (i) knjiž. vrniti se (na kako stvar, dejstvo): pogosto rekurirati na nekatera dejstva, spoznanja. V Velikem slovarju tujk je dana definicija rekurzije (iz latinščine recurrere iti nazaj, vrniti se):

- definiranje funkcije ali postopka s samim seboj (informatika)
- izvajanje veličine ali funkcije, ki jo je treba šele definirati na že znano veličino (matematika). Označuje tudi zaporedje, katerega n-ti člen je določen z enim ali več predhodnimi členi.

Pri rekurziji gre torej za to, da je določena stvar (tudi postopek) določena tako, da v opisu nastopa spet ta ista stvar.

Tudi v vsakdanjem življenju srečamo rekurzijo. Definicija prednika neke osebe je lahko:

- prednik osebe je eden od roditeljev osebe (osnovni primer)
- prednik pa je tudi roditelj kateregakoli prednika (rekurzivni primer)

Če pogledamo na slovensko wikipedijo, vidimo, da rekurzijo razložijo kar s sliko, ki v bistvu prikazuje samo sebe:

Rekurzivna slika, na kateri je rekurzivna slika, na kateri je rekurzivna slika, na kateri ...

Ko govorimo o rekurziji pri programiranju, mislimo na to, da pri sestavljanju algoritma (metode) za reševanje nekega problema v rešitvi uporabimo klic (enega ali več) te metode. Rešitev problema je torej podana s samim problemom, le nad manjšim ali pa drugačnim obsegom podatkov.
V opisu postopka rešitve torej uporabimo kar ta postopek. Če želimo priti do rešitve, seveda ne moremo nadaljevati v nedogled kot npr. pri pesmici, saj se potem naš postopek ne bi nikoli končal. Zato je pri rekurzivnih algoritmih zelo pomembno določiti ustavitveni pogoj. Ta pove, daj v postopku ne uporabimo istega postopka (ne pokličemo te metode). Običajno je to takrat, ko so podatki (parametri metode) taki, da je problem "majhen" (enostaven).

Bistvo rekurzije je, da problem razdelimo na več podproblemov enake narave oz. da rešitev problema podamo s samim istim problemom, le na manjši količini podatkov. Vsaka rekurzija je sestavljena iz rekurzivnega dela, kjer funkcija kliče samo sebe in ustavitvenega pogoja. Slednji določi, kdaj pri reševanju rekurzivnega problema ne bomo uporabili rekurzije oz. kdaj je problem tako majhen, da se delitev na manjše podprobleme ne izplača več.

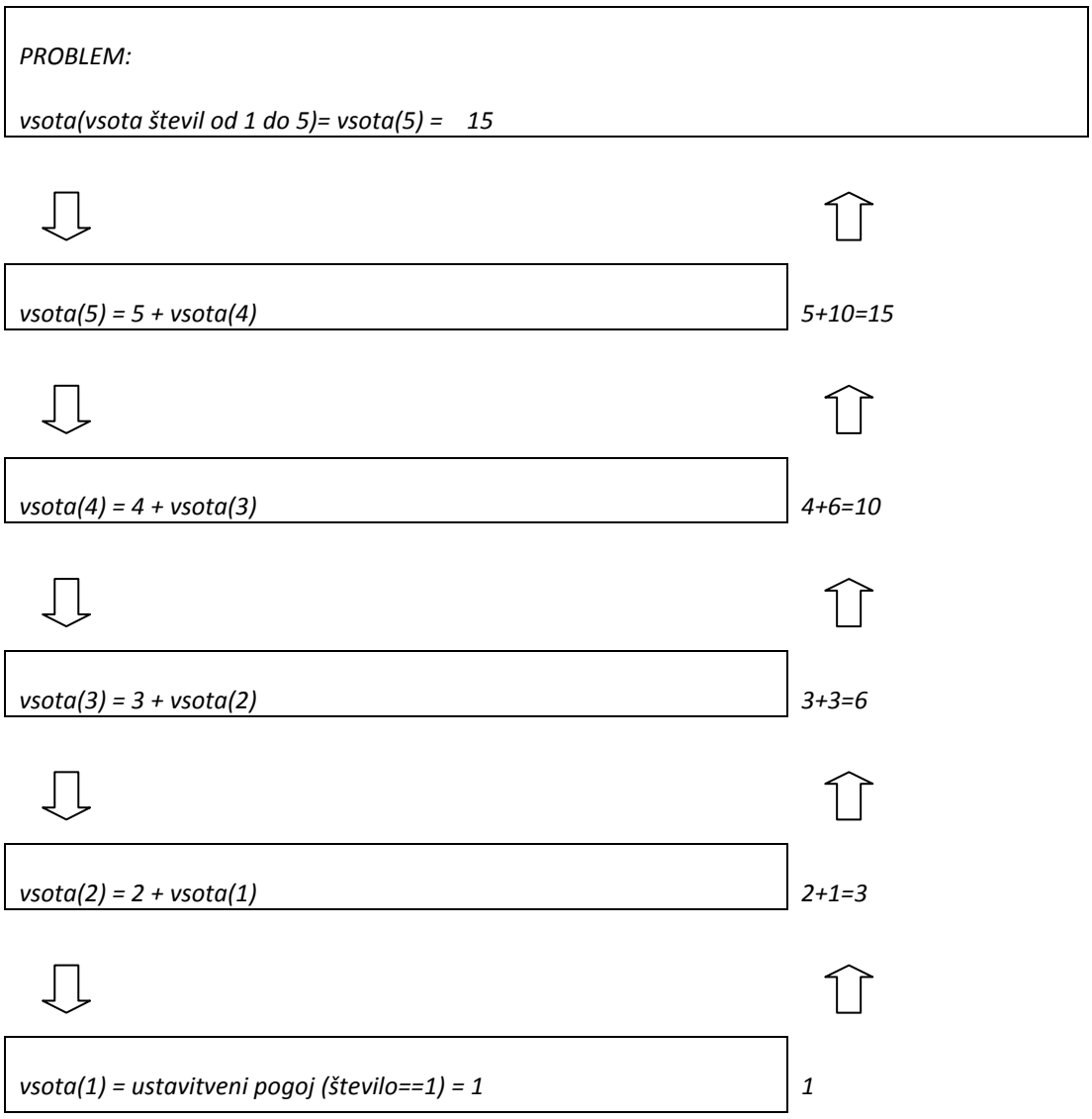
Rekurzivna metoda

```
rekurzivna_metoda (problem)
{
      if (problem majhen)
            return resitev;
      else
        razdeli problem na enega ali več manjših podproblemov enake vrste
        za vse podprobleme
      rekurzivna_metoda (manjši problem);
}
```

Zgledi

Vsota števil

Denimo, da želimo izračunati vsoto celih števil do n, kjer je n poljubno celo število. Recimo, da je n = 5. Potem moramo izračunati vsoto od 1 do 5. Sicer problem že znamo rešiti s pomočjo zanke, a tokrat se problema lotimo z rekurzijo. Vsoto števil do 5 namreč lahko izračunamo tudi tako, da k 5 prištejemo vsoto števil od 1 do 4. Slednje je spet problem iste vrste kot prvotni problem.



Problem *vsota(5)* smo razdelili na dva podproblema. Prvi sploh ni problem in vemo, da je njegova rešitev vrednost *5.* Drugi je *vsota (4).* Ker ne vemo koliko je vsota števil od 1 do 4, ponovno uporabimo rekurzijo in problem razdelimo na dva podproblema, *4* in *vsota(3)*, … Postopek izvajamo, dokler ne naletimo na *ustavitveni pogoj*, ki poskrbi, da se ustavimo. Sedaj sledi postopno vračanje. *Ustavitveni pogoj* nam vrne vrednost *1*, tej vrednosti prištejemo *2,* … Ko se vrnemo nazaj k prvotnemu problemu, dobimo rešitev našega problema, ki je *15*.

Če naše razmišljanje zapišemo v obliki algoritma, vidimo, da smo rekli sledeče

$$\sum_{i=1}^{n} i = n + \sum_{i=1}^{n-1} i$$

Ali če uporabimo zapis v obliki metode: `vsota(n) = n + vsota(n-1);`

Obema zapisoma manjka še ustavitveni pogoj. Dodajmo še tega

$$\sum_{i=1}^{1} i = 1$$

Oziroma `vsota(1) = 1;`

Zapišimo ustrezno metodo

```
public static int Vsota(int stevilo)
{
  if(stevilo == 1)
  {
    return 1;
  } // if
  else
  {
    int rezultat = stevilo + vsota(stevilo - 1); // rekurzivni klic
    return rezultat;
  } // else
} // vsota
```

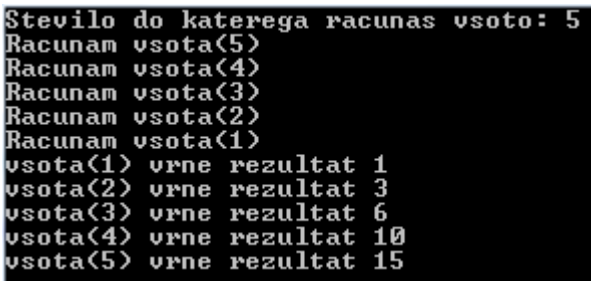
Vidimo, da v metodi sami spet kličemo to metodo. Zaradi tega klica je metoda rekurzivna, saj je za opis postopka spet uporabljena metoda sama.

Zapišimo vse skupaj še v obliki programa. Tokrat našo metodo dopolnimo s klici izpisa, ki povedo, kako kličemo metodo in kakšen rezultat vrača.

```
public static void Main(string[] args)
{
    Console.Write("Število do katerega računaš vsoto: ");
    int stevilo = int.Parse(System.Console.ReadLine());

    vsota(stevilo);
    Console.ReadKey();
} // main

public static int vsota(int stevilo)
{
    System.Console.WriteLine("Računam vsota(" + stevilo + ")");
    if (stevilo == 1) // ustavitveni pogoj
    {
        System.Console.WriteLine("vsota(" + stevilo + ") vrne " +
                                 "rezultat 1");
        return 1;
    } // if
    else
    {
        int rezultat = stevilo + vsota(stevilo - 1); //rekurziven klic
        System.Console.WriteLine("vsota(" + stevilo + ") vrne " +
                                 "rezultat " + rezultat);

        return rezultat;
    } // else
```

```
} // vsota
```

Program prevedemo in poženemo:



Opis programa.

Oglejmo si metodo *Vsota*. Na začetku se nahaja ustavitveni pogoj. Če ta ni izpolnjen, sledi rekurzivni klic. Seveda je izpis znotraj programa namenjen le programerju za lažje razumevanje delovanja programa in v sam program ne spada (metodo bi zares napisali tako, kot smo navedli prej), nam pa potrdi, da je zgornja razlaga metode *Vsota* prava.

V primeru, če kot parameter pri klicu uporabimo negativno število, se program zacikla. Premislimo zakaj! No, če kot podatek vnesemo *-5* , vidimo da začne z izpisovanjem

```
Racunam vsota(-5)
Racunam vsota(-6)
Racunam vsota(-7)
Racunam vsota(-8)
Racunam vsota(-9)
Racunam vsota(-10)
Racunam vsota(-11)
Racunam vsota(-12)
...
```

Popravimo program tako, da bo delal tudi za negativna števila. Dogovoriti se moramo, kaj pomeni vsota števil do negativnega števila n. Recimo, da je n -4. Želimo, da vsota(-4) pomeni -4 + -3 + -2 + -1. Vemo sicer, da bi zadevo lahko enostavno rešili takole

$$Vsota(-n) = -Vsota(n)$$

A da se bomo bolj utrdili v spoznavanju rekurzije, bomo sestavili dve rekurzivni metodi. Prva, imenovana *PozitivnaVsota*, bo obstoječa metoda *Vsota* z drugim imenom, druga, *NegativnaVsota*, pa metoda, ki z rekurzijo izračuna vsoto negativnih števil, torej *vsota(-4) = -4 + vsota(-3)*.

```
public static void Main(string[] args)
{
    System.Console.Write("Število do katerega računaš vsoto: ");
    int stevilo = int.Parse(System.Console.ReadLine());
    if (stevilo >= 0)
    {
        PozitivnaVsota(stevilo);
    } // if
    else
    {
        NegativnaVsota(stevilo);
    } // else
} // main

public static int PozitivnaVsota(int stevilo)
```

```
{
    System.Console.WriteLine("Računam vsota(" + stevilo + ")");
    if (stevilo == 0)
    {
        System.Console.WriteLine("vsota(" + stevilo + ") vrne " +
                        "rezultat 1");
        return 0;
    } // if
    else
    {
        int rezultat = stevilo + PozitivnaVsota(stevilo - 1);
        System.Console.WriteLine("vsota(" + stevilo + ") vrne " +
                        "rezultat " + rezultat);

        return rezultat;
    } // else
} // pozitivnaVsota

public static int NegativnaVsota(int stevilo)
{
    System.Console.WriteLine("Računam vsota(" + stevilo + ")");
    if (stevilo == -1)
    {
        System.Console.WriteLine("vsota(" + stevilo + ") vrne " +
                            "rezultat -1");
        return -1;
    } // if
    else
    {
        int rezultat = stevilo + NegativnaVsota(stevilo + 1);
        System.Console.WriteLine("vsota(" + stevilo + ") vrne " +
                            "rezultat " + rezultat);

        return rezultat;
    } // else
} // negativnaVsota
} // VsotaStevil3
```

Opis programa.
Program je sestavljen iz treh metod, metode *Main* in dveh rekurzivnih metod *PozitivnaVsota* in *NegativnaVsota*. V metodi *Main* preverimo, če je dano število pozitivno in če je, kličemo metodo *PozitivnaVsota*. Če pa je število negativno, kličemo metodo *negativnaVsota*. Ker smo si metodo *PozitivnaVsota* že podrobneje pogledali v prejšnjem zgledu, se bomo osredotočili na metodo *NegativnaVsota*. Kot vsaka rekurzivna metoda se tudi ta začne z ustavitvenim pogojem (ko pridemo do vrednosti *-1,* končamo z rekurzijo) in rekurzivnim delom. Recimo, da je prebrano število *-7*. Metoda *Main* kliče metodo *NegativnaVsota(-7)*. Ker ustavitveni pogoj ni izpolnjen, nadaljujemo z rekurzijo:

–7 + NegativnaVsota(-6);
–7 + (–6) + NegativnaVsota (-5)
–7 – 6 + (–5) + NegativnaVsota (-4)
–7 – 6 – 5 + (– 4) + NegativnaVsota (-3)
–7 – 6 – 5 – 4 + (– 3) + NegativnaVsota (-2)
–7 – 6 – 5 – 4 – 3 + (– 2) + ustavitveni pogoj
–7 – 6 – 5 – 4 – 3 – 2 – 1
–7 – 6 – 5 – 4 – 3 – 3
–7 – 6 – 5 – 4 – 6
–7 – 6 – 5 – 10

–7 – 6 – 15
–7 – 21
–28.

Faktoriela / fakulteta / n!

V matematiki, še posebej v kombinatoriki, pogosto naletimo na pojem faktoriele ali kot tudi rečemo fakultete nekega naravnega števila. Označimo jo s !. Faktoriela števila 7 je torej 7!. Faktoriela je zelo hitro naraščajoča zadeva.

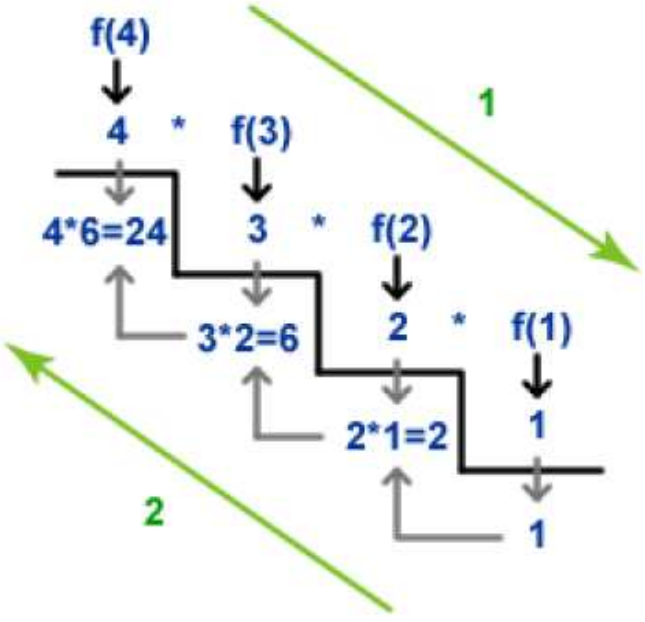
- 3! = 6
- 5! = 120
- 42! = 1405006117752879898543142606244511569936384000000000

V matematiki jo običajno definiramo z rekurzivno definicijo:
n! = n * (n-1)!

N! bomo torej izračunali, če bomo poznali (n-1)!

- 3! = 3 * 2!
  - 2! = 2 * 1! =
    - 1! = 1 * 0! =
      - 0! = 0 * (-1)! = ???

Omenjena rekurzivna definicija torej ni popolna. Manjka ji ustavitveni pogoj! Ta je določen z
0! = 1
Ko nas torej zanima fakulteta števila in je to število 0, ne uporabimo zgoraj omenjene rekurzivne definicije, ampak takoj odgovorimo, da je rezultat 1. Ker torej poznamo 0!, lahko izračunamo 1! (ki je 1). Če nas zanima fakulteta le pozitivnih (naravnih) celih števil, včasih kot ustavitveni pogoj določimo kar podatek 1 in torej za 1! kar neposredno odgovorimo, da je to 1. Če torej n! zapišemo kot f(4), se f(4) izračuna tako, kot kaže naslednja slika.



Postopek računanja 4! torej zahteva izračun 3!. Ta spet zahteva, da poznamo 2!, ki pa spet zahteva poznavanje 1!. A koliko je 1! vemo – to je 1. Zato lahko izračunamo 2! (2 * 1 = 2). Ker pa poznamo 2!, lahko izračunamo 3! (3 * 2 = 6). In ker poznamo 3!, ni ovir, da ne bi izračunali še 4!. To je 4 * 6, torej 24.

Faktoriela – postopek
Če stvar napišemo bolj računalniško, v obliki "kvazi metode". Kot ustavitveni pogoj uporabimo kar podatek 0. 1! se v tem primeru izračuna z enim klicem v globino, kot tudi rečemo, torej kot 1 * 0!

Fak(n):
  Če je n = 0, je rezultat 1
  sicer pa
    rezultat = n * fak(n – 1)

Zapišimo to še v obliki metode v jeziku C#.

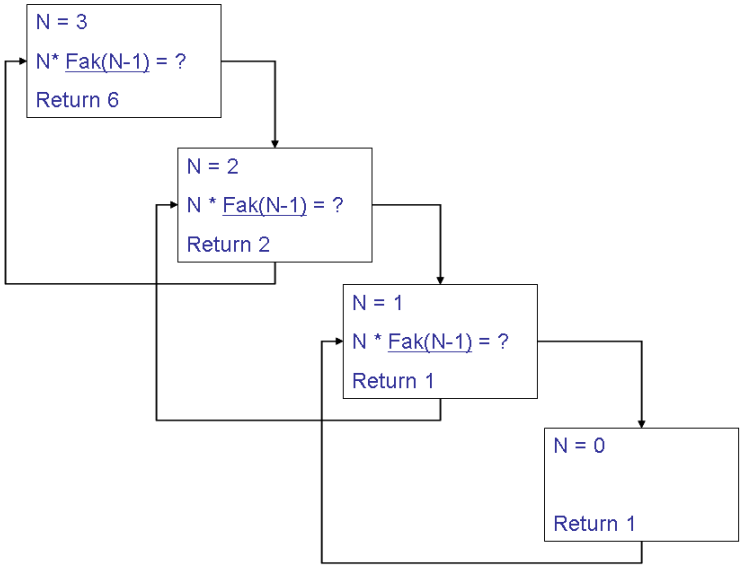
```
public static int fak(int stevilo)
{
    if (stevilo == 0)
    {
        return 1;
    }
    else
    {
        return stevilo * fak(stevilo - 1);
    }
}
```

Še glavni program in klic rekurzivne metode *fak*

```
public static void Main(string[] args)
{
    Console.Write("Vnesti število, katerega fakulteta te zanima:");
    int n = int.Parse(Console.ReadLine());
    Console.WriteLine(n + "! = " + fak(n));
}
```

V primeru, da je vneseno število *n* enako 3 dobimo na ekranu rezultat:

Poglejmo, kaj se dogaja ob klicu Fak(3)



Tisto, kar nam pri razumevanju rekurzije največkrat dela težave je, kje se dogaja vse to "knjigovodstvo" – zakaj je enkrat n 3, pa se potem spremeni v 2 in potem v 1 … Zavedati se je potrebno, da pri vsakem klicu metode sploh ni pomembno, da gre za rekurzivni klic. Ob vsakem klicu začnemo s "svežimi" vrednostmi (svežimi škatlicami kot so na zgornji sliki) in spremenljivke se med seboj "ne motijo". Ko klic metode opravi svoje delo (zve, koliko je rezultat), se vrnemo nazaj na mesto, kjer smo metodo klicali (v "škatlico" na sliki), kjer sedaj seveda veljajo tiste spremenljivke, ki so v tisti škatlici. Za vse te kopije spremenljivk, škatlice, mesta, kamor se mora metoda, ko konča delo vrniti, skrbi prevajalnik in izvajalno okolje.
Vračanje nazaj iz klicev v globino torej poteka avtomatsko, brez našega "vmešavanja":

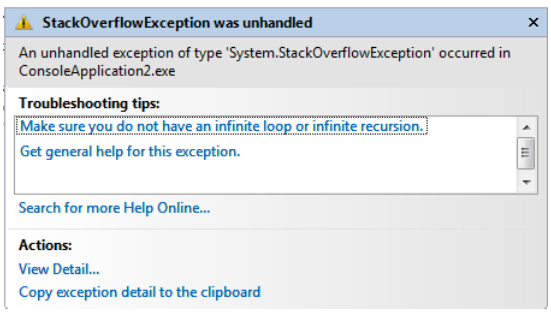
Opozorilo:

> Posebno pozornost namenimo argumentom v rekurzivnem klicu! Če namreč ob klicu argumenta ne spremenimo ustrezno, zlahka dobimo neskončno rekurzijo. Klici v globino se nikoli ne nehajo. Rekurzivni klic mora imeti izstopni pogoj oziroma ustavitveni pogoj.

Če npr. zgornjo metodo napišemo kot

```
public static int fak(int stevilo)
{
   return stevilo * fak(stevilo - 1);
}
```

nas kaj hitro pozdravi opozorilno okno o napaki:



Sistem javlja nekakšen **StackOverflowException**. Gre enostavno za to, da je izvajalnemu okolju zmanjkalo prostora za vse tisto knjigovodstvo, ki ga mora ob klicih metod opravljati. Ker se klic nobene metode ni zaključil (saj je vsaka čakala na rezultat metode, ki jo je ona poklicala) , je izvajalno okolje moralo skbeti za vse metode in enkrat je pač zmanjkalo prostora. Če naletimo na takšno napako, smo po vsej verjetnosti narobe napisali ustavitveni pogoj!

Pri računanju zlahka prekoračimo obseg tipa int. Če namreč poskusimo v zgornjem programu kot podatek vnesti npr. 32, dobimo v primeru izvajanja v okolju Visual Studio rezultat negativen – prekoračili smo namreč obseg celih števil.

Kako pa potem izračunati npr. 42! ? Seveda tudi to gre. Le na v C# vgrajene številske tipe se ne moremo več zanašati. Sestaviti bi bilo potrebno npr. metodo, ki bi rezultat vračala kot niz. Seveda pa bi bilo potem potrebno napisati še metodo, ki bi znala dolgo število (npr. z 80 števkami) predstavljenega kot niz pomnožiti s celim številom. Potem bi napisali nekaj takega:

```
1:      public static string Fak(int stevilo)   {
2:         if (stevilo == 0) {
3:            return "1";
4:         } else {
5:            return PomnoziSteviloInNiz(stevilo, Fak(stevilo - 1));
6:         }
7:      }
```

Metodo PomnoziSteviloInNiz pa napišite za vajo kar sami!

Če nekoliko premislimo in si ogledamo postopek računanja, vidimo, da faktorielo števila lahko izračunamo tudi drugače. Faktoriela je namreč produkt vseh števil od 1 do tistega števila. 7! je torej
7! = 1 × 2 × 3 × 4 × 5 × 6 × 7 = 5040
Zato tu rekurzivni pristop lahko zlahka nadomestimo z iterativnim oz. z zanko

```
public static int fakulteta(int st)
{
     int rezultat = 1;
     for (int j = st; j > 1; j--)
     {
         rezultat = rezultat * j;
     }
     return rezultat;
}
```

Vedno pa ni mogoče tako zlahka rekurzivnega postopka spremeniti v iterativnega. Pogosto je tudi zapis v rekurzivni obliki krajši in bolj pregleden.

Fibonaccijevo zaporedje

Dano je Fibonaccijevo zaporedje : 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ... Vsak člen tega zaporedja (razen prvih dveh) je vsota predhodnih dveh členov. Označimo splošni, n-ti člen tega zaporedja z F(n). Pravilo o tem, kako dobimo n -ti člen , lahko izrazimo rekurzivno takole :

$F(n) = F(n-1) + F(n-2)$ n= 3, 4, 5 ....

Za prva dva člena zaporedja pa predpišemo še poseben pogoj :

$F(1) = 1$, $F(2) = 1$

Formulacija je rekurzivna, ker se v enačbi za splošni člen F(n) sklicujemo tudi na člena F(n-1) in F(n-2). Rekurzivno definicijsko enačbo lahko uporabimo tudi direktno za izračun določenega člena zaporedja, npr. za n=5. To storimo takole :

F(5) = F(4) + F(3)

Sedaj uporabimo pravilo za izračun F(4) in F(3) itn. ;

F(5)= F(4) + F(3) = (F(3) + F(2)) + (F(2) + F(1)) =
= ((F(2) + F(1)) + 1) + (1 + 1) =
= ((1 + 1) + 1) + (1 + 1) = 5

Napišimo sedaj rekurzivno metodo int `fib(int n)`, ki za dani n izračuna ustrezno Fibonaccijevo število . Zgornji rekurzivni princip lahko neposredno uporabimo takole :

če je n enako 1 ali 2, potem je fib(n) = 1,
sicer pa je fib(n) enako fib(n-1) + fib(n-2)

Zapis v C# :

```
static int fib(int n)
{
    if ((n == 1) || (n == 2)) return 1;
    else return (fib(n - 1) + fib(n - 2));
}

//Primer klica te metode v glavnem programu
Console.WriteLine(Fib(33));  //Klic rekurzivne metode
```

Rekurzivna definicija se v gornjem podprogramu zrcali v tem, da podprogram kliče samega sebe (in to dvakrat) v stavku `return (fib(n - 1) + fib(n - 2));`

Takole pa bi napisali metodo za Fibonaccijeva števila brez rekurzije :

```
static int Fib1(int n)
{
    if (n == 1) return 1;
    if (n == 2) return 1;
    int F1 = 1, F2 = 1, P;
    for (int i = 0; i < n-2; i++)
    {
        P = F2;
        F2 = F1 + F2;
        F1 = P;
    }
    return F2;
}

//Primer klica te metode v glavnem programu
Console.WriteLine(Fib1(33)); //Klic iterativne funkcije
```

Za tako rešitev pravimo, da je formulirana iterativno, za razliko od rekurzivne.

Rekurzivna rešitev je krajša, enostavnejša in jasnejša, saj se v njej neposredno zrcali originalna matematična definicija Fibonaccijevega zaporedja in je zato tudi lažje razumljiva. Rekurzivna rešitev poleg tega tudi ne uporablja nobenih dodatnih spremenljivk. Glede varčnosti pomnilniškega prostora pa se izkaže, da je pri rekurzivni rešitvi le navidezna. Zahteve po rekurzivnih klicih se pri rekurzivnem podprogramu skrivajo v rekurzivnih klicih. Vsak rekurzivni klic namreč zahteva nekaj prostora, zapomniti si je potrebno tudi mesto, kam se je potrebno vrniti ob povratku iz podprograma. Za vsak rekurzivni klic pa si je potrebno zapomniti tudi vmesne rezultate. V resnici si vsak rekurzivni klic zgradi svoj povratni naslov in svojo verzijo vmesnih rezultatov

oz. lokalnih spremenljivk. Ti podatki se nalagajo v pomnilniški sklad (stack). Sklad je enostavno shranjen v pomnilniku in zanj poskrbi sam prevajalnik, tako da programerju o njem ni potrebno razmišljati. Sklad tako raste in pada po potrebi. Izkaže se, da tak sklad navadno zahteva več prostora kot iterativni podprogrami, rekurzivni podprogrami pa so zato nekoliko potratnejši od iterativnih. Podobna je situacija glede hitrosti izvajanja. Klic podprograma traja nekoliko dlje kot enostavne operacije v iterativnih zankah. V našem zgledu je rekurzivni podprogram še posebno počasen, saj zahteva več seštevanj kot iterativni.

Rekurzivne formulacije imajo torej prednost pred iterativnimi v tem, da je programiranje bolj enostavno, jasno in razumljivo. Nekoliko manj učinkovit pa je rekurzivni podprogram glede časa izvajanja in prostora v pomnilniku. Od konkretnega primera pa je odvisno, za katero pot se bomo odločili.

Izračun potence števila

Denimo, da želimo izračunati y^n, kjer je y poljubno pozitivno decimalno število, n pa neko naravno število, npr. 16.

y16 = y* y* y *y *y* y *y* y* y *y *y* y *y* y *y *y

Zlahka napišemo metodo, v njej uporabimo zanko in problem je rešen:

```
public static double potenca(double y, int n)
{
    double rezultat = 1;
    int i = 1;
    while (i <= n)
    { //n krat pomnožimo z y
      rezultat = rezultat * y;
      i = i + 1;
    }
    return rezultat;
}
```

A če to metodo uporabljamo zelo velikokrat, nas malo moti, da je potrebno kar 16 množenj. Še posebej, ker nam premislek pokaže, da do rezultata lahko pridemo le s štirimim množenji! Kako pa? Poglejmo:

- y16 = y8 y8

Če torej poznamo y8, lahko do y16 pridemo le z enim množenjem. In ko računamo y8, spet lahko uporabimo isti postopek:

- y8 = y4 y4
- y4 = y2 y2
- y2 = y y

Prihranek je torej precejšen. Še večji je pri večjih potencah. Npr. klasično je za izračun y1024 potrebnih 1024 množenj, z našim postopkom pa le 10. 100x hitreje torej.
A ta postopek gre tako lepo le, če je eksponent potenca števila 2 (torej 2, 4, 8, 16, 32, …). Kaj pa, če n npr.14. Če malo pomislimo, bo šlo z našim postopkom vedno, ko bo eksponent sodo število. Če pa je eksponent lih, naredimo le en vmesen korak:

- y14 = y7 y7
- y7 = y y6
- y6 = y3 y3
- y3 = y y2
- y2 = y y ..... 5 množenj

Ko torej računamo Pot(y, n) in je n sod, izračunamo Pot(y, n/2) in ga shranimo v pomožno spremenljivko, denimo pom. Rezultat metode pa je potem pom * pom.
Če pa je n lih, je rezultat te metode kar y * Pot(y, n – 1)

```
public static double pot(double y, int n)
 {
    if (n % 2 == 0)
    {
       double pom = pot(y, n / 2);
       return pom * pom;
    }
    return y * pot(y, n-1); // lih eksponent
}
```

Metoda pa ni v redu! Zakaj ne? Seveda, pozabili smo na ustavitveni pogoj. Razmislek pokaže, da je smiselno, da se ustavimo, ko je eksponent enak 0. Takrat vemo rezultat – 1 je!

```
public static double pot(double y, int n)
{
    if (n == 0) return 1;
    if (n % 2 == 0)
    {
       double pom = pot(y, n / 2);
       return pom * pom;
    }
    return y * pot(y, n-1); // lih eksponent
}
```

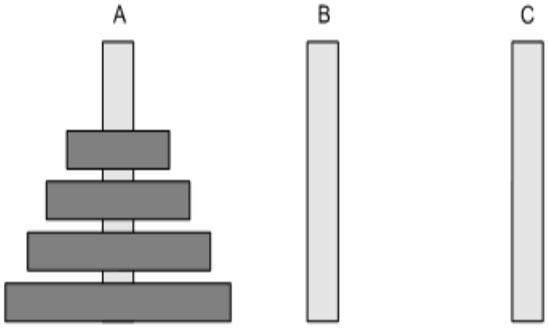
Seveda bi tudi to metodo (z manj množenji) napisali iterativno, a malo bolj bi se že namučili.

Hanoiski stolpiči

Preden se lotimo še enega primera, ki bo res pokazal moč rekurzije, premislimo.
Kaj je torej rekurzija? Kako naj bo v slovarju definirana beseda 'rekurzivno' ? Enostavno:

Piše naj: Glej 'rekurzivno'.

Zakaj gre pri problemu Hanoiskih stolpov?
Obstaja legenda, ki pravi, da so v hindujskem templju tri palice. Na začetku je bil na prvi sklad iz 64 zlatih obročev. Vsi so bili različne velikosti in zloženi po velikosti, tako da so bili manjši obroči na večjih v obliki piramide.

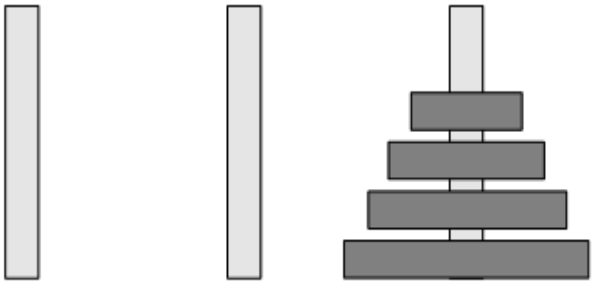


Brahmani premikajo te diske v skladu z pradavnimi pravili. Njihov cilj je vse diske prenesti na zadnjo palico, pri čemer morajo upoštevati pravila:

- vedno lahko premaknejo le po en disk naenkrat (to imenujemo poteza)

- na katerikoli palici morajo biti diski vedno razporejeni od največjega (na dnu) do najmanjšega (na vrhu).

Brahmani marljivo delajo noč in dan. Takoj, ko bi bila opravljena zadnja poteza



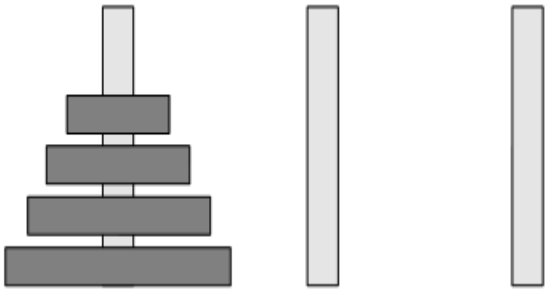
, naj bi se po starodavni prerokbi tempelj sesul v prah in svet naj bi izginil. Da ne boste preveč zaskrbljeni nad svojo usodo in usodo svojih potomcev, naj vas potolažimo: proces bo trajal zagotovo dlje, kot je do sedaj znana starost vesolja (za nekaj velikostnih razredov).
Zanima nas, kako morajo svečeniki prestavljati obroče, da bodo naredili kar se da malo potez (ds se končno znebimo te zasvinjane Zemlje in začnemo na novo). Problem Hanoiskih stolpičev je torej ta, da izpiše navodila, kako prestavljati obroče, če je na začetku na prvi palici n zlatih obročev (no, tudi če so železni ali virtualni, se problem ne spremeni). Npr. če bi našo metodo poimenovali Hanoi, naj bi klic
Hanoi(2, "A", "B", "C")
izpisal navodila:

- Preloži obroč z A na B
- Preloži obroč z A na C
- Preloži obroč z B na C

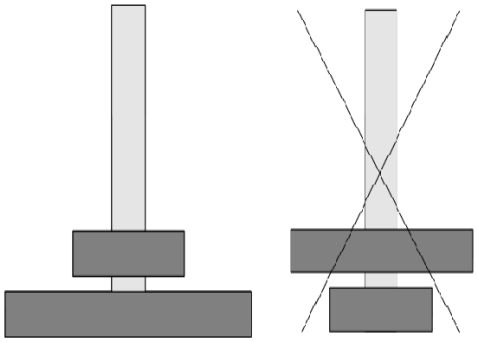
klic Hanoi(3, "P", "D", "T") pa

- Preloži obroč z P na T
- Preloži obroč z P na D
- Preloži obroč z T na D
- Preloži obroč z P na T
- Preloži obroč z D na P
- Preloži obroč z D na T
- Preloži obroč z P na T

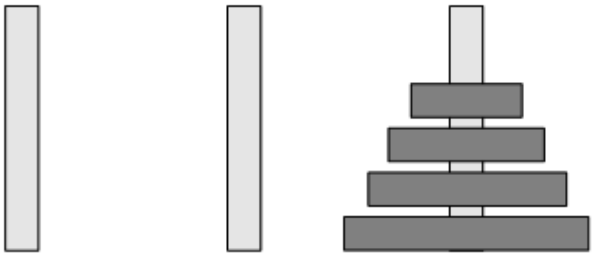
Pri problemu Hanoiskih stolpičev imamo torej začetno stanje



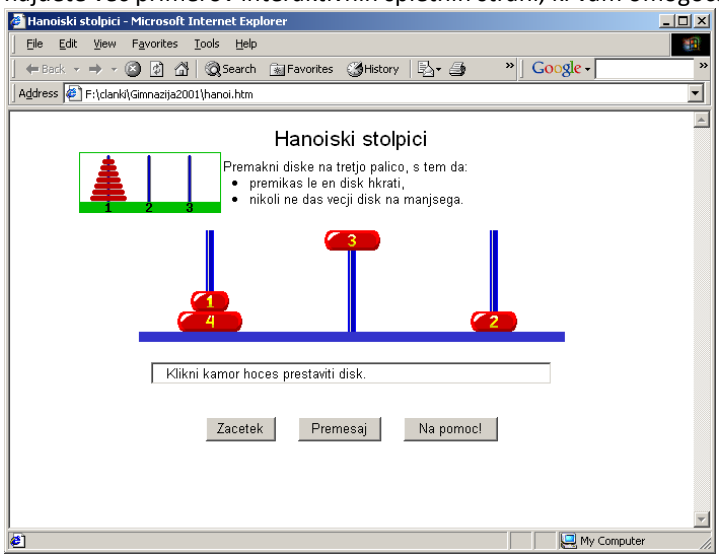
S prelaganjem po enega obroča hkrati, pri čemer pa moramo upoštevati pravilo "nikoli večji na manjšega"



moramo doseči končno stanje

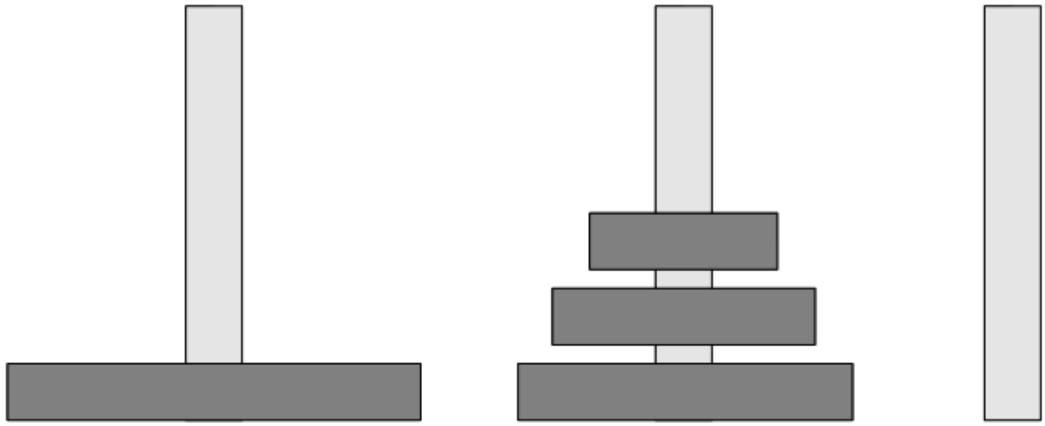


Da boste bolj spoznali problem, lahko na spletu (med drugim tudi v spletni učilnici, ki podpira ta predmet) najdete več primerov interaktivnih spletnih strani, ki vam omogočajo igranje te igre.



Hanoiski stolpiči - ideja

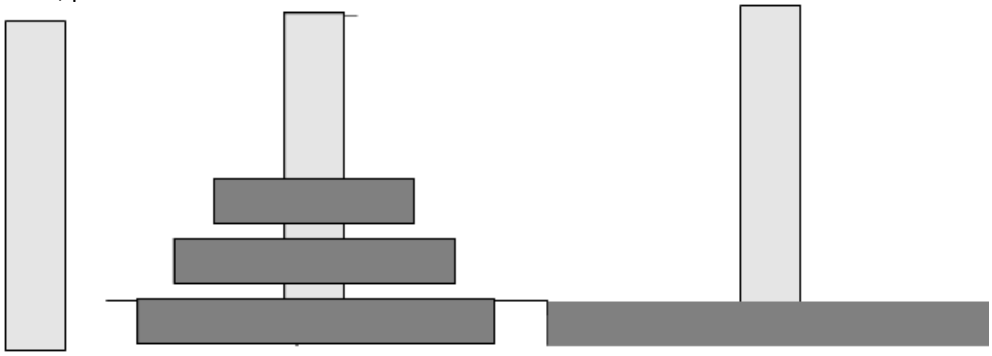
Potem, ko se nekoliko igramo z obroči, hitro spoznamo, da je eno od ključnih stanj pri premikanju tole:



Namreč, če želimo prestaviti največji obroč s stolpa A na stolp C, ne sme biti na stolpu C nobenega obroča (saj bi drugače dajali večji obroč na manjšega). Na stolpu A pa seveda tudi ne sme biti nobenega drugega obroča kot ta zadnji, saj drugače ne moremo do njega. Torej je vseh n – 1 obročev na srednjem stolpu, seveda pravilno zloženih. Vemo tudi, da prej največjega, n-tega obroča, nismo nič premikali (no, ko enkrat dosežemo opisano stanje, si ga lahko izmenično podajamo med A in C, a to nima smisla). Zato če za hip odmislimo največji obroč vidimo, da moramo za dosego zgoraj opisanega stanja pravzaprav rešiti zelo podoben problem.

premakni n-1 obročev z A na B (s pomočjo C)

Ko to naredimo, prestavimo obroč z A na C



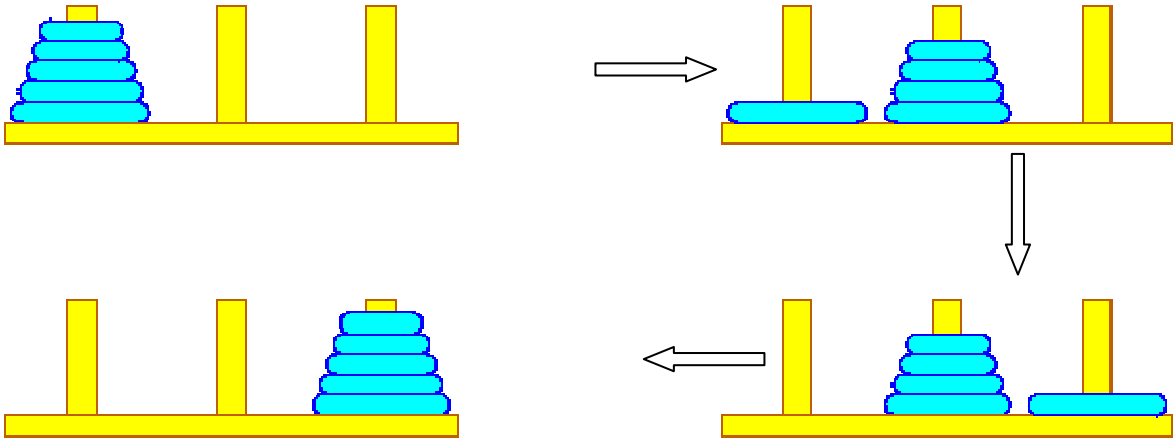
In če sedaj pogledamo sliko, vidimo, da nas največji obroč na C-ju ne bo nič motil – pridno bo čakal na svojem mestu. Zato ga lahko odmislimo. In spet smo pred podobnim problemom

premakni n –1 obročev z B na C (s pomočjo A)

In ko rešimo še ta problem, smo pri končni rešitvi. Če torej postopek reševanja zapišemo shematsko

- Preloži n obročev z A na C s pomočjo B
  - Preloži n-1 obročev z A na B s pomočjo C
  - Daj obroč z A na C
  - Preloži n-1 obročev z B na C s pomočjo A

in če zadevo še enkrat narišemo



Pri opisu postopka smo torej uporabili dva rekurzivna klica. Torej, da rešimo problem velikosti n, moramo rešiti dva enaka problema velikosti n – 1. Vsakič se sicer vloge strbrov malo zamenjajo, a to ne moti …
Seveda je potrebno napisati tudi ustrezni ustavitveni pogoj. Če rešujemo problem le za 1 obroč, potem ni kaj premišljevati. Le damo obroč z A na C in gotovi smo. Če torej upoštevamo še to, je postopek reševanja:

- Preloži n obročev z A na C s pomočjo B
  - Če je n == 1,
    - Daj obroč z A na C
  - sicer pa // n > 1
    - Preloži n-1 obročev z A na B s pomočjo C
    - Daj obroč z A na C
    - Preloži n-1 obročev z B na C s pomočjo A

In če zapišemo še malo bolj računalniško

```
/* n obročev z A na C s pomočjo B */
Hanoi(n, A, B, C)
  Če je n = 1 potem daj obroč z A na C
  sicer pa
     Hanoi(n-1, A, C, B)
     Daj obroč z A na C
     Hanoi(n-1, B, A, C)
```

Sedaj ni več težav s tem, da napišemo ustrezno metodo v jeziku C#

```
1: public static void hanoi(int n, string st1, string st2, string st3)
2: {
3:     if (n == 1)
4:     {
5:       System.Console.WriteLine("Preloži z " + st1 + " na " + st3);
6:     }
7:     else
8:     {
9:        hanoi(n-1, st1, st3, st2);
10:       System.Console.WriteLine("Preloži z " + st1 + " na " + st3);
11:       hanoi(n-1, st2, st1, st3);
12:    }
```

```
13: }
```

Poglejmo, kaj se dogaja, ko pokličemo `hanoi(3, "A", "B", "C")`

Naredimo "škatlico" za to metodo in vanjo shranimo n = 1, st1 = "A", st2 = "B" in st3 = "C".
Po preverjanju pogoja pride do klica hanoi(2, "A", "C", "B"). Zato si izvajalno okolje zapomni trenutne vrednosti škatlice (n, st1, st2, st3) in to, da se mora vrniti po končanem omenjenem klicu na konec vrstice 9. Sedaj se naredi nova škatlica za izvajanje metode. V n te nove škatlice shranimo 2, v st1 = "A", v st2 = "C" in v st3 = "B". Po preverjanju pogoja se tudi pri izvajanju te metode pride do klica Hanoi(1, "A", "B", "C"). Zato si izvajalno okolje zapomni trenutne vrednosti škatlice (n, st1, st2, st3) in to, da se mora vrniti po končanem omenjenem klicu (torej po končanem klicu hanoi(1, "A", "B", "C")) na konec vrstice 9. Sedaj se naredi nova škatlica za izvajanje metode. V n te nove škatlice shranimo 1, st1 = "A", st2 = "B" in st3 = "C". Ko se preveri pogoj, vidimo, da je na vrsti vrstica z izpisom. Ta izpiše niz

"Preloži z " + st1 + " na " + st3

kjer st1 in st3 zamenja z vrednostmi (katerimi – ja tistimi svojimi, iz trenutne škatlice, torej z "A" in "C". Izpiše se torej

Preloži z A na C

Sedaj je konec pogojnega stavka in s tem tudi konec izvajanja klica metode hanoi(1, "A", "B", "C"). Zato se škatlica izvajanja te metode "uniči" in vrnemo se tja, od kjer smo bili poklicani. To je na konec 9. vrstice v izvajanju klica metode hanoi(2, "A", "C", "B"). Sedaj sledi 10 vrstica. Ta izpiše niz

"Prelozi z " + st1 + " na " + st3

kjer st1 in st3 zamenja z vrednostmi (katerimi – ja tistimi svojimi, iz trenutne škatlice, torej z "A" in "B". Izpiše se torej

Prelozi z A na B

Sedaj je na vrsti 11. vrstica. Ne pozabimo, trenutne vrednosti n, st1, st2 in st3 so 2, "A", "C" in "B". Pride torej do klica hanoi(1, "C", "A", "B"). Zato si izvajalno okolje zapomni trenutne vrednosti škatlice (n, st1, st2, st3, torej 2, "A", "C" in "B") in to, da se mora vrniti po končanem omenjenem klicu (torej po končanem klicu hanoi(1, "C", "A", "B")) na konec vrstice 11. Sedaj se naredi nova škatlica za izvajanje metode. V n te nove škatlice shranimo 1, st1 = "C", st2 = "A" in st3 = "B". Ko se preveri pogoj, vidimo, da je na vrsti vrstica 3. Ta izpiše niz

Prelozi z C na B

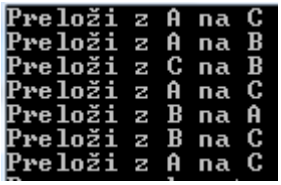
Sedaj je konec pogojnega stavka in s tem tudi konec izvajanja klica metode hanoi(1, "C", "A", "B"). Zato se škatlica izvajanja te metode "uniči" in vrnemo se tja, od kjer smo bili poklicani. To je na konec 11. vrstice v izvajanju klica metode hanoi(2, "A", "C", "B"). S tem pa smo tudi ta klic končali. Torej se vrnemo tja, od koder smo bili poklicani. To pa je konec 9 vrstice izvajanja klica Hanoi(3, "A", "B", "C"). Sedaj je na vrsti 10. vrstica, ki izpiše

Prelozi z A na C

V 11. vrstici spet pride do klica, tokrat hanoi(2, "B", "A","C") …
Če shemo klicanja napišemo shematsko

```
hanoi(3, "A", "B", "C")
        hanoi(2, "A", "C", "B")
                hanoi(1, "A", "B", "C")
                        izpis: Prelozi z A na C
                izpis: Prelozi z A na B
                hanoi(1, "C", "A", "B")
                        izpis: Prelozi z C na B
        izpis: Prelozi z A na C
        hanoi(2, "B", "A", "C")
                hanoi(1, "B", "C", "A")
                        izpis: Prelozi z B na A
                izpis: Prelozi z B na C
                hanoi(1, "A", "B", "C")
                        izpis: Prelozi z A na C
```

Izpis programa:



In koliko resnice je v legendi? Izračunamo lahko število premikov, ki jih morajo opraviti svečeniki. Premikov je točno 264 -1. Če bi vsako sekundo premaknili en obroč, bi potrebovali malo več kot 580 bilijonov let. In če upoštevamo, da je vesolje staro približno 13,7 bilijonov let, smo kar nekaj časa še lahko mirni!

MinMax

Dano imamo tabelo celih števil. Radi bi sestavili rekurzivno metodo public static int[] MinMax(int[] tabela), ki vrne tabelo dveh elementov. V odg[0] je minimalni element tabele stevila, v odg[1] pa maksimalni element tabele stevila.

Ideja:

- tabelo razpolovimo
- poiščemi min/max prvega dela (prvih pol elementov)
- poiščemi min/max drugega dela (druga polovica elementov)
- na podlagi min/max obeh delov poiščemo min/max celotne tabele

Npr.:

- Naj bo tabela {23, 4, 546, 56, 776, 8}.
- Poiščemo min/max {23, 4, 546} --- 4 in 546
- Poiščemo min/max {56, 776, 8} --- 8 in 776
- Odgovor za celo tabelo je 4 in 776

```
public static int[] minMax(int[] stevila)
{
   int[] rez = new int[2];
   if (stevila.Length == 1)
   { // le en element, vemo rezultat
      rez[0] = stevila[0];
      rez[1] = stevila[0];
      return rez;
   }
   // tabela je dolga, razpolavljamo
   int dol = stevila.Length;
   int[] prvidel = new int[dol/2];
   int[] drugidel = new int[dol - dol/2]; // new int[dol/2 + dol%2]
   int i = 0;
   // preložimo prvih pol elementov v tabelo prvidel
   while (i < dol/2)
   {
      prvidel[i] = stevila[i];
      i = i + 1;
   }
   // preložimo drugih pol elementov v tabelo drugidel
   i = 0;
   while (i < drugidel.Length)
   {
      // prelagamo drugi del tabele stevila, zato + dol/2!
      drugidel[i] = stevila[i + dol/2];
      i = i + 1;
```

```
    }
    // poiščemo min/max prvega dela
    int[] rezPrviDel = minMax(prvidel);
    // poiščemo min/max drugega dela
    int[] rezDrugiDel = minMax(drugidel);
    // določimo minimum celotne tabele
    if (rezPrviDel[0] < rezDrugiDel[0])
    {
        rez[0] = rezPrviDel[0];
    }
    else {
        rez[0] = rezDrugiDel[0];
    }
    // določi maksimum celotne tabele
    if (rezPrviDel[1] > rezDrugiDel[1])
    {
        rez[1] = rezPrviDel[1];
    }
    else
    {
        rez[1] = rezDrugiDel[1];
    }
    // vrni rezultat
    return rez;
}
```

MinMax – rešitev II

Zgornja rešitev ni najboljša, ker imamo veliko dela s prelaganjem elementov. Sestavi rešitev, kjer to ne bo potrebno!
Namig: sestavi pomožno metodo public static int[] MinMax(int[] tabela, int odKje, int DoKje), ki poišče minimalni in maksimalni element v delu tabele od odKje do doKje.

Palindrom

S pomočjo rekurzije preveri, če je niz palindrom.
Ideja:

- Prazen niz oziroma niz z enim znakom je palindrom
- Niz je palindrom, če se ujemata prvi in zadnji znak in je tudi srednji del (niz) palindrom

```
public static bool JePalindrom(string niz)
{
    // s pomočjo rekurzije ugotovi, če je niz niz palindrom
    // niz dolzine 0 in niz dolzine 1 JE palindrom
    if (niz.Length < 2) return true;
    // preveri prvi in zadnji znak
    if (niz[0] != niz[niz.Length-1]) return false;
    // vemo, da sta prvi in zadnji enaka
    string srednjiDel = niz.Substring(1, niz.Length - 1);
    // niz BO palindrom, če je srednji del tudi palindrom
    return JePalindrom(srednjiDel);
}
```

Številski sestav

Napišimo rekurzivno metodo za pretvarjanje iz desetiškega sestava v poljuben številski sestav med 2 in 9.

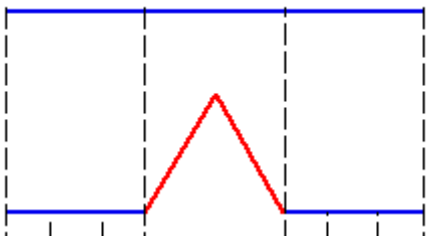
```
static void pretvori (int n,int s)
{
    if (n>0)
    {
        pretvori(n / s,s);
        Console.Write(n%s);
    }
    else Console.Write("\nPretvorjeno število : ");
}
```

Klic metode v glavnem programu:

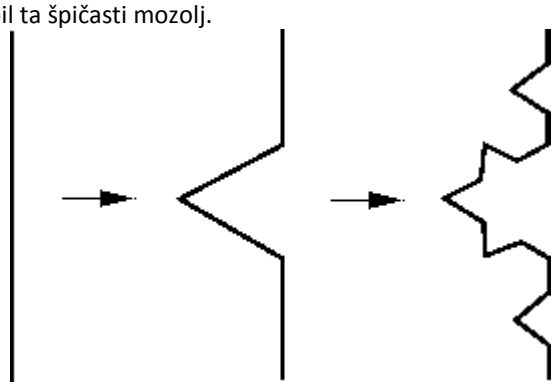
```
static void Main(string[] args)
{
    int n, s;
    Console.Write("Vnesi število : ");
    n=Convert.ToInt32(Console.ReadLine());
    Console.Write("\nŠtevilski sestav : ");
    s=Convert.ToInt32(Console.ReadLine());
    pretvori(n, s);
    Console.WriteLine();
}
```

Črta dobi mozolje

Zvečer je bila črta še čisto normalna. Lepo gladka je potekala od točke A do točke B. A zjutraj se je zbudila s čudnim občutkom. Odtavala je pred ogledalo in groza! Ni bila več lepo gladka. Nad njeno srednjo tretjino se je bohotil mozolj. Ampak kakšen – špičast, trikoten z robovi kar take dolžine, kot je bila prej dolžina srednje črte.



In pri tem se ni ustavilo. Naslednje jutro je vsak njen raven delček dobil nov mozoj – in to enak. Spet je nad srednjo tretjini ravnega dela bil ta špičasti mozolj.



In tako je šlo dan za dnem.

Končno ji je njena najboljša prijateljica, krožnica, povedala za čudovito kremo! Če se namaže z njo po vsakem delčku svoje kože, bo rast mozoljev vsaj ustavljena.

A krema je draga ... In za vsak cm potrebuje črta vsaj 6g te kreme. Koliko jo mora kupiti, če je bila na začetku dolga d in je preteklo že n dni, kar je dobivala mozolje?

```
public static double Dolzina(double razd, int dni)
{
     // izračunamo koliko je dolga črta, če se
     // stvar dogaja na delu njene kože v razdalji razd
     // in mozolje dobiva že n dni
     if (dni == 0)
     { // nobenega mozolja še ni
        // koža je gladka, torej je dolžina
        // kar enaka razdalji
        return razd;
     }
     else
     {
        // dolžina je enaka dogajanjem na 4 delih,
        // pri vsakem delu na razdalji 1/3 te
        // a za en dan pridobivanja mozoljev manj
        return 4 * Dolzina(razd / 3, dni - 1);
     }
}
```

Naloge za utrjevanje znanja iz rekurzij

- Dana je naslednja rekurzivna metoda. Ugotovi, kaj dela. Šele potem!! jo prenesi v testni program in zaženi ter preveri, da res dela tisto, kar si si predstavljal.

```
public static string KajDelam(int stevec)
{
   if(stevec <= 0)
     {
      return "";
     }
     else
     {
       return "" + stevec + ", " + KajDelam(stevec - 1);
     }
}
```

Metodo nato prepiši tako, da bo vrnila niz s števili v obratnem vrstnem redu kot prvotna.

- Nekega dne je Ančka vsa obupana prosila brata Jureta, naj ji napiše program za domačo nalogo. Ta bi moral rekurzivno izračunati vsoto prvih n naravnih števil. A ker se je Juretu mudilo na vsakodnevni žur, je moral zelo hiteti in je zato v programu naredil nekaj napak. Jih najdeš?

```
public static int Vsota(int n)
{
     if(n > 0)
      {
        return 1;
     }
      else
```

```
      {
        return vsota(n-1);
      }
}
```

- Jure je napisal kodo, ki s pomočjo rekurzije izračuna vsoto 1+1/2+1/3+...+1/n. Žal mu je med sprehodom po Ljubljani list s kodo padel v sneg in se je nekaj kode izbrisalo. Dopolni jo!

```
public static double VsotaRec(int n) // n>=1
{
    if (____________)  // ustavitveni pogoj
    {
       return _______________;
    }
    return _______ + VsotaRec(_______); //rekurzivni klic
}
```

- Oglej si naslednji rekurzivni program:

```
public static int puzzle(int baza, int limit)
{
//base in limit sta nenegativni števili
    if (baza > limit)
  {
    return -1;
  }
  else
  {
      if (baza == limit)
    {
      return 1;
    }
    else
    {
        return baza * puzzle(baza + 1, limit);
    }
  }
}
```

Program si najprej oglej, nato pa BREZ uporabe računalnika odgovori na spodnja vprašanja:

- kateri del metode Puzzle je ustavitveni pogoj?
- kje se izvede rekurzivni klic?
- kaj izpišejo sledeči stavki:
  - Console.WriteLine(Puzzle(14,10));
  - Console.WriteLine(Puzzle(4,7));
  - Console.WriteLine(Puzzle(0,0));

- Napiši rekurzivni podprogram, ki dobi za parameter poljubno celo število (npr. 1234567) vrne pa celo število v katerem so cifre zapisane v obratnem vrstnem redu (v našem primeru 7654321).

- Sestavi rekurzivno funkcijo, ki izračuna največji skupni delitelj dveh pozitivnih števil, če veš, da velja:

gcd(n, n) = n
gcd(n, k) = gcd(n - k, k) za n > k
gcd(n, k) = gcd(k, n) za n < k

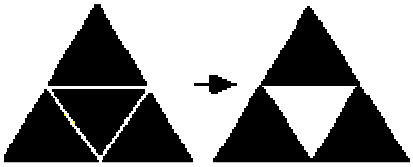
- Sestavi program, ki s pomočjo rekurzije izpiše vse permutacije števil od 1 do n. Permutacije naj bodo izpisane v leksikografskem vrstnem redu.

  1 2 3
  1 3 2
  2 1 3
  2 3 1
  3 1 2
  3 2 1

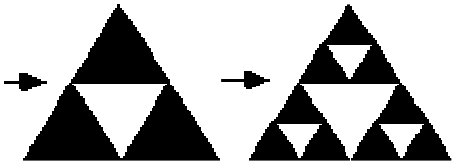
- Napiši rekurzivno funkcijo za izpis **poštevanke** poljubnega števila!

- S pomočjo rekurzije napiši program, ki izračuna vsoto vrste $1^0 + 2^1 + 3^2 + 4^3 + \ldots + x^{(x-1)}$.

- Napiši rekurzivno metodo, ki izračuna ploščino trikotnika Sierpinskega dane stopnje. Osnovni trikotnik je kar črni enakokraki trikotnik. Trikotnik stopnje 1 dobimo tako, da iz osnovnega trikotnika izrežemo trikotnik, katerega oglišča so razpolovišča stranic osnovnega trikotnika. Na tak način dobimo 3 manjše črne trikotnike spodaj levo, spodaj desno in zgoraj:



Trikotnik 2. stopnje dobimo iz trikotnika stopnje 1 tako, da izrežemo trikotnike iz treh črnih trikotnikov na enak način, kot smo ga izrezali iz osnovnega trikotnika:



Trikotnik tretje stopnje spet dobimo tako, da iz vsakega črnega trikotnika v trikotniku 2. stopnje ponovno izrežemo trikotnik itd.
Namig: namesto da »izrežeš« beli trikotnik, raje računaj ploščine črnih trikotnikov – razmisli, kako na tak način prideš iz ene stopnje do druge!
Alternirajoče vsote
Sestavi metodi

- public static int Vsota (int n) , ki izračuna vsoto naravnih števil od ena do n.
- public static int AltVsota (int n) , ki izračuna alternirajočo vsoto naravnih števil med ena in n. Tako klic AltVsota (4) izračuna vsoto 1 - 2 + 3 - 4 = -2.

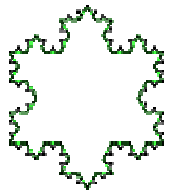
Obe metodi napiši nerekurzivno in rekurzivno!

- Jure je razvil napreden postopek, ki na podlagi makroekonomskih podatkov predvidi gibanje cen delnic podjetij. Algoritem za postopek imenuje JurePostopek. Algoritem sprejme kot parameter tabelo celih števil, deluje pa po sledečih pravilih:
  - če tabela prazna, naj vrne 0
  - če je dolžina tabele enaka 1, vrne ta element povečan za ena
  - sicer vrne kot rezultat (a+c)*JurePostopek(tabela brez prvega elementa), kjer je a prvi, c pa zadnji element tabele

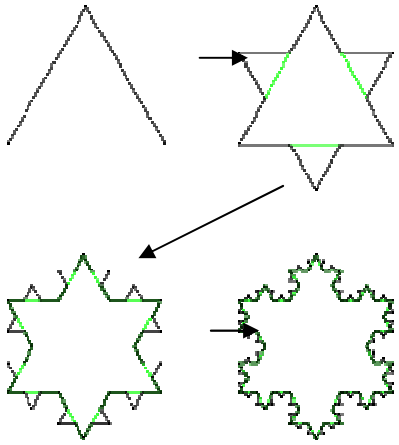
Primer izvedbe postopka na tabeli [1, 2, 3]:
JurePostopek([1, 2, 3]) = (1 + 3) * JurePostopek([2, 3]) = 4 * 20 = 80
JurePostopek([2, 3]) = (2 + 3) * JurePostopek([3]) = 5 * 4 = 20
JurePostopek([3]) = (3 + 1) = 4

Sestavi metodo, ki deluje po principu Juretovega postopka.

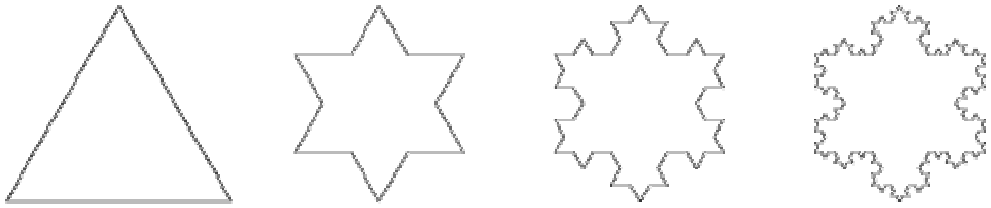
- Na sliki je Kochova snežinka stopnje 4



To lepo snežinko lahko dobimo z naslednjim postopkom.

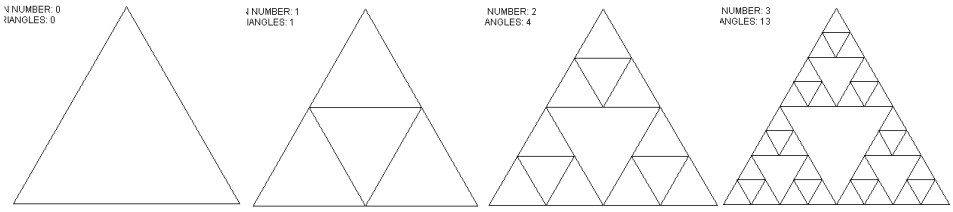


Stopnja 0 le kar enakostranični trikotnik. Nad vsako stranico podobno kot pri Kochovi črti potem srednji del nadomestimo z enakostraničnim trikotnikom.



Sestavi metodo, ki izračuna ploščino Kochove snežinke stopnje d in začetno stranico d.

- Trikotnik Sierpinskega stopnje 0 je običajni, enakostranični trikotnik s stranicami dolžine d. Trikotnik Sierpinskega stopnje n in s stranico d pa nastane, če zložite skupaj tri trikotnike Sierpinskega s pol krajšimi stranicami in stopnje n – 1. Na sliki so zaporedoma trikotniki Sierpinskega vsi z enako dolgimi stranicami, a stopnje 0, 1, 2 in 3.



Denimo, da bi radi iz žice sestavili model trikotnika Sierpinskega. Sestavi metodo, ki vrne true, če je možno iz žice dolžine d sestaviti trikotnik Sierpinskega stopnje st in s stranico s. Tako npr. je iz žice dolžine 20 sestaviti trikotnik Sierpinskega stopnje 1 s stranico 4, saj je skupna dolžine vseh črt v trikotniku

Sierpinskega stopnje 1 in s stranico 2 enaka 18. Iz žice dolžine 10 pa ni mogoče sestaviti trikotnik Sierpinskega stopnje 0 s stranico 4, saj bi za to potrebovali vsaj žico dolžine 12.

Funkcije (metode) - nadgradnja

Funkcija Main()

Ko poženemo katerikoli program, se le-ta začne izvajati v funkciji **Main()**, ki je lahko zapisana kjer koli v programu. Funkcija Main() je torej vstopna točka našega programa, zaključek te funkcije pa je obenem tudi zaključek našega programa. Deklarirana je **znotraj razreda** (pojem razreda bomo opisali v nadaljevanju), **mora biti statična** (pojem statične funkcije bo razložen v poglavju o razredih in objektih) in **ne sme biti public**. Vedno lahko obstaja le ena funkcija Main(). V glavi funkcije Main() je najprej deklaracija tipa, ki ga funkcija vrača, temu sledi ime funkcije in tabela parametrov funkcije Main() - **`string[] args`**. Pomen te tabele parametrov je naslednji: če pri zagonu programa za njegovim imenom zapišemo še parametre (če je parametrov več, so med seboj ločeni s presledki), se le-ti shranijo v to tabelo. Tabela parametrov ni obvezna in jo lahko izpustimo. Za razliko od C in C++ prvi argument te tabele **NI** ime programa.

Funkcija **Main()** lahko torej prav tako sprejme parametre. Ker pa v fazi razvoja programa ne moremo vedeti, kakšne parametre bo uporabnik napisal, v funkciji Main() ne moremo preprosto določiti, naj sprejme **int**, **double**, **char** ipd. Vsi parametri funkcije so shranjeni v tabeli, kjer je vsak element tabele nek niz znakov (konkreten parameter). Tako v funkciji Main() potrebujemo samo en parameter – ime tabele, v kateri so shranjeni njeni parametri

Primer :

```
//Napišimo naslednji program in ga shranimo pod imenom Main1
class Program
{
   static void Main(string[] args) //tabeli parametrov je ime args
   {
      //Izpis skupnega števila parametrov ukazne vrstice programa
      System.Console.WriteLine("Skupno število parametrov: "+args.Length);
      Console.ReadLine();
   }
}
```

Če ga poženemo :

Main1 arg1 arg2 arg3 arg4 <ENTER>

dobimo izpis :

```
Skupno število parametrov: 4
```

Še en primer:

```
//Napišimo naslednji program in ga shranimo pod imenom Main2
static void Main(string[] args)
{
   //Izpis skupnega števila parametrov ukazne vrstice programa
   System.Console.WriteLine("Skupno število parametrov: " + args.Length);
   for (int i = 0; i < args.Length; i++)
      Console.WriteLine(args[i]);
   //Lahko pa uporabimo tudi zanko foreach
   /*
   foreach (string s in args)
   {
      System.Console.WriteLine(s);
   }
   /*
```

```
   Console.ReadLine();
}
```

Če ga poženemo :

Main2 Prvi Drugi Tretji <ENTER>

dobimo izpis :

```
Skupno število parametrov: 3
Prvi
Drugi
Tretji
```

V tabelo z imenom **args** se namreč zaporedoma shranijo vsi trije argumenti, ki smo jih napisali v ukazni vrstici ob klicu programa **Main2**. V elementu tabele z indeksom 0 je torej zapisan string ***Prvi***, v elementi z indeksom 1 je zapisan string ***Drugi***, v elementu z indeksom 2 pa string ***Tretji***.

Vsak niz, ki je ločen s presledkom, je za program nov argument. Včasih pa je to ovira, zato lahko argument, ki vsebuje presledke, zapremo med dvojne narekovaje.

Če torej program Main2 poženemo npr. takole:

Main2 "Celoten stavek je en sam argument" <ENTER>

dobimo izpis :

```
Skupno število parametrov: 1
Celoten stavek je en sam argument
```

Ker je **Main()** funkcija, je lahko tipa **void** (ne vrača ničesar) ali pa ima tip, kar pomeni da vrača rezultat. Če ima funkcija Main() tip, je to običajno **celoštevilski tip – int**. Vračanje celoštevilskega rezultata omogoča drugim programom, da le-tega uporabijo, ko se program v C# konča. Najpogosteje se rezultat uporabi npr. v kakih **batch** programih. Ko se program v C# izvaja npr. v Windows okolju, se vsaka vrnjena vrednost funkcije Main() shrani v okoljsko spremenljivko z imenom `ERRORLEVEL`. S testiranjem le-te, lahko nek **batch** program ugotovi, kakšen je bil rezultat programa v C# (oz. kakšno vrednost je vrnila funkcija Main()). Tradicionalno pomeni vrnjena vrednost **nič** (**0**) uspešen zaključek funkcije **Main()**.

Vaja:

Napiši program, ki bi ga klicali npr. **naravna 10 20** in ki izračuna in izpiše vsoto zapisanih števil (v našem primeru sta števili 10 in 20, vsota pa torej 30)!

```
static void Main(string[] args)
{ //Program sprejme dva parametra - dve celi števili in izračuna ter izpiše njuno vsoto
   if (args.Length == 2)
   {
       int prvo = Convert.ToInt32(args[0]);
       int drugo = Convert.ToInt32(args[1]);
       Console.WriteLine("Vsota: " + (prvo + drugo));
   }
   else Console.WriteLine("Napačno število parametrov!");
}
```

Vaja:

Napiši funkcijo, imenovano **Potenca()**, ki sprejme dva parametra: poljubno realno število in poljubno celo število. Program naj izračuna in izpiše vrednost ustrezne potence – osnovo in eksponent vnesemo v ukazni vrstici ob zagonu programa. Klic programa **Potenca 10 3** naj torej vrne 1000 (ker je 10^3 enako 1000).

```
static double potenca(double st1, int st2)
{
    double temp = st1;
    if (st2 == 0) return 1;
    if (st2 == 1) return st1;
    if (Math.Abs(st2) > 1)
    {
        for (int i = 1; i < Math.Abs(st2); i++)
           st1 = st1 * temp;
    }
    if (st2 > 0) return st1;
    else return 1 / st1;
}

static void Main(string[] args)
{
    if (args.Length == 2)   //preverimo, če smo v ukazni vrstici vnesi dva argumenta
    {
        double st1 = Convert.ToDouble(args[0]);
        int st2 = Convert.ToInt32(args[1]);
        Console.WriteLine("Rezultat: "+potenca(st1, st2));
    }
}
```

Vaja:

```
/*Program za izračun faktoriele poljubnega celega števila. Faktoriela je matematična
operacija, definirana takole:
N!=N * (N-1) * (N – 2) * … * 3 * 2 * 1
Konkretno: 7! = 7 * 6 * 5 * 4 * 3 * 2 * 1
*/
static long Fakt(long i)
{
    return ((i <= 1) ? 1 : (i * Fakt(i - 1)));
}

static int Main(string[] args)
{
   // Test vnosa vhodnega parametra (število, katerega faktorielo želimo izračunati)
   if (args.Length == 0)
   {
        Console.WriteLine("Vnesi prosim numerični argument.");
        Console.WriteLine("Uporaba: Main-Faktoriela <celo število>");
        return 1;
   }
   else
   {
        // Pretvorba vhodnega argumenta v celo število:
        long num = Convert.ToInt64(args[0]);
        Console.WriteLine("Faktoriela od {0} je {1}.", num, Fakt(num));
        return 0;
   }
}
```

Naloge:

- Napiši program, ki mu v ukazni vrstici podamo poljubno število besed, program pa med njimi poišče najdaljšo in najkrajšo besedo!

Preobložene (overloaded) metode

V C# imata lahko dve ali pa več metod enako ime, razlikovati pa se morata v deklaraciji parametrov. Za take metode pravimo da so preobložene – **overloaded**, proces pa se imenuje preobložitev (**overloading**). Preobložene funkcije so torej funkcije z enakim imenom, ki pa se razlikujejo v številu ali pa v tipu parametrov.

Primer:

```
//vse tri funkcije imajo enako ime, a se razlikujejo v številu parametrov - so preobložene

static void Funkcija(string drzava, string padavine, string temp)
{
    Console.WriteLine(drzava + ": " + padavine+" in "+temp);
}

static void Funkcija(string drzava, string padavine)
{
    Console.WriteLine(drzava+": "+padavine);
}

static void Funkcija()
{
    Console.WriteLine("Slovenija: sneg in mraz!");
}

static void Main(string[] args)
{
   Funkcija();                          //Izpis: Slovenija: sneg in mraz!
   Funkcija("Slovenija","dež");         //Izpis: Slovenija: dež!
   Funkcija("Slovenija","dež","toplo"); //Izpis: Slovenija: dež in toplo!
}
```

Vaja:

```
//Preobloženi funkciji za izračun vsote cifer naravnih števil med 0 in n, ter za izračun vsote
cifer med dvema naravnima številoma n in m
static int vsota(int n)
{
    int suma = 0;
    for (int i = 0; i <= n; i++)
       suma += i;
    return suma;
}
static int vsota(int n, int m)
{
    int suma = n;
    for (int i = 0; i <= m; i++)
        suma += i;
    return suma;
}

static void Main(string[] args)
{
    Console.WriteLine("Vsota cifer med 0 in 10 je "+vsota(10));
    Console.WriteLine("Vsota cifer med 5 in 10 je " + vsota(5,10));
}
```

Vaja:

```
//Še en primer preobloženih funkcij in avtomatske konverzije argumentov
static void f(int x)
{
     Console.WriteLine("Znotraj funkcije f f(int): " + x);
}

static void f(double x)
{
     Console.WriteLine("Znotraj funkcije f(double): " + x);
}
```

```
static void Main()
{
    int i = 10;
    double d = 10.1;

    byte b = 99;
    short s = 10;
    float x = 11.5F;

    f(i); // klic f(int)
    f(d); // klic f(double)

    f(b); // klic f(int), avtomatska konverzija byte v int
    f(s); // klic f(int), avtomatska konverzija short v int
    f(x); // klic f(double), avtomatska konverzija float v double
}
```

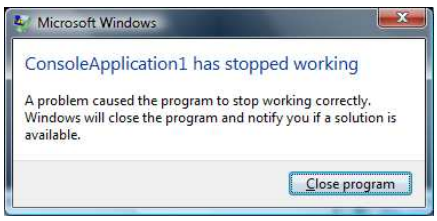
Varovalni bloki (osnove) – obravnava napak in izjem (Exceptions)

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. Na primer, da se uporabi negativni indeks, ali pa da datoteka, kamor naj bi se zapisali rezultati, ne obstaja, da uporabnik kot število, s katerim naj se deli, po pomoti vnese 0 in podobno. Dobro je, če imamo možnost, da v primeru napak program ustrezno reagiramo, pa če drugega ne, izpiše prijazno obvestilo, da je žal prišlo do take in take napake in da bo zaradi tega program nehal 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.

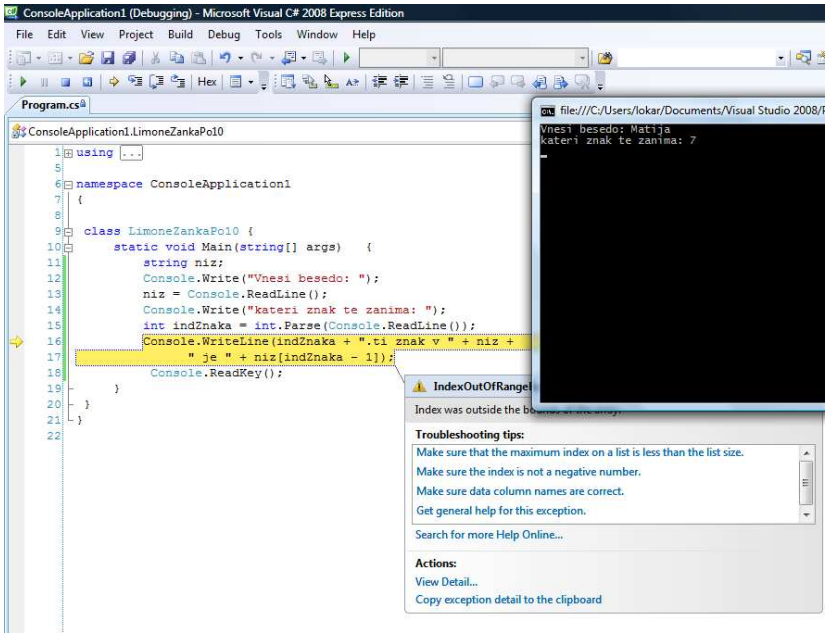
Poglejmo si primer, Napisali smo program, ki prebere niz in uporabniku omogoči, da vnese indeks znaka, ki ga zanima.

```
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



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.



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.

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

- blok za obravnavo prekinitev **try…catch …**,

- večkratni varovalni blok **try … catch … catch …**
- brezpogojni varovalni blok **try … finally …** .

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());
   double rezultat = levo / desno;
   Console.WriteLine(rezultat);
   napaka = false;
 }
 catch
 {
   Console.WriteLine("Ponovno vnesi podatke");
   napaka = true;
 }
```

običajna programska koda

koda za obdelavo napake

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.

Primer:

```
try
{
    int levo = int.Parse(Console.ReadLine());
    int desno = int.Parse(Console.ReadLine());
    double rezultat = levo / desno;
    Console.WriteLine(rezultat);
}
catch (System.Exception napaka)
{
    // Vrsto napake izpišemo na zaslonu
    Console.WriteLine(napaka.Message);
}
```

običajna programska koda

koda za obdelavo napake

V zgornjem primeru je uporabljena kar **splošna** metoda za prestrezanje napake (`System.Exception`); le-ta se odzove/odkrije na vsako izjemo (napako), tudi tako, ki se je morda sploh ne zavedamo. V takem primeru lahko blok **catch** uporabimo tudi brez parametrov takole:

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

Seveda pa obstajajo tudi metode za obdelavo posameznih vrst napak: **`System.FormatException`** (to metodo uporabimo na primer za obdelavo uporabnikovih napačnih vnosov podatkov), **`System.DivideByZeroException`**, **`System.EArgument.Exception`**, …) . Vsaka od teh metod vrne ustrezno obvestilo o napaki, ki ga lahko izpišemo v konzolnem. Obvestilo o napaki je v tem primeru v angleščini; v kolikor pa želimo uporabnikom ponuditi prijazna obvestila o napaki, jih je potrebno napisati posebej, npr. takole:

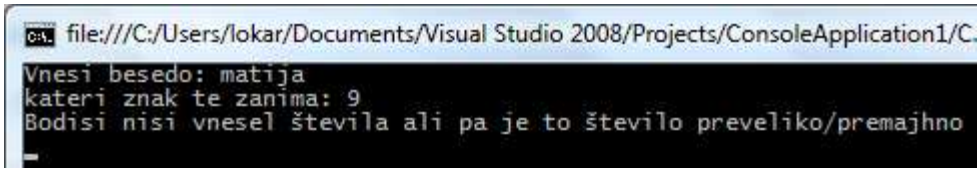
Izjeme lahko prožimo tudi sami programsko. Vendar se v začetnem spoznavanju jezika C# z vsem tem ne bomo ubadali.

Zgled, ki smo ga navedli na začetku, dajmo v varovalni blok.

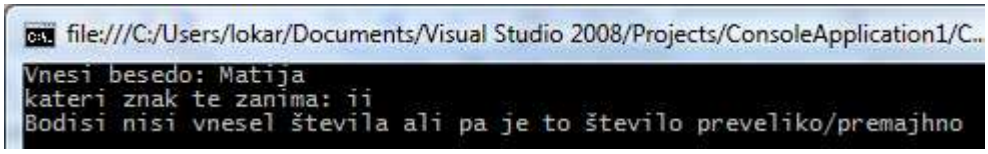
```
    static void Main(string[] args)
    {
        string niz;
```

```
        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



ali pa za indeks ne sploh vnesemo števila



Večkratni varovalni blok za obravnavo prekinitev try … catch … catch …

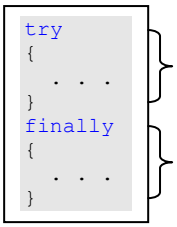
Različne napake seveda proizvedejo različne vrste izjem. Recimo, da imamo operacijo deljenja, pri kateri se lahko zgodi, da je drugi operand enak 0. Izjema, ki se pri tem zgodi, se imenuje **DivideByZeroException**. V takem primeru lahko napišemo večkratni **catch** blok, enega za drugim. Najprej ujamemo najnižji razred izjem, nazadnje pa najvišjega:

```
try
{
   int levo = int.Parse(Console.ReadLine());
   int desno = int.Parse(Console.ReadLine());
   double rezultat = levo / desno;
   Console.WriteLine(rezultat);
}
catch (System.FormatException napaka)
{
   Console.WriteLine("Napaka pri pretvarjanju podatkov!!!");
}
catch (System.DivideByZeroException napaka)
{
   Console.WriteLine("Napaka pri deljenju z 0");
}
```

Seznam vseh možnih izjem dobimo, če v **Debug** meniju kliknemo opcijo **Exceptions**.

Brezpogojni varovalni blok try … finally …

Stavki v bloku **catch** se izvedejo samo ob prekinitvi (napaki), sicer pa se ne izvedejo. Kadar pa za del kode želimo, da se izvede v vsakem primeru, uporabimo brezpogojni varovalni blok **finally**.



Programska koda, ki bi jo napisali tudi če ne bilo varovalnega bloka;

Blok kode, ki se izvede vedno, ne glede ali je v zgornjem bloku prišlo do napake ali ne!

V praksi pogosto uporabljamo tudi kombinacijo obeh metod: blok za obravnavo prekinitev vgnezdimo v brezpogojni varovalni blok:

```
try
{
   // stavki, kjer lahko pride do napake
}
catch
{
   // obdelava napake
}
finally
{
    // blok, ki se izvede vedno, ne glede  na napako
}
```

Vaja:

```
//Varovalne bloke lahko tudi gnezdimo - na ta način lahko npr. uporabnika učinkovito
seznanjamo z napakami pri vnašanju podatkov!
class Narocilo
{
    public string CustomerName;
    public DateTime datumNarocila;
    public DateTime casNarocila;
    public int SteviloEnot;
}

static void Main(string[] args)
{
   Narocilo nar1 = new Narocilo();
   Console.Write("Stranka:  ");
   string stranka = Console.ReadLine();
   try
   {
      Console.Write("Vnesi datum naročila(mm/dd/yyyy):  ");
      nar1.datumNarocila=Convert.ToDateTime(Console.ReadLine());
      try
      {
         Console.Write("Vnesi čas naročila(hh:mm AM/PM): ");
         nar1.casNarocila = Convert.ToDateTime(Console.ReadLine());
         try
         {
             Console.Write("Število enot: ");
             nar1.casNarocila = Convert.ToDateTime(Console.ReadLine());
         }
         catch
         {
             Console.WriteLine("Nepravilen vnos števila enot!");
         }
      }
```

```
        catch
        {
            Console.WriteLine("Nepravilen vnos ure naročila!");
        }
    }
    catch (FormatException)
    {
        Console.WriteLine("Nepravilen vnos datuma");
    }
}
```

Predprocesor

Predprocesorski ukazi

V vseh dosedanjih primerih smo pri prevajanju prevajali celoten projekt/program. Včasih pa se zgodi, da želimo prevesti le del našega programa – npr. pri razhroščevanju ali pa pri izgradnji produkcijske kode.

Predprocesor ni del programskega jezika, nam pa olajša delo pri izdelovanju projektov. Izvorni program vedno najprej prevzame predprocesor, ki prvi pregleda naš program in reagira na vsak predprocesorski ukaz (**ukaz, ki se začne z znakom #),** ki ga prepozna. Na ukaze reagira tako, da spremeni izvorni program, ter ga pripravi za prevajalnik. Predprocesorski ukazi omogočajo definiranje ustreznih označevalcev in nato testiranje njihovega obstoja.

Definiranje predprocesorskih označevalcev/ukazov

#define

Najpogostejši predprocesorski ukaz je **#define**, ki predstavlja definicijo nekega predprocesorskega označevalca. To je edini med predprocesorskimi ukazi, ki mora stati povsem na začetku programa, tudi pred **using** stavki. Stavek **`#define DEBUG`** npr. definira predprocesorki označevalec **`DEBUG`** . Obstoj tega označevalca lahko kasneje prevrejamo s predprocesorskim stavkom **`#if`**.

Primer:

```
#define DEBUG
//... običajna koda - predprocesiranje nanjo nima vpliva
#if DEBUG // preverimo obstoj predproceorskega označevalca DEBUG
// koda, ki jo želimo vključiti pri debugiranju/razhroščevanju
#else
// koda, ki jo želimo vključiti v primeru da DEBUG ne obstaja
#endif
//... ... običajna koda - predprocesiranje nanjo nima vpliva
```

Ko se predprocesor zažene, najde stavek **#define** in tako tudi obstoj označevalca **DEBUG**. V zgornjem primeru zato predprocesor preskoči vse stavke do prvega **#if** - **#else** - **#endif** bloka. **#if** stavek testira obstoj označevalca **DEBUG** – ker ta obstaja, bo koda med **#if** in **#else** vključena v kasnejše prevajanje, koda med **#else** in **#endif** pa ne. Za kodo med stavkoma **#else** in **#endif** torej v tem primeru velja, kot da je sploh ne bi bilo.

Na kodo, ki ni omejena s stavkoma **#else** in **#endif torej** predprocesiranje nima prav nobenega vpliva in bo torej vključena v prevajanje.

Tako kot lahko nek predprocesorski označevalec definiramo (**#define**), ga lahko tudi odstranimo (**#undef**). Predprocesorki označevalec torej obstaja vse do prevega #**undef** stvaka, oz. do tedaj, ko se program konča.

Primer:

```
#define DEBUG
#if DEBUG // preverimo obstoj predproceorskega označevalca DEBUG
// ta koda bo šla v prevajanje
#endif
#undef DEBUG // označevalec DEBUG odstranimo
#if DEBUG // zopet preverimo obstoj predproceorskega označevalca DEBUG
// ta koda se ne bo prevedla
#endif
//..
```

#if, #elif, #else in #endif

Predprocesorski ukaz **switch** ne obstaja, pa saj tudi ni potreben, saj lahko razvejitev rešimo z uporabo predprocesorskih stavkov **#if** , **#elif** in **#else**. Ukaz **#elif** ima enako logiko kot stavek **else if**.

#region in #endregion

Predprocesorski ukaz #regin označi del teksta kot komentar. Osnoven namen tega ukaza je omogočanje orodjem, kot je npr. Visual Studio.NET, da z njim označimo del kode (področje or. regija) in jo nato v editorju začasno skrijemo tako, da se na tem mestu pokaže le vrstica z komentarjem za to področje.

Primer:

Pogled področja/regije v razširjenem stanju (povsem na levi strani te vrstice je v tem primeru kvadratek z znakom -. Če nanj kliknemo, se koda med vrsticama **#region** in **#endregion** skrči:

```
#region Koda za izpis...
   //MojImenskiProstor.Tester.funkcija();
   ...
#endregion
```

Še pogled na kodo v skrčenem stanju (povsem na levi strani te vrstice je v tem primeru kvadratek z znakom +. Če nanj kliknemo, se vrstica zopet razširi):

```
Koda za izpis ...
```

Objektno programiranje

Motivacija

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.

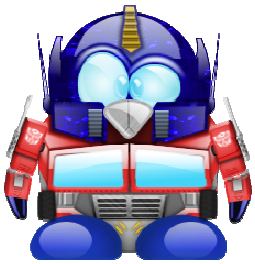
Objektno usmerjenih je danes večina sodobnih programskih jezikov. Pri njihovi uporabi hitro naletimo na pojme, kot so združevanje (**enkapsulacija**), večličnost (**polimorfizem**), dedovanje (**inheritence**), ki so vsaj na prvi pogled zelo zapleteni. Nas podrobnosti ne bodo zanimale in si bomo ogledali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

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.

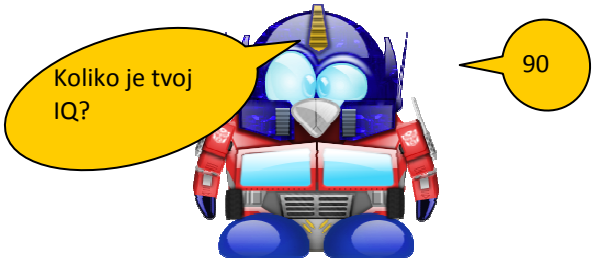
V objektno usmerjenem programiranju (OO) pa sta koda in podatki združeni v nedeljivo celoto – objekt. To prinaša določene prednosti. Ko uporabljamo objekte, nam nikoli ni potrebno pogledati v sam objekt. To je dejansko prvo pravilo OO programiranja – uporabniku ni nikoli potrebno vedeti, kaj se dogaja 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();
```



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

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



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**.

```
//uporaba metode WriteLine na objektu Console
Console.WriteLine();
```

- 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*.

```
//uporaba metode na objektu iz razreda Random
Random naklj = new Random();
int nakljucnoStevilo=naklj.Next(1, 100);
```

- Objekt tipa **StreamReader** predstavlja *vhodni kanal*. Kadar želimo brati s tipkovnice ali z datoteke, naredimo tak objekt in kličemo njegovo metodo **ReadLine** za branje ene vrstice.

```
//kreiranje objekta StreamReader
string datoteka = "Vaja.txt";
StreamReader branje = new StreamReader(datoteka);
string vrstica = branje.ReadLine();
```

Naučili smo se torej uporabnosti objektnega programiranja, nismo pa se še naučili izdelave svojih razredov. 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.

Seveda pa nismo omejeni 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.

Razred

Razred (**class**) je abstraktna definicija objekta (torej opis, kako je naša škatla videti znotraj). Veliko razredov je že vgrajenih (so v standardnih knjižnicah) v C# in njegovo okolje, pogosto pa je za naše aplikacije potrebno napisati tudi kak svoj razred. Z razredom opišemo, kako je neka vrsta objektov videti. Če na primer sestavljamo razred zajec, opisujemo, katere so tiste lastnosti, ki določajo vse zajce.

V razredu zapišemo lastnosti, stanja in obnašanje objektov: zapišemo torej katere **podatke** bomo hranili v objektih te vrste in katere **metode** oz. odzive objektov na sporočila. Stanje objekta torej opišemo s spremenljivkami (rečemo jim tudi **polja ali komponente**), njihovo obnašanje oz. odzive pa z **metodami**.

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

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

Objekt je računalniški model predmeta ali stvari iz vsakdanjega življenja (avto, oseba, datum…). Objekt je kakršenkoli skupek podatkov, s katerimi želimo upravljati. Osnovni pristop objektnega programiranja je torej:

objekt = podatki + metode za delo s podatki.

Objekt je "črna škatla“, ki sprejema in pošilja sporočila. Jedro objekta sestavljajo njegove spremenljivke, okrog katerih se nahajajo njegove metode.

Klasično/objektno programiranje

Pojasnimo sedaj še razliko med klasičnim in objektnim programiranjem. Vsak program je sestavljen iz zaporedja stavkov, v katerih uporabljamo spremenljivke in metode. Metode, ki jih uporabljamo, so bodisi naše lastne metode (napisali smo jih sami) ali pa že obstoječe metode (npr. metode »knjižnjice« *Math*, kot napr. *Sqrt*, *Pow*, *Round*, …).

Ko želimo kak podatek obdelati v klasičnem programiranju pokličemo ustrezno metodo z argumenti, ki naj jih ta metoda obdela. Če smo npr. napisali metodo *Vsota*, ki vrne celo število, ima pa dva celoštevilska parametra, bomo to metodo poklicali npr. takole:

```
int skupaj=Vsota(23,45);
```

Pri objektnem programiranju pa je zadeva nekoliko drugačna. Ker metode pripadajo objektom (ki jih izpeljemo oz. ustvarimo iz razredov), moramo pred imenom metode napisati še ime objekta, ki mu ta metoda pripada. Recimo, da smo napisali razred *Ulomek*, znotraj tega razreda pa smo napisali metodo *Vsota*. Iz razreda *Ulomek* smo nato ustvarili objekt *U1*. Metodo vsota objekta *U1* pokličemo sedaj takole:

```
int skupaj=U1.Vsota(23,45); //pred imenom metode zapišemo še ime objekta
```

Oglejmo si sedaj 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, o pojmu ***dostopnost*** (**enkapsulacija**). 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*). Več o enkapsulaciji, torej pomenu besed *public*, *private* in *protected*, bo napisano v nadaljevanju.

Ustvarjanje objektov

V prejšnjem primeru smo iz razreda *MojR* tvorili objekt *prvi*. Pravimo, da smo deklarirali nov objekt razreda *mojR* z imenom *prvi*.

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

Nov objekt v pomnilniku ustvarimo z ukazom **new**. Brez besedice *new* objekta še **NI**. Naredimo sedaj npr. objekt tipa *MojR*, po pravilih, ki so določena z opisom razreda *MojR*. Novo ustvarjeni objekt se nahaja na naslovu *prvi*. Pravimo, da smo objekt *inicializirali*, oz. da smo objekt *definirali* (naslovu objekta smo priredili vrednost).

```
//z naslednjim ukazom bomo v pomnilniku naredili objekt tipa MojR, po pravilih, ki so določena z opisom razreda MojR. Novo ustvarjeni objekt se bo nahajal na naslovu prvi.
prvi = new MojR("Pozdravljen, moj prvi objekt v C#!");
```

Deklaracijo objekta in njegovo definicijo (prirejanje) lahko seveda združimo v en sam stavek takole:

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

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;
```

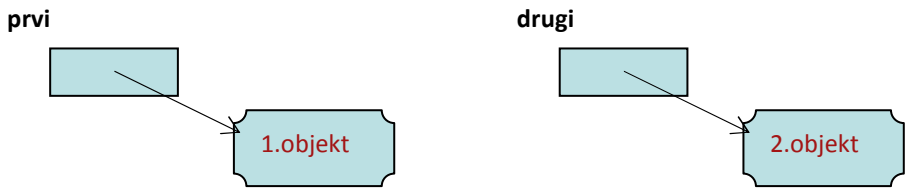
nimamo še nobenega objekta. Imamo le dve spremenljivki, v kateri lahko shranimo naslov, kjer bo operator *new* ustvaril nov objekt. Rečemo tudi, da sta spremenljivki *prvi* in *drugi* kazali na objekta tipa *MojR*. Če potem napišemo

```
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");
```

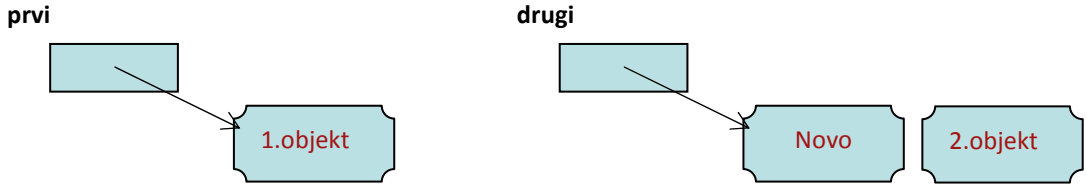
Poglejmo, kakšno je sedaj stanje v pomnilniku. S puščico v spremenljivkah *prvi* in *drugi* smo označili ustrezen naslov objektov. Dejansko je na tistem mestu napisano nekaj v stilu *h002a22* (torej naslov mesta v pomnilniku)



In če sedaj napišemo

```
drugi = new MojR("Novo");
```

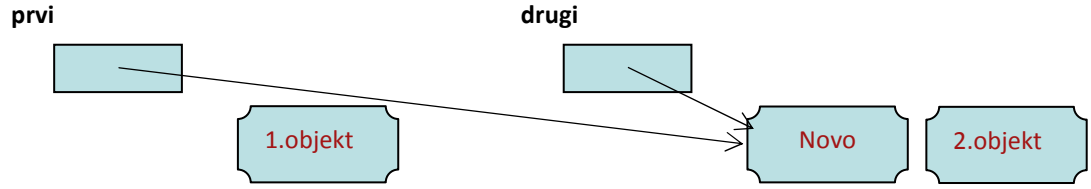
je vse v redu. Le do objekta z vsebino "2. objekt" ne moremo več! Stanje v pomnilniku je



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 (**garbage collector**) je poseben program v C#, za katerega delovanje nam ni potrebno skrbeti in ki poskrbi, da se po pomnilniku ne nabere preveč objektov, katerih naslov ne pozna nihče več.

Primer:
Recimo, da smo napiali razred *Ulomek*. Z naslednjim stavkom bomo iz tega razreda izpeljali objekt *ime*.

```
Ulomek ime = new Ulomek(3, 4);
```

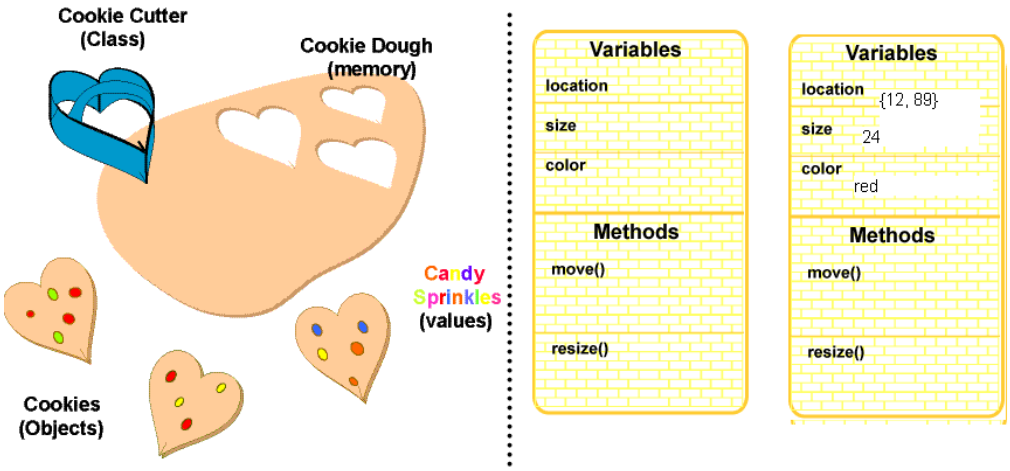
Rečemo: V objektu *ime* je shranjen ulomek ¾ oz. točneje: v spremenljivki *ime* je naslov nekega objekta tipa *Ulomek*, ki predstavlja ulomek ¾. Spremenljivka ime kaže na objekt tipa *Ulomek*, ki …

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.

Pa še enkrat: razred (*class*) je opis vrste objekta (načrt, kako naj bo objekt videti) – opis ideje objekta. Primerek razreda (instanca) pa je konkretni objekt.



Zgledi

Ulomek

Denimo, da bi radi napisali program, ki bo prebral nek ulomek in mu prištel 1/2. "Klasično" se bomo tega lotili takole:

```
static void Main(string[] args)
{
    Console.Write("Števec ulomka: ");
    string beri = Console.ReadLine();
    int stevec = int.Parse(beri);
    Console.Write("Imenovalec ulomka: ");
    beri = Console.ReadLine();
    int imenovalec = int.Parse(beri);
    //ulomku prištejmo 1/2. Ulomka seveda seštejemo po pravilu za
    //seštevanje ulomkov: a/b + c/d =(a*d + c*b)/(b*d)
    stevec = stevec * 2 + imenovalec * 1;
    imenovalec = imenovalec * 2;
    // izpis
    Console.WriteLine("Nov ulomek je: " + stevec + " / " + imenovalec);
    Console.ReadKey();
}
```

Sedaj pa to naredimo še "objektno". Očitno bomo potrebovali razred *Ulomek*. Objekt tipa *Ulomek* hrani podatke o svojem števcu in imenovalcu. Njegovo znanje je, da "se zna" povečati za drug ulomek. To pomeni, da vsebuje metodo, ki ta ulomek poveča za drug ulomek. Celotno kodo razreda *Ulomek* seveda zaenkrat še ne bomo razumeli, a pomembno je to, da dobimo občutek, kaj pravzaprav pojem razred predstavlja.

```
class Ulomek
{   // razred Ulomek naj vsebuje dve polji: stevec in imenovalec
    public int stevec;
    public int imenovalec;

    public Ulomek(int st, int im)  // konstruktor
    {
        this.stevec = st;
        this.imenovalec = im;
    }

    public void Pristej(Ulomek a)//metoda, ki ulomek poveča za drug ulomek
    {
        this.stevec=this.stevec* a.imenovalec + this.imenovalec * a.stevec;
        this.imenovalec = this.imenovalec * a.imenovalec;
    }
}
static void Main(string[] args)
{
    Console.Write("Števec ulomka: ");
    string beri = Console.ReadLine();
    int stevec = int.Parse(beri);
    Console.Write("Imenovalec ulomka: ");
    beri = Console.ReadLine();
    int imenovalec = int.Parse(beri);
    Ulomek moj = new Ulomek(stevec, imenovalec); // NAREDIMO ulomek
    // "delo"
    Ulomek polovica = new Ulomek(1, 2);
    moj.Pristej(polovica); // UKAZUJEMO ulomku
    // izpis
```

```
    Console.WriteLine("Nov ulomek je: " + moj.stevec + " / " +
                      moj.imenovalec);
    Console.ReadKey();
}
```

Krog

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
}
```

Zgradba

Napišimo razred *Zgradba*, ki naj vsebuje dve polji: *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;
```

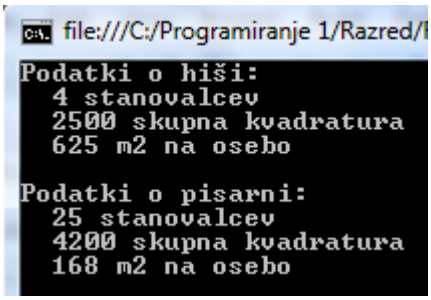
Če imamo torej nek razred *ImeRazreda* in v njem polje *nekoPolje* in je objekt *mojObjekt* tipa *ImeRazreda*, z

```
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:



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.

Evidenca članov kluba

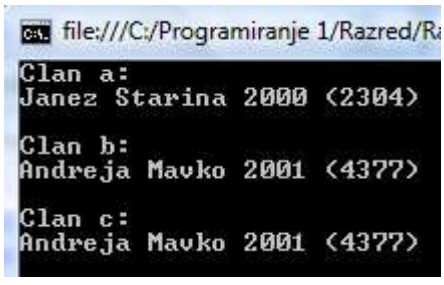
Napišimo program, ki vodi evidenco o članih športnega kluba. Podatki o članu obsegajo ime, priimek, letnico vpisa v klub in vpisno številke (seveda je to poenostavljen primer). Torej objekt, ki predstavlja člana kluba, vsebuje štiri podatke. Ustrezni razred je:

```
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:



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*)!

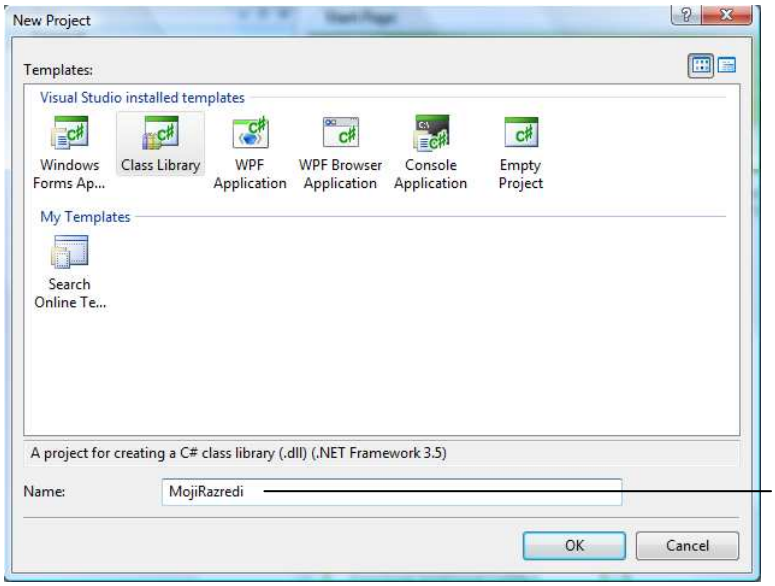
V vrsticah 4 – 7 smo naredili objekt *a* (naredili smo objekt in nanj pokazali z *a*) in nastavili vrednosti njegovih komponent. Takole nastavljanje je v praksi dokaj neprimerno, ker se zlahka zgodi, da kako komponento pozabimo nastaviti. Zato C# omogoča, da delamo nove objekte na bolj praktičen način s pomočjo **konstruktorjev**, o katerih pa bomo govorili nekolik kasneje.

Knjižnjice razredov in datoteka z razredi

V dosedanjih zgledih do smo razrede pisali le za "lokalno uporabo", znotraj določenega programa. Razred (ali pa več razredov) pa lahko zapišemo tudi v svojo datoteko, ki jo potem dodajamo k različnim projketom, ali pa celo zgradimo svojo **knjižnjico razredov**, ki jih bomo uporabljali v različnih programih. Na ta način bomo tudi bolj ločili tisti del programiranja, ko gradimo razrede in tisti del, ko uporabljamo objekte določenega razreda.

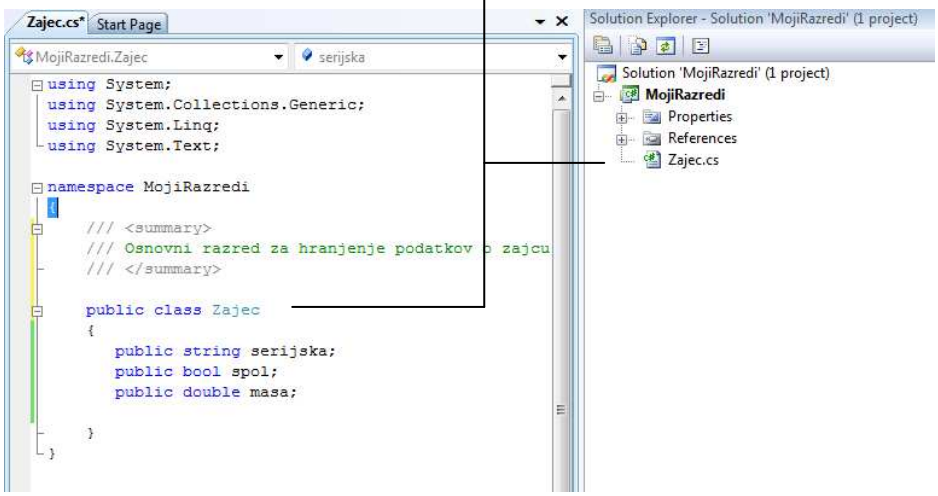
Ustvarjanje nove knjižnjice razredov

Naučimo se najprej, kako zgradimo svojo knjižnjico razredov: v okolju Visual C# tokrat ne izberemo Windows Applications/Console Application, ampak **Class Library**



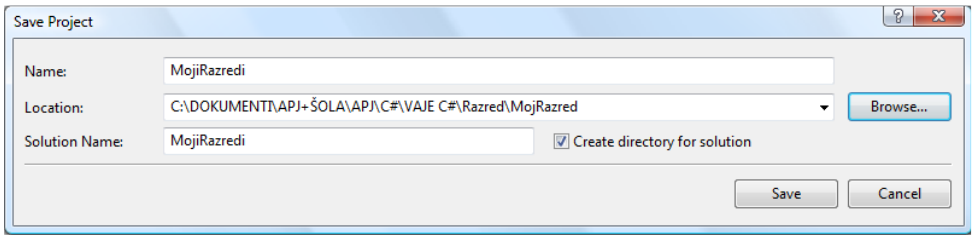
Kot ime knjižnjice bomo napisali npr. *MojiRazredi*. V to knjižnjico bomo kasneje lahko dodajali vse tiste razrede, ki jih bomo ustvarjali.

S klikom na gumb OK potrdimo našo izbiro. Vidimo, da je Visual C# za nas pripravil okolje in naredil tako imenovam imenski prostor *MojiRazredi*. Znotraj tega imenskega prostora bomo zlagali naše razrede. Ker nam ime *MyClass* ni všeč, bomo to spremenili v *Zajec*.



Dodajmo še komentar, ki naj se začne z ///. To so tako imenovani dokumentacijski komentarji. Ti služijo za to, da pripravimo dokumentacijo o razredu. Ta je zelo pomembna, saj nosi tisto informacijo, ki jo kasneje potrebujemo, če želimo razred uporabljati. Zaenkrat si zapomnimo le, da na ustrezno mesto napišemo kratek povzetek o tem, čemu je razred namenjen.
Napisali smo torej zelo poenostavljen opis, kako je določen poljuben zajec. Zavedati pa se moramo, da gre v bistvu le za načrt, kakšni so zajci, ne pa za konkretnega zajca. Opazimo tudi, da v knjižnjici, ki jo pišemo, ni metode *Main*. Knjižnjica namreč ni namenjena izvajanju oz. poganjanju tako kot smo bili navajeni pri dosedanjih programih, ampak je namenjena uporabi v drugih programih, ki jih bomo pisali kasneje.

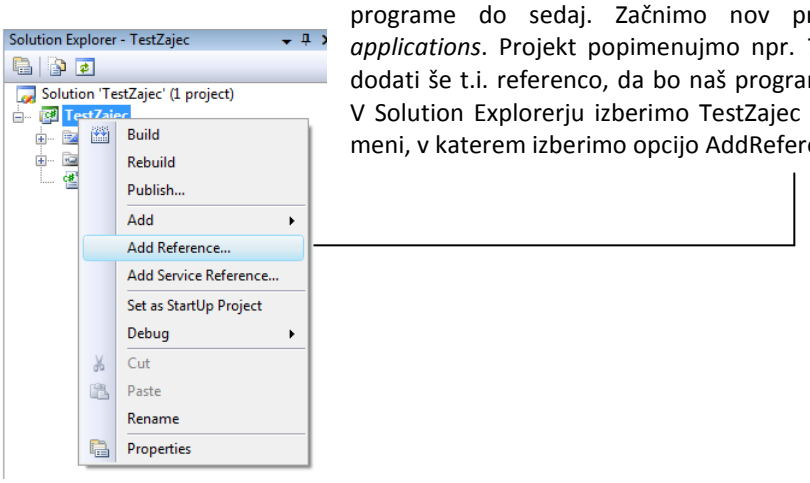
Preden nadaljujemo knjižnjico shranimo: v meniju *File* izberimo opcijo *Save All*. Odpre se pogovorno okno za shranjevanje: s klikom na gumb *Browse* izberimo še mapo, v katero bomo našo knjižnjico shranili (v našem primeru je ime mape *Moj Razred*)



Knjižnjico je sedaj potrebno še prevesti. V meniju *Build* izberimo opcijo *Build Solution* (ali pa stisnimo tipko *F6*) in naša knjižnjica je sedaj pripravljena za uporabo. Ustvarili smo namreč ustrezno prevedeno obliko te knjižnjice, ki je dobila ime *MojiRazredi.dll* (*dll* je kratica za *dinamic link library*). Knjižnjica zaenkrat vsebuje le definicijo razreda *Zajec*. Datoteka se nahaja v mapi ..\ *MojRazred**MojiRazredi**MojiRazredi**bin**Debug*.

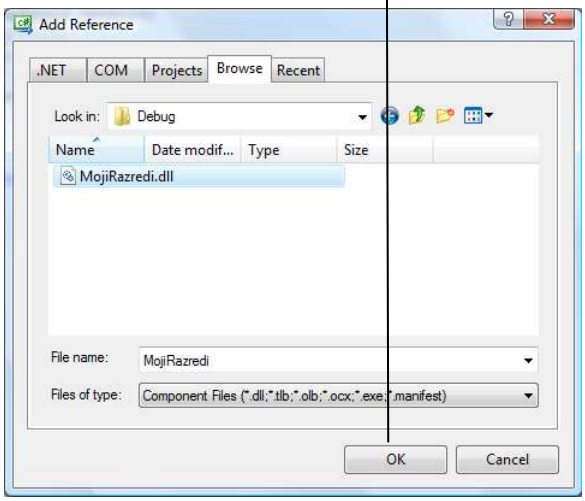
Uporaba knjižnjice razredov

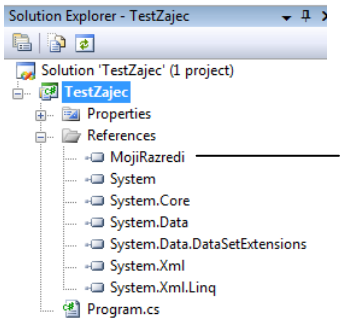
Denimo, da sedaj pišemo nek program, kjer bomo potrebovali Zajce. V ta namen sedaj sestavimo program v C#, ki bo med drugim uporabljal že obstoječo knjižnjico z imenom *MojiRazredi*. Obnašamo se tako kot smo pisali programe do sedaj. Začnimo nov projekt : *File/New Project/Console applications*. Projekt popimenujmo npr. *TestZajec*. V projekt moramo sedaj dodati še t.i. referenco, da bo naš program prepoznal naš razred *MojiRazredi*. V Solution Explorerju izberimo TestZajec in z desnim klikom odprimo lebdeči meni, v katerem izberimo opcijo AddReference



Odpre se okno za izbiro ustrezne reference: izberimo najprej jeziček *Browse* in nato poiščimo datoteko *MojiRazredi.dll*.

Izbiro potrdimo s klikom na gumb OK





Opazimo, da se v *Solution Explorerju* pojavi nova referenca *MojiRazredi*, kar pomeni, da imamo sedaj dostop do vseh razredov knjižnjice *MojiRazredi*.

Da bo dostop do naših razredov še lažji, na začetku datoteke dopišimo še en *using* stavek:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using MojiRazredi; //NOVO dodana knjižnjica
```

Nov objekt razreda *Zajec*, ki je definiran znotraj knjižnjice *MojiRazredi* lahko sedaj naredimo takole:

```
Zajec rjavko = new Zajec();
```

V spremenljivki *rjavko* je v bistvu naslov, kje je novo ustvarjeni zajec (objekt), a kljub temu navadno rečemo, da smo ustvarili objekt *rjavko*, ki smo ga izpeljali iz razreda *Zajec* (ali še drugače: naredili smo novo ***instanco*** razreda *Zajec*). Ustvarili smo torej konkretnega zajca po navodilih za razred *Zajec*. Ta zajec (*rjavko*) ima torej tri podatke / lastnosti / komponente), to pa so spol, serijska številka in masa. Te lastnosti lahko sedaj določimo poljubno, npr. takole:

```
rjavko.spol = true;
rjavko.serijska = "BRGH_17_A";
rjavko.masa = 3.2;
```

S tako ustvarjenimi spremenljivkami lahko sedaj počnemo povsem enako kot z običajnimi spremnljivkami. Še primer celotne vsebine datoteke *TestZajec*:

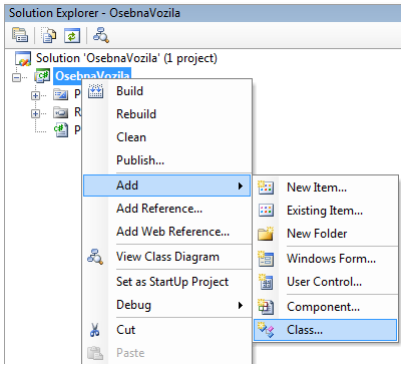
```
...
using MojiRazredi;  //NOVO dodana knjižnjica
```

```
namespace TestZajec
{
  class Program
  {
    static void Main(string[] args)
    {
       Zajec rjavko = new Zajec();//ustvarimo novo instanco razreda Zajec
       rjavko.serijska = "1238-12-0";
       rjavko.spol = false;
       rjavko.masa = 0.12;
       rjavko.masa = rjavko.masa + 0.3; //maso zajca lahko  povečamo

       //posamezna polja objekta rjavko lahko izpisujemo
       Console.WriteLine("Zajec ima ser. št.:" + rjavko.serijska);
       Console.ReadKey();
    }
  }
}
```

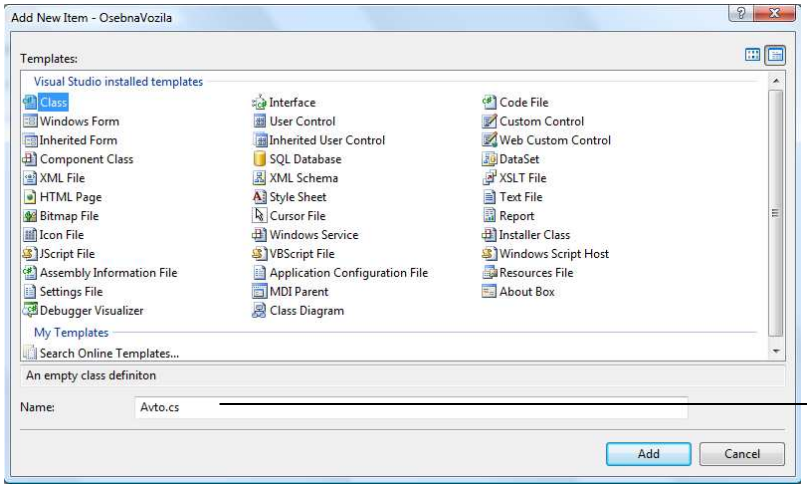
Datoteka z razredom (ali več razredi)

Namesto knjižnjice razredov pa lahko razred (ali pa skupino razredov) zapišemo tudi v posebno datoteko kar znotraj projekta, v bodočih projektih pa to datoteko samo dodamo v projekt.

Začnimo nov projekt (nova konzolna aplikacija) in ga poimenujmo *OsebnaVozila*. Kreirajmo sedaj razred *avto* s štirimi zasebnimi polji (*znamka*, *model*, *najvecjahitrost* in *teza*), ter javno metodo za izpis polj tega razreda. Tej metodi bomo dali ime *IzpisPodatkov*. Razred bomo zapisali v svojo datoteko, ki jo odpremo takole: v *SolutionExplorerju* izberimo vrstico *OsebnaVozila*, nato pa z desnim klikom miške odpremo lebdeči meni. Izberimo opcijo *Add* in nato *Class*.



V oknu, ki se odpre, se prepričajmo, če je izbrana opcija *Class* in vnesimo ime našega razreda – *Avto.cs*.



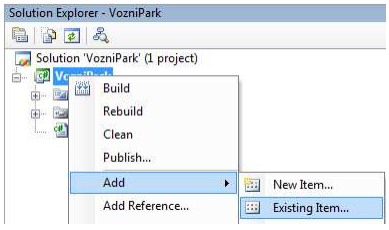
Vse skupaj potrdimo s klikom na gumb *Add*. V *Solution Explorerju* bomo opazili novo vrstico z imenom *Avto*, v urejevalniku pa bo prikazana začetna vsebina datoteke (razreda) *Avto.cs*. Vsebino dopolnimo, tako da je izgled celotne datoteke takle:

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

namespace OsebnaVozila
{
    class Avto
    {
        //polja razreda Avto
        public string znamka;
        public string model;
        public int najvecjahitrost;
        public double teza;
        //javna metoda za izpis podatkov o konkretnem avtomobilu. Več o
        //metodah v razredih bo napisano v nadaljevanju
        public string IzpisPodatkov()
        {
            return ("\nIzpis podatkov o vozilu:\nZnamka: "
                                + znamka
                                + "\nModel: " + model
                                + "\nNajvečja hitrost: " + najvecjahitrost
                                + "\nteža vozila: " + teza);
        }
    }
}
```

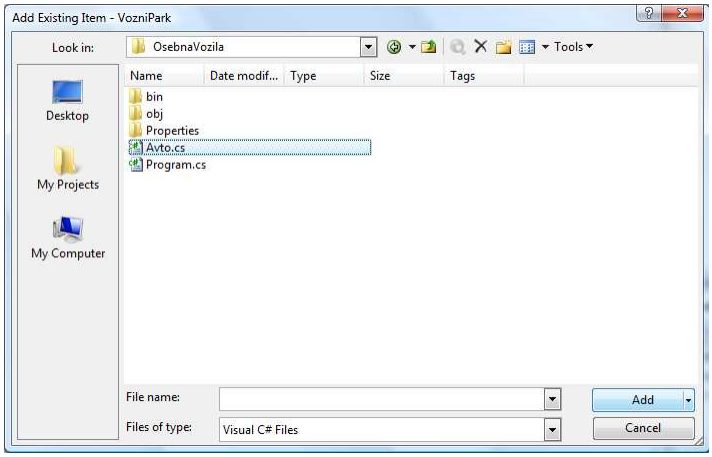
Datoteko shranimo in sedaj ponovno odprimo datoteko z "glavnim" programom (v *Solution Explorerju* dvoklinimo vrstico *Program.cs*). Deklarirajmo nov objekt *moj*, mu določimo začetne vrednosti in pokličimo metodo *IzpisPodatkov*:

```
static void Main(string[] args)
{
    Avto moj = new Avto();//deklaracija novega objekta moj
    //poskrbimo za inicializacijo
    moj.znamka = "Citroen";
    moj.model = "C4 Picasso";
    moj.najvecjahitrost = 185;
    moj.teza = 1850;
    //klic metode IzpisPodatkov, ki jo objekt moj seveda pozna, saj smo ga
    //izpeljali iz razreda Avto, v katerem je ta metoda definirana
    Console.WriteLine(moj.IzpisPodatkov());
}
```

Razred *Avto* smo torej napisali v svoji datoteki in ga nato uporabili v projektu, znotraj katerega smo ta razed tudi kreirali. Sedaj pa moramo še pokazati, kako bi ta razred uporabili v nekem drugem, novem proketu. Postopek je naslednji: ustvarimo nov projekt in ga poimenujmo npr. *VozniPark*. Ker želimo v tem projektu uporabiti razred, ki so ga v prejšnjem projektu zapisali v datoteko *Avto.cs*, ga lahko v naš novi projekt vključimo takole: v *Solution Explorerju* izberemo vrstico *VozniPark*, kliknemo desni miškin gumb in izberimo opcijo *Add*, ter nato *Existing Item*...



Odpre se pogovorno okno, s pomočjo katerega sedaj poiščimo datoteko *Avto.cs*, ki se nahaja znotraj prejšnjega projekta, ki smo ga poimenovali *OsebnaVozila*. Ko jo najdemo, izbiro potrdimo s klikom na gumb *Add*.



V *Solution Explorerju* se ojavi nova vrstica *Avto.cs* (razred *Avto* smo dodali v naš novi projekt). Ker smo razred Avto napisali v projektu Osebna vozila, se nanj sklicujemo takole:

```
static void Main(string[] args)
{
   OsebnaVozila.Avto sosedov = new OsebnaVozila.Avto();
   //...
}
```

Z objektom *sosedov* sedaj delamo popolnoma enakovredno kakor v prejšnjem projektu *OsebnaVozila*, v katerem smo razred *Avto* tudi definirali.

Do razreda *Avto* smo torej prišli preko imenskega prostora *OsebnaVozila*. Če pa ta imenski prostor z *using* stavkom dodamo na začeku programa tako, da napišemo

```
using System;
using System.Collections.Generic;
using System.Text;
using OsebnaVozila; //dodamo imenski prostor
```

potem pa lahko nov objekt razreda *Avto* napovemo takole:

```
static void Main(string[] args)
{
   Avto sosedov = new Avto();
   //...
}
```

Pokazali smo torej, kako kreiramo svojo lastno knjižnjico razredov (dll), ki jo potem dodamo kot referenco k vsem nadaljnjim projektom, prav tako pa smo pokazali kako ustvarimo datoteko z razredom in jo dodajamo k novim projektom. ***Samo zaradi enostavnosti in lažjega razumevanja, pa bomo v nadaljnji razlagi, primerih in zgledih, razrede pisali kar znotraj določenega programa***.

Povzetek

Najenostavnejši razred kreiramo tako, da določimo spremenljivke (polja), ki določajo objekte tega razreda. Te so lahko različnih podatkovnih tipov. Sintaksa najenostavnejše definicije razreda je naslednja:

```
public class ImeRazreda
{
   public podatkovni tip element1;
   public podatkovni tip element2;
      …
   public podatkovni tip elementn;
}
```

Če želimo tak razred uporabljati v nekem programu, moramo v programu dodati referenco na ustrezno prevedeno obliko (*dll*) tega razreda, ali pa datoteko s tem razredom dodati v program.

Objekt tega razreda ustvarimo z `new ImeRazreda()`. S tem dobimo naslov, kje ta objekt je in ga shranimo v spremenljivko tipa *ImeRazreda*:

```
ImeRazreda imeObjekta = new ImeRazreda();
```

Do posameznih spremenljivk (rečemo jim tudi komponente ali pa lastnosti), dostopamo z

```
imeObjekta.imeKomponente;
```

Objektne metode – funkcije razedov

Svoj pravi pomen dobi sestavljanje razredov takrat, ko objektu pridružimo še metode, torej znanje nekega objekta. Kot že vemo iz uporabe v C# vgrajenih razredov, metode nad nekim objektom kličemo tako, da navedemo ime objekta, piko in ime metode. Tako če želimo izvesti metodo *WriteLine* nad objektom *System.Console* napišemo

```
System.Console.WriteLine("To naj se izpiše");
```

Če želimo izvesti metodo *Equals* nad nizom besedilo, napišemo

```
besedilo.Equals(primerjava);
```

in tako dalje.

Kako napišemo svojo lastno metodo znotraj razreda

Metodo znotraj poljubnega razreda napišemo tako kot vsako metodo: napišemo njeno glavo (dostopnost, tip rezultata, ime in parametre), ter telo metode.

Primer:

Razred *Clan*, ki smo ga napisali v prejšnjem poglavju, opremimo še z dvema metodama: metodo *Izpis*, ki izpiše podatke o članu, ter metodo *Opis*, ki vrne podatke o članu v obliki stringa.

```
public class Clan
{
    //Najprej lastnosti (polja) razreda Clan
    public string ime;
    public string priimek;
    public int leto_vpisa;
    public string vpisna_st;

    //Metoda, ki izpiše podatke o članu
```

```
    public void Izpis()
    {
        Console.WriteLine("Clan:\n" + this.ime + " " + priimek + " " +
                          leto_vpisa + " (" + vpisna_st + ")\n");
    }
    //Metoda, ki vrne podatke o članu v obliki stringa
    public string Opis()
    {
        return this.ime + " " + priimek + " " +  leto_vpisa + " (" +
        vpisna_st +")";
    }

}
public static void Main(string[] args)
{
  //Ustvarimo novega člana: nov objekt razreda Clan se imenuje c1
  Clan c1 = new Clan();
  //Članu c1 določimo polja
  c1.ime= "Janez";
  c1.priimek="Starina";
  c1.leto_vpisa=2000;
  c1.vpisna_st="2304";
  //S pomočjo metode Izpis objekta c1 izpišimo podatke o članu c1
  Console.WriteLine("Clan c1"); c1.Izpis();
  //V spremenljivo opisClana shranimo opis člana c1, ki ga dobimo s pomočjo
  //metode Opis objekta c1
  string opisClana = c1.Opis();
  Console.WriteLine("Clan c1:\n" + opisClana); //Izpis
  Console.ReadLine();
}
```

Seveda lahko v obstoječi razred kadarkoli dodajamo še nove metode, ali pa spremenimo (popravimo) obstoječe metode. Ko razred (ali pa projekt) ponovno prevedemo, imamo že vse pripravljeno za "pravilno" izvajanje vseh tistih programov, ki uporabljajo ta razred. Brez kakršnekoli spremembe v uporabniškem programu (in tudi brez ponovnega prevajanja tega programa), metoda sedaj deluje v skladu z v razredu narejenimi spremembami. Če torej opazimo, da je potrebno v razredu narediti kakšen popravek, le vsem našim uporabnikom pošljemo novo različico prevedenega razreda (ali knjižnice, ki vsebuje ta razred). Brez kakršnihkoli sprememb na strani uporabnika njihovi programi delujejo "popravljeno".

Gostota prebivalstva

Razred *Drzava* naj vsebuje polja za osnovne podatke o državi, ter metodo *Gostota* za izračun gostote prebivalstva. Kreirajmo objekt in pokažimo uporabo njegove metode *Gostota*.

```
class Drzava
{
  public string ime;
  public int povrsina;
  public int prebivalci;
  public double Gostota() // javna metoda za izračun gostote prebivalstva
  {
    return prebivalci/povrsina;  // vrnemo celo število!
  }
}
static void Main(string[] args)
{
  Drzava d1=new Drzava();// d1 je nov objekt razreda Drzava
  d1.ime = "Slovenija";
```

```
  d1.povrsina = 20000;
  d1.prebivalci = 1980000;
  // pokličimo metodo Gostota objekta d1
  Console.WriteLine("Gostota prebivalstva v " + d1.ime + " je " +
                    d1.Gostota()+ " preb/km2");
}
```

Polinom druge stopnje

Kreirajmo razred *Polinom*, ki hrani polinome do stopnje dva ($ax^2 + bx + c$). Polinom naj ima tri polja (koeficiente polinoma), pozna pa naj tudi tri metode: metodo za seštevanje dveh polinomov, metodo, ki vrne vrednost polinoma v poljubni točki in metodo *ToString*, ki ta polinom predstavi (vrne) v obliki stringa.

```
public class Polinom
{
   public double a, b, c;//koeficienti polinoma
   //Metoda za seštevanje dveh polinomov
   public Polinom Sestej(Polinom p)
   {
      Polinom zacasni=new Polinom();
      zacasni.a=a + p.a;
      zacasni.b=b + p.b;
      zacasni.c=c + p.c;
      return zacasni;
   }
   //metoda ki izračuna in vrne vrednost polinoma v točki x
   public double vrednostPolinoma(double x)
   {
       return a*x*x+b*x+c;
   }
   //metoda za izpis polinoma v obliki a*x2 + b*x + c
   public string ToString()
   {
       return a + "*x2 + " + b + "*x+ " + c;
   }
}
public static void Main(string[] args)
{
   Polinom P1=new Polinom(); //Prvi polinom
   P1.a = 2;
   P1.b = 3;
   P1.c = 5;
   Console.WriteLine("Prvi polinom: "+P1.ToString());//Izpis polinoma št.1

   Polinom P2 = new Polinom();//Drugi polinom
   P2.a = 1;
   P2.b = 5;
   P2.c = 7;
   Console.WriteLine("Drug polinom: "+P2.ToString());//Izpis polinoma št.2
   //S pomočjo metode Sestej objekta P1 polinomu P1 prištejmo polinom P2 in
   // rezultat izpišimo s pomočjo metode ToString
   Console.WriteLine("Vsota polinomov: " + P1.Sestej(P2).ToString());
   Console.ReadLine();
}
```

Kvader

Kreirajmo razred *Kvader*, ki naj vsebuje tri polja (robove kvadra) in metode za izračun telesne diagonale, površine in prostornine kvadra.

```
public class Kvader
{
    public double a, b, c;
    //Metoda izračuna in vrne telesno diagonala kovadra
    public double TelesnaDiagonala()
    {
       return Math.Sqrt(a*a+b*b+c*c);
    }
    //Metoda izračuna in vrne površino kvadra
    public double Povrsina()
    {
        return 2 * a * b + 2 * a * c + 2 * b * c;
    }
    //Metoda izračuna in vrne prostornino kvadra
    public double Prostornina()
    {
        return a * b * c;
    }
}
static void Main(string[] args)
{
    Kvader K1=new Kvader();//ustvarimo nov objekt K1 tipa Kvader in ga
    K1.a=1;
    K1.b=2;
    K1.c=3;
    //Klic objektne metode TelesnaDiagonala
    Console.WriteLine("Telesna diagonala kvadra: "+K1.TelesnaDiagonala());
    //Klic objektne metode Povrsina
    Console.WriteLine("Površina kvadra: " + K1.Povrsina());
    //Klic objektne metode Prostornina
    Console.WriteLine("Prostornina kvadra: " + K1.Prostornina());
    Console.ReadLine();
}
```

Razred *MojString*

Napišimo razred *MojString*, v katerem bomo napisali nekaj javnih metod za delo s stringi, ki naj pomenijo alternativo obstoječim metodam – npr. *Length*, *Replace*, *ToUpper*,…

```
class MojString
{
    private string stavek; //zasebno polje razreda
    public void DolociStavek(string st) //metoda za inicializacijo polja
                                        //stavek
    {
        stavek = st;
    }
    public int dolzina()//metoda ki vrne število znakov v stringu
    {
        return stavek.Length;
    }
    //metoda za zamenjavo določenih znakov v stringu z novimi znaki
    public void ZamenjajZnak(string stari, string novi)
```

```
    {
        stavek = stavek.Replace(stari, novi);
    }
    public string IzpisStavka()
    {
        return stavek;
    }
    public void VelikeCrke()
    {
        stavek = stavek.ToUpper();
    }
}

static void Main(string[] args)
{
    MojString st=new MojString ();
    st.DolociStavek("Razred MojString: metodam za delo s stringi smo
                    priredili slovenska imena!");
    Console.WriteLine("\nV stavku:\n\n"+st.IzpisStavka()+"\n\nje
                     "+st.dolzina()+" znakov!");
    st.ZamenjajZnak(" ", "X");//presledke nadomestimo z znakom X
    Console.WriteLine(st.IzpisStavka());
    st.ZamenjajZnak("X"," ");//vse znake "X" nadomestimo s presledki
    st.VelikeCrke();  //vse črke v stringu spremenimo v velike črke
    Console.WriteLine(st.IzpisStavka());
}
```

Podpisi metod

Pri pisanju metod ("navadnih" in objektnih) se moramo držati pravila, da se morajo razlikovati ne v imenu, ampak **podpisu**. Podpis metode je sestavljen iz imena metode in tipov vseh parametrov. Tip rezultata neke metode **NI** del podpisa.

Primeri različnih podpisov metod:

- public static int **NekaMetoda**(**double x**)
- public static int **NekaMetoda**()
- public static int **NekaMetoda**(**int x**)
- public static int **NekaMetoda**(**double x,int y**)
- public static double **NekaMetoda**(**int x, int y**)
- public static int **nekaDrugaMetoda**(**double y**)

Preobtežene metode

Tako "navadne", kot tudi objektne metode so lahko ***preobtežene*** (uporablja se tudi izraz ***preobložene***). Preobtežene metode so metode, ki imajo enako ime, razlikujejo pa se v številu ali pa tipu parametrov. Preobteženost torej označuje možnost, da imamo lahko več enako poimenovanih metod, ki pa sprejemajo parametre različnega tipa.

Zakaj je to uporabno? Denimo, da želimo napisati 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 pa problem rešimo enostavno. Napišemo tri metode: ker gre v naslednjih primerih za "navadne" oz. "statične" metode smo dodali še besedico *static* (pravi pomen te besede bomo spoznali v poglavju o statičnih metodah).

```
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 forumule. 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 v programu nastopa npr. nek lik *likX*, ki je bodisi tipa *Kvadrat*, tipa *Krog* ali pa tipa *Pravokotnik*, uporabnik metodo pokliče z *Ploscina(likX)*, 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 pa je še vedno *Ploscina(likX)*!

Konstruktor

V vseh dosedanjih primerih smo najprej ustvarili nov objekt, nato pa mu nastavili začetne vrednosti polj. Tako 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**. **Konstruktor** je metoda, ki jo pokličemo ob tvorbi objekta z *new*. Je brez tipa rezultata. Ne smemo pa ga 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 enako, 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 Zajec()*. Uporabljamo jih za vzpostavitev začetnega stanja objekta.

Primer:

Napišimo konstruktor za razred *Zajec*:

```
public class Zajec
{
  public string serijska;
  public bool spol;
  public double masa;

  //konstruktor: NIMA tipa, njegovo ime pa mora biti enako kot je ime
  //razreda
```

```
  public Zajec()
  {
    this.spol = true; // vsem zajcem na začetku določimo moški spol
    this.masa = 1.0; // in tehtajo 1kg
    this.serijska = "NEDOLOČENO"; // serijska številka je še nedoločena
  }
}
```

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 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 besedico *this*.

Uporaba konstruktorja:

Kot smo povedali že večkrat, je razred le načrt, kako so videti objkekti določenega razreda. Če potrebujemo konkreten primer, v našem primeru Zajca, ga "ustvarimo" z *new*:

```
Zajec rjavko = new Zajec();
```

S tem se je ustvaril konkreten zajec (torej tak, ki ima tri spremenjlivke za hranjenje podatkov), nad katerim smo takoj potem izvedli določeno akcijo po navodilih iz konstruktorja *Zajec()*. Kombinacija *new Zajec()* je torej poskrbela, da ima objekt tipa *Zajec* tri podatke z vrednostmi, kot je predpisano v konstruktorju.

Privzeti konstruktor

Če konstruktorja ne napišemo (kot ga npr. nismo v začetnih primerih poglavja o razredih in objektih), ga "naredi" prevajalnik sam (a metoda ne naredi nič). Konstruktor, ki ga avtomatično generira prevajalnik, je vedno public, nima nobenega tipa (niti void), nima argumentov, vrednosti numeričnih polj postavi na 0, polja tipa *bool* postavi na *false*, vsem referenčnim poljem (spremenljivkam) pa priredi vrednost *null*.

Dokler je bil naš razred *Zajec* takle:

```
public class Zajec
{
  public string serijska;
  public bool spol;
  public double masa;
}
```

je prevajalnik sam dodal privzeti kostruktor.

```
public Zajec()
{
}
```

Kakor hitro pa smo konstruktor *Zajec()* napisali sami, se seveda prevajalnik ni več "vmešaval":

Zgled - 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 vrsta)
    {
        this.ulica = kateraUlica;
        this.stevilka = hisnaStevilka;
        this.vrsta = vrsta;
    }
}
```

```
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();
}
```

V zgornjem konstruktorju smo uporabili besdico *this*. Ta je v konstruktorju obvezna le tedaj, kadar so imena parametrov enaka imenom polj razreda. Če pa se imena parametrov razlikujejo od imen polj, besedica *this* seveda ni potrebna. Zgornji konstruktor bi torej lahko napisali tudi takole:

```
public Nepremicnina(string kateraUlica,int hisnaStevilka, string vrsta)
{
    ulica = kateraUlica;
    stevilka = hisnaStevilka;
    vrsta = vrsta;
}
```

Zgled: razred Denarnica

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 ima dva parametra
    public Denarnica(string ime, double znesek)
    {
        this.ime = ime;
        this.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);
}
```

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]; // vsi elementi tabele dobijo vrednost 0
   }
   // 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];//elementi tabele dobijo vrednost 0
```

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.

Sedaj pišemo program, v katerem bomo potrebovali 100 objektov tipa *Zajec* (denimo, da pišemo program za upravljanje farme zajcev; razred Zajec poznamo že iz prejšnjih primerov). Ker bomo z vsemi zajci (z vsemi objekti tipa *Zajec*) počeli enake operacije, je smiselno, da uporabimo tabelo.

```
Zajec[] tabZajci;
```

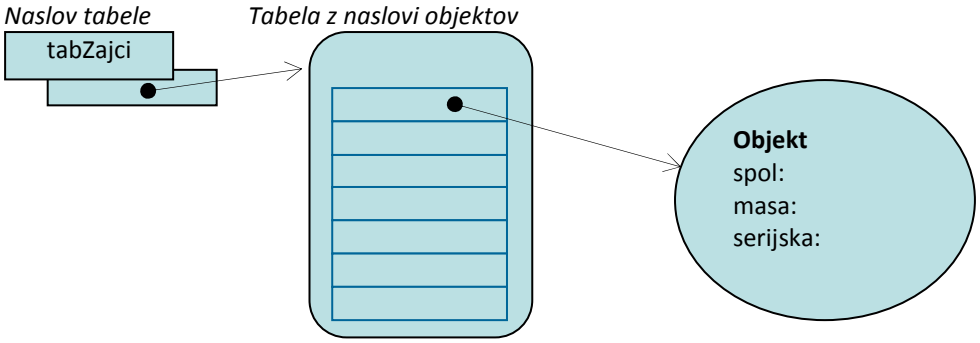
Spemenljivka *tabZajci* nam označuje torej tabelo, v kateri hranimo zajce. A bodimo tokrat še zadnjič pri izražanju zelo natančni. Natančno rečeno nam spremenljivka *tabZajci* označuje **naslov**, kjer bomo ustvarili tabelo, v kateri bomo hranili naslove objektov tipa *Zajec*. Ko torej napišemo

```
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]*.



Seveda pa bomo še vedno govorili, da imamo zajca shranjenega v spremenljivki *tabZajci[0]*. Zapišimo še celoten program za našo farmo zajcev, skupaj za razredom *Zajec*:

```
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++;
```

```
  }
}
```

Zgled - 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 (tabela objektov)
   // s konstruktorjem določimo avtorja, naslov in vse pesmi
   public Zgoscenka(string avtor, string naslov, Pesem[] seznamPesmi)
   {
     this.avtor = avtor;
     this.naslov = naslov;
     //inicializacija tabele pesmi
     this.seznamPesmi = new Pesem[seznamPesmi.Length];
     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);
   // izpiiemo celoten seznam (tabelo) pesmi
   for (int i = 0; i < CD.seznamPesmi.Length; i++)
      Console.WriteLine("Pesem št."+(i+1)+": "+CD.seznamPesmi[i].naslov);
  // pokličemo še metodo Dolzina objekta CD, ki vrne skupno dolžino vseh
  //skladb
  Console.WriteLine("Skupna dožina pesmi je "+CD.Dolzina()+" sekund!");
}
```

Zgled - Firma

Napišimo razred *Zaposleni*, ki zajema osnovne podatke o zaposlenem delavcu v neki firmi. Napišimo program, ki bo omogočal vnos podatkov v tabelo Firma (to ja tabela objektov, ki zajema vse zaposlene v podjetju), ter izpis te tabele.

```
public class Zaposleni
{
    public string imeInPriimek;
    public double dohodek;

    public Zaposleni(string ime, double letniDohodek) //konstruktur
    {
        imeInPriimek = ime;
        dohodek = letniDohodek;
    }
}

static void Main(string[] args)
{
    ////napoved tabele 5 objektov tipa Zaposleni
    Zaposleni[] Firma = new Zaposleni[5];
    for (int i = 0; i < Firma.Length; i++) //Vnos podatkov
    {
        Console.Write("Ime in priimek: ");
        string ime=Console.ReadLine();
        Console.Write("Dohodek: ");
        double dohodek=double.Parse(Console.ReadLine());
        Firma[i]=new Zaposleni(ime,dohodek); //ustvarimo novega zaposlenega
    }
    Console.WriteLine("Tabela zaposlenih in njihovih letnih dohodkov: ");
    //izpis podatkov iz tabele vseh zaposelnih
    for (int i = 0; i < Firma.Length; i++)
    {
        Console.WriteLine(Firma[i].imeInPriimek+": "+Firma[i].dohodek);
    }
    Console.ReadLine();
}
```

Več konstruktorjev

Za metode, tako "navadne" kot objektne, velja, da so lahko preobtežene. Popolnoma enako velja tudi za konstruktorje. Razred lahko vsebuje več konstruktorjev, ki pa se morajo razlikovati v parametrih (v podpisu).

Pri pisanju konstruktorjev pa je pomembno naslednje: če konstruktorja ne napišemo sami, potem ga C# avtomatično doda sam (privzeti konstruktor, brez parametrov). Kakor hitro pa napišemo poljubni konstruktor, se C# ne vmešava več in v tem primeru svojega konstruktorja NE doda. V ta namen si je dobro zapomniti, da v razredih vedno napišemo tudi privzeti konstruktor, ki je brez parametrov!

Kot primer razreda z več konstruktorji vzemimo razred *Zajec* iz prejšnjih poglavij. Uporabniki bi poleg privzetega zajca, radi še možnost, da bi takoj, ko zajca naredijo, le-temu določili še serijsko številko. Radi bi torej konstruktor

```
public Zajec(string serijskaSt)
```

Poleg tega pa včasih na farmo dobijo tudi pošiljko samic. Torej potrebujejo še konstruktor

```
public Zajec(bool spol)
```

Včasih pa so zajci "nestandardni"

```
public Zajec(string serijskaSt, bool spol, double teza)
```

Potrebovali bi torej razred *Zajec* , kim ima poleg osnovnega (oz. privzetega) še tri konstruktorje. A ker morajo imeti vsi konstruktorji ime točno tako, kot je ime razreda, to pomeni, da se morajo razlikovati v tipu oz. številu parametrov . Konstruktorje razreda *Zajec* bi tako napisali takole:

```
public class Zajec
{
   public string serijska;
   public bool spol;
   public double masa;
   public Zajec() //Osnovni konstruktor brez parametrov
   {
        this.spol = true; // vsem zajcem na začetku določimo moški spol
        this.masa = 1.0; // in tehtajo 1kg
        this.serijska = "NEDOLOČENO"; // serijska številka je še nedoločena
   }
   //Preobteženi konstruktor s parametrom tipa string za nastavljanje
   //serijske številke
   public Zajec(string serijskaSt)
   {
       this.serijska = serijskaSt;
   }
   //Preobteženi konstruktor s parametrom tipa bool za nastavljanje spola
   public Zajec(bool spol)
   {
       this.spol = spol;
   }
   //Preobteženi konstruktor s tremi parametri za nastavljanje vseh treh
   //polj
   public Zajec(string serijskaSt, bool spol, double masa)
   {
       this.serijska = serijska;
       this.masa = masa;
       this.spol = spol;
   }
}
```

Ustvarimo sedaj štiri zajce, vsakič pa uporabimo drug konstruktor:

```
//ustvarimo novega zajca in mu s konstruktorjem določimo privzete vrednosti
//polj
Zajec prvi = new Zajec();
//ustvarimo novega zajca in mu s konstruktorjem določimo serijsko številko;
//spol je privzet (false), masa prav tako (0)
Zajec drugi = new Zajec("12001-01-1");
//ustvarimo novega zajca in mu s konstruktorjem določimo spol; serijska
//številka je privzeta (prazen string), masa prav tako (0)
Zajec tretji = new Zajec(true);
//ustvarimo novega zajca in mu s konstruktorjem določimo serijsko, spol in
//maso
Zajec cetrti = new Zajec("12001-01-2",true,1.55);
```

Pri pisanju konstruktorjev se lahko sklicujemo tudi na že napisani konstruktor (oz konstruktorje). Četrti konstruktor bi lahko tako zapisali tudi takole:

```
public Zajec(string serijskaSt, bool spol, double masa)
   :this (serijskaSt) //sklic na drugi konstruktor
{
   this.masa = masa;
   this.spol = spol;
}
```

Zgledi

Datum

Denimo, da v naših programih pogosto delamo z datumi. Zato bomo sestavili ustrezni razred. Najprej moramo pripraviti načrt razreda. Odločiti se moramo torej, kako bi hranili podatke o datumu. Podatki, ki sestavljajo datum, so dan (število), mesec (izbira: število ali pa niz) in leto (število). Denimo, da se odločimo, da bomo dan in leto hranili kot število, mesec pa kot niz (torej kot januar, februar, ...). Sedaj se moramo odločiti, katere metode bo poleg konstruktorjev poznal naš razred. Nekaj idej:

- Lepo izpiši
- Povečaj za 1 dan
- Je datum smiselen
- Je leto prestopno
- Nov datum za toliko in toliko dni pred/za danim datumom
- Dan v tednu
- ...

Potem, ko smo si pripravili načrt, se lotimo sestavljanja razreda in napišimo nekatere od prej naštetih metod:

```
public class Datum
{
    public int dan;
    public string mesec;
    public int leto;
    public Datum() //osnovni konstruktor
    {
      dan = 1;
      mesec = "januar";
      leto = 2000;
    } // privzeti datum je torej 1.1.2000
    //Dodajmo še dva konstruktorja
    public Datum(int leto) : this()
    {
      this.leto = leto; // this je nujen
```

```
    } // datum je torej 1.1.leto
    public Datum(int d,string m,int l):this(l)//upoštevamo prejšnji
                                              //konstruktor
    { // leto smo torej že nastavili
       this.mesec = m; // this ni nujen
       this.dan = d;
    } // datum je torej d.m.l
   //Sedaj se lotimo metod. Kot prvo, nas zanima, ali je leto prestopno
   public bool JePrestopno()
   {
      int leto = this.leto;
      if (leto % 4 != 0) return false;
      if (leto % 400 == 0) return true;
      if (leto % 100 == 0) return false;
      return true;
   }
}
```

Razred *Datum* smo uspešno naredili, sedaj pa ga še koristno uporabimo. Denimo, da želimo ugotoviti, če je letošnje leto prestopno. Dovolj bo, če le naredimo objekt, v katerega shranimo katerikoli letošnji datum in pokličemo pripravljeno metodo *JePrestopno*().

```
static void Main(string[] args)
{
    Datum danes = new Datum(30, "marec", 2009);
    if (danes.JePrestopno())
    {
        Console.WriteLine("Je prestopno leto");
    }
    else
    {
        Console.WriteLine("Ni prestopno leto");
    }
}
```

Kompleksna števila

Deklarirajmo razred *Complex*, ki naj predstavlja kompleksno število. Struktura naj ima dva konstruktorja (privzetega in še enega za nastavljanje realne in imaginarne komponente). Napišimo še metodo za izpis kompleksnega števila, nato pa kreirajmo tabelo 10 kompleksnih števil in jo inicializirajmo z naključnimi vrednostmi obeh komponent.

```
public class Complex
{
    public float realna;
    public float imaginarna;
    public Complex() // privzeti konstruktorr
    {
        realna = 0;
        imaginarna = 0;
    }
    //Konstruktor za nastavljanje vrednosti polj
    public Complex(float real, float imag)
    {
        realna = real;
        imaginarna = imag;
    }
```

```
    //Objektna metoda ki kompleksno število vrne kot string
    public string ToString(Complex C)
    {
        return (C.realna + " + ( " + C.imaginarna + " * i )");
    }
}

public static void Main()
{
    Complex[] tabK = new Complex[10]; //tabela 10 kompleksnih števil
    Random naklj = new Random();
    for (int i = 0; i < tabK.Length; i++)
    {
        tabK[i] = new Complex(naklj.Next(-10, +10), naklj.Next(-10, +10));
        Console.WriteLine("Kompleksno število št "+(i+1) + ": " + tabK[i].
           ToString (tabK[i]));
    }
    Console.ReadLine();
}
```

Padavine

Za vodenje evidence o padavinah v zadnjem letu potrebujemo naslednje podatke: ime kraja in podatke o količini padavin v zadnjih 12 mesecih (12 števil v polju/tabeli). Kreirajmo ustrezeno razred in metodi za vnos podatkov o padavinah za vseh 12 mesecev, ter metodo, ki vrne povprečno mesečno količino padavin ustreznega kraja!

```
public class padavine
{
    public string ime_kraja;
    public double[] kolicina; //deklaracija tabele padavin
    public padavine()//privzeti konstruktor
    {
        kolicina = new double[12]; //inicializacija tabele padavin
    }
    public padavine(string ime) //konstruktor ki nastavi ime kraja
    {
        ime_kraja = ime;
        kolicina = new double[12]; //inicializacija tabele padavin
    }
    public void vnos() //metoda za vnos količine padavin kraja
    {
        Console.WriteLine("\nVnos mesečne količine padavin za kraj: " +
                          ime_kraja);
        for (int i = 0; i < 12; i++)
        {
            Console.Write("Količina padavin v " + (i + 1) + ". mesecu: ");
            kolicina[i] = Convert.ToDouble(Console.ReadLine());
        }
    }
    //metoda za izračun povprečne količine padavin določenega kraja
    public double povp()
    {
        double vsota = 0;
        for (int i = 0; i < 12; i++)
            vsota += kolicina[i];
        //rezultat vrnemo zaokrožen na dve decimalki
        return (Math.Round(vsota / 12, 2));
```

```
    }
};

//glavni program
static void Main(string[] args)
{
    padavine kraj1 = new padavine("Kranj");
    kraj1.vnos(); //klic metode za vnos padavin kraja
    Console.WriteLine("Povprečna količina padavin v mestu " +
                  kraj1.ime_kraja + " je bila " + kraj1.povp());
    Console.ReadLine();
}
```

Točka

Napišimo razred *Tocka*, ki nja ima dve zasebni polji (koordinati točke), privzeti konstruktor, ki obe koordinati postavi na 0, ter konstruktor z dvema parametroma, s katerima inicializiramo koordinati nove točke. Napišimo še metodo, ki izračuna in vrne razdaljo točke od neke druge točke.

```
class Tocka
{
    public Tocka()//lasten privzeti konstruktor
    {
       x = 0;
       y = 0;
    }

    public Tocka(int initX, int initY) //Konstruktor z dvema parametroma
    {
       x = initX;
       y = initY;
    }
    //metoda za izračun razrdalje od poljubne točke
    public double RazdaljaOd(Tocka druga)
    {
       int xRazd = x - druga.x;
       int yRazd = y - druga.y;
       return Math.Sqrt(Math.Pow(xRazd,2) + Math.Pow(yRazd,2));
    }
    private int x, y;
}

static void Main(string[] args)
{
    Tocka A = new Tocka();  //Klic konstruktorja brez parametrov
    Tocka B = new Tocka(600, 800); //Klic konstruktorja z dvema parametroma
    double razdalja = A.RazdaljaOd(B); //Izračun razdalje obeh točk
    Console.WriteLine("Razdalja točke A od točke B je enaka " + razdalja+"
                      enot!");
}
```

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 npr. za našega zajca *rjavka* v programu mirno napišemo

```
rjavko.masa = -3.2;
```

kar pa je seveda traparija. 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 potem moramo 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.

Objekt bomo zaradi tega velikokrat imeli za "črno škatlo" in njegove notranje zgradbe ne bomo "pokazali javnosti". Zakaj je to dobro? Če malo pomilsimo, 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 takih okoliščinah prestavimo iz tretje v četrto prestavo.

S "skrivanjem podrobnosti" omogočimo, da se v primeri ko pride do spremembe tehnologije, za uporabnika se 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 najnovejši verziji programskega jezika C# spremenili način hranjenja podatkov za določanje naključnih števi v objektih razreda *Random*. Ker dejansko nimamo vpogleda (neposrednega dostopa) v te spremenljivke, nas kot uporabnike razreda *Random* ta sprememba nič ne prizadane. 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).

In še enkrat: s tem ko "ne dovolimo" vpogleda v zgradbo objekta, lahko poskrbimo za zaščito pred nedovoljenimi stanji. Do sedaj smo spremenljivke, ki opisujejo objekt, kazali navzven (imele so določilo **public**). To pomeni, da uporabnik lahko neposredno dostopa do teh spremenljivk in morebiti objekt spravi v nemogoče stanje, npr.

```
mojAvto.prestava = -10;
```

Dostopi do stanj

Do sedaj smo pisali programe tako, da smo naredili objekt. Ko smo želeli v objekt zapisati kak podatek, smo uporabili zapis:

```
ime_objekta.stanje = <izraz>;
```

Za primer vzemimo razred *Clan* in primer kreiranja novega člana

```
public class Clan
{
    public string ime;
    public string priimek;
    public int leto_vpisa;
    public string vpisna_st;
}
static void Main(string[] args)
{
    Clan novClan = new Clan();  //Nov objekt razreda Clan
```

```
    novClan.ime = "Katarina";
    novClan.leto_vpisa = 208;
}
```

Problem s takim pristopom nam kaže ravno vrstica, kjer smo se zatipkali in vnesli nemogoč podatek – za leto vpisa smo vnesli 208. Objekt *novClan* sedaj hrani napačen podatek. Radi pa bi, da take tipkarske napake "polovimo". Seveda bi lahko rekli – v redu, napisali bomo metodo, s katero bomo nastavljali leto vpisa in v metodi preverili, če je podatek smiselen. Takega pristopa smo se do sedaj že večkrat poslužili v zgledih in vajah.

Objektna metoda (znotraj razreda *Clan*) za nastavljanje leta vpisa bi izgledala npr. takole:

```
public bool SpremeniLetoVpisa(int leto)
{
  if ((2000 <= leto) && (leto <= 2020))
  {
    this.leto_vpisa = leto;
    return true;//leto je smiselno, popravimo stanje objekta in vrnemo true
  }
  return false; // leto ni smiselno, ne spremenimo nič in vrnemo false
}
```

In še zgled uporabe tega razreda

```
1:  Clan novClan = new Clan();
2:  novClan.ime = "Katarina";
3:  novClan.leto_vpisa = 2007;
4:  novClan.SpremeniLetoVpisa(208);
5:  novClan.SpremeniLetoVpisa(2008);
```

Sedaj v 4. vrstici podatek ne bo popravljen na napačnega, ker letnica ni ustrezna, v 5. vrstici pa bo sprememba uspešna. Če bi želeli preveriti, ali je sprememba uspela ali ne, bi lahko pogledali, kakšno vrednost je vrnila metoda. Po drugi strani pa že iz vrstice 5 vidimo, da je naša "zaščita" zelo pomanjkliva. Če bi napisali še

```
novClan.leto_vpisa = 20008;
```

bi se vrstica veselo izvedla in spet bi imeli v objektu napačen podatek.

Če bi torej preprečili nastavljanje polj (lasnosti) objekta na način

```
ime_objekta.stanje = <izraz>;
```

in bi se stanje spremenljivk, ki opisujejo objekt lahko spreminjalo le preko metod, bi lahko poskrbeli, da bi pred spremembo podatka izvedli ustrezne kontrole in v primeru poskusa nastavljanja napačnih podatkov ustrezno reagirali.

V ta namen imamo v C# možnost, da dostop do spremenljivk lahko nadziramo. V C# poznamo 4 načine dostopa:

- public
- private
- protected
- internal

Zadnjih dveh mi ne bomo uporabljali, zato si njihov pomen in uporabo oglejte kar sami. Dostop *public* smo že ves čas uporabljali, saj smo pisali na primer:

```
public string serijska;
```

```
public double masa;
```

Besedi *public* pomenita, da do teh dveh spremenljivk dostop ni omejen (je **javen** – *public*). Če tega ne želimo, *public* zamenjamo s *private*

```
private string serijska;
private int[] tab;
private Datum datumRojstva;
```

Včasih besedo, ki označuje način dostopa (javen ali privaten) spustimo in napišemo le

```
bool spol;
```

Kakšen dostop velja takrat, je odvisno od okoliščin. Običajno bo to kar *public*, ne pa vedno. Zato se opuščanju navedbe dostopa izogibajmo in ga pišimo vedno. Zakaj bi zgubljali čas s premišljevanjem o tem, v kakšnih okoliščinah je ta spremenljivka in kakšen bo potem dostop do nje.

Oglejmo si sedaj razliko med *public* in *private*. Zavedati pa se moramo, da znotraj razreda (torej ko sestavljamo razred in pišemo njegove metode**) ni omejitev**. Vedno (ne glede na način dostopa) je možen dostop do komponent – vse metode, napisane znotraj zareda, imajo neomejen dostop do vseh polj, ne glede na to, ali so polja javna (*public*) ali zasebna (*private*).

Omejitve pri načinu dostopa veljajo le, ko objekte določenega razreda uporabljamo – torej v drugih programih in razredih!

public

Če je način dostopa nastavljen na *public*, to pomeni, da do lastnosti (komponent, spremnljivk, polj ...) lahko dostopajo vsi, od kjerkoli (iz katerihkoli datotek (razredov)) in sicer z

```
ime_objekta.lastnost
```

Denimo, da smo v razredu `MojObjekt` napisali

```
public int javnaLastnost;
```

Kdorkoli naredi objekt vrste *MojObjekt*, npr. s stavkom:

```
MojObjekt x = new MojObjekt();
```

lahko dostopa do *javnaLastnost* neposredno:

```
x.javnaLastnost
```

Ta način smo uporabljali do sedaj.

private

Dostop *private* pomeni, da do lastnosti ne more dostopati nihče, razen metod znotraj razreda. Denimo, da smo v razredu `MojObjekt` napisali

```
private int privatnaLastnost;
```

Če sedaj naredimo objekt vrste *MojObjekt*:

```
MojObjekt x = new MojObjekt();
```

bo ob poskusu dostopa do *privatnaLastnost* na takle način:

```
x.privatnaLastnost
```

prevajalnik javil napako.

Zgled - Razred Zajec

```
public class Zajec
{
  public string serijska; // serijska stevilka zajca
  public bool spol; // true = moski, false = zenska
  private double masa; // masa zajca ob zadnjem pregledu
  public SpremeniTezo(double x) //Metoda za spremembo mase zajca
  {
     this.masa = x;//OK! Objektna metoda ima dostop tudi do zasebnega polja
  }
}

public static void Main(String[] ar)
{
  Zajec z1 = new Zajec();
  z1.serijska = "1238-12-0";
  z1.spol = false;
  z1.SpremeniTezo(0.11);
  z1.masa = 0.12;
  z1.masa = z1.masa + 0.3;
  System.Console.WriteLine("Zajec ima ser. št.:" + z1.serijska);
}
```

OK pri prevajanju!

Prevajalnik javi napako!

Dostop do stanj/lastnosti

Odločili smo se torej, da bomo iz objektov naredili črne škatle. Uporabnikom bomo z zasebnimi poli (*private*) preprečili, da bodo "kukali" v zgradbo objektov določenih razredov. Če bodo želeli dostopati do lastnosti (podatkov) objekta (zvedeti, kakšne podatke v objektu hranimo) ali pa te podatke spremeniti, bodo morali uporabljati metode. In v teh metodah bo sestavljalec 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 prično z *Get* (Daj)). Druge pa so t.i. "*set*" metode (set / nastavi)[1].

Ponovimo še enkrat, zakaj je pristop z *get* in *set* metodami boljši od uporabe javnih spremenljivk. Spotoma pa bomo navedli še kak razlog in (upamo) prepričali še zadnje dvomljvce, ki se tako težko poslovijo od javnih spremenljivk.

[1] 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 na začetku lastnosti spustili in bomo pisali "svoje" *get* in *set* metode.

Razlogi, zakaj je dostop do podatkov v objektu preko metod boljši kot neposredni dostop (preko javnih spremenljivk) so med drugim:

- Možnost kontrole pravilnosti!
- Možnost kasnejše spremembe načina predstavitve (hranjenja podatkov).
- Možnost oblikovanja pogleda na podatke:
- Podatke uporabniku posredujemo drugače, kot jih hranimo. Tako denimo v razredu *Datum* mesec hranimo kot število, proti uporabniku pa je zadeva videti, kot bi uporabljali besedni opis meseca.
- Dostop do določenih lastnosti lahko omejimo:
    - Npr. spol lahko nastavimo le, ko naredimo objekt (kasneje pa uporabnik spola sploh ne more spremeniti, saj se ne spreminja ... če odmislimo kakšne operacije, določene vrste živali ... seveda)
    - Hranimo lahko tudi določene podatke, ki jih uporabnik sploh ne potrebuje ..

Nastavitve podatkov (stanj)

Potrebujemo torej metode, ki bodo nadomestile prireditveni stavek. Za tiste podatke, za katere želimo, da jih uporabnik spreminja, napišemo ustrezno *set* metodo. Na primer:

```
zajcek.SpremeniTezo(2.5);
```

V bistvu smo s tem dosegli isto kot prej s prireditvenim stavkom

```
zajcek.masa = 2.5;
```

A metoda *spremeniTezo* lahko **PREVERI**, če je taka sprememba teže smiselna! Tako je zapis

```
zajcek.SpremeniTezo(-12.5);
```

načeloma nadomestek stavka

```
zajcek.masa = -12.5;
```

Vendar gre tu za bistveno razliko. Prireditveni stavek se bo zagotovo izvedel (in bo s tem zajec nevarno shujšal). Pri spreminjanju teže zajca z metodo, pa lahko **preprečimo** postavitev lastnosti objekta v napačno stanje! V metodi *SpremeniTezo* lahko izvedemo ustrezne kontrole in če sprememba ni smislena, je sploh ne izvedemo.

Najprej poskusimo takole:

```
public void SpremeniTezo(double novaTeza)
{
   // smiselna nova teza je le med 0 in 10 kg
   if ((0 < novaTeza) && (novaTeza <= 10))
      this.masa = novaTeza;
      // v nasprotnem primeru NE spremenimo teže
}

public bool SpremeniTezo(double novaTeza)
{
   // smislena nova teza je le med 0 in 10 kg
   if ((0 < novaTeza) && (novaTeza <= 10)){
   this.masa = novaTeza;
      return true; // sprememba uspela
   }
   // v nasprotnem primeru NE spremenimo teže
   // in javimo, da spremembe nismo naredili
   return false;
```

```
}
```

Ali imamo lahko obe metodi? Žal ne, saj imata obe enak podpis. Spomnimo se, da podpis sestavljajo ime metode in tipi parametrov. Tip rezultata pa ni del podpisa! Pri sestavljanju razreda se bomo torej odločili za eno teh dveh metod (odvisno od ocenjenih potreb).
Metoda je seveda lahko bolj kompleksna. Denimo da vemo, da se teža zajca med dvema tehtanjima ne more spremeniti bolj kot za 15%. Zato lahko z metodo "polovimo" morda napačne poskuse sprememb.

```
public bool SpremeniTezoVmejah(double novaTeza)
{
   // smislena nova teža je le med 0 in 10 kg
   // in če ni več kot 15% spremembe od zadnjič
   int sprememba = (int)(0.5 + (100 * Math.Abs(this.masa - novaTeza) /
                         this.masa));

   if ((0 < novaTeza) && (novaTeza <= 10) && (sprememba <= 15) )
   {
      masa = novaTeza; // this.masa ... Lahko pa this spustimo!
      return true; // sprememba uspela
   }
   // v nasprotnem primeru NE spremenimo teže
   // in javimo, da spremembe nismo naredili
   return false;
}
```

Kaj pa metodi za spreminjanje serijske številke in spola? Na zadnjo lahko mirno pozabimo. Zajec naj ima ves čas tak spol, kot mu ga določimo pri "stvarjenju" (torej v konstruktorju). Zato *set* metode za spremninjanje spola sploh ne bomo napisali. In ker je spremenljivka spol zaščitena s *private*, je uporabnik ne bo mogel spremeniti. Seveda, če bomo naš razred *Zajec* potrebovali za kakšen genetski laboratorij, kjer se gredo čudne stvari, pa bo morda metoda *SpremeniSpol* potrebna.

Tudi serijske številke verjetno ne bomo spreminjali. No, če malo bolje premislimo, pa nam konstruktor brez parametrov ustvari zajca, ki ima za vrednost serijske številke niz "NEDOLOČENO". Takim zajcem bomo verjetno želeli spremniti serijsko številko. Zato naj metoda *SpremeniSerijsko* pusti spreminjati le nedoločene serijske številke!

```
public bool SpremeniSerijsko(string seStev)
{
   // sprememba dopustna le, če serijske številke še ni
   if (this.serijska.Equals("NEDOLOČENO"))
   {
     this.serijska = seStev;
     return true;
   }
   return false; // ne smemo spremniti že obstoječe!
}
```

Opozorimo še na stvar, ki jo pri sestavljanju razreda pogosto pozabimo. Glavni razlog, da smo napisali te *set* metode je, da bi zagotovili, da so podatki pravilni. Zato je smiselno, da zagotovimo, da je teža ustrezna že ves čas! A zaenkrat smo na nekaj pozabili! Kaj pa začetno stanje, ki ga nastavimo v konstruktorju! Tudi v konstruktorju preverimo, če se uporabnik "obnaša lepo". Pogosto na to pozabimo in uporabnik lahko napiše npr.

```
Zajec neki = new Zajec("X532", true, 105);
```

105 kilogramskega zajca verjetno ni, a uporabnik je pozabil na decimalno piko v 1.05. Zato je kontrola smiselnosti podatkov potrebna tudi v konstruktorjih!

Dostop do stanj

Pogosto nam je vseeno, če uporabnik "vidi", kako hranimo podatke v objektu. A zaradi zagotavljanja pravilnosti in možnosti kontrole sprememb podatkov, smo dostop nastavili na *private*. S tem smo seveda onemogočili tudi enostaven način poizvedbe po vrednosti podatka v obliki *rjavko.masa*. Če bomo torej potrebovali težo zajca, bo potrebno pripraviti ustrezno metodo. Če bomo torej potrebovali težo zajca *zajcek*, bomo napisali

```
zajcek.PovejTezo()
```

Večina tovrstnih *get* metod je enostavnih in v telesu metode vsebujejo le ukaz *return*.

```
public double PovejTezo()
{
   return this.masa; // ali return masa
}
```

Zgled – razred Datum

Kot primer oblikovanja pogleda na podatke si oglejmo še razred *Datum*. Običajno ni smiselno, da podatke o mesecu hranimo kot niz, raje jih hranimo kot celo število. A uporabniku bi jih še vedno raje "servirali" kot *februar* in ne kot *2*. Zato bi metoda *KateriMesec()* bila lahko taka

```
public class Datum
{
  private int dan;
  private int mesec;
  private int leto;

  public Datum(int dan, int mesec, int leto) // konstruktor
  {
     this.dan = dan;
     this.mesec = mesec;
     this.leto = leto;
  }
  public string KateriMesec()
  {
     string[] imenaMesecev ={"januar", "februar", "marec", "april", "maj",
     "junij","julij","avgust","september","oktober","november","december"};

     return imenaMesecev[this.mesec-1]; // metoda vrne opis meseca
  }
}
static void Main(string[] args)
{
    Datum danes=new Datum(1,5,2009);
    //izpis tekočega meseca v opisni obliki
    Console.WriteLine(danes.KateriMesec()); //izpis "maj"
    Console.ReadLine();
}
```

Seveda so potrebne še spremembe v drugim metodah. Npr. uporabnik bo verjetno želel imeti metodi *NastaviMesec*, ki bosta sprejeli bodisi številčni ali pa opisni podatek.

Ostale metode

Poleg *get/set* metod in konstruktorjev v razredu napišemo tudi druge metode, saj objektov ne potrebujemo samo kot "hranilnikov" podatkov. S temi metodami določimo

- odzive objektov
- "znanje objektov"

Če se na primer spomnimo na objekte tipa *string*:

- Znajo povedati, kje se v njih začne nek podniz:
  `"niz".IndexOf("i")`
- Znajo vrniti spremiti črke v male in vrniti nov niz:
  `nekNiz.toLower()`
- Znajo povedati, če so enaki nekemu drugemu nizu:
  `mojPriimek.Equals("Lokar")`

Zato bomo tudi v "naših" razredih sprogramirali znanje objektov določenega razreda s tem, da bomo napisali ustrezne metode. Katere bodo te metode je pač odvisno od načrtovane uporabe naših razredov.

Metoda ToString

Poglejmo si naslednji program:

```
public class Zajec
{
   private string serijska; // serijska stevilka zajca
   private bool spol; // true = moski, false = zenska
   private double masa; // masa zajca ob zadnjem pregledu
   public Zajec() //Osnovni konstruktor brez parametrov
   {
        this.spol = true; // vsem zajcem na začetku določimo moški spol
        this.masa = 1.0; // in tehtajo 1kg
        this.serijska = "NEDOLOČENO"; // serijska številka je še nedoločena
   }
}

public static void Main(String[] ar)
{
  Zajec zeko = new Zajec();
  Console.WriteLine("To je zajec: " + zeko );
  Console.ReadLine();
}
```

Če ga izvedemo, dobimo na zaslonu približno takle izpis:

Od kod, zakaj? Očitno objekte lahko tudi izpišemo. In če napišemo

```
string niz = "Zajec " + zeko + "!";
```

prevajalnik tudi nič ne protestira. Torej se tudi objekti "obnašajo" tako kot *int*, *double* ... in se torej po potrebi pretvorijo v niz.

Obnašanje, ki smo ga opazili (da se števila, objekti ...) pretvorijo v niz, omogoča metoda `ToString`. Metoda je nekaj posebnega, saj se lahko "pokliče kar sama". Namreč, če na določenem mestu potrebujemo *niz*, a naletimo na objekt, se metoda pokliče avtomatično. Torej

```
string niz = "Zajec " + zeko + "!";
```

dejansko pomeni

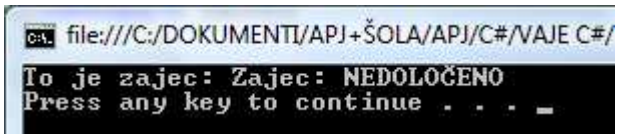
```
string niz = "Zajec " + zeko.ToString() + "!";
```

Od kje pride ta metoda, saj je v razredu *Zajec* nismo napisali. Gre spet za eno od tistih "čarovnij", ki jo počne prevajalnik, kot na primer pri konstruktorjih? Zadeva je malček drugačna in jo bomo *pojasnili* v nadaljevanju. Zaenkrat bodi dovolj le to, da ta metoda vedno obstaja, tudi če je ne napišemo in omogoča pretvorbo poljubnega tipa v niz.

A s pretvorbo nismo ravno zadovoljni. Radi bi kakšne bolj smiselne informacije. To storimo tako, da v razredu, ki ga definiramo (recimo *Zajec*), sami napišemo lastno metodo `ToString`.

```
public override string ToString()
{
    return "Zajec: " + this.PovejSerijsko();
}
```

Če to storimo in zaženemo isti program, dobimo izpis:



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 " podedovano metodo.

Seveda bi lahko napisali tudi neko drugo metodo, na primer *Opis*, ki bi prav tako vrnila niz s smiselnim opisom objekta.

```
public string Opis()
{
    return "Zajec: " + this.PovejSerijsko();
}
```

A med metodama *ToString* in *Opis* je bistvena razlika: metodo *Opis* moramo obvezno poklicati sami, metoda *ToString* pa se kliče avtomatsko (če je potrebno). Če torej napišemo

```
string niz1 = "Zajec " + zeko.ToString() + "!";
string niz2 = "Zajec " + zeko + "!";
string niz3 = "Zajec " + zeko.Opis() + "!";
```

so vsi trije dobljeni nizi enaki!

Zapomnimo si torej, da metoda *ToString* obstaja tudi, če jo ne napišemo. A verjetno z vrnjenim nizom nismo najbolj zadovoljni, zato jo praktično vedno napišemo sami.

Zgled: celotni razred Zajec

Oglejmo si sedaj celotno kodo razreda *Zajec* s spremembami, ki smo jih naredili. Spomnimo se, kaj vse smo opravili:

- Pripravili smo nekaj konstruktorjev
- Zaščitili smo dostop do podatkov (*private*)
- Napisali smo ustrezne metode za spreminjanje podatkov o objektu
- Napisali smo metode za poizvedovanje po vrednosti podatkov
- Napisali smo metodo *ToString*, ki vrne opis objekta
- Napisali smo dve metodi, ki pomenita "znanje" objektov

```
public class Zajec
{
   private string serijska;
   private bool spol;
   private double masa;
   // Osnovni konstruktor
   public Zajec()
   {
      this.spol = true; // vsem zajcem na začetku določimo m. spol
      this.masa = 1.0; // in tehtajo 1kg
      this.serijska = "NEDOLOČENO";
   }
   //Dodatni konstruktor za nastavljanje serijske številke
   public Zajec(string serijskaStev)
        : this() // poklicali smo osnovni konstruktor Zajec()
   {
      this.serijska = serijskaStev;
   }
   //Dodatni konstruktor za nastavljanje vseh treh polj
   public Zajec(string serijskaStev, bool spol, double masa)
   {
      this.serijska = serijskaStev;
      this.SpremeniTezo(masa); // uporabimo metodo za sprem.
      this.spol = spol;
   }

   public double PovejTezo()
   {
      // težo bomo povedali le na 0.5 kg natančno
      int tezaKg = (int)this.masa;
      int decim = (int)((this.masa -tezaKg) * 10);
      if (decim < 3) return tezaKg + 0.0;
      if (decim < 8) return tezaKg + 0.5;
      return tezaKg + 1.0;
   }

   public bool SpremeniTezo(double novaTeza)
   {
     // smislena nova teža je le med 0 in 10 kg
     if ((0 < novaTeza) && (novaTeza <= 10)){
        masa = novaTeza;
        return true; // sprememba uspela
     }
```

```
      // v nasprotnem primeru NE spremenimo teže
      // in javimo, da spremembe nismo naredili
      return false;
    }

    public string PovejSerijsko()
    {
      return this.serijska;
    }

    public bool SpremeniSerijsko(string seStev)
    {
      // sprememba dopustna le, če serijska številka še ni določena
      if (this.serijska.Equals("NEDLOČENO"))
      {
        this.serijska = seStev;
        return true;
      }
      return false; // ne smemo spremeniti že obstoječe!
    }

    public bool JeSamec()
    {
      return this.spol;
    }

    // ker se spol naknadno NE spremeni, metode za
    // spreminjanje spola sploh ne ponudimo uporabniku!
    public double Vrednost(double cenaZaKg)
    {
      // pove vrednost zajca
      // zajce tehtamo na dogovorjeno natančnost
      return cenaZaKg * this.PovejTezo();
    }

    public bool MeNapojiti()
    {
      //ugotovimo, če ga je smiselno napojiti, da bomo "ujeli" naslednjih
      //pol kg! Dejanska teža je večja od "izmerjene"
      return (this.masa - this.PovejTezo() > 0);
    }

    public override string ToString()
    {
         return "Zajec: " + this.PovejSerijsko();
    }
}
```

Uporaba razreda Zajec

Poglejmo si, kako opravljene spremembe vplivale na uporabo rezreda Zajec v uporabniških programih. Pred spremembami so bili programski stavki približno takšni:

```
Zajec z1 = new Zajec();
z1.serijska = "1238-12-0";
z1.spol = false;
z1.masa = 0.12;
```

```
z1.masa = z1.masa + 0.3;
System.Console.WriteLine("Zajec ima ser. št.:" + z1.serijska);
```

Z novim razredom *Zajec* seveda se način uporabe povsem spremeni, saj ne moremo pričakovati, da bodo tako drastične spremembe, ki smo jih naredili, pustile uporabniške programe nespremenjene. Še enkrat povejmo, da razreda *Zajec* ne bi nikoli "zares" napisali tako, kot smo ga po starem (z javnimi spremenljivkami). Uporaba novega razreda *Zajec* bi bila npr. takale:

```
// novega zajca stvarimo s konstruktorjem
Zajec z1 = new Zajec("1238-12-0", false, 0.12);
// uporabimo nekatere izmed metod razreda Zajec
if (z1.SpremeniTezo(z1.PovejTezo() + 0.3))
    System.Console.WriteLine("Teža je spremenjena na " + z1.PovejTezo());
else System.Console.WriteLine("Teža je nespremenjena!"
     + " Prirastek je prevelik! Preveri!");
System.Console.WriteLine("Zajec ima ser. št.:" + z1.PovejSerijsko());
```

Ustvarjanje razredov in objektov: povzetek

Naredimo sedaj povzetek vsega povedanega.:

Konkretni objekt je primerek nekega razreda. Nastane s pomočjo operatorja *new*. Izjema so na prvi pogled objekti razreda *string*. A dejansko ne gre za izjemo, gre le drugačen zapis, ki "skrije" uporabo operatiorja *new*! Podobno je pri tabelah ko uporabimo inicializacijo začetnih vrednosti, naštetih v zavitih oklepajih.

Posamičnim lastnostim objekta vedno določimo privatni način dostopa (da so torej dostopni le metodam znotraj razreda). To povemo tako, da ob deklaraciji napišemo *private*. Razlog za to je v tem, da uporabnika ne zanima način hranjenja lastnosti objekta, ampak le smiselna uporaba objektov. S tem tudi omogočimo bolj kontrolirano delo s podatki v objektu. V ta namen pripravimo metode za delo s podatki/lastnosti kot so :

```
VrniPodatek, NastaviPodatek, SpremeniPodatek, ...
```

V razredu moramo obvezno imeti tudi konstruktorje. To so metode, ki povedo, kaj se zgodi ob kreaciji novega objekta. Prvenstveno poskrbijo, da so vsi podatki o objektu takoj nastavljeni na smislene vrednosti.

Ko pišemo metode, s katerimi nastavljamo vrednost (*set*), ne smemo pozabiti na preverjanje pravilnosti parametrov. Stanje objekta mora biti ves čas pravilno. Že pri pisanju konstruktorjev ne smemo pozabiti na ustrezno preverjanje parametrov. Kot smo omenili, morajo tudi ti poskrbeti, da je novoustvarjeni objek v "smiselnem stanju").

Pri metodah, s katerimi vračamo vrednost (t.i. *get* metode, večjih zapletov ne gre pričakovati. Seveda pa lahko metode izkoristimo, da podatke, ki jih v objektu hranimo na en način, vrnemo na drug način (npr. podatek o mesecu, ki je hranjen številčno vrnemo opisno).

Vsak naš razred naj bi imel tudi metodo *ToString*. Ta poskrbi, da se bodisi ob klicu, bodisi avtomatično (če se na tistem mestu namesto objekta pričakuje niz) objekt pretvori v niz. Uporabljamo jo za “predstavitev” objekta. Ker zaradi mehanizma dedovanja ta metoda avtomatično obstaja v vsakem razredu, moramo na začetek dodati še besedo *override*.

Poleg metod za nastavljanje/vračanje podatkov in metode *ToString* imamo v razredu običajno še več drugih metod. Tiste, katerih uporaba je namenjena uporabnikom, se začno z določilom *public*, “interne” (pomožne) metode pa so *private*.

Predvsem ori konstruktorjih smo uporabili tehniko preobteževanja (*overloading*). Gre za to, da imamo lahko več metod, ki se imenujeo enako, razlikujejo pa se v seznamu parametrov.

Lastnost (Property)

Polja so znotraj razreda običajno deklarirana kot zasebna (private). Vrednosti smo jim doslej prirejali tako, da smo zapisali enega ali več konstruktorjev, ali pa smo napisali posebno javno metodo (imenovali smo jo *get/set* metodo) za prirejanje vrednosti polj. Ostaja pa še tretji način, ki je namenjen izkušenim programerjem – za vsako polje lahko definiramo ustrezno lastnost (*property*), s pomočjo katere dostopamo do posameznega polja, ali pa z njeno pomočjo prirejamo (nastavljamo) vrednosti polja. Lastnosti v razredih torej uporabljamo za inicializacijo oziroma dostop do polj razreda (objektov). Lastnost (*property*) je nekakšen križanec med spremenljivko in metodo. Pomen lastnosti je v tem, da ko jo beremo, ali pa vanjo pišemo, se izvede koda, ki jo zapišemo pri tej lastnosti. Branje in izpis (dostop) vrednosti je znotraj lastnosti realizirana s pomočjo rezerviranih besed **get** in **set**: ***get*** mora vrniti vrednost, ki mora biti istega tipa kot lastnost (seveda pa mora biti lastnost istega tipa kot polje, kateremu je namenjena), v ***set*** pa s pomočjo implicitnega parametra **value** lastnosti priredimo (nastavimo) vrednost.

Navzven pa so lastnosti vidne kot spremenljivke, zato lastnosti v izrazih in prirejanjih uporabljamo kot običajne spremenljivke.

Zgled:

```
class Demo
{
    int polje;              //zasebno polje
    public Demo()    //konstruktor
    {
       polje = 0;
    }
    //deklaracija lastnosti (property) razreda Demo
    public int MojaLastnost
    {
        get  //metoda get za pridobivaje vrednosti lastnosti polje
        {
            return polje  ;
        }
        set  //metoda set za prirejanje vrednosti lastnosti polje
        {
            polje = value;
        }
    }
}

static void Main(string[] args)
{
    Demo ob = new Demo();//nov objekt razreda Demo
    Console.WriteLine("Originalna vrednost ob.MojaLastnost: " +
                      ob.MojaLastnost);

    ob.MojaLastnost = 100;
    Console.WriteLine("Vrednost ob.MojaLastnost: " + ob.MojaLastnost);

    Console.WriteLine("Prirejanje nove vrednosti -10 za ob.MojaLastnost");
    ob.MojaLastnost = -10;
    Console.WriteLine("Vrednost ob.MojaLastnost: " + ob.MojaLastnost);
}
```

Zgled Oseba

Deklarirajmo razred *Oseba* z dvema poljema (*ime* in *priimek*) ter javno metodo za nastavljanje vrednosti obeh polj. Razred naj ima tudi lastnost *PolnoIme*, za prirejanje in vračanje polnega imena osebe (imena in priimka). Ustvarimo nov objekt in demonstrirajmo uporabo lastnosti *PolnoIme*

```
class Oseba
{
    private string priimek; //zasebno polje razreda Oseba
    private string ime;     //zasebno polje razreda Oseba

    public void NastaviIme(string ime,string priimek)//javna metoda razreda
    {
        this.priimek = priimek;
        this.ime = ime;
    }
    public string PolnoIme   //javna lastnost razreda
    {
        get
        {
            return ime + " " + priimek;
        }
        set
        {
            string zacasna = value;
            //Celotno ime razdelimo na posamezne besede
            string[] imena = zacasna.Split(' ');
            //in jih spravimo  tabelo
            ime = imena[0];  //za ime vzamemo prvi string v tabeli
            //za priimek vzamemo drugi string v tabeli
            priimek = imena[imena.Length - 1];
        }
    }
}

static void Main(string[] args)
{
    Oseba politik = new Oseba();
    politik.NastaviIme("Nelson", "Mandela");
    //Izpis: Polno ime osebe je Neslon Mandela
    Console.WriteLine("Polno ime osebe je " + politik.PolnoIme);
    politik.PolnoIme = "Barack Obama";
    //Izpis: Polno ime osebe je Barack Obama
    Console.WriteLine("Polno ime osebe je " + politik.PolnoIme);
    politik.PolnoIme = "France Prešeren";
    //Izpis: Polno ime osebe je France Prešeren
    Console.WriteLine("Polno ime osebe je " + politik.PolnoIme);
}
```

Za posamezno polje lahko napišemo le eno lastnost, v kateri s pomočjo stavkov **get** ali **set** dostopamo oz. nastavljamo vrednosti posameznega polja.

Besedici **get** ali **set** lahko izpustimo in dobimo ***write-only*** ali ***read-only*** lastnosti.

```
class MojRazred
{
   double A = 3;  //zasebno polje
   double B = 4;  //zasebno polje
```

```
   //MojaVrednost je ReadOnly: vsebuje le get, ne pa tudi set
   public double MojaVrednost
   {
      get { return A * B; } //lastnost vrne vrednost produkta obeh polj
   }
}

static void Main(string[] args)
{
    MojRazred c = new MojRazred();   //nov objekt tipa MojRazred
    Console.WriteLine("MojaVrednost: "+c.MojaVrednost);
}
```

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, takole:

```
class Math
{
    public static double Sqrt(double d)
    {
       . . .
    }
}
```

Zapomnimo pa si, da statično metodo ne kličemo tako kot objekt. 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.

Napišimo razred točka in v njem statično metodo, katere naloga je le ta, da vrne string, v katerem so zapisane osnovni podatki o tem razredu

```
class Tocka
{
   public static string Navodila()//Statična metoda
   {
      string stavek="Vsaka točka ima dve koordinati/polji: x je abscisa, y
                     je ordinata!";
      return stavek;
   }
}

//Zato, da pokličemo statično metodo oz. statično polje NE POTREBUJEMO
//OBJEKTA!!!  Primer klica npr. v glavnem programu
Console.WriteLine(Tocka.Navodila());//Klic statične metode
```

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.

```
class Test
{
    public const double Konstanta = 20;//Statično polje
}

//Zato, da pokličemo statično metodo/polje NE POTREBUJEMO OBJEKTA!!!
Console.WriteLine(Test.Konstanta);//klic statičnega polja
```

Primer:

```
public class Foo
{
   static public int x = 12; //x je STATIČNO polje (pripada razredu)
   public int y; //y je običajno, objektno polje (pripada objektom)

   public Foo(int z)
   {
      this.y = z;
   }

   public static int F(int a)
   {
      return x + a;
      //ali return Foo.x + a;
   }

   public int G(int a)
   {
      /*Znotraj metode G() označuje this objekt, na katerem je metoda
        poklicana. Objektna metoda G ima dostop do komponente this.y, kjer
        je this objekt, na katerem je metoda g klicana. Prav tako ima
        dostop do (statične) komponente x. */
        return this.y + Foo.x + a;
       //ali return y + Foo.x + a;
       //ali return  y + x + y;
       //ali return  ...
```

```
    }
}


static void Main(string[] args)
{
    /*Objektno metodo G()v razredu Foo lahko izvedemo, če imamo neki objekt
      obj1 razreda Foo, z ukazom obj1.G().*/

    Foo obj1 = new Foo(12);
    Foo obj2 = new Foo(30);
    /*Oba objekta (obj1 in obj2 ) vsebujeta svojo kopijo komponente y. Ker
    pa je komponenta x statična, vedno obstaja v eni sami kopiji, tudi če
    nimamo nobenih objektov razreda Foo*/

    int p = obj1.G(7);      // p == 12 + 7 == 19
    Console.WriteLine(p);  // Izpis 19
    /*Statična metoda F ima dostop do (statične) komponente x. Dostopa do
    komponente y nima,saj znotraj statične metode ne moremo pisati this.y*/
    Foo.x = -3;

    /*Statično metodo F() v razredu Foo lahko vedno izvedemo z ukazom
     Foo.F(). Znotraj statične metode objekt this ni definiran, ker se
     statična metode ne kliče na objektu.*/
    int q = Foo.F(5);      // q == -3 + 5 == 2
    Console.WriteLine(q); // Izpis 12
    Foo t = new Foo(100);
    Foo s = new Foo(200);
    int r = t.G(50);       // r == 100 + (-3) + 50 == 147
    Console.WriteLine(r); // Izpis 147
    Console.ReadKey();
}
```

Zgled

```
class Nekaj
{
  static int stevilo=0;  //Statično polje
  public Nekaj()         // Konstruktor
  {
     stevilo++;        //število objektov se je povečalo
  }

  public void Hello()
  {
     if (stevilo>3)
     {
        Console.WriteLine("Sedaj smo pa že več kot trije! Skupaj nas je že
                           "+stevilo);
     }
     switch (stevilo)
     {
        case 1 : Console.WriteLine("V tej vrstici  sem sam !!");
                 break;
        case 2 : Console.WriteLine("Aha, še nekdo je tukaj, v tej
                                    vrstici sva dva!!");
                 break;
        case 3 : Console.WriteLine("Opala, v tej vrstici smo že trije.");
                 break;
     }
```

```
  }
}

static void Main(string[] args)
{
   Nekaj a=new Nekaj();              //konstruktor za a
   a.Hello();
   {
      Nekaj b=new Nekaj();          //konstruktor za b
      b.Hello();
      {
         Nekaj c=new Nekaj();      //konstruktor za c
         c.Hello();
         {
            Nekaj d=new Nekaj(); //konstruktor za d
            d.Hello();
         }
      }
   }
}  //Konec programa-> Garbage Collector poskrbi za uničenje vseh objektov
```

Polje razreda je lahko statično a se njegova vrednost ne more spremeniti: pri deklaraciji takega statičnega polja zapišemo besedico **const** (const = Constant – konstanta). Besedica **static** pri deklaraciji konstantnega polja **NI** potrebna, pa je polje še vedno statično. Konstantno polje je torej avtomatično statično in je tako dostopno preko imena razreda in ne preko imena objekta! **Vrednost statičnega polja se NE da spremeniti**. Tako je npr. deklarirana konstanta **PI** razreda **Math**.

Primer:

```
class Test
{
    public static double kons1 = 3.14; //Statično polje
    public const double kons = 3.1416; //Konstantno polje je avtomatično
                                       //tudi statično
}

//Zato, da pokličemo statično polje NE POTREBUJEMO OBJEKTA!!!
Console.WriteLine(Test.kons1);//klic statičnega polja
Console.WriteLine(Test.kons);//klic konstantnega statičnega polja
Test.kons1= Test.kons1 + 1; //OK -> Statično polje LAHKO spremenimo
Test.kons= Test.kons + 1; //NAPAKA, konstantnega polja ne moremo spremeniti
```

Že iz prejšnjih primerov je razvidno, kdaj v razredih uporabljamo statična poja. Uporabljamo jih za različne konstante, za enolične identifikacije (ko naj ima npr. objekt svojo serijsko številko in so te številke zaporedne), vedno, ko je določen podatek skupen za vse objekte tega razreda

Kako vemo, ali naj bo komponenta (polje ali pa metoda) statična ali objektna? Statične komponente so tiste, ki so skupne za celoten razred – niso posebna lastnost enega (ali nekaj) primerkov objektov tega razreda. Lahko so zasebne (*private*) ali pa javne (*public*), odvisno pač od željenega načina dostopa. Če jih želimo skriti pred uporabniki, bo način dostopa *private*.

Zgledi

Kvader

Razred *Kvader* vsebuje tri zasebna polja (robove kvadra), statično polje *kolikoNasJe*, ki hrani podatek o številu živečih objektov razreda *Kvader*, dve statični *readonly* polji *MIN_DOL_STR* (minimalna dolžina roba) in *MAX_DOL_STR* (maksimalna dolžina roba). Za polje, ki je označeno kot *readonly* velja podobno kot za konstante in predstavlja vrednost, ki je kasneje ne moremo več spremeniti. Readonly polje lahko inicializiramo ob deklaraciji ali pa v statičnem konstruktorju (statični konstruktorji se uporabljajo le za inicializacijo statičnih polj). V razredu *Kvader* so štirje konstruktorji.

```
public class Kvader
{
    //polja so private:uporabnika ne zanima, kako imamo shranjene podatke *
    private double a;
    private double b;
    private double c;

    private int serijskaStevilka; // serijska števila objekta

    //skupno število ustvarjenih objektov
    private static int kolikoNasJe = 0;
    //minimalna dolžina roba kvadra
    public static readonly int MIN_DOL_STR = 1;
    //maksimalna dolžina roba kvadra
    public static readonly int MAX_DOL_STR = 100;
    /* Ker menimo, da je smiselno, da uporabnik neposredno vidi ti dve
       količini je dostop public. Spreminjati ju ne more zaradi readonly.
       Dostop: Kvader.MIN_DOL_STR
       Določilo static pomeni, da gre za razredno spremenljivko, torej je
       skupna za vse objekte
     */

    // vrni podatek
    public int PovejA()
    { // omogočimo uporabniku dostop do vrednosti zanj zanimivih podatkov *
        return (int)this.a;
    }
    public int PovejB()
    {
        return (int)this.b;
    }
    public int PovejC()
    {
        return (int)c;
    }

    private bool Kontrola(int x)
    { /* pomožna metoda, zato private */
        // preverimo, ce velja MIN_DOL_STR <= x <= MAX_DOL_STR
        return ((MIN_DOL_STR <= x) && (x <= MAX_DOL_STR));
    }

    // nastavi podatek
    public void StranicaA(int a)
    { /* omogočimo uporabniku spreminjanje podatkov */
        // Če je podatek v mejah med 1 in 100, ga spremenimo,
        // drugače pustimo takšnega, kot je
        if (Kontrola(a))
```

```
            this.a = a;  /* this je potreben, da ločimo med
                imenom podatka objekta in parametrom a */
        /* verjetno bi bilo bolj smiselno namesto
         a parameter poimenovati s ali kako drugače */
    }
    public void StranicaB(int a)
    {
        // Če je podatek v mejah med 1 in 100, ga spremenimo,
        // drugače pustimo takšnega, kot je
        if (Kontrola(a))

            this.b = a; /* this v nasprotju s prejšnjo metodo
               tu ni potreben. Še vedno pa velja, da bi bilo boljše,
               da bi parameter a poimenovali drugače - npr. s */
    }
    public void StranicaC(int s)
    {
        // Če je podatek v mejah med 1 in 100, ga spremenimo,
        // drugače pustimo takšnega, kot je
        if (Kontrola(s)) /* na ta način bi verjetno "zares"
                  sprogramirali tudi zgornji metodi */
            c = s;
    }

    // konstruktorji
    public Kvader()
    {
        a = b = c = 1;
        kolikoNasJe++;
        serijskaStevilka = kolikoNasJe;
    }

    public Kvader(int s)
    {
        // kocka z robom s. Če podatek ni v redu - kocka z robom 1
        if (!Kontrola(s)) // če podatek ni v mejah,
            s = 1;        // ga postavimo na 1

        a = b = c = s;
        kolikoNasJe++;
        serijskaStevilka = kolikoNasJe;
    }

    public Kvader(int s1, int s2, int s3)
    { // kvader s1 x s2 x s3
        // Če kak podatek ni v redu - ga postavimo na 1
        if (!Kontrola(s1)) s1 = 1; // če podatek ni v mejah,
          a = s1;  // ga postavimo na 1

        if (!Kontrola(s2)) s2 = 1; // če podatek ni v mejah,
          b = s2;  // ga postavimo na 1


        if (!Kontrola(s3)) s3 = 1; // če podatek ni v mejah,
          c = s3;  // ga postavimo na 1

        kolikoNasJe++;//povečamo število objektov
        serijskaStevilka = kolikoNasJe;
    }
```

```
    public Kvader(Kvader sk)
    {
        // Novi objekt je kopija obstoječega objekta sk
        this.a = sk.a; /* this ni potreben */

        /*tudi v razredu se splača uporabljati metode za delo s podatki
        čeprav imamo neposreden dostop do spremenljivk, saj nam ob
        morebitni spremembi predstavitve ni potrebno toliko popravljati!*/
        StranicaB(sk.PovejB());
        StranicaC(sk.PovejC());
        kolikoNasJe++;
        serijskaStevilka = kolikoNasJe;
    }

    public static int PovejKolikoNasJe()
    {
        return kolikoNasJe;
    }

    public override string ToString()
    {  /* prekrivanje metode iz razreda Object */
       // Metoda vrne podatek v obliki Kvader: a x b x c
       return "Kvader: "+Kvader.kolikoNasJe+": "+ + PovejA() + " x " +
              PovejB() + " x " + PovejC();
    }

} // class Kvader
static void Main(string[] args)
{
    //Uporabimo privzeti konstruktor: vsi trije robovi bodo enaki 1
    Kvader K1 = new Kvader();
    Console.WriteLine(K1.ToString());

    //Uporabimo drugi konstruktor: vsi trije robovi bodo enaki 5
    Kvader K2 = new Kvader(5);
    Console.WriteLine(K2.ToString());

    //Uporabimo tretji konstruktor: robove nastavimo na 2, 4 in 6
    Kvader K3 = new Kvader(2, 4, 6);
    Console.WriteLine(K3.ToString());

    Kvader K4 = new Kvader(K3);//ustvarimo kopijo kvadra K3
    Console.WriteLine(K4.ToString());
    //Klic statične metode PovejKolikoNasJe, ki pove, koliko kvadrov smo
    //doslej že ustvarili
    Console.WriteLine("Skupno število živečih kvadrov: " +
                      Kvader.PovejKolikoNasJe());

    Kvader K5 = new Kvader();
    //velikosti robov petega kvadra nastavimo z našimi Set metodami
    K5.StranicaA(6);
    K5.StranicaB(8);
    K5.StranicaC(10);
    Console.WriteLine(K5.ToString());

    //Skušajmo ustvariti kvader z nemogočimi robovi
    //metoda Kontrola bo rob a in rob b spremenila na 1
    Kvader K6 = new Kvader(-2, 48888, 6);
    Console.WriteLine(K6.ToString());
    Console.WriteLine("Skupno število živečih kvadrov: " +
```

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

Dedovanje (Inheritance) – izpeljani razredi

Kaj je to dedovanje

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 Microsoft C# bi lahko za ta primer modelirali dva razreda: prvega bi poimenovali Sesalec, in drugega Konj, in obenem deklarirali, da je Konj podeduje (inherits) 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.

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. Za razliko od C++, lahko razred v C# 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:

Radi bi napisali razred, s katerim bi lahko predstavili točko v dvodimenzionalnem koordinatnem sistemu. Razred poimenujmo Tocka:

```
class Tocka //bazični razred
{
    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
{
   //telo razreda Tocka3D - tukaj zapišemo še dodatne metode tega razreda!
}
```

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 (rezervirana besedica base). 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
}
```

Določanje oz. prirejanje razredov

Poglejmo še, kako lahko kreiramo objekte iz izpeljanih razredov. Kot primer vzemimo zgornji razred *Tocka* in iz njega izpeljani razred *Tocka3D*.

```
class Tocka //bazični razred
{
   //telo razreda Tocka
}
```

```
class Tocka3D: Tocka //razred Tocka3D podeduje razred Tocka
{
    //telo razreda Tocka3D
}
```

Iz razreda *Tocka* izpeljimo še dodatni razred *Daljica*

```
class Daljica:Tocka  //razred Daljica podeduje razred Tocka
{
    //telo razreda Daljica
}
```

Kreirajmo sedaj nekaj objektov:

```
Tocka A = new Tocka();
Tocka3D B = A;  //NAPAKA - različni tipi


Tocka3D C = new Tocka3D ();
Tocka D = C; //OK!
```

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. Če npr. napišemo razred *Tocka* in nato iz njega izpeljemo razred *Tocka3D*,

```
class Tocka   //bazni razred
{
     private int x, y;  //polji razreda Tocka

     public  void Izpis()  //metoda za izpis koordinat razreda Tocka
     {
          Console.WriteLine("Koordinate točke:\nx = " + x + "\ny = " + y);
     }
}

class Tocka3D : Tocka   //razred Tocka3D je izpeljan iz razreda Tocka
{
     private int z;      //dodatno polje razreda Tocka3D

     //Metoda Izpis prepiše istoimensko metodo razreda Tocka
     public void Izpis()
     {
         Console.Write("z = " + z + "\n");
     }
}
```

nam bo prevajalnik zgeneriral opozorilo, s katerim nas obvesti, da metoda *Tocka3D.Izpis* prekrije metodo *Tocka.Izpis*:

'ConsoleApplication1.Program.Tocka3D.Izpis()' hides inherited member 'ConsoleApplication1.Program.Tocka.Izpis()'. Use the new keyword if hiding was intended. Program.cs

Program se bo sicer prevedel in tudi zagnal, a opozorilo moramo vzeti resno. Če namreč napišemo razred, ki bo podedoval razred *Tocka3D*, bo uporabnik morda pričakoval, da se bo pri klicu metode *Izpis* pognala metoda bazičnega razreda, a v našem primeru se bo zagnala metoda razreda *Tocka3D*. Problem seveda lahko rešimo tako, da metodo *Izpis* v izpeljanem razredu preimenujemo (npr. Izpis1), še boljša rešitev pa je ta, da v izpeljanem razredu eksplicitno povemo, da gre za NOVO metodo – to storimo z uporabo operatorja *new*.

```
class Tocka   //bazni razred
```

```
{
     private int x, y;  //polji razreda Tocka

     public  void Izpis()  //metoda za izpis koordinat razreda Tocka
     {
          Console.WriteLine("Koordinate točke:\nx = " + x + "\ny = " + y);
     }
}

class Tocka3D : Tocka   //razred Tocka3D je izpeljan iz razreda Tocka
{
     private int z;      //dodatno polje razreda Tocka3D
     //Z operatorjem new napovemo, da ima razred Tocka3D SVOJO LASTNO
     //metodo Izpis
     new public void Izpis()
     {
         Console.Write("z = " + z + "\n");
     }
}
```

Zgled:

Napišimo razred *Krog* z zasebnim poljem *polmer* in lastnostmi *Premer*, *Obseg* in *Kvadratura*. Iz razreda nato izpeljimo razred *Krogla*, dodaj razredu lastno metodo *Kvadratura* in novo metodo *Volumen*.

```
class Krog  //Bazični razred
 {
     private double polmer;//zasebno polje razreda Krog

     public double Polmer  //lastnost za dostop do polja polmer
     {
        get
        {  if (polmer < 0)
             return 0.00;
           else
             return polmer;
        }
        set { polmer = value;}
        }
        public double Premer  //lastnost, ki vrne premer kroga
        {
            get {  return Polmer * 2;  }
        }
        public double Obseg  //lastnost, ki vrne obseg kroga
        {
            get  {  return Premer * Math.PI;  }
        }
        public double Kvadratura  //lastnost, ki vrne ploščino kroga
        {
            get  {  return Math.PI*Math.Pow(Polmer,2); }
        }
}
/*Kreirajmo sedaj razred Krogla ki naj podeduje razred Krog. Razred Krogla
bo podedoval polje polmer, podedoval bo lastnosti Premer in Obseg, imel bo
SVOJO lastnost za izračun kvadrature, poleg tega pa še novo lastnost za
izračun prostornine krogle */
class Krogla : Krog
{
     new public double Kvadratura  //z  operatorjem new smo označili, da
```

```
                        //ima razred krog SVOJO lastno lastnost kvadratura
    {
        get { return 4 * Math.PI*Math.Pow(Polmer,2); }
    }

    public double Volumen  //nova lastnost razreda Krogla
    {
        get {  return 4 * Math.PI * Math.Pow(Polmer,2) / 3; }
    }
}

//glavni program
static void Main()
{
     Krog c = new Krog();//nov objekt razreda Krog
     c.Polmer = 25.55;
     Console.WriteLine("Karakteristike kroga");
     Console.WriteLine("Polmer  : " + c.Polmer);
     Console.WriteLine("Premer  : " + c.Premer);
     Console.WriteLine("Obseg   : " + c.Obseg);
     Console.WriteLine("Ploščina: " + c.Kvadratura);

     Krogla s = new Krogla();//nov objekt razreda Krogla
     s.Polmer = 25.55;

     Console.WriteLine("\nKarakteristike krogle");
     Console.WriteLine("Polmer     : " + s.Polmer);
     Console.WriteLine("Premer     : " + s.Premer);
     Console.WriteLine("Obseg      : " + s.Obseg);
     Console.WriteLine("Površina   : " + s.Kvadratura);
     Console.WriteLine("Prostornina: " + s.Volumen);
}
```

Virtualne metode

Pogosto želimo metodo, ki smo je napisali v bazičnem razredu, v višjih (izpeljanih) razredih skriti in napisati novo metodo, ki pa bo imela enako ime in enake parametre. Eden izmed načinov je uporaba operatorja new za tako metodo, drug način pa je z uporabo rezervirane besede virtual. Metodo, za katero želimo že v bazičnem razredu označiti, da jo bomo lahko v nadrejenih razredih nadomestili z novo metodo (jo prekriti), označimo kot ***virtualno*** (*virtual*), npr.:

```
//virtualna metoda v bazičnem razredu - v izpeljanih razredih bo lahko
//prekrita(override)
public virtual void Koordinate()
{...}
```

V nadrejenem razredu moramo v takem primeru pri metodi z enakim imenom uporabiti rezervirano besedico override, s katero povemo, da bo ta metoda prekrila/prepisala bazično metodo z enakim imenom in enakimi parametri.

```
//izpeljani razred - besedica override pomeni, da smo s to metodo prekrili
//bazično metodo
public override void Koordinate()
{ ...}
```

Ločiti pa moramo razliko med tem, ali neka metoda prepiše bazično metodo (override) ali pa jo skrije. Prepisovanje metode (overriding) je mehanizem, kako izvesti drugo, novo implementacijo iste metode –

virtualne in override metode so si v tem primeru sorodne, saj se pričakuje, da bodo opravljale enako nalogo, a nad različnimi objekti (izpeljanimi iz bazičnih razredov, ali pa iz podedovanih razredov). Skrivanje metode (hiding) pa pomeni, da želimo neko metodo nadomestiti z drugo – metode v tem primeru niso povezane in lahko opravljajo povsem različne naloge.

Primer:

Naslednji primer prikazuje zapis virtualne metode *Koordinate*() v bazičnem razredu in *override* metode *Koordinate*() v izpeljanem razredu.

```
class Tocka   //bazični razred
{
    private int x, y;   //polji razreda Tocka

    //metoda Koordinate je virtualna, kar pomeni, da jo lahko prepišemo
    // (override)

    //metoda za izpis koordinat razreda Tocka
    public virtual void Koordinate()
    {
        Console.WriteLine( "Koordinate točke:\nx = "+x + "\ny = " + y);
    }
}

class Tocka3D : Tocka   //razred Tocka3D je izpeljan iz razreda Tocka
{
    private int z;      //dodatno polje razreda Tocka3D

    //Metoda Koordinate je označena kot override - prepiše istoimensko
    //metodo razreda Tocka
    public override void Koordinate()
    {
       base.Koordinate();  //klic bazične metode Koordinate razreda Tocka
       Console.Write("z = "+z+"\n");
    }
}
```

Prekrivne (Override) metode

Kadar je bazičnem razredu neka metoda označena kot virtualna (*virtual*), jo torej lahko v nadrejenih razredih prekrijemo/povozimo (override). Pri deklaraciji takih metod (pravimo jim **polimorfne** metode) z uporabo rezerviranih besed *virtual* in *override*, pa se moramo držati nekaterih pomembnih pravil:

- Metoda tipa *virtual* oz. *override* NE more biti zasebna (ne more biti *private*);
- Obe metodi, tako *virtualna* kot *override* morata biti identični: imeti morata enako ime, enako število in tip parametrov in enak tip vrednosti, ki jo vračata;
- Obe metodi morata imeti enak dostop. Če je npr. virtualna metoda označena kot javna (*public*), mora biti javna tudi metoda *override*;
- Prepišemo (prekrijemo/povozimo) lahko le virtualno metodo. Če metoda ni označena kot virtualna in bomo v nadrejenem razredu skušali narediti *override*, bomo dobili obvestilo o napaki;
- Če v nadrejenem razredu ne bomo uporabili besedice *override*, bazična metoda ne bo prekrita. To pa hkrati pomeni, da se bo tudi v izpeljanem razredu izvajala metoda bazičnega razreda in ne tista, ki smo napisali v izpeljanem razredu;
- Če neko metodo označimo kot *override*, jo lahko v nadrejenih razredih ponovno prekrijemo z novo metodo.

Zgled: razred Tocka

Razred *Tocka* in razred *Tocka3D*, ki je izpeljan iz razreda *Tocka* sta implementirana v naslednjem primeru:

```
class Tocka   //bazni razred
{
    private int x, y;         //polji razreda Tocka

    public Tocka(int x,int y) //konstruktor
    {
        this.x = x;
        this.y = y;
    }
    //metoda Koordinate je virtualna, kar pomeni, da jo lahko preobtežimo
    //(naredimo override)
    public virtual void Koordinate()//metoda za izpis koord. razreda Tocka
    {
        Console.WriteLine( "Koordinate točke:\nx = "+x + "\ny = " + y);
    }
}

class Tocka3D : Tocka   //razred Tocka3D je izpeljan iz razreda Tocka
{
    private int z;      //dodatno polje razreda Tocka3D
    //konstruktor - parametra x in y potrebujemo za dedovanje bazičnega
    //konstruktorja razreda Tocka
    public Tocka3D(int x,int y,int z)
         : base(x,y)
    {
       this.z = z;
    }
    //Metoda Koordinate prepiše istoimensko metodo razreda Tocka
    public override void Koordinate()
    {
       base.Koordinate();  //klic bazične metode Koordinate razreda Tocka
       Console.Write("z = "+z+"\n");
    }
}

static void Main(string[] args)
{
    Tocka A = new Tocka(1,1);   //Nov objekt razreda Tocka
    A.Koordinate();             //klic metode Koordinate razreda Tocka
    Tocka3D A3D = new Tocka3D(1, 2, 3); //Nov objekt razerda Tocka3D
    A3D.Koordinate();  //klic preobložene metode Koordinate razreda Tocka3D
}
```

Zgled: razred Kolobar deduje razred Krog

Napišimo razred *Krog* z zasebnim poljem polmer in virtualno metodo *ploscina*, nato pa še razred *Kolobar*. Razred *Kolobar* naj deduje razred K*rog*, dodano naj ima še eno zasebno polje *notranjiPolmer* in svojo lastno(override) metodo *ploščina*

```
class Krog   //bazni razred
{
    private int polmer;   //polje razreda Krog

    //metoda ploscina je virtualna, kar pomeni, da jo lahko preobtežimo
```

```
    public virtual double ploscina()
    {
        return Math.PI * polmer * polmer;
    }
   //lastnost razreda Krog, za dostop in inicializacijo polja polmer
    public int Polmer
    {
        get
        {
           return polmer;
        }
        set
        {
           polmer = value;
        }
    }
}

class Kolobar : Krog   //razred Kolobar podeduje razred Krog
{
    //ker Kolobar deduje Krog, že pozna polje polmer
    private int notranjiPolmer; //dodatno polje razreda Kolobar

    //metoda ploscina prepiše istoimensko metodo bazičnega razreda Krog
    public override double ploscina()
    {
        return Math.PI * (Polmer * Polmer - notranjiPolmer*notranjiPolmer);
    }

    //ker Kolobar deduje Krog, že pozna njegovo lastnost Polmer
    //dodatna lastnost (property) razreda Kolobar, za dostop in inic. Polja
    // notranjiPolmer
    public int NotranjiPolmer
    {
        get
        {
            return notranjiPolmer;
        }
        set
        {
            notranjiPolmer = value;
        }
    }
}

static void Main(string[] args)
{
    Random naklj = new Random();
    Krog k=new Krog();              //nov objekt razreda Krog
    k.Polmer = naklj.Next(1, 10);   //polje polmer inicializiramo preko
                                    //lastnosti Polmer

    Kolobar kol = new Kolobar();    //nov objekt razreda Kolobar
    //tudi polje polmer objekta kol inicializiramo preko lastnosti Polmer
    kol.Polmer = naklj.Next(1, 10);
    //polje notranjiPolmer inicializiramo preko lastnosti notranjiPolmer
    kol.NotranjiPolmer = naklj.Next(1, 10);

    //izpis ploščine kroga na 4 decimalke
    Console.WriteLine("Ploščina kroga: {0:F4}",k.ploscina());
```

```
    //izpis ploščine kolobarja na 4 decimalke
    Console.WriteLine("Ploščina kolobarja: {0:F4}", kol.ploscina());
}
```

Zgled: razred Kmetovalec deduje razred Oseba

Deklarirajmo razred *Oseba*, nato pa razred *Kmetovalec*, ki naj podeduje razred *Oseba*. Razred *Oseba* naj ima polje *ime*, razred *Kmetovalec* pa še dodatno polje *velikostPosesti*. Za oba razreda napišimo tudi konstruktor in virtualno metodo *ToString* za izpis podatkov o posameznem objektu!

```
public class Oseba             //bazični razred
{
    public string ime;         //bazično polje

    public Oseba(string ime)   //bazični konstruktor
    {
       this.ime = ime;
    }

    public virtual string ToString()  //virtualna metoda bazičnega polja
    {
        return "Ime=" + ime;
    }
}

public class Kmetovalec : Oseba  //razred Kmetovalec deduje razred Oseba
{
    int velikostPosesti;  //dodatno polje razreda Kmetovalec

    //konstruktor razreda Kmetovalec podeduje bazično polje ime
    public Kmetovalec(string ime, int velikostPosesti)
        : base(ime)
    {
        this.velikostPosesti = velikostPosesti;
    }
    //metoda ToString prekrije bazično metodo ToString
    public override string ToString()
    {
        return base.ToString() + "; Kvadratnih metorv = " +
        velikostPosesti.ToString();
    }
}

//glavni program
static void Main(string[] args)
{
    ArrayList osebe = new ArrayList();  //zbirka oseb

    Oseba mike = new Oseba("mike");
    osebe.Add(mike);
    Console.WriteLine("Osebe:");
    //izpišemo seznam vseh oseb
    foreach (Oseba p in osebe)
       Console.WriteLine(p.ToString());

    ArrayList kmetovalci = new ArrayList();  //zbirka kmetovalcev
    Kmetovalec john = new Kmetovalec("john", 10);
```

```
    Kmetovalec bob = new Kmetovalec("bob", 20);
    kmetovalci.Add(john);
    kmetovalci.Add(bob);
    Console.WriteLine();
    Console.WriteLine("Kmetovalci:");
    //izpišemo seznam vseh kmetovalcev
    foreach (Kmetovalec f in kmetovalci)
       Console.WriteLine(f.ToString());

    ArrayList vsiSkupaj = new ArrayList();  //zbirka vseh skupaj
    //napolnimo skupno zbirko vseh oseb
    foreach (Oseba o in osebe)
      vsiSkupaj.Add(o);
    foreach (Kmetovalec f in kmetovalci)
      vsiSkupaj.Add(f);
    Console.WriteLine();
    Console.WriteLine("Vsi skupaj:");
    //izpišemo seznam vseh oseb
    foreach (Oseba p in vsiSkupaj)
       Console.WriteLine(p.ToString());
}
```

Polimorfizem - mnogoličnost

V izpeljanih razredih se srečamo še z enim temeljnim pojmom objektno orientiranega programiranja – to je pojem **polimorfizem** oz. **mnogoličnost**. Pojem **polimorfizem** označuje princip, da lahko različni objekti razumejo isto sporočilo in se nanj odzovejo vsak na svoj način. Pomeni tudi, da je ista operacija lahko implementirana na več različnih načinov oz. zanjo obstaja več metod. Dedovanje in polimorfizem sta lastnosti, ki predstavljata osnovo objektnega programiranja in omogočata hitrejši razvoj in lažje vzdrževanje programske kode.

Kot primer za prikaz polimorfizma deklarirajmo razred *OsnovniRazred*, ki ima eno samo metodo z imenom *Slika*. Ker želimo, da bo vsak objekt, ki bo izpeljan iz tega razreda (tudi tisti v pododedovanih – izpeljanih razredih) ohranil sebi lastno obnašanje ob klicu metode *Slika*, moramo uporabiti **polimorfno redefinicijo**. Polimorfno redefinicijo omogočimo z že znano definicijo *virtualne metode*. V telesu te metode bomo za vajo in zaradi enostavnosti prikazali le neko sporočilno okno!

```
public class OsnovniRazred   //temeljni razred
{
   //polimorfna redefinicija - omogočimo jo z virtualno metodo
   public virtual void Slika()
   {
        Console.WriteLine("Osnovni objekt!");
   }
}
```

Ker smo metodo *Slika* definirali kot *virtualno*, smo s tem napovedali, da bodo izpeljani objekti lahko uporabljali svojo (prekrivno oz. override) metodo *Slika*, ki pa bo imela enako ime.

Razred *OsnovniRazred* bo naš temeljni razred, iz katerega bomo tvorili izpeljane razrede in v njih tvorili nove objekte. Ker je metoda *Slika* virtualna to pomeni, da lahko v izpeljanih razredih to metodo prekrijemo (*override*) z metodo, ki bo imela enako ime a drugačen pomen (drugačno vsebino).

Napišimo sedaj še tri razrede, ki naj bodo izpeljani iz razreda *OsnovniRazred* in ki imajo svojo metodo *Slika*. Pred tipom takih metod mora stati besedice override, ki označuje, da se bodo objekti izpeljani iz teh razredov na to metodo odzivali vsak na svoj način. Osnovni pogoj pa je, da imajo take override metode enako raven

zaščite (npr. vse so public) , enako ime in enake parametre kot jih ima osnovna virtualna metoda v bazičnem razredu.

```
//Crta je razred, ki podeduje razred OsnovniRazred
public class Crta : OsnovniRazred
{
   public override void Slika()    //preobložena metoda razreda Crta
   {
      //Telo override metode je seveda lahko drugačno!!!
      Console.WriteLine("Črta.");
   }
}

//Tudi Krog je razred, ki podeduje razred OsnovniRazred
public class Krog : OsnovniRazred
{
   public override void Slika()
   {
      //Telo override metode je seveda lahko drugačno!!!
      Console.WriteLine ("Krog.");
   }
}

//Tudi Kvadrat je razred, ki podeduje razred OsnovniRazred
public class Kvadrat: OsnovniRazred
{
   //Telo override metode je seveda spet lahko drugačno!!!
   public override void Slika()
   {
       Console.WriteLine ("Kvadrat.");
   }
}
```

Poglejmo sedaj, kako bi te štiri razrede sedaj uporabili in na primeru izpeljanih objektov prikazali princip polimorfizma. V ta namen kreirajmo tabelo objektov. Ime tabele je *dObj*, tabela pa naj bo inicializirana tako, da so v njej lahko štiri objekti tipa *OsnovniRazred*.

```
OsnovniRazred[] dObj = new OsnovniRazred[4];//tabela objektov
```

Ker so razredi *Crta*, *Krog* in *Kvadrat* izpeljani iz bazičnega razreda *OsnovniRazred*, jih lahko priredimo isti tabeli *dObj*. Če te zmožnosti ne bi bilo, bi morali za vsak nov objekt, izpeljan iz kateregakoli od teh štirih razredov, kreirati svojo tabelo. Dedovanje pa nam omogoča, da se vsak od izpeljanih objektov obnaša tako kot njegov bazični razred.

```
dObj[0] = new Crta();          //konstruktor objekta dObj[0]
dObj[1] = new Krog();          //konstruktor objekta dObj[1]
dObj[2] = new Kvadrat();       //konstruktor objekta dObj[2]
dObj[3] = new OsnovniRazred();//konstruktor objekta dObj[3]

foreach (OsnovniRazred objektZaRisanje in dObj)
{
     objektZaRisanje.Slika();  //klic metode Slika ustreznega objekta
}
```

Ko je tabela inicializirana, lahko npr. s *foreach* zanko pregledamo vsakega od objektov v tabeli. Zaradi načela polimorfizma se v zagnanem programu vsak objekt obnaša po svoje, pač odvisno od tega, iz katerega razreda je bil izpeljan. Ker smo v izpeljanih razredih prepisali virtualno metodo *Slika* , se ta metoda v izpeljanih objektih

izvaja različno, pač glede na njeno definicijo v izpeljanih razredih. Pri vsakem prehodu zanke bomo tako dobili drugačno sporočilno okno.

Uporaba označevalca protected

Uporaba označevalcev **private** in **public** predstavlja obe skrajni možnosti dostopa do članov razreda (oz. objekta). Javna polja in metode so dostopne vsem, zasebna polja in metode pa so dostopne le znotraj razreda. Pogosto pa je koristno, da bazični razred dovoljuje izpeljanim razredom, da le-ti dostopajo do nekaterih bazičnih članov, obenem pa ne dovoljujejo dostopa razredom, ki niso del iste hierarhije. V takem primeru uporabimo za dostop označevalec ***protected***.

Nadrejeni razred torej lahko dostopa do člana bazičnega razreda, ki je označen kot *protected*, kar praktično pomeni, da bazični član, ki je označen kot *protected*, v izpeljanem razredu postane javen (*public*). Javen je tudi v vseh višjih razredih.

V primeru, ko pa nek razred ni izpeljani razred, pa nima dostopa do članov razreda, ki so označeni kot *protected* – znotraj razreda, ki ni izpeljani razred, je torej član razreda, ki je označen kot *protected* enak zasebenemu članu (*private*).

Vaje

- Dan je razred *Zival*, ki vsebuje tri zasebna polja in metode za nastavljanje in spreminjanje le-teh. Vsebuje tudi konstruktor brez parametrov, ki postavi spremenljivke na smiselno začetno vrednost.

```
public class Zival
{
    //objektne spremenljivke
    private int steviloNog;
    private string vrsta;
    private string ime;

    /*Konstruktor - ustvari objekt Zival s stirimi nogami vrste pes
        in imenom Pika.*/
    public Zival()
    {
        steviloNog = 4;
        vrsta = "pes";
        ime = "Pika";
    }

    //Metoda, ki vrne ime živali.
    public string VrniIme()
    {
        return this.ime;
    }

    // Metoda, ki vrne vrsto živali.
    public string VrniVrsto()
    {
        return this.vrsta;
    }
    /* Metoda, ki vrne število nog živali.
```

```
       Zagotovljeno je, da bo število nog vedno vecje od 0. */
    public int VrniSteviloNog()
    {
        return this.steviloNog;
    }
    /*Metoda, ki nastavi ime živali.*/
    public void NastaviIme(string vrednost)
    {
        if (vrednost != null)
        {
            this.ime = vrednost;
        }
    }
    /* Metoda, ki nastavi vrsto živali.*/
    public void NastaviVrsto(string vrednost)
    {
        if (vrednost != null)
        {
            this.vrsta = vrednost;
        }
    }
    /* Metoda, ki nastavi število nog živali.*/
    public void NastaviSteviloNog(int vrednost)
    {
        if (vrednost > 0)
        {
            this.steviloNog = vrednost;
        }
    }
}
```

Dopolnite razred z metodo *override public String ToString()*, ki naj izpiše podatke o objektih v obliki: *#Ime*: je vrste *#vrsta* in ima *#steviloNog* nog. Pri tem seveda zamenjajte *#vrsta* in *#steviloNog* z vrednostmi v istoimenskih spremenljivkah.

Nato napišite testni program, kjer ustvarite 5 živali in jim nastavite smiselno vrsto in imena ter število nog. Izpišite jih!

- Sestavi razred *Kolega*, ki ima tri komponente: ime, priimek in telefonska številka. Vse tri komponente naj bodo tipa string in javno dostopne. Napiši vsaj dva konstruktorja: prazen konstruktor, ki vse tri komponente nastavi na "NI PODATKOV" in konstruktor, ki sprejme vse tri podatke in ustrezno nastavi komponente. Napiši tudi metodo *public string ToString()*, ki vrne niz s smiselnim izpisom podatkov o objektu tipa Kolega (ime, priimek in telefonska številka). Sestavi testni program, v katerem ustvariš dva objekta tipa *Kolega*. En objekt ustvari s praznim konstruktorjem, drugega s konstruktorjem, ki sprejme tri parametre. Oba objekta tudi izpiši na zaslon. Napiši program, ki sestavi tabelo 10ih objektov tipa *Kolega*, prebere telefonsko številko in izpiše tiste objekte iz tabele, ki imajo tako telefonsko številko. Če takega objekta v tabeli ni, naj se izpiše: "Noben moj kolega nima take telefonske številke."

- V spodnji kodi je napaka. Poišči jo in jo odpravi (brez uporabe računalnika).

```
public class X
{
  public int a = 0;
  public int b = 2;
```

```
  override public ToString()
  {
     return "a = " + a + " b = " + b;
  }
}
```

- Dana je koda:

```
public class Y
{
    private int x;
    private int y;

    public Y(int row, int col)
    {
        x = row;
        y = col;
    }
}
```

Kateri izmed spodnjih konstruktorjev je pravilen:

a) `y1 = new Y(2,3);`
b) `y2 = new Y(2);`
c) `y3 = new Y;`
d) `y4 = new Y();`
e) `y5 = new Y(10.0, 3);`
f) `y6 = new Y[5];`

Razredu Y dodamo še konstruktor

```
public Y(int row)
{
  x = row;
  y = 80;
}
```

Kateri izmed konstruktorjev pa so sedaj pravilni?

a) `y1 = new Y(2,3);`
b) `y2 = new Y(2);`
c) `y3 = new Y;`
d) `y4 = new Y();`
e) `y5 = new Y(10.0, 3);`
f) `y6 = new Y[5];`

- Dana je koda:

```
public class Z
{
      private int a = 0;
      private int b =0;

      public Z(int aa, int bb)
      {
            a = aa;
            b = bb;
      }
}
```

Ustvarimo dva objekta tipa `Z`, in sicer `z1 = new Z(5, 4)` ter `z2 = new Z(5, 4)`.

- Kakšno vrednost nam vrne test z1 == z2 ? Trditev obrazloži.
- Kako »pravilno« preveriti, če sta dva objekta »enaka« ? Namig: metoda *equals*.
- Ali lahko metodo »*equals*« takoj uporabiš v zgornji kodi, ali moraš razred kaj spremeniti?

Napiši razred **Kosarka**, za spremljanje košarkaške tekme. Voditi moraš število prekrškov za vsakega tekmovalca (10 igralcev), število doseženih točk (posebej 1 točka, 2 točki in 3 točke), ter metodo za izpis statistike tekme. Doseganje košev in prekrškov realiziraj preko metode *zadelProstiMet*(), *zadelZa2Tocki*, *zadelZa3Tocke* in *prekrsek*.

Sestavi razred, ki bo v svoja polja lahko shranil ulico, številko nepremičnine ter vrsto nepremičnine. Ustvari poljuben objekt, ga inicializiraj in ustvari izpis, ki naj zgleda približno takole: Na naslovu Cankarjeva ulica 32, Kranj je blok.

Dana je koda razreda *Povecaj*:

```
public class PovecajR
{
	public void Povecaj(int a)
	{
		a++;
	}
	public void Povecaj(int[] a)
	{
		for(int i =0; i < a.Length; i++)
		a[i] = a[i] + 2;
	}
}
```

Ali je v kodi kakšna napaka? Če je, jo odpravi! Kaj izpiše sledeča koda?

```
PovecajR inc = new PovecajR();

int a = 5;
int[] b = {5};

inc.Povecaj (a);
inc.Povecaj (b);
Console.WriteLine("a = " + a);
Console.WriteLine("b = " + b[0]);
```

Sestavi razred *Datum*, ki predstavlja datum (dan, mesec in leto). Definiraj ustrezne komponente in konstruktorje. Definiraj metodo *override public string ToString()*, ki vrne datum predstavljen z nizom znakov.

Definiraj še metodo *public bool Equals(Datum d)*, ki vrne *True*, če je datum *this* enak datumu *d*. Razredu *Datum* dodaj metodo *public int CompareTo(Datum d)*, ki primerja *this* in *d* ter vrne celo število:

- < 0, če je *this* pred *d*
- 0, če je *this* enak *d*
- > 0, če je this za *d*

Sestavite razred Majica, ki hrani osnovne podatke o majici: velikost (število med 1 in 5), barvo (poljuben niz) in ali ima majica kratke ali dolge rokave. Vse spremenljivke razreda morajo biti privatne, razred pa mora tudi poznati metode za nastavljanje in branje teh vrednosti. Podpisi teh metod naj bodo:

- public int *VrniVelikost()* ter public void *SpremeniVelikost(int velikost)*
- public string *VrniBarvo()* ter public void *SpremeniBarvo(string barva)*
- public boolean *ImaKratkeRokave()* ter *public void NastaviKratkeRokave(boolean imaKratkeRokave)*

Metoda za nastavljanje velikosti majice naj poskrbi tudi za to, da bo velikost vedno ostala znotraj sprejemljivih meja! Če uporabnik poskusi določiti napačno velikost, naj se obdrži prejšnja vrednost.

Kandidat za direktorja banke: v banki je direktor uspel povzročiti politični škandal takšnih razsežnosti, da je moral odstopiti (seveda z odpravnino v višini 50 povprečnih plač) in zato sedaj iščejo novega direktorja. Za to službo se je prijavilo kar nekaj kandidatov, od katerih je vsak predložil sledeče podatke: ime, priimek, starost (v letih) in stopnjo izobrazbe - številka med 3 in 8 (vključno). Banka želi izbrati direktorja, ki ne bi bil starejši od 50 let (da bo lepo izgledal v medijih) in ima vsaj 6. stopnjo izobrazbe. Tvoja naloga je:sestaviti razred, ki bo hranil podatke o kandidatih za direktorja in dopolniti priložen testni program tako, da bo našel primerne kandidate za to mesto in jih izpisal.
Razred Kandidat mora poleg ustreznih spremenljivk (ki naj bodo privatne!) vsebovati vsaj javne metode:

- *public Kandidat ()* - konstruktor brez parametrov - ustvari kandidata s starostjo 20, stopnjo izobrazbe 6 in imenom in priimkom Neznani Neznanec.
- *public void NastaviImePriimek(string ime, string priimek)* - metoda, ki nastavi ime in priimek kandidata. Če je slučajno ime oz. priimek enak null, naj starega imena in priimka ne spremeni.
- *public string[] VrniImePriimek()* - metoda, ki vrne ime in priimek kandidata. Metoda naj vrne seznam nizov dolžine 2 - v prvem polju naj bo zapisano ime, v drugem pa priimek kandidata.
- *public void NastaviStarost(int starost)* - metoda, ki nastavi starost kandidata. Starost mora biti v mejah med 20 in 80 let - če ni, naj se stara starost ne spremeni.
- *public int VrniStarost()* - metoda, ki vrne starost kandidata.
- *public void NastaviIzobrazbo(int izobrazba)* - metoda, ki nastavi izobrazbo kandidata. Izobrazba mora biti v mejah med 3 in 8 - če ni, naj se stara izobrazba ne spremeni.
- *public int VrniIzobrazbo()* - metoda, ki vrne izobrazbo kandidata.
- public string ToString()

Dano je okostje programa, ki obravnava prijavljene kandidate.

```
public static Kandidat NajPrimernejši(Kandidat[] tabelaKandid)
{
      // tu poskrbi, da bodo v ime, priimek, ...
    // prišli podatki o najprimern. kandidatu
      string ime = "Moj";
      string priimek = "Prijatelj";
      int starost = 42;
      int izobrazba = 6;
      Kandidat nov = new Kandidat();
      nov.NastaviImePriimek(ime, priimek);
      nov.NastaviIzobrazbo(izobrazba);
      nov.NastaviStarost(starost);
      return nov;
}
public static void Main(string[] args)
{
    public string[] imena = {"Jana", "Jure", "Bernarda",
            "Anže", "Danijel", "Primož", "Anita", "Tina", "Matija"};
    public string[] priimki = {"Mali", "Ankimer", "Zupan",
            "Zevnik", "Bogataj", "Koci", "Vilis", "Jerše", "Lokar"};
    int kolikoKandidatov = 30;
    public Kandidat[] kandidati = new Kandidati[kolikoKandidatov];
    Random gen = new Random();

    for(int i=0; i < kolikoKandidatov; i++)
```

```
    {
            Kandidat tmp = new Kandidat();
            int randime = gen.Next(0, imena.Length - 1);
            int randpriimek = gen.Next(0,priimki.Length - 1);
            int randizobrazba = gen.Next(3,8);
            int randstarost = gen.Next(20,90);
            tmp.NastaviImePriimek(randime, randpriimek);
            tmp.NastaviIzobrazbo(randizobrazba);
            tmp.NastaviStarost(randstarost);
            kandidati[i] = tmp;
    }
    // tu poišči najprimernejšega kandidata in ga izpiši
    Console.WriteLine("Najprimernejsi je: " + NajPrimernejsi(kandidati));
    Console.Write("Press any key to continue . . . ");
    Console.ReadKey(true);
}
```

Spremeni metodo *NajPrimernejsi,* tako, da vrne kandidata, ki je najmlajši in ima zahtevano stopnjo izobrazbe (ali višjo). Recimo, da so si v banki premislili in bi radi namesto mladega neizkušenega direktorja zaposlili najstarejšega (ne glede na izobrazbo). Poišči ga!

Sestavi razred *Denarnica*, ki vsebuje celoštevilsko spremenljivko, ki predstavlja količino denarja v njej kot javno spremenljivko (v centih) in metodo public string *ToString()*, ki vrne niz "V denarnici je # centov". Ustvari tudi tabelo desetih denarnic z naključno mnogo denarja in jih izpiši.

- Ali lahko na ta način zagotoviš, da v denarnici nikoli ne bo negativno mnogo denarja?
- Seveda ne. To bi se na primer zgodilo, če bi programer s firme Nekaj pacamo d.o.o. po pomoti vstavil v spremenljivko, ki predstavlja količino denarja, negativno vrednost. Zato moramo razred popraviti! Namesto javne spremenljivke vzemimo privatno, poleg tega pa dopišimo še metodi dvigni in placaj, ki v denarnico dodata oz. iz nje odvzameta denar. Če želimo plačati več, kot imamo denarja v denarnici, naj metoda placaj vrže napako!

  Opomba: namesto teh dveh metod bi lahko napisali samo t.i. "set" metodo: *public void SetVrednost(int vrednost)*, ki količino denarja v denarnici nastavi na vrednost (in pri tem seveda pregleda, če je ta vrednost pozitivna). Ampak glede na naš razred je izbira dveh metod bolj naravna.
- Sestavi tudi testni program, kjer preizkusiš delovanje tojega razreda. V denarnico dodaj nekaj denarja, nekaj ga odvzemi in se prepričaj, da je v njej pravilna količina denarja. Preveri tudi plačilo prevelikega zneska in ustrezno odreagiraj!
- Ali prejšnjo nalogo sploh lahko v celoti izpolniš? Denar res lahko dodaš in ga odvzameš, ker pa je količina denarja privatna spremenljivka razreda, je od zunaj nikakor ne moreš prebrati! Ta problem se reši tako, da v razred dodaš javno metodo public int KolicinaDenarja(), ki nam vrne količino denarja v denarnici.
- Dopolni razred s konstruktorjem public Denarnica(int denar), ki ustvari denarnico z dano količino denarja. S pomočjo metode KolicinaDenarja preveri, če se je tvoja denarnica pravilno inicializirala. Ustvari tudi denarnico z negativno mnogo denarja in preveri, če metode razreda Denarnica v tem primeru vržejo napako. Ne pozabi je tudi ujeti v glavnem programu, sicer se ti bo program sesul!
- Denarnica ima tudi svojega lastnika. Dodaj privatno spremenljivko lastnik tipa string, ki vsebuje ime lastnika denarnice. Dopolni konstruktor tako, da poleg količine denarja sprejme tudi ime lastnika. Razredu dodaj tudi metodo VrniLastnik, ki vrne ime lastnika denarnice. Popravi tudi metodo ToString, tako da izpis vključuje tudi ime lastnika denarnice.
- Dopolni testni program tako, da ustvariš tabelo desetih denarnic z naključno količino denarja med 0 in 1000. Izpiši jih!
- Kdo izmed lastnikov si lahko privošči nakup banjice sladoleda, ki stane 800 centov?

Sestavite razred *PodatkovniNosilec*, ki predstavlja medij za shranjevanje podakov. Vsebuje naj informacije o kapaciteti medija (celo število, enota je bajt), izdelovalcu (niz), osnovni ceni (celo število, enota je evroCent) in stopnji obdavčitve (celo število, enota je procent). Vsebovati mora vsaj javne metode:

- *public PodatkovniNosilec()* - konstruktor, ki ustvari nosilec kapacitete 650Mb, proizvajalca "neznan", s ceno 100 centov in 20% davkom.
- *public PodatkovniNosilec(string ime, int kapaciteta, int cena, int davek)* - konstruktor s parametri.
- *public void NastaviKapaciteto(int bajti)* - nastavi kapaciteto medija v bajtih. Če je kapaciteta negativna, naj se ne spremeni.
- *public void NastaviIme(string ime)* - nastavi ime izdelovalca medija. Če je ime enako null, naj se ne spremeni.
- *public void NastaviCeno(int cena)* - nastavi ceno. Če je negativna, naj se ne spremeni.
- *public void NastaviObdavcitev(int pocent)* - nastavi obdavčitev v procentih. Če so procenti izven meja 0-100, naj se ne stara vrednost ne spremeni.
- *public int VrniOsnovnoCeno()* - vrne osnovno ceno medija (brez davka).
- *public double VrniDavek()* - koliko davka je potrebno plačati za ta medij (v evrih).
- *public double VrniProdajnoCeno()* - vrne osnovno ceno + davek (v evrih).
- *public string VrniIme()* - vrne ime proizvajalca medija.
- *public double VrniKapaciteto(char enota)* - vrni kapaciteto v enoti, ki jo določa znak enota:
  - 'b' - bajti
  - 'k' - kilobajti
  - 'M' - megabajti
  - 'G' - gigabajti

  Pazi na to, da je en kilobajt 1024 (in ne 1000) bajtov!
- *public override string ToString()* - vrne niz, ki smiselno opisuje razred.
- Sestavi tudi testni program, v katerem preveriš delovanje tvojega razreda.

Podana je koda razreda *Ulomek*. Dopolni manjkajoče metode ter v glavnem programu (metoda *Main*) ustvari dva ulomka, katerih števec in imenovalec naj bo naključno število med med 1 in 9 (vključno z mejama). Ulomka seštej in rezultat izpiši na zaslon.

```
public class Ulomek
{
  int stevec;
  int imenovalec;

  // Privzeti konstruktor ustvari ulomek 1/1
  // DOPOLNI!


  // Ustvari podani ulomek. Pazi tudi na pravilnost podatkov. Če so
  //napačni, ustvari enak ulomek kot v privzetem konstruktorju.
  public Ulomek(int stevec, int imenovalec)
  {
    // DOPOLNI!
  }

  // Metoda prišteje ulomek u k trenutnemu ulomku.
  public void Pristej(Ulomek u)
  {
    // DOPOLNI!
  }

  /*Metoda zmnoži trenutni ulomek z ulomkom u ter zmnožek vrne kot
  rezultat.Trenutni ulomek ter ulomek u se pri tem ne smeta spremeniti*/
```

```
  public DOPOLNI Zmnozi(Ulomek u)
  {
    // DOPOLNI!
  }
  // Metoda vrne vrednost ulomka kot realno število.
  public DOPOLNI Vrednost()
  {
    // DOPOLNI!
  }

  // Vrne niz oblike »stevec/imenovalec«
  DOPOLNI string ToString()
  {
    // DOPOLNI!
  }

  // Metoda okrajša dani ulomek
  public void Okrajsaj()
  {
    // DOPOLNI
  }

  // Obratna vrednost.
  public void Obrni()
  {
  // DOPOLNI
  }
}
```

Na osnovi kode spodaj podanega razreda *Oseba* sestavite razred *Stranka*, ki predstavlja bančno stranko tega komitenta. Poleg lastnosti, so enake kot v razredu *Oseba*, mora hraniti tudi podatke o tekočem računu (niz), številki EMŠO (niz) ter povprečnem mesečnem prilivu na tekoči račun (realno število).

```
public class Oseba
{
    private string ime;
    private string priimek;

    public Oseba(string ime, string priimek)
    {
        this.ime = ime;
        this.priimek = priimek;
    }
    public string VrniIme()
    {
        return this.ime;
    }
    public string VrniPriimek()
    {
        return this.priimek;
    }
}
```

Pri tem mora razred zadoščati naslednjim pogojem:

- vse spremenljivke v njem morajo biti privatne

- Poznati mora javni konstruktor, ki sprejme pet parametrov: ime, priimek, številko tekočega računa, številko EMŠO ter podatek o povprečnem mesečnem prilivu na račun. Pri tem mora biti zadnji podatek pozitivni+o realno število, prvi štirje pa neprazni nizi. Če to ni izpolnjeno, ustrezno reagiraj.
- Poznati mora metode za vračanje podatkov, ki jih hrani: imena, priimka, številke tekočega računa, številke EMŠO ter povprečnega priliva na račun.
- Metoto ToString, ki na smiselen način izpiše podatke o stranki.

Nazadnje sestavite testni program, kjer ustvarite tabelo desetih strank z izmišljenimi podatki, ter med njimi poščite stranko z najvišjim mesečnim prilivom na račun ter tistega z po leksikografski urejenosti prvim imenom (če jih je več z enakim največjim prilivom ali enakim prvim imenom poiščite kateregakoli) ter izpišite njune podatke na zaslon (z uporabo metode ToString).

Dalmatinci I (brez računalnika). Denimo, da smo želeli sestaviti razred Dalmatinec, ki ima lastnosti ime psa in število njegovih pik. Koda razreda je:

```
public class Dalmatinec
{
    public string ime;
    private int steviloPik;

    public Dalmatinec()
    {
        this.ime = "Reks";
        this.steviloPik = 0;
    }

    public void ImePsa(string ime)
    {
        ime = this.ime;
    }

    private void NastaviIme(string ime)
    {
        this.ime = ime;
    }

    public void NastaviSteviloPik(int steviloPik)
    {
        this.steviloPik = steviloPik;
    }
}
```

V glavnem programu smo ustvarili objekt *Dalmatinec* z imenom *d* in mu želimo nastaviti število pik na 100 in ime na *Pika*. Kateri način je pravilen? Pri nepravilnih povej, kaj in zakaj ni pravilno.

```
a) d.NastaviIme("Pika"); d.NastaviSteviloPik(100);
b) d.ime = "Pika"; d.steviloPik = 100;
c) d.ime = "Pika"; d.NastaviSteviloPik(100);
d) d.imePsa("Pika"); d.NastaviSteviloPik(100);
e) d.imePsa("Pika"); d.steviloPik = 100;
f) d.NastaviIme("Pika"); d.steviloPik = 100;
g) nobeden, ker tega sploh ne moremo storiti
```

Sedaj želimo našemu razredu Dalmatinec dodati tudi podatke o spolu psa. Ta podatek bomo hranili v spremenljivki spol. Interno (znotraj razreda) naj logična vrednost true pomeni ženski, false pa moški spol. Dopolnite razred tako, da bo zadoščal naslednjim trem pogojem:

- Spremenljivka spol naj ne bo dostopna izven razreda *Dalmatinec*

- obstaja naj metoda *KaksenSpol*, ki v primeru samca vrne 'm', v primeru samice pa 'f'.
- spol se nastavi le ob ustvarjanju objekta. Morali boste torej napisati konstruktor razreda *Dalmatinec*, ki sprejme kot parameter znak za spol. Ta naj bo kot zgoraj 'm' za samca in 'f' za samico. Predpostavite, da bo parameter zagotovo znak 'm' ali 'f'.

Sestavi še metodo *public static int SteviloSamcev(Dalmatinec[] dalmatinci)*, ki prešteje število samcev v tabeli dalmatincev.

Potrebujemo razred, ki bo hranil podatke o objektih tipa *Instrukcije* z naslednjimi komponentami: vrsta inštrukcije, število opravljenih ur in ali je inštrukcija možna. Razred naj ima tudi metode in sicer: inštrukcija se opravlja, inštrukcija se ne opravlja. Z osnovnim konstruktorjem določi vrednost (poljubno) vsem komponentam razreda. Z dodatnim konstruktorjem poskrbi za možnost nastavitve začetne vrednosti za vrsto inštrukcije, opravljene ure ter ali je inštrukcija možna.
Na osnovi razreda *Instrukcije* napiši še testni program, ki bo kreiral in izpisal dva objekta razreda *Instrukcije* in sicer tako, da bodo začetne vrednosti pri prvem objektu nastavljene z osnovnim konstruktorjem, pri drugem objektu pa z dodatnim konstruktorjem.

Sestavi razred, ki predstavlja osnovo za izdelavo programa, s pomočjo katerega bomo pregledovali rezultate nekega športnega tekmovanja. Sestavi razred *Tekmovalec*, ki ima naslednje komponente: startno številko, ime, priimek in klub. Vsa polja so tipa string. Napiši vsaj dva konstruktorja in pripravi ustrezne *get/set* metode za vse te podatke. Z metodo *public String toString()* naj se izpišejo podatki o tekmovalcih (startna številka, ime, priimek, klub).

Sestavi razred *Pacient*, ki ima tri komponente: *ime*, *priimek*, *krvna_skupina*. Komponenti *ime* in *priimek* naj bosta *public*, *krvna_skupina private*, vse tri pa tipa *string*. Napiši vsaj dva konstruktorja:

- prazen konstruktor, ki vse tri komponente nastavi na "NI PODATKOV",
- konstruktor, ki sprejme vse tri podatke in ustrezno nastavi komponente.

Z metodo *public String toString()* naj se izpišejo podatki o pacientu (ime, priimek, krvna skupina).

Sestavi razred *Piramida*, ki predstavlja pokončno piramido, ki ima za osnovno ploskev kvadrat. V razredu hrani podatke o dolžini stranice osnovne ploskve in višino piramide. Obe komponenti naj imata tip dostopa **private**. Višina in stranica nista nujno celi števili. Napiši vsaj dva konstruktorja:

- prazen konstruktor, ki ustvari piramido višine 1 in s stranico osnovne ploskve dolžine 1
- konstruktor, ki sprejme podatka o višini in dolžini stranice osnovne ploskve
- Napiši program, ki ustvari tabelo 50 naključnih piramid in med njimi poišče piramido z največjo višino.

Napiši **get** in **set** metode. V **set** metodah pazi na smiselnost podatkov (višina in dolžina stranice ne smeta biti negativni,...). Napiši tudi metodo *toString*, ki vrne niz s smiselnim izpisom podatkov o piramidi.

Sestavi razred, kjer boš v objektih te vrste hranil ime države, njeno glavno mesto, površino (v km^2) in število prebivalcev. Pripravi ustrezne *get/set* metode za vse te podatke in vsaj dva konstruktorja. Ne pozabi tudi na smiselno metodo toString.

Sestavi razred *Igrača*, ki ima tri privatne spremenljivke: tip igrače(tip *string*), število igrač na zalogi(tip *int*) ter ceno igrače(tip *double*).

- sestavi prazen konstruktor;
- sestavi konstruktor s tremi parametri;
- napiši ustrezne *set* in *get* metode (pazi na smiselne vrednosti spremenljivk);
- napiši metodo *toString*, ki naj smiselno izpiše, kateri tip igrače, koliko kosov je na zalogi in kakšna je cena;

- dodaj še metodo *zalogaIgrac*, ki sprejme pozitivno celo število, če se je število igrač povečalo (dobava novih) ter negativno celo število če se je število igrač zmanjšalo (prodane igrače). Temu primerno spremeni število igrač na zalogi in pri tem pazi, da število igrač ne pade pod nič;
- napiši metodo *znizanaCena*, ki sprejme celo število med 0 in 100 (%), to je za koliko procentov se bo znižala cena igrače, in nastavi novo ceno igrače;

napiši glavni program, ki bo ustvaril pet tipov igrač, tri izmed njih znižal za 10, 20 in 50% ter dvema spremenil zalogo.

V kemijskem laboratoriju večkrat preverjamo kislost snovi in jo definiramo s *pH* vrednostjo. Napiši razred *Snov*, ki bo imel tri komponente: *imeSnovi*, *ion* in *kislost*. Prvi dve sta tipa *string*, tretja pa *int*. Ker je kislost zelo pomemben podatek o snovi, zagotovi, da bo zagotovo pravilen. Zato bo ustrezna spremenljivka imela privaten dostop. Sestavi tudi ustrezni *get* in *set* metodi. Pazi, da boš tudi v konstruktorju s parametri poskrbel, da ne bo prišlo do napačne nastavitve. Če bo uporabnik podal napačno kislost, ustvari objekt, kjer za prva dva podatka napišeš, da sta neobstoječa, kislino pa pustiš tako, kot jo je vnesel uporabnik. Sestavi tudi konstruktor brez parametrov, ki naredi objekt, ki predstavlja vodo. Kreiraj tabelo snovi in napiši stavke za izpis vseh objektov.

Napiši razred *Kocka* tako, da bo izpeljan iz razreda *Kvadrat*. Razreda naj poleg svojih podatkov vsebujeta še privzeti konstruktor, metode za postavitev vrednosti podatkov in metode za izračun površine, obsega in volumna.

Sklad in kopica

Pomnilnik, ki je namenjen spremenljivkam, je razdeljen na dva dela: **kopica** (*heap*) in **sklad** (*stack*).

V **sklad** se shranjujejo t.i. **vrednostne** spremenljivke (**value type**). To so spremenljivke tipa **bool**, char, **int**, **float**, double, **decimal**, **struct**, **enum** in spremenljivke, ki se tvorijo ob klicu funkcije. Na skladu se hranijo tudi parametri funkcij, ki se obnašajo tako kot spremenljivke, ki se tvorijo ob klicu funkcij. Pri klicu parametrov ***po vrednosti***, se spremenljivka, ki nastopa kot parameter funkcije prepiše (prekopira) na sklad in v funkciji delamo v bistvu z njeno kopijo. Ob klicih funkcij se torej sklad povečuje, ob zaključku izvajanja funkcij pa se zmanjšuje. Za ponazoritev sklada lahko vzamemo tudi nekaj primerov iz narave: skladovnica knjig (ena knjiga na drugi), PEZ bonboni, … Elemente vstavljamo (**push**) in tudi jemljemo (**pop**) s sklada na enem (istem) koncu. Njihova značilnost je dejstvo, da gre element, ki ga vstavimo v tak sklad prvega ven zadnji. Tak princip imenujemo **LIFO** (**Last In First Out**). Primeri implementacije sklada v računalništvu pa so npr.: obiskane strani v brskalniku, zaporedje UNDO ukazov, rekurzija, …

Sklad običajno imenujemo tudi del pomnilnika, kjer so shranjeni podatki. Običajno dostopamo samo do zadnjega, vrhnjega podatka. Najbolj značilni operaciji za sklad sta

- Postavi na sklad (**push**)
- Vzemi s sklada (**pop**)

V **kopico** pa se zlagajo t.i. **referenčne** spremenljivke (**reference type**), to je spremenljivke ki imajo poleg svoje vrednosti tudi referenco oz kazalec nanjo. To so npr. vse tabelarične, vsi iz razredov izpeljani objekti, … Dva podatka na kopici imata torej lahko isto referenco in operacija na referenci lahko vpliva na več podatkov. To so v bistvu spremenljivke, ki jih tvorimo izrecno. Kopica je torej namenjena vsem dinamično kreiranim spremenljivkam, ki jih tvorimo z ukazom ***new***. **Vse take spremenljivke se samodejno inicializirajo** (takoj, ko se pojavijo, dobijo neko začetno vrednost – pri tabeli celih števil je ta začetna vrednost npr. enaka 0). Značilnost kopice je, da je to podatkovna struktura v katero vstavljamo podatke na repu, brišemo oz. jemljemo pa jih s čela. Zanjo velja princip **FIFO** (**First In First Out**). Spremenljivkam, ki jih ustvarimo s pomočjo operatorja **new** pravimo tudi **objekti**.

Pomen sklada lahko ponazorimo tudi pri funkcijah: pri klicu parametrov ***po referenci*** pa se spremenljivka, ki nastopa kot parameter funkcije prenese v funkcijo in v funkciji dejansko delamo prav s to spremenljivko. Sam prenos je v tem primeru realiziran tako, da se tokrat na sklad prenese **referenca** na spremenljivko in funkcija potem ve, da ima opravka z referenco, ne pa z vrednostjo.

Zbirke - Collections

Podobno kot tabele, je tudi **zbirka** (**collection**) objekt, ki lahko vsebuje enega ali pa več elementov. Za razliko od tabel, kjer moramo že ob definiciji napovedati njeno dolžino, pa **zbirka NIMA** fiksne dolžine. Zbirke so torej lahko spremenljive dolžine.

Zbirke so lahko **netipizirane** (vsebujejo elemente različnih tipov) ali pa **tipizirane**: vsi elementi zbirke so enakega tipa.

Netipizirane zbirke

Pred deklaracijo **netipizirane** zbirke moramo poskrbeti za vključitev ustreznega imenskega prostora

```
using System.Collections;
```

Splošna deklaracija zbirke:

```
ArrayList ime_zbirke = new ArrayList();
```

Elemente dodajamo v zbirko s pomočjo metode Add().

```
Ime_zbirke.Add(7); //v zbirko smo dodali nov element - tip elementa ni pomemben
```

Do posameznih elementov zbirke dostopamo preko indeksa (tako kot pri tabelaričnih spremenljivkah). Začetni indeks je enak 0! Za razliko od tabelaričnih spremenljivk pa preko indeksa elementa zbirke ni dovoljena kar poljubna aritmetika!

Primer napačnega prirejanja in pravilnega prirejanja:

```
Ime_zbirke[1]=Ime_zbirke[0]+100;     //NAPAKA!!!!!!!! - aritmetični operatorji niso dovoljeni
Ime_zbirke[1]=Ime_zbirke[0];         //OK
Ime_zbirke[1]= "Danes je lep dan! "; //OK
```

Primer:

```
ArrayList stevila = new ArrayList();
stevila.Add(3);
stevila.Add(7);
stevila.Add("test"); //prevajanje normalno, a pri uporabi v nadaljevanju (for zanka) pride do
                     //napake
int vsota=0;

//Count je metoda zbirke, ki vrne trenutno število elementov zbirke
for (int i=0;i<stevila.Count;i++)
{
    int stevilo=(int)stevila[i];//potrebna eksplicitna konverzija (int)
    vsota+=stevilo;
}
```

Še en primer:

```
// POZOR, preveri, če si vključil imenski prostor: using System.Collections;
ArrayList al = new ArrayList();
```

```
Console.WriteLine("Začetna kapaciteta zbirke: " + al.Capacity);
Console.WriteLine("Začetno število elementov zbirke: " + al.Count);

Console.WriteLine();

Console.WriteLine("Dodajmo 6 elementov");
// Dodajanje elementov v zbirko
al.Add('C');
al.Add('A');
al.Add('E');
al.Add('B');
al.Add('D');
al.Add('F');

Console.WriteLine("Trenutna kapaciteta zbirke: " + al.Capacity);
Console.WriteLine("Trenutno število elementov zbirke: " + al.Count);

//Izpis elementov zbirke s pomočjo indeksa posameznega elementa zbirke.
Console.Write("Trenutna vsebina zbirke (izpis s pomočjo zanke for: ");
for (int i = 0; i < al.Count; i++)
     Console.Write(al[i] + " ");
Console.WriteLine("\n");

Console.WriteLine("Odstanimo 2 elementa");
// Odstranjevanje elementov iz zbirke
al.Remove('F');
al.Remove('A');

Console.WriteLine("Trenutna kapaciteta zbirke: " + al.Capacity);
Console.WriteLine("Trenutno število elementov zbirke: " + al.Count);

// Uporaba zanke foreach za izpis vsebine zbirke.
Console.Write("Vsebina zbirke (izpis s pomočjo zanke foreach): ");
foreach (char c in al)
   Console.Write(c + " ");
Console.WriteLine("\n");

Console.WriteLine("Dodajmo še 20 elementov");

for (int i = 0; i < 20; i++)
    al.Add((char)('a' + i));
Console.WriteLine("Trenutna kapaciteta zbirke: " + al.Capacity);
Console.WriteLine("Število elementov zbirke po dodajanju 20 elementov: " + al.Count);
Console.Write("Vsebina zbirke: ");
foreach (char c in al)
    Console.Write(c + " ");
Console.WriteLine("\n");

// Spremenimo vsebino zbirke s pomočjo tabelaričnega zapisa elementov zbirke.
Console.WriteLine("Spremenimo prve tri elemente zbirke");
al[0] = 'X';
al[1] = 'Y';
al[2] = 'Z';
Console.Write("Vsebina zbirke: ");
foreach (char c in al)
   Console.Write(c + " ");
Console.WriteLine();
```

Vaja:

```
/*Dan je poljuben stavek. Posamezne besede iz tega stavka prepišimo v zbirko z imenom besede,
nato pa te besede izpiši s pomočjo foreach zanke - vsaka beseda v svoji vrstici */

ArrayList besede = new ArrayList();
string stavek = "Danes je pa res en lep dan.";

//metoda split vse znake med dvema presledkoma shrani v ustrezen element objekta besede
besede.AddRange(stavek.Split(' '));

foreach (string beseda in besede) //izpis posameznih besed, vsaka beseda v novi vrstici
                Console.WriteLine(beseda);
```

Vaja:

```
/*Kreiraj zbirko 1000 naključnih realnih števil med 0 in 1000 z dvema decimalkama.
Napiši funkcijo, ki ugotovi in vrne največje število v zbirki
Napiši funkcijo, ki sešteje in vrne samo decimalke vseh števil v zbirki*/

//funkcija, ki ugotovi in izpiše največje število v zbirki
static void najvecje(ArrayList stevila)
{
    double najv=0;
    //Count je metoda zbirke, ki vrne trenutno število elementov zbirke
    for (int i = 0; i < stevila.Count;i++ )
        if ((double)stevila[i] > najv)
            najv = (double)stevila[i];  //vsak element zbirke moramo pretvoriti v double
    Console.WriteLine("Največje število v zbirki je "+najv);
}

//funkcija, ki sešteje in vrne vsoto vseh decimalk zbirke
static double vsotadec(ArrayList stevila)
{
    double vsota = 0;
    for (int i = 0; i < stevila.Count; i++)
    /*decimalke posameznega števila dobimo tako, da od števila odštejemo njegov celi del
    (funkcija truncate vrne celi del števila)*/
    //rezultat še zaokrožimo na dve decimalki
    vsota = vsota + Math.Round((double)stevila[i] - Math.Truncate((double)stevila[i]),2);
    return vsota;
}

static void Main(string[] args)
{
    //POZOR - imenski prostor ->>>>> using System.Collections;
     ArrayList stevila = new ArrayList();
     Random naklj = new Random();
     for (int i=0;i<1000;i++) //v zbirko dodamo 1000 naključnih števil
        stevila.Add(Math.Round(naklj.NextDouble()*1000,2));
     //klic funkcije, ki poišče največje število v zbirki
     najvecje(stevila);
     //klic funkcije, ki vrne vsoto decimalk zbirke
     Console.WriteLine("Vsota vseh decimalk zbirke je " + vsotadec(stevila));
}
```

Tipizirane zbirke

Pred deklaracijo **tipizirane** zbirke moramo poskrbeti za vključitev ustreznega imenskega prostora

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

Tipizirane zbirke imajo dve prednosti pred **netipiziranimi**. Prva je ta, da preverjajo tip vsakega elementa posebej že pri prevajanju, kar zagotavlja, da pri izvajanju programa ne pride do napak tipa **Run Time Error**. Druga prednost pa je seveda ta, da se na ta način zmanjša čas potreben za pretvarjanje podatkov iz zbirke (**casting**).

Tabela najpogosteje uporabljenih zbirk:

Zbirka	Razlaga
List<T>	Zbirka uporablja indeks za dostop do elementov, podobno kot tabela. Zelo je uporabna pri sekvenčnem dostopu do elementov zbirke. Lahko pa je neučinkovita pri vstavljanju elementov na sredino zbirke.
SortedList<K, V>	Uporablja ključ za pridobivanje vrednosti. Ključ je lahko poljuben objekt. Zbirka je lahko neučinkovita pri sekvenčnem dostopanju do posameznih elementov zbirke, je pa zelo učinkovita pri vstavljanju novih elementov na sredino zbirke.
Queue<T>	Uporablja metode za dodajanje in brisanje elementov.

Stack<T>	Uporablja metode za dodajanje in brisanje elementov.

Primer:

```
List<int> stevila = new List<int>();//deklaracija zbirke celih števil
//v zbirko lahko dodajamo le cela števila, njihovo število je poljubno!
stevila.Add(3);
stevila.Add(5);
//stevila.add("Test"); //prevajalnik bi tule javil napako!
int vsota = 0;
//Count je metoda zbirke, ki vrne trenutno število elementov zbirke
for (int i = 0; i < stevila.Count; i++)
{
   int stevilo = stevila[i];//eksplicitna konverzija NI potrebna
   vsota += stevilo;
}
```

Najpogosteje uporabljena zbirka je zbirka **List**. Nekaj primerov deklaracije takšnih zbirk:

```
//zbirka stringov
List <string> naslovi=new List<string>();

//zbirka decimalnih števil
List<decimal> cene = new List<decimal>();

//zbirka treh stringov. Seveda pa to ne pomeni, da stringov ne more biti več. A vsaka
//prekoračitev navedene dimenzije podvoji kapaciteto seznama!
List<string> kratice = new List<string>(3);
```

Najpomembnejše lastnosti in metode zbirke List

Indeks	Razlaga
[indeks]	Dostop do posameznega elementa zbirke. Indeks prvega elementa zbirje je tako kot pri tabelah enak 0.
Lastnost	**Razlaga**
Capacity	Nastavitev števila elementov ki jih lahko zbirka **List** hrani.
Count	Pridobivanje števila elementov v zbirki.
Lastnost	**Razlaga**
Add(objekt)	Dodajanje elementa na konec seznama in vračanje njegovega indeksa.
Clear()	Odstranitev vseh elementov iz seznama, lastnost **Count** postane enaka 0.
Contains(objekt)	Logična vrednost, ki ponazarja, ali seznam vsebuje navedeni objekt.
Insert(indeks,objekt)	Vrivanje elementa v seznam na mesto z navedenim indeksom.
Remove(objekt)	Odstranitev prve pojavitve navedenega objekta iz seznama.
RemoveAt(indeks)	Odstranitev elementa z navedenim indeksom iz seznama.
BinarySearch(objekt)	Iskanje navedenega objekta v seznamu in vračanje njegovega indeksa.
Sort()	Urejanje elementov seznama v naraščajočem zaporedju.

Primer:

```
List<string> priimki = new List<string>(3); //seznam treh stringov
priimki.Add("Ming");
priimki.Add("Lakovič");
priimki.Add("Taylor");
priimki.Add("House");//kapaciteta se podvoji na 6 elementov
priimki.Add("Menendez");
```

```
priimki.Add("Murach");
priimki.Add("Socrates");//kapaciteta se podvoji na 12 elementov
//urejanje seznama
priimki.Sort();

string vsiPriimki = "";
for (int i = 0; i < priimki.Count; i++)
{
   vsiPriimki += priimki[i] + "\n";
}
//urejen seznam vseh priimkov prikažemo v sporočilnem oknu
Console.WriteLine (vsiPriimki);
```

Še nekaj primerov:

```
//definicija tabele decimalnih števil
decimal[] novecene ={3275.68m, 4378.12m, 54123.34m, 100.00m };
//deklaracija seznama decimalnih števil
List<decimal> cene = new List<decimal>();
//v seznam dodamo vsa decimalna števila iz tabele novecene
foreach (decimal d in novecene)
      cene.Add(d);

decimal cena1 = cene[0];//dostop do prvega elementa zbirke
cene.Insert(0, 2345.11m);//vstavljanje nove cene na začetek seznama
cena1 = cene[0];  //cena1 dobi novo vrednost 2345.11

decimal cena2=cene[1];  //cena2 dobi vrednost 3275.68
cene.RemoveAt(1);    //odstranimo DRUGI elemenet (prvi elemeni ima indeks 0!) iz seznama
cena2 = cene[1];  ////cena2 dobi vrednost 4378.12

decimal x = 100.00m;
//če v zbirki najdemo znesek 100.00, da odstranimo iz zbirke
if (cene.Contains(x))
{
     cene.Remove(x);
     Console.WriteLine("Znesek odstranjen iz seznama!");
}
```

Uporaba zbirk tipa **vrsta** (**queue**) in **stack** je podobna, a imata obe vrsti zbirk svoje posebnosti in zaradi tega nista tako pogosto uporabljeni.

Naloge:

- Kreiraj zbirko n naključnih decimalnih števil (tudi n je podatek) večjih od 0 in manjših od 200. Ugotovi in izpiši koliko od teh števil je manjših od 10, med 10 (vključno) in 100 (vključno) in koliko večjih od 100.
- Preberi 10 besed in jih shrani v netipizirano zbirko. Ugotovi in izpiši najdaljšo besedo v tej zbirki.
- Napiši funkcijo, ki dobi za parameter poljuben stavek, vrne pa najdaljšo besedo v tem stavku. Navodilo: s pomočjo metod **AddRange** in **Split** besede iz tega stavka shrani v netipizirano zbirko, ki jo nato obdelaj!

Datoteke

Kaj je datoteka

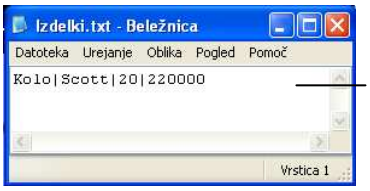
V Slovarju slovenskega knjižnega jezika (SSKJ) piše :
datoteka -e ž (e) elektr. urejena skupina podatkov pri (elektronskem) računalniku, podatkovna zbirka: vnesti nove podatke v datoteko / datoteka s programom.

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:

a) ***tekstovne*** 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).



b) ***binarne*** datoteke
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"). Binarne datoteke ne vsebujejo vrstic.

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. Izgled vsebine **binarne** datoteke, če jo odpremo z Beležnico

Ukvarjali se bomo več ali manj le s tekstovnimi datotekami.

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 \. Spomnimo se, da ima znak \, če ga v jeziku java in C# zapišemo kot sestavni del niza, poseben pomen. Šele v kombinaciji z naslednjim znakom predstavljajo nek znak (npr. \n - nova vrstica, \t - tabulator, \r - return). Zato, 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 @.

Knjižnica (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;
```

```
1 using System;
2 using System.IO;
```

Navedemo torej uporabo omenjenega imenskega prostora. S tem poenostavimo klice teh metod.

Podatkovni tokovi
Kratica `IO` v imenskem prostoru `System.IO` predstavlja vhodne (*input*) in izhodne (*output*) tokove[1] (*streams*). S pomočjo njih lahko opravljamo želene operacije nad datotekami (npr. branje, pisanje). V podrobnosti glede tokov se ne bomo spuščali in bomo predstavili poenostavljeno zgodbo. Pisalni ali izhodni tok v jeziku C# predstavimo s spremenljivko tipa `StreamWriter`. Bralni ali vhodni tok pa je tipa `StreamReader`.

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).

Branje in pisanje na tekstovne datoteke

Tekstovne datoteke
Tekstovne datoteke vsebujejo nize znakov oziroma zlogov, ki so ljudem berljivi. Katere znake vsebujejo, je odvisno od uporabljene kodne tabele.

Ustvarimo datoteko
Poglejmo si najenostavnejši način, kako ustvarimo tekstovno datoteko. "Čarobna" vrstica z ukazom je #8.

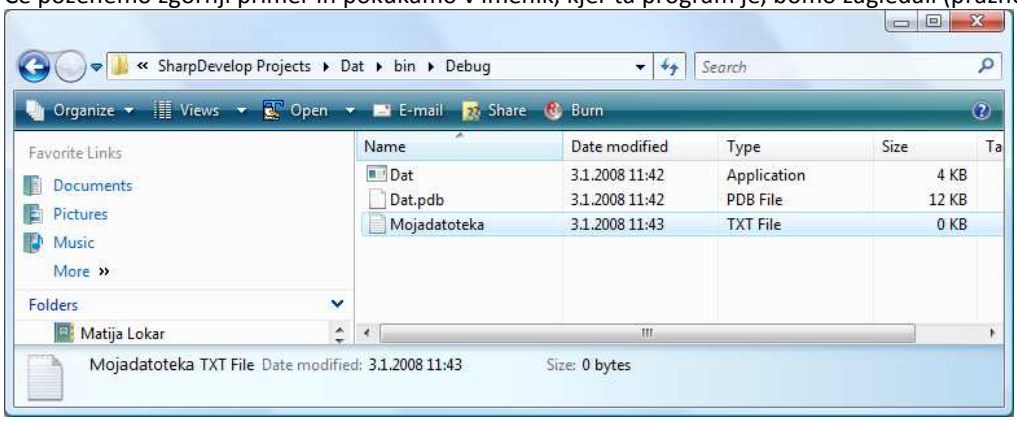
```
1:      using System;
2:      using System.IO;
```

[1] Tok ponazarja potek informacij od enega objekta k drugemu objektu.

```
3:      public class Program
4:      {
5:        public static void Main(string[] args)
6:        {
7:          File.CreateText("Mojadatoteka.txt");
8:        }
9:      }
```

"Zanimiva" je le vrstica 7. 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.

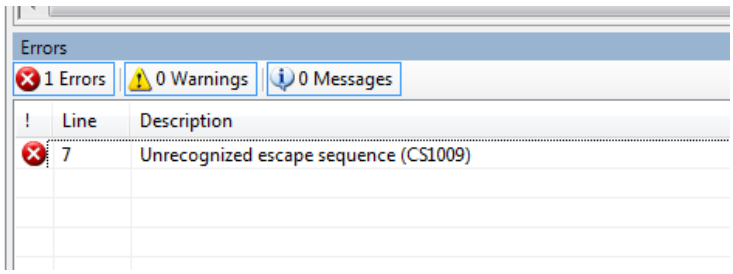
Če poženemo zgornji primer in pokukamo v imenik, kjer ta program je, bomo zagledali (prazno) datoteko.



Če bi želeli datoteko ustvariti v kakšnem drugem imeniku, moramo seveda napisati polno ime datoteke. Seveda takole

```
1:      using System;
2:      using System.IO;
3:      public class Program
4:      {
5:        public static void Main(string[] args)
6:        {
7:          File.CreateText("C:\temp\Mojadatoteka.txt");
8:        }
9:      }
```

ne gre, saj se prevajalnik hitro oglasi z



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 \ morda malo nepregleden, lahko pred nizem 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");
```

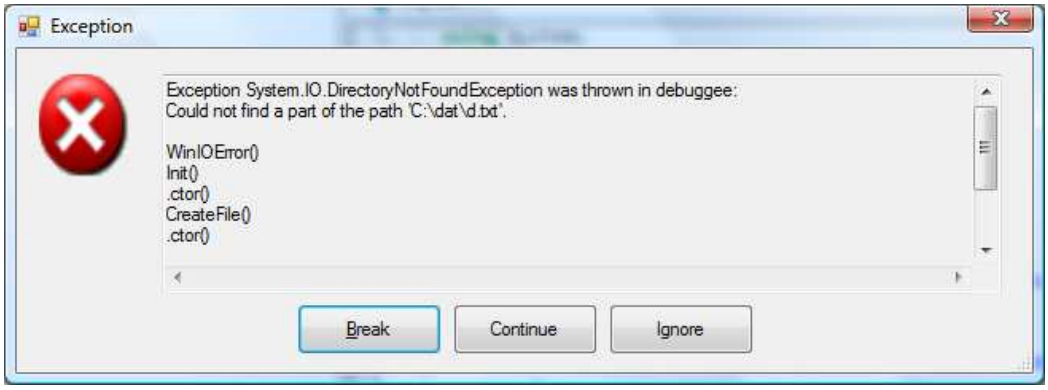
Vendar moramo biti pri uporabi tega ukaza previdni. Namreč, če datoteka že obstaja, jo s tem ukazom "povozimo". Izgubimo torej staro vsebino in ustvarimo novo, prazno datoteko.

Zato je 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"

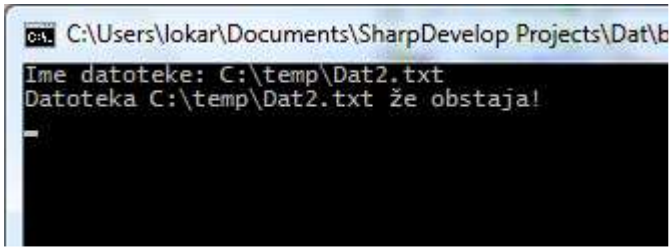


Zgled: Ustvari datoteko

Naredimo zgled, ki uporabnika vpraša po imenu datoteke. Če datoteke še ni, jo naredi, drugače pa izpiše ustrezno obvestilo.

```
10:   using System;
1:    using System.IO;
2:    public class Program
3:    {
4:      public static void Main(string[] args)
5:      {
6:        Console.Write("Ime datoteke: ");
7:        string ime = Console.ReadLine();
8:        if (File.Exists(ime)) {
9:          Console.WriteLine("Datoteka " + ime + " že obstaja!");
10:       } else {
11:         File.CreateText(ime);
12:         Console.WriteLine("Datoteko " + ime + " smo naredili!");
13:       }
14:       Console.ReadLine();
15:     }
```

```
16:  }
```



Iz slike vidimo še eno stvar. Ko tipkamo ime datoteke, \ navajamo običajno (enkratno), saj C# ve, da vnašamo "običajne" znake.

Razlaga programa: V 7. in 8. vrstici od uporabnika preberemo ime datoteke. V 9. vrstici preverimo, če datoteka obstaja. Če da, v 10. vrstici le izpišemo ustrezno obvestilo, drugače pa jo v 12. vrstici ustvarimo in uporabnika ustrezno obvestimo.

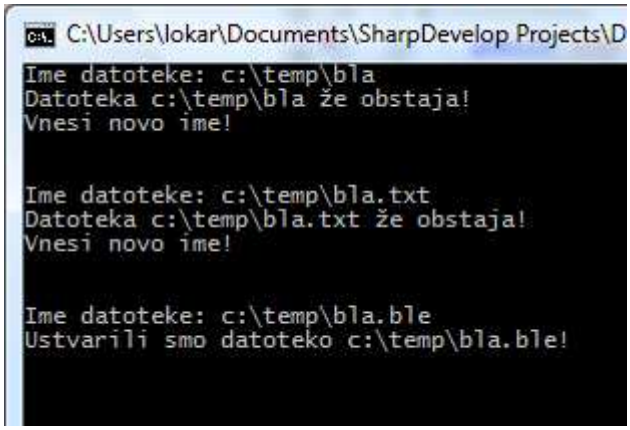
Zgled: Zagotovo ustvari datoteko

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.

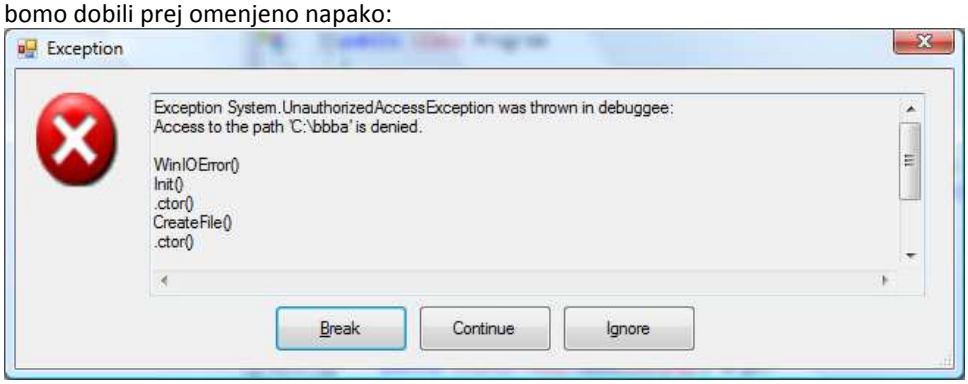
Ideja:

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.

```
1:   using System;
2:   using System.IO;
3:   public class Program
4:   {
5:     public static string NovoIme() {
6:       Console.Write("Ime datoteke: ");
7:       string ime = Console.ReadLine();
8:       while (File.Exists(ime)) { // èe datoteka že obstaja
9:         Console.WriteLine("Datoteka " + ime + " že obstaja!");
10:        Console.WriteLine("Vnesi novo ime!\n\n");
11:        Console.Write("Ime datoteke: ");
12:        ime = Console.ReadLine();
13:      }
14:      return ime;
15:    }
16:    public static void Main(string[] args)
17:    {
18:      string imeDatoteke = NovoIme();
19:      File.CreateText(imeDatoteke);
20:      Console.WriteLine("Ustvarili smo datoteko " + imeDatoteke + "!");
21:      Console.ReadLine();
22:    }
23:  }
```



Seveda pa rešitev še ni "idealna". Namreč, če nimamo pravice, da bi datoteko v določenem imeniku ustvarili, bomo dobili prej omenjeno napako:



Temu se lahko izognemo na ta način, da preverimo pravice, ki jih izvajalni program ima. A to že presega kontekst naše zgodbe.

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");
```

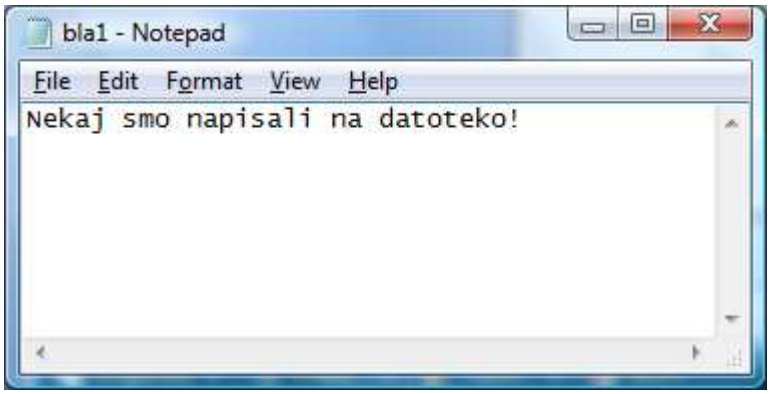
smo sedaj napisali omenjeno besedilo na datoteko, ki smo jo prej odprli in povezali z oznako.
Morebitne nejasnosti bo razjasnil zgled.

```
1:      public static void Main(string[] args)
2:      {
3:        string imeDatoteke = NovoIme();
4:        StreamWriter oznaka;
5:       oznaka = File.CreateText(imeDatoteke);
6:        oznaka.WriteLine("Nekaj smo napisali na datoteko!");
7:      }
```

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

```
1:      public static void Main(string[] args)
2:      {
3:        string imeDatoteke = NovoIme();
4:        StreamWriter oznaka;
5:        oznaka = File.CreateText(imeDatoteke);
6:        oznaka.WriteLine("Nekaj smo napisali na datoteko!");
7:        oznaka.Close();
8:      }
```

in si sedaj ogledamo datoteko, vidimo, da ni več dolga 0 zlogov. Če jo odpremo v najkoristnejšem programu Beležnici:



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 s 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# poskrbita, da je medpomnilnik izpraznjen tudi, če še ni povsem poln.

Seveda bi program lahko napisali tudi tako, da bi vrstici 4 in 5 združili

```
1:      public static void Main(string[] args)
2:      {
3:        string imeDatoteke = NovoIme();
4:        StreamWriter oznaka = File.CreateText(imeDatoteke);
5:        oznaka.WriteLine("Nekaj smo napisali na datoteko!");
6:        oznaka.Close();
7:      }
```

Oglejmo si sedaj nekaj zgledov programov, kjer pišemo na datoteke.

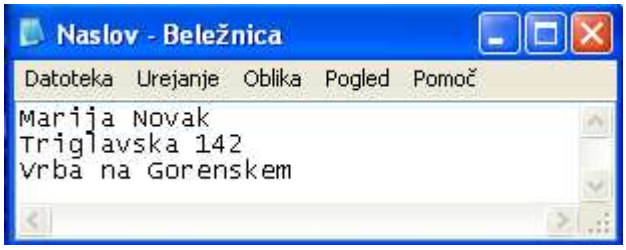
Osebni podatki

Na enoti `C` v korenskem imeniku ustvarimo mapo `Tekstovna`. V tej podmapi ustvarimo tekstovno datoteko `Naslov.txt` in vanjo zapišimo svoj naslov. To naredimo le, če datoteke še ni.

```
C1:   using System;
C2:   using System.IO;
C3:   public class OsebniPodatki{
```

```
C4:      public static void Main(string[] args){
C5:         // Pot in ime
C6:         string ime = @"C:\Tekstovna\Naslov.txt";
C7:
C8:         // Preverimo obstoj datoteke
C9:         if(File.Exists(ime)){
C10:           Console.WriteLine("Datoteka ze obstaja.");
C11:           return;
C12:        }
C13:
C14:        // Ustvarimo novo datoteko
C15:        StreamWriter dat = File.CreateText(ime);
C16:
C17:        // Zapišemo osebne podatke
C18:        dat.WriteLine("Marija Novak");
C19:        dat.WriteLine("Triglavska 142");
C20:        dat.WriteLine("Vrba na Gorenskem");
C21:
C22:        // Zapremo datoteko za pisanje
C23:        dat.Close();
C24:     }
C25:  }
```

Zapis na datoteki Naslov.txt:



Razlaga. Najprej določimo niz, ki predstavlja polno ime datoteke (vrstica C6). Če bi za niz določili le kratko ime, bi se datoteka ustvarila tam, kjer bi izvedli program `OsebniPodatki.cs`. V vrstici C9 preverimo obstoj datoteke. Če datoteka obstaja, izpišemo obvestilo (vrstica C10) in končamo izvajanje programa (vrstica C11). Nato v vrstici C15 odprem datoteko za pisanje. S klicem metode `dat.WriteLine()` podatke zapišemo na datoteko (vrstice C18 - C20). Po končanem zapisu datoteko zapremo (vrstica C23). Če tega ne bi storili, datoteka ne bi imela vsebine.

Števila

V tekstovno datoteko `Stevila.txt` zapišimo prvih `n` sodih števil, ki niso deljiva s šest. Števila izpišimo ločena z znakom '`/`', torej na primer kot:

```
1/2/3/8/14/16/20/22/
```

Ideja.
Napišemo metodo, ki ima dva parametra:

- *ime* - ime datoteke
- *n* - koliko števil bo v datoteki

Rezultat metode bo *void*, saj ima metoda le učinek ustvarila bo datoteko.

```
private static void Stevila(string ime, int n)
```

Najprej preverimo, če datoteka *ime* obstaja. V primeru obstoja izpišimo opozorilo in prekinimo izvajanje metode. Lepše bi bilo, če bi namesto izpisa obvestila metoda vrgla izjemo. A ker se z izjemami v jeziku C# ne bomo ukvarjali, bomo napako javili z izpisom obvestila.

```
if (File.Exists(ime)){
   Console.WriteLine("Napaka.");
   return;
```

```
}
```

Nato ustvarimo datoteko in jo povežemo s tokom za pisanje.

```
StreamWriter dat;
dat = File.CreateText(ime);
```

Ustvarimo še števec za štetje v datoteko zapisanih števil in spremenljivko, ki bo hranila tekoče sodo število.

```
int stevec = 1;
int soda = 0;
```

Nato naredimo zanko, ki se bo izvajala toliko časa, dokler ne bo *stevec* postal večji kot *n*. V zanki:

- Preverimo, če vrednost spremenljivke *soda* ni deljiva s *6*.
  - Če je pogoj izpolnjen, vrednost spremenljivke *soda* zapišemo na datoteko.
  - Povečamo spremenljivko *stevec* za 1.
- Spremenljivko *soda* nastavimo na naslednje sodo število..

```
while (stevec <= n){
   if (soda % 6 != 0){
      dat.Write(soda + "/");
      stevec++;
   }
   soda = soda + 2;
}
```

Zatem zapremo podatkovni tok.

```
dat.Close();
```

Na koncu še napišemo metodo *Main*. V metodi:

- napišemo ime datoteke,
- preberemo število *n* in
- pokličemo metodo *Stevila*.

```
public static void Main(string[] args){
   // Ime datoteke
   string datoteka = "Stevilo.txt";
   // Število števil v datoteki
   Console.Write("Vnesi n: ");
   int n = int.Parse(Console.ReadLine());
   Stevila(datoteka,n); // Klic metode
}
```

Sestavimo zapisane dele programa v celoto.

```
C1:   using System;
C2:   using System.IO;
C3:   public class Program{
C4:      private static void Stevila(string ime, int n){
C5:         // Preverjanje obstoja datoteke
C6:         if (File.Exists(ime)){
C7:            Console.WriteLine("Napaka.");
C8:            return;
C9:         }
C10:
C11:        StreamWriter dat; // Ustvarimo tok za pisanje na datoteko
C12:        dat = File.CreateText(ime);
C13:
C14:        // Pomožni spremenljivki
C15:        int stevec = 1;
C16:        int soda = 0;
C17:
C18:        // Zapisujemo števila na datoteko
C19:        while (stevec <= n){
C20:           if (soda % 6 != 0){
C21:              dat.Write(soda + "/");
```

```
C22:                stevec++;
C23:             }
C24:             soda = soda + 2;
C25:          }
C26:
C27:          // Zapremo datoteko za pisanje
C28:          dat.Close();
C29:       }
C30:
C31:       public static void Main(string[] args){
C32:          // Ime datoteke
C33:          string datoteka = "Stevilo.txt";
C34:          // Število števil v datoteki
C35:          Console.Write("Vnesi n: ");
C36:          int n = int.Parse(Console.ReadLine());
C37:          Stevila(datoteka,n); // Klic metode
C38:       }
C39:    }
```

Zapis 100 vrstic

Napišimo sedaj program, ki bo v datoteko StoVrstic.txt zapisal 100 vrstic, denimo takih: 1. vrstica, 2. vrstica, ..., 100. vrstica

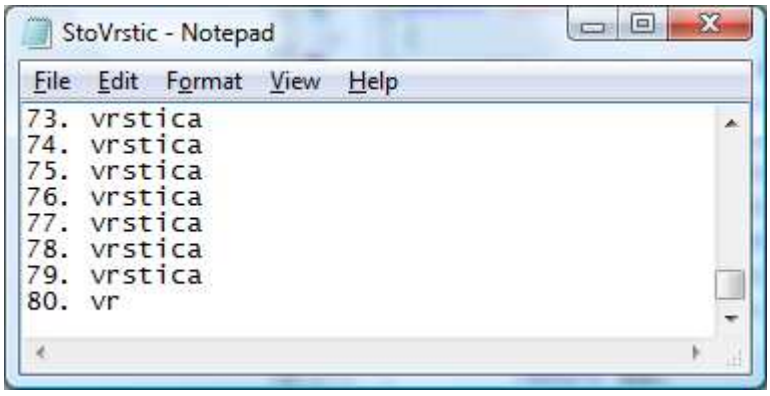
Naloga je enostavna:

- Ustvarimo datoteko StoVrstic.txt in jo povezeno z pisalnim podatkovnim tokom
- V zanki izpišemo števec in besedilo ". vrstica"

Nato ... Nič! To je vse.

```
1:      public static void Main(string[] args)
2:      {
3:        StreamWriter oznaka;
4:        oznaka = File.CreateText(@"c:\temp\StoVrstic.txt");
5:        for (int i = 1; i <= 100; i++) {
6:          oznaka.WriteLine(i + ". vrstica");
7:        }
8:      }
```

Poglejmo datoteko. Na začetku je vse lepo in prav, a kaj pa je s koncem:



20 vrstic manjka! Razlog je v tem, da smo pozabili na Close. Če torej dodamo

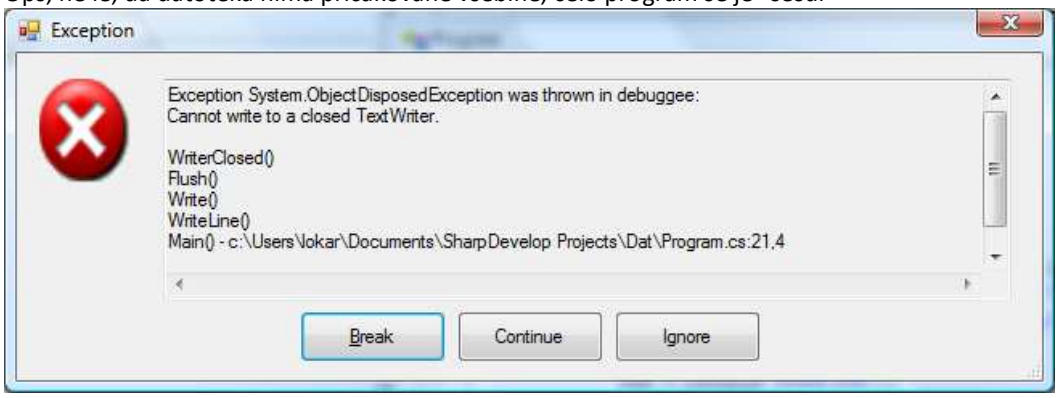
```
oznaka.Close()
```

bo datoteka StoVrstic.txt taka, kot pričakujemo.

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

```
2:      {
3:        StreamWriter oznaka;
4:        oznaka = File.CreateText(@"c:\temp\StoVrstic.txt");
5:        for (int i = 1; i <= 100; i++) {
6:          oznaka.WriteLine(i + ". vrstica");
7:          oznaka.Close();
8:        }
9:      }
```

Ops, ne le, da datoteka nima pričakovane vsebine, celo program se je "sesul"



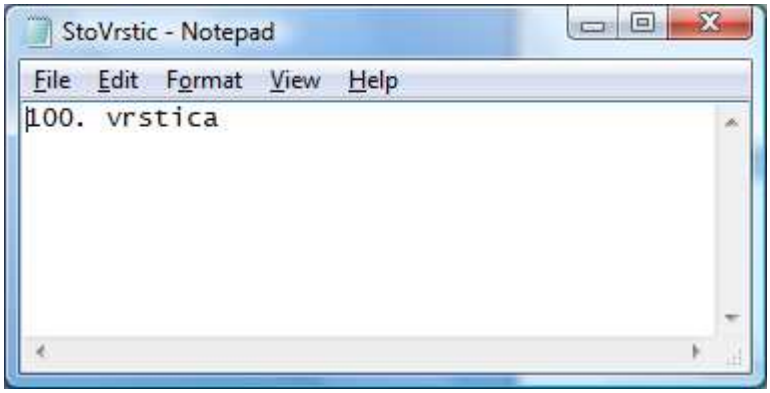
Razlog nam lepo pojasni obvestilno okno. Piše namreč:

Cannot write to a closed TextWriter.

Datoteko smo torej zaprli na napačnem mestu, v zanki, ko bi morala biti še odprta. Popravimo program

```
1:    public static void Main(string[] args)
2:      {
3:        StreamWriter oznaka;
4:        for (int i = 1; i <= 100; i++) {
5:          oznaka = File.CreateText(@"c:\temp\StoVrstic.txt");
6:          oznaka.WriteLine(i + ". vrstica");
7:          oznaka.Close();
8:        }
9:      }
```

Tako. Program sedaj deluje lepo in prav. A ko odpremo datoteko SttoVrstic.txt nas čaka presenečenje:



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

```
1:    public static void Main(string[] args)
2:      {
3:        StreamWriter oznaka;
4:        oznaka = File.CreateText(@"c:\temp\StoVrstic.txt");
5:        for (int i = 1; i <= 100; i++) {
6:          oznaka.WriteLine(i + ". vrstica");
7:        }
8:        oznaka.Close();
9:      }
```

je datoteka končno taka, kot smo pričakovali.
Bralcem se za "neumne" napake seveda opravičujemo. 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.

Datoteka naključnih števil

Sestavi metodo, ki ustvari datoteko NakStev**X**.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 izpiši levo poravnana po k 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 NakStev**X**.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)) { // èe 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 naredimo zanko, ki se bo izvedla n-krat. V njej

- Ustvarimo naključno število
- Ga zapišemo na datoteko. S tabulatorjem "\t" poskrbuimo 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
- Zapremo podatkovni tok

```
if (stevec % kolikoVVrsti != 0) {
          // če prej nismo ravno končali z vrstico
       pisiDat.WriteLine(); // zaključèimo vrstico
}
pisiDat.Close();
```

Poglejmo sedaj še celotni program

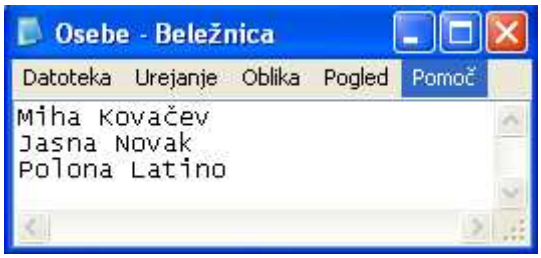
```
1:    using System;
2:    using System.IO;
3:    public class Program
4:    {
5:      public static void UstvariDat(int n, int spMeja, int zgMeja,
6:                                    int kolikoVVrsti) {
7:        string osnovaImena = @"C:\temp\NakStev";
8:        int katera = 1;
9:        string ime = osnovaImena + katera + ".dat";
10:       while (File.Exists(ime)) { // èe datoteka že obstaja
11:         katera++;
12:         ime = osnovaImena + katera + ".dat";
13:       }
14:       // našli smo "prosto" ime
15:       StreamWriter pisiDat;
16:       pisiDat = File.CreateText(ime);
17:       Random genNak = new Random();
18:       int stevec = 0;
19:       while (stevec < n) {
20:         int nakStev = genNak.Next(spMeja, zgMeja);
21:         pisiDat.Write("\t " + nakStev);
22:               // s tabulatorji poskrbimo za poravnavo
23:         stevec++; // zapisali smo še eno število
24:         if (stevec % kolikoVVrsti == 0) {
25:           pisiDat.WriteLine(); // zaključèimo vrstico
26:         }
27:       }
```

```
28:       if (stevec % kolikoVVrsti != 0) {
29:           // èe prej nismo ravno konèali z vrstico
30:         pisiDat.WriteLine(); // zaključèimo vrstico
31:       }
32:       pisiDat.Close();
33:     }
34:
35:     public static void Main(string[] args)
36:     {
37:       UstvariDat(32, 1, 1000, 5);
38:     }
39:   }
```

Kako v datotekah hranimo podatke

V datoteki hranimo podatke po nekem pravilu, imenovanem format zapisa, ki nam pove, kako so podatki zapisani. Pri formatu zapisa moramo biti precej pozorni, sicer se preproste zadeve hitro zapletejo. Oglejmo si primer.

Imamo datoteko v katero zapisujemo ime in priimek poljubnih oseb.



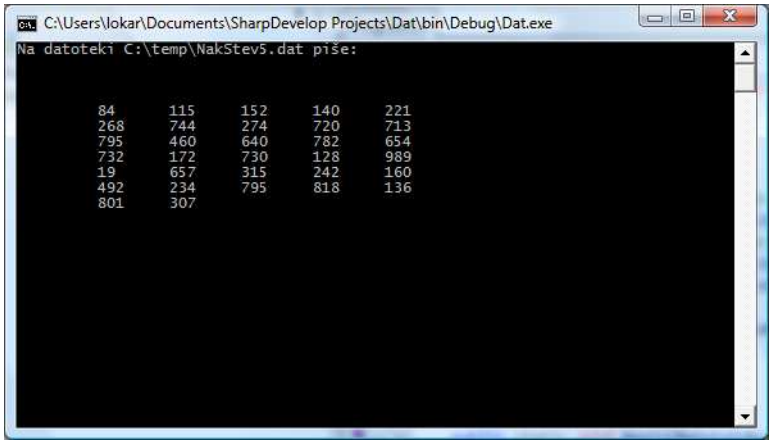
Problem nastopi, ko vpisujemo osebo, ki ima dve ali več imen. Na primer Ana Marija Težava. Kako ravnati v primeru, ko imamo med besedama Ana in Marija presledek? Sicer s samim zapison ne bo probelma. Ta pa se bo pojavil, ko bomo tak podatek brali. Ali sedaj Marija spada pod priimek in kam potem spada njen priimek Težava?

Da se izognemo takšnim težavam, je priporočljivo, da naredimo pred zapisovanjem načrt formata za obstoječe podatke. Ko delamo načrt, se držimo nekaterih pomembnih nasvetov:

- Format zapisa mora biti načrtovan tako, da ne more priti do zmede. Na primer, če so podatki med seboj ločeni z presledki, potem polj v zapisu ne smemo ločiti s presledki.
- Format zapisa mora biti tak, da programerju olajša pisanje podatkov iz programa na datoteko ter branje iz datoteke v program.
- Dobro premislimo, v kakšnem vrstnem redu naj bodo podatki zapisani.
- Pomembno je, da so zapisani podatki pregledni, da jih ljudje razumejo. Še pomembneje pa je, da so podatki shranjeni tako, da jih je lahko brati in pisati s programi.

Branje tekstovnih datotek

Poglejmo si sedaj, kako bi prebrali tisto, kar smo v prejšnjem razdelku napisali na datoteko.



Branje po vrsticah

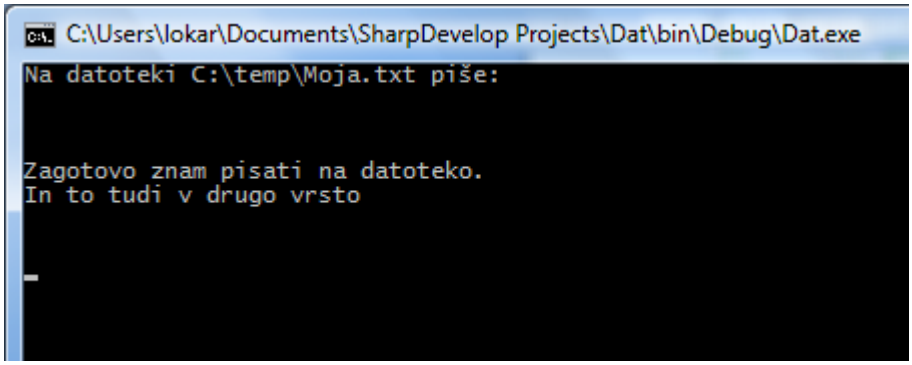
S tekstovnih datotek najlažje beremo tako, da preberemo kar celo vrstico hkrati. Poglejmo si naslednjo metodo

```
1:      public static void IzpisDatoteke(string ime) {
2:        Console.WriteLine("Na datoteki " + ime + " piše: \n\n\n");
3:        // odpremo datoteko za
4:        StreamReader izhTok;
5:        izhTok = File.OpenText(ime);
6:        // preberemo prvo vrstico in jo izpišemo na zaslon
7:        Console.WriteLine(izhTok.ReadLine());
8:        // preberi z datoteke naslednji dve vrstici in ju izpiši na zaslon
9:        Console.WriteLine(izhTok.ReadLine());
10:       Console.WriteLine(izhTok.ReadLine());
11:       // preberi preostale vrstice
12:       Console.WriteLine(izhTok.ReadToEnd());
13:       // zaprimo tok
14:       izhTok.Close();
15:     }
```

Razlaga. V 2. vrstici najprej na zaslon izpišemo obvestilo in spustimo tri prazne vrstice. Nato v vrstici 4 določimo spremenljivko, ki bo označevala izhodni tok. V v. 5 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.

Branje datoteke z dvema vrsticama

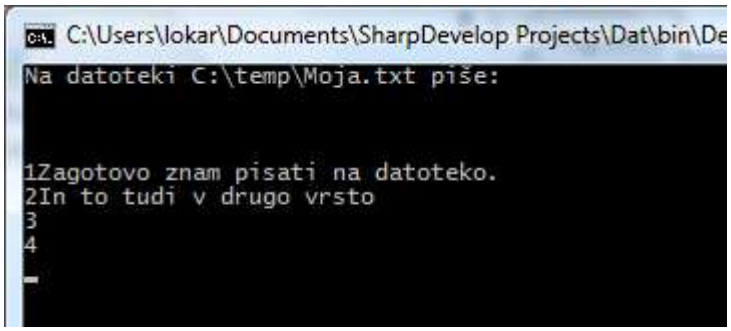
Z zgornjo metodo preberimo datoteko, ki ima le dve vrstici.



Kot vidimo, smo izpisali štiri vrstice! Kako to? Če si z Beležnico ogledamo datoteko Moja.txtx 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());
```

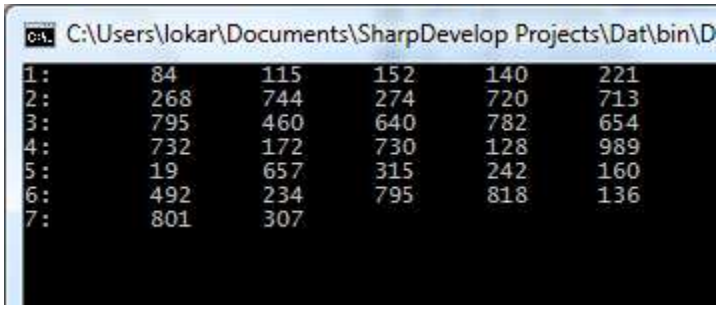
na zaslon dobimo



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.
Dejstvo, da metoda ReadLine() vrne neobstoječ niz, bomo uporabili v naslednjem zgledu.

Branje in številčenje vrstic

Denimo, da želimo prebrati neko tekstovno datoteko in jo izpisati z oštevilčenimi vrsticami. Npr. takole:



Ali pa, če je vsebina datoteke `Vrba.txt`

```
O Vrba! srečna, draga vas domača,
kjer hiša mojega stoji očeta;
de b' uka žeja me iz tvojga sve'ta
speljala ne bila, goljfiva kača!

Ne vedel bi, kako se v strup prebrača
vse, kar srce si sladkega obeta;
mi ne bila bi vera v sebe vzeta,
ne bil viharjov no'tranjih b' igrača!

Zvesto' srce in delavno ročico
za doto, ki je nima miljonarka,
bi bil dobil z izvoljeno devico;

mi mirno plavala bi moja barka,
pred ognjam dom, pred točo mi pšenico
bi bližnji sosed va'roval - svet' Marka.
```

potem naj program izpiše

```
1       O Vrba! srečna, draga vas domača,
2       kjer hiša mojega stoji očeta;
3       de b' uka žeja me iz tvojga sve'ta
4       speljala ne bila, goljfiva kača!
5
6       Ne vedel bi, kako se v strup prebrača
7       vse, kar srce si sladkega obeta;
8       mi ne bila bi vera v sebe vzeta,
9       ne bil viharjov no'tranjih b' igrača!
10
11      Zvesto' srce in delavno ročico
12      za doto, ki je nima miljonarka,
13      bi bil dobil z izvoljeno devico;
14
15      mi mirno plavala bi moja barka,
16      pred ognjam dom, pred točo mi pšenico
17      bi bližnji sosed va'roval - svet' Marka.
```

Glavni "igralec" tukaj bo zanka, ki bo izpisala posamezno vrstico in prebrala naslednjo vrstico. To bomo počeli toliko časa, dokler prebrana vrstica ne bo neobstoječ niz.

```
while (vrstica != null) {
  Console.WriteLine(stevec + ": " + vrstica);
  vrstica = izhTok.ReadLine();
```

```
    stevec++;
  }
```

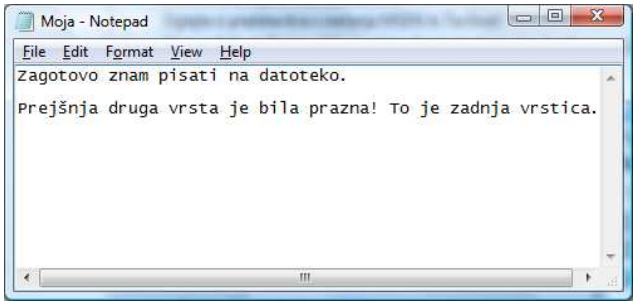
In še celotna metoda

```
public static void IzpisDatoteke2(string ime) {
  StreamReader izhTok;
  izhTok = File.OpenText(ime);
  // preberemo prvo vrstico in jo izpišemo na zaslon
  string vrstica = izhTok.ReadLine();
  int stevec = 1;
  while (vrstica != null) {
    Console.WriteLine(stevec + ": " + vrstica);
    vrstica = izhTok.ReadLine();
    stevec++;
  }
  // zaprimo tok
  izhTok.Close();
}
```

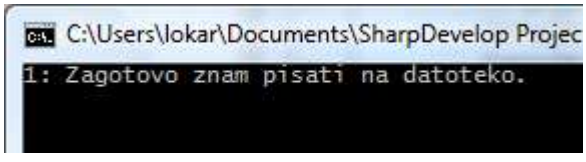
Moramo pa paziti. Če bi namreč zanko napisali kot

```
  while (vrstica != "") {
    Console.WriteLine(stevec + ": " + vrstica);
    vrstica = izhTok.ReadLine();
    stevec++;
  }
```

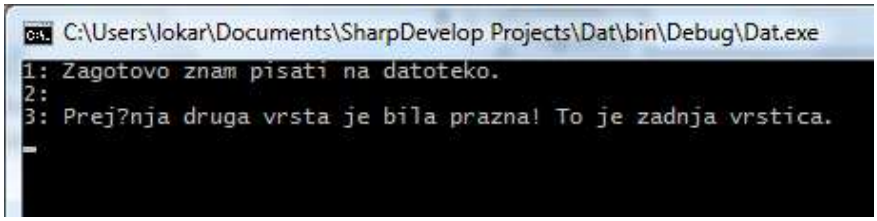
bi npr. pri datotekah, ki vsebujejo kakšno prazno vrstico, dobili izpisane vrstice le do te prazne vrstice. Npr. če metodo izvedemo nad



dobimo pri spremenjeni kodi



namesto



Pogosto bomo metodo za branje datoteke po vrsticah videli zapisano v obliki

```
public static void IzpisDatoteke1(string ime) {
  StreamReader s = File.OpenText(ime);
  string beri = null;
  while ((beri = s.ReadLine()) != null)
  {
    Console.WriteLine(beri);
  }
  s.Close();
}
```

"Čudna" je le oblika `while ((beri = s.ReadLine()) != null)`. Gre za uporabo dejstva, da C# pozna tudi tako imenovan **prireditveni izraz**. To je "izraz z učinkom".
Če napišemo

```
x = 3;
```

je to prireditveni stavek, ki v spremenljivko x shrani 3. Če pa napišemo

```
x = 3;
```

(torej brez podpičja), gre za izraz. Ta izraz ima enak učinek, kot prireditveni stavek (torej v x shrani 3), poleg tega pa ima tako kot vsi izrazi tudi vrednost. Vrednost prireditvenega izraza je tista vrednost, ki se je priredila. V našem primeru torej 3. Z uporabo prireditvenih stavkov lahko napišemo npr.

```
a = b = 0;
```

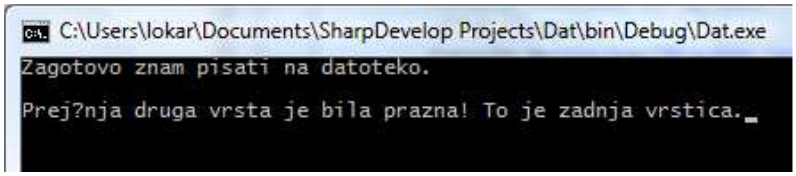
Zakaj gre? Napisali smo prireditveni stavek, ki v spremenljivko a shrani ... kaj? Vrednost izraza, kaj pa drugega. In kaj je izraz – gre za prireditveni izraz (b = 0). Ta v b shrani 0 (ima učinek) in kot svoj rezultat vrne vrednost, ki smo jo shranili v b, torej 0. V a se bo torej tudi shranila 0.
Sedaj lahko razumemo, zakaj gre pri `beri = s.ReadLine()) != null` To je pogojni izraz (torej nekaj, kar ima vrednost true ali false), ki sprašuje, ali je nekaj različno (!=) od null. Tisto nekaj pa je vrednost prireditvenega izraza `beri = s.ReadLine()`. Vrednost tega izraza je tisto, kar se shrani v spremenljivko `beri`, torej to, kar prebere metoda `ReadLine`. Z (beri = s.**ReadLine**()) torej preberemo vrstico iz vhodnega toka s. Prebrano shranimo v spremeljivko beri in zraven še primerjamo, če je ta shranjena vrednost različna od null.
Ko se torej preverja pogoj `((beri = s.ReadLine()) != null)` sta dve možnosti. Če v vhodnem toku ni več vrstic, bomo v `beri` shranili vrednost null in pogoj bo imel vrednost `false`. Če pa vrstice še obstajajo, bomo tekočo vrstico prebrali in shranili v `beri`. Pogoj pa se bo torej izračunal v `true` (in se bo zanka nadaljevala). Ko smo torej v telesu zanke, smo ravno prebrali tekočo vrstico, ki obstaja (ni `null`). Zanka pa se preneha izvajati, ko ravno "preberemo" neobstoječo vrstico (`ReadLine` vrne `null`, kar shranimo v `beri`).

Branje po znakih

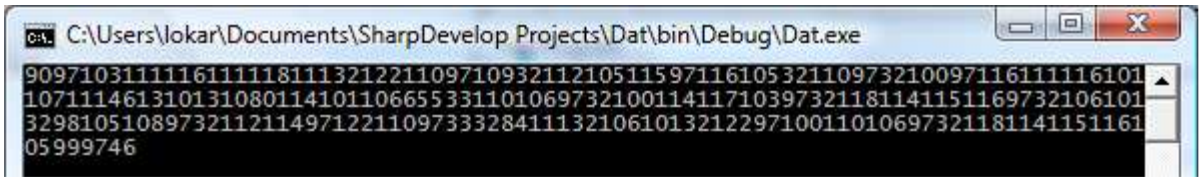
Tekstovne datoteke lahko beremo tudi znak po znak. Poglejmo si kar metodo, ki datoteko znak po znak prepiše na zaslon.

```
public static void IzpisDatotekePoZnakih(string ime) {
  StreamReader s = File.OpenText(ime);
  int beri;
  beri = s.Read();
  while (beri != -1) // konec datoteke
  {
    Console.Write((char)beri); // izpisujemo ustrezne znake
    beri = s.Read();
  }
  s.Close();
}
```

Ko beremo datoteko po znakih uporabljamo metodo Read. Ta nam vrne kodo znaka, ki ga preberemo. Ko bomo naleteli na konec datoteko, 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



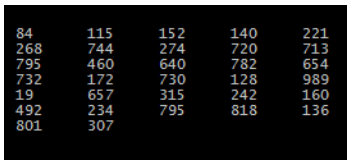
dobili (če bi bila 7. vrstica torej `Console.Write(beri);`)



Če verjamete ali ne, gre za isto datoteko, le da so druga poleg druge izpisane kode znakov namesto njihovih grafičnih podob.

Iz vrstic izlušči posamezne znake

Kako iz datoteke, kjer je v vrstici več števil,

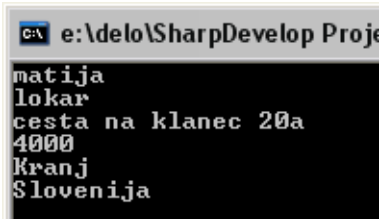


dobiti posamezna števila.
V C# imamo na voljo metodo `split`, ki zna niz razdeliti na posamezne dele. Ta del kode bo povedal vse:

```
string info = "matija;lokar;cesta na klanec 20a;4000;Kranj;Slovenija";
string[] tabInfo;
//določimo znake, ki predstavljajo ločila med podatki
char[] locila  = {';'}; // mi smo za ločilo 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 ;.



Če pa bi med ločila dodali še npr. presledek

```
char[] locila  = {';', ' '}; // locili sta presledek in ;
```

bi dobili dodatne tri vrstice

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

Poznati je potrebno strukturo datoteke, da znamo določiti separatorski 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!

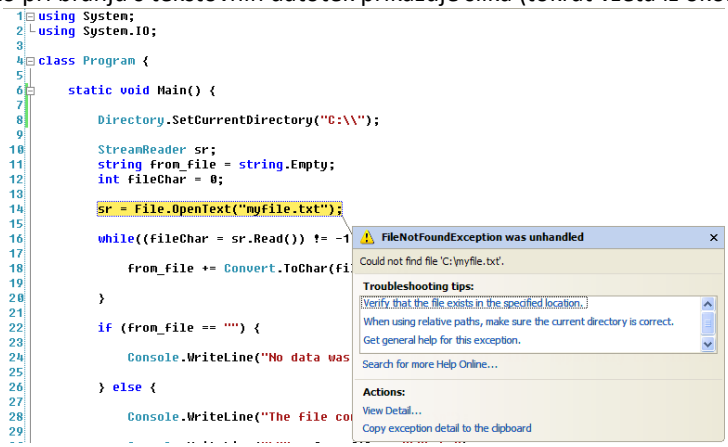
Predpostavimo spet, da v datoteki ni več kot 100 števil.

```
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] != "") { // èe nismo dobili le prazni niz
          tabela[koliko] = int.Parse(tabInfo[x]);
          koliko++;
        }
      }
      vrst = beri.ReadLine();
    }
    beri.Close();
    return tabela;
  }
```

Datoteke ni

Najbolj pogosto napako pri branju s tekstovnih datotek prikazuje slika (tokrat vzeta iz okolja Visual Studio)



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 enekim 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

```
1:    public static void IzpisDatoteke4(string ime) {
2:       if (!File.Exists(ime)) {
3:         Console.WriteLine("Datoteka " + ime + " ne obstaja!");
4:       } else {
5:         StreamReader izhTok;
6:         izhTok = File.OpenText(ime);
7:         // preberemo prvo vrstico in jo izpišemo na zaslon
```

```
8:          string vrstica = izhTok.ReadLine();
9:          int stevec = 1;
10:         while (vrstica != "") {
11:           Console.WriteLine(stevec + ": " + vrstica);
12:           vrstica = izhTok.ReadLine();
13:           stevec++;
14:         }
15:         // zaprimo tok
16:         izhTok.Close();
17:       }
18:     }
```

Ponovitev

Ponovimo pomembne metode, ki jih srečamo pri delu s tekstovnimi datotekami:

- `StreamReader`: Tip spremenljivke za branje toka.
- `StreamWriter`: Tip spremenljivke, ki ga uporabimo pri spremenljivkah, v katerih je shranjena oznaka podatkovnega toka.
- `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() in Write()`: Metodi, ki zapišeta dani niz na datoteko
- `Close()`: Metoda, s katero zapremo datoteko.

Zgledi

Prepiši datoteko

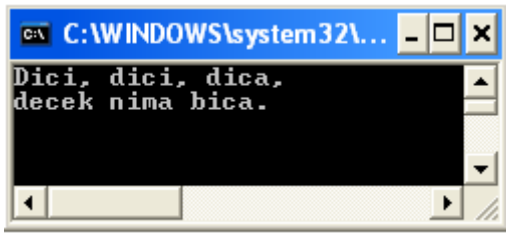
Podano imaš datoteko Decek.txt, ki vsebuje le dve vrstici. Napiši metodo, ki za vhodni podatek sprejme ime datoteke in na zaslon izpiše dani vrstici.

```
1:    using System;
2:    using System.IO;
3:    namespace Branje
4:    {
5:      class Branje
6:      {
7:        public static void IzpisNaZaslon1(string ime){
8:           if(!File.Exists(ime)){
9:                Console.WriteLine("Datoteka ne obstaja");
10:          } else{
11:                 StreamReader dat;
12:                 dat = File.OpenText(ime);
```

```
13:                 Console.WriteLine(dat.ReadLine());
14:                 Console.WriteLine(dat.ReadToEnd());
15:                 dat.Close();
16:             }
17:         }
18:     }
19: }
```

- V 1. vrstici napovemo uporabo imenskega prostora, ki vsebuje osnovne tipe (int, double, …)
- V 2. vrstici vidimo ukaz using System.IO, s katerim C# povemo, da bomo uporabili razrede in metode iz imenskega prostora System.IO. Ta imenski prostor vsebuje vse potrebno za delo z datotekami.
- V 8. vrstici vidimo, da z metodo File.Exists() preverimo, ali dana datoteka sploh obstaja. Zelo smiselno je, da pred odpiranjem datotek preverimo, če datoteka obstaja. Če želene datoteke ni, se program sicer sesuje. V primeru, da datoteka obstaja, ukaz File.Exists() vrne true, sicer false.
- V 9. vrstici izpišemo opozorilo, ki se izpiše kadar iskana datoteka ne obstaja.
- V 11. vrstici vidimo, kako določimo spremenljivko, ki bo označevala izhodni tok.
- V 12. vrstici z metodo File.OpenText() povežemo izhodni tok z datoteko z imenom ime.
- V 13. vrstici z metodo ReadLine() preberemo celotno vrstico in jo z ukazom Console.WriteLine() izpišemo na zaslon.
- V 14. vrstici smo za branje uporabili tudi metodo ReadToEnd(). Ta prebere vse vrstice v datoteki od trenutne vrstice dalje (torej od druge) pa do konca. V našem primeru bi bilo povsem vseeno, če bi povnovno uporabili kar metodo ReadLine(), saj je v datoteki le še ta vrstica.
- V 15. vrstici vidimo, da na koncu datoteko zapremo z metodo Close(). To naredimo vedno, ko končamo z uporabo datoteke.

Rezultat izpisa:



Opozorili:

- V splošnem ne vemo vnaprej, koliko vrstic ima podana datoteka.
- Če z metodo ReadLine() ali ReadToEnd() beremo potem, ko datoteka nima več obstoječih vrstic (trenutne vrstice ni več, oziroma smo na koncu datoteke), metoda kot rezultat vrne null (neobstoječ niz). Neobstoječi niz je ob izpisu na zaslon viden kot prazen niz (prazna vrstica).

Odstrani prazne vrstice

Na datoteki Dekle.txt je pesem zapisana po kiticah. To pomeni, da so med kiticami prazne vrstice. Napiši metodo, ki bo za dano ime datoteke vse kitice združila in jih izpisala na zaslon.

Vsebina datoteke Dekle.txt:	Izpis na zaslon:
Dekle je po vodo šlo *na visoke planine.* *Vodo je zajemala,* *je ribico zajela.* *Ribica jo je prosila:* *oj, pusti me živeti.* *Dekle b'la je usmiljena,* *je ribico spustila.* *Ribica je zaplavala,* *je dekle poškropila.*	*Dekle je po vodo šlo* *na visoke planine.* *Vodo je zajemala,* *je ribico zajela.* *Ribica jo je prosila:* *oj, pusti me živeti.* *Dekle b'la je usmiljena,* *je ribico spustila.* *Ribica je zaplavala,* *je dekle poškropila.*

Koda metode:

```
public static void IzpisNaZaslon(string ime)
      // preverimo, če datoteka obstaja
      if (!File.Exists(ime)) {
           Console.WriteLine("Datoteka ne obstaja");
      } else {
         // odpremo datoteko za branje
         StreamReader dat = File.OpenText(ime);
         //pomožna spremenljivka,ki nam bo predstavljala eno vrstico
         string vrsta = dat.ReadLine();
             // zanka, ki teče dokler je še kakšen obstoječ niz
             while(vrsta != null){
               // če niz ni prazen (ni šlo za prazno vrstico) ga izpišemo
               if(vrsta.Length != 0){
                  Console.WriteLine(vrsta);
               }
               //preberemo novo vrstico
               vrsta = dat.ReadLine();
         }
             //zapremo datoteko
             dat.Close();
          }
      }
```

Glavna "trika" v tej metodi sta:

- zanka, ki se izvaja toliko časa, dokler je v datoteki še kakšna neprebrana vrstica oziroma obstoječ niz.

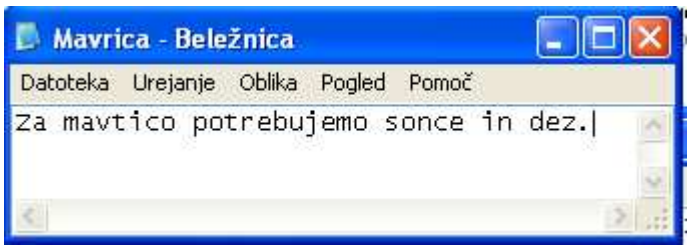
```
while(vrsta != null){…}
```

- pogojni stavek, ki pregleduje, ali je prebrana vrstica prazna (brez znakov). V primeru, da je vrstica polna, jo izpišemo. Pri preverjanju polnosti vrstice se spomnimo, da so vrstice v resnici nizi. Za nize pa vemo, da je niz prazen, če je dolžina niza enako 0.

```
if(vrsta.Length != 0) {…}
```

Izpiši datoteko na zaslon znak po znak

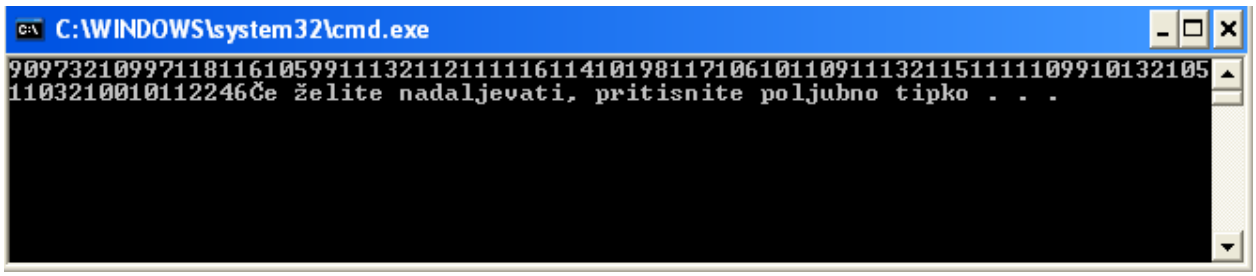
Podano imaš datoteko Mavrica.txt. Napiši metodo, ki na zaslon izpiše besedilo datoteke. Besedilo prepiši na zaslon tako, da pišeš znak po znak na zaslon.



Koda metode:

```
public static void IzpisNaZaslon(string ime){
      if(!File.Exists(ime)){
          Console.WriteLine("Datoteka ne obstaja");
      }else{
          StreamReader dat = File.OpenText(ime);
          int beri = dat.Read();
          while(beri != -1){
            Console.Write((char)beri);
            beri = dat.Read();
          }
             dat.Close();
           }
       }
```

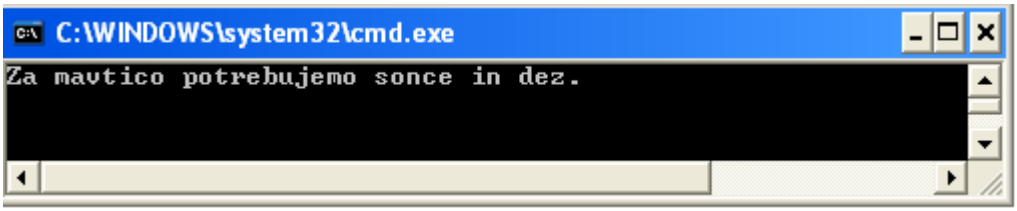
Glavni trik v tej metodi je uporaba metode `Read()`, ki jo uporabimo za branje po znakih. Značilnost te metode je, da znak prebere v obliki njegove kode. Primer datoteke Mavrica zapisane v kodi:



Tak izpis bi dobili, če bi vrstico 8 napisali kot

```
Console.Write(beri);
```

Seveda te kode navadni smrtniki ne znamo brati. Zato moramo besedilo pri izpisu prevesti nazaj v znake. Primer datoteke Mavrica zapisan v obliki, ki smo jo vajeni:



Opomba: Kako računalnik ve, kdaj je konec zapisa na datoteki? Zadnji znak datoteke je oznaka EOF, katere koda je -1. Ko torej preberemo kodo -1 vemo, da smo na koncu besedila zapisanega v datoteki.

Kopija datoteke (po vrsticah)

Sestavimo metodo, ki bo naredil kopijo datoteke.

1. Najprej bomo odprli dve datoteki. Prvo za branje (File.OpenText), drugo za pisanje File.CreateText)
2. potem bomo brali s prve datoteke vrstico po vrstico (metoda ReadLine) in jih sproti zapisovali na izhodno datoteko.
3. 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();
  }
```

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();
  }
```

Datoteka s števili (vsako 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 danio 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();
  while(vrst != null){
    tabela[koliko] = int.Parse(vrst);
    koliko++;
    vrst = beri.ReadLine();
  }
  beri.Close();
  return tabela;
}
```

Kaj če ne vemo koliko je števil:

- najprej en prehod preko datoteke in preštejemo število vrstic

Kaj, če želimo vrniti ravno tabelo "idealne" velikosti, torej točno tako veliko, kot je minimalno potrebno (kot je število vrstic v datoteki)

- Če ne vemo, koliko je podatkov – rešitev prej
- Če vemo, koliko jih je največ:
  - beremo kot prej
  - ko smo naredili tabelo, naredimo novo tabelo ustrezne velikosti in prepišemo le ustrezni del

Zgled: 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:

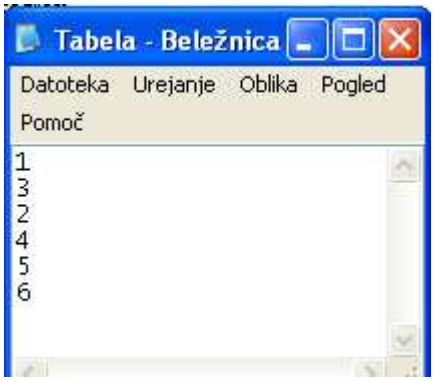
$$[1 \quad 3 \quad 2 \quad 4 \quad 5 \quad 6]$$

Koda metode:

```
using System;
using System.IO;
namespace Pisanje{
class Pisanje{
     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();
        }
    }
}
```

Prikaz podatkov iz datoteke Tabela.txt:



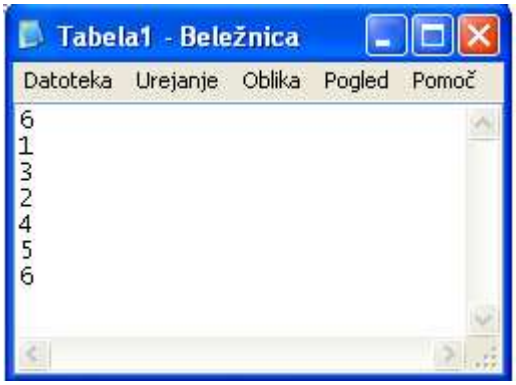
Napisano imamo datoteko z imenom Tabela.txt. Ali sedaj znamo zapisati te iste podatke iz datoteke nazaj v tabelo? Sicer bi šlo tako, kot smo si ogledali v prejšnjem zgledu. Najprej napišemo metodo, ki vrne število vrstic naše datoteka. To metodo potem uporabimo zato, da ustvarimo primerno veliko tabelo 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();
   }
```

Ponoven prikaz datoteke Tabela1.txt:



Ponovitev branje iz tekstovne datoteke

Podatke na datoteke shranjujemo za kasnejšo uporabo. Če želimo podatke iz tekstovne datoteke prebirati, moramo datoteko odpreti v načinu za branje. To nalogo v jeziku C# prevzeme stavek

```
StreamReader vhodTok = File.OpenText(ime_datoteke);
```

Pri odpiranju datotek za branje pogosto navedemo ime neobstoječe datoteke. To povzroči napako med izvajanjem programa. Napako znamo preprečiti. Spomnimo se izraza `File.Exists(ime_datoteke`. V razdelku o pisanju smo ga uporabili zato, da nismo ustvarili datoteke z enakim imenom, kot jo ima že obstoječa datoteka. Torej je tudi pred odpiranjem datoteke za branje smiselno, da prej preverimo, ali datoteka z določenim imenom obstaja.

Da iz datoteke preberemo podatke, so na voljo metode, ki so prikazane v spodnji tabeli. Najdemo jih v razredu `StreamReader`.

C#	Razlaga
ReadLine()	Prebere naslednjo vrstico podatkov z datoteke in jo vrne kot niz.
Read()	Prebere naslednji razpoložljiv znak z datoteke. ***Pozor:*** Metoda vrne kodo prebranega znaka.
ReadToEnd()	Prebere podatke v datoteki od trenutnega mesta v datoteki pa do konca. Podatke vrne kot niz. Navadno to metodo uporabljamo, da preberemo celotno vsebino datoteke.
Peek()	Vrne naslednji znak v datoteki, brez premika na naslednji znak. Če ni na voljo nobenega znaka več, metoda vrne vrednost `-1`.

Odstranimo prazne vrstice

Na datoteki je pesem zapisana po kiticah. Med kiticami so prazne vrstice. Napišimo metodo, ki za dano ime datoteke vse kitice izpiše na zaslon. Pri tem prazne vrstice izpusti.

Vsebina datoteke `Dekle.txt`:	Izpis na zaslon:
Dekle je po vodo šlo *na visoke planine.* *Vodo je zajemala,* *je ribico zajela.* *Ribica jo je prosila:* *oj, pusti me živeti.* *Dekle b'la je usmiljena,* *je ribico spustila.* *Ribica je zaplavala,* *je dekle poškropila.*	*Dekle je po vodo šlo* *na visoke planine.* *Vodo je zajemala,* *je ribico zajela.* *Ribica jo je prosila:* *oj, pusti me živeti.* *Dekle b'la je usmiljena,* *je ribico spustila.* *Ribica je zaplavala,* *je dekle poškropila.*

Koda:

```
C1:   public static void Dekle(string ime){
C2:      // Preverimo obstoj datoteke
C3:      if(!File.Exists(ime)){
C4:         Console.WriteLine("Datoteka " + ime + "ne obstaja.");
C5:         return;
C6:      }
C7:
C8:      // Odpremo datoteko za branje
C9:      StreamReader dat = File.OpenText(ime);
C10:
C11:     // Beremo in izpisujemo (neprazne) vrstice datoteke
C12:     while(dat.Peek() != -1){ // Beremo, dokler ne preberemo kode oznake EOF
C13:        string vrstica = dat.ReadLine();
C14:        if(vrstica.Length != 0){ // Prebrana vrstica ni prazna
C15:          Console.WriteLine(vrstica);
C16:        }
C17:     }
C18:
C19:     // Zapremo datoteko za branje
C20:     dat.Close();
C21:  }
```

Razlaga. Najprej preverimo obstoj datoteke (C3). Če datoteka ne obstaja, izpišemo obvestilo (C4) in prekinemo izvajanje metode (C5). Odpremo datoteko za branje (C9). V vrstici C12 naredimo zanko, ki ponavlja vrstice od C13 do C15 toliko časa, dokler ne pridemo do konca datoteke. To prepoznamo po tem, da je na vrsti za branje znak EOF[2]. Ta znak ima kodo -1. V C13. vrstici beremo posamezno vrstico datoteke. Če je dolžina vrstice različna od 0 (če torej vrstica ni prazna), vrstico izpišemo (C15). Po končanem branju datoteko zapremo (C20).
Zadnji znak datoteke je oznaka EOF, katerega koda je -1. Ko je na vrsti znak s to kodo, vemo, da smo na koncu besedila, zapisanega na datoteki. Oznake EOF ob odprtju datoteke ne vidimo.

Opomba: Tudi datoteke, s katerih v C# beremo, se spodobi zapreti. Sicer neposrednih posledic (kot pri pisanju) ne bo. A če bomo odprto datoteko poskusili odpreti ponovno, se bo program sesul. Prav tako odprte datoteke trošijo sistemske vire računalnika. Zato je dobro, da se navadimo datoteke, potem ko jih ne potrebujemo več, vedno zapreti.

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

[2] Kratica EOF pomeni End-Of-File oziroma konec datoteke.

datoteka za branje obstaja, datoteka za pisanje pa ne (oziroma da njeno morebitno prejšnjo vsebino lahko izgubimo)).

Koraki programa:

- 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 števka, ga bomo prepisali v izhodno datoteko.
- To bomo počeli toliko časa, dokler ne bomo prebrali znaka s kodo -1 (torej znak EOF).

Zapis v C#:

```
C1:   public static void PrepisiBrez(string imeVhod, string imeIzhod){
C2:      StreamReader datZaBranje; // Ustvari podatkovni tok za branje na datoteki
C3:      datZaBranje = File.OpenText(imeVhod);
C4:      StreamWriter datZaPisanje; // Ustvari tok za pisanje na datoteko
C5:      datZaPisanje = File.CreateText(imeIzhod);
C6:
C7:      // Prepis znakov iz ene datoteke na drugo datoteko
C8:      int znak = datZaBranje.Read(); // Prebere en znak
C9:      while (znak != -1){
C10:        // Primerjamo ali je prebrani znak števka
C11:        if (!('0' <= znak && znak <= '9'))
C12:           datZaPisanje.Write("" + (char)znak); // Če ni števka, zapišemo znak
C13:        znak = datZaBranje.Read();
C14:     }
C15:
C16:     // Zapremo tokova datotek
C17:     datZaBranje.Close();
C18:     datZaPisanje.Close();
C19:  }
```

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).

Zapis v C#:

```
C1:   public static bool Primerjava(string ime1, string ime2){
C2:      // Odpremo obe datoteki za branje
C3:      StreamReader dat1 = File.OpenText(ime1);
C4:      StreamReader dat2 = File.OpenText(ime2);
C5:
C6:      // Preberemo vsebino prve in druge datoteke
C7:      string beri1 = dat1.ReadToEnd();
C8:      string beri2 = dat2.ReadToEnd();
C9:
C10:     // Zapremo obe datoteki za branje
C11:     dat1.Close();
C12:     dat2.Close();
C13:
C14:     // Primerjamo vsebini
C15:     return (beri1.Equals(beri2));
C16:  }
```

Razlaga. Najprej odpremo datoteki (C3 – C4). V niz `beri1` s pomočjo metode `ReadToEnd()` preberemo celotno vsebino datoteke `dat1` (C7). V niz `beri2` shranimo celotno vsebino datoteke `dat2` (C8). Zapremo datoteki (C11 – C12). Na koncu vrnemo rezultat, ki ga dobimo pri primerjavi niza `niz1` z nizom `niz2`.

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,
- preverimo obstoj datotek,

- 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.

Koda:

```
C1:   using System;
C2:   using System.IO;
C3:   public class Zamenjava{
C4:      public static void Main(string[] args){
C5:         // Vnos imen datotek
C6:         Console.Write("Vnesi ime vhodne datoteke: ");
C7:         string vhod = Console.ReadLine();
C8:         Console.Write("Vnesi ime izhodne datoteke: ");
C9:         string izhod = Console.ReadLine();
C10:
C11:        // Preverjanje obstoja: vhodna mora obstajati, izhodna pa ne
C12:        if (!File.Exists(vhod) || File.Exists(izhod)) {
C13:           Console.WriteLine("Napaka v imenu datotek.");
C14:           return;
C15:        }
C16:
C17:        // Odprtje datotek
C18:        StreamReader branje = File.OpenText(vhod);
C19:        StreamWriter pisanje = File.CreateText(izhod);
C20:
C21:        // Prenos podatkov
C22:        while (branje.Peek() != -1) { // Do konca datoteke
C23:           char znak = (char)branje.Read();
C24:           if (znak == 'a') {
C25:              pisanje.Write("ap"); // Zadnji 'a' bo dodal naslednji stavek
C26:           }
C27:           pisanje.Write(znak);
C28:        }
C29:
C30:        // Zapremo za datoteke
C31:        branje.Close();
C32:        pisanje.Close();
C33:     }
C34:  }
```

Razlaga. Preberemo ime vhodne in izhodne datoteke (C6 - C9). Nato preverimo obstoj datotek (C12). Če vhodna datoteka ne obstaja ali izhodna datoteka obstaja, izpišemo obvestilo (C13) ter prekinemo izvajanje metode (C14). Datoteki odpremo za branje oziroma pisanje (C18 - C19). Nato z zanko `while` beremo posamezne znake iz vhodne datoteke in jih prepisujemo na izhodno datoteko (C22 - C28). Če je prebran znak 'a' (C24), pred njim na izhodno datoteko zapišemo še niz "ap" (C25). Znak 'a' iz vhodne datoteke na ta način zapišemo na izhodno datotako z nizom "apa". Zanko končamo, ko preberemo kodo oznake EOF. Po koncu prepisovanja znakov datoteki zapremo (C31 - C32).

Kopiranje datotek

Sestavimo metodo `public static void Kopiranje(string vhod, string izhod)`, ki vsebino ene datoteke prepiše na drugo datoteko. Ime prve datoteke naj predstavlja prvi argument, ime druge datoteke pa naj predstavlja drugi argument. Predpostavimo, da je obstoj datotek že preverjen. Kopiranje izvedemo tako, da prepisujemo vrstico po vrstico.

Da preverimo, če smo že na koncu datoteke, lahko uporabimo tudi rezultat metode `ReadLine()`. Če omenjeno metodo uporabimo na neobstoječi vrstici (če smo torej že na koncu datoteke), nam metoda vrne vrednost `null`.

Zapis v C#:

```
C1:   public static void Kopija(string vhod, string izhod){
C2:      // Odpremo datoteko za branje in pisanje
C3:      StreamReader beri = File.OpenText(vhod);
C4:      StreamWriter pisi = File.CreateText(izhod);
```

```
C5:
C6:       // Beremo in kopiramo posamezne vrstice
C7:       string vrsta = beri.ReadLine();
C8:       while(vrsta != null){// Beremo toliko časa, dokler ne preberemo
C9:                            // vseh vrstic
C10:         pisi.WriteLine(vrsta);
C11:         vrsta = beri.ReadLine();
C12:      }
C13:
C14:      //Zapremo datoteko za branje in pisanje
C15:      beri.Close();
C16:      pisi.Close();
C17:   }
```

Razlaga. Najprej datoteki odpremo za branje oziroma pisanje (C3 - C4). Nato z zanko `while` beremo posamezne vrstice vhodne datoteke in jih prepisujemo v izhodno datoteko (C7 - C12). Zanko končamo, ko prebrane vrstice ni (ustrezni niz dobi vrednost `null`). Po končani zanki datoteki zapremo (C15 - C16).
Če naloga ne bi zahtevala prepisovanje po vrsticah, bi seveda lahko naenkrat prebrali kar celotno vsebino datoteke. Prepisovanje pa bi lahko izvedli tudi znak po znak.

Vaje

1. Sestavi metodo, ki tabelo nizov prepiše na datoteko, določeno z nizom, ki je tudi parameter metode. Vsak niz naj bo v svoji vrstici.
2. Dana je tekstovna datoteka. Izpiši jo tako, da zamenjaš sode in lihe vrstice.
3. Dana je metoda, ki naj bi preštela število vrstic v datoteki. A v njej so napake. Odpravi jih!

```
1:     public static int PrestejVrstice(string ime){
2:        StreamReader dat = File.OpenText(ime);
3:        int vrstice = 0;
4:        string vrst = dat.ReadLine();
5:        while(vrst != ""){
6:           vrstice = vrstice + 1;
7:        }
8:        dat.Close();
9:        return vrstice;
10:    }
```

4. Z metodo

```
1.  public static int PrestejZnake(string ime){
2.        StreamReader dat = File.OpenText(ime);
3.        int znaki = 0;
4.
5.        while(dat.ReadLine() != null ){
6.           znaki = znaki + dat.ReadLine().Length;
7.           //dat.readLine() je niz,
8.           // pristejemo njegoco dolzino
9.        }
10.       dat.Close();
11.       return znaki;
12.   }
```

naj bi prešteli število znakov v datoteki. Žal se metoda sicer prevede, očitno pa ne dela prav. Odpravi napake!

5. Sestavi metodo, ki "obrne" dano datoteko (zadnja vrstica postane prva, predzadnja druga …). Prepodstavi, da datoteka zagotovo nima več kot 100 vrstic.
6. Arheologi so v jami blizu Gize našli čuden papirus. Besedilo na njem je bilo videti zelo čudno, a po drugi strani zelo podobno našemu. Dolgo so poskušali vse, da bi ga razvozlali. Na koncu se je oglasil 6 letni Mihec, sin glavnega arheologa: "Oči, zakaj si pa na ta papir pisal v napačno smer?" In res. Šlo je za povsem običajni tekst, le besedilo je bilo zapisano od desne proti levi. Ker pa je nam tako besedilo malček zporno brati, pomagaj arheologom in sestavi metodo, ki za dano ime datoteke sestavi novo datoteko z ravno obrnjenim imenom in enako končnico (iz bla.txt torej naredi alb.txt, iz mojaDat.cs pa taDajom.cs). Ta nova datoteka naj ima ravno obrnjene vrstice prvotne datoteke.
7. Na spletnem naslovu http://www.ljse.si/ najdeš podatke o gibanju cen različnih vrednostnih papirjev (delnic) - glej arhiv enotnih tečajev. Izberi si delnico, poberi s spletne strani vse možne podatke o gibanju njenega tečaja. Podatke dobiš tako, da najprej izbereš delnico, v koledarju izbereš obdobje, za katere te gibanje delnice zanima (denimo 1.1.2007 - 1.11.2007). Nato klikneš na Shrani. Podatki se shranijo na datoteko v formatu CSV (comma separated values) v taki obliki: `3.10.2001;1.983,67;100` Podatki so ločeni s ;. Prvi podatek je datum, drugi SBI (slovenski borzni indeks), tretji pa vrednost delnice. Prvi in drugi podatek nas NE zanimata. Pripravi metodo, ki bo iz podatkov kot jih boš pobral s spletne strani (torej iz datoteke), ustvarila ustrezno tabelo z vrednostmi delnice.
8. Napiši metodo BrisiKomentarje, ki sprejme imeni vhodne in izhodne datoteke. Nato na izhodno datoteko prepiše tiste vrstice iz vhodne datoteke, ki se ne začnejo z znakom '%'.
9. Napiši metodo, ki prepiše datoteko imevhodna v datoteko imeizhodna. Pri tem vse znake 'a' zamenja z znaki 'e'.
10. Sestavite program, ki iz ukazne vrstice prebere pot do datoteke, jo odpre in na zaslon izpiše prvih deset vrstic njene vsebine (če je datoteka krajša, jih lahko izpiše tudi manj).
11. Napiš metodo Primerjaj, ki sprejme imeni dveh datotek in primerja njuno vsebino. Če sta datoteki enaki, metoda vrne vrednost true, sicer vrne false. Pozor: datoteki nista nujno enako dolgi. Pri tem si lahko pomagaš z naslednjo metodo, ki pa ima napake!

```
public static bool Primerjaj(string ime1, string ime2) {
  if (ime1 == ime2) return true; // gre za isto datoteko
  StreamReader dat1 = File.OpenText(ime1);
  StreamReader dat2 = File.OpenText(ime2);
  bool enaki = true;
  bool konec = false;

  while(!konec){
    string s1 = dat1.ReadLine();
    string s2 = dat2.ReadLine(); // ce je 2. datoteke ze konec,
                                 // ima s2 vrednost null
    if(!(s1.equals (s2))){
      enaki = false;
     konec = true;
    }
  }

  if(s2 != null){  // 2. datoteka je daljsa od prve
    enaki = false;
  }
  dat1.Close();
```

```
    dat2.Close();
    return enaki;
  }
```

12. Generiraj N (podatek) naključnih števil med 1 in 100. Rezultate zapiši v datoteko `rezultatN.txt`. V datoteki naj bo za vsak korak zapisana vrstica `#zap št. #vrednost (npr. 1 23)`

13. Na datoteki manekenke.dat so shranjeni podatki o velikosti manekenk. V vsaki vrsti je zapisana velikost manekenke v centimetrih (celo število), nato pa sledita ime in priimek. Polja so ločena z dvopičji. Primer datoteke:

```
179:Cindy:Crawford
182:Naomi:Campbel
185:Nina:Gazibara
180:Elle:Mac Perhson
180:Eva:Herzigova
```

Napiši program, ki prebere datoteko manekenke.dat in na zaslon izpiše višino, ime in priimek največje in najmanjše manekenke. V zgornjem primeru bi program deloval takole:

```
Najmanjsa je Cindy Crawford, ki meri 179 cm.
Najvecja je Nina Gazibara, ki meri 185 cm.
```

14. Dana je metoda PrestejBesede, ki sprejme niz znakov in vrne število besed v nizu. Za besedo šteje poljubno neprazno zaporedje znakov med dvema presledkoma. Za presledke štejemo tudi tabulator '\t' in znak za novo vrsto '\n'. Na primer, niz znakov `"Kdor ne delja, naj ne je."` ima 6 besed, niz znakov `"++ !! ax"` ima 3 besede, niz `"Ena dva,tri stiri pet"` pa ima 4 besede. (Več zaporednih presledkov seveda šteje kot en presledek.)

```
public static int PrestejBesede(string s){
  int steviloBesed=0;
  bool presledki=true; //pove, ali se trenutno nahajamo v besedi
  for(int i = 0; i < s.Length; i = i + 1){
    if(presledki){
      if(!Char.IsWhiteSpace(s[i])){
          // iz presledkov smo prisli v besedo
        steviloBesed++;
        presledki = false;
      }
    }
    else
      if(Char.IsWhiteSpace(s[i])){
        // iz besede smo prisli v presledke
      presledki = true;
    }
  }
  return steviloBesed;
}
```

S pomočjo te metode preštej število besed v pesnitvi *Krst pri Savici*, ki ga najdeš na primer na http://haka.fmf.uni-lj.si/pra-racunalnistvo-1/arhiv-2003/lekcija08/krst.txt

15. Napiš program Tezisce, ki iz datoteke z imenom "podatki.dat" prebere koordinate točk in izpiši na zaslon koordinate njihovega težišča. Koordinate vsake točke so napisane v svoji vrstici in ločne s presledkom. Na primer, datoteka

```
1.2 3.41
0.0 1.8
9.0 3.19
2.2 2.2
```

vsebuje podatke, ki predstavljajo toče s koordinatami (1.2, 3.41), (0.0, 1.8), (9.0, 3.19) in (2.2, 2.2). Za ta primer ima težišče koordinate (3.1, 2.65).
Nauk: koordinate težišča so povprečne vrednosti koordinat točk.

16. Janko in Metka sta za sporazumevanje izumila posebno kodo: Vsako črko v sporočilu sta zamenjala s črko, ki v (angleški) abecedi leži 6 črk za originalno. Menjava je seveda ciklična, tako da črko *a* zamenjata s črko *f*, črko *z* pa s črko *e*. Napiši metodi `zakodiraj` in `odkodiraj`, ki sprejmeta ime originalne in kodirane datoteke. Seveda ločil in ostalih znakov ne kodirata – pomagaj si z metodami `Char.IsLetter(char c)`, ki za dani znak pove, če je črka. `Char.IsLower(char c)`, ki za dani znak pove, če je mala črka, `Char.IsUpper(char c)`, ki za dani znak pove, če je velika črka.

17. Marko je agent pri varnostni agenciji ČUK. Poskrbeti mora, da bodo datoteke varno prišle od ene agenture do druge. V ta namen bo napisal metodo Razdeli(ime, koliko), ki iz datoteke z imenom ime.cuk ustvari koliko datotek z imenom ime_1.cuk, ime_2.cuk, ..., ime_koliko.cuk. Na prvi datoteki (ime_1.cuk) je 1, 1 + koliko, 1 + 2*koliko, … - ti znak iz vsake vrstice datoteke ime.cuk, na drugi 2, 2 + koliko, 2 + 2*koliko, … - ti znak iz vsake vrstice, …

    Prav tako je napisal še metodo Zdruzi(ime, koliko), ki izvaja obratno operacijo. A kot zakleto se je koda metod zgubila. Pomagaj Marku in metodi napiši na novo.

18. Napiši metodo, ki prepiše datoteko vhodna v datoteko izhodna. Pri tem naj vse samoglasnike podvoji.

19. Dana je tekstovna datoteka. Izpiši jo tako, da bodo vrstice zapisane v obratnem vrstnem redu.

    Primer:

Tekstovna datoteka:	Izpis:
Velik korak za človeka, mali korak za človečnost.	, akevolč az karok koleV .tsončevolč az karok ilam

    Namig: Napiši metodo, ki bo obrnila poljuben niz.

20. Sestavi metodo, ki niz prepiše na datoteko, tako da bo vsak znak niza v svoji vrstici. Niz in ime datoteke sta vhodna podatka.

21. Napiši metodo BrišiVrstice, ki sprejme ime vhodne in izhodne datoteke. Nato na izhodno datoteko prepiše tiste vrstice iz vhodne datoteke, ki se ne začnejo s samoglasnikom.

22. Poznamo imeni dveh datotek. Zanima nas, ali imata ti dve datoteki enako število znakov. Napiši metodo Primerjaj, ki vrne true, če je število znakov enako, sicer false.

23. Napiši metodo Potroji, ki sprejme ime vhodne in izhodne datoteke. Nato na izhodno datoteko prepiši vse vrstice vhodne datoteke in sicer tako, da se bo vsaka vrstica zaporedoma potrojila.

    Primer:

Vhodna datoteka:	Izhodna datoteka:
Velik korak za človeka, mali korak za človečnost.	Velik korak za človeka, Velik korak za človeka, Velik korak za človeka, mali korak za človečnost.

	mali korak za človečnost. mali korak za človečnost.

24. Dana je tekstovna datoteka. Izpiši vsa števila, ki se pojavijo v njej.
25. Tekstovna datoteka naj bo sestavljena iz vrst, napolnjenih s celimi števili, ločenimi s presledki. Sestavi statično metodo MinMax(ime datoteke), ki vrne maksimum od minimumov števil v vsaki vrsti v datoteki.

    Primer:
    1 2 3 4
    5 2
    0 1 2 -1

    Rezultat primera: 2
26. Napišite program, v katerem napolnite tekstovno datoteko s 100 števili. Ta števila potem preberite iz datoteke in izpišite povprečje vseh lihih števil.
27. Napišite program, v katerem v tekstovni datoteki z besedilom zamenjate vsak samoglasnik z * in tvorite novo datoteko s tako zamenjanimi črkami.
28. Napiši metodo, ki za vhodni podatek sprejme ime vhodne in izhodne datoteke. Metoda naj prepiše vrstice v vhodni datoteki na izhodno datoteko, pri čemer združi vse vrstice v eno vrstico. Na primer, če je na datoteki Dat1.txt zapisano

    ```
    prva vrstica,
    druga vrstica,
    tretja vrstica,
    četrta vrstica,
    peta vrstica.
    ```

    potem je zapis na datoteki Dat2.txt

    ```
    prva vrstica, druga vrstica, tretja vrstica, četrta vrstica, peta
    vrstica.
    ```

29. Napiši program Tezisce, ki iz datoteke z imenom "Podatki.txt" prebere koordinate točk in izpiše na zaslon koordinate njihovega težišča. Koordinate vsake točke so napisane v svoji vrstici in ločene s presledkom. Na primer, datoteka

    ```
    1.2 3.41
    0.0 1.8
    9.0 3.19
    2.2 2.2
    ```

    vsebuje podatke, ki predstavljajo točke s koordinatami (1.2, 3.41), (0.0, 1.8), (9.0, 3.19) in (2.2, 2.2). Za ta primer ima težišče koordinate (3.1, 2.65).

30. Na datoteki Manekenke.txt so shranjeni podatki o velikosti manekenk. V vsaki vrsti je zapisana velikost manekenke v centimetrih (celo število), nato pa sledita ime in priimek. Polja so ločena s presledki. Primer datoteke:

    ```
    179 Cindy Crawford
    182 Naomi Campbel
    185 Nina Gazibara
    180 Elle Mac Perhson
    180 Eva Herzigova
    ```

Napiši metode, ki za vhodni podatek sprejmejo ime metode, kot izhodni podatek pa nam vrnejo:

- Število, ki nam pove koliko manekenk je v datoteki.
- Povprečno telesno višino manekenk.
- Višino tiste manekenke, ki ima najdaljše ime.
- Vse priimke manekenk (pozor, priimek četrte manekenke je Mac Perhson!)

Rešitve za podani primer: 5, 181.2, 179, "Crawford, Campbel, Gazibara, Mac Perhson, Herzigova"

31. Sestavi metodo, ki za vhodni podatek sprejme ime datoteke, na kateri so zapisane koordinate točk v ravnini. Na primer, vsebina takšne datoteke bi lahko bila:

```
-1.060 -3.456
2.850 0.056
0.000 2.718
-9.023 6.003
2.540 9.023
```

V vsaki vrstici sta zapisani koordinati x in y ene točke, ločeni s presledkom. Metoda naj vrne število točk v prvem kvadrantu (to so tiste točke, ki imajo obe koordinati strogo pozitivni). Na primer, za zgornjo datoteko, program izpiše 2, ker sta točki (2.850, 0.056) in (2.540, 9.023) v prvem kvadrantu.

32. Direktor podjetja je dobil datoteko Priporocilo.txt na kateri je pisalo:

Spoštovani gospod direktor,

Janeza Novaka, mojega asistenta pri delu, vedno vidite, kako trdo dela v svoji mali pisarni. Janez dela neodvisno in ne lenari ali se pogovarja s sodelovci. Nikoli se ne zgodi, da bi zavrnil kakšnega sodelovca, ki potrebuje pomoč. Do sedaj je vedno končal z delom pravočasno. Zelo pogosto si vzeme podaljšan delovni čas, da konča svoje delo, pri čemer včasih preskoči odmor. Janez je takšen delavec, ko nima absolutno nobenega spodrslajaja pri opravljenih delih, ima visoke dosežke in je širokega znanja na njegovem področju. Moje mnenje je, da ga lahko takoj uvrstimo med tiste najbolj vzorne delovce, ki jih nikoli ne odpustimo. Prav tako vam vljudno predlagam, da je moj predlog o napredovanju tega izjemnjega, vzornega in nepogrešljivega delavca izvršen kakor hitro je mogoče.

Lep pozdrav!

Že se je spravil pisati predlog za napredovanje, ko je po e-pošti prispel dopis:

> *Direktor!*
> *Ta idiot je stal za menoj, ko sem pisal prejšnje priporočilo.*
> *Prosim znova preberite vsako drugo vrstico tega pisma.*

Direktor je sedaj povsem zmeden. Pomagaj mu in sestavi program, ki na datoteko NovoPriporocilo.txt napiše vsako drugo vrstico prvotnega priporočil!

33. Sestavi metodo, ki izpiše seznam vseh datotek v imeniku (directory) skupaj z vsemi podimeniki. V ta namen poišči ustrezne metode v C#, s pomočjo katerih za dano ime datoteke ugotovi, ali gre za "navadno" datoteko ali imenik.

Primer:
klic DirR (@"D:\j2sdk1.4.1") izpiše (... označujejo spuščene vrstice):

```
D:\j2sdk1.4.1\bin\appletviewer.exe
D:\j2sdk1.4.1\bin\extcheck.exe
...
D:\j2sdk1.4.1\bin\tnameserv.exe
D:\j2sdk1.4.1\COPYRIGHT
D:\j2sdk1.4.1\demo\applets\Animator\Animation.class
...
D:\j2sdk1.4.1\demo\applets\Animator\Animator.java
D:\j2sdk1.4.1\demo\applets\Animator\audio\0.au
...
D:\j2sdk1.4.1\demo\applets\Animator\audio\spacemusic.au
D:\j2sdk1.4.1\demo\applets\Animator\DescriptionFrame.class
...
D:\j2sdk1.4.1\demo\applets\Animator\example4.html
D:\j2sdk1.4.1\demo\applets\Animator\images\Beans\T1.gif
...
D:\j2sdk1.4.1\demo\applets\Animator\images\Beans\T9.gif

(izpuščenih je še približno 20 strani datotek)
```

34. Globina podimenikov
    Uporabi idejo prejšnje naloge in ugotovi, kako "globoko" segajo podimeniki danega imenika. Torej za imenik brez poimenikov bo rezultat 0. Če bo dani imenik vseboval podimenik, ki bo vseboval podimenik, ki bo vseboval podimenik, ki bo vseboval le običajne datoteke, bo rezultat 3.
35. Sestavi metodo

    ```
    public static void PrepisBrez(string imeVhod, string imeIzhod)
    ```

    ki tekstovno datoteko, katere ime je v `imeVhod` *prepiše na novo datoteko z imenom, kot ga določa* `imeIzhod`*. Pri tem naj spusti vse številke, struktura vrstic pa naj se ohrani. Primer:*

Vhodna datoteka	Izhodna datoteka
Triglav je visok 2864m. To je visoko. *Dne 25.3.2008 pišemo izpit iz* *predmeta Programiranje 2.* ***To je predzadnja vrstica.*** ***Cela datoteka ima 5 vrstic.***	*Triglav je visok m. To je visoko.* *Dne .. pišemo izpit iz* *predmeta Programiranje .* *To je predzadnja vrstica.* *Cela datoteka ima vrstic.*

36. Dana je tabela znakov. V tabeli so znaki bodisi presledki, bodisi znaki `'*'`. Tabela je napisana na datoteki in sicer tako, da je v prvi vrstici datoteke podana dimenziji tabele (najprej cello število, kipredstavlja število vrstic in nato presledek in nato celo število, ki predstavlja število stolpcev), nato pa vrstica tabele ustreza vrstici na datoteki. Sestavi rekurzivno metodo

    ```
    Zapolni(string ImeVhodDat, string imeIzhodDat, int vr, int st)
    ```

    ki prebere podatke z vhodne datoteke in ustvari datoteko, ki predstavlja tabelo, kjer z znakom '+' "zapolni" zaprt del lika, katerega koordinati (indeksa) sta tretji in četrti parametra metode. Indekse štejemo od 0 dalje.

    Tako klic `Zapolni("Vh.txt", "Izh.txt", 2, 8)` ustvari naslednjo datoteko

Vhodna datoteka Vh.txt

```
10 9
* * * * * * * * * *
*           * *     *
*     *     *       *
*     * * * * *     *
* * * *             *
*     *     *   *   *
* *   *     * * *   *
*     *         *   *
* * * * * * * * * * *
```

Rezultat (Izh.txt)

```
10 9
* * * * * * * * * *
*           * * + + *
*     *     * + + + *
*     * * * * * * + *
* * * * + + + + + + *
*     * + * + * + + *
* *   * + * * * * + *
*     * + + + * + + *
* * * * * * * * * * *
```

Dva elementa tabele sta med seboj povezana po štirih smereh (in ne morda osmih). Predpostavi, da imamo na robovih tabele vedno znake '*'. Če je na ustreznem mestu znak #, se seveda tabela ne spremeni (torej je izhod nova datoteka z identično vsebino kot jo ima vhodna datoteka)! Prav tako dobimo izhodno datoteko z identično vsebino kot je vhodna datoteka, če so koordinate izven tabele.

37. Napišite program, v katerem tekstovno datoteko napolnite s 100 poljubnimi celimi števili. Ta števila potem preberite iz datoteke in izpišite povprečje vseh lihih.

38. Napišite program `Sestej`, ki prebere ime datoteke, v kateri so zapisana cela števila, vsako v svoji vrsti. Program naj na zaslon izpiše vsoto števil iz datoteke. Denimo, da je v datoteki `Stevila.txt` zapisano

```
10
4
100
12
```

Spodnji okvir ponazarja izvajanje programa:

```
Ime datoteke: Sestej.txt
Vsota: 126
```

39. Napišite program, ki prebere imeni vhodne in izhodne datoteke ter vhodno datoteko prepiše v izhodno. Pri tem naj vse pojavitve znaka 'e' zamenja z znakom 'E'.

40. Napišite program `VrednostPolinoma`, ki iz datoteke "`polinom.txt`" prebere celo število `t` in celoštevilske vrednosti polinoma $p(x) = a_0 + a_1x + \ldots + a_nx^n$ ter izpiše na zaslon vrednost `p(t)`. Datoteka vsebuje dve vrstici: v prvi vrstici je zapisano celo število `t`, v drugi vrstici pa so koeficienti polinoma `p`, ločeni s presledki.

Primer:
Zapis v datoteki
```
2
1 2 0 3 4
```
Za ta primer mora program izpisati na zaslon število `29`, ker podatki predstavljajo `t=2` in polinom $p(x)=1+2x+4x^3+3x^4$, od koder dobimo $p(t)=1+2\cdot2+4\cdot2^3+3\cdot2^4=29$.

41. Tekstovna datoteka naj bo sestavljena iz vrst, napolnjenih s celimi števili, ločenimi s presledki. Sestavite statično metodo `MinMax(imeDatoteke)`, ki vrne maksimum minimumov števil v vsaki vrstici v datoteki.

Primer: Če je v datoteki `Stevila.txt` zapisano

```
2 3 4
5 -1
8 9 6 3
```

je rezultat metode `MinMax("Stevila.txt")` enak `3`.

42. Ustvarite tekstovno datoteko `APJ.txt`. V to datoteko zapišite poljubno število stavkov (berete jih preko tipkovnice, znak za konec vnosa je `*`). Vsak stavek naj bo na datoteki v svoji vrstici, med vrsticami pa naj bo po ena prazna vrstica. Napišite metodo, ki dobi za parameter ime te datoteke. Vrne naj, koliko znakov vsebuje celotna datoteka! Prazna vrstica seveda ne vsebuje nobenega znaka.

43. Dana je tekstovna datoteka `Naloga.txt`. V vsaki vrstici te datoteke so po tri cela števila, med seboj ločena s presledkom. Datoteko obdelajte tako, da za vsako vrstico na zaslon izpišete vsoto vseh treh števil, na koncu pa še skupno vsoto vseh števil!

44. V tekstovni datoteki so zapisani podatki o porabi bencina za posamezne tipe vozila. V vsaki vrstici je zapisan tip vozila, nato pa podatek o porabi goriva na 100 km (realno število). Koliko vozil je v datoteki? Ugotovite in izpišite tip vozila z najmanjšo porabo goriva. Podatka o tipu vozila in porabi goriva sta razmejena z ločilnim znakom |.

45. Napišite program, ki v poljubni tekstovni datoteki prešteje vse števke in na koncu izpiše, kolikokrat se vsaka števka pojavi v tej datoteki. Ime datoteke programu podamo preko konzolnega okna.

46. V datoteki `realna.txt` so zapisana realna števila (vsako v svoji vrstici). Napišite program, ki ugotovi in izpiše, koliko je vseh števil v datoteki in kakšen je procent števil 0.

Primer izpisa:
```
V datoteki je 20 stevil, od tega 25 procentov nicel.
```

Dodatno gradivo

Tu je navedeno gradivo, ki ga načeloma v sklopu predavanj in vaj ne bomo uporabljali. Navedene so določene dodatne metode. Prav tako so uporabljeni posamezni prijemi iz "stare" snovi, ki jih nismo obravnavali. Ta del je zgolj informativne narave.

Dodatne metode imenskega prostora System.IO

Kot smo omenili, za delo z datotekami v C# potrebujemo imenski prostor **System.IO**, ki vsebuje kopico **razredov**, za upravljanje z imeniki (direktoriji), datotekami in potmi do datotek. Razredi tega imenskega prostora pa vsebujejo cel kup metod za raznorazne vhodno-izhodne operacije, med katerimi so za nas najpomembnejše metode za kreiranje, zapisovanje, branje in na splošno manipuliranje s tekstovnimi in binarnimi datotekami.

Razredi imenskega prostora System.IO za delo z imeniki, datotekami in potmi do datotek

Razred	Razlaga
Directory	Uporablja se za kreiranje, urejanje, brisanje ali pridobivanje informacij o imenikih.
File	Uporablja se za kreiranje, urejanje, brisanje ali za pridobivanje informacij o datotekah.
Path	Uporablja se za pridobivanje informacij o poteh do datotek.

Najpomembnejše metode razreda Directory

Metoda	Razlaga
Exists(path)	Vrne logično vrednost, ki ponazarja ali nek imenik obstaja ali ne.
CreateDirectory(path)	Kreira imenike v navedeni poti.
Delete(path)	Brisanje imenika in njegove vsebine.
GetFiles(path)	Pridobivanje imen datotek navedene poti

Primer uporabe:

```
//Kreiranje novega imenika C# 2005 in v njem še podimenika datoteke
string dir ="C: \\C# 2005\\Datoteke\\";
//ali
//znak @ pred definicijo pomeni, da znak \ v stringu ne predstavlja escape sekvence
string dir1 = @"C: \C# 2005\Datoteke\";
if (!Directory.Exists(dir))  //če imenik še ne obstaja, ga skreiramo
    Directory.CreateDirectory(dir);
```

Najpomembnejše metode razreda File

Metoda	Razlaga
Exists(path)	Vrne logično vrednost, ki ponazarja ali neka datoteka obstaja ali ne.
Delete(path)	Brisanje datoteke.
Copy(source, dest)	Kopiranje datoteke iz izvorne poti (**source**) do končne poti (**dest**).
Move(source, dest)	Premik datoteke iz izvorne poti (**source**) do končne poti (**dest**).

Napisanih je le nekaj najpomembnejših metod razredov **Directory** in **File**. Ostale metode in njihovo uporabo lahko dobimo v sistemu pomoči, ki je sestavni del okolja C#.

Primer uporabe:

```
string dir = @"c:\C# 2005\Datoteke\";
string pot = dir + "Izdelki.txt";
//preverimo obstoj datoteke Izdelki.txt v navedenem imeniku (c:\C# 2005\Datoteke\)
if (File.Exists(pot))
    File.Delete(pot);  //če datoteka obstaja, jo pobrišemo
```

```
/*Preveri, če na disku C že obstaja mapa z imenom Vaje C#. Če še ne obstaja, jo kreiraj in v
mapi zgeneriraj datoteki Vaja1 in Vaja2 (datoteki bosta seveda PRAZNI!).*/

static void Main(string[] args)
{
    string dir = "C: \\Vaje C#";
    //ALI PA:  string dir = @"C:\Vaje C#";
    //Preverimo, če imenik C:\Vaje C# že obstaja - če še ne obstaja, ga skreiramo
    if (!Directory.Exists(dir))
        Directory.CreateDirectory(dir);

    string datoteka = "Vaja1";
    string datoteka1 = "Vaja2";
    //Za datoteko Vaja1 preverimo obstoj v imeniku C:\Vaje C#: če že obstaja,jo prej pobrišimo
    if (File.Exists(dir + "\\" + datoteka))
    {
         Console.WriteLine("Datoteka Vaja1 že obstaja in bo pobrisana!");
         File.Delete(dir + "\\" + datoteka);  //če datoteka obstaja, jo pobrišemo
    }
    //skreirajmo NOVO datoteko Vaja1 v imeniku C:\Vaje C#
    File.Create(dir + "\\" + datoteka);
    //skreirajmo NOVO datoteko Vaja2 v imeniku C:\Vaje C#
```

```
    File.Create(dir + "\\" + datoteka1);
    //S pomočjo metode GetFiles imena vseh datotek izbrane mape shranimo v tabelo datoteke
    string[] datoteke = Directory.GetFiles(dir);
    Console.WriteLine("Seznam datotek imenika Vaje C# na disku C:");
    for (int i = 0; i < datoteke.Length; i++) {
       string imedatoteke = datoteke[i]; //izpišimo imena vseh datotek mape C:\Vaje C#
       Console.WriteLine(imedatoteke);
    }
}
```

Dodatne naloge z datotekami

1. Preveri, če na disku D že obstaja imenik d:\C#\Vaje\Datoteke. Če še ne obstaja, ga ustvari, nato pa v njem ustvari datoteki Vaja1.txt in Vaja2.txt. Če imenik že obstaja, najprej preveri, če datoteki Vaja1.txt in Vaja2.txt že obstajata – če ne, ju skreiraj.

2. Preveri, če na disku C že obstaja imenik z imenom, ki ga vneseš preko tipkovnice. Če imenik že obstaja, ugotovi in izpiši, koliko datotek vsebuje. Izpiši tudi imena vseh datotek.

3. Ustvari poljubni imenik in v njej datoteko s poljubnim imenom – imeni vnese uporabnik preko tipkovnice. Pred kreiranjem preveri, če imenik oz. datoteka že obstaja.

4. Ustvari drevesno strukturo imenikov d:\P1\C#\VajeC#\Letnik_1

Delo s podatkovnimi tokovi

Ko uporabljamo razrede imenskega prostora **System.IO** za delo z vhodno izhodnimi operacijami, lahko

Za delo z vhodno-izhodnimi (**I/O** – **Input/Output**) operacijami s tekstovnimi in binarnimi datotekami, **.NET Framework** uporablja tokove (**streams**). Tok (**stream**) si lahko predstavljamo kot pretakanje podatkov iz ene lokacije na drugo. Izhodni tok (**otput stream**) si torej predstavljamo kot tok podatkov z internega pomnilnika aplikacije v datoteko na disku, vhodni tok (**input stream**) pa kot tok podatkov z diska v interni pomnilnik. Pri delu s tekstovnimi datotekami uporabljamo tekstovni tok podatkov (**text stream**), pri delu z binarnimi datotekami pa binarni tok (**binary stream**).

Tokovi podatkov za delo z datotekami

Podatkovni tok	Razlaga
Text	Uporablja se za prenos tekstovnih podatkov.
Binary	Uporablja se za prenos binarnih podatkov.

V tem razdelku bodo prikazani razredi imenskega prostora **System IO**, ki jih uporabljamo za delo s tokovi in datotekami. Za kreiranje toka, ki nas poveže z datoteko, tako npr. uporabimo razred **FileStream**. Za branje podatkov iz datoteke preko tekstovnega toka uporabimo npr. razred **StreamReader**, za branje podatkov iz binarne datoteke preko binarnega toka pa razred **BinaryReader**.

Vrste podatkovnih tokov

Razred	Razlaga
Stream	Splošen podatkovni tok
FileStream	Zagotavljanje dostopa do vhodnih in izhodnih datotek (podatkovni tok namenjen

	datotekam).
StreamReader	Uporablja se za branje tekstovnih podatkov v podatkovni tok (npr. iz tekstovne datoteke).
StreamWriter	Uporablja se za zapisovanje toka tekstovnih podatkov (npr. v tekstovno datoteko).
BinaryReader	Uporablja se za branje binarnih podatkov v podatkovni tok (npr. iz binarne datoteke).
BinaryWriter	Uporablja se za zapisovanje toka binarnih podatkov (npr. v binarno datoteko).

Razred **Stream** je osnovni razred za vse podatkovne tokove. Predstavljamo si ga lahko kot sekvenco/zaporedje zlogov (bytov), kot npr. datoteka, podatki vneseni preko tipkovnice, podatki, ki smo jih poslali na zaslon. Razred **Stream** torej omogoča splošen oz. enoličen pogled in obdelavo podatkov ne glede na njihov različen izvor, tako da se programerjem ni potrebno ukvarjati s posebnostmi operacijskega sistema in pripadajočo opremo.

V binarne datoteke lahko shranimo prav vse vgrajene numerične podatkovne tipe, zaradi česar so binarne datoteke bolj primerne za aplikacije, ki operirajo z numeričnimi podatki. V nasprotju, pa so vsi numerični podatki v tekstovnih datotekah shranjeni kot zaporedje znakov, zaradi česar jih moramo, če jih hočemo uporabiti v aritmetičnih operacijah, spremeniti v numerične podatke.

Ko shranimo nek tekst v tekstovno datoteko, lahko za to datoteko uporabimo poljubno končnico (ekstenzijo). Bolj naravno pa je (tako bomo delali tudi v nadaljevanju), da za tekstovne datoteke uporabimo končnico ***txt***, za binarne datoteke pa končnico ***dat***. Tako nam že same končnice datotek povedo, ali gre za tekstovne ali pa za binarne datoteke.

Uporaba razreda FileStream

Za kreiranje podatkovnega toka, ki nas poveže z neko datoteko na disku, uporabimo razred **FileStream**.

Sintaksa za kreiranje novega objekta razreda **FileStream**:

```
FileStream fs = new FileStream(pot, mode [, access [, share]])
```

V prvem parametru povemo, kako se datoteka imenuje, lahko pa zapišemo tudi pot do te datoteke (imenike in podimenike), z drugim parametrom pa povemo, kako bomo to datoteko odprli (ali bomo kreirali novo datoteko, ali bomo datoteko le odprli ali pa bomo podatke dodajali k obstoječim v datoteki, ipd.). Prva dva parametra (pot in način kreiranja – **FileMode**) sta obvezna, druga dva pa le opcijska. Če tretji parameter (**access**) ni naveden, lahko podate v datoteko tako zapisujemo, kot tudi beremo iz nje. Za kodiranje (zapis) argumentov `mode`, `access` in `share` uporabljamo zaporedoma naštevne tipe **FileMode**, **FileAccess** in **FileShare**.

Če želimo npr. kreirati datoteko, ki še ne obstaja, bomo uporabili način **FileMode.Create** in tako kreirali novo datoteko. Če pa datoteka z imenom, ki smo jo navedli v prvem parametru (pot) že obstaja, bo njena vsebina prepisana z novo vsebino. Če pa seveda nočemo prepisati vsebine že obstoječe datoteke, bomo raje uporabili način **FileMode.CreateNew**. Naslednja tabela prikazuje vse možne načine odpiranja datoteke:

Tabela načinov kreiranja datoteke – FileMode

Način kreiranja	Razlaga
Append	Odpiranje datoteke, če le-ta obstaja in obenem se postavimo na njen konec. Če datoteka ne obstaja, se skreira nova. **Ta način kreiranja datoteke lahko uporabimo le kadar želimo v datoteko pisati**, ne pa tudi kadar želimo iz nje samo brati podatke.
Create	Kreiranje nove datoteke. **Če datoteka že obstaja, bo njena vsebina prepisana**.
CreateNew	Kreiranje nove datoteke. Če datoteka že obstaja, pride do izjeme (napake - **exception**).
Open	Odpiranje že obstoječe datoteke. Če datoteka še ne obstaja, pride do izjeme (**exception**).

OpenOrCreate	Odpiranje datoteke, če le ta obstaja, oziroma kreiranje nove datoteke, če le-ta še ne obstaja.
Truncate	Odpiranje obstoječe datoteke in jo skrajšati (izprazniti), tako da je njena dolžina nič bytov.

Z drugim parametrom **access** povemo, ali bomo podatke iz datoteke brali, jih vanjo zapisovali ali pa oboje. Ta parameter lahko izpustimo, a v tem primeru bo vzeta privzeta vrednost – v datoteko bomo lahko podatke zapisovali in jih hkrati brali iz nje. Naslednja tabela opisuje vse tri možne načine manipulacije z neko datoteko:

Tabela načinov manipulacije s podatki – FileAccess

Način manipulacije	Razlaga
Read	Podatke iz datoteke lahko le beremo, ne pa tudi zapisujemo.
ReadWrite	Podatke iz datoteke lahko beremo in tudi zapisujemo vanjo. **Ta način je privzet**.
Write	Podatke lahko v datoteko le zapisujemo, ne pa tudi beremo.

S tretjim **share** argumentom povemo, ali bodo imeli dostop do te datoteke tudi drugi uporabniki in kakšne pravice za dostop bodo imeli. V naslednji tabeli so prikazani vsi možni načini:

Tabela načinov porazdelitev (dostopa) podatkov z drugimi aplikacijami – FileShare

Način porazdelitve	Razlaga
None	Datoteko ne more odpreti nobena druga aplikacija.
Read	Omogoča, da datoteko lahko odprejo tudi druge aplikacije, a le za branje.
ReadWrite	Omogoča, da datoteko lahko odprejo tudi druge aplikacije in to za branje in pisanje.
Write	Omogoča, da datoteko lahko odprejo tudi druge aplikacije, a le za branje.

Primer:

Kreirajmo nov objekt izpeljan iz razreda **FileStream** in ga poimenujmo **fs**. Objektu smo s tretjim parametrom priredili le možnost **pisanja** v datoteko.

```
string pot = @"C:\Datoteke\Izdelki.txt";
FileStream fs = new FileStream(pot, FileMode.Create, FileAccess.Write);
```

V primeru smo uporabili metodo **FileMode.Create** za kreiranje nove datoteke (oz. za prepis vsebine že obstoječe datoteke). Če pa smo v prvem parametru **pot** uporabili pot, ki ne obstaja (npr. navedli smo neobstoječi imenik), bo prišlo do izjeme (napake) tipa **DirectoryNotFoundException** (več o izjemah je razloženo v poglavju **Varovalni bloki – obravnava izjem**).

Kreirajmo nov objekt izpeljan iz razreda **FileStream** in ga poimenujmo **fs**. Objektu smo s tretjim parametrom priredili le možnost **branja** v datoteko. Tudi v tem primeru, tako kot v prejšnjem, smo uporabili metodo **FileMode.Create** za kreiranje nove datoteke (oz. za prepis vsebine že obstoječe datoteke). Velja tako kot za prvi primer: če smo v prvem parametru **pot** uporabili pot, ki ne obstaja (npr. navedli neobstoječi imenik), bo prišlo do izjeme (napake) tipa **DirectoryNotFoundException**.

```
string pot = @"C: \C# 2005\Datoteke\Izdelki.txt";
FileStream fs = new FileStream(pot, FileMode.Create, FileAccess.Read);
```

Najpogostejše metode razreda FileStream

Metoda	Razlaga
Close( )	Zapiranje podatkovnega toka in sproščanje pomnilnika za vse vire vezane na ta tok.
Seek( )	Nastavitev trenutnega položaja potkavnega toka na določeno vrednost
Flush( )	Izpraznitev podatkovnega toka in dokončen zapis vseh podatkov iz toka npr. na disk

Delo s tekstovnimi datotekami

Za branje in pisanje podatkov v tekstovne datoteke uporabljamo razreda **StreamReader** in **StreamWriter**.

Pisanje podatkov v tekstovno datoteko

Za pisanje podatkov v tekstovno datoteko uporabljamo metodi **Write** in **WriteLine** ki pripadata razredu **StreamWriter**. Pri uporabi metode **WriteLine** je v datoteko avtomatsko dodan še znak za konec tekoče vrstice. V kolikor v datoteko shranjujemo posamezne podatke (in ne cele stavke), lahko le-te med seboj ločimo z ustreznim ločilom (npr znakom **|**).

Da lahko pričnemo s pisanjem podatkov v tekstovno datoteko, moramo najprej ustvariti nov objekt tipa **StreamWriter**. To storimo s stavkom:

```
StreamWriter textOut=new StreamWriter(podatkovni_tok);
```

Pri tem je **`podatkovni_tok`** objekt (spremenljivka) tipa **FileStream**, ki smo ga ustvarili že prej (npr. s stavkom `FileStream podatkovni_tok = new FileStream(@"C:\APJ\Vaja.txt", FileMode.Create);`

Obstaja pa seveda tudi krajši način vzpostavljanja podatkovnega toka za pisanje v neko tekstovno datoteko.

```
string datoteka = @"D:\APJ\Vaja.txt";  //ime datoteke, ki jo želimo ustvariti in vanjo pisati
StreamWriter textOut = new StreamWriter(datoteka); /*ker smo uporabili PRIVZETI način za
odpiranje nove datoteke, bo stara vsebina datoteke prepisana z novo vsebino, ne glede na
prejšnjo vsebino datoteke!!!!*/
```

Nastavitve so v tem primeru torej **privzete** – ustvarila se bo **nova** datoteka (ne glede na to ali datoteka s tem imenom že obstaja), v to datoteko bomo pisali, **če pa datoteka s tem imenom že obstaja, bo njena vsebina prepisana z novo vsebino brez opozorila**!)

Metode razreda StreamWriter	Razlaga
Write(podatki)	Zapiše podatke v izhodni tok.
WriteLine(podatki)	Zapiše podatke v izhodni tok in na koncu doda še znak za konec vrstice (običajno znaka **\r\n – carriage return** in **line feed**).
Close( )	Zapre objekt tipa **StremWriter** in pripadajoči objekt **FileStream**.

Kot primer uporabe napišimo kodo, ki v tekstovno datoteko zapiše eno samo vrstico s tremi podatki, ki so med seboj ločeni z ločilom **|**.

```
//Najprej deklariramo string, ki označuje pot do datoteke in njeno ime.
//Pot (imenik in podimenik) MORA že obstajati, sicer pride do napake
string pot = @"C:\Tekstovna\Izdelki.txt";
```

```
//dinamično kreiramo podatkovni tok: napovemo pot in ime datoteke, način kako jo bomo kreirali
//(FileMode.create) in kakšna bo manipulacija s podatki (FileAccess.write)
FileStream fs=new FileStream(pot,FileMode.Create,FileAccess.Write);

//dinamično kreiramo nov objekt tipa StreamWriter za zapisovanje v podatkovni tok - datoteko
StreamWriter textOut=new StreamWriter(fs);

textOut.Write("Kolo"+"|"); //zapis prvega podatka v datoteko, za njim pa še ločilnega znaka |
//lahko bi zapisali tudi textOut.Write("Scott|");
textOut.Write("Scott"+"|");

int komadov = 20;
textOut.Write(komadov + "|"); //enakovredno kot textOut.Write(komadov.Tostring() + "|");

//podatek 220000 je sicer numeričen, a zaradi avtomat. konverzije v string ne pride do napake
textOut.WriteLine(220000);
//enakovreden zapis zadnjega stavka bi bil tudi textOut.WriteLine("220000");

textOut.Close();  //zapiranje podatkovnega toka in s tem datoteke
fs.Close(); //zapremo podatkovni tok fs, da bo datoteka na voljo drugim uporabnikom
```

V zgornjem primeru smo v tekstovno datoteko zapisali tudi numerična podatka. Pri vpisovanju pride do avtomatske konverzije numeričnega podatka v **string**, zaradi česar dodatna uporaba metode **ToString()** ni potrebna. Za konverzijo poskrbi kar sama metoda **Write** oz. **WriteLine**.

Zapomnimo si: pri delu s tekstovno datoteko moramo najprej ustvariti ustrezen **podatkovni tok** (kjer navedemo ime in pot do datoteke, način kreiranja datoteke – **FileMode** in pa način dostopa do datoteeke – **FileAccess**), nato pa moramo kreirati še ustrezen **objekt za zapisovanje podatkov v podatkovni tok** (za zapisovanje v datoteko je to objekt tipa **StreamWriter** , za branje podatkov iz datoteke pa objekt tipa **StreamReader**).

Vaja:

```
/*Na disku C kreiraj mapo Tekstovna. V tej podmapi skreiraj tekstovno datoteko Naslov.txt in
vanjo zapiši svoje osebne podatke*/

string dir = @"C: \Tekstovna";

if (!Directory.Exists(dir))  //če imenik še ne obstaja, ga skreiramo
    Directory.CreateDirectory(dir);
else Console.WriteLine("Mapa že obstaja! Vsebina datoteke bo prepisana!");

string datoteke = dir+@"\Izdelki.txt";

/*ustvarimo podatkovni tok za povezavo z datoteko na disku. Uporabimo opcijo Create (Če
datoteka že obstaja, bo njena vsebina prepisana z novo. Tretji parameter nastavimo na
FileAccess.Write - v datoteko lahko podatke zapisujemo.*/
FileStream fs = new FileStream(datoteke, FileMode.Create, FileAccess.Write);

//kreiramo nov objekt tipa StreamWriter za zapisovanje v podatkovni tok
StreamWriter textOut = new StreamWriter(fs);
//Podatkovni tok je pripravljen za pisanje v datoteko
textOut.WriteLine("France Prešeren"); //zapis prvega stavka v datoteko
textOut.WriteLine("Triglavska 123"); //zapis drugega stavka v datoteko
textOut.WriteLine("Vrba na Gorenjskem");//zapis tretjega stavka v datoteko
textOut.Close();  //zapiranje podatkovnega toka in s tem datoteke
fs.Close(); //zapremo podatkovni tok fs, da bo datoteka na voljo drugim uporabnikom
```

Vaja:

```
/*tekstovno datoteko "c:\Tekstovna\Nakljucna.txt" zapiši 500 naključnih celih števil med 0 in
1000. V vsaki vrstici naj bo natanko 20 števil!*/

string dir = @"C: \Tekstovna";  //imenik in pot

//preverimo obstoj imenika
if (!Directory.Exists(dir))  //če imenik še ne obstaja, ga skreiramo
Directory.CreateDirectory(dir);
string datoteke = dir + @"\Nakljucna.txt";
```

```
/*ustvarimo podatkovni tok za povezavo z datoteko na disku. Uporabimo opcijo Create (Če
datoteka že obstaja, bo njena vsebina prepisana z novo. Tretji parameter nastavimo na
FileAccess.Write - v datoteko lahko podatke zapisujemo.*/
FileStream fs = new FileStream(datoteke, FileMode.Create, FileAccess.Write);

//kreiramo nov objekt tipa StreamWriter za zapisovanje v podatkovni tok
StreamWriter textOut = new StreamWriter(fs);
//Podatkovni tok je pripravljen za pisanje v datoteko
Random naklj = new Random();
for (int i = 0; i < 500; i++)
{
   //Zapis formatiramo tako, da za vsako število predvidimo natanko 5 mest, desna poravnava
   textOut.Write("{0,5}",naklj.Next(1001));
   //textOut.Write("{0,-5}", naklj.Next(1001));//TAKOLE pa bi izgledala LEVA poravnava števila
   if ((i + 1) % 20 == 0)  //v vsaki vrstici naj bo le po 20 števil
      textOut.WriteLine(); //skok v novo vrstico
}
textOut.Close();  //zapiranje podatkovnega toka in s tem datoteke
fs.Close(); //zapremo podatkovni tok fs, da bo datoteka na voljo drugim uporabnikom
```

Vaja:

```
/*v tekstovno datoteko, katere ime določi uporabnik (nahaja pa se v mapi "c:\Tekstovna" zapiši
poštevanko števila, ki ga prebereš preko tipkovnice!*/

string dir = @"C: \Tekstovna";  //imenik in pot

if (!Directory.Exists(dir))   //če imenik še ne obstaja, ga skreiramo
    Directory.CreateDirectory(dir);

Console.Write("Ime datoteke (brez končnice): ");
string ime = Console.ReadLine();                    //Preberemo ime datoteke
ime = dir + @"\" + ime + ".txt";
Console.Write("Število za poštevanko: ");
int stevilo = Convert.ToInt32(Console.ReadLine()); //Preberemo ime število za poštevanko
/*ustvarimo podatkovni tok za povezavo z datoteko na disku. Uporabimo opcijo Create (Če
datoteka že obstaja, bo njena vsebina prepisana z novo. Tretji parameter nastavimo na
FileAccess.Write - v datoteko lahko podatke zapisujemo.*/

FileStream fs = new FileStream(ime, FileMode.Create, FileAccess.Write);

//kreiramo nov objekt tipa StreamWriter za zapisovanje v podatkovni tok
StreamWriter textOut = new StreamWriter(fs);

for (int i = 1; i < 11; i++)
{
    //Zapis v datoteko formatiramo
    textOut.WriteLine("{0,3} * {1,3} = {2,5}", i,stevilo,stevilo*i);
}
textOut.Close();  //zapiranje podatkovnega toka in s tem datoteke
fs.Close(); //zapremo podatkovni tok fs, da bo datoteka na voljo drugim uporabnikom
```

Vaja:

```
/*Kreiraj dvodimenzionalno tabelo 50 x 20 naključnih celih števil med vključno -100 in +100.
Tabelo nato zapiši v tekstovno datoteko Tabela2D.txt*/

Random naklj = new Random();
//zgenerirajmo tabelo in jo napolnimo z naključnimi števili
int [,]tabela2D=new int[50,20];
for (int i = 0; i < 50; i++)
  for (int j = 0; j < 20; j++)
     tabela2D[i,j] = naklj.Next(-100, 101);

//tabelo sedaj prepišimo v tekstovno datoteko c:\Tekstovna\Tabela2D.txt
string dir = @"C: \Tekstovna";  //imenik in pot

//preverimo obstoj imenika
if (!Directory.Exists(dir))  //če imenik še ne obstaja, ga skreiramo
  Directory.CreateDirectory(dir);

string datoteke = dir + @"\Tabela2D.txt"; //datoteka, v katero bomo prepisali našo 2D tabelo
```

```
/*ustvarimo podatkovni tok za povezavo z datoteko na disku. Uporabimo opcijo Create (Če
datoteka že obstaja, bo njena vsebina prepisana z novo. Tretji parameter nastavimo na
FileAccess.Write - v datoteko lahko podatke zapisujemo.*/
FileStream fs = new FileStream(datoteke, FileMode.Create, FileAccess.Write);

//kreiramo nov objekt tipa StreamWriter za zapisovanje v podatkovni tok
StreamWriter textOut = new StreamWriter(fs);
//vsebino tabele prepišemo v datoteko
for (int i = 0; i < 50; i++)
{
   for (int j = 0; j < 20; j++)
   {
      //Zapis v datoteko formatiramo na 6 mest
      textOut.Write("{0,6}",tabela2D[i, j]);
   }
   textOut.WriteLine(); //skok v novo vrstico
}
textOut.Close();  //zapiranje podatkovnega toka in s tem datoteke
fs.Close(); //zapremo podatkovni tok fs, da bo datoteka na voljo drugim uporabnikom
```

Naloge:

- V tekstovno datoteko NASLOV.TXT zapiši svoje osebne podatke (1. vrstica: ime in priimek, 2. vrstica: naslov, 3. vrstica naj bo prazna, 4.vrstica: pošta in kraj)!
- Napiši program, ki bo v zanki bral podatke o določenem kraju in njegovi poštni številki. Podatke zapisuj v tekstovno datoteko, za vsak kraj v svojo vrstico. Zanka (vnos podatkov) se zaključi, ko uporabnik vnese prazen kraj!
- Kreiraj tekstovno datoteko OCENE.TXT in vanjo zapiši štiri vrstice: v vsaki vrstici naj bo naziv predmeta in tvoja ocena pri tem predmetu. Vpis naj bo formatiran (za naziv predmeta natanko 40 znakov – leva poravnava, za oceno pa natanko 2 znaka - desna poravnava!)
- Napiši program, ki bere stavke vnesene preko tipkovnice in stavke zapisuje v tekstovno datoteko v obratnem vrstnem redu.
- V tekstovno datoteko Stevila.txt zapiši prvih 100 sodih števil, ki niso deljiva s 6! Med števila zapiši poljuben ločilni znak!
- Napiši program za pisanje/dodajanje podatkov v tekstovno datoteko DRZAVE.TXT. Podatke vpisuj/dodajaj s močjo funkcije, ki naj na začetku preveri, ali datoteka sploh obstaja – če datoteka še ne obstaja naj jo funkcija najprej ustvari. Vsaka vrstica v datoteki naj bo sestavljena iz imena države (40 znakov) in števila prebivalcev v njej (celo število, formatirano na 15 mest)
- Kreiraj tekstovno datoteko KOCKA.TXT, v katero želiš shraniti rezultate 1000 metov kocke. V vsako vrstico te datoteke nato zapiši zaporedno številko vrstice in naključno število med 1 in 6 (met kocke) – vpis meta kocke formatiraj na 3 mesta. Izgled datoteke:

  1._ _ 4
  2._ _ 6
  3._ _ 1
  …

Branje podatkov iz tekstovne datoteke

Za branje podatkov iz tekstovne datoteke je na voljo več metod razreda **StreamReader**, najpomembnejši pa sta metodi **Read** in **ReadLine**.

Da lahko pričnemo z branjem podatkov iz tekstovne datoteke, moramo najprej ustvariti nov objekt tipa **StreamReader**. To storimo npr. takole:

```
string datoteka = @"C: \Tekstovna\Padavine.txt";  //ime in pot do datoteke
//najprej ustvarimo objekt za povezavo z datoteko na disku. Uporabimo opcijo Open.
FileStream podatkovni_tok = new FileStream(datoteka, FileMode.Open);
//kreiramo nov objekt tipa StreamReader za branje v podatkovni tok
StreamReader textIn = new StreamReader(podatkovni_tok);
```

Obstaja pa tudi krajši način za odpiranje datoteke za branje, brez predhodnega kreiranja objekta tipa **FileStream**. V tem primeru so nastavitve ob odpiranju datoteke privzete (datoteko odpremo za branje, vsebina datoteke pa bo drugim uporabnikom nedostopna vse dokler je ne bomo zaprli).

```
string datoteka = @"C: \Tekstovna\Padavine.txt";  //ime in pot do datoteke
StreamReader textIn = new StreamReader(datoteka); //privzeto odpiranje datoteke za branje
```

Metode razreda StreamReader	Razlaga
Peek( )	Vrne naslednji razpoložljivi znak v vhodni tok, brez premika na naslednjo pozicijo (naslednji znak). Če ni na voljo nobenega znaka več, metoda vrne vrednost -1.
EndOfStream()	Metoda vrne **true**, če smo že na koncu podatkovnega toka, sicer pa vrne vrednost **false**.
Read( )	Bere naslednji razpoložljivi znak z vhodnega toka. **POZOR**: metoda vrne **CELO ŠTEVILO**, ki predstavlja **ASCII** kodo tega znaka.
ReadLine( )	Bere naslednjo vrstico podatkov z vhodnega toka in jo vrne kot **string**.
ReadToEnd( )	Bere podatke s trenutne pozicije v vhodnem toku, vse do konca toka in podatke vrne kot **string**. Navadno se ta metoda uporablja za branje celotne vsebine datoteke.
Close( )	Zapre objekt tipa **StremReader** in pripadajoči objekt **FileStream**.
Lastnost	**Razlaga**
EndOfStream	Pridobivanje vrednosti ki nam pove, ali smo že na koncu podatkovnega toka. V tem primeru je lastnost enaka **true**, sicer pa **false**.

Primer:

```
/*Dana je tekstovna datoteka c:\Tekstovna\Padavine.txt. Izpišimo jo na zaslon. Uporabili bomo
vse tri načine:
1) Branje vsakega znaka posebej
2) Branje vrstice za vrstico
3) Enkratno branje celotne datoteke*/

string datoteka = @"C: \Tekstovna\Padavine.txt";  //ime in pot do datoteke

if (File.Exists(datoteka))  //preverimo obstoj datoteke
{
   /*ustvarimo podatkovni tok za povezavo z datoteko na disku. Uporabimo opcijo Open. Tretji
     parameter nastavimo na FileAccess.Read = branje datoteke*/
   FileStream fs = new FileStream(datoteka, FileMode.Open, FileAccess.Read);
   //kreiramo nov objekt tipa StreamReader za zapisovanje v podatkovni tok
   StreamReader textIn = new StreamReader(fs);

   //1) Branje vsakega znaka posebej
   while (textIn.Peek() != -1) //iz datoteke beremo posamezne znake dokler jih ne zmanjka
   {
      char znak = (char)textIn.Read(); //preberemo vsak znak posebej
      Console.Write(znak);             //prebrani znak izpišemo na zaslon
   }
   textIn.Close();  //zapiranje podatkovnega toka in s tem datoteke
```

```
   Console.WriteLine("\n-----------------------------------------------------------------------");

   //2) Branje vrstice za vrstico
   fs = new FileStream(datoteka, FileMode.Open, FileAccess.Read);
   textIn = new StreamReader(fs);
   while (textIn.Peek() != -1) //iz datoteke beremo podatke dokler jih ne zmanjka
   /*Lahko bi zapisali tudi:
     while (!textIn.EndOfStream)   //dokler NI konec toka
        {…}

   ali pa:
     string vrstica;
     //dokler metoda ReadLine vrača vrednost različno od null
      while ((vrstica = branje.ReadLine()) != null)
     {…}
   */

   {
      string stavek = textIn.ReadLine(); //preberemo celo vrstico
      Console.WriteLine(stavek);         //prebrano vrstico izpišemo na zaslon
   }
   textIn.Close();  //zapiranje podatkovnega toka in s tem datoteke

   Console.WriteLine("\n-----------------------------------------------------------------------");

   //3) Enkratno branje cele datoteke
   fs = new FileStream(datoteka, FileMode.Open, FileAccess.Read);
   textIn = new StreamReader(fs);
   string vsebinaDatoteke = textIn.ReadToEnd(); //naenkrat preberemo celotno vsebino datoteke
   Console.WriteLine(vsebinaDatoteke);          //prebrano vsebino datoteke izpišemo na zaslon
   textIn.Close();  //zapiranje podatkovnega toka in s tem datoteke
   fs.Close();
}
```

Vaja:

```
/*Dana je tekstovna datoteka c:\Tekstovna\Padavine.txt. Prekopiraj jo v datoteko
Padavine.rez. Nalogo reši na štiri načine:
   1) Datoteko prekopiraš z metodo Copy razreda File
   2) Iz datoteke bereš vsak znak posebej in znake sproti zapisuješ v novo datoteko
   3) Iz datoteke bereš stavke in te stavke sproti zapisuješ v novo datoteko
   4) Naenkrat prebereš celotno vsebino datoteke in to vsebino nato zapišeš v novo datoteko*/

string datoteka = @"C: \Tekstovna\Padavine.txt";  //ime in pot do datoteke

if (File.Exists(datoteka))  //preverimo obstoj datoteke
{
    //1)Metoda Copy
    if (!File.Exists(datoteka))
    File.Copy(datoteka, @"C: \Tekstovna\Padavine.rez");

    //2) Branje vsakega znaka posebej
    FileStream fs = new FileStream(datoteka, FileMode.Open, FileAccess.Read);//tok za branje
    FileStream f = new FileStream(@"C: \Tekstovna\Padavine1.rez", FileMode.Create,
                                  FileAccess.Write);//podatkovni tok za pisanje
    //kreiramo nov objekta tipa StreamReader in StreamWriter
    StreamReader textIn = new StreamReader(fs);//podatke bomo brali v tok
    StreamWriter textOut = new StreamWriter(f);//podatke bom s tokom zapisovali

    while (textIn.Peek() != -1) //iz datoteke beremo podatke dokler jih ne zmanjka
    {
        char znak = (char)textIn.Read(); //preberemo vsak znak posebej
        textOut.Write(znak);             //prebrani znak izpišemo v novo datoteko
    }
    textIn.Close();  //zapiranje podatkovnega toka in s tem datoteke
    textOut.Close(); //zapiranje podatkovnega toka in s tem datoteke

    //3) Branje vrstice za vrstico
    fs = new FileStream(datoteka, FileMode.Open, FileAccess.Read);
    f = new FileStream(@"C: \Tekstovna\Padavine2.rez", FileMode.Create,
                       FileAccess.Write);//podatkovni tok za pisanje
    textIn = new StreamReader(fs);//podatke bomo brali v tok
    textOut = new StreamWriter(f);//podatke bom s tokom zapisovali
```

```
    while (textIn.Peek() != -1) //iz datoteke beremo podatke dokler jih ne zmanjka
    {
       string stavek = textIn.ReadLine(); //preberemo celo vrstico
       textOut.WriteLine(stavek);         //prebrano vrstico izpišemo v novo datoteko
    }
    textIn.Close();  //zapiranje podatkovnega toka in s tem datoteke
    textOut.Close(); //zapiranje podatkovnega toka in s tem datoteke

    //4) Enkratno branje cele datoteke
    fs = new FileStream(datoteka, FileMode.Open, FileAccess.Read);
    f = new FileStream(@"C: \Tekstovna\Padavine3.rez", FileMode.Create,
                       FileAccess.Write);//podatkovni tok za pisanje
    textIn = new StreamReader(fs);//podatke bomo brali v tok
    textOut = new StreamWriter(f);//podatke bom s tokom zapisovali
    string vsebinaDatoteke = textIn.ReadToEnd(); //naenkrat preberemo celotno vsebino datoteke
    textOut.WriteLine(vsebinaDatoteke);          //prebrano vsebino izpišemo v novo datoteko
    textIn.Close();  //zapiranje podatkovnega toka in s tem datoteke
    textOut.Close(); //zapiranje podatkovnega toka in s tem datoteke
    fs.Close();
}
```

Vaja:

```
/*Za poljubno tekstovno datoteko (ime vnese uporabnik) ugotovi
  1) Koliko vrstic vsebuje
  2) Koliko je vseh znakov v datoteki
  3) Koliko samoglasnikov vsebuje
  4) Najdaljši stavek v datoteki*/

string datoteka;
while (true)  //neskončna zanka
{
    Console.Write("Ime datoteke: ");
    datoteka = Console.ReadLine();
    if (File.Exists(datoteka))
      break;  //če datoteka obstaja, gremo iz zanke ven
    else Console.WriteLine("Datoteka s tem imenom ne obstaja!");
}

FileStream fs = new FileStream(datoteka, FileMode.Open, FileAccess.Read);//tok za branje
//kreiramo nov objekt tipa StreamReader in StreamWriter za zapisovanje v podatkovni tok
StreamReader textIn = new StreamReader(fs);//podatke bomo brali v tok

int znakov = 0,vrstic=0,samoglasnikov=0;
string najdaljsi="";
while (textIn.Peek() != -1) //iz datoteke beremo podatke dokler jih ne zmanjka
{
    string stavek = textIn.ReadLine();    //preberemo cel stavek
    if (stavek.Length > najdaljsi.Length) //preverimo, če je ta stavek daljši od doslej
        najdaljsi = stavek;               //najdaljšega
    vrstic++;                            //povečamo število vrstic
    znakov = znakov + stavek.Length;    //povečamo število vseh znakov
    for (int i = 0; i < stavek.Length; i++)
    {
        char znak = char.ToUpper(stavek[i]); //vsak znak spremenimo v veliko črko (če znak ni
                                             //črka, se ne zgodi ničesar)
        if (znak == 'A' || znak == 'E' || znak == 'I' || znak == 'O' || znak == 'U')
           samoglasnikov++;
    }
    //Namesto zgornje rešitve bi lahko brali tudi vsak znak posebej
    //char znak = (char)textIn.Read(); //preberemo vsak znak posebej
    //znaka za konec vrstice sta (char)13 in (char)10
    //znak za konec datoteke pa je textIn.EndOfStream
    //Preverimo, če smo na koncu vrstice oz na koncu datoteke
    //if ((znak==(char)10)||(znak==(char)13) || (textIn.EndOfStream))
}
textIn.Close();  //zapiranje podatkovnega toka in s tem datoteke
Console.WriteLine("Skupno število znakov v datoteki: " + znakov);
Console.WriteLine("Skupno število vrstic v datoteki: " + vrstic);
Console.WriteLine("Skupno število samoglasnikov v datoteki: " + samoglasnikov);
Console.WriteLine("Najdaljši stavek v datoteki: " + najdaljsi);
```

Vaja:

```
/*Za poljubno tekstovno datoteko ugotovi in nato izpiši najdaljšo besedo v tej datoteki*/

Console.Write("ime datoteke: ");
string ime=Console.ReadLine();
if(File.Exists(ime))
{
   FileStream fs = new FileStream(ime, FileMode.Open);
   StreamReader textIn = new StreamReader(fs);
   string najdaljsa = "";
   while (textIn.Peek() != -1) //dokler ni konec datoteke
   {
       string vrstica = textIn.ReadLine();//preberemo celo vrstico
       //z metodo Split posamezne besede iz vrstice shranimo v tabelo
       string[] tabela = vrstica.Split(' ');
       for (int i = 0; i < tabela.Length; i++)
       {
           if (tabela[i].Length > najdaljsa.Length)
           {
               najdaljsa = tabela[i];
           }
       }
   }
    textIn.Close();
    fs.Close();
    Console.WriteLine("najdaljsa beseda: "+najdaljsa);
}
```

Vaja:

```
/*PREPROSTA MENJALNICA!  Program, za vnos novega tečaja, izpis tečajne liste in menjavo
denarja. Podatki so v datoteki Tecaji.txt - to je datoteka deviznih tečajev (v vsaki vrstici
je oznaka države, oznaka valute in trenutni prodajni tečaj (realno število).*/
static void vnos(string imeDat)
{
   FileStream fs;
   if (!File.Exists(imeDat))//če datoteka še ne obstaja bomo naredili novo
      fs = new FileStream(imeDat, FileMode.Create, FileAccess.Write);
   else  //če datoteka že obstaja, bomo podatke dodajali
      fs = new FileStream(imeDat, FileMode.Append, FileAccess.Write);

   StreamWriter textOut=new StreamWriter(fs);
   //vnos podatkov o novi valuti
   Console.Write("Država: ");
   string drzava = Console.ReadLine();
   Console.Write("Oznaka valute: ");
   string valuta = Console.ReadLine();
   Console.Write("Trenutni tečaj: ");
   double tecaj = Convert.ToDouble(Console.ReadLine());
   textOut.WriteLine(drzava+"|"+valuta + "|" + tecaj);//zapis v datoteko, med podatki je
                                                      //ločilni znak "|"
   textOut.Close();
   fs.Close();
}

static void izpis(string imeDat)
{
   FileStream fs;
   if (!File.Exists(imeDat))//če datoteka še ne obstaja bomo naredili novo
      Console.WriteLine("Datoteka še ne obstaja!");
   else  //če datoteka že obstaja, bomo podatke dodajali
   {
      fs = new FileStream(imeDat, FileMode.Open, FileAccess.Read);
      StreamReader textIn = new StreamReader(fs);
      Console.WriteLine("Država              Oznaka valute    Tečaj");
      Console.WriteLine("-----------------------------------------");
      while (textIn.Peek() != -1) //podatke beremo dokler ne pridemo do konca datoteke
      {
         string vrstica=textIn.ReadLine();
         string [] tabela=vrstica.Split('|');  //ker so podatki v prebrani vrstici razmejeni z
                                               //znakom "|", jih lahko ločimo z metodo split
         Console.WriteLine("{0,-20}{1,-17}{2,-10:5}",tabela[0],tabela[1],tabela[2]);
      }
```

```
      textIn.Close();
      fs.Close();
   }
   Console.ReadLine();
}

static void menjava(string imeDat)
{
   FileStream fs;
   if (!File.Exists(imeDat))//če datoteka še ne obstaja bomo naredili novo
      Console.WriteLine("Datoteka še ne obstaja!");
   else  //če datoteka že obstaja, bomo podatke dodajali
   {
      fs = new FileStream(imeDat, FileMode.Open, FileAccess.Read);
      StreamReader textIn = new StreamReader(fs);
      //preberem celo datoteko, da vem, katere valute so že v njej
      int stevec=1;
      Console.WriteLine();
      //Najprej preberemo vse vrstice, da ugotovimo, koliko podatkov (in katere valute) je že
      //v datoteki
      while (textIn.Peek() != -1)
      {
         string vrstica = textIn.ReadLine();
         string[] tabela = vrstica.Split('|');
         Console.Write(stevec+") "+tabela[1]+"  "); //oznake valut, ki so že v tabeli v obliki
                                                    //menija izpišemo na zaslon
         stevec++;
      }
      textIn.Close();
      fs = new FileStream(imeDat, FileMode.Open, FileAccess.Read);
      textIn = new StreamReader(fs);
      //uporabnikova izbira valute se ujema z zaporedno številko valute (=vrstice) v datoteki
      Console.Write("\nIzberi ustrezno valuto: ");
      int izb = Convert.ToInt32(Console.ReadLine());
      if (izb > 0 || izb < stevec)
      {
         Console.Write("Znesek za menjavo: ");
         double znesek = Convert.ToDouble(Console.ReadLine());
         int stvrstice=1;
         //poiščemo zaporedno številko vrstice v datoteki, ki se ujema z izbarno valuto
         while (true)
         {
            string vrstica = textIn.ReadLine();
            string[] tabela = vrstica.Split('|');
            if (stvrstice==izb)
            {
               Console.WriteLine();
               Console.WriteLine("Oznaka valute: "+tabela[1]);
               Console.WriteLine("Prodajni tečaj: "+tabela[2]);
               Console.WriteLine("Znesek za menjavo: "+znesek);
               Console.WriteLine("Znesek v valuti: "+znesek*Convert.ToDouble(tabela[2])
                                 +"  "+tabela[1]);
               break;
            }
            else
               stvrstice++;
         }
      }
   }
   Console.ReadLine();
}

//glavni program
static void Main(string[] args)
{
   char izbira;
   do
   {
      Console.Clear();
      Console.WriteLine("V-Vnos oz. dodajanje,I-Izpis,M-Menjava,K-Konec");
      Console.Write("Vnesi izbiro: ");
      izbira=char.ToUpper(Convert.ToChar(Console.ReadLine()));
      switch (izbira)
      {
         case 'V':{
```

```
                    vnos(@"c:\Tekstovna\Tecaji.txt");
                    break;
                }
            case 'I':{
                    izpis(@"c:\Tekstovna\Tecaji.txt");
                    break;
                }
            case 'M':{
                    menjava(@"c:\Tekstovna\Tecaji.txt");
                    break;
                }
        }
    }
    while (char.ToUpper(izbira) != 'K');
}
```

Vaja:

```
/*Dana je tekstovna datoteka PADAVINE.TXT.  V njej je neznano število stavkov s podatki o
količini padavin v določenem kraju. Vsaka vrstica je sestavljena iz imena kraja in količine
letnih padavin. Med imenoom kraja in količino padavin je ločilni znak |.
Napiši funkcijo za izpis vsebine datoteke na zaslon.
Napiši funkcijo, ki dobi za parameter to datoteko in ki ugotovi ter izpiše skupno količino
vseh padavin!
Napiši še funkcijo, ki vrne naziv kraja z največ padavinami*/

static void Main(string[] args)
{
   string datoteka=@"c:\Tekstovna\Padavine.txt";
   if (!File.Exists(datoteka))
      Console.WriteLine("Datoteka ne obstaja!");
   else
   {
      izpis(datoteka);
      skupajPadavin(datoteka);
      Console.WriteLine("Kraj z največ padavinami: "+najvecPadavin(datoteka));
   }
}

static void izpis(string datoteka)
{
   //vzpostavimo povezavo z datoteko na disku
   FileStream fs = new FileStream(datoteka, FileMode.Open, FileAccess.Read);
   StreamReader textIn = new StreamReader(fs);//podatkovni tok za branje

   Console.WriteLine("Vsebina datoteke Padavine.txt");
   string vrstica;
   while (textIn.Peek() != -1)
   {
      vrstica = textIn.ReadLine(); //preberemo stavek iz datoteke
      Console.WriteLine(vrstica);//vrstico izpišemo na zaslon
   }
   textIn.Close();
   fs.Close();
}

static void skupajPadavin(string datoteka)
{
   FileStream fs = new FileStream(datoteka, FileMode.Open, FileAccess.Read);
   StreamReader textIn = new StreamReader(fs);//podatkovni tok za branje

   string vrstica;
   double vsota=0;
   while (textIn.Peek() != -1)
   {
      vrstica = textIn.ReadLine(); //preberemo stavek iz datoteke
      string []tabela = vrstica.Split('|'); //stavek razbijemo na posamezne dele, glede na
                                            //ločilni znak |
      vsota = vsota + Convert.ToDouble(tabela[1]);
   }
   Console.WriteLine("Skupna količina padavin: "+vsota);
}

static string najvecPadavin(string datoteka)
```

```
{
   FileStream fs = new FileStream(datoteka, FileMode.Open, FileAccess.Read);
   StreamReader textIn = new StreamReader(fs);//podatkovni tok za branje
   string vrstica, najKraj="";
   double najPad = 0;
   while (textIn.Peek() != -1)
   {
      vrstica = textIn.ReadLine(); //preberemo stavek iz datoteke
      string[] tabela = vrstica.Split('|');
      if (Convert.ToDouble(tabela[1]) > najPad)
      {
         najPad = Convert.ToDouble(tabela[1]);
         najKraj=tabela[0];
      }
   }
   return najKraj;
}
```

Vaja:

V naslednji vaji bomo prebrali podatke iz prej kreiranje tekstovne datoteke **Izdelki.txt**. Podatke bomo shranili v tabelo izdelkov. Vsak izdelek je objekt razreda Izdelek, ki ga moramo seveda najprej deklarirati. Deklariramo ga seveda izven vseh metod, a znotraj imenskega prostora, ali pa znotraj razreda, ki pripada obrazcu, ki ga trenutno obdelujemo.

```
public class Izdelek
{
   public string naziv;
   public string proizvajalec;
   public int komadov;
   public decimal cena;

   public Izdelek() //konstruktor
   { }

}
```

Še koda za branje podatkov iz datoteke in zapis v tabelo objektov tipa Izdelek:

```
string pot = @"C: \Tekstovna\Izdelki.txt";
FileStream fs=new FileStream(pot,FileMode.OpenOrCreate,FileAccess.Read);
StreamReader textIn = new StreamReader(fs);

//enodimenzionalna tabela objektov tipa izdelek
Izdelek [] tabelaizdelkov=new Izdelek[10];

int indeks = 0;//zaporedna štev. izdelka in hkrati zaporedna vrstica v tabeli tabelaizdelkov

while (textIn.Peek() != -1) //iz datoteke beremo podatke dokler jih ne zmanjka
{
    string vrstica = textIn.ReadLine();//preberemo celo vrstico

    //v tabelo stolpci zaporedoma zložimo zaporedja znakov med ločili |
    string[] stolpci = vrstica.Split('|');

    Izdelek Izd = new Izdelek();//nov objekt tipa Izdelek

    //objektu izd, ki je izplejan iz razreda Izdelek priredimo vrednosti, ki smo jih izluščili
    //iz prebrane vrstice. To nam je uspelo zato, ker so bili podatki ločeni z ločilom |
    Izd.naziv = stolpci[0];
    Izd.proizvajalec = stolpci[1];
    Izd.komadov = Convert.ToInt32(stolpci[2]);
    Izd.cena = Convert.ToDecimal(stolpci[3]);

    tabelaizdelkov[indeks] = Izd;//objekt izd zapišemo v tabelo izdelkov z ustreznim indeksom
    indeks++;  //povečamo indeks
}
```

Vaja:

V naslednji vaji je prikazana uporaba metode **Seek** razreda **Filestrem**, ki omogoča, da se premikamo po podatkovnem toku. Metoda ima dva parametra. S prvim parametrom povemo, kolikšen nja bo relativni odmik (**offset**) od pozicije, ki jo določimo z drugim parametrom. Drugi parameter (**origin**) določa, ali želimo odmik (ki je podan s prvim parametrom) izvesti od začetka podatkovnega toka, od konca podatkovnega toka ali pa od trenutne pozicije. Izberemo lahko torej eno izmed treh vrednosti, ki pripadajo naštevnemu tipu SeekOrigin, ki ima tri vrednosti: **SeekOrigin.Begin**, **SeekOrigin.Current** in **SeekOrigin.End**.

```
/*V Tekstovno datoteko zapišimo 10 naključnih celih števil. S pomočjo  metode Seek se nato
postavimo na zečetek toka in preberemo zapisane podatke*/

string datoteka = "Stevila.txt";

FileStream fs = new FileStream(datoteka, FileMode.OpenOrCreate,FileAccess.ReadWrite);
StreamWriter textOut = new StreamWriter(fs);  //Tekstovni podatkovni tok
Random naklj = new Random();  //generator naključnih števil
//v datoteko zapišemo 10 celih števil
for (int i = 0; i < 10; i++)
{
    int stevilo = naklj.Next(0, 101);
    textOut.WriteLine(stevilo); //zapis v tekstovno datoteko
    Console.Write(stevilo+", ");
}

//Poskrbimo za fizičen zapis podatkov v toku na disk: podatki se namreč
//dokončno zapišejo na disk šele ko zapremo podatkovni tok, ali pa ko
//uporabimo metodo Flush()
textOut.Flush();  //Fizičen zapis podatkov v toku na disk!!!
StreamReader textIn = new StreamReader(fs);
//z metodo Seek se postavimo na začetek toka fs - SeekOrigin.Begin (TA JE
//OSTAL ODPRT), offset(odmik) od začetka pa je enak 0.
fs.Seek(0, SeekOrigin.Begin);
Console.WriteLine("\nVsebina datoteke: \n");
//while (!textIn.EndOfStream) - while zanke NE moremo uporabiti, ker
//dejansko šele metoda close "zapiše" KONEC datoteke
for (int i = 0; i < 10; i++)
{
    int st = Convert.ToInt32(textIn.ReadLine());
    Console.Write(st + ", ");
}
textOut.Close(); //zapremo podatkovne tokove
textIn.Close();
fs.Close();
```

Naloge:

- Napiši funkcijo, ki dobi za parameter ime poljubne tekstovne datoteke in ki njeno vsebino prikaže na zaslonu!
- Dana je tekstovna datoteka DIJAKI.txt. V vsaki vrstici te datoteke je prvih 30 znakov rezervirano za ime dijaka, naslednjih 15 pa za učni uspeh (od 1 do 5).
  - Koliko dijakov je v datoteki
  - Koliko dijakov ima splošni učni uspeh enak 5
  - Kolikšen je povprečen učni uspeh vseh dijakov
- Kreiraj tekstovno datoteko APJ.TXT. V to datoteko zapiši poljubno število stavkov (bereš jih preko tipkovnice, znak za konec vnosa je prazen stavek!). Vsak stavek naj bo v svoji vrstici, med vrsticami pa naj bo po ena prazna vrstica. Napiši funkcijo, ki dobi za parameter ime te datoteke in ki ugotovi in izpiše, koliko znakov vsebuje celotna datoteka!

- Dana je tekstovna datoteka Naloga.txt. V vsaki vrstici te datoteke so po tri cela števila, med seboj ločena s presledkom. Datoteko obdelaj tako, da za vsako vrstico na ekran izpišeš vsoto vseh treh števil, na koncu pa še skupno vsoto vseh števil!

- V tekstovni datoteki temperature.TXT so shranjeni podatki o temperaturi v določenem kraju. V vsaki vrstici je prvih 20 znakov (desna poravnava) rezerviranih za ime kraja, sledi pa podatek o temperaturi (realno število). Ugotovi povprečno temperaturo, ter izpiši ime kraja z največjo temperaturo!

- V tekstovni datoteki so zapisani podatki o porabi bencina za posamezne tipe vozila. V vsaki vrstici je zapisan tip vozila, nato pa podatek o porabi goriva na 100 km (decimalno število)! Koliko vozil je v datoteki? Ugotovi in izpiši tip vozila z najmanjšo porabo goriva. Podatka otipu vozila in porabi goriva sta razmejena z ločilnim znakom **|**.

- Napišite program, ki izpiše 10 najdaljših besed v poljubni tekstovni datoteki.

- Napiši program, ki v poljubni tekstovni datoteki prešteje vse cifre (znake med 0 in 9) in na koncu izpiše, kolikokrat se vsaka cifra pojavi v tej datoteki. Ime datoteke programu podamo kot parameter ukazne vrstice.

- Napiši program **BrisiKomentarje**, ki z ukazne vrstice sprejme ime datoteke v kateri je zapisan nek izvorni program v **C#**. Program naj v datoteko z enakim imenom, a s končnico **rez** prepiše tiste vrstice iz vhodne datoteke, ki se ne začnejo z znakoma za enovrstični komentar '//'.

- Za poljubno tekstovno datoteko ugotovi in izpiši vse besede iz te datoteke in kolikokrat se posamezna beseda pojavi v tej datoteki. Pri tem ne delaj razlike med malimi in velikimi črkami!

Delo z binarnimi datotekami

Za branje in pisanje podatkov v binarne datoteke uporabljamo razreda **BinaryReader** in **BinaryWriter**.

Pisanje podatkov v binarno datoteko

Za pisanje podatkov v binarno datoteko uporabljamo metodo **Write**, ki pripada razredu **BinaryWriter**. Da lahko pričnemo s pisanjem podatkov v binarno datoteko, moramo najprej ustvariti nov objekt tipa **BinaryWriter**. To storimo npr. takole:

```
string datoteka = @"C: \Tekstovna\Padavine.dat";  //ime in pot do datoteke
//povezava z datoteko na disku
FileStream podatkovni tok = new FileStream(datoteka, FileMode.Create, FileAccess.Write);
//Kreiramo podatkovni tok za pisanje
BinaryWriter binaryOut = new BinaryWriter(podatkovni_tok);
```

Krajši način, takšen kot smo ga spoznali pri tekstovnih datotekah, pri pisanju v binarno datoteko **NE** obstaja!!!

Metode razreda BinaryWriter	Razlaga
Write(podatki)	Zapiše podatke v izhodni tok.
Close()	Zapre objekt tipa **BinaryWriter** in pripadajoči objekt tipa **FileStream**.

Vaja:

```
/*V binarno datoteko Stevila.dat zapiši 10000 naključnih celih števil med -100 in +100 */

string datoteka = @"C: \Binarne\Stevila.dat";
string dir=@"c:\Binarne";
if (!Directory.Exists(dir))  //če imenik še ne obstaja, ga skreiramo
   Directory.CreateDirectory(dir);

//povezava z datoteko na disku
FileStream fs = new FileStream(datoteka, FileMode.Create, FileAccess.Write);
BinaryWriter binaryOut = new BinaryWriter(fs);  //Binarni podatkovni tok
Random naklj=new Random();  //generator naključnih števil

for (int i=0;i<10000;i++)   //zanka za generiranje 10000 naključnih števil
    binaryOut.Write(naklj.Next(-100,101)); //zapis v binarno datoteko

binaryOut.Close(); //zapremo podatkovni tok
fs.Close();
```

V naslednjem primeru bomo uporabili objekt razreda **Izdelek**, ki smo ga deklarirali v prejšnjem primeru. Podatke bomo zapisovali v datoteko z metodo **Write**. Ta pred zapisovanjem najprej preveri tip podatka, ki ga želimo zapisati in nato **TA TIP podatka** zapiše v datoteko takšnega kot je. Če torej metodi **Write** posredujemo podatek, ki je tipa **decimal**, metoda tega podatka ne bo spremenila v **string**, ampak bo v datoteko zapisala decimalni podatek. Ker smo uporabili metodo `FileMode.Create` bodo tekoči podatki prepisali prejšnje, če

le-ti seveda obstajajo. V datoteko bomo vpisovali vsako polje posebej, saj objekt **BinaryWriter** ne omogoča, da bi v podatkovni tok (in s tem tudi v datoteko) zapisali celotno strukturo naenkrat.

```
public class Izdelek
{
   public string naziv;
   public string proizvajalec;
   public int komadov;
   public decimal cena;

   public Izdelek() //konstruktor
   { }
}
```

```
//glavni program
//določimo pot in ime datoteke
string pot = @"C: \Binarne\Datoteke\Izdelki.dat";
//dinamično kreiramo nov podatkovni tok
FileStream fs=new FileStream(pot,FileMode.Create,FileAccess.Write);
//dinamično kreiramo nov objekt tipa BinaryWriter
BinaryWriter binaryOut = new BinaryWriter(fs);

Izdelek Izd = new Izdelek();//nov objekt tipa Izdelek
//določimo vrednosti članom razreda. Seveda bi lahko te podatke vnesel uporabnik, npr. v
//gradnike TextBox nekega obrazca
Izd.naziv = "Kolo";
Izd.proizvajalec = "Scott";
Izd.komadov = 110;
Izd.cena =220000.00m;

//podatke zapišemo v binarno datoteko
binaryOut.Write(Izd.naziv);
binaryOut.Write(Izd.proizvajalec);
binaryOut.Write(Izd.komadov);
binaryOut.Write(Izd.cena);
//zapremo tok podatkov in s tem tudi datoteko
binaryOut.Close();
fs.Close();
```

Naloge:

- V binarno datoteko *PopolnaStevila* zapiši vsa števila med 1 in 1000000, ki imajo to lastnost, da so enaka vsoti svojih deliteljev! Koliko je takih števil?
- Napiši program, ki bo računal vrednost eksponentne funkcije e^x. Program naj se izvaja tako dolgo, dokler se v enem koraku vrednost spremeni za več kot 0.00001. Vrednosti funkcije na vsakem koraku shranjuj v binarno datoteko. Za izračun uporabi matematično vrsto (pravilo):

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

  Pri tem je **3!** enako produktu *3 * 2 * 1*, **n!** pa je enako produktu *n * (n-1) * (n – 2) * ... * 3 * 2 * 1*
- V binarno datoteko *LepaStevila* zapiši vsa števila med 1 in 1000000, ki imajo to lastnost, da so enaka vsoti svojih deliteljev! Koliko je takih števil?

Branje podatkov iz binarne datoteke

Za branje podatkov iz binarne datoteke uporabljamo metode, ki pripadajo razredu **BinaryReader**. Da lahko pričnemo z branjem podatkov iz binarne datoteke, moramo najprej ustvariti nov objekt tipa **BinaryReader**. To storimo npr. takole:

```
string datoteka = @"C: \Tekstovna\Padavine.dat";  //ime in pot do datoteke
//povezava z datoteko na disku
FileStream podatkovni_tok = new FileStream(datoteka, FileMode.Open);
//Kreiramo podatkovni tok za branje
BinaryReader binaryIn = new BinaryReader(podatkovni_tok);
```

Razred **BinaryReader** vsebuje kar nekaj metod, ki jih lahko uporabimo pri obdelavi oz. branju binarne datoteke

Lastnostrazreda BinaryReader	**Razlaga**
BaseStream	Omogoča dodatne možnosti dostopa za pripadajoči podatkovni tok, npr. premik po toku (datoteki)
Metode razreda BinaryReader	**Razlaga**
PeekChar( )	Vrne naslednji razpoložljivi znak v vhodni tok, brez premika na naslednjo pozicijo (naslednji znak). Če ni na voljo nobenega znaka več, metoda vrne vrednost -1.
Read( )	Vrne naslednji razpoložljivi znak iz vhodnega toka in napreduje na novo pozicijo v datoteki.
ReadBoolean()	Vrne logično vrednost z vhodnega toka in napreduje naprej od trenutne pozicije v podatkovnem toku za en **byte**.
ReadByte( )	Vrne byte iz vhodnega toka in ustrezno napreduje naprej od trenutne pozicije v podatkovnem toku.
ReadChar( )	Vrne znak iz vhodnega toka in ustrezno napreduje naprej od trenutne pozicije v podatkovnem toku.
ReadDecimal( )	Vrne decimalno vrednost iz vhodnega toka in napreduje naprej od trenutne pozicije v podatkovnem toku za 16 **bytov**.
ReadInt32( )	Vrne 4 **byte** dolgo predznačeno celo število iz vhodnega toka in napreduje naprej od trenutne pozicije v podatkovnem toku za 4 **byte**.
ReadString( )	Vrne **string** iz vhodnega toka in napreduje naprej od trenutne pozicije v podatkovnem toku za toliko, kot je skupno število znakov v tem **stringu**.
Close( )	Zapre objekt **BinaryReader** in pripadajoči objekt **FileStrem**.

Objekta tipa **BinaryWriter** in **BinaryReader** omogočata zapisovanje in branje izključno osnovnih (primitivnih) podatkovnih tipov (to so znaki, cela števila, decimalna števila, logične vrednosti in zaporedja znakov – stringe). V podatkovni tok tako NE MOREMO zapisati ali pa iz njega prebrati npr. celotne strukture. Če hočemo v podatkovni tok (in nato v datoteko) zapisati strukturo, moramo zapisati vsako polje posebej.

Primer:

Datoteko, ki smo jo kreirali v prejšnjem primeru bomo sedaj odprli za branje in jo prebrali. Prebrane podatke bomo izpisali na zaslon!

```
//določimo pot in ime datoteke
string pot = @"C: \Binarne\Izdelki.dat";
//kreiramo nov podatkovni tok za našo datoteko; če datoteke še ne obstaja se zgradi nova
FileStream fs = new FileStream(pot, FileMode.OpenOrCreate, FileAccess.Read);
//dinamično kreiramo nov objekt tipa BinaryReader
BinaryReader binaryIn = new BinaryReader(fs);

while (binaryIn.PeekChar() != -1) //doler ne zmanjka podatkov (znakov) v datoteki
{
    Izdelek Izd = new Izdelek();//nov objekt tipa Izdelek
    Izd.naziv = binaryIn.ReadString();//podatek o nazivu preberemo kot string
```

```
    Izd.proizvajalec = binaryIn.ReadString();//podatek o proizvajalcu preberemo kot string
    Izd.komadov = binaryIn.ReadInt32();//podatek o komadih preberemo kot CELO ŠTEVLO
    Izd.cena = binaryIn.ReadDecimal();//podatek o ceni preberemo kot DECIMALNO ŠTEVLO

    //prebrane podatke prikažemo na zaslonu
    Console.WriteLine("Izdelek: " + Izd.naziv
                          + "\nProizvajalec: " + Izd.proizvajalec
                          + "\nKomadov: " + Izd.komadov
                          + "\nCena: " + Izd.cena);
}
//zapremo tok podatkov in s tem tudi datoteko
binaryIn.Close();
fs.Close();
```

Lastnost **BaseStream** omogoča dodatne možnosti dostopa za pripadajoči podatkovni tok. Ne glede na to, na katerem mestu v datoteki se trenutno nahajamo, se lahko s pomočjo te lastnosti premaknemo naprej in nazaj po podatkovnem toku (datoteki). Če smo podatkovni tok odprli z lastnostjo **`FileMode.OpenOrCreate`**, lahko v datoteko nekaj zapišemo, se pomaknemo kamorkoli znotraj datoteke in iz nje beremo.

Primer:

```
FileStream fs = new FileStream(@"C:\Binarne\temp.dat", FileMode.OpenOrCreate);
BinaryReader binaryIn = new BinaryReader(fs);

//po podatkovnem toku (datoteki) se premaknimo za dve celi števili naprej
binaryIn.BaseStream.Position = sizeof(int)*2;

//po podatkovnem toku (datoteki) se premaknimo za eno celo število nazaj
binaryIn.BaseStream.Position = -sizeof(int);

//skok na začetek podatkovnega toka
binaryIn.BaseStream.Position = 0;
```

Vaja:

```
/*V Binarno datoteko zapišimo 10 naključnih celih števil. S pomočjo lastnosti
Position se nato postavimo na začetek te datoteke (ne da bi zaprl podatkovni tok), preberimo
števila iz te datoteke in jih izpišimo na zaslon!!!!*/

string datoteka = @"D:\Stevila.dat";
FileStream fs = new FileStream(datoteka, FileMode.OpenOrCreate);
BinaryWriter binaryOut = new BinaryWriter(fs);  //Binarni podatkovni tok
Random naklj = new Random();  //generator naključnih števil
//v datoteko zapišemo 10 celih števil
for (int i = 0; i < 10; i++)
  binaryOut.Write(naklj.Next(-100, 101)); //zapis v binarno datoteko
//Datoteko (podatkovni tok) pustimo ODPRT in se postavimo na začetek datoteke
binaryOut.BaseStream.Position = 0;
//preberemo vsebino datoteke in jo izpišemo na zaslon
BinaryReader binaryIn = new BinaryReader(fs);
Console.WriteLine("Vsebina datoteke: \n");
try
{
   for (int i = 0; i < 10; i++)//preberemo 10 števil iz datoteke
   {
      int st = binaryIn.ReadInt32();
      Console.Write(st + " ");
   }
}
catch { }
binaryOut.Close(); //zapremo podatkovni tok
binaryIn.Close();
fs.Close();
```

Vaja:

```
/*Primer zapisovanje različnih osnovnih podatkovnih tipov v binarno datoteko, ter branje iz
take datoteke*/
```

```
Console.WriteLine("*Kreiranje binarne datoteke in zapis ter branje binarnih podatkov *");
FileStream fs = new FileStream(@"c:\Binarne\temp.dat", FileMode.OpenOrCreate,
                               FileAccess.ReadWrite);

BinaryWriter binOut = new BinaryWriter(fs);

int mojInt = 9;
float mojFloat = 9.8F;
bool mojBool = false;
char[] mojaTabelaZnakov= { 'P', 'o', 'z', 'd', 'r', 'a', 'v' };

binOut.Write(mojInt);
binOut.Write(mojFloat);
binOut.Write(mojBool);
binOut.Write(mojaTabelaZnakov);

binOut.BaseStream.Position = 0;  //premik na začetek podatkovnega toka

Console.WriteLine("\nBranje binarnih podatkov - tako, kot so bili zapisani v datoteko!\n");
BinaryReader binIn = new BinaryReader(fs);

Console.WriteLine(binIn.ReadInt32());   //izpis: 9
Console.WriteLine(binIn.ReadSingle());  //izpis: 9,8
Console.WriteLine(binIn.ReadBoolean()); //izpis: false
for (int i=0;i<7;i++)
   Console.Write(binIn.ReadChar());    //izpis: Pozdrav

Console.WriteLine("\n\nBranje binarnih podatkov - vsak byte posebej!\n");

binOut.BaseStream.Position = 0;  //premik na začetek podatkovnega toka
try
{
    while (true)
    {
       Console.Write(binIn.ReadByte()+" ");
    }
}
catch { }
//zapiranje podatkovnih tokov in povezave z datoteko na disku
binOut.Close();
binIn.Close();
fs.Close();
```

Vaja:

```
/*V binarno datoteko Stev.dat zapiši 100 naključnih celih števil med 65 in 90
  Datoteko nato preberi vse podatke kot cela števila
  Datoteko preberi še kot datoteko znakov*/

//določimo pot in ime datoteke
string datoteka = @"C: \Binarne\Stev.dat";
string dir = @"c:\Binarne";

if (!Directory.Exists(dir))  //če imenik še ne obstaja, ga skreiramo
   Directory.CreateDirectory(dir);

FileStream fs = new FileStream(datoteka, FileMode.Create, FileAccess.Write);
BinaryWriter binaryOut = new BinaryWriter(fs);  //Binarni podatkovni tok
Random naklj = new Random();            //generator naključnih števil
for (int i = 0; i < 100; i++)           //zanka za generiranje 100 naključnih števil
   binaryOut.Write(naklj.Next(65, 91)); //zapis v binarno datoteko

binaryOut.Close(); //zapremo podatkovni tok

//Iz datoteke berimo podatke kot cela števila
fs = new FileStream(datoteka, FileMode.Open, FileAccess.Read);  //povezava z datoteko na disku
BinaryReader binaryIn = new BinaryReader(fs);  //Binarni podatkovni tok
Console.WriteLine("Vsebina datoteke Stev.dat, iz katere binarno beremo cela števila!");

try //uporabimo varovalni blok - ko bo podatkov konec se bo program nadaljeval za catch
{
   while (true)
```

```
   {
      int i = binaryIn.ReadInt32();
      Console.Write(i + "  ");
   }
}
catch
{  }
binaryIn.Close(); //zapremo podatkovni tok

fs = new FileStream(datoteka, FileMode.Open, FileAccess.Read);  //povezava z datoteko na disku
binaryIn = new BinaryReader(fs);  //Binarni podatkovni tok
Console.WriteLine("Vsebina datoteke Stev.dat, iz katere binarno beremo znake!");
try  //uporabimo varovalni blok - ko bo podatkov konec se bo program nadaljeval za catch
{
   while (true)
   {
      char znak = binaryIn.ReadChar();//Iz datoteke berimo podatke kot znake
      Console.Write(znak);
   }
}
catch{  }
binaryIn.Close(); //zapremo podatkovni tok
fs.Close();
```

Vaja:

```
/*V binarno datoteko Tocke.dat zapišimo 10 točk z naključnimi koordinatami. Vsaka točka je
struktura z dvema komponentama - x in y koordinato. Datoteko nato odpri in koordinate vseh
točk izpiši na zaslon*/

struct tocka
{
   public int x, y;
   public tocka(int x,int y)
   {
      this.x=x;
      this.y=y;
   }
}

static void Main(string[] args)
{
   //določimo pot in ime datoteke
   string datoteka = @"C:\Binarne\Tocke.dat";//ime datoteke
   string dir = @"c:\Binarne";
   if (!Directory.Exists(dir))  //če imenik še ne obstaja, ga skreiramo
      Directory.CreateDirectory(dir);

   FileStream fs = new FileStream(datoteka, FileMode.Create, FileAccess.Write);
   BinaryWriter binaryOut = new BinaryWriter(fs);  //Binarni podatkovni tok
   Random naklj = new Random();  //generator naključnih števil
   tocka nova;
   for (int i = 0; i < 10; i++)   //zanka za generiranje 10000 naključnih števil
   {
      nova=new tocka(naklj.Next(101),naklj.Next(101));
      binaryOut.Write(nova.x);
      binaryOut.Write(nova.y);
   }
   binaryOut.Close(); //zapremo podatkovni tok

   //Obdelava binarne datoteke: izpis točk, ki so v datoteki
   fs = new FileStream(datoteka, FileMode.Open, FileAccess.Read);
   BinaryReader binaryIn = new BinaryReader(fs);  //Binarni podatkovni tok
   try
   {
       Console.WriteLine("Seznam točk iz datoteke: ");
       while (true)
       {
          tocka T = new tocka();
          //POZOR! Ker so v datoteki podatki tipa int moramo uporabiti metodo ReadInt32()
          T.x = binaryIn.ReadInt32();//preberemo en podatek - število iz datoteke
          T.y = binaryIn.ReadInt32();//preberemo en podatek - število iz datoteke
          Console.WriteLine("("+T.x + " , " +T.y+")");
```

```
        }
    }
    catch { }
    binaryIn.Close();
    fs.Close();
}
```

Naloge:

Kreiraj binarno datoteko 1000 naključnih celih števil med 1 in 10000. Števila iz te datoteke nato prepiši v tekstovno datoteko tako, da bo vsako število v svoji vrsti, pred številom pa bo še z besedo zapisano število mest tega števila. (če je torej število enako 546 bo pred njim besedica tri…).

Urejanje podatkov

Urejanje podatkov je eno najpogostejših opravil v računalniški praksi (urejanje po abecedi, urejanje po velikosti, ipd.). Pojem **urejanje** oz. **sortiranje** podatkov pomeni preurejanje množice podatkov a_1, a_2, a_3, … a_n, v nek nov novi vrstni red a_{1k}, a_{2k}, a_{3k}, …, a_{nk}, tako, da velja $a_{1k} < a_{2k} < a_{3k} < \ldots < a_{nk}$. Kot metodo za urejanje lahko uporabimo eno od številnih že obstoječih metod za urejanje. Namen urejanja podatkov je olajšati kasnejše postopke, npr. iskanje določenega podatka ali skupine podatkov.

Poglavje o urejanju oz. sortiranju ima več namenov: podati pregled najpomembnejših sortirnih metod, narediti primerjave med njimi, opisati njihove zahtevnosti, napisati primere za posamezne metode, ter ugotoviti, kdaj je katera primernejša od druge. Videli bomo tudi, da za urejanje podatkov obstajajo različni algoritmi, od katerih ima vsak svoje dobre in slabe strani. Načini urejanja se lahko ocenjujejo po različnih kriterijih in je pogosto težko povsem nedvoumno reči, kateri algoritem je nasplošno boljši ali slabši. V računalništvu gre največkrat za kompromise med različnimi kriteriji, kot so na primer prostorska - časovna zahtevnost ali primernost metode za majhno ali veliko število podatkov.

Sortirne metode najprej delimo glede na to, ali urejamo podatke, ki so v neki tabeli, ali pa urejamo podatke, ki so v neki datoteki:

- Za podatke, ki so v tabeli je značilno, da so istočasno v hitrem pomnilniku. Da bi lahko urejali čim več podatkov, si postavimo omejitev, da smemo operirati le z eno kopijo podatkov. Primer: sortiranje odprtih kart na mizi;
- Za podatke v datotekah pa je značilno, da so na nekem masovnem mediju za hrambo podatkov (diskih, trakovih,..), ki so navidezno neomejeni. Primer: sortiranje velikega števila kart po kupčkih.

V tem poglavju se bomo omejili na sortiranje polj (tabel). Omejitve, ki se jih pri tem moramo zavedati, oz. jih upoštevati so:

- Prostorske omejitve (ena kopija podatkov, oz. ena tabela);
- Časovne omejitve (preverjanje časovne zahtevnosti – čim manjše število primerjav in premikov podatkov, ki so v tabeli).

V osnovi poznamo dve vrsti sortirnih metod

- **enostavne:** preprosti algoritmi, na njih sloni velika večina sestavljenih algoritmov, zahtevnost pa narašča praviloma s kvadratom števila podatkov;
- **sestavljene:** zahtevnejši algoritmi, pogosto rekurzivni, njihove prednosti se izkažejo pri velikem številu podatkov.

Navadne metode za urejanje podatkov

Sortiranje z navadnim vstavljanjem

Metoda **Navadnega vstavljanja** je uspešna metoda za *malo podatkov* in v posebnih primerih *delno urejenih podatkov*, pa tudi osnova zelo uspešnih sestavljenih metod **Shellsort** in **Quicksort**.

Zaporedje podatkov razdelimo na **urejeni** in **neurejeni** del. Na začetku vzamemo kot urejenega prvi podatek, ostali so neurejeni. Podatke iz neurejenega dela po vrsti vstavljamo v urejeni tako, da zaporedoma primerjamo podatek, ki ga vstavljamo, s podatkom v urejenem zaporedju in ju menjamo, če je novi podatek manjši.

Primer preurejanja elementov v tabeli: prikazano je začetno stanje tabele 8 celih števil in posamezni koraki urejanja, ki nas privedejo do urejene tabele:

12	**5**	**9**	**14**	**6**	**3**	**21**	**10**
5	**12**	**9**	**14**	**6**	**3**	**21**	**10**
5	**9**	**12**	**14**	**6**	**3**	**21**	**10**
5	**9**	**12**	**14**	**6**	**3**	**21**	**10**
5	**6**	**9**	**12**	**14**	**3**	**21**	**10**
3	**5**	**6**	**9**	**12**	**14**	**21**	**10**
3	**5**	**6**	**9**	**12**	**14**	**21**	**10**
3	**5**	**6**	**9**	**10**	**12**	**14**	**21**

Rešitev v C#:

```
static void vstavljanje(int[] tabela)
{
   int i, j, trenutni;
   for (i = 1; i < elementov; i++)
   {
       trenutni = tabela[i];
       j = i - 1 ;
       while ((j>=0)&&(trenutni < tabela[j]))
       {
           tabela[j + 1] = tabela[j];
           j = j - 1;
       }
       tabela[j + 1] = trenutni;
   }
}
```

Število primerjav je konstantno za vse primere. V vsakem koraku i (i=1…n-1) je potrebno podatek iz neurejenega dela tabele vstaviti v urejeni del. Ta metoda za urejanje podatkov je primerna za urejanje do nekaj tisoč podatkov.

Izboljšana vrsta navadnega vstavljanje se imenuje **dvojiško iskanje.**

Urejanje z izbiranjem

V tem primeru se vse zaporedje vzame za neurejeni del, iz katerega se izbira najmanjši podatek. Tega zamenjamo s prvim podatkom v neurejenem delu, ki se s tem uvrsti v urejenega. V primerjavi z navadnim vstavljanjem imamo pri navadnem izbiranju v vsakem koraku le dve zamenjavi, vendar moramo izbrati najmanjši podatek izmed vseh v neurejenem zaporedju. Primer takega urejanja podatkov v vsakdanjem življenju je npr. urejanje kart pri taroku.

Primer preurejanja elementov v tabeli: prikazano je začetno stanje tabele 8 celih števil in posamezni koraki urejanja, ki nas privedejo do urejene tabele:

12	5	19	2	9	17	3	10
2	5	19	12	9	17	3	10
2	3	19	12	9	17	5	10
2	3	5	12	9	17	19	10
2	3	5	9	12	17	19	10
2	3	5	9	10	17	19	12
2	3	5	9	10	12	19	17
2	3	5	9	10	12	17	19

Rešitev v C#:

```
static void izbiranje(int[] tabela)
{
    int i,j,x,k;
    for (i = 0; i <= N - 2; i++) //N je število elementov tabele
    {
       k = i; x = tabela[i];
       for (j = i + 1; j < N; j++)
       {
          if (tabela[j] < x)
          {
             k = j;
             x = tabela[j];
          }
       }
       tabela[k] = tabela[i]; tabela[i] = x;
    }
}
```

Število primerjav je konstantno za vse primere. V vsakem koraku i (i=1…n-1) je potrebno izbrati najmanjši podatek izmed preostalih. Ta metoda za urejanje podatkov je tako kot metoda vstavljanja primerna za urejanje do nekaj tisoč podatkov.

Urejanje s premenami - Bubblesort

Osnovna značilnost so premene. Podatki se obnašajo kot različno težki mehurčki, ki jih vzgon dviga proti vrhu cevke s tekočino – zato **bubblesort**. V spodnjem primeru so vhodni podatki vpisani v prvem stolpcu.

Primer preurejanja elementov v tabeli: prikazano je začetno stanje tabele 8 celih števil in posamezni koraki urejanja, ki nas privedejo do urejene tabele:

12	5	19	2	9	17	3	10
2	12	5	19	3	9	17	10
2	3	12	5	19	9	10	17
2	3	5	12	9	19	10	17

2	3	5	9	12	10	19	17
2	3	5	9	10	12	17	19
2	3	5	9	10	12	17	19
2	3	5	9	10	12	17	19

Rešitev v C#

```
static void mehurcki(int[] tabela)
{
   int x;
   for (int i = 1; i < elementov; i++)
   {
      for (int j = elementov - 1; j >= i; j--)
      {
         if (tabela[j - 1] > tabela[j])
         {
             x = tabela[j - 1];
             tabela[j - 1] = tabela[j];
             tabela[j] = x;
         }
      }
   }
}
```

Izboljšani Bubblesort – Shakersort (metoda stresanja)

Glavna slabost metodo **bubbelsort** je, da so premene na kratko razdaljo (za eno mesto) neugodne; majhni podatki se hitro dvigujejo, veliki počasi padajo. Posodobljena različica te metode se imenuje **metoda stresanja** – **shakesort**. Na vsakem koraku primerjamo dva sosednja elementa v tabeli po njuni vrednosti, ter ju, če še nista v pravilnem vrstnem redu, zamenjamo. Postopek ponavljanja teče izmenično, enkrat z leve strani, drugič z desne strani tabele. Prednost tega algoritma je v tem, da se zmanjša število primerjanj, število zamenjav pa ostane enako kot pri urejanju z mehurčki. Metodo **shakesort** uporabljamo tedaj, kadar sortiramo manjše tabele, saj je časovna zahtevnost takega urejanja precejšnja.

Sestavljene sortirne metode

Sortiranje z vstavljanjem s padajočim prirastkom – Shellsort

Ideja algoritma je enostavna: množico elementov posebej sortiramo po skupinah, pri čemer v isto skupino spadajo elementi, ki so določeno število mest narazen.

Algoritem ne ureja podatkov samostojno, ampak si pomaga z drugimi algoritmi, najpogosteje z navadnim vstavljanjem. Izboljšan je tako, da pri urejanju ne primerja vseh elementov po vrsti, ampak v začetku ureja na "daleč", torej z večjimi koraki. Algoritem je zanimiv zgolj zaradi tega, ker poveča učinkovitost drugih algoritmov. Z grupiranjem podatkov se na nek način zmanjša čas, ki bi bil drugače potreben pri procesu urejanja podatkov. Največ menjav opravimo na začetku, kasneje so skupine večinoma urejene in je potrebno le manjše število korakov. V vsaki skupini je majhno število elementov ali pa so že kar dobro urejeni, zato ni potrebno veliko število korakov. Na koncu uporabimo navadno vstavljanje, ki v najslabšem primeru postori celotno delo in nas pripelje do urejene tabele.

Rešitev v C#:

```
static void shellSort(int[] tabela)
{
   int k = 1; // izračun koraka za prvo fazo
   // določimo največji korak, ki ga lahko izvedemo znotraj tabele
   while (3 * k + 1 < elementov)
       k = 3 * k + 1;
   while (k > 0) // zanka za faze
   {
      for (int i = k; i < elementov; i++)
      {
         int x = tabela[i]; // izberemo zadnji element skupine
         int j = i - k;
         while (j >= 0 && x < (tabela[j]))//če je element x v skupini manjši od el. pred njim
         {
            tabela[j + k] = tabela[j]; // ju zamenjamo
            j = j - k;
         }
         tabela[j + k] = x; // nov trenutni element
      }
      k = k / 3;            // vsi elementi v skupini so urejeni, zmanjšamo korak
   }
}
```

Hitro urejanje – Quicksort

Metoda **quicksort** je najboljša metoda za sortiranje polj. Metoda je rekurzivna in spominja na **bubblesort** - premene na čim večje razdalje.

Ideja urejanja je takale: izberemo sredinski element - **pivot** (najugodneje bi bilo, če bi bil po vrednosti na sredini vseh elementov). Nato z ene strani iščemo podatek, ki je večji, z druge pa takega, ki je manjši od njega. Podatka zamenjamo. Tako dobimo dve zaporedji s podatki, večjimi (ali enakimi) oz. manjšimi od pivota. Za obe zaporedji postopek ponavljamo, dokler niso zaporedja prazna ali vsebujejo en sam podatek.

Rešitev v C#:

```
public static void quickSort(int[] tabela)
{
    quicksort(tabela, 0, elementov - 1);
}

public static void quicksort(int[] tabela, int i, int j)
{
   if (i < j)
   {
      int p = premeci(tabela, i, j);
      quicksort(tabela, i, p - 1);
      quicksort(tabela, p + 1, j);
   }
}

static int premeci(int[] tabela, int i, int j)
{
   int pivot = tabela[i];
   int m = i;
   int n = j;
   while (m < n)
   {
      while (m < n && tabela[m] <= pivot)
         m = m + 1;

      while (m < n && tabela[n] > pivot)
         n = n - 1;
      if (m < n)
```

```
      {
         int t = tabela[m];
         tabela[m] = tabela[n];
         tabela[n] = t;
      }
   }
   if (tabela[m] > pivot)
     m = m - 1;
   tabela[i] = tabela[m];
   tabela[m] = pivot;
   return m;
}
```

Našteli smo le nekaj metod za urejanje. Poleg naštetih metod obstaja seveda še vrsta drugih. Nekatere med njimi so enostavne in inventivne – urejanje je zaradi tega počasno, druge pa ekstremno kompliciranе - urejanje pa je zelo hitro.

Naslednji program prikazuje primerjavo med posameznimi metodami, saj za vsako od metod za urejanje merimo potreben čas za urejanje elementov tabele 10000 celih števil. Preizkus pokaže, da je pri takšnem številu podatkov najpočasnejše urejanje z metodo mehurčkov, sledita metoda izbiranja in vstavljanja, nato pa metoda s padajočim prirastkom **- shellsort**. Pri tako velikem številu podatkov je seveda najhitrejša metoda hitrega urejanja – **quicksort**.

```
static int elementov = 10000;//število elementov tabele
static int [] tabela;
static Random naklj;
static void Main(string[] args)
{   //metode za urejanje podatkov
    char izbira;
    tabela=new int[elementov];
    naklj=new Random();
    for (int i=0;i<elementov;i++)
       tabela[i]=naklj.Next(1000);
    do
    {
        Console.WriteLine("\nA) Urejanje z vstavljanjem");
        Console.WriteLine("B) Urejanje z izbiranjem");
        Console.WriteLine("C) Urejanje z metodo mehurčkov (BubbleSort)");
        Console.WriteLine("D) Shellsort");
        Console.WriteLine("E) Quicksort");
        Console.WriteLine("I) Izpis trenutne tabele");
        Console.WriteLine("N) Nova tabela");
        Console.WriteLine("K) Konec");
        Console.WriteLine("-----------------------------------");
        Console.Write("Vnesi izbiro: ");
        izbira = Convert.ToChar(Console.ReadLine().ToUpper());
        switch (izbira)
        {
            case 'A':
            {
                vstavljanje(tabela);
                break;
            }
            case 'B':
            {
                izbiranje(tabela);
                break;
            }
            case 'C':
            {
                mehurcki(tabela);
                break;
            }
            case 'D':
            {
                shellSort(tabela);
                break;
            }
            case 'E':
            {
                quickSort(tabela);
                break;
```

```
            }
            case 'I':
            {
                izpis(tabela);
                break;
            }
            case 'N':
            {
                novaTabela(tabela);
                break;
            }

        }
    } while (izbira != 'K');
}

static void izpis(int[] tabela)
{
   Console.WriteLine("Tabela "+elementov+" celih števil:");
   for (int i=0;i<elementov;i++)
   {
        Console.Write("{0,4}",tabela[i]);
        if ((i+1)%10==0)
           Console.WriteLine();
   }
}

static void vstavljanje(int[] tabela)
{
   int i, j, trenutni;
   DateTime zacetek = DateTime.Now;
   for (i = 1; i < elementov; i++)
   { //element z indeksom 0 uporabimo za pomožno spremenljivko
       trenutni = tabela[i];
       j = i - 1 ;
       while ((j>=0)&&(trenutni < tabela[j]))
       {
          tabela[j + 1] = tabela[j];
          j = j - 1;
       }
       tabela[j + 1] = trenutni;
   }
   DateTime konec = DateTime.Now;
   Console.WriteLine("Potreben čas za urejanje: " + Convert.ToString(konec - zacetek));
}

static void izbiranje(int[] tabela)
{
   int i,j,x,k;
   DateTime zacetek = DateTime.Now;
   for (i = 0; i <= elementov - 2; i++)
   {
      k = i; x = tabela[i];
      for (j = i + 1; j < elementov; j++)
      {
         if (tabela[j] < x)
         {
            k = j;
            x = tabela[j];
         }
      }
      tabela[k] = tabela[i]; tabela[i] = x;
   }
   DateTime konec = DateTime.Now;
   Console.WriteLine("Potreben čas za urejanje: " + Convert.ToString(konec - zacetek));
}

static void mehurcki(int[] tabela)
{
   int x;
   DateTime zacetek = DateTime.Now;
   for (int i = 1; i < elementov; i++)
   {
      for (int j = elementov - 1; j >= i; j--)
      {
```

```
         if (tabela[j - 1] > tabela[j])
         {
            x = tabela[j - 1];
            tabela[j - 1] = tabela[j];
            tabela[j] = x;
         }
      }
   }
   DateTime konec = DateTime.Now;
   Console.WriteLine("Potreben čas za urejanje: " + Convert.ToString(konec - zacetek));
}

static void shellSort(int[] tabela)
{
   DateTime zacetek = DateTime.Now;
   int k = 1; // izračun koraka za prvo fazo
   while (3 * k + 1 < elementov) // določimo največji korak, ki ga lahko izvedemo znotraj
                                 //tabele  (1, 4, 13, 40, 121, ...)
       k = 3 * k + 1;
   while (k > 0) // zanka za faze
   {
       for (int i = k; i < elementov; i++)
       {
          int x = tabela[i]; // izberemo zadnji element skupine
          int j = i - k;
          while (j >= 0 && x < (tabela[j])) //če je element x v skupini manjši od tistega pred
                                            //njim
          {
             tabela[j + k] = tabela[j]; // ju zamenjamo
             j = j - k;
          }
          tabela[j + k] = x; // nov trenutni element
       }
       k = k / 3;       // vsi elementi v skupini so urejeni, zmanjšamo korak  }}
   }
   DateTime konec = DateTime.Now;
   Console.WriteLine("Potreben čas za urejanje: " + Convert.ToString(konec - zacetek));
}

public static void quickSort(int[] tabela)
{
   DateTime zacetek = DateTime.Now;
   quicksort(tabela, 0, elementov - 1);
   DateTime konec = DateTime.Now;
   Console.WriteLine("Potreben čas za urejanje: " + Convert.ToString(konec - zacetek));
}

public static void quicksort(int[] tabela, int i, int j)
{
   if (i < j)
   {
      int p = premeci(tabela, i, j);
      quicksort(tabela, i, p - 1);
      quicksort(tabela, p + 1, j);
   }

}

static int premeci(int[] tabela, int i, int j)
{
   int pivot = tabela[i];
   int m = i;
   int n = j;
   while (m < n)
   {
      while (m < n && tabela[m] <= pivot)
         m = m + 1;
      while (m < n && tabela[n] > pivot)
         n = n - 1;
      if (m < n)
      {
         int t = tabela[m];
         tabela[m] = tabela[n];
         tabela[n] = t;
      }
```

```
   }
   if (tabela[m] > pivot)
   { m = m - 1; }
   tabela[i] = tabela[m];
   tabela[m] = pivot;
   return m;
}

static void novaTabela(int[] tabela)
{
   for (int i=0;i<elementov;i++)
     tabela[i]=naklj.Next(1000);
}
```

Predstavitev ASP.NET

Aplikacije, ki so dostopne preko Interneta se imenujejo spletne (Web) aplikacije. Za zagotavljanje prikaza uporabniškega vmesnika, potrebujejo spletne aplikacije nek spletni brskalnik.

Razumevanje Interneta kot infrastrukture

Internet je v bistvu neko ogromno omrežje, tako da so informacije, do katerih lahko preko tega omrežja pridemo, lahko resnično zelo oddaljene. To dejstvo mora seveda vplivati na način, kako bomo načrtovali naše spletne aplikacije. Pri manjših namiznih aplikacijah se lahko npr. pri uporabnikovem ažuriranju podatkov v bazi podatkov poslužujemo zaklepanja zapisov, pri dostopu do podatkov preko Interneta pa tak način prav gotovo ni dober niti ni izvedljiv. Po drugi strani pa je Internet zastrašujoče obsežno omrežje in zaradi tega zelo nepredvidljivo. Odvisni smo od številnih strežnikov, ki usmerjajo naše povpraševanje do določene spletne aplikacije (spletne strani) ki jo želimo doseči, po isti poti pa potuje tudi odgovor. Omrežni protokoli in mehanizmi za predstavitev podatkov, ki so podlaga Internetu, odražajo dejstvo, da so omrežja lahko nezanesljiva in da različni uporabniki uporabljajo različne spletne brskalnike in celo različne operacijske sisteme.

Razumevanje povpraševanj (zahtev) in odgovorov spletnih strežnikov

Uporabnik pridobi preko Interneta s pomočjo spletnega brskalnika možnost komunikacije z neko spletno aplikacijo s pomočjo HTTP protokola (Hypertext Transfer Protocol). Aplikacije so navadno nameščene na nekem gostiteljskem spletnem strežniku, ki bere HTTP zahteve in odloča o tem, katero aplikacijo naj uporabi kot odgovor na določeno povpraševanje. Beseda »aplikacija« je v tem primeru lahko izvršilni progran, ki ga spletni strežnik izvede za izvedbo neke zahteve, lahko pa neko povpraševanje strežnik obdela sam z uporabo lastne interne logike. Kakorkoli bo že povpraševanje obdelano, spletni strežnik bo klientu poslal ustrezen odgovor, spet z uporabo HTTP. Vsebina HTTP odgovora je običajno predstavljena kot HTML stran (Hyper Markup Language); to je jezik, ki ga večina spletnih brskalnikov pozna in razume in ga seveda tudi zna prikazati.

> ❖ Opomba: Aplikacije, ki jih poganjajo uporabniki, ki želijo preko Interneta dostopati do drugih aplikacij, se običajno imenujejo klienti (client applications). Aplikacije, do katerih dostopamo preko Interneta, pa so strežniške aplikacije (server applications).

HTTP je protokol brez povezave, kar pomeni, da je povpraševanje (ali pa odgovor) samostojni paket podatkov. Tipična izmenjava med klientom in aplikacijo, ki teče na spletnem strežniku, lahko vključuje številna povraševanja. Uporabnik si lahko na svojem računalniku samo ogleduje določeno spletno stran, lahko pa tudi v spletno stran vnaša podatke, klika gumbe in na osnovi svojih akcij določa naslednje strani, ki mu omogočajo vnos novih podatkov. Vsako povpraševanje, ki ga pošlje klient nekemu strežniku je ločeno od vseh ostalih zahtev, čeprav jih je poslal isti klient, seveda pa je vsako povpraševanje ločeno tudi od vseh ostalih klientov, ki mogoče istočasno prav tako dostopajo do istega strežnika. Problem pa je, da klientova povpraševanja pogosto zahtevajo neke vrste stanja podatkov na spletnem strežiku. Kot primer vzemimo spletno aplikacijo, ki uporabniku omogoča listanje prodajnih artiklov. Uporabnik mogoče želi kupiti določene artikle iz seznama, v ta namen pa artikle postavlja na neko imaginarno nakupovalno kartico. Zelo uporabna lastnost spletnih aplikacij je npr. zmožnost prikaza trenutne vsebine take nakupovalne kartice. Sedaj pa se seveda pojavi vprašanje, kje naj bo taka nakupovalna kartica shranjena: na spletnem strežniku ali kar pri klientu. V primeru, da bi bila kartica shranjena na spletnem strežniku, bi le-ta moral znati spraviti skupaj različna HTTP povpraševanja, in seveda znati združevati artikle za posameznega klienta. To je seveda možno, a zahteva dodatne obdelave in pa seveda neke vrste bazo podatkov za vodenje kartic klientov, ki pa bi morala biti seveda zelo velika. Alternativa je torej v hranjenju trenutnega stanja podatkov (npr. kartice izbranih artiklov) na klientovem računalniku. V ta namen je bil razvit protokol imenovan ***Piškotki*** (***Cookies***), ki spletnim strežnikom omogoča, da hranijo določene podatke v piškotkih (cookies – to so majhne datoteke) na klientovem računalniku. Pomankljivost takega pristopa je v tem, da mora aplikacija podatke v piškotkih preurediti, preden jih pošlje preko spleta kot del vsakega HTTP povpraševanja, da bo pač spletni strežnik lahko do teh podatkov ustrezno

dostopil. Aplikacija mora tudi poskrbeti za to, da je velikost piškotkov ustrezno omejena. Največja slaba stran piškotkov pa je v tem, da lahko uporabnik na svojem računalniku piškotke onemogoči in ta način spletnemu strežniku onemogoči njihovo hrambo na klientovem računalniku.

Razumevanje ASP.NET

Za izgradnjo in poganjanje spletnih aplikacij mora torej razvojno okolje takih aplikacij upoštevati številne pomembne dejavnike:

- Podpirati mora standard HTTP
- Učinkovito mora znati upravljati s stanjem klienta
- Zagotavljati mora orodja, ki omogočajo enostaven razvoj spletnih aplikacij
- Generirati mora aplikacije, ki so lahko zlahka dostopne preko kateregakoli brskalnika ki podpira HTML
- Biti mora hitro, odzivno in stopenjsko.

Kot odgovor na naštete zahteve je Microsoft je razvil model imenovan ASP (Active Server Page). ASP model omogoča razvijalcem da vključijo svojo programsko kodo v HTML strani. Spletni strežnik, kot npr. IIS (Internet Information Services) lahko izvaja kodo takih aplikacij in jo uporabi za generiranje HTML odgovora. Seveda pa ima ASP model tudi svoje probleme: napisati moramo ogromno kode za izvedbo relativno preprostih zadev, kot je npr. predstavitev strani podatkov iz neke podatkovne baze. Prav tako mešanje programske kode in HTML –ja povzroča probleme berljivosti in podpore takih aplikacij, zmožnosti aplikacij pa niso vedno take kot bi morale biti, saj mora ASP ob vsakem povpraševanju znati prevesti programsko kodo v HTML in vsakič znova, pa čeprav gre za isto kodo.

S prihodom .NET platforme je Microsoft posodobil tudi ASP ogrodje in tako je nastal ASP.NET. Glavne posebnosti ASP.NET so naslednje:

- Poenostavljen programski model, ki uporablja spletne strani, ki imajo vlogo prezentacije in datoteke s kodo, za ločeno hrambo poslovne logike. Kodo lahko pišemo v kateremkoli od podprtih .NET jezikov, vključno s C#. ASP.NET spletni obrazci se prevedejo in se shranijo na spletnem strežniku, kar povečuje zmogljivost spletne aplikacije.
- Strežniške kontrole ki podpirajo dogodke na strežniški strani, a so vrnjene kot HTML in jim tako omogočajo pravilno delovanje v kateremkoli HTML brskalniku. Prav tako je Microsoft razširil številne standardne HTML kontrole, z namenom da jih lahko uporabimo v naši kodi.
- Močne podatkovne kontrole za prikaz, ažuriranje in vzdrževaje podatkov v bazah podatkov.
- Opcije za zajemanje klientovega stanja z uporabo piškotkov na klientovem računalniku, ali pa v posebnem servisu na spletnem strežniku oz. v Microsoft SQL bazi podatkov.

Vsaka nova verzija Visual Studia prinaša številne nove izboljšave, njihov namen pa je čimvečja optimizacija in čimlažje vzdrževanje spletnih aplikacij.

Kreiranje spletnih aplikacij z ASP.NET

Spletna aplikacija, ki uporablja ASP.NET je navadno sestavljena iz ene ali več ASP.NET strani ali spletnih (Web) obrazcov, datotek s programsko kodo in konfiguracijskih datotek.

Spletni (Web) obrazec je shranjen v datoteki s končnico **.aspx**, ki je v bistvu HML datoteka z nekaterimi specifičnimi Microsoft.NET oznakami. Aspx datoteka določa zgradbo in izgled strani. Vsaki .aspx datoteki navadno pripada še datoteka s programsko kodo, ki vsebuje programsko logiko za komponente v .aspx datoteki, kot so npr. metode za obdelavo dogodkov (event handlers) ali pa razne druge koristne metode. Oznaka (tag) ali pa ukaz na začetku vsake .aspx datoteke določa ime in lokacijo pripadajoče datoteke s kodo. ASP.NET prav tako podpira dogodke na nivoju celotne aplikacije, ki pa so definirani v datotekah **Global.asax**.

Vsaka spletna aplikacija lahko vsebuje tudi konfiguracijsko datoteko imenovano *Web.config*. Ta datoteka je v XML formatu, vsebuje pa informacije o varnosti, prevajanju strani, upravljanju z začasnim (cache) pomnilnikom itd.

Kreiranje spletne aplikacije

Za kreiranje spletne aplikacije neposredno v okolju Visual C# potrebujemo višje verzije razvojnega okolja Visual Studio, ali pa orodje Visual Web Developer (lahko tudi express edition).

V naslednjih vajah bomo za razvoj spletnih aplikacij potrebovali vsaj standard verzijo Visual Studia, seveda pa lahko uporabimo tudi katerokoli verzjo Visual Web Developer.

> ❖ POZOR: Za potrebe razvoja spletnih aplikacij ni potrebno, da na našem računalniku teče IIS strežnik. Visual Studio znotraj sebe vključuje lastnen razvojni strežnik imenovan Development Web Server. Ko razvijamo in poganjamo spletno aplikacijo, bo Visual Studio avtomatično uporabil svo lastni strežnik. Po končanem razvoju pa bomo morali za potrebe poganjanja in testiranja spletnih aplikacij nujno uporabiti spletni strežnik IIS.

1. V mapi, v kateri bomo hranili naše projekte, kreirajmo mapo z imenom *Kadri*.

2. V meniju *File* izberimo *New* in kliknimo *Web Site*. Prikaže se pogovorno okno *New Web Site*.

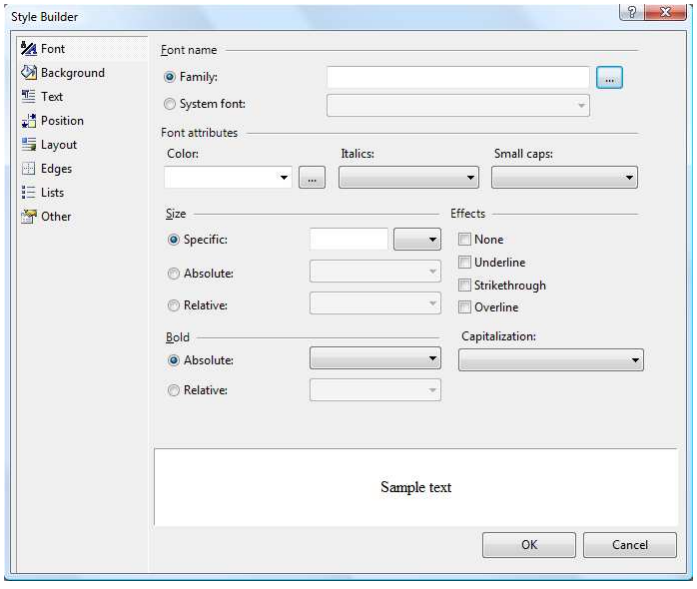
3. Kliknimo gumb *Browse*, se prestavimo v mapo *Kadri* in kliknimo gumb *OK*. Odpre se nov spletni projekt.

4. V oknu *Solution Explorer* izberimo datoteko *Default.aspx*. V oknu *Properties* nato spremenimo privzeto ime (*Default.aspx*) v *Kadri.aspx*.

5. Kliknimo gumb *Design* na dnu osrednjega okna (obrazca), da se prikaže oblikovni pogled našega začetnega obrazca. Obrazec, ki se prikaže, je seveda še prazen. Sedaj lahko pričnemo z dodajanjem gradnikov na obrauec. Gradniki so, tako kot pri kreiranju Windows aplikacije, zbrani v oknu *Toolbox* v levem robu razvojnega okolja.

6. Izberimo okno *Design View*. V oknu *Properties* spremenimo lastnost *Title* objekta *DOCUMENT* v *Kadri*. Besedilo, ki ga bomo napisali, se bo kasneje izpisalo na naslovnem jezičku spletnega brskalnika, ko bomo zagnali našo spletno aplikacijo.

7. Izberimo lastnost *Style* in kliknimo na gumb s tropičjem. Prikaže se pogovorno okno *Style Builder*. Okno omogoča kreiranje stila oz. sloga obrazca (privzeti font, barvo, izgled, in ostale atribute za spletni obrazec in njegove kontrole).

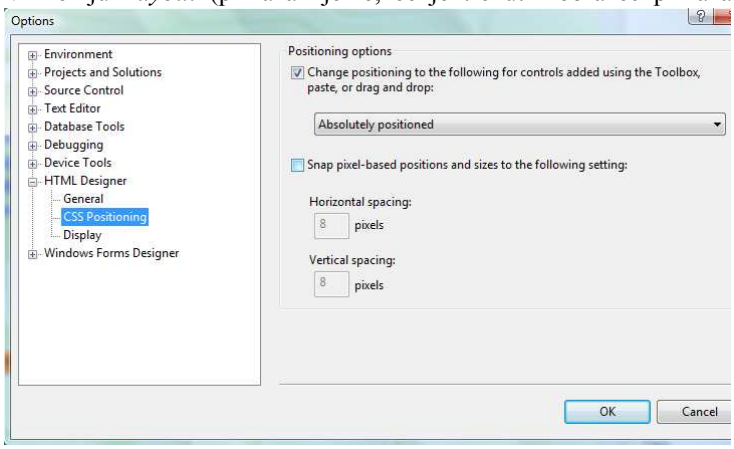


8. Izberimo (levo zgoraj) opcijo *Font* in nato kliknimo na gumb s tropičjem na desni strani. V pogovornem oknu *Font Picker* izberimo *Arial* font in nato kliknimo gumb >> in s tem izbrani font dodamo v seznam *Selected Fonts*. Kliknimo *OK* da se vrnemo v okno *Style Builder*.

9. V seznamu *Color* izberimo *Blue*.

10. V levem delu okna izberimo *Background*. Prikaže se stran za nastavitev ozadja. Odkljukajmo izbirno polje *Transparent*. Če hočemo nastaviti ozadje našega obrazca, potrebujemo najprej neko sliko za ozadje. To sliko skopirajmo v mapo *Kadri*, nato pa zaprimo in ponovno odprimo naš projekt (v nasprotnem primeru sliko za ozadje ne bomo mogli izbrati, ampak bi morali ime datoteke napisati eksplicitno!). Izberimo *Design View*, nato pa v oknu *Properties* ponovno izberimo opcijo *Style* in v pogovornem oknu *Style Builder* izberimo *Background*. Kliknimo na gumb s tropičjem in v mapi *Kadri* izberimo za ozadje sliko, ki smo jo prej skopirali v to mapo. Izbiro potrdimo s klikom na gumb OK in izbrana slika bo sestavljala ozadje naše spletne strani.

11. V meniju *Layout* (prikazan je le, če je trenutni obrazec prikazan v pogledu *Designer*) na levi strani



izberimo opcijo *Position* in kliknimo *Auto-position Options*. V pogovornem oknu nato razširimo opcijo *HTML Designer* (če še ni razširjena) in kliknimo *CSS Positioning*. Na desni strani tega okna označimo (kljukica) *Change positioning to te following for controls added by using the Toolbox, paste, or drag and drop* in nato v spustnem seznamu izberimo opcijo *Absolutely positioned*. S tem smo dosegli, da bomo vse bodoče gradnike, ki jih bomo postavili na obrazec, lahko premaknili kamorkoli in ne bo potrebna nastavitev za vsak gradnik posebej.

12. Odprimo okno *Toolbox* in v njem razširimo skupino gradnikov *Standard*. Ta vsebuje kontrole (gradnike) ki jih lahko postavljamo na spletne obrazce. Kontrole so podobne tistim, ki smo jih uporabljali pri kreiranju okenskih obrazcev, razlika je le v tem, da so te kontrole načrtovane za uporabo v HTML okolju.

13. Na obrazec sedaj drugega za drugim postavimo gradnike in jih razporedimo tako kot prikazuje slika:

Label1, FontName=ArialBlack, FontSize=XX-Large (ostale labele naj imajo privzeto velikost fonta).

TextBox – ime gradnika naj bo *tbID*.
TextBox – ime gradnika je *tbIme*.
TextBox – ime gradnika je *tbPriimek*.

DropDownList – ime gradnika je *ddlPozicija*.

4 x RadioButton – imena gradnikov so *rbDelavec*, *rbVodja*,

rbPodpredsednik in *rbPredsednik*. Vsem štirim gumbom določimo tudi lastnost *GroupName* – ta naj bo pri vseh štirih radijskih gumbih enaka *DelovnoMesto*. Z lastnostjo GroupName določimo, v katero grupa spada določena množica gumbov. Za vse radijske gumbe, ki so v neki grupi namreč velja, da je hkrati lahko izbran le eden od njih. V tej vaji bi sicer lahko uporabili tudi gradnik *RadioButtonList*, ki je sicer bolj primeren, kadar je radijskih gumbov več kot en sam, a zaradi enostavnosti smo v tem primeru uporabili kar navadne radijske gumbe.

- Opomba: Pri nastavljanju enakih lastnosti večim radijskim gumbom lahko stisnemo tipko Shift, nato izberemo vse radijske gumbe in nato v oknu Properties nastavimo ustrezne lastnosti vsem gumbom hkrati.

Na dnu strani sta še dva gumba z imenoma *bShrani* in *bPreklici*, ki ju po želji še pobarvajmo.

- Opomba: Gradnik postavimo na obrazec tako, da ga primemo z miško in povlečemo na obrazec. Pozicija gradnika na obrazcu bo na začetku povsem v zgornjem levem kotu obrazca, od koder lahko potem ta gradnik povlečemo na želeno pozicijo na obrazcu (seveda če smo prej nastavili opcijo *Absolutely positioned*. Drug način postavljanja gradnikov na obrazec pa je ta, da na ustrezen gradnik le dvokliknemo – gradnik se bo prav tako pojavil v zgornjem levem kotu obrazca. Seveda pa lahko položaj gradnika, ko je enkrat že na obrazcu, določimo tudi v oknu Properties.

Gradnike, ki so že na obrazcu lahko poravnamo v poljubni smeri tako, da gradanike, ki jih želimo poravnati, najprej označimo (držimo tipko *shift* in klikamo po gradnikih), nato pa v meniju *Format* izberemo opcijo *Align* in končno še ustrezno vrsto poravnave.

V vsakem trenutku si lahko ogledamo tudi trenutno izvorno kodo, ki jo je za nas generiralo razvojno okolje. Na dnu obrazca kliknimo gumb Source in prikazala se bo HTML prezentacija našega obrazca. V primeru našega obrazca izgleda HTML koda takole:

```
<%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Kadri.aspx.cs"
Inherits=" Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Kadri</title>
</head>
<body style="background-image: url(Predstavitev1.jpg); color: blue; font-family:
Arial">
    <form id="form1" runat="server">
    <div style="background-image: url(Predstavitev1.jpg)">
        <asp:Label ID="Label1" runat="server" Style="z-index: 100; left: 29px;
position: absolute;
            top: 30px" Text="Podatki o zaposlenem delavcu" Font-Bold="True" Font-
Names="Arial Black" Font-Size="XX-Large"></asp:Label>
        <asp:Label ID="Label2" runat="server" Style="z-index: 101; left: 120px;
position: absolute;
            top: 141px" Text="Ime"></asp:Label>
        <asp:TextBox ID="tbIme" runat="server" Style="z-index: 102; left: 162px;
position: absolute;
            top: 140px" Width="132px"></asp:TextBox>
        <asp:Label ID="Label3" runat="server" Style="z-index: 103; left: 367px;
position: absolute;
            top: 142px" Text="Priimek"></asp:Label>
        <asp:TextBox ID="tbPrimek" runat="server" Style="z-index: 104; left: 436px;
position: absolute;
            top: 140px" Width="220px"></asp:TextBox>
        <asp:Label ID="Label4" runat="server" Style="z-index: 105; left: 36px;
position: absolute;
            top: 108px" Text="Številka delavca"></asp:Label>
```

```
        <asp:TextBox ID="tbID" runat="server" Style="z-index: 106; left: 162px;
position: absolute;
            top: 107px" Width="89px"></asp:TextBox>
        <asp:Label ID="Label5" runat="server" Style="z-index: 107; left: 40px;
position: absolute;
            top: 190px" Text="Delovno mesto"></asp:Label>
        <asp:RadioButton ID="rbDelavec" runat="server" Checked="True"
GroupName="DelovnoMesto"
            Style="z-index: 108; left: 162px; position: absolute; top: 188px"
Text="Delavec" />
        <asp:RadioButton ID="rbPredsednik" runat="server" GroupName="DelovnoMesto"
Style="z-index: 109;
            left: 163px; position: absolute; top: 260px" Text="Predsednik"
AutoPostBack="True" OnCheckedChanged="rbPredsednik_CheckedChanged" />
        <asp:RadioButton ID="rbPodpredsednik" runat="server" GroupName="DelovnoMesto"
Style="z-index: 110;
            left: 164px; position: absolute; top: 236px" Text="Podpredsednik"
AutoPostBack="True" OnCheckedChanged="rbPodpredsednik_CheckedChanged" />
        <asp:RadioButton ID="rbVodja" runat="server" GroupName="DelovnoMesto" Style="z-
index: 111;
            left: 163px; position: absolute; top: 212px" Text="Vodja"
AutoPostBack="True" OnCheckedChanged="rbVodja CheckedChanged" />
        <asp:DropDownList ID="ddlPozicija" runat="server" Style="z-index: 112; left:
438px;
            position: absolute; top: 188px" Width="221px">
        </asp:DropDownList>
        <asp:Button ID="bShrani" runat="server" BackColor="#FF8000" ForeColor="Black"
Style="z-index: 113;
            left: 156px; position: absolute; top: 356px" Text="Shrani" Width="95px"
OnClick="bShrani_Click" />
        <asp:Label ID="Label6" runat="server" Style="z-index: 114; left: 365px;
position: absolute;
            top: 190px" Text="Pozicija"></asp:Label>
        <asp:Button ID="bPocisti" runat="server" BackColor="#FF8000" ForeColor="Black"
Style="z-index: 115;
            left: 345px; position: absolute; top: 356px" Text="Počisti" Width="95px" />
        <asp:Label ID="lInfo" runat="server" Style="z-index: 116; left: 166px;
position: absolute;
            top: 310px" Text="Info..."></asp:Label>
          

    </div>
    </form>
</body>
</html>
```

Gradnike, ki so že na obrazcu lahko poravnamo v poljubni smeri tako, da gradanike, ki jih želimo poravnati, najprej označimo (držimo tipko *shift* in klikamo po gradnikih), nato pa v meniju *Format* izberemo opcijo *Align* in končno še ustrezno vrsto poravnave.

14. Ko smo na obrazec postavili vse potrebne gradnike odprimo Debug meni in kliknimo opcijo Start Debugging. Visual Studi nam zgradil aplikacijo, startal se bo ASP.NET Development Server, nato pa se zažene še Internet Explorer, ki nam prikaže naš obrazec. **Za razhroščevanje (debugging) naše aplikacije pa moramo obvezno imeti vse administratorske pravice**, sicer razhroščevanje NI možno! V kolikor je na računalniku nameščen operacijski sistem Windows Vista, moramo torej Visual Studio pognati z opcijo *Run As Administrator*.

- Opomba: Pri prvem zagonu spletne aplikacije z uporabo opcije Start Debugging, se prikaže sporočilno okno s sporočilom, da razhroščevanje (debugging) ni omogočeno. Izberemo lahko eno od dveh ponujenih opcij: »Run without Debuggin« ali pa »Modify the Web.config file to enable debugging«. Delo v načinu Debug je seveda zelo koristno, še posebej če želimo s pomočjo prekinitvenih točk spremljati delovanje programa stavek za stavkom. Seveda pa je v tem primeru delovanje programa počasnejše in je zato priporočljivo, da preden damo našo spletno stran na nek strežnik v eksplotacijo, razhroščevanje onemogočimo. To storimo tako, da v datoteki Web.config odstranimo vrstico `<compilation debug="true"/>`.

15. Skupno ime za vse kontrole (oz. za vse gradnike), ki jih postavljamo na obrazce, je strežniške (server) kontrole. Strežniške kontrole so podobne standardnim HTML enotam, ki jih lahko uporabljamo pri kreiranju običajnih spletnih strani, razlika pa je seveda v tem, da so bolj programabilne. Za večino strežniških kontrol namreč obstajajo obdelovalci dogodkov (event handlers), metode in pa seveda lastnosti. Programska koda, ki jo napišemo za ustrezen gradnik, se izvaja na nekem strežniku dinamično v času izvajanja aplikacije. Poglejmo sedaj še enkrat del HTML kode našega obrazca. Za vsako kontrolo (gradnik) na našem obrazcu, obstaja namreč ustrezna HTML koda in takole izgleda npr. HTML koda za labelo z imenom Label1:

```
<asp:Label ID="Label1" runat="server"
    Style="z-index: 100; left: 37px;
    position: absolute; top: 24px"
    Text="Podatki o zaposlenem delavcu"
    Font-Bold="True" Font-Names="Arial Black"
    Font-Size="XX-Large">
</asp:Label>
```

Prvo kar opazimo je tip kontrole (gradnika): **`asp:Label`**. Vse kontrole spletnih obrazcev živijo v imenskem prostoru »**asp**«, ker so pač tako definirane s strani Microsofta. Drugi pomemben atribut v zgornjem stavku je atribut **`runat="server"`**. Ta atribut označuje, da lahko do njega dostopamo programsko s pomočjo kode, ki teče na spletnem strežniku. S pomočjo te kode lahko poizvedujemo in spreminjamo vrednosti katerekoli lastnosti teh kontrol (npr. spremenimo lastnost *Text*).

> ❖ Opomba: Vse kontrole v HTML kodi morajo biti pravilno zaključene z oznako **`</asp:kontrola>`**, kjer besedica ***kontrola*** seveda označuje vrsto kontrole (gradnika), npr. Label, Button…

ASP.NET podpira tudi HTML kontrole. Če v oknu *Toolbox* razširimo skupino gradnikov HTML, dobimo seznam vseh HTML kontrol. To so kontrole, ki jih Microsoft zagotavlja v originalnem Active Server Page modelu. Njihova glavna vloga je zagotavljanje, da lahko obstoječe ASP strani lažje uvozimo v ASP.NET. V kolikor pa želimo graditi neko spletno aplikacijo od samega začetka, pa bomo seveda raje uporabljali standarne kontrole za izgradnjo spletnih obrazcev. HTML kontrole imajo poseben atribut imeovan *runat*, ki razvijalcem omogoča, da sami določijo kje naj se izvaja koda, ki jo napišemo znotraj nekega dogodka. Za razliko od kontrol na spletnih obrazcih, je privzeta lokacija za izvedbo kode pri HTML kontrolah kar na strani uporabnika, v njegovem spletnem brskalniku in ne na strežniku. Seveda pa mora v tem primeru uporabnikov brskalnik to funkcionalnost podpirati.

16. Spletna stran, ki smo jo pripravili seveda zahteva tudi nekaj obdelav dogodkov: v seznam pozicij (*ddlPozicija*) je potrebno dinamično zapisati vse možne pozicije glede na izbiro delovnega mesta, potrebno pa je tudi je napisati kodo, ki se bo izvedla ob kliki na gumba *bShrani* in *bPreklic*. V Solution Explorerju razširimo datoteko *Kadri.aspx* in prikaže se datoteka *Kadri.aspx.cs*. To je datoteka, ki bo v resnici hranila kodo zapisano v C# in ki se bo odzivala na dogodke, ki jih želimo napisati. Ta datoteka je poznana tudi pod imenom datoteka s kodo v ozadju (*code behind file*). Ta zmožnost ASP.NET-a nam omogoča, da lahko povsem ločimo kodo napisano v C# od prikazane logike spletne aplikacije. Oglejmo si kodo v datoteki Kadri.aspx v oknu Source View. Zanima nas le prva vrstica, ki izgleda nekako takole:

```
<%@ Page Language="C#"  . . .  CodeFile="Kadri.aspx.cs"  . . . %>
```

Ta stavek nam pove, da ta datoteka vsebuje programsko kodo za spletno aplikacijo, jezik v katerem je ta koda napisana pa je C#. Poleg C# sta podprta še jezika Visual Basic in JScript.

17. V Solution Explorer-ju dvokliknimo na datoteko Kadri.aspx.cs. Vsebina te datoteke se prikaže v oknu Code and Text. Na začetku datoteke je množica using stavkov, ki nam sporočajo, da bomo uporabljali imenski prostor **System.Web** in nekatere podprostore, kjer se nahajajo razredi ASP.NET. Celotna koda je znotraj razreda imenovanega **_Default**, ki je izpeljan iz **System.Web.UI.Page**. To je razred, iz katerega so izpeljani vsi spletni obrazci. Trenutno je znotraj razreda _Default ena sama metoda imenovana Page_Load, ki se izvede vsakič, ko je ta spletna stran prikazana. Znotraj te metode lahko napišemo poljubno kodo, za katero inicializacijo kakršnih koli podatkov, ki jih zahteva naš spletni obrazec.

18. Najprej napišimo novo metodo in jo poimenujmo inicIzobrazba(). To metodo bomo poklicali znotraj metode Page_Load. Obe metodi izgledata takole:

```
protected void Page Load(object sender, EventArgs e)
    {
        if (!IsPostBack)//lastnost IsPosback je false le pri prvem prikazu spletne
                        //strani, ki ji pripada tale datoteka
        {
            inicIzobrazba();
        }
    }
    private void inicIzobrazba()
    {
        //v spustni seznam dodajmo tri postavke za delovno mesto Delavec(privzeti gumb)
        ddlPozicija.Items.Clear();
        ddlPozicija.Enabled = true;
        ddlPozicija.Items.Add("Analitik");
        ddlPozicija.Items.Add("Razvijalec");
        ddlPozicija.Items.Add("Oblikovalec");
        rbPodpredsednik.Checked = false;
        rbPredsednik.Checked = false;
        rbVodja.Checked = false;
        rbDelavec.Checked = true;
    }
```

Ker smo v dogodek *Page_Load* vstavili klic dogodka *inicIzobrazba*, moramo vedeti, da se bo ta dogodek izvedel vsakič ko bo strežnik poslal ta obrazec uporabnikovemu brskalniku in ne le ko ga bo poslal prvič. Ko bo npr. uporabnik kliknil nek gumb na obrazcu, se obrazec pošlje na strežnik v obdelavo. Strežnik bo obrazec obdelal in ga poslal nazaj uporabnikovemu brskalniku. Mi pa seveda ne želimo, da bi se metoda inicIzobrazba izvajala ob vsakem prikazu obrazca, saj gre v tem primeru za povsem nepotrebno obdelavo, pa še pri izdelavi komercialnih spletnih strani s tem izgubljamo na njihovih performansah (če drugega ne, je za vsako obdelavo potreben čas, ki pa ga uporabniki nimajo na pretek). Zaradi tega obstaja možnost, da določimo kdaj naj se določeni stavki znotraj metode *Page_Load* izvajajo, kdaj pa ne. To nam omogoča lastnost *IsPostBack*, ki je na začetku *false*, ko pa je spletna enkrat že prikazana, pa je vrednost te lastnosti enaka true. Ker mi želimo, da se metoda *inicIzobrazba* izvede le pri prvem prikazu obrazca, smo jo postavili v ustrezen if stavek, katerega pogoj je testiranje lastnosti *IsPostBack*.

19. Preklopimo na *Kadri.aspx* datoteko in pogled obrazca spremenimo v *Design View*. Na obrazcu izberimo radijski gumb *rbDelavec*, v oknu properties kliknimo gumb *Events*, nato pa dokliknimo na dogodek *CheckedChanged*. Ta dogodek se zgodi vselej kadar uporabnik klikne na radijski gumb in je njegova vrednost spremenjena. Visual Studio nam je za obdelavo tega dogodka sedaj zgeneriral ogrodje metode *rbDelavec_ CheckedChanged*. V telo te metode zapišimo stavka

```
inicIzobrazba();
rbDelavec.AutoPostBack = true;
```

Ob kliku na ta radijski gumb se torej izvede metoda *inicIzobrazba*(), ki poskrbi za ustrezno vsebino gradnika *ddlPozicija*. Z drugim stavkom omogočimo, da se bo pri kasnejših izbirah tega radijskega gumba ustrezen dogodek obdelal na strežniku.

20. Preklopimo spet na oblikovni pogled obrazca *Kadri.aspx*, izberimo radijski gumb *rbVodja*, v oknu Properties kliknimo gumb *Events* in nato dvokliknimo na dogodek *CheckedChanged*. V telo metode zapišimo stavke:

```
ddlPozicija.Items.Clear();
ddlPozicija.Enabled = true;
ddlPozicija.Items.Add("Vodja projektov");
ddlPozicija.Items.Add("Vodja oddelka");
ddlPozicija.Items.Add("Vodja Informatike");
```

Podobno zapišimo še dogodek *CheckedChanged* za radijski gumb rbPodpredsednik:

```
ddlPozicija.Items.Clear();
ddlPozicija.Enabled = true;
ddlPozicija.Items.Add("Prodaja");
```

```
ddlPozicija.Items.Add("Marketing");
ddlPozicija.Items.Add("Prizvodnja");
ddlPozicija.Items.Add("Kadrovski oddelek");
```

in končno še dogodek *CheckedChanged* za radijski gumb rbPredsednik:

```
ddlPozicija.Items.Clear();
ddlPozicija.Enabled = false;
```

Za predsednika družbe ni na voljo nobene izbire v spustnem seznamu, zato smo lastnost *enable* gradnika *ddlPozicija* postavili na *false*.

21. Zapišimo še dogodka *Click* za gumba *bShrani* in *bPocisti*. Pri kliku na gumb bShrani naj bi bil dogodek sicer namenjen shranjevanju v neko bazo podatkov, a zankrat bomo ob kliku le oblikovali nek string (sporočilo), ki ga bomo nato prikazali na obrazcu. Dvokliknimo torej na dogodek *Click* gradnika *bShrani* in napišimo naslednje stavke:

```
string pozicija = "";
if (rbDelavec.Checked)
      pozicija = "Delavec";
if (rbVodja.Checked)
      pozicija = "Vodja";
if (rbPodpredsednik.Checked)
      pozicija = "Podpredsednik";
if (rbPredsednik.Checked)
      pozicija = "Predsednik";
lInfo.Text = "Zaposleni: " + tbID.Text + " " + tbIme.Text + " " +
      tbPrimek.Text +",   &nbspIdentifikacijska številka: " +
      tbID.Text + ",   &nbspPozicija: " + pozicija;
```

Znak **` `** je oznaka za presledek v HTML kodi: če bi v tekstu naredili le običajni znak za presledek (s pomočjo tipke *Space*), bi brskalnik ignoriral vse presledke razen prvega! Labela *Linfo* ima vlogo prikaza uporabnikovega vnosa, izbranega radijskega gumba in ustrezne izbire v spustnem seznamu.

Še dogodek *Click* za gumb *bPocisti*:

```
tbIme.Text = "";
tbPrimek.Text = "";
tbID.Text = "";
rbPodpredsednik.Checked = false;
rbPredsednik.Checked = false;
rbVodja.Checked = false;
rbDelavec.Checked = true;
rbDelavec.AutoPostBack = true;
inicIzobrazba();
lInfo.Text = "";
```

Ta koda pobriše dosedanje vnose in vzpostavi začetno stanje – polja so prazna, izbran je radijski gumb *rbDelavec*, vsebina spustnega seznama pa je prirejena opciji *Delavec*.

22. Spletna aplikacija je sedaj pripravljena za uporabo. Ko jo poženemo in klikamo na radijske gumbe bomo opazili, da ob kliku na radijski gumb obrazec za trenutek zatrepeta – to je seveda zaradi tega, ker se ob kliku na gumb celoten obrazec pošlje strežniku, da ga obdela in ga vrne ustrezno spremenjenega (obdela se ustrezni dogodek, osveži oz. spremeni se spustni seznam, …).

23. V Internet Explorerju (medtem, ko imamo zagnano našo spletno aplikacijo) kliknimo na gumb *View Source* in si oglejmo izvorno kodo naše HTML strani. Zažene se Notepad in prikaže se HTML koda. Opazimo, da besedica »***asp***« ni omenjena nikjer več, nikjer tudi ne bomo opazili kode zapisane v C#. Namesto tega so vse strežniške kontrole in njihove vsebine prevedene v ustrezne HTML kontrole (in nekatere JavaScripte). To je ena izmed temeljnih značilnosti strežniških kontrol. Do njih dostopamo s programsko tako kot do običajnih .NET Framework objektov, skupaj z njihovimi lastnostmi, metodami in dogodki, ko pa jih vrnemo spletnemu strežniku, so konvertirane v HTML. Na ta način je lahko taka spletna stran prikazana v poljubnem HTML brskalniku.

Strežniške kontrole so nedvomno zelo močna sestavina ASP.NET, a imajo seveda svojo ceno. Zapomniti si moramo, da čeprav so dogodki sproženi s strani uporabnika, je celotna koda izvedena na spletnem strežniku. Vsakič, ko je sprožen nek dogodek, se preko omrežja pošlje HTML koda na spletni strežnik, da jo le-ta obdela. Naloga spletnega strežnika je seveda obdelava te zahteve in vračanje HTML odgovora na to zahtevo nazaj uporabniku. Strežnik pa mora od uporabnika dobiti tudi podatke o njegovih vnosih v polja na obrazcu, saj bo le tako vrnjeni obdelani obrazec imel pravilno vsebino. Če pogledamo še enkrat izvorno kodo naše HTML strani (VieW Code v Internet Brskalniku), bomo nekja na dnu opazili vrstico s skritim vnosnim poljem (hidden field). Ta vrstica izgleda pribljižno takole:

```
<input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"
value="/wEWDQKO2eW7DgKy4rnAAwKI3ea7DAK57dLhAQLmyYSqDAK8quKBDgKRpfOzBwLp6PtuAuPfxpwHArmphqcKAqzBmVcCt
puFuQoC5KPrwAzGQvp5F6q4stA8sFK7x5GhWs4BCw==" />
```

Informacija, ki jo vsebuje ta vrstica, je vsebina kontrol (gradnikov) našega obrazca, ki pa je zapisana v kodirani obliki. Vrstica je poslana strežniku vsakič ko kateri od dogodkov povzroči t.i. *postback*. Spletni strežnik bo po generiranju HTML odgovora to informacijo uporabil za osveževanje polj na naši spletni strani. Vsi ti podatki pa imajo seveda velik vpliv na delovanje naše spletne strani. Več kontrol ko imamo na obrazcu, več podatkov o njihovem stanju je posredovano med spletnim strežnikom in spletnim brskalnikom. Če smo na naši strani napisali veliko odzivnih dogodkov, bo njihovo procesiranje na spletnem strežniku pogosto, kar pa seveda ni najboljše, saj lahko postane odzivnos počasna. Zaradi tega se moramo držati navodila, da naj bo naša spletna stran relativno enostavna, izogibati se moramo prekomernemu številu odzivnih dogodkov in biti selektivni pri določanju lastnosti *EnableViewState*, da na ta način preprečimo nepotrebno prenašanje podatkov po omrežju. Največkrat zelo pomaga že nastavitev lastnosti *EnableViewState* na *false*.

Kreiranje in uporaba tem (Theme)

Ko pričnemo z oblikovanjem spletnega obrazca lahko najprej določimo njegov stil. S stilom obrazca določimo privzeti font, barvo kontrol na obrazcu, lahko pa določimo tudi privzete vrednosti za druge atribute, kot npr. način kako so oblikovani in številčeni seznami, … Seveda pa se na ta način določeni stil nanaša le na trenutni obrazec, običajno pa se naša spltena stran sestoji iz več deset ali tudi sto obrazcev. Če bi določali stil vsakemu obrazcu posebej, je to nepotrebna časovna potrata, pa še ob spremembi stila je potrebno ponovno spremeniti stil vsakemu obrazcu posebej. V takem primeru je zelo koristna uporaba t.i. tem (*Theme*). Tema je nabor lastnosti, stilov in slik, ki jih lahko dodamo kontrolam na naši strani, ali pa jih dodamo globalno za vse strani naše spletne aplikacije.

Definicija nove teme

Pojem Tema v C# označuje množico datotek, ki hranijo podatke o stilih obrazcev (t.i. *Skin Files* oz. datoteke *preoblek*), nahajajo pa se v podimeniku z imenom *App_Themes* vsake spletne strani. Vsaka taka datoteka označuje privzete lastnosti za določen tip kontrol (gradnikov), pri čemet je uporabljena sintaksa, ki je zelo podobna sintaksi spletnega obrazca, kadar jo odpremo v oknu *Source View Window*. Tule primer vsebine vsebine ene od takih datotek, ki vsebuje privzete nastavitve za kontrole tipa *TextBox* in *Label*:

```
<asp:TextBox BackColor="Blue" ForeColor="White" runat="server" />
<asp:Label BackColor="White" ForeColor="Blue" runat="server" Font_Bold="True"/>
```

V datoteki *Skin File* lahko hranimo številne lastnosti obrazcev, ne pa vseh. V tej datoteki ne moremo npr. hraniti vrednost lastnosti *AutoPostBack*. Prav tako ne moremo kreirati *Skin* datoteke za prav vsako vrsto kontrole, a za večino običajnih kontrol pa lahko.

Uporaba nove teme

Ko smo kreirali *Skin* datoteke za neko novo temo, lahko to novo temo uporabimo v naši spletni strani tako, da spremenimo atribut *@Page*, ki je zapisan na začetku prikazane vsebine strani v oknu *Source View Window*. Če smo npr. Skin datoteke za našo spletno stran shranili v mapi *App_Theme/ModraTema* znotraj mape, v kateri je naša spletna stran, jo lahko uporabimo takole:

```
<%@ Page Theme="ModaTema" . . . %>
```

V kolikor pa želimo novo temo uporabiti na vseh straneh naše spletne aplikacije, pa lahko spremenimo datoteko *Web.config* in to temo zapiemo v elementu ***pages*** takole:

```
<configuration>
   <system.web>
      < pages theme=="ModaTema" />
   </system.web>
</configuration>
```

Če na ta način spremenimo definicijo neke teme, bodo to novo obliko pri ponovnem prikazu prevzele vse kontrole in vse spletne strani.

Za vajo kreirajmo neko svojo temo, nato pa jo bomo uporabili na naši začetni spletni aplikaciji *Kadri*.

1. Odprimo splteno stran Kadri in v Solution Explorerju napravimo desni klik na projektno mapo Kadri. Izberimo Opcijo *Add ASP.NET Folder* in kliknimo vrstico *Theme*. V projektu se pojavi nova mapa *App_Theme*, pod njo pa podmapa imenovana *Theme1*. To podmapo preimenujmo v *KadriTheme*.

2. V Solution Explorerju napravimo desni klik na mapo KadriTheme in nato izberimo opcijo Add New Item. Pojavi se pogovorno okno Add New Item, v njem pa prikazane vrste datotek, ki jih lahko hranimo v mapi t temami.

3. Kliknimo na vzorec *Skin File*, za ime teme vtipkajmo *Kadri.skin* in nato kliknimo gumb *Add*. V mapo *KadriTheme* smo na ta način dodali datoteko *Kadri.skin*, vsebina te nove datoteke pa je tudi prikazana v oknu *Code and Text Window*. V to datoteko sedaj dodajmo naslednje vrstice:

   ```
   <asp:TextBox BackColor="Red" ForeColor="Whire" Font_Bold="True" Runat="Server" />
   <asp:Label BackColor="White" ForeColor="Red" Runat="Server" Font-Bold="True" />
   <asp:RadioButton BackColor="White" ForeColor="Red" Runat="Server" />
   <asp:Button BackColor="Red" ForeColor="White" Runat="Server" Font-Bold="True" />
   <asp:DropDownList BackColor="White" ForeColor="Red" Runat="Server" />
   ```

   V preprostem seznamu lastnosti smo gradnikom *TextBox*, *Button* in *DropDownList* nastavili lastnosti tako, da so kontrole Bele barve na rdečem ozadju, gradnikoma Label in RadioButon pa rdeči tekst na belem ozadju. Tekst pri labelah in radijskih gumbih bo prikazan v odebeljeni obliki izbranega fonta.

   ❖ Opomba: Urejevalnik preoblek (Skin editor) je zelo enostaven in ne vključuje tehnologije Intellisense. Če pri pisanjju vsebine te datoteke naredimo kakršnokoli napako, se bo aplikacija kljub temu izvedla, a vnosi v tej datoteki bodo lahko prezrti. Pri kasnejšem zagonu te aplikacije moramo biti pozorni na to, če so vse kontrole prikazane tako kot smo želeli – v nasprotnem primeru ponovno natanko preglejmo sebino datoteke *xxx.skin* in popravimo napake.

   Že prej je bilo omenjeno, da sta vsaj dva načina, kako lahko neko temo apliciramo (vstavimo) v naš spletni obrazec: nastavimo lahko atribut *@Page* za vsako stran posebej, ali pa z uporabo konfiguracijske datoteke Web.config določimo globalno temo preko vseh strani naenkrat. S pomočjo tega drugega načina bomo v naslednjih korakih globalno spremenili temo naše spletne strani *Kadri*.

4. Če v *Solution Explorerju* že obstaja datoteka *Web.config*, jo dvokliknimo, da se nam prikaže njena vsebina. Če pa datoteka *Web.config* še ne obstaja, pa v *Solution Explorerju* napravimo desni klik na projekt *Kadri* in nato *Add New Item*. Prikaže se pogovorno okno *Add New Item*, ki prikazuje vse tipe datotek, ki jih lahko dodamo k projektu. Izberimo *Web Configuratin File*, se prepričajmo, da je ime datoteke enako *Web.config*, nato pa kliknimo gumb *Add* – na ekranu se prikaže vsebina te datoteke.

5. Premknimo se na konec datoteke Web.config in ***vstavimo novo vrstico tik pred vrstico </system.web>***.

   ```
   <pages theme="KadriTheme" />
   ```

6. V meniju *Debug* sedaj izberimo opcijo *Start Without Debugging*. Zažene se privzeti brskalnik in nam prikaže novo oblikovan obrazec naše spletne strani. Prepričajmo se, da so spremembe res takšne kot smo želeli, v nasprotnem primeru pa popravimo vsebino datoteke *Kadri.skin*. V datoteki *Kadri.skin* smo

namenoma izbrali kričeče barve, da bomo spremembe na spletni strani res opazili. Nastavitve barv v datoteki *Kadri.skin* zato sedaj popravimo oz. jih nastavimo po svojem okusu.

- Opomba: Če se v brskalniku namesto spletne strani prikaže seznam map in datotek, ki se stavljajo našo spletno stran, zaprite Interne Explorer in se vrnite v Visual Studio. V Solution Explorerju desno kliknite obrazec Kadri.aspx in nato opcijo Set As Start Page. Nato ponovno zaženite spletno aplikacijo.

Kontrola vnosa potakov na spletnih obrazcih

Tako kot pri Windows okenskih aplikacijah, je tudi pri spletnih aplikacijah kontrola uporabnikovih vnosov zelo pomemben del projekta. V običajnih Windows okenskih aplikacijah smo kontrole uporabnikovih vnoso realizirali s pomočjo dogodkov, ki smo jih pripeli na določene gradnike in obrazce znotraj aplikacije. Pri spletnih obrazcih, pa moramo razmisliti o naslednjem: ali naj izvedemo kontrolo vnosa na uporabnikovi (client) strani, torej v brskalniku, ali pa naj kontrolo izvajamo na strežniški strani.

Primerjava kontrol vnosa na strežniški strani in na strani uporabnika

Odprimo ponovno našo spletno aplikacijo *Kadri*. Uporabnik mora v vnosna polja na spletni strani vnesti številko delavca, ki mora biti neko celo število, nato pa še ime in priimek tega delavca, ki pa je lahko poljuben string. V Windows okenski aplikaciji, bi za kontrolo vnosa kreirali dogodek *Validating*, v katerem bi se prepričali, da je uporabnik za številko delavca res vnesel numeričen podatek, ter da vsebini polj Ime in Priimek nista prazni. Spletni obrazci pa nimajo dogodka *Validating*, kar pomeni da pri spletnem programiranju ne moremo uporabiti enakega pristopa.

Kontrola (validacija) vnosa na strani strežnika

Če pogledamo npr. kontrolo (gradnik) *TextBox*, bomo v oknu *Properties* med dogodki opazili tudi dogodek *TextChanged*. Ta se zgodi vsakič, ko se obrazec pošlje nazaj na strežnik, potem ko je uporabnik spremenil vneseni tekst v tem gradniku. Tako kot dogodki spletnega strežnika, se tudi dogodek *TextChanged* izvede na spletnem strežniku. Celotna operacija vključuje pošiljanje podatkov s spletnega brskalnika na spletni strežnik, procesiranje (obdelava) validacijskega dogodka na strežniku za kontrolo vnesenih podatkov, nato pa oblikovanje validacijskih sporočil o napakah v HTML odgovor in pošiljanje nazaj k klientu. Tak način kontrole je s stališča programiranja sprejemljiv le v primeru, če je validacija, ki naj se izvede kompleksna, ali pa zahteva obdelavo, ki se lahko izvede le na spletnem strežniku (npr. ugotavljanje, ali številka zaposlenega, ki jo je vnesel uporabnik, že obstaja v bazi podatkov). V primeru, da pa hočemo le preverjati vnesene podatke v enem samem gradniku TextBox (npr. ali je uporabnik vnesel celo število in ne nekega stringa), potem je način tak validacije nesprejemljiv, saj po nepotrebnem obremenjujemo spletni strežnik. Aplikacijo moramo napisati tako, da se take kontrole oz. preverjanja izvajajo kar v spletnem brskalniku, torej na klientovi strani.

Kontrola (validacija) vnosa na klientovi strani

Za kontrolo vnosa podatkov v spletne obrazce na uporabnikovi strani, omogoča razvojno okolje kontrolo vnesenih podatkov preko uporabe t.i. potrditvenih (*validation*) kontrol. Če uporabnik uporablja brskalnik kot je npr. Microsoft Interner Explorer 4 ali pa še kasnejšo verzijo, ki podpira dinamični HTML, te kontrole generirajo JavaScript kodo, ki teče v brskalniku in tako ni potrebna obdelava na strežniku. V kolikor pa je na uporabnikovem računalniku kakšen od starejših brskalnikov, pa validaciske kontrole generirajo kodo, ki se bo izvedla na strežniški strani. Bistvo je v tem, da ko razvijalcu, ki kreira neko spletno stran, ni potrebno skrbeti kje se bodo te kontrole izvajale – zaznavanje brskalnika in posebnosti pri generiranju ustrezne kode so že vgrajene v ustrezne validacijske kontrole. Razvijalec mora tako kontrolo le postaviti na spletni obrazec, nastaviti njene lastnosti z uporabo okna Propertites Window, ali pa s pisanjem kode. Potrebno je še določiti ustrezna validacijska pravila in pa sporočila o napakah, ki naj se prikažejo v primeru napak.

V ASP.NET obstaja pet tipov validacijskih kontrol:

- *RequiredFieldValidator*: ta kontrola zagotovlja, da bo uporabnik sploh vnesel podatek v kontrolo.
- *CompareValidator*: kontrola se uporablja za primerjanje vnesenega podatka z neko konstantno vrednostjo, vrednostjo lastnosti neke druge kontrole, ali pa z vrednostjo pridobljeno iz baze podatkov.
- *RangeValidator*: kontrola se uporablja za ugotavljanje, ali je podatek, ki ga je vnesel uporabnik, znotraj ali pa zunaj nekih vnaprej določenih meja.
- *RegularExpressionValidator*: s to kontrolo ugotavljamo, ali uporabnikov vnos ustreza nekemu izrazu, formuli, vzorcu ali nekemu formatu (npr. telefonski številki, ali pa poštni številki).
- *CustomValidator*: s pomočjo te kontrole pa lahko definiramo lastno kontrolno logiko in jo pripnemo k kontroli, za katero smo jo definirali.

Čeprav vsaka od teh kontrol izvaja svojo lastno, zgoraj opisano validacijsko logiko, v praksi velikokrat uporabljamo njihove kombinacije. Če hočemo npr. doseči, da bo uporabnik zagotovo vnesel neko vsebino v *TextBox* in da bo vnesena vrednost znotraj nekih meja, bomo temu *TextBox*-u pripeli tako *RequiredFieldValidator*, kot tudi *RangeValidator*.

Vse te kontrole pa lahko delajo v povezavi z kontrolo *ValidationSummary*, ki se uporablja za prikaz sporočil o napakah.

Implementacija kontrole vnosa (validacija) na strani klienta (v spletnem brskalniku)

V spletni aplikaciji *Kadri* bomo kontrolo vnosa tipa *RequiredFieldValidator* zahtevali za kontroli *tbID*, *tbIme* in *tbPriimek*. Vnesena vrednost v polje *tbID* mora poleg tega biti numerična in še pozitivno celo število. S pomočjo validatorja *RangeValidator* bomo določili, da mora biti celoštevilčna vrednost za številko zaposlenega med 1 in 5000.

1. V Solution Explorerju desno kliknimo na datoteko *Kadri.aspx*, nato pa opcijo *Set As Start Page*.

2. Ponovno desno kliknimo datoteko *Kadri.aspx*, nato kliknimo na *View Designer*. V Oknu *Design View* se prikaže spletni obrazec *Kadri.aspx*.

3. V oknu *Toolbox* razširimo skupino gradnikov *Validation* in na obrazec postavimo kontrolo *RequiredFieldValidator*. Kontrolo nato na obrazcu premaknimo pod vnosno polje rbIme (za premik mora biti seveda omogočena lastnost absolute positioning).

4. Označimo kontolo *RequiredFieldValidator* (ime kontrole je *RequiredFieldValidator1*), nato pa v oknu *Properties* nastavimo lastnost *ControlToValidator* na *tbIme*. V lastnost *ErrorMessage* vtipkajmo tekst **Vnesti morate ime zaposlenega**. To sporočilo se bo prikazalo v primeru, da bo uporabnik pustil polje tbIme prazno. Po vnosu tega teksta, se bo vneseni tekst prikazal tudi na obrazcu.

5. Dodajmo še dve validacijski kontroli tipa *RequiredFieldValidator* in ju postavimo pod kontrolo *rbPriimek* in pod kontrolo *rbID*. Za obe kontroli nastavimo lastnosti *ControlToValidate* (za prvo kontrolo nastavimo lastnost za *tbPriimek*, za drugo pa za *tbIme*). Za prvo kontrolo vtipkajmo za lastnost *ErrorMessage* tekst **Vnesti morate priimek zaposlenega**, za drugo kontrolo pa tekst **Vnesti morate številko zaposlenega**.

6. V meniju *Debug* kliknimo opcijo *Start Without Debugging*, da poženemo naš spletni obrazec in stestiramo njegovo delovanje.

7. Pri prvem prikazu spletnega obrazca v Microsoft Internet Explorerrju, bo vsebina vseh *TextBox*-ov seveda prazna. Če takoj kliknemu gumb *bShrani*, se bodo pod kontrolami prikazala vsa tri sporočila tipa *RequireFieldValidator*. Njihova vsebina bo seveda takšna, kot smo jo napisali v oknu *Properties*.
   Ob tem je zelo pomembno, da se zaradi validacijskih sporočil dogodek *Click*, ki je vezan na gumb *Shrani* sploh ni zgodil, zaradi česar se seveda tudi ne pokaže spremenjena vsebina labele *lInfo*. Validacijske kontrole zaradi napak pri vnosih namreč preprečijo pošiljanje sporočil strežniku in oblikujejo kodo ki se izvede kar v spletnem brskalniku na strani klienta. Pošiljanje podatkov na strežnik bo onemogočeno toliko časa, dokler ne bodo na strani klienta odpravljene vse napake v vnosih podatkov.

8. Vtipkajmo sedaj neko ime v kontrolo *tbIme* (polje *Ime*). Takoj ko se bomo premaknili stran od te kontrole, bo sporočilo o napaki pod to kontrolo izginilo. Če pa vsebino kontrole *tbIme* pobrišemo in se ponovno premaknemo na neko drugo kontrolo, se sporočilo o napaki pojavi ponovno. Vsa ta funkcionalnost se seveda dogaja pri klientu, brez kakršnegakoli pošiljanja podatkov preko omrežja na strežnik.

9. Vnesimo sedaj še neko vrednost v polje *Priimek* in polje *Številka delavca* in nato kliknimo gumb *Shrani*. Dogodek *Click* tega gumba se sedaj izvede, na labeli *lInfo* pa se prikaže ustrezno sporočilo o vnesenih podatkih.

10. Dodajmo še kontrolo *RangeValidator*, s katero bomo ustrezno preverjali uporabnikov vnos v polje *tbID*. V oknu Toolbox izberimo gradnik *RangeValidator*, ga postavimo na obrazec in nato premaknimo pod kontrolo *tbID* (verjetno bo za ta namen potrebno malo razmakniti vnosna polja). V oknu Properties nato nastavimo še lastnost *ControlToValidate* na *tbID*, lastnost *ErorMessage* na **Številka delavca mora biti med 1 in 5000**, lastnost *MaximumValue* nastavimo na 5000, *MinimumValue* na 1, lastnost *Type* pa na *Integer*. Če ponovno zaženemo našo spletno aplikacijo in v polje *Številka delavca* vnesemo vrednost, ki ni celo število med 1 in 5000, bomo pod kontrolo dobili ustrezno obvestilo.

11. V Internet Explorerju si oglejmo kodo, ki jo je zgeneriral brskalnik (View Source). Koda namenjena validacijskim kontrolam je JavaScript koda, nahaja pa se bolj pri dnu te datoteke. Ta koda je bila zgenerirana kot rezultat nastavitev lastnosti validacijskih kontrol, ii smo jih postavili na obrazec.

Kontrola *ValidationSummray*

V prejšnjem primeru smo pokazali, kako lahko s pomočjo validacijskih kontrol preverjamo pravilnost uporabnikovih vnosov. Tak način preverjanja je sicer povsem legalen, saj se preverjanje se izvaja na klientovi strani, ni pa najlepši, saj je prikaz sporočil o napaki dokaj neposrečen. V naslednji vaji bo zato prikazana uporaba kontrole *ValidationSummray*, s pomočjo katere bo prikazan drugačen način sporočanja napak.

1. V spletni aplikaciji *Kadri* izberimo kontrolo *RequiredFieldValidator1*, ki se nahaja pod pljem za vnos imena delavca in v oknu Properties lastnost Text spremenimo v *. Sporočilo, ki se sedaj prikaže v primeru, da uporabnik pusti polje prazno, se spremeniv zvezdico (*). Kontrolo *RequiredFieldValidator1* nato še premaknimo na desno stran polja za vnos imena.

2. Enako storimo še s kontrolama *RequiredFieldValidator2* in *RequiredFieldValidator3* – lastnost Text spremenimo v *, kontroli pa premaknimo za polje *Priimek* oz. za polje *Številka delavca*. Nato storimo enako še za kontrolo *RangeValidator1*.

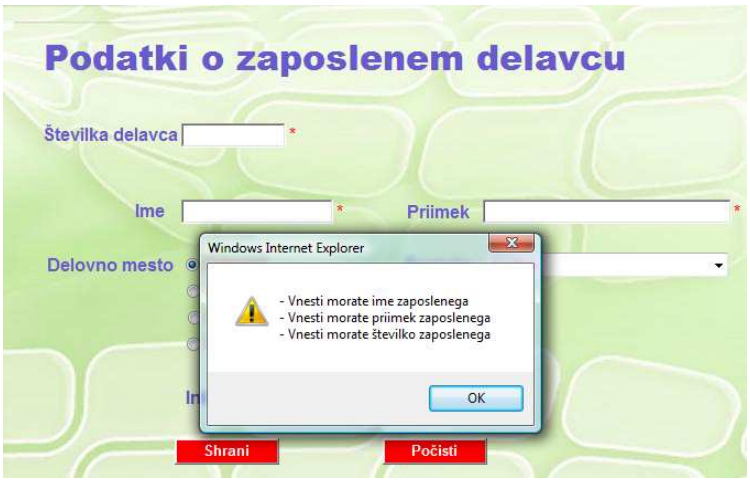
3. V oknu *Toolbox* izberimo kontrolo *ValidationSummary* in jo postavimo na obrazec desno od radijskih gumbov. Kontrola *ValidationSummary* bo prikazala sporočila o napakah za vse validacijske kontrole na obrazcu.

4. Nastavitve kontrole *ValidationSummary* pustimo zaenkrat kar nespremenjene, prepričajmo se le, da je lastnost *ShowSummary* nastavljena na *True*.

5. Zaženimo naš spletni obrazec, nato ne da bi karkoli vnašali takoj kliknimo gumb *Shrani*. Ob vsakem tekstovnem oknu se pojavi zvezdica, v kontroli *SummaryValidation* pa se pojavijo ustrezna sporočila o

napakah. Vnesimo ime in priimek, za številko delavca pa vnesimo **AAA**. Ko se premikamo od enega polja do drugega zvezdice za vnosnimi polji izginjajo, ostane le zvezdica ob polju za vnos številke delavca, v validacijski kontroli pa je prikazano sporočilo o napačnem vnosu številke delavca. Ko za številko delavca vnesemo npr. 101 in kliknemo gumb *Shrani*, izginejo vas spročila o napakah.

Če uporabnikov brskalnik podpira dinamični HTML, lahko sporočila o napakah namesto na obrazcu prikažemo v sporočilnem oknu. V tem primeru gradniku *ValidationSummary* nastavimo lastnost *ShowMessageBox* na *True*, lastnost *ShowSummary* pa na *False*.



Onemogočanje validacije podatkov na strani uporabnika

V prejšnji vaji smo videli, da je bila kontrola vnosa podatkov na strani uporabnika realizirana s pomočjo kode v jeziku JavaScrip v brskalniku. ASP.NET generira to kodo avtomatično, odvisno od sposobnosti uporabnikovega brskalnika. Če brskalnik ne podpira jezika JavaScript, se bodo vse validacijske kontrole izvedle s pomočjo kode, ki se bo izvedla na spletnem strežniku.

Razvijalci spletne aplikacije pa imamo možnost aplikacijo razviti tako, da se vse kontrole vnosov avtomatično izvajajo na strežniku in v nobenem primeru na klientovi strani. To storimo tako, da nastavimo lastnost *EnableClientScript* za vsako validacijsko kontrolo na *false*. Tak način kontrole vnosa podatkov je včasih celo priporočljiv, npr. v primeru ko v projekt vključimo t.i. *CustomValidation* kontrole, ali pa pri bolj kompleksnih kontrolah vnosa, oz. takrat, ko kontrola vnosa vključuje dostop do podatkov, ki se nahajajo na strežniku. Obstaja pa tudi posebna kontrola imenovana *CustomControl*, ki ima med svojimi dogodki tudi dogodek *ServerValidate*, ki ga lahko uporabimo za eksplicitno realizacijo kontrole vnosa na strežniški strani, pa čeprav je lastnost *EnableClientScript* nastavljena na *True*.

Zagotavljanje varnosti spletne strani in dostop do podatkov s pomočjo spletnih obrazcev

Pri razvoju aplikacij, ki so dostopne preko Interneta, je zelo pomembno, da so ustrezno zavarovane oz. da preverjajo uporabnikovo identiteto. To velja še posebno tedaj, kadar aplikacije delajo z bazami podatkov, saj v tem primeru anonimnim uporabnikom navadno ne dovolimo dostopa.

Uporaba gradnika GridView na spletnih obrazcih

V spletnih obrazcih uporabljamo za prikaz podatkov tabele iz podatkovne baze gradnik *GridView*. Ta je zelo podoben gradniku *DataGridView* v okenskih aplikacijah, ima pa nekatere posebnosti, saj ga je Microsoft načrtoval za uporabo v ASP.NET okolju. Namen gradnika pa je enak – prikaz in ažuriranje podatkov pridobljenih iz nekega podatkovnega vira. Glavna sprememba je pri pridobivanju in prikazu večje količine podatkov. V spletnih aplikacijah so uporabniki pogosto zelo oddaljeni od strežnika, na katerem je baza podatkov, zaradi tega je nujno da omrežje uporabljamo kar se da smotrno. V spletnih aplikacijah je nujno k uporabniku pretakati le tiste podatke, ki jih želi in nič drugega. Spletni gradnik *GridView* podpira lastnost *paging*, ki omogoča prodobivanje nove strani podatkov s strežnika, ko se uporabnik premika naprej oz. nazaj po vsebini *DataSet*-a.

Tako kot okenska kontrola *DataGridView*, je tudi spletna kontrola *GridView* mačrtovana za uporabo medtem ko je povezava z bazo prekinjena. Za priklop na bazo lahko kreiramo objekt tipa *SqlDataSource*, napolnimo ustrezen *Dataset* s podatki, nato pa povezavo z bazo prekinemo. *DataSet* kontrole *SqlDataSource* nato lahko povežemo z gradnikom *GridView*. Za razliko od okenske kontrole DataGridView, pa je informacija v spletniem gradniku *GridView* predstavljena v *Read-Only* obliki (prodobljena kot HTML tabela v brskalniku). Seveda pa lastnosti spletnega gradnika *GridView* uporabniku omogočajo tudi urejevanje podatkov, ki izbrano vrstico podatkovne tabele spremeni v množico vnosnih polj, uporabnik pa ta polja lahko uporablja za spreminjanje predstavljenih podatkov.

Zagotavljanje varnosti spletnih aplikacij

Aplikacije razvite z uporabo Microsoft ASP.NET razvojnega okolja imajo številne mehanizme ki zagotavljajo, da imajo uporabniki, ki te aplikacije uporabljajo, ustrezne pravice dostopa. Nekatere od teh tehnologij se zanašajo na avtentikacijo uporabnikov, ki bazira na nekem obrazcu za identifikacijo in vnosom gelsa, druge pa bazirajo na varnostnih posebnostih operacijskega sistema Windows. Če kreiramo spletno aplikacijo, do katere bodo uporabniki dostopali preko Interneta, potem uporaba varnostnega sistema operacijskega sistema Windows ni prava izbira, saj uporabniki pogosto uporabljajo tudi druge operacijske sisteme. Najboljše zagotavljanje varnosti je tako s pomočjo posebnega prijavnega spletnega obrazca.

Razumevanje varnosti preko spletnih obrazcev

Varnostna zaščita spletne aplikacije s pomočjo obrazcev omogoča identifikacijo uporabnika preko prijavnega obrazca, ki uporabniku ponudi vnos njegovega ID-ja in gesla. Po uspešni avtentikaciji uporabnika, lahko le-ta dostopa do različnih spletnih strani, pri čemer se pred prikazom vsake strani preverja, ali ima trenutni uporabnik res pravico do prikaza te strani. Posameznim uporabnikom namreč lahko omejimo prikaz določenih spletnih strani.

Za realizacijo ASP.NET varnosti, ki temelji na prijavnem obrazcu, moramo narediti nekatere spremembe v konfiguracijski datoteki spletne strani (datoteka *Web.config*), obenem pa moramo seveda pripraviti ustrezen obrazec za avtentikacijo uporabnika. Varnostni obrazec se bo prikazal vsakič, ko bo nek uporabnik skušal dostopiti do katerekoli strani v aplikaciji, v kolikor se uporabnik seveda še ni uspešno logiral. Uporabnik bo lahko nadaljeval z delom v spletnem obracu le ob uspešni prijavi. Zgodi se namreč lahko, da bo nekdo v našo spletno stran želel

vstopiti ne preko začetnega obrazca, ampak preko nekega podrejenega obrazca, da bi se na ta način izognil prijavnemu obrazcu.

Mehanizem ASP.NET ki temelji na prijavnih obrazcih je zelo robusten, in če predvidevamo, da je spletni strežnik varen, potem naj bi bil ta mehanizem enak za večino naših aplikacij.

Implementacija zaščite preko spletnega obrazca

Spletno aplikacijo lahko pred zlorabami zavarujemo tako, da ima uporabnik dostop do je le preko spletnega obrazca za avtentikacijo.

1. Kreirajmo nov spletni projekt in ga poimenujmo *Northwind*. V Solution Explorerju preimenujmo datoteko *Default.asp*x v *Potrosnik.aspx*.

2. Desno kliknimo datoteko *Potrosnik.aspx* in kliknimo na *Set As Start Page*.

3. V oknu *Source View*, ki prikazuje HTML kodo spletne strani, kliknimo (spodaj levo) jeziček *Design*.

4. V meniju *Layout* izberimo opcijo *Position* in kliknimo *Auto-position Options*. V pogovornem oknu nato preverimo, da je kljukica v »*Change positioning to the folowing for controls added using the Toolbox, paste, or drag and drop*«, in se prepričajmo da je v spustnem seznamu .izbrana opcija *Absolutely positioned*. Nato kliknimo gumb OK.

5. V oknu *Toolbox* izberimo gradnik *Label* in ga postavimo na obrazec. Labelo nato premaknimo na sredino obrazca in ji v oknu properties začasno zapišimo lastnost *Text*: **Ta obrazec bo implementiran kasneje**.

V naslednjih vajah bomo skreirali prijavni obrazec za avtentikacijo uporabnika in konfigurirali varnost spletne aplikacije preko tega prijavnega obrazca. Prijavni obrazec se bo prikazal vselej, ko bo nek uporabnik, ki še ni bil verificiran, skušal dostopiti do aplikacije. Kadar bomo varnost naše aplikacije zagotavljali preko prijavnega obrazca, bo ASP.NET vsak poskus dostopa do aplikacije, ki bo prišel s strani uporabnika, ki še ni šel skozi postopek avtentikacije, preusmerjen na prijavni obrazec.

Implementacija prijavnega obrazca pri kreiranju zaščite aplikacije, ki temelji na obrazcih, je tako pogosta operacija, da je Microsoft za poenostavitev tega postopka pripravil množico prijavnih kontrol. Naslednji primer prikazuje uporabo ene od teh kontrol.

1. V meniju *Website* kliknimo na opcijo *Add New Item*. Pojavi se pogovorno okno, v katerem izbremo opcijo *Web Form*, za ime obrazca pa vtipkajmo *LoginForm*. Preverimo še, da je izbrani jezik Visual C#, da je kljukica v okencu *Place Code in separate file* in je okence *Select master Page* prazno. Končno kliknimo gumb *Add* za kreiranje obrazca.

   Odpre se nov spletni obrazec, na ekranu pa je v oknu *Source View* prikazana njegova HTML koda.

2. Klinimo jeziček *Design* za prikaz datoteke *LoginForm.aspx* v oknu *Design View*.

3. V oknu Toolbox razširimo skupino gradnikov *Login* in na spletni obrazec postavimo kontrolo *Login*. Kliknimo kamorkoli na obrazec, da se na ta način skrije prikazani *Login Tasks* meni.

   Kontrola *Login* je sestavljena kontrola, ki vsebuje več label, dve vnosni polji, v kateri uporabnik vnaša uporabniško ime geslo, ter gumb za potrditev prijave. Večino od teh kantrol lahko s omočjo okna *Properties* nastavimo po svoje, prav tako pa laho spremenimo stil same kontrole.

4. Premaknimo kontrolo *Login* na sredino obrazca, nato pa kliknimo na gumb s trikotnikom v zgornjem desnem robu kontrole. V meniju, ki se prikaže, kliknimo *Auto Format*. Prikaže se pogovorno okno *Auto Format*, v katerem lahko po svojem okusu izberemo izgled in občutek *Login* kontrole.

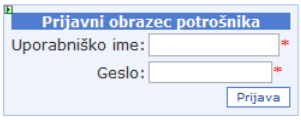
5. V pogovornem oknu *Auto Format* na levi strani kliknimo opcijo *Classic* in nato gumb OK. *Login Tasks* menu nato zaprimo.

6. V oknu Properties spremenimo nekatere lastnosti kontrole *Login*, tako kot to prikazuje naslednja tabela

Lastnost	Vrednost
DisplayRememeberMe	false
FailureText	Nepravilno uporabniško ime (User Name) ali pa geslo. Vnesite prosim veljavno uporabniško ime in geslo.
TitleText	Prijavni obrazec potrošnika
DestinationPageUrl	Klin na tropičje in nati izberimo Potrosnik.aspx. Vseina polja bo tako **~/Potrosnik.aspx**
LoginButtonText	Prijava
PaswordLabelText	Geslo:
PasswordRequiredErrorMessage	Potreben je vnos gesla
RememeberMeNextTime	Zapomni si geslo
UserNameLabelText	Uporabniško ime:
UserNameRequiredErrorMessage	Uporabniško ime je obvezno

Lastnost *DestinationPageUrl* določa spletno stran, na katero bo preusmerjen uporabnik ob uspešni prijavi. Predpona »~/« označuje, da je spletna stran, na katero bo ob uspešni prijavi preusmerjen uporabnik, kar matična mapa spletne strani in ne katera druga podrejena mapa. Če bo prijava neuspešna, se bo prikazalo sporočilo, ki smo ga določili z lastnostjo *FailureText*, uporabnik pa bo imel na voljo ponovni poizkus prijave.

Login kontrola naj bi sedaj izgledala takole:



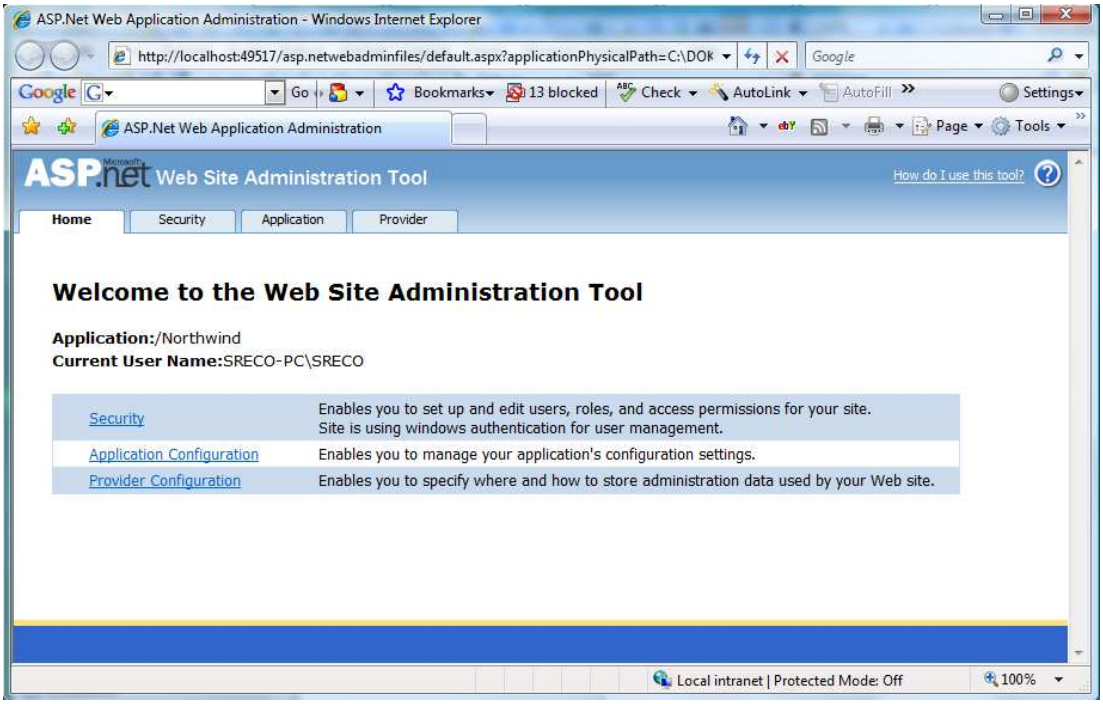
Ko uporabnik klikne gumb *Prijava*, se mora izvesti njegova avtentikacija. Če sta vnesena pravilna uporabniško ime in geslo, mora aplikacija uporabniku dovoliti vstop v spletni obrazec *Potrosnik*, sicer pa se morata prikazati sporočilo zapisano v lastnosti FailureText. Za realizacijo opisanih zahtev imamo na voljo vsaj dve možnosti:

- Napišemo kodo, ki obdela dogodek *Autenticate* naše *Login* kontrole. Ta dogodek se izvede vselej, ko uporabnik klikne na gumb *Prijava*. V kodi, ki jo za ta namen napišemo, moramo preveriti vneseno uporabniško ime in pa geslo in če sta vnosa pravilna, uporabniku omogočimo nadaljevanje na tisti spletni strani, ki smo jo določili z lastnostjo *DestinationPageUrl*. Opisani način je enostaven, a zahteva da najprej kreiramo nek seznam uporabniških imen in gesel, v katerem bomo preverjali uporabnikove vnose.

- Uporabimo v Visual Studio vgrajeno zmožnost, da z uporabniškimi imeni in gesli upravlja orodje ASP.NET *Administration Tool*. V tem primeru prepustimo kontroli *Login*, da sama opravi z obdelavo preverjanja uporabnikovih vnosov, ko le-ta pritisne gumb *Prijava*. ASP.NET *Web Site Administration Tool* vzdržuje svojo lastno bazo uporabniških imen in gesel, ter seveda vključuje tudi ustreznega čarovnika, ki omogoča dodaanje novih uporabnikov.

V naslednji vaji bomo za potrebe avtentikacije uporabili drugo varianto, saj je prvi način glede na opisan postopek povsem enostaven.

1. Odprimo *Website* meni in kliknimo *ASP.NET Configuration*. Zagnal se bo *ASP.NET Development Server* in ta bo zagnal spletno aplikacijo imenovano *ASP.NET WEB Site Administration Tool*, ki uporablja svojo lastno instanco *ASP.NET Development Server*-ja, neodvisno od naše spletne aplikacije. Zažene se Internet Explorer, v njem pa se prikaže *ASP.NET Web Site Administration Tool*.



   Orodje nam omogoča dodajanje in upravljanje z uporabniki naše spletne strani, nastavljamo lahko razne nastavitve, ki bodo shranjene v konfiguracijski datoteki naše spletne aplikacije, določimo pa lahko tudi kako bodo informacije o varnosti shranjene (npr. uporabniška imena in gesla). Privzeti način je, da *ASP.NET Web Site Configuration Tool* shrani informacije o varnosti v SQL Server bazo podatkov z imenom *ASPNETDB.MDF*, ki jo skreira v mapi *App_Data* naše spletne strani, pri tem pa uporabi *AspNetSqlProvider*.

2. Kliknimo zavihek *Security*, da se prikaže stran *Security*. Stran se uporablja za upravljanje z uporabniki, določanje mehanizma za avtentikacijo, ki jo uporablja spletna stran, definiranje uporabniških funkcij (prikladni mehanizem za nastavljanje pravic uporabniškim skupinam) in za specifikacijo dostopnih funkcij za kontrolo dostopa do spletne strani.

   - POZOR: Če razvijalec spletne aplikacije nima vseh potrebnih pravic do mape, v kateri dela spletno aplikacijo, se lahko zgodi, da se ob kliku na gumb *Security* pokaže obvestilo, da nima ustreznih pravic za generiranje novih uporabnikov. V takem primeru je potrebno poskrbeti za ustrezne pravice, za potrebe vaje pa je včasih dovolj, da celotno mapo s spletnim projektom premaknemo kam drugam!

3. V sekciji *Users* kliknimo na link *Select authentication type*. Odpre se nova stran, kjer moramo določiti, kako bodo uporabniki dostopali do naše spletne strani. Na voljo sta dve opciji: *From the Internet* in *From a local network*. Privzeta opcija je *From a local network*, s katero za našo spletno stran določimo, da bo uporabljala kar Windows avtentikacijo. V tem primeru morajo biti vsi uporabniki člani Windows domene, do katere ima naša spletna stran dostop. Ker pa bo naša spletna stran dostopna preko interneta, bomo izbrali rajši prvo opcijo, to je *From the Internet*.

4. Izberimo torej opcijo *From the Internet*. Za našo aplikacijo na ta način določimo, da bo uporabljala zaščito, ki temelji na obrazcih. Za vstop v spletno aplikacijo bomo uporabili prijavni obrazec, ki smo ga naredili v prejšnji vaji. Odločitev potrdimo s klikom na gumb *Done*.

5. V sekciji *Users* je trenutno število obstoječih uporabnikov, ki lahko dostopijo do naše spletne strani, enako 0. Kliknimo na bližnjico *Create User*.

6. Na strani Create User dodajmo novega uporabnika. Vnesimo npr. naslednje vrenosti:

Polje	Vnesena vrednost vrednost
Use Name	John
Password	Pa$$w9rd
Confirm Password	Pa$$w9rd
E-mail	john@guest.arnes.si
Security Question	Kakšno je bilo ime tvoje prve domače živali
Security Answer	Thomas

Vnesti **moramo** vse podatke. Podatki *E-mail*, *Security Question* in *Security Answer* so potrebni za kontrolo *PasswordRecovery* za primer resetiranja, oziroma za potrebe nadomestitve uporabniškega gesla (npr. v primeru, da je le-ta geslo pozabil). Kontrola *PasswordRecovery* se nahaja v oknu Toolbox v skupini gradnikov *Login* in jo lahko dodamo na naš prijavni obrazec, ter nastavimo ustrezne lastnosti za potrebe resetiranja oz nadomestitve gesla.

7. Prepričajmo se, da je kljukica v oknu *Active User Box* in kliknimo *Create User*. Pokaže se sporočilo »*Complete. Your account has been successfully created.*«

8. Kliknimo *Continue*. Ponovno se prikaže stran *Create User*, ki nam omogoča dodajanje novih uporabnikov. Kliknimo *Back*, da se vrnemo na stran *Security*. Število obstoječih uporabnikov je sedaj enako 1.

   - Opomba: Opcijo *Manage Links* na tej strani lahko uporabimo za spremembo naslova elektronske pošte, za vnos opisa tega uporabnika in za brisanje uporabnika. Določimo pa lahko tudi, da uporabnik svoje geslo spremeni, oziroma ga nadomesti z novim, v primeru da ga je pozabil (na prijavni obrazec je v tem primeru potrebno dodati še gradnika *ChangePassword* in *PasswordRecovery*).

9. V sekciji *Access Rules* kliknimo »*Create access rules*«. Prikaže se stran *Add New Access Rules*, kjer lahko za vsakega uporabnika posebej določimo, do katerih strani naše spletne aplikacije bo imel dostop.

   Prepričajmo se, da je povsem na levi strani okna (opcija *Select a directory for this rule*) izbrana mapa *Podjetje*. Pri opciji *Rule applies to* izberimo *user* in vtipkajmo *John*. Za *Permission* izberimo *Allow* in stran zaprimo s klikom na *OK*. Napisali smo pravilo oz. predpis, ki bo uporabniku, ki se bo prijavil kot *John* omogočil dostop do naše spletne strani.

10. V sekciji *Access Rules* ponovno kliknimo na »*Create access rules*«. Na strani *Add New Access Rule* se najprej prepričajmo, da je v »*Select a directory for this rule*« izbrana mapa *Podjetje*, nato pa pod »*Rule applies to*« kliknimo *Anonimous Users*. Prepričajmo se še, da je za »*Permission*« izbrana opcija *Deny*, nato pa kliknimo *OK*.

    Pravilo, ki smo ga napisali zagotavlja, da uporabniki, ki se ne bodo prijavili, do naše spletne strani, do nje ne bodo imeli dostopa. Na ekranu se ponovno prikaže stran *Security*.

11. Zaprimo Internet Explorer, ki prikazuje stran *ASP.NET Web Site Administration Tool* in se tako vrnemo v Visual Studio.

12. V Solution Explorerju kliknimo gumb *Refresh*. V mapi *App_Data* se prikaže datoteka ASP-NETDB.MDF (baza podatkov s shranjenimi uporabnikovimi nastavitvami), na dnu Solution Explorerja pa se prikaže še datoteka *Web.config*. Dvokliknimo na datoteko *Web.config* in na ekranu se prikaže njena vsebina. Vsebino je skreiralo orodje *ASP.NET NET Web Site Administration Tool*, izgleda pa takole:

```
<?xml version="1.0"?>
<configuration>
       <system.web>
              <authorization>
                     <allow users="John"/>
                     <deny users="?"/>
              </authorization>
             <authentication mode="Forms" />
       </system.web>
</configuration>
```

Element `<authorization>` določa uporabnike, ki imajo oz. nimajo dostopa do naše spletne strani. (»?« označuje anonimnega uporabnika). Atribut *mode* elementa `<authentication>` določa, da spletna stran uporablja avtentikacijo, ki temelji na obrazcih.

13. Spremenimo element `<authentication>` in dodajmo element `<forms>` takole (ne pozabimo dodati tudi elementa `</authentication>`!)

```
<authentication mode="Forms">
      <forms loginUrl="LoginForm.aspx" timeout="5"
             cookieless="AutoDetect" protection="All"/>
    </authentication>
```

Element `<forms>` ima vlogo konfiguracije parametrov za avtentikacijo, ki temelji na obrazcih. Atributi, ki jih določimo v tej sekciji določajo, da v kolikor poskuša do spletne strani dostopiti neavtenticiran uporabnik, bo le-ta preusmerjen na prijavno stran, v našem primeru *LoginForm.aspx*. Če je uporabnik neaktiven več kot 5 minut, se bo moral pri dostopu do naše spletne strani ponovno prijaviti. V številnih spletnih straneh, ki uporabljajo avtentikacijo temelječo na obrazcih, so uporabnikove informacije shranjene v piškotku (cookie) v njegovem računalniku. Vendar pa številni spletni brskalniki omogočajo, da uporabnik lahko določi, da piškotkov ne bo uporabljal (ker so lahko piškotki zlorabljeni preko spletnih strani, ki želijo škodovati našemu računalniku). Za take primere lahko določimo lastnost *cookieless= »AutoDetect«*. S to lstnostjo smo določili, da v kolikor na uporabnikovem računalniku piškotki niso onemogočeni, jih bo naša spletna aplikacija uporabila, sicer pa se bo informacija o uporabniku posredovala nazaj in naprej med spletno stranjo in uporabnikovim računalnikom kot del vsake zahteve.

Informacija o uporabniku vsebuje uporabniško ime in geslo, ki pa morata biti seveda ostalim uporabnikom omrežja nedostopna. Zaradi tega se za enkripcijo teh podatkov uporablja atribut *protection*, ki je bil uporabljen tudi v našem primeru.

14. V meniju *Debug* kliknimo *Start Without Debugging*. Odpre se Internet Explorer. Začetna stran naše aplikacije je sicer stran *Potrosnik.aspx*, ker pa se na to stran še nismo prijavili, smo preusmerjeni na stran *LoginForm* – potrebna je predhodna uspešna prijava.

15. V prijavno stran vtipkajmo naključno uporabniško ime in geslo in kliknimo *Log In*. Ponovno se prikaže prijavna stran, a tokrat z obvestilom »*Nepravilno uporabniško ime (User Name) ali pa geslo. Vnesite prosim veljavno uporabniško ime in geslo.*«.

16. Vtipkajmo sedaj pravilno uporabniško ime *John* in pravilno geslo **Pa$$w9rd** in kliknimo *Log In*. Prikaže se nam spletna stran *Potrosnik.aspx* in v njej sporočilo »*Ta obrazec bo nadgrajen kasneje*«.

Podatkovne poizvedbe v spletnih aplikacijah

Potem ko smo v spletni aplikaciji določili kontrole dostopa, se lahko osredotočimo na poizvedbe podatkov iz podatkovnih baz in na manipulacije s temi podatki.

Prikaz podatkov poljubne tabele iz podatkovne baze

V naslednji vaji bomo v spletnem obrazcu prikazali vse podatke iz tabele *Customer* podatkovne baze *Northwind*.

1. Ponovno odprimo spletni projekt *Podjetje*, ki smo ga zgradili v prejšnjem poglavju. Na obrazcu *Potrosnik.aspx* pobrišimo labelo z napisom »*Ta obrazec bo nadgrajen kasneje*«.

2. V oknu Toolbox razširimo skupino gradnikov *Data*. Na spletni obrazec postavimo kontrolo *SqlDataSource*.

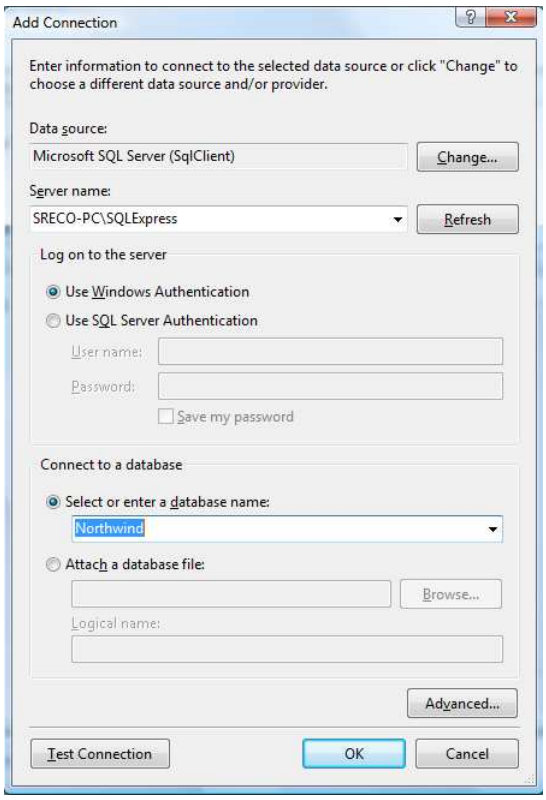
3. Kliknimo kamorkoli na obrazec, da se skrije prikazani meni kontrole *SqlDataSource*. Obrazcu smo tako dodali kontrolo *SqlDataSource1*. Kontrola *SqlDataSource1* je kontrola spletnega strežnika, ki ima enako vlogo kot podatkovni izvor v Windows aplikacijah.

   ❖ Opomba: Čeprav je kontrola *SqlDataSource* v fazi razvoja aplikacije vidna, se bo po zagonu aplikacije skrila in torej ne bo vidna.

4. S pomočjo okna *Properties*, spremenimo lastnost *ID* tega gradnika v *PotrosnikInfoSource*.

5. Na spletnem obrazcu *Potrosnik.aspx* sedaj označimo (kliknimo) kontrolo *PotrosnikInfoSource*. Kliknimo na ikonco v zgornjem desnem kotu tega gradnika, da se prikaže *SqlDataSource Tasks* meni in nato kliknimo na *Configure Data Source*. Prikaže se okno *Configure Data Source*, ki je zelo podobno (ne pa enako) oknu za določanje podatkovnega vira v Windows aplikacijah. Okno bomo uporabili za določanje povezave s podatkovno bazo in za pridobivanje podatkov iz tabele *Customers* podatkovne baze *Northwind*.

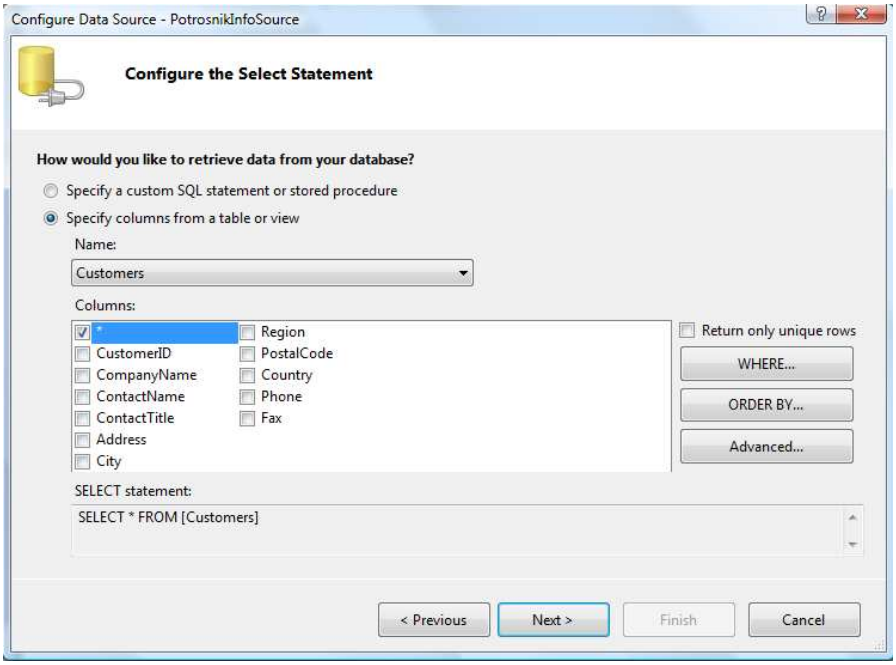
6. Kliknimo gumb *New Connection*. V oknu *Add Connection* nastavimo lastnosti tako kot prikazuje naslednja slika, ter kliknimo gumb *OK*.



7. Ponovno se prikaže okno *Configure Data Source*, v katerem kliknimo gumb *Next*.

8. V oknu *Configure Data Source* shranimo *Connection string* kot *NorthwindConnectionString* in kliknimo *Next*.

9. Na strani *Configure the Select Statement page* se prepričajmo, da je izbrana opcija »*Specify columns from a table or view*«. V spustnem seznamu *Name* izberimo tabelo *Customers*, v seznamu *Columns* pa odkljukajmo vrstico »*«.



10. Kliknimo gumb *Advanced*. V oknu *Advanced SQL Generation Options* odkljukajmo stavek »*Generate INSERT, UPDATE, and DELETE*«. Kliknimo gumb *OK* in nato kliknimo *Next*.

    ❖ Opomba: Tudi če ne izberemo opcije Generate INSERT, UPDATE, and DELETE, bomo še vedno imeli možnost spremembe podatkov, a teh sprememb ne bomo mogli poslati nazaj na strežnik. Lahko pa dodamo ukaze za spremembo baze podatkov po kreiranju ustreznega podatkovnega izvora in spremembo lastnosti *DeleteQuery*, *InsertQuery* in *UpdateQuery* in seveda z zapisom ustreznih SQL stavkov.

11. V oknu *Configure Data Source* preverimo pravkar nastavljeno poizvedbo tako, da kliknemo gumb *Test Query*. V tabeli se prikažejo se podatki iz tabele *Customers*.

12. Kliknimo gumb *Finish*.

13. S klikom na ikonco v desnem zgornjem kotu gradnika *SqlDataSource* zaprimo *SqlDataSource* meni.

V naslednji vaji bomo na naš spletni obrazec dodali še gradnik *GridView* in ga povezali z podatkovnim izvorom *PotrosnikInfoSource*.

1. V oknu Toolbox kliknimo kontrolo *GridView* in jo postavimo na obrazec. S klikom kamorkoli na površino obrazce skrijmo prikazani meni kontrole *GridView*. Gradnik GridView nato razširimo tako, da zapolni večji del obrazca.

2. Z uporabo okna *Properties* spremenimo lastnost *ID* kontrole *GridView* v *CustomerGrid*.

3. Kontrola GridView naj bo še naprej izbrana. Kliknimo ikonco v zgornjem desnem kotu tega gradnika da se prikaže *GridView Tasks* meni. V tem meniju kliknimo link *Auto Format*.

4. V pogovornem oknu *Auto Format* izberimo opcijo *Classic Sheme* in kliknimo gumb *OK*.

- Opomba: Če nam ne ustreza noben od ponujenih formatov shem tabele potrošnikov, lahko stil elementov kontrole *GridView* spremenimo tudo ročno v oknu Properties. Najpogostejše lastnosti, ki jih spreminjamo so *BackColor*, *BorderStyle*, *BorderWidth*, *FooterStyle*, *HeaderStyle* in *RowStyle*..

5. V meniju *Grid View Tasks* (ki je še kar odprt) iz spustnega seznama *Choose Data Source* izberimo *PotrosnikInfoSource*. Na našem spletnem obrazcu se na vrhu stolpcev prikažejo imena polj iz tabele *Customers*.

6. Kliknimo ikonco v zgornjem desnem kotu gradnika *GridView*, da se meni zapre.

7. V meniju Debug kliknimo *Start Without Debugging*.

8. V prijavni obrazec se prijavimo kot **John** in vtipkajmo geslo **Pa$$w9rd**. Po uspešni prijavi se prikaže obrazec *Potrosnik.aspx*, ki prikazuje podatke iz celotne tabele Customers podatkovne baze Northwind.

CustomerID	CompanyName	ContactName	ContactTitle	Address	City	Region	PostalCode	Country	Phone	Fax
ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative	Obere Str. 57	Berlin		12209	Germany	030-0074321	030-0076545
ANATR	Ana Trujillo Emparedados y helados	Ana Trujillo	Owner	Avda. de la Constitución 2222	México D.F.		05021	Mexico	(5) 555-4729	(5) 555-3745
ANTON	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312	México D.F.		05023	Mexico	(5) 555-3932	
AROUT	Around the Horn	Thomas Hardy	Sales Representative	120 Hanover Sq.	London		WA1 1DP	UK	(171) 555-7788	(171) 555-6750
BERGS	Berglunds snabbköp	Christina Berglund	Order Administrator	Berguvsvägen 8	Luleå		S-958 22	Sweden	0921-12 34 65	0921-12 34 67
BLAUS	Blauer See Delikatessen	Hanna Moos	Sales Representative	Forsterstr. 57	Mannheim		68306	Germany	0621-08460	0621-08924
BLONP	Blondesddsl père et fils	Frédérique Citeaux	Marketing Manager	24, place Kléber	Strasbourg		67000	France	88.60.15.31	88.60.15.32
BOLID	Bólido Comidas preparadas	Martin Sommer	Owner	C/ Araquil, 67	Madrid		28023	Spain	(91) 555 22 82	(91) 555 91 99

Stran je seveda read-only, podatkov v tabeli ne moremo spreminjati. V nadaljevanju bomo spletni obrazec preuredili tako, da bo uporabnik podatke v tabeli lahko tudi ažuriral.

9. Zaprimo Internet Explorer in se vrnimo v Visual Studio.

Zaščita spletnih strani in SQL Server

Kadar za zagon spletne aplikacije, katere varnost temelji na obrazcih, uporabljamo *ASP.NET Development Server*, se le-ta izvaja v kontekstu uporabniškega računa, ki ga uporabljamo za zagon Visual Studia. Pri predpostavki, da smo pri kreiranju baze *Northwind* uporabili isti račun, naša spletna aplikacija z dostopom do baze podatkov ne bi smela imeto nobenih težav.

Situacija pa se povsem spremeni, če našo spletno stran poganjamo v IIS (Internet Information Services). IIS izvaja aplikacije, katerih zaščita temelji na obrazcih s počjo računa ASPNET. Ta račun ima v osnovi zelo malo pravic namenjenih varnosti. V splošnem sploh ne bo možna povezava na SQL Server Express in poizvedovaje iz baze Northwind. Zaradi tega je potrebno računu ASPNET najprej omogočiti dostop do SQL Server Express-a in ga dodati kot uporabnika v bazo Northwind.

Prikaz podatkov na posameznih straneh

Prenašanje podatkov vseh potrošnikov iz tabele *Customer* s strežnika v spletni obrazec je zelo koristno, a predpostavimo, da je teh podatkov ogromno. Malo verjetno je, da bo uporabnik želel aktivno brskati po tisočerih vrsticah. Generiranje strani z ogromnim število vrstic in njihovim prikazovanjem na ekranu je zaradi tega potrata časa in nepotrebno obremenjevanje omrežja. Namesto tega bi bilo bolje, da podatke prikažemo v posameznih kosih in uporabniku omogočiti, da se pomika po posameznih straneh s podatki. V naslednji vaji bo prikazan tak način prikazovanja podatkov.

1. Odprimo spletno aplikacijo *Podjetje* in aktivirajmo obrazec *Potrosnik.aspx*, ki naj bo prikazan v načinu *Design View*. Na njem izberimo kontrolo *CustomerGrid*. V oknu Propeties nastavimo lastnost *AllowPaging* na *true*.

   Gradniku *GridView* se pri tem na spodnjem koncu doda dodatna vrstica (noga), ki vsebuje tudi podatek o številu strani. Stil »noge« lahko oblikujemo sami, prikazana oblika pa je privzeta.

2. V oknu Properties nastavimo lastnost *PageSize* na 8. V gradniku *GridView* bo naenkrat prikazano le 8 vrstic podatkov.

3. Razširimo lastnost *PageStyle*. Prikazane podlastnosti lahko uporabljamo za formatiranje posamezne strani. Nastavimo lastnost *HorizontalAllign* na *Left*. Številke strani na dno gradnika GridView se pomaknejo na levo stran.

4. Razširimo še lastnost PageSettings. Prikazane podlastnosti se uporabljajo za določanje izgleda navigacije po naših spletnih straneh. Navigacijo lahko določamo na dva načina: kot številke spletnih strani, ali pa le s prikazom puščic za pomikanje naprej oziroma nazaj po spletnih straneh. Za številčni prikaz strani in puščicama za skok na začetek in na konec strani nastavimo lasnost *Mode* na *NumericFirstLast*. Lastnost PageButtonCount nastavimo na 5 in s tem bodo strani na dnu obrazca oštevilčene v grupah po 5. Po straneh se lahko pomikamo s klikanjem na ustrezne številke strani, v vsakem trenutku pa se lahko pomaknemo tudi na začetno oziroma na zadnjo stran.

   Če hočemo uporabljati puščice naprej/nazaj, lahko spremenimo privzeta znaka »>« in »<« tako, da spremenimo lastnosti *NexPageText* in *PreviousPageText*. Prav tako lahko spremenimo oznaki »<<« in »>>« za skok na prvo oziroma na zadnjo stran podatkov. Vrednosti, ki jih vpisujemo za te lastnosti pa morajo HTML oznake, sicer izpis ne bo pravilen (npr. simbol »>« je definiran kot »@gt;«). Če želimo, lahko določimo tudi ime neke datoteke s sliko, ki se bo na spletni strani prikazala v obliki gumba za premikanje po straneh. Te slikce nastavimo v lastnostih *FirstPageImageUrl*, *LastPageImageUrl*, *PreviousPageImageUrl* in *NextPageImageUrl* (slike bodo prikazane le, če uporabnikov brskalnik njihov prikaz omogoča!).

5. Zaprimo Internet Explorer in se vrnimo v Visual Studio.

Optimiziranje dostopa do podatkov

Podatkovna skladišča (baze) in upravljanje s podatki

V naslednjih poglavjih bo razložena in prikazana manipulacija s podatkovnimi bazami. Naučili se bomo kreirati podatkovno bazo kar znotraj okolja Visual C# Express Edition, kasneje pa bomo uporabljali že izdelane primere podatkovnih baz, ki že vsebujejo podatkovne tabele s testnimi podatki. Za izdelavo podatkovnih baz pa obstajajo seveda tudi druga, bolj specializirana orodja, npr. Microsoftov **SQL Server Management Studio Express**. Razumevanje naslednjih poglavij bo tako možno le v primeru, da poznamo osnove relacijskih podatkovnih baz.

Uporaba baze podatkov

Za izdelavo okenskih aplikacij, ki znajo upravljati z bazami podatkov, je v okolju Visual Studio namenjena skupina objektov **ADO.NET**. To je v bistvu nadgradnja objektov **ActiveX Data Objects** (**ADO**), ki pa je posebej načrtovana in optimizirana za .NET okolje (**.NET Framework**).

Uporaba ADO.NET podatkovnih baz

S prihodom **.NET** orodij, se je Microsoft odločil tudi za nadgradnjo svojega modela dostopa do podatkovnih baz (**ActiveX Data Objects – ADO**) in tako je nastal **ADO.NET**. S pomočjo mehanizmov **ADO.NET**, je lahko povezava s podatkovno bazo uporabljena v različnih aplikacijah, pri čemer obstaja možnost začasne prekinitve povezave in ponovne vzpostavitve kar znotraj aplikacije. Tak način dela mnogokrat predstavlja občuten prihranek časa.

Kreiranje nove baze podatkov

Razvojno Visual C# Express Edition 2008 (pa tudi vse prejšnje in pa seveda višje verzije) omogoča tudi kreiranje podatkovne baze in seveda ustreznih tabel, indeksov in vsega ostalega znotraj baze. Seveda nam morajo biti prej poznani ključni pojmi glede baz podatkov: kaj pravzaprav baza je, čemu služi, kaj so to podatkovne tabele znotraj baze, kaj je to polje, tipi polj, pojem zapisa oz. recorda, indeksne datoteke, …

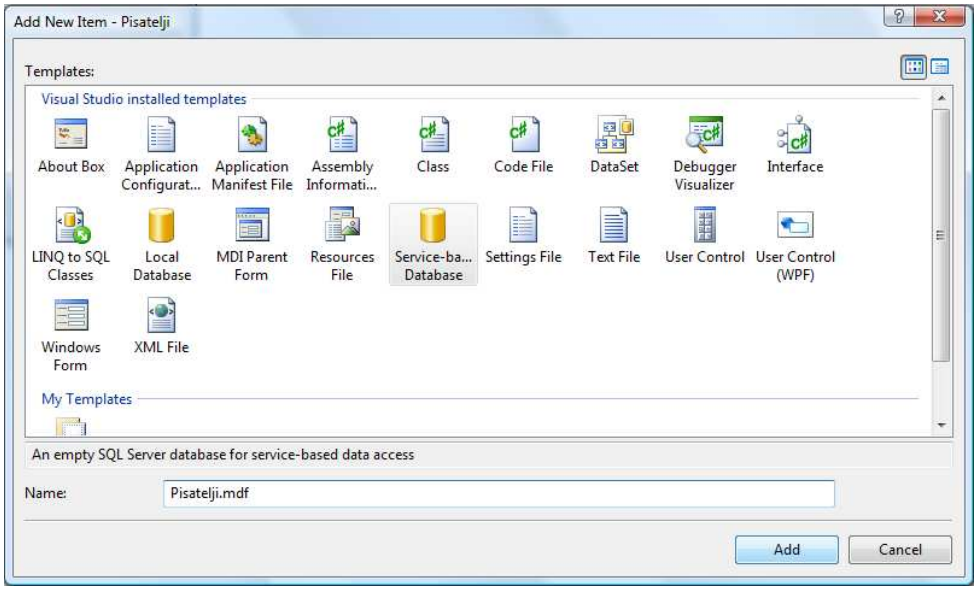
Pri kreiranju baze kar znotraj razvojnega okolja Visual C# Express Edition, imamo na izbiro dve vrsti baze podatkov:

- Local Database in
- Service Based Database.

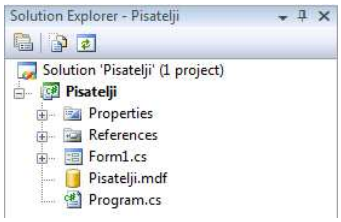
Local Database je t.i. compact edition baza, ki temelji na datotekah. Za neposreden dostop do podatkov v taki bazi potrebujemo le ustrezen gonilnik (*Driver*). Taka baza ne podpira t.i. stored procedur, ki predstavljajo zelo močan mehanizem, ki skrbi za varnost podatkov. Datoteka, v kateri je baza shranjena, ima končnico SDF (*SQL Server Compact Edition format*). Za dostop do take baze na lokalnem računalniku ni potrebna instalacija lokalnega strežnika.

Service Based Database pa je baza podatkov, ki je dostopna le s pomočjo SQL strežnika. Datoteka, v kateri je baza shranjena, ima končnico MDF, ki predstavlja *SQL Server format*. Za priklop na *SQL Server* bazo podatkov je na lokalnem računalniku potreben zagon *SQL Server* servisa, saj le preko tega servisa lahko dostopamo do podatkovnih tabel v bazi.

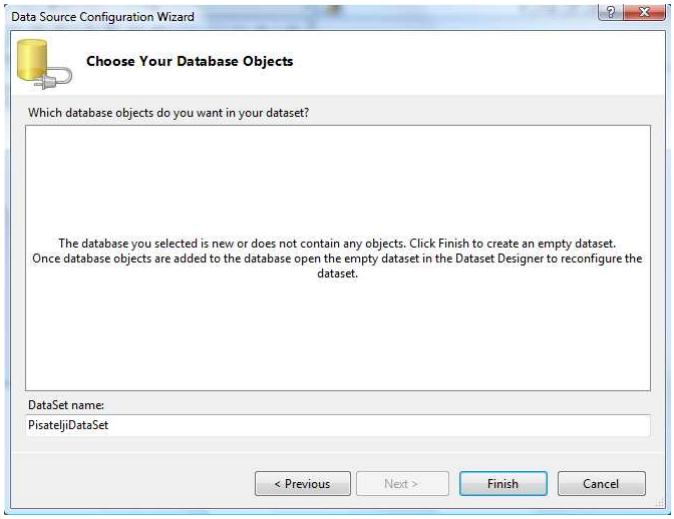
Kreirajmo nov projekt in ga poimenujmo *Pisatelji*. Novo bazo, ki jo bomo v tem projektu uporabili, kreiramo najprej tako, da v meniju *Project* izberemo *Add New Item…*



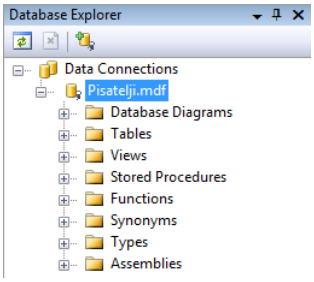
Izberimo *Service-based Database*, bazo poimenujmo *Pisatelji.mdf* in kliknimo *Add*. V *Solution Explorerju* se pojavi nova datoteka *Pisatelji.mdf*, na ekranu pa se čez nekaj časa pojavi okno *Data Source Configuration Wizard*. V nem je opozorilo, da ja baza, ki smo jo izbrali, nova oz. da še ne vsebuje nobenega objekta (npr. nobene podatkovne tabele).



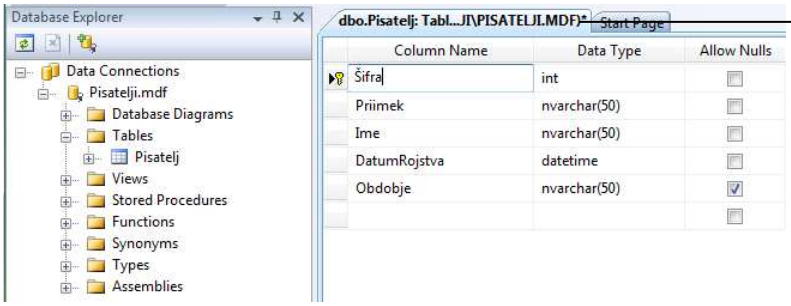
Kliknimo gumb *Finish*: skreiral se bo prazen *DataSet*. Kasneje, ko bomo v bazo že dodali (skreirali) podatkovne tabele, bomo odprli prazen *DataSet* in s tem rekonfigurirali našo bazo podatkov.



Dvokliknimo sedaj na datoteko *Pisatelji.mdf* v *Solution Explorerju* in (običajno na levi strani) se nam odpre novo okno *Database Explorer*.

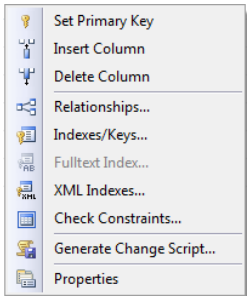


V *Database Explorer*-ju Izberimo vrstico *Tables*, kliknimo desni miškin gumb in nato *Add New Table*. Odpre se urejevalnik polj za novo tabelo. Pisatelja bomo opisali z naslednjimi polji:

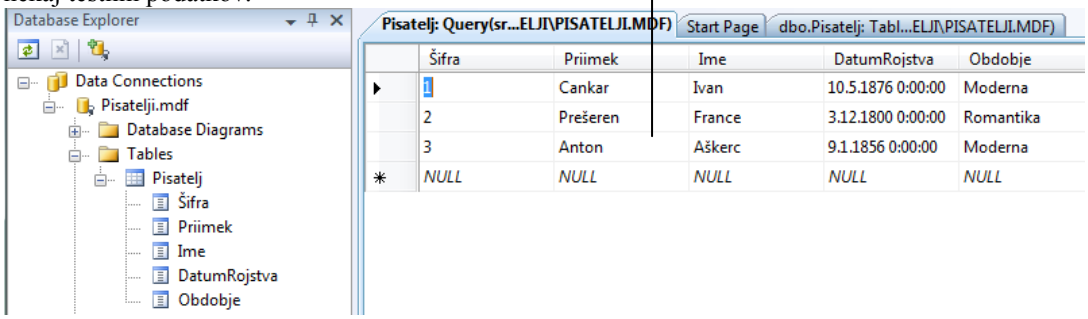


Ko smo vnesli imena polj, se z miško postavimo na zavihek nad urejevalnikom polj in kliknimo desni miškin gumb. Pojavi se okno za vnos imena podatkovne tabele: vnesimo *Pisatelj*.

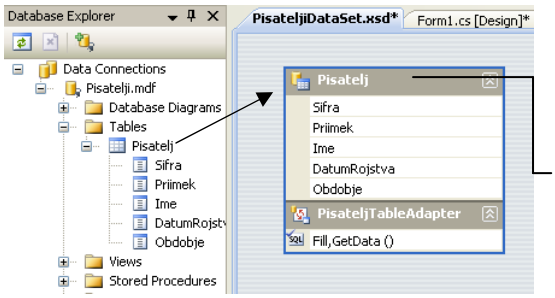
Nato se z miško postavimo še v polje *Šifra* in kliknimo desni gumb. V oknu, ki se prikaže, izberimo *Set Primary Key*: nastavili smo t.i. primarni ključ (vrednost polja *Šifra* se v tabeli ne more nikoli ponoviti).



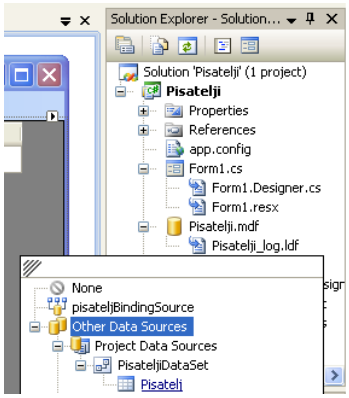
Končno se ponovno postavimo z miško na zavihek nad urejevalnikom polj, kliknimo desni miškin gumb in izberimo opcijo *Save Pisatelj*. Kreiranje prve tabele znotraj naše baze podatkov *Pisatelji* je tako končano.
Sedaj lahko pričnemo z vpisovanjem testnih podatkov v to tabelo. V *DataBase Explorerju* napravimo desni klik na tabelo *Pisatelji* in izberimo opcijo *Show Table Data*. V tabelo, ki se prikaže, lahko že kar na tem vnesemo nekaj testnih podatkov.



Pri kreiranju zgornje baze *Pisatelji* in v njej tabele *Pisatelj*, nam je Visual C# skreiral tudi prazen *DataSet* in ga poimenoval *PisateljiDataSet* (ime smo pustili kar privzeto). V ta *DataSet* bomo sedaj dodali našo tabelo Pisatelji in jo tako naredili dostopno gradnikom na obrazcu. V *Solution Explorer* –ju dvokliknimo na *PisateljiDataSet*, nato pa v okno, ki se odpre, povlecimo našo tabelo Pisatelj. Tabela je sedaj pripravljena in do nje lahko dostopamo tudi preko lastnosti gradnikov. Na tem mestu lahko tudi vidimo predogled vsebine podatkovne tabele Pisatelj: miško postavimo v naslovno vrstico in ob desnem kliku se odpre meni, v katerem izberemo *Preview Data*. Odpre se okno *Preview Data*, v katerem samo še kliknemo gumb *Preview*.



Vsebino tabele *Pisatelj* bomo sedaj prikazali npr. v gradniku *DataGridView*. Na zaenkrat še prazen obrazec našega projekta (na začetku smo ga poimenovali *Pisatelji*) dodajmo gradnik *DataGridView* (njegovo ime naj bo kar privzeto – *dataGridView1*). Lastnost *Dock* mu natavimo na *Fill*, da se razširi čez celoten obrazec. V tem gradniku bi sedaj radi prikazali vsebino tabele *Pisatelj* naše baze *Pisatelji*. Postopek je sledeč: izberimo gradnik *dataGridView1* in v oknu *Properties* lastnost *DataSource*. V spustnem seznamu izberemo *Other Data Sources*. Kliknemo na znak +, da se seznam razširi, nato razširimo še *ProjectDataSources*, pa *PisateljiDataSet* in končno izberemo *Pisatelj*.



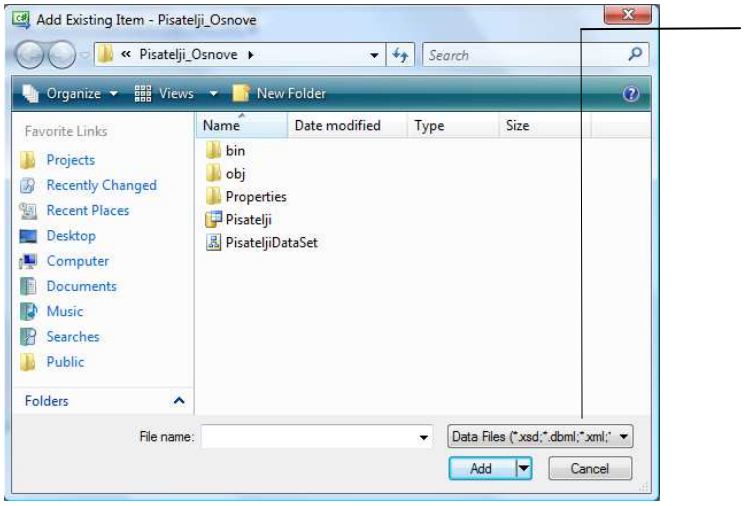
Naš obrazec sedaj izgleda takole: v gradniku *dataGridView1* so nastali stolpci, ki so dobili enaka imena kot so polja v tabeli *Pisatelji* (seveda lahko napise na vrhu stolpcev poljubno preimenujemo – v našem primeru smo na vrh prvega stolpca napisali *Šifra* namesto *Sifra*). Na obrazcu je vsebina tabele *Pisatelji* zaenkrat še nevidna, a če projekt zaženemo, je gradnik *dataGridView1* napolnjen z ustreznimi podatki tabele *Pisatelj*.

Ko smo gradili projekt nam je Visual C# na obrazec dodal nekaj gradnikov, katerih pomen in uporabo bomo spoznali v nadaljevanju.

Izdelava novega projekta, ki dela nad obstoječo bazo podatkov

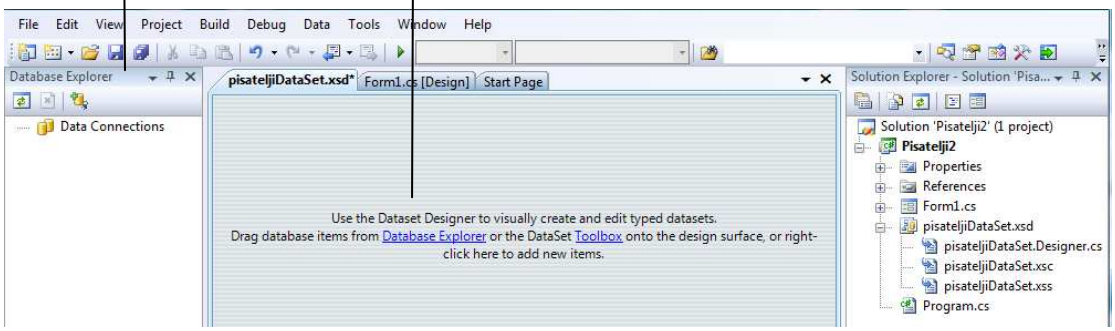
Pri kreiranju novega projekta pa navadno uporabjamo že obstoječo bazo podatkov (bodisi smo jo naredili že kdaj prej, bodisi jo je za nas naredil že kdo drug). Obstoječo bazo podatkov lahko vključimo v naš projekt na več načinov, npr. takole:

- v *Solution Explorerju* označimo naš projekt, kliknemo desni gumb miške, izberemo opcijo *Add* in nato *Add Existing Item*. Odpre se okno za dodajanje obstoječe datoteke v projekt. V spustnem seznamu določimo vrsto datoteke, ki jo bomo dodali v projekt (datoteke tipa *.mdf*).

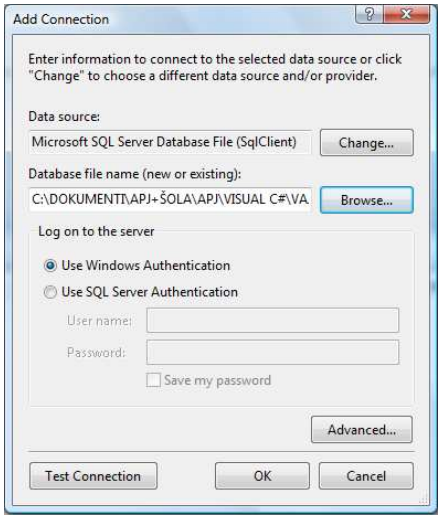


Sedaj moramo le še izbrati ustrezno datoteko – če smo le-to prej skopirali v mapo z našim projektom jo le izberemo, sicer pa se premaknemo v ustrezno mapo, kjer je baza podatkov in jo izberemo. S klikom na gumb *Add* jo vključimo v naš projekt.

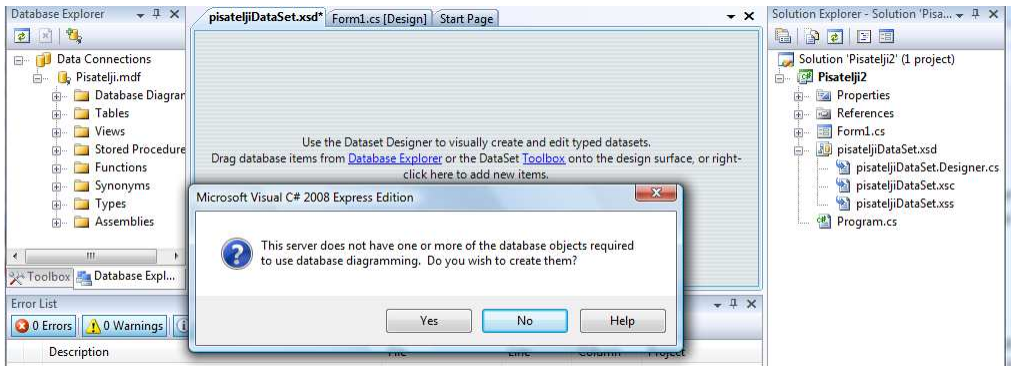
- V meniju *Project* izberemo *ADD New Item*, izberemo *Dataset* in ga poimenujemo *pisateljiDataSet*. Prikaže se okno *Dataset Designer*. V oknu kliknemo na *Database Explorer*, da se prikaže okno *Database Explorer*



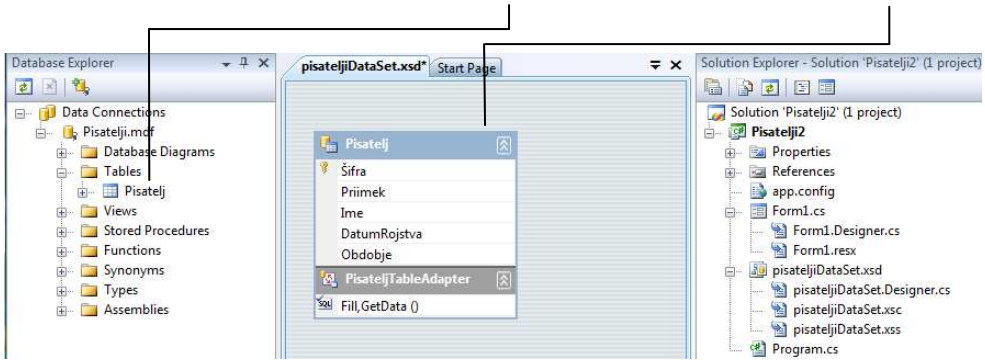
Z miško se postavimo v okno *Database Explorer* in z desnim klikom se nam prikaže okno, v katerem izberemo opcijo *Add Connection*. Prikaže se okno *Add Connection*: za *Data source* izberemo *Microsoft SQL Server Database File*, nato pa s klikom na gumb *Browse* poiščemo našo bazo podatkov (v našem primeru *Pisatelji.mdb*). Po želji še preverimo, če povezava z bazo deluje (klik na gumb *Test Connection*), nato s klikom na gumb *OK* okno zapremo.



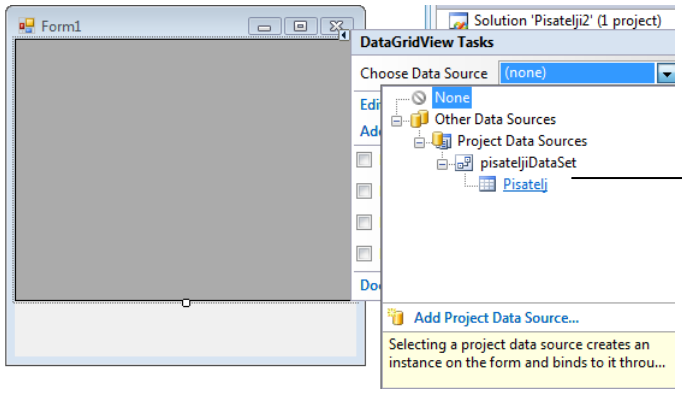
V *Database Explorerju* se pokaže nova vrstica, to je naša baza *Pisatelji.mdf*. Kliknimo nanjo in odpre se okno, kjer nas Visual C# opozori, da v našem projektu še ne obstaja noben *database* objekt, ki je potreben za delo z bazo podatkov. S klikom na gumb *Yes* ga sedaj skreirajmo.



V oknu *Database Explorer* izberimo tabelo *Pisatelj* in jo potegnimo v okno *pisateljiDataSet*

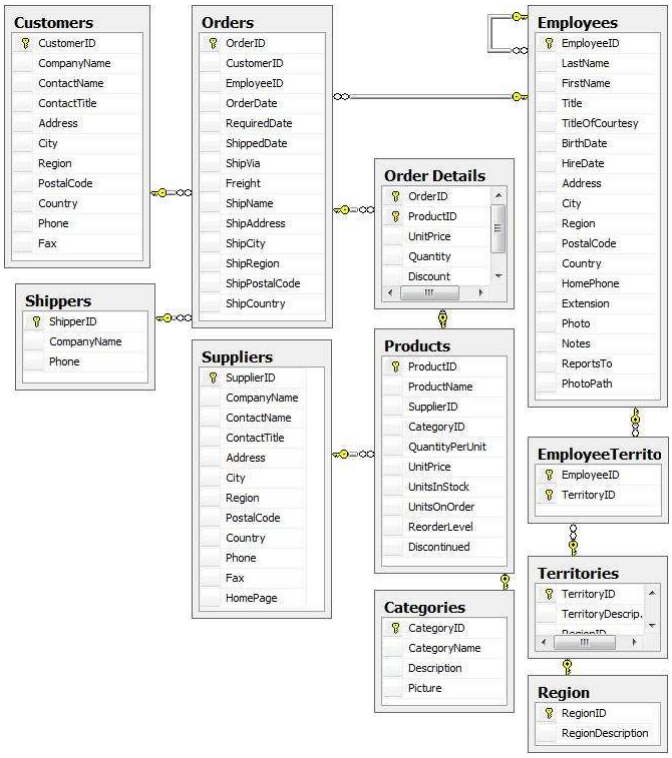


Objekt *pisateljiDataSet* je sedaj pripravljen. Če želimo sedaj vsebino tabele *Pisatelj* prikazati npr. v gradniku *DataGridView*, postavimo ta gradnik na obrazec in v lastnosti *DataSource* izberemo vrstico *Pisatelj*.



Instalacija in uporaba že obstoječih baz podatkov

Preden nadaljujemo z delom, moramo poskrbeti za instalacijo vsaj ene podatkovne baze, ki nam bo služila kot osnova za nadaljnje delo. Sestavni del orodja **Visual Studio** je tudi že izdelana testna baza podatkov z imenom **Northwind**. Gre za bazo, ki predstavlja neko fiktivno podjetje, ki se ukvarja s prodajo hrane. Baza vsebuje vrsto tabel z informacijami o izdelkih, ki jih podjetje prodaja, o svojih kupcih, pošiljateljih in o svojih zaposlenih.



Slika prikazuje vse tabele, ki sestavljajo celotno bazo in način, kako so tabele med seboj povezane.

Podatkovno bazo **Northwind** pa je potrebno na naš računalnik še instalirati. Instalacijo bomo najlaže opravili kar v ukaznem oknu operacijskega sistema Windows (**Command Prompt**)

- Angleška verzija oken (gumb **Start**, **AllPrograms**, **Accessories** in nato **Command Prompt**)
- Slovenska verzija oken (gumb **Start**, **Vsi programi**, **Pripomočki** in nato **Ukazni poziv**)

V ukazni vrstici se prestavimo v mapo **\Microsoft Press\Visual CSharp Step By Step\Chapter 23** znotraj mape **MyDocuments** in vtipkajmo naslednji ukaz:

sqlcmd –S imeRačunalnika\SQLExpress –E –iinstnwnd.sql

pri čemer moramo seveda namesto **imeRačunalnika** vnesti dejansko ime našega računalnika

- POZOR: Ime svojega računalnika lahko preprosto izvemo tudi tako, da v ukazni vrstici (**Command Prompt**) zapišemo ukaz **hostname**

V tej vrstici smo uporabili ukaz **sqlcmd** za vzpostavitev povezave z lokalno instanco serverja **SQL Server 2005 Express** in nato pognali *skripto* **iinstnwnd.sql**. Ta skripta vsebuje **SQL** stavke, ki skreirajo podatkovno bazo **Northwind** in njene pripadajoče tabele, obenem pa te tabele še napolni s testnimi podatki.

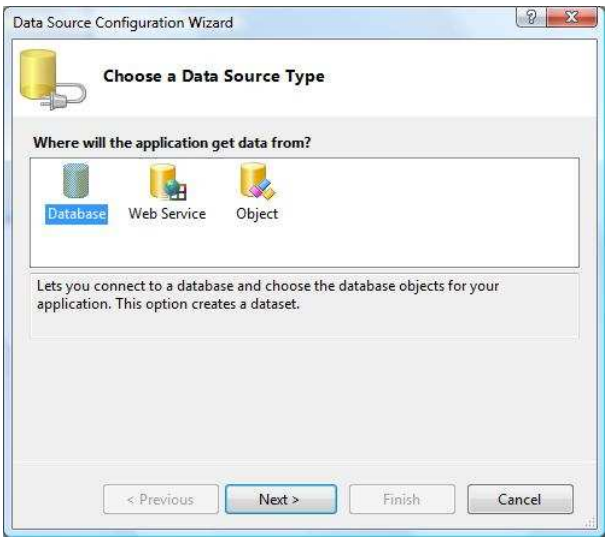
Ko se izvajanje skripte zaključi (vse skupaj traja nekaj sekund), lahko ukazno okno zapremo.

- POZOR: Za izvedbo primerov in vaj v tem poglavju, je potrebna predhodna instalacija orodja Microsoft SQL Server 2005. Microsoft SQL Server 2005 je sicer sestavni del orodja Visual Studio 2008 (oz. 2005) , možen pa je tudi download s strani **http://www.microsoft.com/sql/default.mspx**.

Dostop do baze podatkov

Osnova vsake aplikacije, ki dela z bazo podatkov, je seveda priklop (**connect**) na to bazo. V *Visual Studiu* lahko priklop na bazo in prikaz vsebine poljubne tabele ali več tabel iz baze realiziramo s pomočjo čarovnika, ali pa seveda čisto programsko. V tem delu bomo pokazali, kako se z bazo povežemo s pomočjo čarovnika. **Za uspešno realizacijo naslednjega primera pa potrebujemo vsaj verzijo Visual Studio Academic ali višjo verzijo.**

Kreirajmo nov projekt in ga poimenujmo npr. **Database_Northwind1** in ga shranimo v poljubno mapo. Odprimo meni **Data** in nato **ADD New Data Source**. Zažene se čarovnik z imenom **Data Source Configuration**, ki v splošnem omogoča priklop na poljuben podatkovni izvor – bazo podatkov, objekt ali pa na spletni (**Web**) servis.

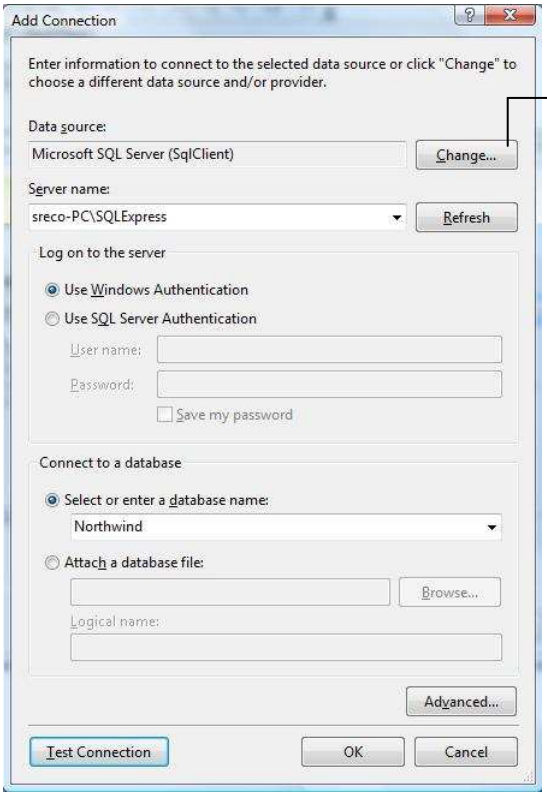


Izberimo ikono **Database**, nato pa kliknimo gumb **Next**.

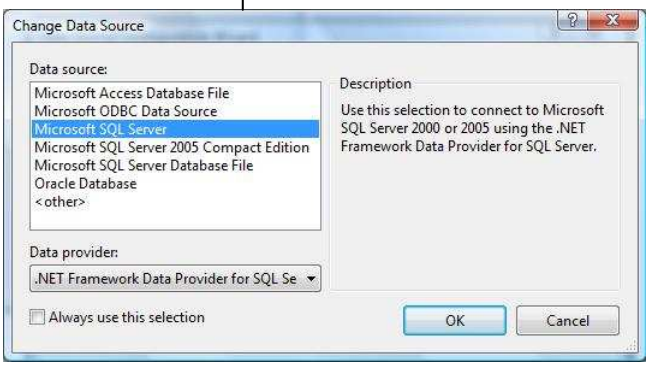
V naslednjem koraku bomo s pomočjo čarovnika določili vrsto povezave za bazo. Odpre se naslednje okno:

Vzpostaviti želimo novo povezavo s podatki, zato kliknimo gumb **New Connection**. Odpre se okno **Choose Data Source**. Izbrati moramo podatkovni izvor (**Data source**), s katerim določimo tip baze, ki jo želimo uporabiti, nato pa še podatkovni skrbnik (**Data provider)**, s katerim bomo določili, kako se bomo na to podatkovno bazo priklopili. Povezavo na *SQL server* lahko dosežemo z uporabo **.NET Framework Data Provider for SQL Server**, ali s pomočjo **.NET Framework Data Provider for OLE DB**. Pri tem velja, da je *.NET Framework Data Provider for SQL Server* optimiziran za povezave z bazami tipa *SQL Server*, *.NET*

Framework Data Provider for OLE DB pa z množico različnih podatkovnih izvorov, ne le z bazami tipa *SQL Server*.

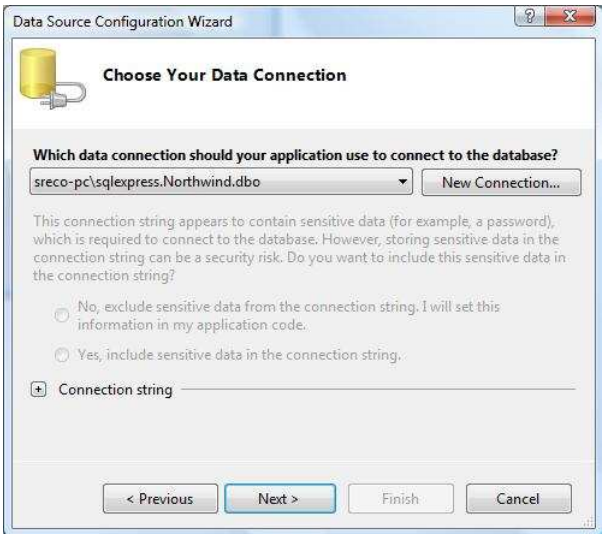


Kliknimo na gumb **Change**, da se odpre pogovorno okno za določanje podatkovnega izvora (**Data source**) in podatkovnega skrbnika (**Data provider**)



V našem primeru izberimo za **Data source** opcijo **Microsoft SQL Server**, za **Data provider** pa **.NET Framework Data Provider for SQL Server**. S klikom na gumb **OK** okno zapremo in vrnemo se nazaj v okno **Add Connection**.

V okno **Server name** vtipkajmo ***ime_serverja*\SQLExpress**, pri čemer je ***ime_serverja*** ime našega računalnika. Za logiranje na strežnik (server) nato izberimo opcijo **Use Windows Authentication** – ta opcija za priklop na bazo uporabi kar ime našega računa (tak način logiranja je priporočen). Končno izberemo še ustrezno bazo podatkov, v našem primeru je to baza **Northwind**. Vse izbire potrdimo s klikom na gumb *OK* in vrnemo se v okno **Data Source Configuration Wizard**: nova podatkovna povezava je dobila ime ***ime_računalnika*\sqlexpress.Northwind.dbo**.



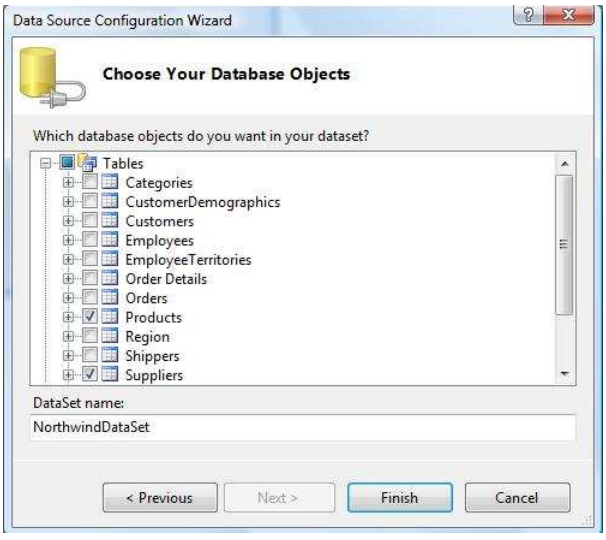
Če kliknemo na gumb + **Connection string**, se nam v oknu prikaže stavek/informacija, ki vsebuje podrobnosti povezave, ki smo ja pravkar določili. Ta informacija je zapisana v formatu, ki jo v nadaljevanju uporabi **SQL provider** za priklop na strežnik.

Kliknimo gumb *Next*, da se odpre okno za shranjevanje informacije o povezavi v ustrezno konfiguracijsko datoteko. Ta opcija nam omogoča, da lahko **string** s podatki o povezavi na strežnik kadarkoli kasneje spremenimo, ne da bi bilo potrebno aplikacijo ponovno prevajati. To je še posebej pomembno, če že v tej fazi izgradnje aplikacije predvidevamo, da bomo kadarkoli kasneje uporabili bazo z drugim imenom, ali pa bomo spremenili pot do nje.

Informacijo o povezavi shranimo kar pod imenom, ki nam jo ponudi čarovnik (**NorthwindConnectionString**).

Kliknimo gumb **Next**, da se odpre okno, v katerem iz izbrane podatkovne baze (**Northwind**) izberemo podatke, ki jih želimo v nadaljevanju uporabiti. Podatke lahko pridobimo iz tabel (**Tables**), ki so sestavni del naše baze, ali pa iz baznih pogledov (**Views**). Ob tem bo čarovnik zgeneriral objekt ***DataSet*** imenovan **NorthwindDataSet**, ki ga bomo kasneje lahko uporabili za manipuliranje z pridobljenimi podatki. Objekt **DataSet** je v bistvu v delovnem pomnilniku shranjena kopija izbranih tabel in stolpcev iz podatkovne baze.

Končno kliknemo gumb **Finish** in s tem je uporaba čarovnika zaključena.



Med uporabo čarovnika za priklop na bazo smo nekje pri koncu tudi shranili informacijo o povezavi v ustrezno konfiguracijsko datoteko. Ta datoteka se imenuje **app.config.** Njeno ime se pojavi tudi v **Solution Explorerju** – gre za XML datoteko in si jo lahko kadarkoli tudi ogledamo:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
    </configSections>
    <connectionStrings>
```

```
        <add name="Database_Northwind1.Properties.Settings.NorthwindConnectionString"
            connectionString="Data Source=sreco-PC\SQLExpress;Initial
Catalog=Northwind;Integrated Security=True"
            providerName="System.Data.SqlClient" />
    </connectionStrings>
</configuration>
```

String za povezavo s podatkovno bazo se nahaja v delu datoteke označenim z **<connectionStrings>**. Ko aplikacijo prevajamo, se vsebina datoteke **app.config** uporabi za generiranje datoteke ***Database_Northwind1.exe.config***, ki vsebuje natanko iste informacije, v enakem formatu, shranjena pa je v isti mapi kot izvršilni program. Ta datoteka se imenuje **application configuration file** in **jo moramo skupaj z izvršilnim programom skopirati na ciljni računalnik** (tja, kjer pač želimo aplikacijo uporabljati). Če se želimo na tem računalniku priklopiti na drugo bazo podatkov (ali pa mogoče le spremeniti pot do baze z enakim imenom), lahko s poljubnim urejevalnikom le spremenimo vsebino datoteke v delu **<connectionStrings>**. Ko bomo ponovno zagnali našo aplikacijo, bo avtomatično uporabljena nova povezava.

Razumevanje pojmov DataSet, DataTable in TableAdapter

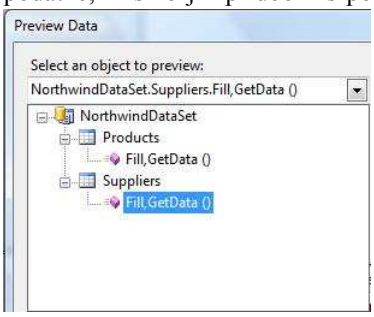
Objektni model ADO.NET uporablja za pridobivanje podatkov iz podatkovne baze objekte tipa **DataSet**. Aplikacija, v kateri so definirani **DataSet** objekti lahko izvaja poizvedbe preko katerih pridobi podatke, nato pa le-te prikaže ali pa jih ažurira. Navznoter pa objekt tipa **DataSet** vsebuje enega ali več objektov **DataTable**; vsak **DataTable** ustreza neki tabeli, ki smo jo označili pri definiciji objekta **DataSet**. V naslednji vaji bomo v objektu **NorthwindDataSet** prikazali dva objekta **DataTable**, imenovana **Products** in **Suppliers**.

V podatkovni bazi *Northwind* imata tabeli *Products* in *Suppliers* relacijo ***več proti ena*** (**many-to-one**): vsak produkt je dobavljen od enega samega dobavitelja, a vsak dobavitelj lahko dostavi več produktov. To relacijsko povezavo zagotavlja baza *Northwind* z uporabo primarnih (**primary**) in zunanjih (**foreign**) ključev. Čarovnik *Data Source Configuration* uporabi to informacijo za kreiranje objekta **DataRelation** ki je del objekta **NorthwindDataSet**. Objekt *DataRelation* zato zagotavlja, da je enaka relacija, kot obstaja podatkovni bazi, podprta tudi med objektoma *Products* in *Suppliers* v delovnem pomnilniku.

Ostane še vprašanje, kako naša aplikacija v času izvajanja zgradi ustrezni objekt *DataSet*. Odgovor leži v objektu imenovanem **TableAdapter**. Ta vsebuje metode, ki jih lahko uporabimo za izgradnjo objekta *DataSet*. Najpogostejši sta metodi imenovani **Fill** in **GetData**. Metoda *Fill* napolni obstoječi *DataSet* in vrne celo število, ki pove število vrstic pridobljenih podatkov, metoda *GetData* pa skreira nov, s podatki napolnjen *DataSet*.

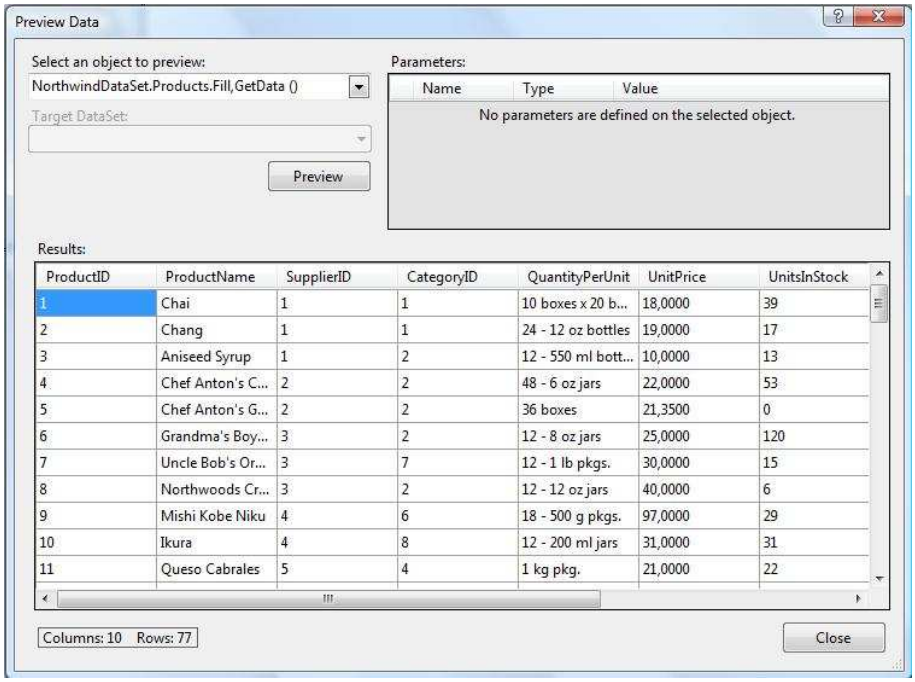
Poglejmo sedaj, kako vse skupaj izgleda v praksi:

V meniju **Data** kliknimo **Preview Data**. Prikaže se pogovorno okno *Preview Data*, ki omogoča vpogled v podatke, ki smo jih pridobili s pomočjo objekta *DataSource*, ki smo ga pravkar kreirali. V tem oknu odprimo



spustni seznam **Select an object to preview** in prikazala se bo drevesna struktura, ki vsebuje izbrano vsebino našega objekta *NorthwindDataSet*. Prikazana sta dva objekta imenovana *Products* in *Suppliers*, pod vsakim od teh dveh objektov pa je še vozlišče imenovano **Fill, GetData()** (če vozlišče ni vidno, kliknite na znak +). Ta prikaz seveda ustreza objektu *TableAdapter* za vsak *TableAdapter*.

Kliknimo vrstico *Fill, GetData()* pod vozliščem *Products*, nato pa še gumb *Preview*. V spodnjem oknu z imenom *Results* se prikažejo vrstice iz tabele *Products* baze *Northwind*.

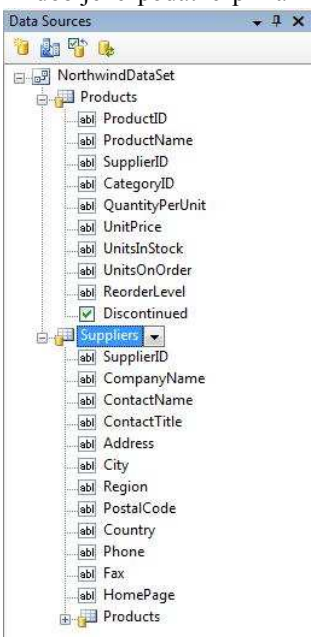


Če bi si želeli ogledati vsebino tabele *Suppliers*, postopek ponovimo za vrstico Fill, *GetData()* pod vozliščem *Suppliers*.

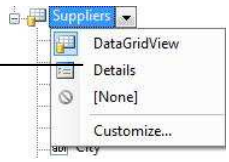
Okno zaprimo s klikom na gumb **Close**.

Prikaz podatkov iz neke podatkovne tabele v naši aplikaciji

Pridobljene podatke prikažimo sedaj še na obrazcu naše aplikacije. V Naši aplikaciji **Database_Northwind1** preklopimo obrazec na *Designer View* in lastnost *Text* preimenujmo v *Suppliers and Products*. Velikost obrazca nastavimo npr. na 800x410. V meniju *Data* kliknimo *Show Data Sources*. Prikaže se okno *Data Sources* in v njem objekt *NorthwindDataSet* z objektoma *DataTable* - to sta seveda objekta *Products* in *Suppliers*. S klikom na vozlišče + razširimo oba objekta.



Kliknimo tabelo (*DataTable*) *Supliers*. Prikaže se spustni seznam/meni in izberimo opcijo **Details**.

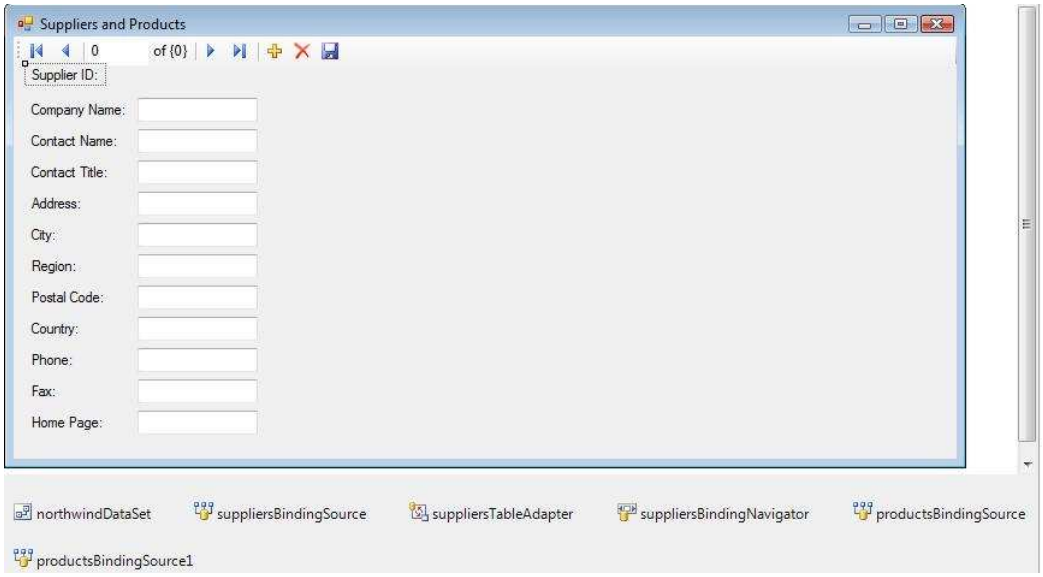


Na ta način bo vrstica tabele *Suppliers* prikazana kot množica podatkovnih polj in ne v tabelarični obliki (oblika **DataGridView**). Prikaz podatkov s pomočjo opcije *Details* je zelo uporaben za prikaz podatkov na strani »ena« v relacijski povezavi več proti ena, medtem ko je prikaz *DataGridView* bolj primeren na strani »več«.

Kliknimo še v v vrstico *SupplierID* v tabeli *Suppliers*. Prikaže se nov spustni seznam: odprimo ga in prikažejo se nam različne možnosti prikaza podatkov *SuplierID*. Ker pa je polje *SupplierID* klučno polje v tabeli *Suppliers* in ga kot takega ne moremo (ne smemo) spreminjati, bomo v ponujenem spustnem seznamu izbrali opcijo

Label. Nato celotno tabelo *Suppliers* primimo z miško in jo potegnimo v zgornji levi rob našega obrazca. Na obrazcu se prikažejo polja iz tabele *Suppliers*, pod obrazcem pa se pojavi nekaj novih gradnikov.

- OPOMBA: Če so polja na obrazcu prikazana previsoko na obrazcu in prekrijejo gradnik za navigacijo, jih (medtem ko so še označena) samo potegnimo nekoliko niže po obrazcu.

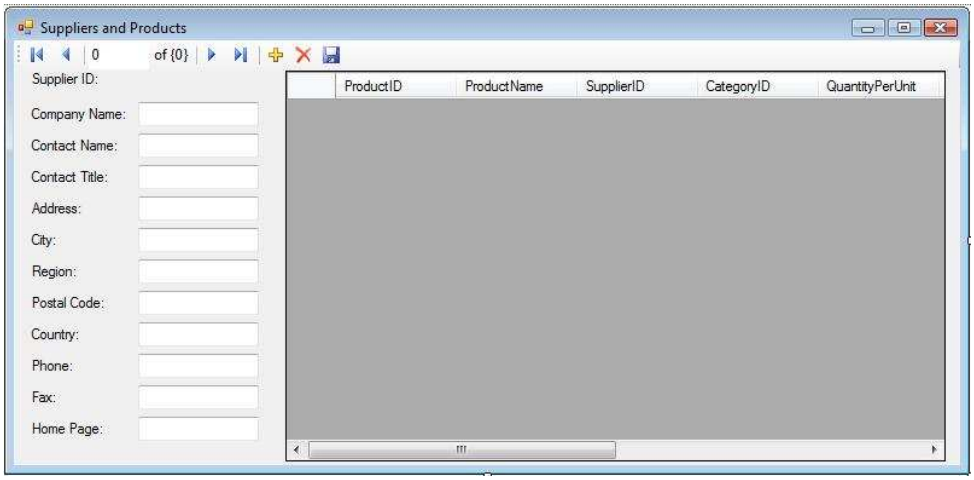


Komponenta	Razlaga
northwindDataSet	Podatkovni izvor uporabljen s pomočjo obrazca. To je objekt *NorthwindDataSet*. Z njegovo pomočjo imamo dostop do metod za ažuriranje podatkov v podatkovni bazi
suppliersBindingSource	Naloga te komponente je nekakšna povezava med gradniki na obrazcu in podatkovnim izvorom. Komponenta *BindingSource* poskrbi za pravilen prikaz podatkov v vrsticah. Vsebuje pa tudi metode za navigacijo po objektu *DataSet* (dodajanje, brisanje in ažuriranje podatkov)
suppliersTableAdapter	Objekt *TableAdapter* za tabelo *Suppliers*, ki zagotavlja metode za pridobitev podatkov iz tabele *Suppliers* podatkovne baze *Northwind*
suppliersBindingNavigator	Gradnik *BindingNavigator* je nekakšen standardiziran mehanizem za navigacijo po vrsticah v objektu *DataSet*. Ta gradnik je na obrazcu tudi viden, pojavi se pod vrhom obrazca, vsebuje pa gradnike za najpomembnejše operacije za podatkovno tabelo

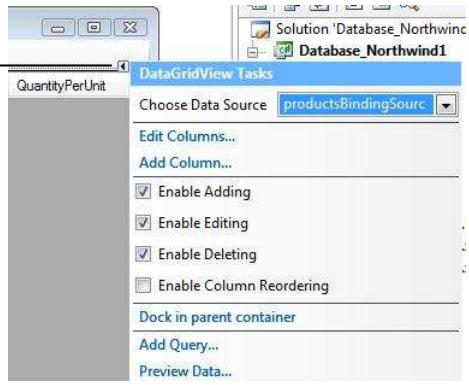
Na obrazcu moramo sedaj prikazati še vsebino tabele *Products*. Kliknimo objekt *DataTable* z imenom *Products* **ki je gnezden znotraj tabele Suppliers** in ga povlecimo na obrazec, desno od polj tabele *Suppliers*. Na obrazcu se prikaže gradnik *DataGridView*, pod obrazcem pa dva nova gradnika: *productsBindingSource* (za koordinacijo vrstic v *DataGridView* z objektom *northwindDataSet*) in *productsTableAdapter* (za pridobitev vrstic podatkov iz podatkovne baze v *Products Data Table*).

- OPOMBA: Prepričajmo se, da je objekt *Products*, ki smo ga potegnili na obrazec res tisti, ki je **gnezden znotraj objekta *Suppliers*** in ne tisti, ki je na vrhu okna *Data Sources*.

Gradnik *DataGridView* nato razširimo čez celotno preostalo površino našega obrazca.

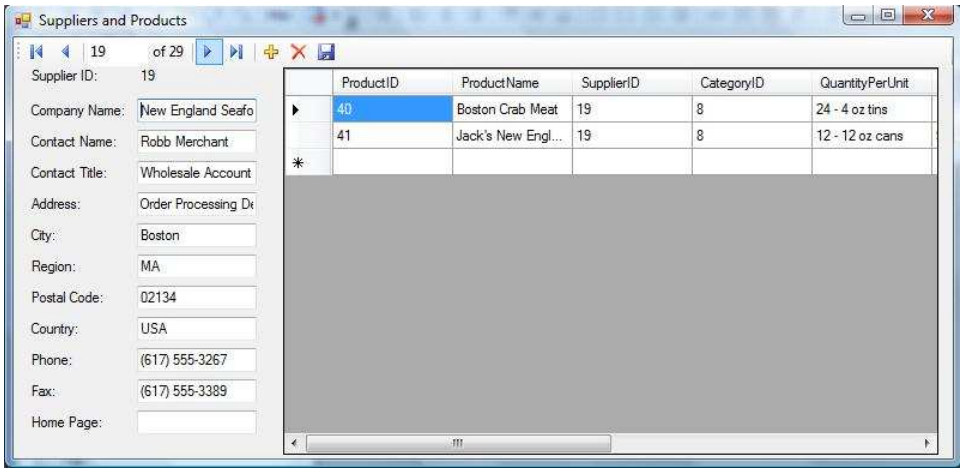


V zgornjem desnem robu gradnika *DataGridView* (medtem, ko je ta še izbran) kliknimo na ročico, da se odpre dialog *Data Grid View Tasks*.



To pogovorno okno lahko uporabljamo za hitro spreminjanje najpogosteje uporabljenih lastnosti gradnika *DataViewGrid*, za spreminjanje lastnosti prikazanih stolpcev in za spreminjanje opravil ki so podprta s tem gradnikom. Da ne bi po nepotrebnem v naši tabeli naredili nezaželene spremembe, zaenkrat odstranimo kljukice, ki omogočajo ažuriranje podatkov (*Enable Editing*, *Enable Deleing* in *Enable Column Reordering*).

Poženimo sedaj našo aplikacijo. V levem robu obrazca je prikazan prvi dobavitelj, desno pa izdelki, ki nam jih ta dobavitelj dobavlja. S pomočjo navigatorja (gradnik nad polji, ki prikazujejo dobavitelja) se lahko premikamo po dobaviteljih, obenem pa se v desni tabeli prikazujejo izdelki izbranega dobavitelja. S pomočjo navigatorja lahko dodajamo nove dobavitelje, oz. brišemo obstoječe.



- POZOR: Dejanske spremembe v bazi bodo izvedene le v primeru, da po koncu ažurirriranja (dodajanja, spreminjanja ali brisanja) kliknemo ikono za shranjevanje.

Programska uporaba ADO.NET (brez čarovnika)

Pri izdelavi bolj kompleksnih aplikacij nam uporaba čarovnika v večini primerov ne bo zadoščala. Dostop do baze podatkov in kreiranje ustrezne poizvedbe bo v takih primerih potrebno realizirati programsko, s pisanjem kode. **Programski dostop do baze podatkov in primeri, ki bodo sledili, so možni tudi v verzijah Express Edition**.

Pri programskem dostopu do baze podatkov potrebujemo dostop do razredov, ki tak dostop omogočajo. Ti razredi se nahajajo v imenskem prostoru **System.Data.SqlClient**, ki ga dodamo v naš projekt.

```
using System.Data.SqlClient;
```

Imenski prostor **System.Data.SqlClient** vsebuje specializirane ADO.NET razrede, ki se uporabljajo za dostop do SQL strežnika.

Deklarirajmo še nov objekt *SQLConnection* (**stavek vrinimo npr. za konstruktorjem razreda Form1**). **SQLConnection** je podrazed, ki je namenjen samo povezavam z bazami podatkov tipa SQL Server.

```
SqlConnection dataConnection = new SqlConnection();
```

Nadaljnji stavki pa so odvisni od tega, ali želimo kreirati samo neko **poizvedbo** (npr. nek SELECT stavek) ali pa želimo zvesti pravo **SQL transakcijo** v bazi podatkov (npr. INSERT ali pa UPDATE stavek). Tako pri poizvedbi, kot tudi pri pravi SQL transakciji je nujna uporaba varovanega bloka, saj pri povezavah lahko pride do številnih problemov in obvestilo o napaki ter vrsti napake je še kako potrebno in dobrodošlo.

Splošna oblika kreiranja poizvedbe iz baze podatkov

Splošna oblika poizvedbe iz neke podatkovne tabele neke baze podatkov ima naslednjo obliko:

```
try
{
  SqlConnection sqlConnection = new SqlConnection("Integrated Security=true;Initial
        Catalog=imeBazePodatkov;Data Source=imeRacunalnika\\SQLExpress");
  SqlCommand dataCommand = new SqlCommand();
  dataCommand.CommandText = "SELECT  . . . ";
  dataCommand.Connection = sqlConnection;
  sqlConnection.Open();
   //obdelava ..
catch (Exception ep)
{
  //Obvestilo o napaki, oz. obdelava napak!
}
finally
{
  //Na koncu prekinemo povezavo z bazo, da bodo spremembe res ažurirane
  dataConnection.Close();
}
```

Primer:

Naslednji primer prikazuje, kako bi izvedli poizvedbo iz baze *nabavaSQL* (testna baza se nahaja na ***http://uranic.tsckr.si/VISUAL%20C%23/***). V bazi so tri tabele (*Artikli*, *Dobavitelji* in *Skupine_artiklov*). Radi bi naredili poizvedbo, s počjo katere bi iz tabele *Artikli* dobili vse tiste artikle, ki imajo vneseno polje *Kolicina* (torej je polje *Kolicina* različno od ***NULL***). Nato želimo še ugotoviti in izpisati skupno količino vseh artiklov v tabeli *Artikli*. To nam omogoča razred *SqlDataReader*, ki vsebuje metode za branje oz. pridobivanje posameznih vrednosti podatkov. Celotno kodo lahko zapišemo npr. v dogodek *Load* poljubnega obrazca. Ko bomo obrazec odprli, bomo v sporočilnem oknu dobili izpisano skupno število artiklov v tabeli *Artikli* znotraj podatkovne baze *nabavaSQL*.

```
//Kreiramo nov objekt za povezavo z bazo podatkov
```

```
SqlConnection dataConnection = new SqlConnection();
//Objektu za povezavo z bazo določimo parametre, ki so potrebni za povezavo
dataConnection.ConnectionString = "Integrated Security=true;" +
                                  "Initial Catalog=nabavaSQL;" +
                                  "Data Source= mojRacunalnik \\SQLExpress";
/*lahko tudi krajše:
   SqlConnection dataConnection = new SqlConnection("Integrated Security=true;Initial
            Catalog=nabavaSQL;Data Source=mojRacunalnik\\SQLExpress");
*/

//Kreiramo nov objekt za SQL poizvedbo
//SqlCommand je razred v ADO.NET, načrtovan za pridobivanje dostopa do SQL Server-ja
SqlCommand dataCommand = new SqlCommand();
//napišemo še katere podatke želimo pridobiti in iz katere tabele ->zapišemo SELECT stavek
dataCommand.CommandText = "SELECT Kolicina From Artikli WHERE Kolicina IS NOT NULL";
//lastnost Connecton sedaj nastavimo na ustrezni objekt dataConnection
dataCommand.Connection = dataConnection;
//Odpremo povezavo z bazo
dataCommand.Connection.Open();

//Kreiramo podatkovni tok za branje in obdelavo vrstic pridobljenih podatkov
/*najhitrejši način za pridobivanje podatkov iz SQL Server baze podatkov je z uporabo
      razreda SqlDataReader. Ta razred pridobi vrstice iz baze podatkov najhitreje kot omrežje
      sploh omogoča in podatke preda naši aplikaciji*/
SqlDataReader dataReader = dataCommand.ExecuteReader();

int skupaj=0;
//metoda Read vrne true, če je poizvedba uspešna, sicer vrne false.
while (dataReader.Read())//dokler obstajajo podatki
{
   try
   {
      /*ker smo v SELECT stavku navedli le en podatek iz vsake vrstice tabele Artikli
        (Kolicina) ima ta podatek znotraj enega stavka indeks enak 0!*/
      skupaj = skupaj + Convert.ToInt32(dataReader.GetValue(0));
   }
   catch
   {
     MessageBox.Show("Napaka pri branju podatkov!");
   }
}
dataReader.Close();//zapremo podatkovni tok
dataCommand.Connection.Close(); //Zapremo povezavo z bazo
```

- POZOR: V stavku **ConnectionString** moramo seveda *mojRacunalnik* nadomestiti z pravim imenom našega računalnika.

Vsebina stavka **ConnectionString** objekta **dataConnection** je povsem enaka, kot jo je zgeneriral čarovnik v primeru, da smo za dostop in prikaz podatkov podatkovne baze *Northwind* uporabili čarovnika. **Posamezni deli stavka ConnectionString so med seboj ločeni s podpičjem**. V stavku smo označili, da bomo za povezavo z bazo naše lokalne instance SQL Server-ja uporabili kar takšno prijavo, kot jo imamo v okolju Windows (*Windows Authentication*). Tak način je priporočljiv, saj v tem primeru uporabniku ni potrebno vnašati njegovega uporabniškega imena in gesla.

Kadar pa bomo pisali aplikacijo namenjeno uporabnikom, ki bodo do nje dostopali npr. preko omrežnih povezav ali celo preko interneta (**remote users**), bomo v stavku *ConnectionString* zapisali tudi uporabniško ime in geslo, npr. takole:

```
string username= . . .;
string password= . . .;
//uporabniku bomo ponudili npr. dve vnosni polji za vnos up. imena in gesla
SqlConnection MojaPovezava = new SqlConnection();
MojaPovezava.ConnectionString = "User ID=" + username + ";Password=" + password +
            ";Initial Catalog=Northwind;Data Source=ime_Racunalnika\\SQLExpress";
```

Splošna oblika SQL transakcije nad bazo podatkov

Splošna oblika SQL transakcije nad neko bazo podatkov ima takole obliko:

```
try
{
  SqlConnection sqlConnection = new SqlConnection("Integrated Security=true;Initial
          Catalog=imeBazePodatkov;Data Source=imeRacunalnika\\SQLExpress");
  SqlCommand dataCommand = new SqlCommand();
  dataCommand.CommandText = "Insert INTO . . . ')";
  dataCommand.Connection = sqlConnection;
  sqlConnection.Open();
  dataCommand.ExecuteNonQuery();
catch (Exception ep)
{
  //Obvestilo o napaki, oz. obdelava napak!
}
finally
{
  //Na koncu prekinemo povezavo z bazo, da bodo spremembe res ažurirane
  dataConnection.Close();
}
```

Primer:

Naslednji primer prikazuje, kako bi izvedli **pravo SQL transakcijo** v tabeli *Artikli* baze *nabavaSQL* (testna baza se nahaja na ***http://uranic.tsckr.si/VISUAL%20C%23/***). V bazi so tri tabele (*Artikli*, *Dobavitelji* in *Skupine_artiklov*). V tabelo *Skupine_Artiklov* bi radi vstavili novo vrstico. V vsaki vrstici tabele sta dve polji (dva podatka): *ID_Skupina* (ključno polje, *Auto Increment* polje) in *Skupina* (niz 50 znakov). Ker je polje *ID_Skupina* deklarirano kot *AutoIncrement* (vrednost se pri dodajanju nove vrstice v tabelo dodeli avtomatično), moramo v novo vrstico tabele *Skupine_artiklov* dodati le vrednost polja *Skupina*. Celotno kodo lahko zapišemo npr. v odzivni dogodek *Click* nekega gumba na poljubnem obrazcu. Ob kliku na ta gumb se bo v tabelo *Skupina_Artiklov* dodala nova vrstica.

```
//Kreiramo nov objekt za povezavo z bazo podatkov
SqlConnection dataConnection = new SqlConnection();
//Objektu za povezavo z bazo določimo parametre, ki so potrebni za povezavo
dataConnection.ConnectionString = "Integrated Security=true;" +
                                  "Initial Catalog=nabavaSQL;" +
                                  "Data Source= mojRacunalnik \\SQLExpress";
/*lahko tudi krajše:
   SqlConnection dataConnection = new SqlConnection("Integrated Security=true;Initial
            Catalog=nabavaSQL;Data Source=mojRacunalnik\\SQLExpress");
*/

//Kreiramo nov objekt za SQL poizvedbo
SqlCommand dataCommand = new SqlCommand();

//Trenutno poizvedbo vežemo na ustrezno povezavo z bazo
dataCommand.Connection = dataConnection;

try
{
   //Kreiramo transakcijski SQL stavek, npr:
   dataCommand.CommandText = "Insert INTO Skupine_artiklov (Skupina) VALUES ('Bla bla')";
   //Napovemo tip poizvedbe: privzeti tip je Text, zato naslednji stavek v tem primeru NI
   //potreben
   dataCommand.CommandType = CommandType.Text;
   //Odpremo povezavo z bazo
   dataConnection.Open();

   //dejansko izvedemo transakcijski SQL stavek; metoda vrne celo število ki pove, na koliko
   //vrstic je vpilvala naša transakcija!
   dataCommand.ExecuteNonQuery();
}
catch (Exception ep)
{
   //Obvestilo o napaki, oz. obdelava napak!
   MessageBox.Show(e.ToString());
}
finally
{
   //Na koncu prekinemo povezavo z bazo, da bodo spremembe res ažurirane
```

```
    dataConnection.Close();
}
```

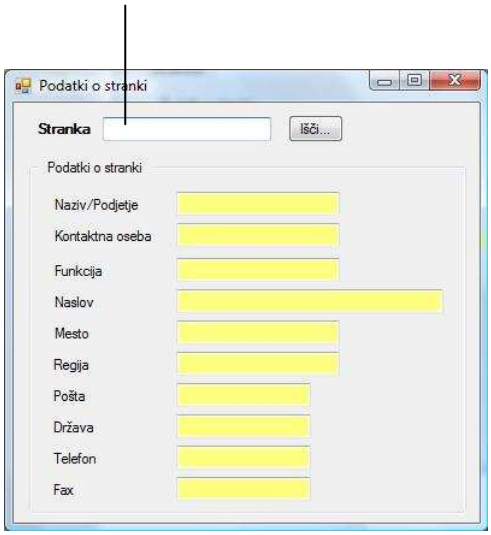
Vaja:

Odprimo nov projekt in ga poimenujmo *Northwind_Customer*. Obrazec preklopimo na kodirni pogled (v Solution Explorerju kliknimo *View Code*, ali pa v meniju *View* kliknimo *Code*) in dodajmo *using* stavek

```
using System.Data.SqlClient;
```

Na obrazec postavimo nekaj gradnikov, v katerih bomo prikazali podatke o določeni stranki tabele *Customers* podatkovne baze *Northwind*.

Vnosno polje, v katerega bomo vnesli ID stranke, katere podatki nas zanimajo.



Po kliku na gumb **Išči…** (**button1**) se prične iskanje stranke v tabeli *Customers*. Koda, ki pripada dogodku **Click** gumba **Išči…** je naslednja (zaradi enostavnosti v tem primeru nismo uporabili varovalnega bloka!!!):

```
private void button1_Click(object sender, EventArgs e)
{
    //SqlCommand je razred v ADO.NET, načrtovan za pridobivanje dostopa do SQL Server-ja
    SqlCommand dataCommand = new SqlCommand();//kreiranje objekta SqlCommand
    //lastnost Connecton sedaj nastavimo na ustrezni objekt dataConnection
    dataCommand.Connection = dataConnection;
    //napišemo še katere podatke želimo pridobiti in iz katere tabele ->zapišemo SELECT stavek
    dataCommand.CommandText = "SELECT CustomerID,CompanyName,ContactName,ContactTitle,Address,
                              City, Region, PostalCode, Country,Phone,Fax ";
    dataCommand.CommandText += "From Customers WHERE CustomerID='" + customerID.Text + "'";
    /*najhitrejši način za pridobivanje podatkov iz SQL Server baze podatkov je z uporabo
      razreda SqlDataReader. Ta razred pridobi vrstice iz baze podatkov najhitreje kot omrežje
      sploh omogoča in podatke preda naši aplikaciji*/
    SqlDataReader dataReader = dataCommand.ExecuteReader();
    //metoda Read vrne true, če je poizvedba uspešna, sicer vrne false.
    if (dataReader.Read() == true) //preverimo če smo zahtevano stranko našli
    {
        //če je neko polje v bazi, katerega vsebino želimo prebrati, enako null, metode
        //dataReager.GetXXX vrnejo napako (izjemo)zato uporabimo metodo
        //dataReader.GetValue(X), prebrano vrednost pa shranimo v spremenljivko tipa object
        object pomozna;
        pomozna = dataReader.GetValue(1);//1 pomeni polje iz SELECT stavka z indeksom
                                         //število 1 (začetni inedks je 0!!!)
        tBPodjetje.Text = pomozna.ToString();

        pomozna = dataReader.GetValue(2);
        tBKontaktna.Text = pomozna.ToString();

        pomozna = dataReader.GetValue(3);
```

```
        tbFunkcija.Text = pomozna.ToString();

        pomozna = dataReader.GetValue(4);
        tBNaslov.Text = pomozna.ToString();

        pomozna = dataReader.GetValue(5);
        tBMesto.Text = pomozna.ToString();

        pomozna = dataReader.GetValue(6);
        tBRegija.Text = pomozna.ToString();

        pomozna = dataReader.GetValue(7);
        tBPosta.Text = pomozna.ToString();

        pomozna = dataReader.GetValue(8);
        tBDrzava.Text = pomozna.ToString();

        pomozna = dataReader.GetValue(9);
        tBTelefon.Text = pomozna.ToString();
        pomozna = dataReader.GetValue(10);
        tBFax.Text = pomozna.ToString();
    }
    else
        MessageBox.Show("Stranka s tem imenom ne obstaja!");
    //ko je poizvedba zaključena, sprostimo odprte vire za dostop do podatkov
    dataReader.Close();
}
```

Ko delo z bazo podatkov končamo, zapremo (prekinemo) še povezavo z podatkovno bazo:

```
private void Form1 FormClosing(object sender, FormClosingEventArgs e)
{
    dataConnection.Close();
}
```

Vaja:

Kreirajmo obrazec, ki bo za izbrano stranko iz tabele *Customers* podatkovne baze *Northwind* prikazal vsa njena naročila. Dodajmo *using* stavek

```
using System.Data.SqlClient;
```

Deklarirajmo še nov objekt *SQLConnection* (**stavek vrinimo npr. za konstruktorjem razreda Form1**).

```
SqlConnection dataConnection = new SqlConnection();
```

V dogodek **Load** obrazca **Form1** zapišimo varovalni blok, znotraj katerega bomo skušali vzpostaviti povezavo z bazo podatkov *Northwind*. Če povezava uspe, napolnimo gradnik *ComboBox* (ime gradnika je comboBox1) z imeni (*CustomerID*) vseh strank iz tabele *Customers*. Ko projekt zaženemo in se obrazec odpre, imamo v gradniku *ComboBox* tako že vse stranke iz tabele *Customer*.

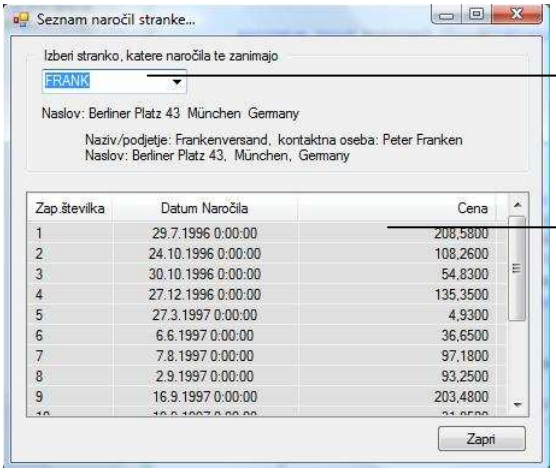
```
private void Form1 Load(object sender, EventArgs e)
{
     try
     {
         dataConnection.ConnectionString = "Integrated Security=true;" +
                                           "Initial Catalog=Northwind;" +
                                           "Data Source=mojRacunalnik\\SQLExpress";
         dataConnection.Open();
         SqlCommand dataCommand = new SqlCommand();
         dataCommand.Connection = dataConnection;
         //kreiramo poizvedbo
         dataCommand.CommandText = "SELECT CustomerID ";
         dataCommand.CommandText += "From Customers";
         SqlDataReader dataReader = dataCommand.ExecuteReader();
         //v comboBox1 zapišemo vse stranke iz tabele Customers
         while (dataReader.Read())//dokler obstajajo podatki
         {
             comboBox1.Items.Add(dataReader.GetValue(0).ToString());
```

```
            }
            dataReader.Close();//zapremo podatkovni tok
        }
        catch
        {
            MessageBox.Show("Napaka pri dostopu do baze podatkov!");
        }
}
```

Na obrazec postavimo sedaj gradnik **ListView** in mu nastavimo naslednje lastnosti:

- *Columns* – kliknimo na **tropičje** in nato v oknu, ki se prikaže dodajmo tri stolpce (3 x kliknemo na gumb Add). Stolpce poimenujemo (lastnost *Text*) *Zap.Številka* (*TextAlign Left*), *Datum naročila* (*TextAlign Center*) in *Cena* (*TextAlign Right*).
- *GridLines* = **true**
- *HeaderStyle* = **Nonclickable**
- *View* = **Details**

Na obrazec dodamo še tri gradnike **Label**, za prikaz vseh matičnih podatkov (ime, naslov) izbrane stranke.



ComboBox

ListWiew

Ob izbiri stranke v gradniku *ComboBox* želimo, da se v gradniku *ListBox* prikažejo vsa naročila te stranke. Ustrezno kodo zapišemo v dogodek **SelectedIndexChange** našega gradnika *ComboBox*.

```
private void comboBox1 SelectedIndexChanged(object sender, EventArgs e)
{
    listView1.Items.Clear();//brišemo prejšnjo vsebino gradnika ListBox
    //iz tabele Customers najprej pridobimo matične podatke izbrane stranke
    SqlCommand dataCommand = new SqlCommand();
    dataCommand.Connection = dataConnection;
    dataCommand.CommandText="SELECT CustomerID,CompanyName,ContactName,Address,City,Country ";
    dataCommand.CommandText += "From Customers WHERE CustomerID='" + comboBox1.Text + "'";
    SqlDataReader dataReader = dataCommand.ExecuteReader();
    if (dataReader.Read() == true)
    {
        object pom;
        pom = dataReader.GetValue(1);
        //matične podatke izbrane stranke zapišemo na obe labeli
        label2.Text = "Naziv/podjetje: " + (dataReader.GetValue(1)).ToString()+",  kontaktna
                       oseba: " + (dataReader.GetValue(2)).ToString();
        label3.Text = "Naslov: "+(dataReader.GetValue(3)).ToString()+",
                      "+(dataReader.GetValue(4)).ToString()+",
                      "+(dataReader.GetValue(5)).ToString();
        dataReader.Close();//zapremo podatkovni tok
    }

    //iz tabele Orders pridobimo vsa naročila izbrane stranke
    dataCommand.CommandText = "SELECT OrderDate,Freight,ShipName,ShipAddress,ShipCity,
                              ShipCountry ";
    dataCommand.CommandText += "From Orders WHERE CustomerID='" + comboBox1.Text + "'";
    dataReader = dataCommand.ExecuteReader();

    int i = 0;
    string naslov = "";
```

```
    while (dataReader.Read())//dokler ne zmanjka podatkov
    {
        //vsako vrstico iz tabele Order na primeren način zapišemo v vrstico gradnika ListBox
        object pomozna;
        listView1.Items.Add((i + 1).ToString());//v prvi koloni je zaporedna številka vrstice
        pomozna = dataReader.GetValue(0);//0 = inndeks polja iz SELECT stavka
        listView1.Items[i].SubItems.Add(pomozna.ToString());
        pomozna = dataReader.GetValue(1);
        listView1.Items[i].SubItems.Add(pomozna.ToString());
        pomozna = dataReader.GetValue(2);
        listView1.Items[i].SubItems.Add(pomozna.ToString());
        naslov = (dataReader.GetValue(3)).ToString() + "  " +
        (dataReader.GetValue(4)).ToString() + "  " + (dataReader.GetValue(5)).ToString();
        i++; //povečamo indeks vrstice gradnika ListBox
    }
    label1.Text = "Naslov: " + naslov; //na labelo zapišemo strankin naslov
    dataReader.Close();//zapremo podatkovni tok
}
```

Ko delo z bazo podatkov končamo, zapremo (prekinemo) še povezavo z podatkovno bazo:

```
private void Form1 FormClosing(object sender, FormClosingEventArgs e)
{
    dataConnection.Close();
}
```

Razred SqlDataReader

Med največjimi pomanjkljivostmi večuporabniških aplikacij, ki delajo z bazami podatkov, je zaklepanje podatkov. Pogosto se zgodi, da pri pridobivanju podatkov (vrstic) iz tabel, aplikacije te vrstice v tabeli zaklenejo saj jih na ta način zaščitijo, da jih ne bi med tem ažurirali drugi uporabniki. V ekstremnih primerih aplikacija drugim uporabnikom celo prepreči branje vrstic podatkov, ki so zaklenjene. Če je poizvedba taka, da je število vrstic zelo veliko, se seveda zgodi, da je zaklenjen kar dobršen del tabele, iz katere se izvaja poizvedba. Ob tem se seveda lahko še zgodi, da tabelo istočasno uporablja veliko število porabnikov. Uporabniki morajo zaradi tega čakati na sproščanje zaklenjenih vrstic, kar pa ima seveda za posledico počasno delovanje programa, zmedo in slabo voljo uporabnikov.

Razred **SqlDataReader** je bil narejen prav z namenom, da do prej opisanih težav ne bi prihajalo. Podatke iz tabel pridobiva enega za drugim in ne pušča zaklenjenih vrstic v tabeli, ko je vsebina vrstice enkrat pridobljena. Tak način pridobivanja podatkov (vrstic) iz tabel pa seveda pomeni veliko prednost pri izboljšanju konkurenčnosti naše aplikacije.

Tudi pri enostavnem pridobivanju podatkov nudi razred *SqlDataReader* večje zmožnosti kot pa uporaba razreda *DataSet*. Objekti tipa *DataSet* za predstavitev podatkov ki jih hranijo uporabljajo XML. Tak način seveda nudi veliko fleksibilnost, saj lahko vsaka komponenta, ki zna brati XML format, te podatke kasneje tudi obdeluje. Razred *SqlDataReader* pa uporablja naraven **SQL Server data-transfer** format za pridobivanje podatkov neposredno iz baze, tako da podatke ni potrebno najprej konvertirati v nek drug format, npr. XML. Razred *SqlDatReader* se torej odlikuje z boljšimi v performansami, na drugi strani pa nam razred *DataSet* nudi več fleksibilnosti.

Zapiranje povezav z bazo podatkov

V starejših aplikacijah so programerji uporabljali pristop, da se je povezava z bazo vzpostavila ob zagonu aplikacije, prekinila pa šele po zaključku dela z aplikacijo. Tak pristop so zagovarjali zaradi tega, ker je bila operacija odpiranja in zapiranja povezave z bazo dragocena in časovno potratna. Slaba stran takih aplikacij je seveda ta, da ima vsak uporabnik, ki uporablja to aplikacijo, z bazo vzpostavljeno neprekinjeno povezavo, ne glede nato, ali jo uporablja ali pa ne (tudi če je šel npr. na kosilo). Poleg tega ima večina baz tudi omejeno število uporabnikov, ki lahko do nje hkrati dostopajo (pogosto je temu tako tudi zaradi omejenega števila licenc!).

Večina *.NET Framework* podatkovnih skrbnikov (data providers), med njimi seveda tudi *SQL Server provider*, je opremljenih s tehnologijo imenovano **connection pooling**. Podatkovne povezave so kreirane in zadržane v nekem skupnem bazenu (*pool*). Če aplikacija zahteva povezavo, podatkovni skrbnik (**data access provider**) povzame naslednjo prosto povezavo iz tega bazena, ob zapiranju pa je le ta vrnjena v bazen in postane dostopna naslednji aplikaciji, ki zahteva povezavo. To pa pomeni, da odpiranje in zapiranje povezav s podatkovnimi tabelami ni več neka potratna operacija, saj zapiranje in odpiranje povezave ne pomeni več priklop oz. odklop iz baze podatkov. Odpiranje povezave je preprosto le stvar pridobitve že odprte povezave iz bazena. Zaradi tega se pri izdelavi aplikacij, ki delajo z bazami podatkov držimo načela, da neko povezavo odpremo šele tedaj, ko jo potrebujemo, ko pa je ne potrebujemo več pa jo takoj zapremo.

Delo s povezavami podatkov (Data Binding) in objekti razreda DataSet

V prejšnjem poglavju smo spoznali temelje uporabe orodja Microsoft ADO.NET za kreiranje SQL poizvedb in za ažuriranje baze podatkov. Uporabili smo bodisi čarovnika, ki nam ga nudi razvojno okolje Visual C#, bodisi smo enostavne poizvedbe kreirali sami – programsko. Za potrebe povezave s podatki smo napisali zelo malo kode. V tem poglavju pa bomo spoznali, da nam Microsoft Visual Studio omogoča tudi pisanje mnogo bolj kompleksnih aplikacij, s katerimi bomo dosegli veliko večje funkcionalnosti, obenem pa bomo ob pisanju spoznali ozadje takih povezav.

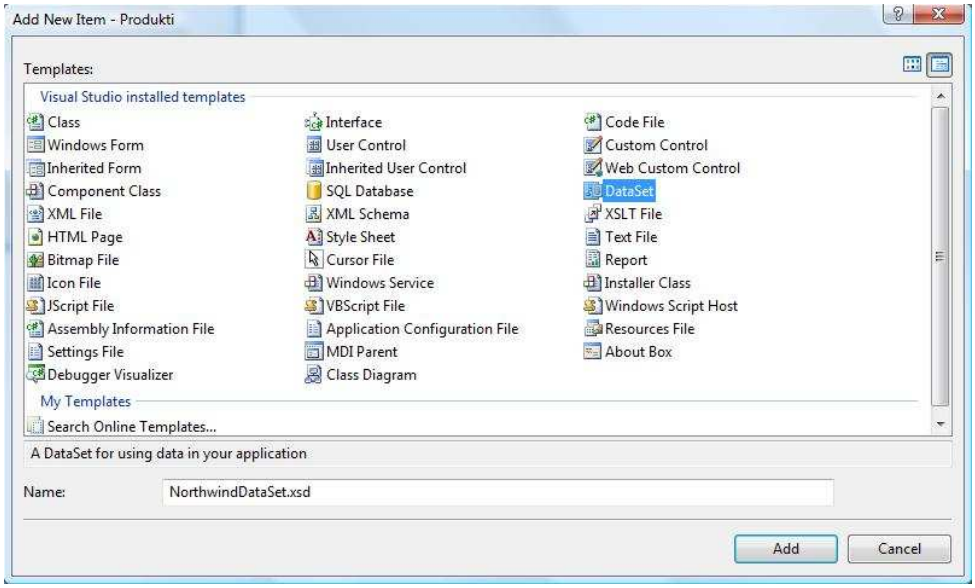
Povezava okenskih gradnikov s podatkovnim izvorom

Mnoge izmed okenskih gradnikov lahko povežemo z nekim podatkovnim izvorom, ali pa so temu namenjeni. Ko je povezava vzpostavljena, je lastnost gradnika, ki je namenjena povezavi, nadomeščena s lastnostjo, ki poskrbi za povezavo in obratno. Okenski obrazci sicer podpirajo dva tipa povezav z izvorom podatkov: enostavnega in kompleksnega.

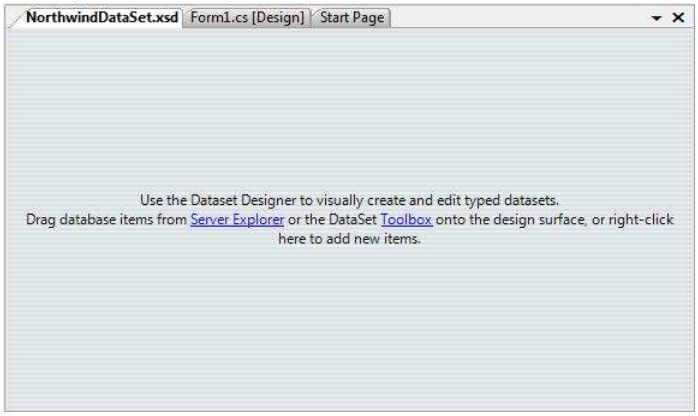
Objekt DataSet in enostavno povezovanje s podatkovnim izvorom

Izvor podatkov je lahko praktično karkoli: od celice v gradniku *DataSet*, do vrednosti neke lastnosti in preproste spremenljivke. Preproste povezave s podatki lahko izvedemo z uporabo lastnosti **DataBinding** izbranega gradnika.

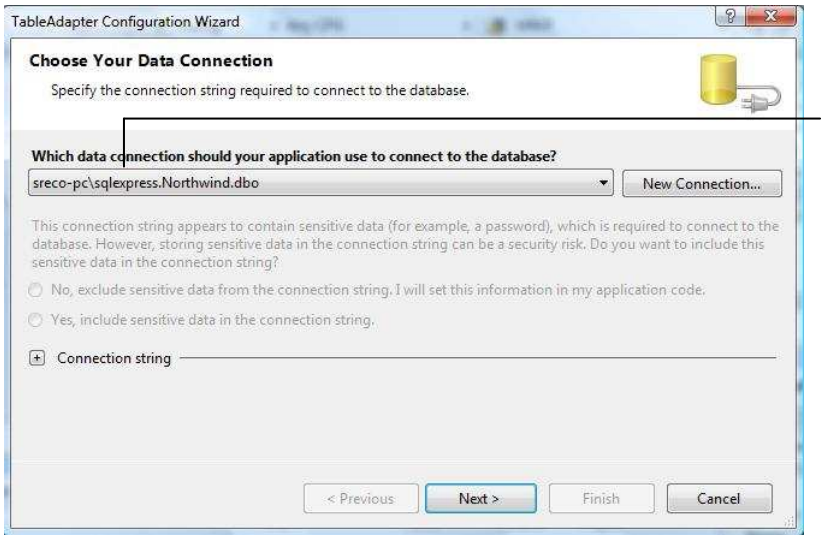
Naslednji primer prikazuje kreiranje objekta *DataSet*, ki bo predstavljal izvor podatkov, ki bo vrnil en sam podatek, nato pa bomo s tem podatkom povezali lastnost *Text* gradnika *Label*. Kreirajmo torej nov projekt in ga poimenujmo *ProductMaintenance*. V meniju *Project* kliknimo opcijo **Add New Item**. Odpre se pogovorno okno **Add New Item**, ki prikazuje seznam predlog/objektov, ki jih lahko dodamo k projektu. Izberimo predlogo *DataSet* in jo poimenujmo *NorthWindDataSet.xsd* (kratica xsd je obvezna: definiran *DataSet* je bistvu XML shema, ki jo Visual Studio uporabi za generiranje kode ko gradimo aplikacijo).



Kliknimo *Add* in na ekranu se odpre okno **DatSet Designer**.

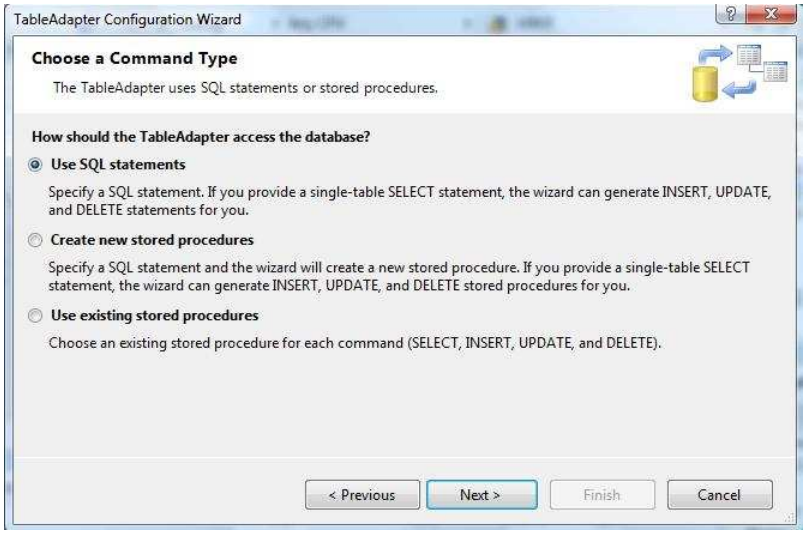


V oknu *Toolbox* (orodjarna), razširimo skupino *DataSet* in izberimo na gradnik *TableAdapter*. Nato kliknimo kamorkoli v okno *DataSet Designer*. V to okno se s tem dodata objekta *DataTable* in *TableAdapter*, prikaže pa se čarovnik **TableAdapter Configuratin Wizard**.



V oknu izberimo ustrezno povezavo (le to smo skreirali že v eni od prejšnjih vaj, če pa povezave še nimamo, jo skreiramo tako, kot smo to naredili v poglavju **Uporaba baze podatkov).**

Izbiro povezave potrdimo s klikom na gumb *Next*. Prikaže se okno **Save the connection string to the application configuration file** – konfiguracijo shranimo pod privzetim imenom **NorthwindConnectionStrings** in kliknimo na gumb *Next*. Odpre se okno **Choose a Command Type**.



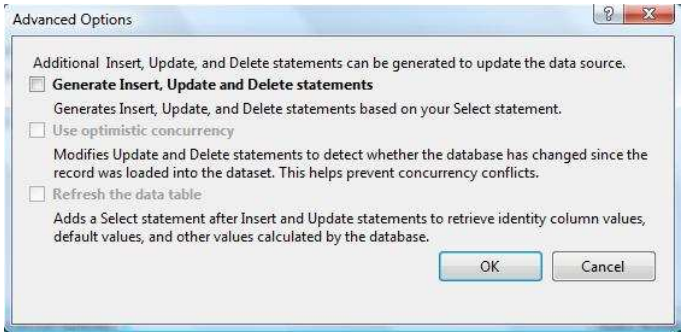
V oknu lahko izberemo, kako bo naš *TableAdapter* dostopal do baze podatkov. Zagotovimo si lahko lastne SQL stavke, lahko dostopamo do čarovnika za generiranje t.i. *stored* procedur, ki bodo za nas enkapsulirale SQL stavke, ali pa lahko uporabimo že obstoječe *stored* procedure. Za našo vajo izberimo opcijo **Use SQL statements** in kliknimo gumb *Next*.

Odpre se okno **Enter a SQL Statement**, v katerega vtipkajmo naslednji SQL SELECT stavek, ki bo ugotovil število vrstic podatkov v tabeli *Products*:

SELECT COUNT (*) AS NumProducts
FROM Products

Po vnosu SELECT stavka kliknimo gumb **Advanced Options** in prikaže se okno **Advanced Options Dialog Box**.

Gradnik oz. objekt *TableAdapter* nam namreč poleg pridobivanja podatkov iz baze podatkov omogoča tudi vstavljanje, ažuriranje in brisanje vrstic. Čarovnik nam tako lahko avtomatično zgenerira tudi SQL INSERT, UPDATE in DELETE stavke za podatkovno tabelo, ki smo jo določili v SELECT stavku.



V našem primeru teh dodatnih stavkov ne bomo potrebovali, saj želimo izvedeti le za število vrstic v tabeli *Products*, zato odstranimo oznako (kljukico) v gradnikih *Check Box* v oknu **Advanced Options** in kliknimo gumb OK.

Zopet se vrnemo v okno **Choose a Command Type**, kjer le kliknimo gumb *Next*. Odpre se pogovorno okno **Choose Methods to Generate**. Objekt *TableAdapter* namreč lahko zgenerira dve metodi za zasedanje objekta *DatSet* s podatki pridobljenimi z baze podatkov

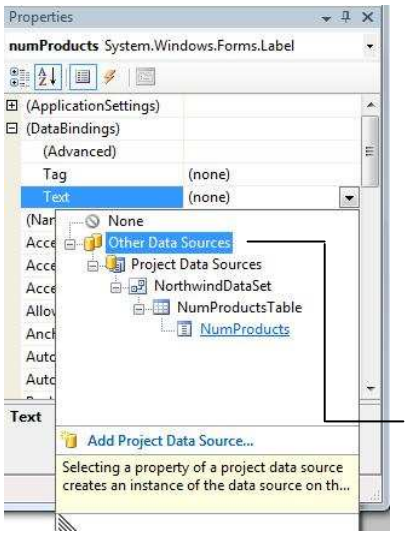
- metoda *Fill* pričakuje za parameter že obstoječi objekt *DataTable* ali pa *DataSet*, ki je napolnjen s podatki
- metoda *GetData* kreira nov objekt *DataTable* in ga napolni s podatki.

V našem primeru pustimo izbrani obe metodi in kliknimo gumb *Next*. Vse informacije, ki smo jih v prejšnjih oknih določili oz. napisali sedaj čarovnik uporabi in zgenerira nov razred **TableAdapter**. Čarovnika sedaj zaprimo s klikom na gumb *Finish*.

V oknu *DataSet Designer* se prikažeta razreda *TableAdapter*, imenovan *DataTable1TableAdapter* in pripadajoči razred *DataTable* imenovan *DataTable1*. V oknu izberimo postavko *DataTable1* in nato v oknu *Properties* ime (lastnost Name) spremenimo v **NumProductsTable**. Nato izberimo še postavko *DataTable1TableAdapter* in ime spremenimo v **NumProductsTableTableAdapter**. Okno sedaj izgleda takole:

V meniju *Build* sedaj kliknimo opcijo *Build Solution* in Visual C# nam bo zgeneriral kodo in objekte potrebne za nadaljevanje te vaje.

Adapter *NumProductsTableAdapter* torej iz baze pridobi ustrezen podatek (število zapisov podatkovne tabele Products) in le-tega shrani v objekt *DataTable* z imenom *NumPoducts*, vse skupaj pa je sestavni del objekta *DataSet* z imenom *NorthwindDataSet*.



V nadaljevanju bomo ta podatek (število zapisov tabele *Products*) povezali z lastnostjo *Text* gradnika *Label*. Ko bomo kasneje aplikacijo zagnali, se bo na labeli izpisal ustrezen podatek o številu zapisov. Obrazec *Form1* sedaj najprej preimenujmo v **ProductsForm.cs**, spremenimo njegovo lastnost *Text* prav tako v **ProductsForm** in na obrazec postavimo eno poleg druge dve labeli. Levi labeli spremenimo lastnost *Text* v **Število produktov:** , desno labelo pa poimenujmo (lastnost Name) **numProducts**. Nato razširimo lastnost *DataBindings* druge labele (labele z imenom *steviloProduktov*), kliknimo na lastnost *Text* (znotraj *DataBindings*) in kliknimo na padajoči meni, ki se prikaže. Meni prikazuje drevesni pogled na dostopne vire podatkov.
Razširimo vozel **Other Data Sources**, razširimo **ProjectData Sources**, razširimo **NorthwindDataSet**, razširimo **NumProductsTable** in končno izberimo **NumProducts**. S tem smo povezali lastnost *Text* naše labele s stolpcem *NumProducts* v objektu *DataTable* z imenom **NumProductsTable**.

Visual Studio nam je ob tem skreiral instanco razreda *NorthwindDataSet* in ga poimenoval **northwindDataSet**, instanco razreda *NumProductsTableTableAdapter* in ga poimenoval **numProductsTableTableAdapter**, in objekt *BindingSource* imenovan **numProductsTableBindingSource**. Vsi trije objekti so v projektu vidni pod obrazcem.

Kliknimo na objekt *numProductsTableBindingSource* pod obrazcem. V oknu *Properties* kliknimo na lastnost *DataSource*. Ta lastnost določa objekt *DataSet*, ki ga objekt *BindingSource* uporablja za priklop na bazo podatkov.

Kliknimo na lastnost *DataMember*. Ta lastnost določa, da objekt *DataTable* v *northwindDataSet*, ki se obnaša kot izvor podatkov; nastavljen je na *NumProductsTable*.

Kliknimo še na labelo *numProducts* in razširimo lastnost *DataBinding*. Lastnost *Text* je nastavljena numProductsTableBindingSource – *NumProducts*.

Poglejmo še kodo, ki pripada obrazcu. Preklopimo pogled obrazca na *View – Code* in si oglejmo metodo *Produkti_Load*. Kodo je zgeneriral Visual Studio sam, izvede pa se seveda ob zagonu naše aplikacije, pred prvim odpiranjem obrazca.

```
private void Produkti_Load(object sender, EventArgs e)
{
```

```
    this.numProductsTableTableAdapter.Fill(this.northwindDataSet.NumProductsTable);
}
```

Ta stavek nam pove, kako naša labela pridobi podatke iz baze:

- adapter **numProductsTableTableAdapter** se poveže z bazo in z izvedbo SQL SELECT stavka vrne podatek o številu vrstic tabele *Products*
- ko se obrazec zažene, adapter **numProductsTableTableAdapter** nato to informacijo uporabi za napolnitev objekta *DataTable* z imenom **NumProductsTable**
- objekt **numProductsTableBindingSource** poveže stolpec *NumProducts* v objektu **NumProductsTable** (ki je tipa *DataSet*) z lastnostjo *Text* labele na obrazcu.

Končno poženimo našo aplikacijo. Na labeli na našem obrazcu se izpiše število vrstic tabele *Products* (če v tabeli nismo delali nikakršnih sprememb je ta številka enaka 77).

V zgornjem primeru smo za povezavo lastnosti gradnika *Label* in podatkovnim izvorom uporabili čarovnika. Lastnost *DataBinding* nekega gradnika pa lahko nastavimo tudi dinamično, tako da napišemo kodo, ki se bo izvedla v času izvajanja naše aplikacije. Lastnost *DataBinding* vsebuje tudi metodo *Add*, ki jo lahko uporabimo za povezavo lastnosti gradnika z podatkovnim izvorom. Na obrazec npr. postavimo še eno labelo in jo poimenujmo *mojaLabela*. Lastnost *Text* znotraj lastnosti *DataBindings* lahko sedaj povežemo s stolpcem *NumProducts* v *numProductsTableBindingSource* takole:

```
mojaLabela.DataBindings.Add("Text", numProductsTableBindingSource, "NumProducts");
```

Parametri, ki smo jih posredovali metodi *Add* so ime lastnosti gradnika, ki ga želimo povezati s podatkovnim izvorom, ime objekta, ki vsebuje podatke ki jih želimo povezati , tretji parameter pa je član, ki je trenutno zadolžen za oskrbo s podatki.

Metodo *Add* lahko uporabimo tudi za povezavo z lastnostjo drugih gradnikov, neodvisno od podatkovnih izvorov. Naslednji primer prikazuje povezavo lastnosti *Text* labele *mojaLabela* z lastnostjo *Text* gradnika *TextBox* z imenom *mojText*.

```
mojaLabela.DataBindings.Add("Text", mojText, "Text");
```

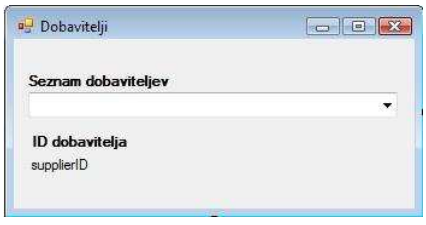
Ko tipkamo besedilo v gradnik *mojText*, se labela *mojaLabela* avtomatično spreminja skladno z vpisanim tekstom.

Kompleksno povezovanje s podatkovnim izvorom

V prejšnjih primerih smo pokazali, kako uporabimo preprosto podatkovno povezavo med lastnostjo nekega objekta in posamezno vrednostjo v podatkovnem izvoru. Če pa želimo prikazati večjo količino podatkov nekega podatkovnega izvora, pa je podatkovno povezovanje bolj zamotano, kompleksno.

V naslednjem primeru bomo podatkovno povezovanje uporabili za prikaz imen dobaviteljev iz baze podatkov v gradniku *ComboBox* in nato izpisali podatek *SupplierID* dobavitelja, ki ga bo uporabnik izbral.

Odprimo nov projekt in ga poimenujmo npr. *Northwind_ComboBox*. Na obrazec postavimo tri labele in en *ComboBox*. Lastnost *Text* obrazca spremenimo v *Dobavitelji*. Gradnik *ComboBox* na obrazcu poimenujmo *supplierList*, spodnjo labelo pa *supplierID*.

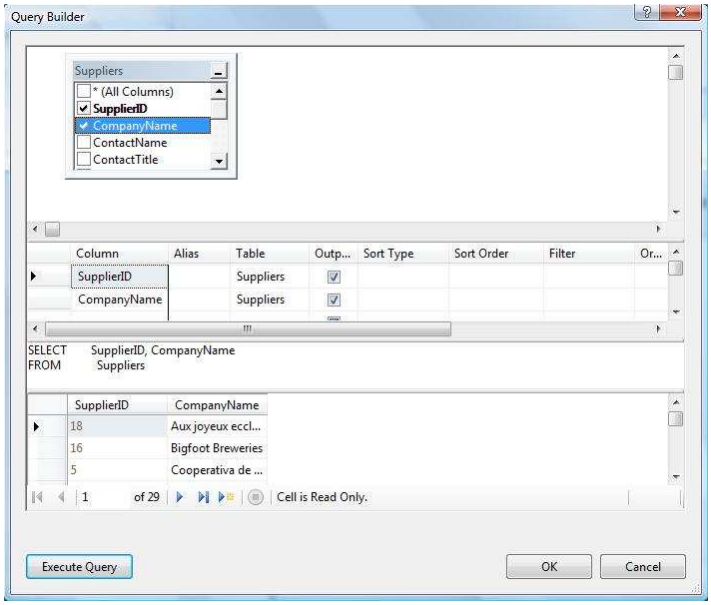


V meniju *Project* izberimo *Add New Item*, nato pa v oknu izberimo predlogo *DataSet*, jo poimenujmo *northwindDataSet* in kliknimo gumb *Add*. Prikaže se okno *DataSetDesigner*.

Z uporabo okna *Toolbox* dodajmo v *DataSet* nov *TableAdapter*. Odpre se okno *TableAdapter Configuration Wizard*. V oknu izberimo ustrezno povezavo - *ime_računalnika\sqlexpress.Northwind.dbo* (le to smo skreirali že v eni od prejšnjih vaj, če pa povezave še nimamo, jo skreiramo tako, kot smo to naredili v poglavju **Uporaba baze podatkov**). Izbiro povezave potrdimo s klikom na gumb *Next*.

Prikaže se okno *Save the connection string to the application configuration file* – konfiguracijo shranimo pod privzetim imenom *NorthwindConnectionString* in kliknimo na gumb *Next*. Odpre se okno *Choose a Command Type*, v katerem izberemo opcijo *Use SQL statements* in kliknimo *Next*.

V oknu *Enter a Statement page* kliknimo *Query Builder*. Odpreta se pogovorni okni *Query Builder* in *Add Table*, ki nam omogočata oblikovanje SELECT stavkov s pomočjo čarovnika. V oknu *Add Table* izberimo tabelo *Suppliers*, kliknimo *Add* in nato *Close*. Tabela *Suppliers* se doda v pogovorno okno *Query Builder*. Odkljukajmo vrstici *SupplierID* in *CompanyName*.



V spodnjem delu tega okna se prikaže SQL stavek, ki ustreza naši izbiri:

```
SELECT     SupplierID,  CompanyName
FROM       Suppliers
```

Če v oknu kliknemo gumb *Execute Query*, se nad gumbom prikaže rezultat poizvedbe.

Okno zaprimo s klikom na gumb *OK*, da se vrnemo v okno *TableAdapter Configuration Wizzard*. SELECT stavek se ob tem skopira v to okno.

Ker aplikacija, ki jo pišemo ne bo spreminjala dobaviteljevih podatkov (izvedli bomo le poizvedbo), kliknimo gumb *Advanced Options* in odstranimo kljukico v kontrolnih poljih *Generate Insert*, *Update* in *Delete*. Kliknimo *OK* in nato *Finish*.

V okno *DataSet Designer* se dodata dva nova razreda: nov razred *Suppliers* tipa *DataTable* in nov razred *SuppliersTableAdapter* tipa *DataTableAdapter*.

Končno moramo še povezati gradnik *ComboBox* z našo podatkovno tabelo *Suppliers*. Na obrazcu izberimo gradnik *ComboBox* in oknu *Properties* kliknimo lastnost *DataSource*. Prikaže se spustni meni v katerem razširimo *Other Data Sources*, razširimo *Project Data Sources*, razširimo *NorthwindDataSet* in kliknimo *Suppliers*. Gradnik *ComboBox* sm s tem povezali s podatkovno tabelo *Suppliers* in generirali nov objekt tipa *BindingSource*, ki se imenuje *suppliersBindingSource*, ter instanco razreda *SuppliersTableAdapter*.

Med tem, ko je gradnik *ComboBox* še kar izbran, nastavimo njegovo lastnost *DisplayMember* na *CompanyName* in lastnost *ValueMember* na *SupplierID*. Ko bomo projekt zagnali, bomo v *ComboBox*-u dobili seznam vseh dobaviteljev, ob izbiri poljubnega dobavitelja iz seznama pa nam bo na voljo vrednost polja *SupplierID*.

Ob izbranem gradniku *ComboBox* v oknu *Properties* kliknimo gumb *Events* in nato dvokliknimo v vrstico *SelectedIndexChenged*. V telo prikazane metode dopišimo stavek

```
if (supplierList.SelectedValue != null)
{
     //če izbrana vrednost ni prazna, se ID dobavitelja izpiše na labeli supplierID
     supplierID.Text = supplierList.SelectedValue.ToString();
}
```

Če sedaj aplikacijo poženemo, se ob izbiri poljubnega dobavitelja na labeli prikaže njegova identifikacijska številka.

Ažuriranje baze podatkov z uporabo objektov DataSet

Podatke iz podatkovne baze smo v prejšnjih primerih in vajah le prikazovali, v tem poglavju pa bomo pokazali, kako podatke v tabele dodajamo, jih ažuriramo ali brišemo. Pred tem pa moramo razmisliti o nekaterih potencialnih problemih, do katerih lahko pride pri takih opravilih in kako le-te odpravimo.

Podatkovne baze so namenjene za hkratno delo več uporabnikom, pri čemer je to število v večini primerov omejeno le na določeno število hkratnih povezav. V aplikaciji, ki pridobiva oz. prikazuje podatke iz baze, nikoli ne vemo v naprej, koliko časa se bo nek uporabnik listal po podatkih v neki tabeli, zaradi česar ni dobro, da je povezava z podatkovno bazo odprta samo nek določen čas. Boljši pristop pri delu z bazami je ta, da se uporabnik na bazo priklopi, prenese potrebne podatek v objekt *DataSet*, nato pa povezavo z bazo prekine. Uporabnik se nato premika po pridobljenih podatkih in naredi poljubne spremembe. Ko so spremembe gotove, se ponovno priključi na bazo in predloži spremembe, ki naj se izvedejo tudi v bazi podatkov. Pri tem seveda lahko nastopijo težave, kot npr. kaj naj se zgodi v primeru, da dva uporabnika zahtevata spremembo istega podatka, nova vrednost pa je pri obeh različna. Kateri podatek bo tem primeru shranjen v bazo? O reševanju takih in podobnih težav bo napisano v nadaljevanju.

Upravljanje s povezavami

V vseh dosedanjih primerih smo videli, da ko definiramo objekt *DataSet*, lahko določimo tudi povezavo, ki jo uporabimo za povezavo z bazo podatkov. To informacijo je potem uporabil objekt *TableAdapter* za pridobitev ustreznih podatkov in napolnitev objekta *DataSet*. Ko želimo izvesti metode *Fill* in *GetData*, koda, ki jo je generiral Visual Studio najprej pregleda stanje povezave s podatkovno bazo. Če je potrebna povezava že vzpostavljena, jo uporabi za pridobivanje podatkov, in jo na koncu operacije pusti odprto. V kolikor pa je povezava z bazo zaprta, jo metodi *Fill* in *GetData* odpreta, pridobita podatke in povezavo zapreta. V tem primeru se objekt *DataSet* torej obnaša kot *disconnected DataSet* in ne vzdržuje še naprej aktivne povezave z bazo. Taki (*disconnected DataSet*) objekti se torej obnašajo kot t.i. *data cache* v aplikacijah. Podatke v objektu *DataSet* lahko poljubno spreminjamo, kasneje pa ponovno odpremo povezavo z bazo in ji pošljemo narejene spremembe.

Povezavo z bazo lahko seveda odpremo tudi ročno (programsko) tako, da kreiramo objekt *SqlConnection*, določimo njegovo lastnost *ConnectionString*, nato pa pokličemo njegovo metodo *Open*. Z nastavitvijo lastnosti *Connection* lahko nato odprto povezavo povežemo z objektom *TableAdapter*. Naslednji primer prikazuje, kako se priklopimo na bazo in napolnimo objekt *SuppliersDataTable*. V tem primeru bo ostala povezava z bazo odprta tudi po zaključku metode *Fill*.

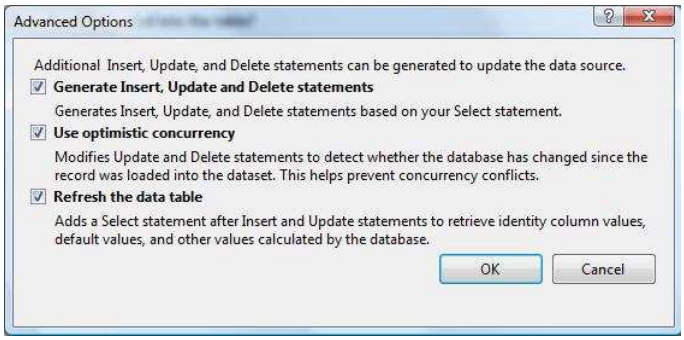
```
SqlConnection dataConnection = new SqlConnection();
dataConnection.ConnectionString = "Integrated Security=true;Initial Catalog=Northwind; "+
                                  "Data Source=ime_Računalnika\\SQLExpress";
dataConnection.Open();
suppliersTableAdapter.Connection = dataConnection;
suppliersTableAdapter.Fill(northwindDataSet.Suppliers);
```

Razen če zato nimamo res dobrega razloga, moramo povezavo s podatkovno bazo prekiniti takoj, ko so podatki pridobljeni. Pustimo metodama *Fill* in *GetData* da skrbita za odpiranje in zapiranje povezav.

Obravnava istočasnega posodabljanja podatkov več uporabnikov (multi-user updates)

Pri hkratnem dostopanju, ažuriranju ali posodabljanju podatkov lahko pride do nepredvidenih težav. Za reševanje tega problema sta možna vsaj dva različna pristopa. Vsak od njiju ima svoje prednosti in svoje slabosti.

Prvi pristop k reševanju problemov pri hkratnem ažuriranju je tehnika, ki zajema opcijo *Use optimistic concurrency option* v pogovornem oknu *Advanced Options* pri uporabi čarovnika *TableAdapter Configuration*.



Če ta opcija NI izbrana, bodo vrstice podatkov, ki smo jih pridobili v objekt *DataSet* zaklenjene, saj na ta način drugim uporabnikom preprečimo, da bi med tem te podatke spreminjali. Tak način povezovanja s podatkovno bazo oz. zaklepanja je poznan kot *pessimistic concurrency*, saj nam zagotavlja, da spremembe, ki jih bomo napravili v podatkih ne bodo prišle v kolizijo s spremembami, ki jih bodo nad istimi podatki napravili drugi uporabniki. Slaba stran tega načina je seveda ta, da med našim ažuriranjem drugi uporabniki nimajo dostopa do teh podatkov. Če smo v našo aplikacijo pritegnili veliko količino podatkov, ažuriramo pa le njihov manjši del, smo ostalim uporabnikom preprečili možnost ažuriranja celotne količine pridobljenih podatkov, ne le tistih, ki jih potrebujemo. Pa še eno slabo točko ima tak način dela: zaklepanje podatkov zahteva, da povezava, ki smo uporabili za pridobivanje podatkov, ostane ves čas odprta. Če torej uporabimo način *pessimistic concurancy*, lahko tvegamo kar precejšnje zmanjšanje konkurenčnosti naših povezav. Vendar pa je tak način najbolj enostaven, saj nam ni potrebno pisati nobene kode, ki bi preverjala eventuelna posodabljanja ostalih uporabnikov pred ažuriranjem podatkov.

Drug način povezovanja pa se imenuje *optimistic concurrency* – v oknu *Advanced Options* v tem primeru izberemo opcijo *Use optimistic cuncurrency*. Podatki v tem primeru niso zaklenjeni, povezavo z bazo pa lahko zapremo takoj ko pridobimo vse podatke. Slaba stran takega načina pa je, da moramo tem primeru napisati kodo, s katero se prepričamo, ali so mogoče spremembe, ki jih uporabnik naredi v podatkih, v koliziji s tistimi, ki jih je naredil nek drug uporabnik. Pisanje takšne kode pa je lahko precej težavno, težavno pa je tudi razhroščevanje. Objekt *TableAdapter*, ki ga zgenerira *TableAdapter Configuration Wizard*, sicer rešuje večji del teh težav, a vseeno moramo biti pripravljeni na obdelavo izjem, do katerih lahko pride pri takšnih konfliktih. Več o reševanju takih izjem bo zapisano na koncu tega poglavja.

Uporaba objekta DataSet v povezavi z gradnikom DataGridView

Med najpogostejšimi gradniki, v katerih prikazujemo vsebino neke tabele iz baze podatkov je gradnik *DataGridView*. V naslednjem primeru bo mo v tem gradniku prikazali vsebino tabele *Employees* iz podatkovne baze *Northwind*.

Kreirajmo nov projekt in ga poimenujmo *Zaposleni*. Lastnost *Text* obrazce spremenimo v *Seznam zaposlenih v podjetju Northwind*. V projekt dodajmo podatkovni izvor: meni *Project*, *Add New Item*, izberimo *DataSet* in ga poimenujmo *DSNorthwind*, nato pa kliknimo gumb *Add*. Prikaže se okno *DataSetDesigner*.

Z uporabo okna *Toolbox* dodajmo v *DataSet* nov *TableAdapter*. Odpre se okno *TableAdapter Configuration Wizard*. V oknu izberimo ustrezno povezavo - *ime_računalnika\sqlexpress.Northwind.dbo*. Izbiro povezave potrdimo s klikom na gumb *Next*.

Prikaže se okno *Save the connection string to the application configuration file* – konfiguracijo shranimo pod privzetim imenom *NorthwindConnectionString* in kliknimo na gumb *Next*. Odpre se okno *Choose a Command Type*, v katerem izberemo opcijo *Use SQL statements* in kliknimo *Next*.

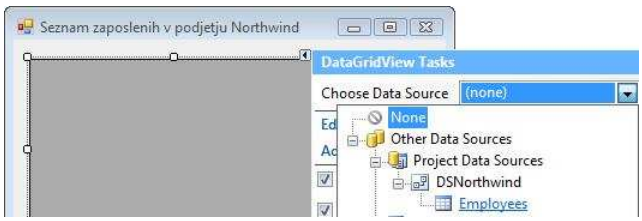
V oknu *Enter a Statement page* kliknimo *Query Builder*. Odpreta se pogovorni okni *Query Builder* in *Add Table*, ki nam omogočata oblikovanje SELECT stavkov s pomočjo čarovnika. V oknu *Add Table* izberimo tabelo *Empoyees*, kliknimo *Add* in nato *Close*. Tabela *Empoyees* se doda v pogovorno okno *Query Builder*. Odkljukajmo vrstico *All Columns* in izbiro potrdimo s klikom na gumb *OK*. Kliknimo še na gumb *Advanced*

Options in odkljukajmo vse tri možne opcije. S klikom na gumb *OK* okno zapremo, nato pa še s klikom na gumb *Finish* zaključimo nastavitve našega *TableAdapter*-ja.

Podatkovni izvor je tako pripravljen, podatke moramo le še prikazati na gradniku *DataGridView*. Na obrazec zato postavilo gradnik *DataGridView*. Sedaj je potrebna le še povezava med našim podatkovnim izvorom in gradnikom *DataGridView*. Le-to pa lahko naredimo na dva načina: s pomočjo lastnosti *DataSet* gradnika *DataGridView*, ali pa čisto programsko. Pokažimo oba načina:

Povezava gradnika DataGridView s podatkovnim izvorom preko lastnosti DataSet

Kliknimo na trikotniček v desnem zgornjem kotu gradnika *DataGridView*. Razširimo *OtherDataSource*, razširimo *ProjectDataSources*, razširimo *DSNorthwind* in zberimo *Employees*. Če sedaj projekt poženemo, bo vsebina gradnika *TableGridView* napolnjena s podatki o zaposlenih v podjetju *Northwind*.



Povezava gradnika DataGridView s podatkovnim izvorom preko kode - programsko

Preklopimo na pogled obrazca *View Code* in najprej deklariramo novo instanco našega podatkovnega izvora:

```
private DSNorthwind dSNorthwind = new DSNorthwind();//Nov objekt za naš podatkovni izvor
```

Povezavo s podatkovnim izvorom bomo zapisali v dogodek *Load* našega obrazca

```
private void Form1_Load(object sender, EventArgs e)
{
    //Napovemo novo instanco objekta DSNorthwindTableAdapters.EmployeesTa z imenom zaposleniTA
    DSNorthwindTableAdapters.EmployeesTableAdapter zaposleniTA;
    //Nov objekt tudi dejansko ustvarimo
    zaposleniTA = new DSNorthwindTableAdapters.EmployeesTableAdapter();
    /*objekt TableAdapter z imenom zaposleniTA napolnimo s podatki iz podatkovne tabele. Ko se
      bo objekt EmployeesTableAdapter napolnil s podatki, bo povezava z bazo avtomatsko
      prekinjena, saj do tega trenutka ni bilo vzpostavljene nobene povezave!*/
    zaposleniTA.Fill(this.dSNorthwind.Employees);
    //ustvarimo nov objekt tipa BindingSource za prenos podatkov iz TableAdapterja
    BindingSource employeesBS = new BindingSource(this.dSNorthwind, "Employees");
    //povežemo lastnost DataSource gradnika DataGridView z objektom employeesBS
    dataGridView1.DataSource = employeesBS;
}
```

Ko aplikacijo poženemo, se vsebina gradnika *DataGridView* napolni s podatki o zaposlenih v podjetju *NorthWind*.

Podatke, ki so prikazani v posameznih stolpcih lahko urejamo po velikosti oz. po abecednem redu (klikamo v ustrezno polje v naslovni vrstici podatkov). Pri prvem kliku so podatki urejeni v naraščajočem vrstnem redu, pri drugem kliku pa v padajočem.

Če kliknemo v katerokoli celico gradnika *DataGridView* (razen v celico *EmployeesID*) lahko preko tipkovnice v polje vnesemo novo vsebino. V primeru, da v celico, ki sicer vsebuje numerične podatke, vnesemo nek alfanumeričen string, dobimo pri poskusu premika na drugo celico obvestilo o napaki. Okno z obvestilom zaprimo in popravimo vsebino celice (ali pa pritisnimo tipko *Escape* za preklic vnosa). Obvestilo o napaki je privzeto, lahko pa ga nadomestimo s poljubnim svojim obvestilom o napaki – to storimo tako, da napišemo svoj *DataError* dogodek, ki je sestavni del dogodkov gradnika *DataGridView*, npr.:

```
private void dataGridView1_DataError(object sender, DataGridViewDataErrorEventArgs e)
{
    MessageBox.Show("Napačen vnos podatkov!");
}
```

Po podatkih v tabeli se lahko premikamo s pomočjo horizontalnega in vertikalnega drsnika, ki se pojavita na robovih gradnika *DataGridView*. Če se premaknemo za nekaj polj bolj v desno, bomo v eni od celic opazili tudi sliko, ki predstavlja fotografijo zaposlenega. V posamezni celici gradnika *DataGridView* lahko napreč prikažemo prav vsak tip podatka.

Premaknimo se na dno tabele, torej na konec podatkov. Pojavi se prazna vrstica, na njenem začetku pa se pojavi zvezdica. V prazno vrstico lahko dodamo nove podatke, pri čemer se nova številka zaposlenega (*EmployeesID*) generira avtomatično, saj gre za ključ v tabeli *Empoyees*.

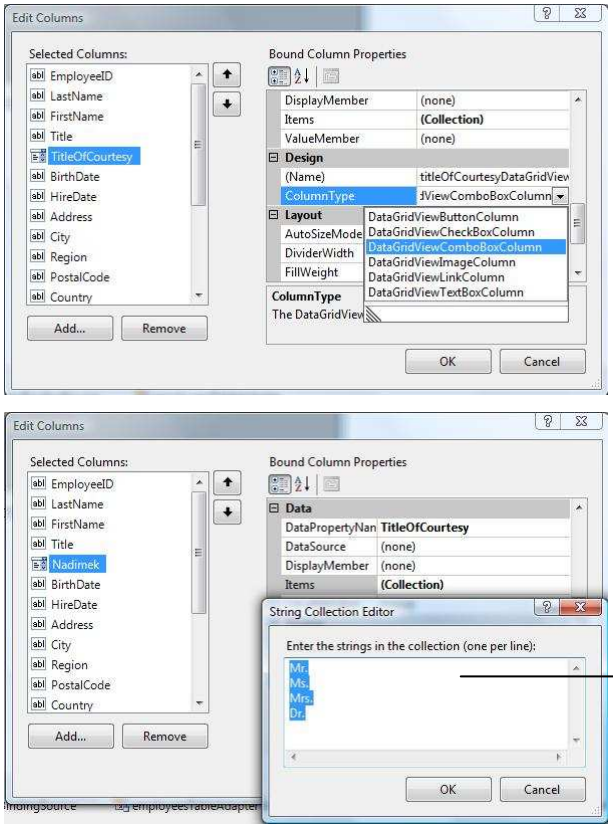
Če kliknemo v sivo polje na levi strani katerekoli vrstice, se obarva celotna vrstica. Ob pritisku na gumb *Delete* se celotna vrstica pobriše.

Ko z delom v gradniku *DataGridView* končamo, obrazec zaprimo. **Spremembe, ki smo jih v tabeli naredili se ne bodo shranile**, saj za potrebe shranjevanja podatkov še nismo napisali ustrezne kode.

Gradnik *DataGridView* vsebuje cel kup lastnosti, s pomočjo katerih oblikujemo vizuelni izgled gradnika in prikazanih podatkov. Med lastnostmi ima še poseben pomen zelo uporabna lastnost *Columns*, ki nam omogoča nastavljanje velikega števila podlastnosti tega gradnika: izgled, velikost celic, barve, izgled naslovne vrstice in celo nastavljanje izgleda posamezne celice (kot tekstovno polje, kot navaden gumb, kot *ComboBox*, …).

V gradniku DataGridView nastavimo lastnosti tako, da bomo v zagnanem projektu lahko polje *TitleOfCourtesy* spremenili/ažurirali s pomočjo spustnega seznama, polje *BirthDate* pa z izbiro datuma na koledarčku.

Spustni seznam za izbiro nadimka pripravimo takole:



- na obrazcu izberimo gradnik *DataGridView* (ime gradnika je *dataGridView1*)
- v oknu *Properties* kliknimo *Columns*
- izberimo polje *TitleOfCourtesy*
- Lastnost *ColumnType* nastavimo na *DataGridViewComboBoxColumn*
- Lastnost *HeaderText* nastavimo na *Nadimek* – spremenili smo napis na vrhu stolpca
- Kliknimo lastnost *Items* in v okno vnesimo vrstice, ki so na sliki

Ko sedaj projekt poženemo, lahko vsebino stolpca *Nadimek* (ki dejansko pomeni vsebino polja *TitleOfCourtesy*) spremenimo le s pomočjo spustnega seznama, ki se prikaže pod to celico.

Poglejmo še, kako lahko spremenimo vsebino datumskega polja s pomočjo koledarčka. V ta namen pripravimo nov obrazec in ga poimenujmo *FKoledar*. Na obrazec FKoledar nato postavimo gradnik *MonthCalendar* (ime gradnika naj ostane kar *monthCalendar1*). Ker želimo imeti do tega gradnika dostop iz prejšnjega obrazca, moramo ta gradnik narediti javno dostopen (v oknu *Solution Explorer* kliknimo na znak + pred obrazcem *FKoledar* in nato izberimo okno *FKoledar.Designer.cs*. Na dnu tega okna popravimo označevalec *private* za ta gradnik v *public*

```
public System.Windows.Forms.MonthCalendar monthCalendar1;
```

Na obrazec *FKoledar* postavimo še gumb z napisom *OK* (lastnost *DialogResult* nastavimo na *OK*) in gumb z napisom *Prekliči* (*DialogResult* nastavimo na *Cancel*) in vse skupaj shranimo. Preklopimo ponovno na obrazec, na katerem imamo gradnik *DataGridView* in ta gradnik tudi izberimo. V oknu *Properties* preklopimo na seznam dogodkov, dvokliknimo v dogodek *CellClick* in v telo tega dogodka zapišimo naslednjo kodo:

```
//če kliknemo v stolpec BirthDate, se prikaže koledarček
if (dataGridView1.Columns[e.ColumnIndex].DataPropertyName == "BirthDate")
{
    //nova instanca obrazca FKoledar, ki vsebuje gradnik MonthCalendar
    FKoledar koledar=new FKoledar();
    //datum na obrazcu nastavimo glede na datum v izbrani celici
    koledar.monthCalendar1.SetDate((DateTime)dataGridView1[e.ColumnIndex, e.RowIndex].Value);
    //obrazec FKoledar odpremo modalno
    if (koledar.ShowDialog() == DialogResult.OK) //če uporabnik klikne gumb OK
    {
       //datum v celici popravimo glede na izbrani datum v koledarčku
       dataGridView1[e.ColumnIndex, e.RowIndex].Value = koledar.monthCalendar1.SelectionEnd;
    }
}
```

Ob kliku v celico *BirthDate* se bo odprl obrazec s koledarčkom. Na koledarček bo prenesen datum iz celice, ob izbiri novega datuma na koledarčku in kliku na gumb *OK*, pa se bo v celico prenesel novi datum.

Vaja:

Vaja prikazuje, kako lahko podatke iz neke tabele v podatkovni bazi prikažemo v gradniku *DataGridView* izključno s pisanjem kode. Kreirajmo nov projekt in na obrazec *Form1* postavimo dva gradnika *DataGridView* in ju poimenujmo *DataGridViewArtikli* in *DataGridViewStranke*. Preklopimo na *Code View* in nato kodo v *Form1* dopolnimo takole:

```
using System.Data.SqlClient; //Ker bo povezava programska, ne pozabimo na using stavek
```

```
//Deklarirajmo nov objekt SQLConnection za povezavo z SQL bazo
SqlConnection dataConnection = new SqlConnection();
//Napoved adapterjev za posamezno tabelo iz baze
SqlDataAdapter DAArtikli,DAStranke;
//Napoved objekta dataset, ki bo vseboval datke iz baze Northwind
private DataSet dsNorthwind;

public Form1()
{
    InitializeComponent();
    try
    {
        string izvor="Integrated Security=true;Initial Catalog=Northwind;Data Source = "+
                     "imeRacunalnika\\SQLExpress";
        string SQL="SELECT * FROM Products";
        dataConnection=new SqlConnection(izvor); // povezava z bazo
        DAArtikli = new SqlDataAdapter(SQL,dataConnection);
        DAStranke = new SqlDataAdapter(SQL, dataConnection);

        dsNorthwind = new DataSet();
        //s pomočjo metode Fill adapterja DAArtikli potegnemo v dsArtikli prvo tebelo
        DAArtikli.Fill(dsNorthwind, "Products");

        SQL = "SELECT * FROM Customers";
        DAStranke = new SqlDataAdapter(SQL, dataConnection);
        //s pomočjo metode Fill adapterja DAArtikli potegnemo v dsArtikli še drugo tebelo
```

```
        DAStranke.Fill(dsNorthwind, "Customers");
        //V gradniku tableGridViewArtikli prikažemo prvo tabelo iz našega dsArtikli -indeks 0!
        dataGridViewArtikli.DataSource = dsNorthwind.Tables[0];
        //V gradniku tableGridViewStranke prikažemo drugo tabelo iz našega dsArtikli-indeks 1!
        dataGridViewStranke.DataSource = dsNorthwind.Tables[1];
    }
    catch
    {
        MessageBox.Show("Napaka pri dostopu do baze podatkov!");
    }
}
```

Ko projekt poženemo, bomo v gradnikih *DataGridView* zagledali podatke iz tabele *Products* in iz tabele *Customers*.

Na obrazec postavimo gumb. Ob kliku na gumb želimo obdelati tabelo Products tako, da ugotovimo in na koncu izpišemo skupno število izdelkov na zalogi. Ime polja, ki vsebuje podatke o številu posameznih izdelkov je UnitsInStock, zaporedna številka polja v tabeli pa je 6.

```
private void button1 Click(object sender, EventArgs e)
{
    try
    {
        string izvor = "Integrated Security=true;Initial Catalog=Northwind;Data Source ="+
                       "imeRacunalnika\\SQLExpress";
        string SQL = "SELECT * FROM Products";
        dataConnection.Open();
        SqlCommand dataCommand = new SqlCommand();
        dataCommand.Connection = dataConnection;
        //kreiramo poizvedbo
        dataCommand.CommandText = "SELECT * FROM Products";
        SqlDataReader dataReader = dataCommand.ExecuteReader();
        int skupaj = 0;
        //ugotovimo skupno število artiklov na zalogi
        while (dataReader.Read())//dokler obstajajo podatki
        {
           try
           {
              //število komadov je v polju z indeksom 6. Metoda GetValue vrne tip Object
              object komadov = dataReader.GetValue(6);
              skupaj = skupaj + Convert.ToInt32(komadov);
           }
        catch { }
        }
        //rezultat prikažemo v sporoči oknu
        MessageBox.Show("Skupaj na zalogi: " + skupaj.ToString());
        dataReader.Close();//zapremo podatkovni tok
        dataConnection.Close();//Obvezno zapremo povezavoz bazo
    }
    catch
    {
        MessageBox.Show("Napaka pri dostopu do baze podatkov!");
    }
```

Kontrola uporabnikovih vnosov v gradnik DataGridView

Pred shranjevanjem podatkov nazaj v bazo podatkov, se moramo prepričati, da so spremembe, ki jih je napravil uporabnik, veljavne oz. pravilne. V celice gradnika *DataGridView* lahko namreč uporabnik vtipka karkoli, zato moramo s programsko kodo poskrbeti, da bodo vnosi smiselni in se tako zavarovati pred napakami pri shranjevanju podatkov v bazo.

Za prikaz nekaterih kontrol uporabnikovih vnosov bomo zopet uporabili bazo *Northwind*. Kreirajmo nov projekt in ga imenujmo *Produkti*. V projekt dodajmo podatkovni izvor: meni *Project*, *Add New Item*, izberimo *DataSet* in ga poimenujmo *DSNorthwind*, nato pa kliknimo gumb *Add*. Prikaže se okno *DataSetDesigner*.

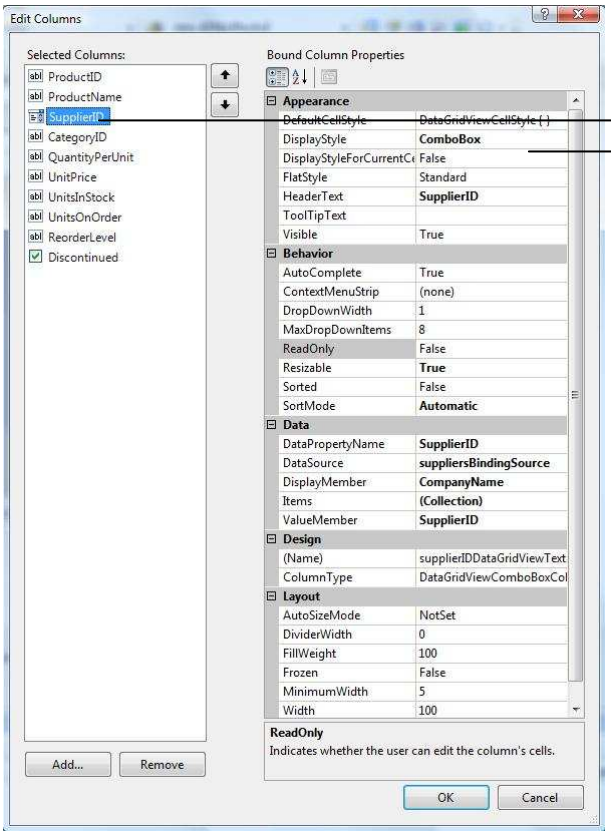
Z uporabo okna *Toolbox* dodajmo v *DataSet* nov *TableAdapter*. Odpre se okno *TableAdapter Configuration Wizard*. V oknu izberimo ustrezno povezavo - *ime_računalnika\sqlexpress.Northwind.dbo*. Izbiro povezave potrdimo s klikom na gumb *Next*.

Prikaže se okno *Save the connection string to the application configuration file* – konfiguracijo shranimo pod privzetim imenom *NorthwindConnectionString* in kliknimo na gumb *Next*. Odpre se okno *Choose a Command Type*, v katerem izberemo opcijo *Use SQL statements* in kliknimo *Next*.

V oknu *Enter a Statement page* kliknimo *Query Builder*. Odpreta se pogovorni okni *Query Builder* in *Add Table*, ki nam omogočata oblikovanje SELECT stavkov s pomočjo čarovnika. V oknu *Add Table* izberimo tabelo *Products*, kliknimo *Add* in nato *Close*. Tabela *Products* se doda v pogovorno okno *Query Builder*. Odkljukajmo vrstico *All Columns* in izbiro potrdimo s klikom na gumb *OK*. Kliknimo še na gumb *Advanced Options* in odkljukajmo vse tri možne opcije. S klikom na gumb *OK* okno zapremo, nato pa še s klikom na gumb *Finish* zaključimo nastavitve našega *TableAdapter*-ja. Na enak način dodajmo v naš *DataSet* še tabelo *Supplier*.

Podatkovni izvor je tako pripravljen, podatke moramo le še prikazati na gradniku *DataGridView*. Na obrazec zato postavilo gradnik *DataGridView* in ga poimenujmo *productsGrid*. Kliknimo na trikotniček v desnem zgornjem kotu gradnika *DataGridView*. Razširimo *OtherDataSource*, razširimo *ProjectDataSources*, razširimo *DSNorthwind* in zberimo *Products*. Če sedaj projekt poženemo, bo vsebina gradnika *TableGridView* napolnjena s podatki iz tabele *Products*.

Vrnimo se v *Design View* našega obrazca in izberimo gradnik *DataGridView*. V oknu Properties kliknimo lastnost *Columns* in izberimo najprej polje *ProductID*. Prepričajmo se, da je lastnost *ReadOnly* v *Bound Column Properties* nastavljena na *true*. Vrednosti tega stolpca se namreč generira avtomatično, zato uporabniku ne smemo dopustiti, da ga spreminjal sam.



Izberimo sedaj polje *SupplierID* in v *Bound Column Properties* nastavimo lastnosti tako, kot kaže slika. Najprej nastavimo lastnost *ColumnType* na *supplierIDDataGridViewColumns*, šele nato pa lastnost *DisplayStyle* (ta lastnost je na začetku nevidna). Nastavimo še *DataSource* na *suppliersBindingSource*, *DisplayMember* na *CompanyName* in ValueMember na *SupplierID*. Okno zarimo s klikom na *OK* in ponovno poženimo projekt. V stolpcu *SupplierID* so sedaj gradniki *ComboBox*. Če kateregakoli od njih odpremo dobimo spustni seznam vseh dobaviteljev – izberemo lahko kateregakoli in s tem zamenjamo prejšnjega. Na ta način smo dosegli, da se uporabnik pri vnosu dobavitelja nikakor ne more zmotiti in vnesti *ID* dobavitelja, ki ne obstaja.

Na istem primeru pokažimo še obdelavo nekaterih drugih dogodkov, s katerimi lahko kontroliramo uporabnikove vnose in se tem zavarujemo pred napačnimi vnosi podatkov. Ti dogodki so *CellValidating*, *CellEndEdir* in *DataError*.

Izberimo gradnik *DataGridView* imenovan *productsGrid* in v oknu *Properties* kliknimo gumb Events. Dvokliknimo v dogodek *CellValidating* in Visual C# nam zgenerira ogrodje te odzivne metode. Zapišimo naslednjo kodo:

```
int zacasna;
productsGrid.Rows[e.RowIndex].ErrorText = "";

if ((productsGrid.Columns[e.ColumnIndex].DataPropertyName == "UnitsInStock") ||
    (productsGrid.Columns[e.ColumnIndex].DataPropertyName == "UnitsOnOrder") ||
    (productsGrid.Columns[e.ColumnIndex].DataPropertyName == "ReorderLevel"))
{
    if (!int.TryParse(e.FormattedValue.ToString(), out zacasna) || zacasna < 0)
    {
        productsGrid.Rows[e.RowIndex].ErrorText = "Vrednost mora biti nenegativno število";
        e.Cancel = true;
    }
}
```

Namen te metode je zagotoviti, da bo uporabnik v polja *UnitsInStock*, *UnitsOnOrder* in *ReorderLevel* zagotovo vnesel nenegativna cela števila. Drugi parameter te metode, parameter ***e***, je objekt tipa *DataGridViewCellValidatingEventArgs*. Vsebuje številne lastnosti, ki jih lahko uporabimo npr. za ugotavljanje, katera celica je bila ažurirana: *e.ColumnIndex* vsebuje številko stolpca, *e.RowIndex* pa številko vrstice (prva vrstica in prvi stolpec imata indeks enak 0). V prvem *if* stavku zgornje kode preverjamo, če smo v enem od treh celoštevilskih stolpcev/polj. Upoštevati moramo, da razred *DataGridView* vsebuje zbirko *Columns*, ki hrani informacije o vsakem prikazanem stolpcu. Vrednost *e.ColumnIndex* je uporabljena kot indeks elementa iz te zbirke, vrednost lastnosti *DataPropertyName* pa smo uporabili za identifikacijo ustreznega stolpca.

Metoda *int.TryParse* je zelo koristna v primerih ugotavljanja ali nek string vsebuje vrednost, ki se lahko konvertira v celo število. Metoda vrne *true*, če je pretvarjanje uspešno, obenem pa v svoj drugi parameter (v našem primeru *zacasna*), ki je klican po referenci zapiše pretvorjeno vrednost. Drugi if stavek v zgornji kodi torej s pomočjo metode try.Parse preveri, ali je uporabnik v izbrano celico vnesel celoštevilsko vrednost. Če temu ni tako (uporabnik je vnesel npr. *string*, ki vsebuje črke, ali pa je vnesena vrednost negativna), se v lastnost *ErrorText* trenutne vrstice v tabeli zapiše ustrezno obvestilo o napaki. Na začetku te vrstice se v tem primeru prikaže ikona s klicajem in če na ikono postavimo kazalnik miške, se nam pod njo izpiše obvestilo o napaki (v našem primeru tekst `"Vrednost mora biti nenegativno število"`). Z naslednjim stavkom (**`e.Cancel = true;`**) pa poskrbimo, da se uporabnik nikakor ne more premakniti v drugo celico vse dotlej, dokler v celico ne vnese pravilnega podatka, ali pa celoten vnos ne prekliče s tipko *Esc*.

Za gradnik *DataGridView* zapišimo še dogodek *CellEndEdit*. Koda je naslednja:

```
productsGrid.Rows[e.RowIndex].ErrorText = "";
```

Ta metoda se izvede, ko je uporabnikov vnos veljaven in se uporabnik premakne v drugo celico. S tem stavkom enostavno pobrišemo vsa obvestila o napaki zaradi nepravilnega vnosa podatkov v trenutno izbrano celico.

Dvokliknimo še dogodek *DataError* gradnika *DataGridView* in v ogrodje prikazane metode zapišimo stavka:

```
productsGrid.Rows[e.RowIndex].ErrorText="Napačen vnos. Poskusite ponovno!";
e.Cancel = true;
```

Dogodek *DataError* ujame prav vse napake, ki nastanejo pri preverjanju uporabnikovega vnosa v katerokoli celico (predvsem velja to za celice, katerih vneseno vsebino nismo preverjali preko imena celice, tako kot v zgornjem primeru). Vsebina e metode poskrbi za splošno obvestilo uporabniku, obenem pa prepreči, da bi se uporabnik kljub napačnemu vnosu premaknil iz trenutne celice.

Poženimo sedaj naš projekt in poizkusimo v poljubne celice vnesti poljubno vsebino. Če vneseni tekst ne ustreza celici, katere vsebino vnašamo, bomo dobili obvestilo o napaki. Iz take celice se lahko premaknemo le če vnesemo pravilno vsebino, ali pa če vnos prekličemo s tipko *Esc*.

Ažuriranje uporabnikovih vnosov z uporabo objekta DataSet

Ko smo se prepričali, da so naši spremenjeni oz. dodani podatki res veljavni oz. pravilni, jih poskušamo shraniti nazaj v bazo.

Spremembe podatkov, narejene z uporabo gradnika *DataGridView* , se avtomatično skopirajo nazaj v objekt *DataSet*, ki igra vlogo podatkovnega izvora. Shranjevanje spremenjenih oz. dodanih podatkov vključuje ponoven priklop na bazo, izvedbo ustreznih SQL INSERT, UPDATE in DELETE stavkov, in nato odklop od baze. Pri teh operacijah lahko pride do raznih napak, na obdelavo katerih pa moramo biti pripravljeni. Razred *TableAdapter*, ki smo ga ustvarili za naš *DataSet* vsebuje kar nekaj metod, ki nam pri tem zelo pomagajo.

Tipične postopke, potrebne pri shranjevanju podatkov nazaj v bazo bomo prikazali kar na konkretnem primeru iz prejšnjega projekta *Produkti*. Na glavni obrazec postavimo gumb in mu spremenimo lastnosti *Text* v *Shrani* in lastnost *Name* v *shrani*. Nato dvokliknimo na ta gumb in Visual C# nam zgenerira ogrodje metode *shrani_Click*. V telo te metode zapišimo varovalni blok z naslednjo kodo:

```
try
{
     NorthwindDataSet sprememe = (NorthwindDataSet)northwindDataSet.GetChanges();
     if (sprememe == null)
     {
         return;
     }
     //Kontrola napak

     //Če napak ni, ažuriramo bazo, sicer pa obvestimo uporabnika o napakah

}
catch (Exception ex)
{
      MessageBox.Show("Napaka: " + ex.Message, "Napake", MessageBoxButtons.OK,
                      MessageBoxIcon.Error);
      northwindDataSet.RejectChanges();
}
```

Na začetku smo za ustvarjanje novega objekta razreda *NorthwindDataSet* (objektu smo dali ime *spremembe*) uporabili metodo *GetChanges()*. Ta metoda v objekt *spremembe* zapiše le tiste vrstice, ki so bile spremenjene. Eksplicitna konverzija `(NorthwindDataSet)northwindDataSet.GetChanges()` je nujno potrebna, ker metoda *GetChanges* vrača objekt tipa *DataSet*. Metode GetChanges nam sicer ne bilo potrebno uporabiti, je pa shranjevanje nazaj v bazo na ta način hitrejše, saj metodam za shranjevanje v tem primeru ni potrebno preverjati za vsako vrstico posebej, ali je bila le-ta spremenjena. Če ni bilo v gradniki *DataGridView* narejene nobene spremembe, potem ni nobene spremembe tudi v objektu *spremembe* – ta dobi vrednost *null* in metoda se zaključi. Če pa so spremembe bile, se izvede kontrola eventuelnih napak v podatkih, nakar se baza ažurira (ta del kode sledi v nadaljevanju). Če pride do napake med ažuriranjem baze, se bo uporabniku prikazalo sporočilno okno, vse dosedanje spremembe v bazi podatkov pa bodo zaradi uporabe metode *RejectChanges* preklicane.

Dodajmo sedaj še kodo za kontrolo napak, s katero v zgornji kodi nadomestimo vrstico `//Kontrola napak`.

```
DataTable dt = sprememe.Tables["Products"];
DataRow[] NapacneVrstice = dt.GetErrors();
```

V prvem stavku smo v objekt *dt* tipa *DataTable* zapisali vse spremembe, ki so bile shranjene v objektu *spremembe*. Metoda *GetErrors* vrne tabelo vseh vrstic iz te tabele, v katerih obstaja ena ali več napak v veljavnosti vnesenih podatkov. V primeru, da ni nobene napake, metoda *GetErrors* vrne prazno tabelo.

Vrstico `//Če napak ni, ažuriramo bazo, sicer pa obvestimo uporabnika o napakah` sedaj nadomestimo s stavki:

```
if (NapacneVrstice.Length==0)
{
   //Ažuriranje baze
}
else
{
   //Poiščemo napake in obvestimo uporabnika
```

```
}
```

Obstaja kar nekaj strategij za obveščanje uporabnika o napakah. Ena izmed najbolj uporabnih je ta, da napišemo kodo, ki zgenerira seznam prav vseh napak in le-te izpiše uporabniku v nekem sporočilnem oknu.

Nadomestimo kodo `//Poiščemo napake in obvestimo uporabnika` z naslednjimi stavki:

```
string errorMsg=null;
foreach (DataRow vrstica in NapacneVrstice)
{
    foreach (DataColumn stolpec in vrstica.GetColumnsInError())
    {
         errorMsg+=vrstica.GetColumnError(stolpec)+"\n";
    }
}
MessageBox.Show("Napake v podatkih: "+errorMsg,"Prosim popravite
                        podatke!",MessageBoxButtons.OK,MessageBoxIcon.Error);
```

Ta koda pregleda vse vrstice v tabeli *NapacneVrstice*. V vsaki od teh vrstic je zato po ena ali več napak. Metoda *GetColumnsInError* vrne zbirko, ki vsebuje vse stolpce z napačnimi podatki, metoda *GetColumnError* pa vrne obvestilo o napaki za posamezen stolpec. Vsako obvestilo o napaki je zatem dodano v string *errorMsg*. Ko so pregledane vse vrstice in preverjeni vsi stolpci, aplikacija v sporočilnem oknu izpiše skupno obvestilo o vseh napakah hkrati. Uporabnik naj bi s pomočjo te informacije popravil napačne vnose in nato poskusil s ponovnim shranjevanjem.
Če pa so vsi vneseni podatki v redu, jih končno lahko pošljemo v bazo podatkov. Kodo `//Ažuriranje baze` nadomestimo s stavki:

```
int steviloVrstic = productsTableAdapter.Update(spremembe);
MessageBox.Show("Ažuriranih vrstic " + steviloVrstic, "Shranjevanje uspešno");
northwindDataSet.AcceptChanges();
```

S pomočjo metode *Update* našega *productTableAdapter*-ja so spremenjeni podatki poslani v bazo podatkov. Ko so spremembe v bazi dejansko izvedene, smo v sporočilnem oknu uporabnika obvestili o številu izvedenih sprememb in metoda *AcceptChanges* te spremembe označi kot dejansko opravljene tudi v objektu *DataSet*.

- POZOR: V primeru, da je nek drug uporabnik spremenil vrstice, v katerih smo naredili spremembe tudi mi, bo metoda *Update* te spremembe zaznala, in dobili bomo ustrezno obvestilo o napaki. Razvijalec aplikacije se mora v takem primeru odločiti, kako bo obdelal to situacijo v svoji aplikaciji: uporabniku bo npr. ponudil možnost, da podatke kljub vsemu shrani in s tem prekrije spremembe, ki jih je napravil nek drug uporabnik, ali pa da prekliče svoje spremembe in osveži podatke v objektu *DataSet* z novim stanjem v bazi podatkov.

Naša aplikacija je sedaj pripravljena tudi za dodajanje novih podatkov v bazo, oz. za ažuriranje in brisanje le-teh.

Pravila o integriteti v objektih *DataSet*

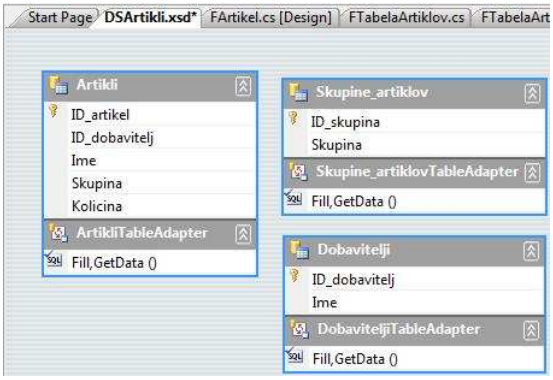
V prejšnjem primeru, ki uporablja en sam objekt *DataSet*, je malo verjetno, da nam bo Metoda *GetErrors* pri shranjevanju vrnila kakršnokoli napako. Ko smo generirali objekt *northwindDataSet*, je ta že vseboval informacijo o primarnih ključih, tipih podatkov za posamezne stolpce, pravila integritete in vse ostale informacije pridobljene iz baze podatkov. Uporabnik ažurira podatke oz. dela spremembe običajno preko objektov, kot je npr. *DataGridView*, ti objekti pa imajo zopet svoje mehanizme za kontroliranje uporabnikovih vnosov – nekatere izmed njih smo spoznali v prejšnjem poglavju. Kljub temu, da je bila izvedena kontrola podatkov že pri vnosu v celice gradnika *DataGridView*, pa je ponovna kontrola pred dokončnim zapisovanjem v bazo prav tako nujno potrebna. Le tako se bomo izognili vsem možnim napakam.

V primeru, da *DataSet* vsebuje večje število tabel, ki imajo primarne ključe oz. so med njimi relacijske povezave, bi bilo zelo moteče, če bi bil uporabnik prisiljen vnašati podatke v nekem posebnem zaporedju. Do take situacije pride npr., kadar metoda *GetErrors* nastopa samostojno. Metoda v tem primeru predvideva, da je uporabnik prenehal z vnašanjem podatkov in da lahko prične s kompleksnim navskrižnim preverjanjem med podatkovnimi tabelami in lovi vse možne napake.

Razvijalec aplikacij ne more nikoli zagotovo vedeti, kdaj bodo uporabniki vnesli nek neobičajen podatek, ki ga razvijalec sploh ni predvidel. Načelo dobre prakse je, da naj bo napisana koda defenzivna, kar pomeni, da mora biti napisana tako, da bo ujela in obdelala vse napake, kjerkoli se že pojavijo.

Vaja:

Naslednja vaja *Artikli* prikazuje malo bolj kompleksen projekt z več obrazci, namenjenimi za delo s podatkovnimi tabelami. Glavni obrazec vsebuje meni s tremi opcijami: *Šifranti*, *Artikli* in *Konec*. Projekt bo namenjen delu s podatkovno bazo *nabavaSQL* (baza se nahaja na ***http://uranic.tsckr.si/VISUAL%20C%23/***). Baza vsebuje tri tabele: *Artikli*, *Skupine_Artiklov* in *Dobavitelji*.

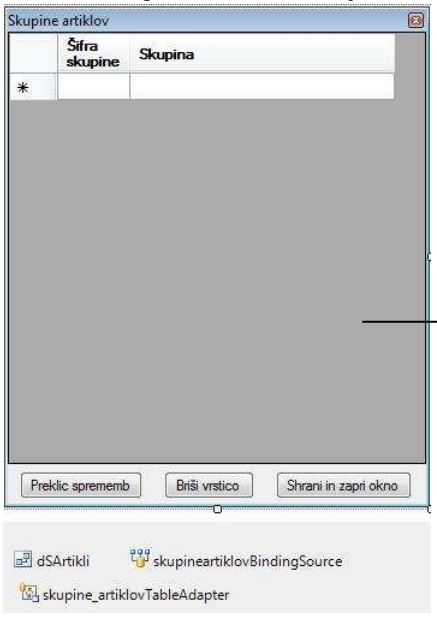


V projekt vključimo gradnik *DataSet* (*Project -> Add New Item -> DataSet*) in ga poimenujmo *DSArtikli*. V *DSArtikli* nato dojamo tri gradnike *Table Adapter* za vsako tabelo iz baze *nabavaSQL* posebej. Pri kreiranju adapterjev upoštevajmo dejstvo, da bomo vse tri tabele tudi ažurirali in dodajali nove zapise.

Opcija *Šifranti* vsebuje podopcijo *Skupine Artiklov*. Njen namen je delo s podatkovno tabelo Skupine_Artiklov. V ta namen oblikujmo nov obrazec z imenom *FSkupine*. Obrazec odpremo iz glavnega menija s stavki

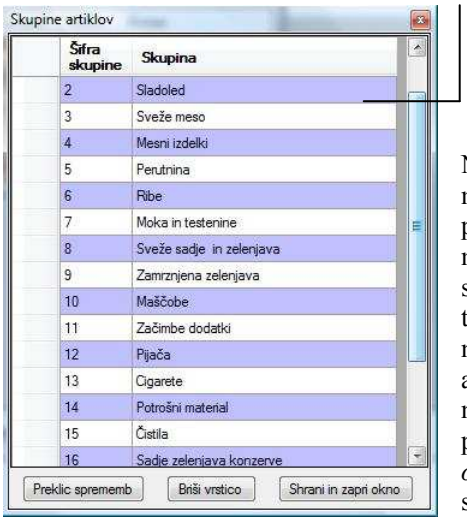
```
FSkupine SkupineArtiklov = new FSkupine();//Nov objekt razreda (obrazca) FSkupine
SkupineArtiklov.ShowDialog();//NNov obrazec odpremo z metod ShowDialog
```

Obrzec *FSkupine* moramo sedaj še kreirati. Odprimo prazen obrazec (*Project -> Add Windows Form*) in ga shranimo pod imenom *FSkupine*. Na nov obrazec nato dodajmo še gradnik *DataGridView* (ime gradnika pustimo kar *dataGridView1*) in mu nastavimo lastnost *DataSource* na *DSArtikli -> Skupine_Artiklov*. Za lepši izgled razširimo še lastnost *ColumnHeadersDefaultCallStyle* in nastavimo lastnost *Font* na *Bold* in po želji še lastnost *BackColor*. Za boljši pregled vrstic nastavimo še različni prikaz sodih in lihih vrstic: razširimo lastnost *AlternatingRowsDefaultCellStyle*, nato pa izberimo ustrezno barvo za lastnost *BackColor*. Lastnost *MultiSelect* za naše potrebe nastavimo na *False*.



Gradnik *DataGridView* z imenom *dataGridView1* v fazi načrtovanja

Izgled obrazca v zagnanem projektu je takle:



Na obrazcu so še trije gumbi, s pomočjo katerih bomo izvajali manipulacije z bazo podatkov. Ker je podatek o šifri skupine ključni podatek (ključ v tabeli Skupina_Artiklov), tega podatka seveda ne moremo spremeniti. Podatek v stolpcu *Skupina* pa seveda lahko spremenimo. Z miško kliknemo v ustrezno celico in s pomočjo tipkovnice spremenimo vsebino vrstice. Če se premaknemo v naslednjo celico, so spremembe v celici shranjene, niso pa še ažurirane v bazi podatkov in to ne glede na to, v koliko vrsticah smo naredili spremembe. Kodo za to moramo napisati posebej, v našem primeru pa to kodo zapišimo v dogodek *Click* gumba *Shrani in zapri okno*. Koda je podobna tisti iz prejšnjega poglavja, kjer so posamezni stavki tudi razloženi:

```
private void bShrani_Click(object sender, EventArgs e)
{
     try
     {
          DSArtikli spremembe = (DSArtikli)dSArtikli.GetChanges();
          if (spremembe == null)
          {
               Close();
               return;
          }
          DataTable dt = spremembe.Tables["Skupine artiklov"];
          DataRow[] NapacneVrstice = dt.GetErrors();

          if (NapacneVrstice.Length == 0)
          {
              int steviloVrstic = skupine_artiklovTableAdapter.Update(spremembe);
              dSArtikli.AcceptChanges();
              Close();//Po uspešnem shranjevanju v bazo zapremo obrazec
          }
          else
          {
              string errorMsg=null;
              foreach (DataRow vrstica in NapacneVrstice)
              {
                  foreach (DataColumn stolpec in vrstica.GetColumnsInError())
                  {
                      errorMsg+=vrstica.GetColumnError(stolpec)+"\n";
                  }
              }
              MessageBox.Show("Napake v podatkih: "+errorMsg,"Preverite vnos in popravite
                              Vnesene podatke!",MessageBoxButtons.OK,MessageBoxIcon.Error);
          }
     }
     catch (Exception ex)
     {
         MessageBox.Show("Napaka: " + ex.Message, "Napake", MessageBoxButtons.OK,
                          MessageBoxIcon.Error);
         dSArtikli.RejectChanges();
     }
}
```

Spremembe lahko tudi prekličemo (seveda pred še pred klikom na gumb za dokončno shranjevanje v bazo). Preklicu je namenjen gumb *Preklic sprememb*, kateremu v dogodek *Click* zapišimo naslednjo kodo:

```
        private void bPreklic_Click(object sender, EventArgs e)
        {
            /*Preklic vseh sprememb - v gradniku dataGridView1 bo vzpostavljeno stanje po
              zadnjem shranjevanju*/
            dSArtikli.RejectChanges();
        }
```

Na obrazcu *SkupineArtiklov* je še gumb za brisanje vrstice. Koda za brisanje je sicer enostavna, a v našem primeru se moramo naprej prepričati, ali v tabeli Artikli mogoče obstaja artikel tiste skupine, ki jo želimo pobrisati. V tem primeru moramo seveda brisanje preprečiti.

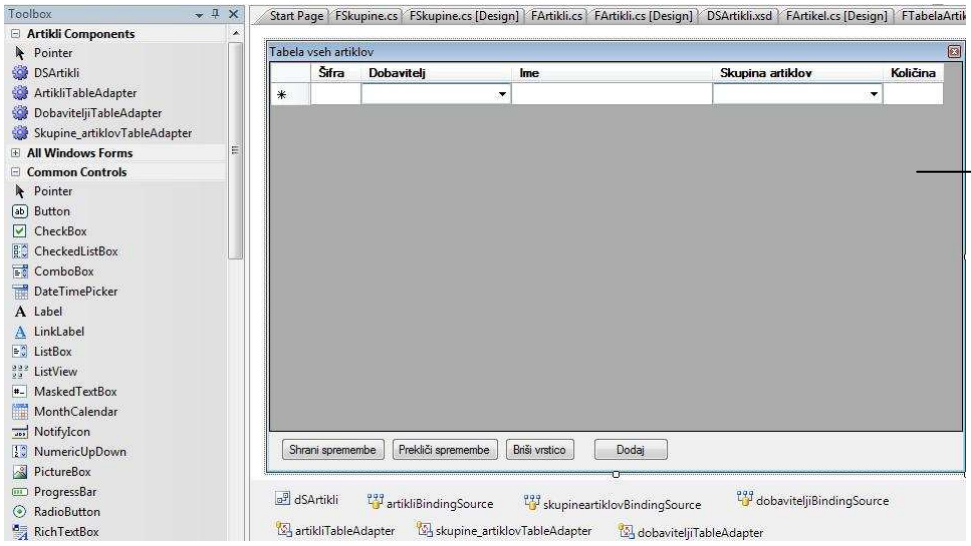
```
private void bBrisi_Click(object sender, EventArgs e)
{
    try
    {
       //pred brisanjem preverimo, če v tabeli Artikli obstaja kak artikel iz te skupine
       SqlCommand dataCommand = new SqlCommand();
       DSArtikliTableAdapters.ArtikliTableAdapter art;
       Art = new DSArtikliTableAdapters.ArtikliTableAdapter();
       dataCommand.Connection = art.Connection;
       dataCommand.Connection.Open();
       //kreiramo poizvedbo
       int stvrstice = dataGridView1.CurrentCell.RowIndex;
       //iz izbrane vrstice potegnemo vrednost skupine artiklov, ki jo želimo pobrisati
       int Skupina = (int)dataGridView1[0, stvrstice].Value;//0 pomeni indeks stolpca
       dataCommand.CommandText = "SELECT * From Artikli WHERE Skupina='" + Skupina + "'";
       SqlDataReader dataReader = dataCommand.ExecuteReader();
       //Če smo v tabeli Artikli našli artikel iz te skupine, potem skupine NE SMEMO brisati
       if (dataReader.Read() == true)
       {
           MessageBox.Show("Brisanje NI možno!\n\nV tabeli Artikli obstaja vsaj en artikel, ki
               pripada tej skupini!", "POZOR!", MessageBoxButtons.OK,MessageBoxIcon.Warning);
       }
       else
       {
          if (MessageBox.Show("Brisanje vrstice", "BRISANJE", MessageBoxButtons.OKCancel,
              MessageBoxIcon.Question) == DialogResult.OK)
          {
              dataGridView1.Rows.RemoveAt(stvrstice); //Pobrišemo vrstico v dataGridView1
          }
       }
       dataReader.Close();//zapremo podatkovni tok
       dataCommand.Connection.Close(); //Zapremo povezavo z bazo
    }
    catch
    {
       MessageBox.Show("Napaka pri dostopu do baze podatkov!");
    }
}
```

V zgornji kodi smo pobrisali vrstico le v gradniku *dataGridView1*. Potrebno je še ažurirati novo stanje v bazi podatkov. Spremembe zapišemo v bazo ali pa jih prekličemo s klikom na gumba *Shrani in zapri okno* oz. *Preklic sprememb*.

Opcija *Artikli* vsebuje dve podopcij, *Tabela Artiklov* in *Obdelava Artiklov*. Namen prve je tabelarični prikaz in ažuriranje tabele Artikli, namen druge podopcije pa preprosta obdelava oz. poizvedba iz te tabele. Za potrebe podopcije *Tabela Artiklov* skreirajmo nov obrazec in ga poimenujmo *fTabelaArtiklov*. Nov objekt naj bo statičen, da bomo lahko do njegovih gradnikov oz. metod dostopali kar preko imena obrazca. Nov obrazec bomo odprli s klikom na podopcijo *Tabele Artiklov* na glavnem obrazcu, koda pa je naslednja:

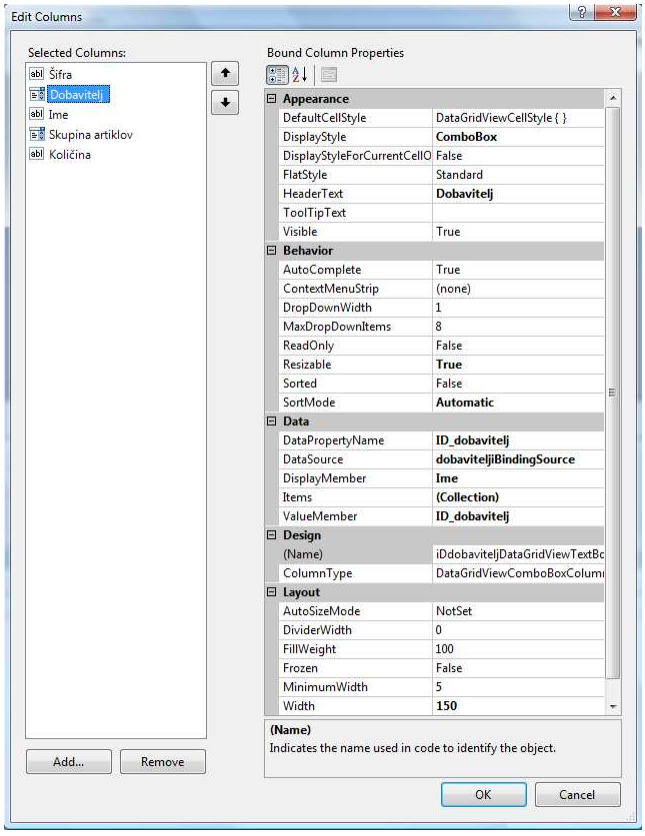
```
//Statičen objekt potreben za dostop do gradnika DataGridView na tem obrazcu
static public fTabelaArtiklov TabelaArtiklov;
private void tabelaArtiklovToolStripMenuItem Click(object sender, EventArgs e)
{   //Nova istanca obrazca fTabelaArtiklov ima ime TabelaArtiklov
    TabelaArtiklov = new fTabelaArtiklov();
    TabelaArtiklov.ShowDialog();
}
```

Obrazec *fTabelaArtiklov* oblikujmo tako, kot prikazuje slika:



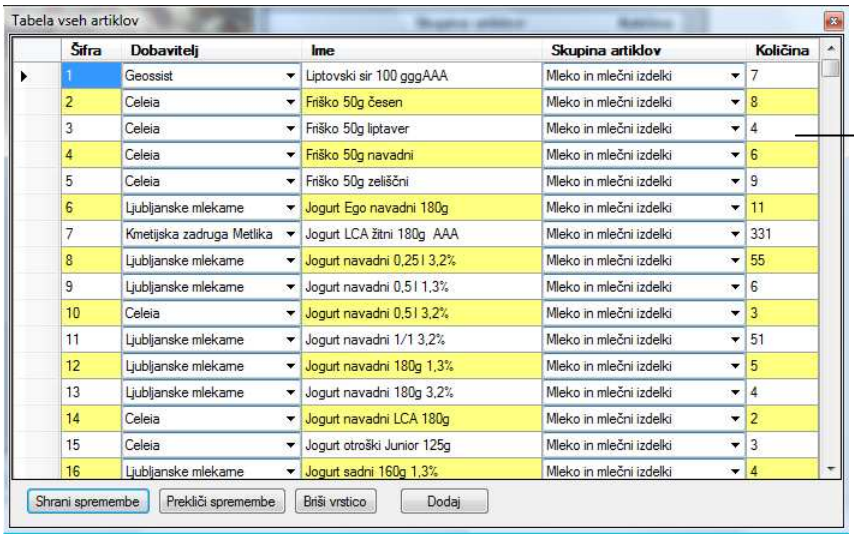
DataGridView z imenom *dataGridView1*

Gradniku *dataGridView1* določimo nastavimo lastnost *DataSource* na *DSArtikli -> Artikli*. . Za lepši izgled razširimo še lastnost *ColumnHeadersDefaultCallStyle* in nastavimo lastnost *Font* na *Bold* in po želji še lastnost *BackColor*. Za boljši pregled vrstic nastavimo še različni prikaz sodih in lihih vrstic: razširimo lastnost *AlternatingRowsDefaultCellStyle*, nato pa izberimo ustrezno barvo za lastnost *BackColor*. Kliknimo na tropičje pri lastnosti *Columns* in v oknu *Edit Columns* naredimo naslednje spremembe:



- Spremenimo tekst na vrhu stolpcev (lastnost *Header Text*)
- Stolpcema *Dobavitelji* in *Skupine* artiklov spremenimo lastnost *ColumnType* (dosedanja lastnost je *DataGridViewTextBoxColumn*) na *DataGridViewComboBoxColumn*. Ko smo to naredili, obema spremenimo lastnost *DisplayStyle* na *ComboBox*, nato pa še lastnosti razdelka *Data*. Stolpcu *Dobavitelji* nastavimo *DisplayMember* na *ID_Dobavitelj*, lastnost ValueMember pa na *ID_Dobavitelj*. Celice v tem stolpcu bodo imele obliko *ComboBox*-a, v katerega se bodo ob zagonu programa zapisali vsi dobavitelji, ob izbiri ustreznega dobavitelja pa se bo v celico zapisala vrednost polja *ID_Dobavitelj*.

Podobno storimo s poljem *Skupina Artiklov – DisplayMember* nastavimo na *Skupina*, *ValueMember* pa na *ID_Skupina*.



Objekt *TabelaArtiklov* v zagnanem projektu.

Če smo lastnosti gradniku *dataGridView1* nastavili pravilno, nam je *Visual C#* sam zgeneriral kodo za dogodek *Load* tega obrazca:

```
private void fTabelaArtiklov Load(object sender, EventArgs e)
{
   this.dobaviteljiTableAdapter.Fill(this.dSArtikli.Dobavitelji);
   this.skupine_artiklovTableAdapter.Fill(this.dSArtikli.Skupine_artiklov);
   this.artikliTableAdapter.Fill(this.dSArtikli.Artikli);
}
```

Podatke v *dataGridView1* bomo v zagnanem projektu lahko spreminjali (razen stolpca *Šifra*, ki je ključno polje v tabeli *Artikli*). Spremenimo lahko *Dobavitelja* in *Skupino_Artiklov* (z ustrezno izbiro v *ComboBox* –ih), ime artikla (stolpec *Ime*) in pa podatek o količino (stolpec *Kolicina*). Zadnji stolpec je tudi edini, za katerega moramo še posebej preveriti pravilnost uporabnikovega vnosa. Gradniku *dataGridView1* bomo zato zapisali kodo za dogodek *CellValidating*. Razlaga kode je zapisana v teoretičnem delu tega poglavja.

```
private void dataGridView1_CellValidating(object sender,DataGridViewCellValidatingEventArgs e)
{
    int zacasna;
    dataGridView1.Rows[e.RowIndex].ErrorText = "";
    //Kontroliramo le celico Kolicina
    if (dataGridView1.Columns[e.ColumnIndex].DataPropertyName == "Kolicina")
    {
        if (!int.TryParse(e.FormattedValue.ToString(), out zacasna) || zacasna < 0)
        {
           dataGridView1.Rows[e.RowIndex].ErrorText = "Vnesi pozitivno celo število";
           e.Cancel = true;
        }
    }
}
```

Vse spremembe v *dataGridView1* lahko shranimo s klikom na gumb *Shrani Spremembe*:

```
private void bShrani Click(object sender, EventArgs e)
{
   try
   {
       DSArtikli sprememe = (DSArtikli)dSArtikli.GetChanges();
       if (sprememe == null)
       {
          return;
       }
       DataTable dt = sprememe.Tables["Skupine_artiklov"];
       DataRow[] NapacneVrstice = dt.GetErrors();
       if (NapacneVrstice.Length == 0)
       {
          int steviloVrstic = artikliTableAdapter.Update(sprememe);
```

```
            dSArtikli.AcceptChanges();
        }
        else
        {
            string errorMsg = null;
            foreach (DataRow vrstica in NapacneVrstice)
            {
                foreach (DataColumn stolpec in vrstica.GetColumnsInError())
                {
                    errorMsg += vrstica.GetColumnError(stolpec) + "\n";
                }
            }
            MessageBox.Show("Napake v podatkih: " + errorMsg, "Prosim popravite podatke!",
                MessageBoxButtons.OK, MessageBoxIcon.Error);
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show("Napaka: " + ex.Message, "Napake", MessageBoxButtons.OK,
                  MessageBoxIcon.Error);
        dSArtikli.RejectChanges();
    }
}
```

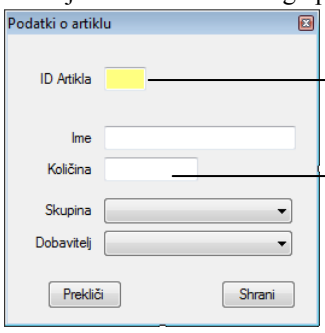
Spremembe prekličemo s klikom na gumb *Prekliči spremembe*:

```
private void bPreklici_Click(object sender, EventArgs e)
{
    dSArtikli.RejectChanges();
}
```

Brisanje vrstice zapišemo v odzivni dogodek *Click* gumba *Briši vrstico*:

```
private void bBrisi Click(object sender, EventArgs e)
{
    if (MessageBox.Show("Brisanje vrstice", "BRISANJE", MessageBoxButtons.OKCancel,
            MessageBoxIcon.Question) == DialogResult.OK)
    {
        //Ugotovimo številko izbrane vrstice
        int stvrstice = dataGridView1.CurrentCell.RowIndex;
        //pobrišemo izbrano vrstico
        dataGridView1.Rows.RemoveAt(stvrstice);
    }
}
```

Pokažimo še, kako bi ažuriranje podatkov izbrane vrstice realizirali s pomočjo novega obrazca. V ta namen kreirajmo nov obrazec in ga poimenujmo *FArtikel*. Na obrazec postavimo tri *TextBox*-e, dva *ComboBoxa* in dva gumba.



Lastnost *ReadOnly* nastavimo na *True*, saj gre za ključno polje.

Ker je polje *Količina* celoštevilsko, zapišimo za ta gradnik še dogodek *KeyPress*, ki porabniku omogoča le vnos cifer in decimalnega ločila:

```
private void tBKolicina KeyPress(object sender, KeyPressEventArgs e)
{
    if (((e.KeyChar < '0') || (e.KeyChar > '9'))
        && (e.KeyChar != ',') && (e.KeyChar != (char)(8))) e.Handled = true;
}
```

Obrazec *FArtikel* bomo odprli z dvoklikom na gradnik *dataGridView1* obrazca *FTabelaArtiklov* (dogodek *DoubleClick* gradnika *DataGridView1*). Pred prikazom tega obrazca pa moramo seveda poskrbeti za prenos podatkov izbrane vrstica gradnika *dataGridView1* v ustrezne gradnike obrazca *FArtikel*, poleg tega pa moramo v

ComboBox-a obrazca prenesti celotni vsebini tabele *Dobavitelji* in *SkupineArtiklov*. Samo kot primer bomo pri shranjevanju uporabnikovih podatkov napisali kodo za shranjevanje podatkov neposredno v bazo (in potem takoj ažurirali vsebino *dataGridView1*) in pa kodo za shranjevanje v gradnik *dataGridView1* (s klikom na gumb *Shrani Spremembe* pa se podatki nato zapišejo še v bazo).

```
private void dataGridView1_DoubleClick(object sender, EventArgs e)
{
   /*Ažuriranje vrstice s pomočjo novega obrazca: SPREMENJENE PODATKE BOMO ZAPISALI NAzAJ V
     GRADNIK DataGridView IN ŠELE NATO AŽURIRALI BAZO*/
   FArtikel IzbraniArtikel = new FArtikel();
   string ID = Convert.ToString( dataGridView1[0, dataGridView1.CurrentCell.RowIndex].Value);
   //objekti tBID, tBIme, tBKolicina na obrazcu FArtikel morajo biti public!!!!
   IzbraniArtikel.tBID.Text = ID;
   string imeArtikla;
   imeArtikla = Convert.ToString( dataGridView1[2, dataGridView1.CurrentCell.RowIndex].Value);
   IzbraniArtikel.tBIme.Text = imeArtikla;
   string Kolicina;
   Kolicina = Convert.ToString( dataGridView1[4, dataGridView1.CurrentCell.RowIndex].Value);
   IzbraniArtikel.tBKolicina.Text = Kolicina;
   int ID skupina;
   ID_Skupina =  Convert.ToInt32(dataGridView1[3, dataGridView1.CurrentCell.RowIndex].Value);
   int ID_dobavit=0;
   try
   {
     ID dobavit = Convert.ToInt32(dataGridView1[1, dataGridView1.CurrentCell.RowIndex].Value);
   }
   catch { }
   //ComboBox CBSkupina napolnimo z vrednostmi iz tabele Skupina_Artiklov
   string skupina="",dobavit="";
   try
   {
      SqlCommand dataCommand = new SqlCommand();
      dataCommand.Connection = skupine_artiklovTableAdapter.Connection;
      dataCommand.Connection.Open();
      //kreiramo poizvedbo
      dataCommand.CommandText = "SELECT ID Skupina,Skupina ";
      dataCommand.CommandText += "From Skupine artiklov";
      SqlDataReader dataReader = dataCommand.ExecuteReader();
      //v ComboBox cBSkupina zapišemo vse stranke iz tabele Skupine_artiklov
      while (dataReader.Read())//dokler obstajajo podatki
      {
          /*v cBSkupina zapišemo dato ID skupine, kot tudi ime skupine - med njima je LOČILNI
             znak PRESLEDEK*/
          IzbraniArtikel.cBSkupina.Items.Add(dataReader.GetValue(0).ToString()+"
                  "+dataReader.GetValue(1).ToString());
          if ((int)dataReader.GetValue(0) == ID_skupina)
          skupina = dataReader.GetValue(0).ToString()+" "+dataReader.GetValue(1).ToString();
      }
      dataReader.Close();//zapremo podatkovni tok
      dataCommand.Connection.Close(); //Zapremo povezavo z bazo

      //v cBDobavitelji zapišemo seznam vseh dobaviteljev
      dataCommand.Connection = dobaviteljiTableAdapter.Connection;
      dataCommand.Connection.Open();
      //kreiramo poizvedbo
      dataCommand.CommandText = "SELECT ID_Dobavitelj,Ime ";
      dataCommand.CommandText += "From Dobavitelji";
      dataReader = dataCommand.ExecuteReader();
      //v ComboBox cBDobavitelji zapišemo vse stranke iz tabele Skupine artiklov
      while (dataReader.Read())//dokler obstajajo podatki
      {
        /*v cBDobavitelji zapišemo dato ID_Dobavitelja, kot tudi ime  - med njima je LOČILNI
            znak PRESLEDEK*/
        IzbraniArtikel.cBDobavitelj.Items.Add(dataReader.GetValue(0).ToString() + " " +
              dataReader.GetValue(1).ToString());
        if ((int)dataReader.GetValue(0) == ID dobavit)
          dobavit = dataReader.GetValue(0).ToString()+" " + dataReader.GetValue(1).ToString();
      }
      dataReader.Close();//zapremo podatkovni tok
      dataCommand.Connection.Close(); //Zapremo povezavo z bazo
   }
   catch
   {
      MessageBox.Show("  Napaka pri dostopu do baze podatkov!");
```

```
    }
    IzbraniArtikel.cBSkupina.Text =skupina;
    IzbraniArtikel.cBDobavitelj.Text = dobavit;

    if (IzbraniArtikel.ShowDialog() == DialogResult.OK)
    {
       try
       {
          /*SAMO ZA VAJO: Prikaz dveh načinov shranjevanja podatkov vnešenih oz. izbranih na
            objektu IzbraniArtikel
            PONUDIMO dva načina: shranjevanje NEPOSREDNO v bazo (in takojšnja osvežitev vsebine
            dataGridView1) in  shranjevanje v gradnik dataGridView1 in nato po želji še
            shranjevanje v bazo podatkov*/
            if (MessageBox.Show("Shranjevanje neposredno v bazo?","Shranjevanje podatkov!",
                MessageBoxButtons.OKCancel, MessageBoxIcon.Question)==DialogResult.OK)
            {
              /*spremenjene podatke shranimo neposredno v bazo in nato osvežimo sebino
                 gradnika dataGridView1*/
              SqlCommand dataCommand1 = new SqlCommand();
              dataCommand1.Connection = artikliTableAdapter.Connection;
              dataCommand1.CommandText = "UPDATE Artikli SET ID_dobavitelj = @ID_dobavitelj,
                                        Ime = @Ime, Skupina = @Skupina, Kolicina = @Kolicina ";
              dataCommand1.CommandText += "WHERE (ID artikel = @Original ID artikel) AND
                                          (ID_dobavitelj IS NULL OR ";
              dataCommand1.CommandText += "ID_dobavitelj = @Original_ID_dobavitelj) AND (Ime
                                          IS NULL OR ";
              dataCommand1.CommandText += "Ime = @Original_Ime) AND (Skupina IS NULL OR
                                          Skupina = @Original Skupina) ";
              dataCommand1.CommandText += "AND (Kolicina IS NULL OR Kolicina =
                                          @Original_Kolicina)";
              dataCommand1.CommandType=CommandType.Text;

              dataCommand1.Parameters.AddWithValue("@Original ID Artikel", dataGridView1[0,
                           dataGridView1.CurrentCell.RowIndex].Value);
              dataCommand1.Parameters.AddWithValue("@Original_ID_Dobavitelj", dataGridView1[1,
                           dataGridView1.CurrentCell.RowIndex].Value);
              dataCommand1.Parameters.AddWithValue("@Original_Ime", dataGridView1[2,
                           dataGridView1.CurrentCell.RowIndex].Value);
              dataCommand1.Parameters.AddWithValue("@Original Skupina", dataGridView1[3,
                           dataGridView1.CurrentCell.RowIndex].Value);
              dataCommand1.Parameters.AddWithValue("@Original_Kolicina", dataGridView1[4,
                           dataGridView1.CurrentCell.RowIndex].Value);

              string[] tabDob = (IzbraniArtikel.cBDobavitelj.Text).Split();
              int dobavitelj = Convert.ToInt32(tabDob[0]);
              //metoda AddWithValues v ustrezni parameter zapiše izbrane oz. vnešene vrednosti
              dataCommand1.Parameters.AddWithValue("@ID_Dobavitelj",dobavitelj);
              dataCommand1.Parameters.AddWithValue("@Ime",IzbraniArtikel.tBIme.Text);
              //s pomočjo metode Split iz izbrane vrstice izločimo le ID številko skupine
              string[] tab = (IzbraniArtikel.cBSkupina.Text).Split();
              int skupinaArt = Convert.ToInt32(tab[0]);
              dataCommand1.Parameters.AddWithValue("@Skupina",skupinaArt);
              dataCommand1.Parameters.AddWithValue("@Kolicina",IzbraniArtikel.tBKolicina.Text);
              dataCommand1.Connection.Open();  //odpremo poezavo z bazo podatkov
              dataCommand1.ExecuteNonQuery();  //izvedemo SQL stavek
              dataCommand1.Connection.Close(); //OBVEZNO zapremo povezavo z bazo, sicer
                                               //spremembe ne bodo zavedene
              //ažuriramo daArtikli s tem pa tudi stanje v dataGridView1
              this.artikliTableAdapter.Fill(this.dSArtikli.Artikli);
            }
            else
            {
              //spremenjene podatke shranimo nazaj dataGridView1, šele kasneje pa jih lahko
              //shranimo tudi v bazo
              string[] tabDob = (IzbraniArtikel.cBDobavitelj.Text).Split();
              int dobavitelj = Convert.ToInt32(tabDob[0]);
              dataGridView1[1, dataGridView1.CurrentCell.RowIndex].Value = dobavitelj;
              dataGridView1[2,dataGridView1.CurrentCell.RowIndex].Value =
                   IzbraniArtikel.tBIme.Text;
              string[] tab = (IzbraniArtikel.cBSkupina.Text).Split();
              int skupinaArt = Convert.ToInt32(tab[0]);
              dataGridView1[3, dataGridView1.CurrentCell.RowIndex].Value = skupinaArt;
              dataGridView1[4, dataGridView1.CurrentCell.RowIndex].Value =
                   IzbraniArtikel.tBKolicina.Text;
              dataGridView1.Refresh();
```

```
            }
        }
        catch
        {
            MessageBox.Show("Napaka pri shranjevanju podatkov! "+e.ToString());
        }
    }
}
```

Na obrazcu *FTabelaArtiklov* realizirajmo še gumb *Dodaj*, za dodajanje nove vrstice podatkov v tabelo *Artikli*. Za dodajanje bomo uporabili kar obrazec *FArtilkli*, le koda, ki jo zapišemo v odzivni dogodek *Click* gumba *Dodaj* je nekoliko drugačna. Vneseni podatki se bodo v našem primeru shranili neposredno v bazo, zato je po shranjevanju potrebno ažurirati vsebino *dataGridView1*.

```
private void bDodaj Click(object sender, EventArgs e)
{
    FArtikel IzbraniArtikel = new FArtikel();
    SqlCommand dataCommand = new SqlCommand();
    dataCommand.Connection = skupine_artiklovTableAdapter.Connection;
    dataCommand.Connection.Open();
    //kreiramo poizvedbo
    dataCommand.CommandText = "SELECT ID Skupina,Skupina ";
    dataCommand.CommandText += "From Skupine artiklov";
    SqlDataReader dataReader = dataCommand.ExecuteReader();
    //v cBSkupina zapišemo vse podatke iz tabele Skupine_artiklov
    while (dataReader.Read())//dokler obstajajo podatki
    {
      //v cBSkupina zapišemo ID skupine in tudi ime skupine-med njima je LOČILNI znak PRESLEDEK
      IzbraniArtikel.cBSkupina.Items.Add(dataReader.GetValue(0).ToString() + " " +
          dataReader.GetValue(1).ToString());
    }
    dataReader.Close();//zapremo podatkovni tok
    dataCommand.Connection.Close(); //Zapremo povezavo z bazo

    //v cBDobavitelji zapišemo seznam vseh dobaviteljev
    dataCommand.Connection = dobaviteljiTableAdapter.Connection;
    dataCommand.Connection.Open();
    //kreiramo poizvedbo
    dataCommand.CommandText = "SELECT ID Dobavitelj,Ime ";
    dataCommand.CommandText += "From Dobavitelji";
    dataReader = dataCommand.ExecuteReader();
    //v cBSkupina zapišemo vse vrstice iz tabele Skupine_artiklov
    while (dataReader.Read())//dokler obstajajo podatki
    {
      //v cBDobavitelji zapišemo ID Dobavitelja in ime  - med njima je LOČILNI znak PRELEDEK
      IzbraniArtikel.cBDobavitelj.Items.Add(dataReader.GetValue(0).ToString() + " " +
            dataReader.GetValue(1).ToString());
    }
    dataReader.Close();//zapremo podatkovni tok
    dataCommand.Connection.Close(); //Zapremo povezavo z bazo

    if (IzbraniArtikel.ShowDialog() == DialogResult.OK)
    {
       try
       {
         //NOVO vrstico shranimo neposredno v tabelo v bazi podatkov, nato pa osvežimo
         //dSArtikli in s tem tudi vsebino dataGridView1
         SqlCommand dataCommand1 = new SqlCommand();
         dataCommand1.Connection = artikliTableAdapter.Connection;
         //v SQL stavku določimo štiri parametre, preko katerih bomo v ustrezna polja tabele
         //Artikli posredovali nove vrednosti
         dataCommand1.CommandText = "INSERT INTO Artikli(ID dobavitelj, Ime, Skupina, Kolicina)
                                     VALUES (@ID Dobavitelj,@Ime,@Skupina,@Kolicina)";
         dataCommand1.CommandType=CommandType.Text;
         //s pomočjo metode Split iz izbrane vrstice izločimo le ID številko dobavitelja
         string[] tabDob = (IzbraniArtikel.cBDobavitelj.Text).Split();
         int dobavitelj = Convert.ToInt32(tabDob[0]);
         //metoda AddWithValues v ustrezni parameter zapiše izbrane oz. vnešene vrednosti
         dataCommand1.Parameters.AddWithValue("@ID Dobavitelj",dobavitelj);
         dataCommand1.Parameters.AddWithValue("@Ime",IzbraniArtikel.tBIme.Text);
         //s pomočjo metode Split iz izbrane vrstice izločimo le številko skupine
         string[] tab = (IzbraniArtikel.cBSkupina.Text).Split();
         int skupinaArt = Convert.ToInt32(tab[0]);
         dataCommand1.Parameters.AddWithValue("@Skupina",skupinaArt);
```

```
        dataCommand1.Parameters.AddWithValue("@Kolicina",IzbraniArtikel.tBKolicina.Text);

        dataCommand1.Connection.Open();  //odpremo pvezavo z bazo podatkov
        dataCommand1.ExecuteNonQuery();  //izvedemo SQL stavek
        dataCommand1.Connection.Close(); //OBVEZNO zapremo povezavo z bazo, sicer spremembe ne
                                         //bodo zavedene
        //ažuriramo daArtikli s tem pa tudi stanje v dataGridView1
        this.artikliTableAdapter.Fill(this.dSArtikli.Artikli);
      }
      catch
      { MessageBox.Show("Napaka pri shranjevanju podatkov!"); }
   }
}
```

Napišimo še kodo za podopcijo *Obdelava Artiklov*, ki je sestavni del glavnega menija na osnovnem obrazcu projekta. Ugotovili bomo in na koncu v sporočilnem oknu tudi izpisali, kolikšna je skupna količina vseh artiklov v tabeli *Artikli*.

```
private void obdelavaArtiklovToolStripMenuItem_Click(object sender, EventArgs e)
{
   DSArtikliTableAdapters.ArtikliTableAdapter aArtikli = new
                  DSArtikliTableAdapters.ArtikliTableAdapter();
   SqlCommand dataCommand = new SqlCommand();
   dataCommand.Connection = aArtikli.Connection ;
   dataCommand.Connection.Open();
   //kreiramo poizvedbo vseh vrednosti Kolicina iz tabele Artikli, ki so različna od null
   dataCommand.CommandText = "SELECT Kolicina From Artikli WHERE Kolicina IS NOT NULL";
   SqlDataReader dataReader = dataCommand.ExecuteReader();
   int skupaj=0;
   while (dataReader.Read())//dokler obstajajo podatki
   {
     //v cBSkupina zapišemo ID skupine, in ime skupine - med njima je LOČILNI znak PRESLEDEK
     try
     {
        skupaj = skupaj + Convert.ToInt32(dataReader.GetValue(0));
     }
     catch
     {
       MessageBox.Show("Napaka pri branju podatkov!");
     }
   }
   dataReader.Close();//zapremo podatkovni tok
   dataCommand.Connection.Close(); //Zapremo povezavo z bazo
   MessageBox.Show("Skupna količina vseh artiklov v tabeli: "+skupaj.ToString());
}
```

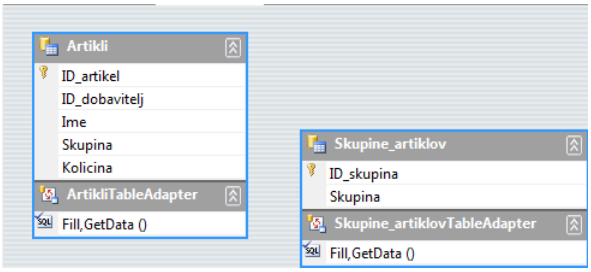
Transakcije

Kadar je operacija nad podatki sestavljena iz več ukazov (primer v bazi *nabavaSQL* je npr. brisanje vrstice v tabeli *Skupina_Artikov* in hkratno brisanje vseh artiklov iz tabele *Artikli*, ki pripadajo pobrisani vrstici v tabeli *Skupina_Artiklov*), uporabimo transakcijo. Transakcija običajno (ni pa nujno) povzroči več sprememb v eni ali pa v več tabelah. Bistvena lastnost transakcij je, da se bodo izvedli vsi SQL ukazi znotraj transakcije, ali pa nobeden. Če se torej transakcija ne izvede v celoti, se vse spremembe zavržejo, vzpostavi se prvotno stanje (stanje pred transakcijo). Transakcije torej uporabljamo za zavarovanje podatkov v primeru, da med izvajanjem SQL stavkov pride do napake in je potrebno vzpostaviti prvotno stanje.

Transakcijo pričnemo s kreiranjem novega objekta razreda *SQLTransaction*, končamo pa s stavkom *COMMIT* (ki dokončno potrdi spremembe v bazi), ali pa z ukazom *ROLLBACK* (ki spremembe zavrže).

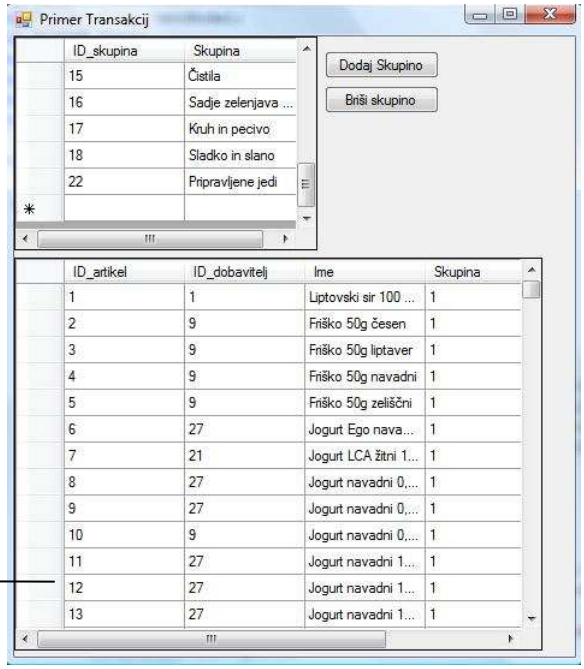
Primer:

Kreirajmo nov projekt in v projekt dodajmo *DataSet* in ga povežimo z bazo *nabavaSQL*. V *DataSet* dodajmo dva *TableAdapter*-ja.



Na osnovni obrazec dodajmo dva gradnika *DataGridView* in jima določimo lastnost *DataSource* tako, da bomo imeli v zgornjem gradniku (*dataGridView1*) vsebino tabele *Skupine_artiklov*, v spodnjem gradniku *dataGridView2* pa vsebino tabele *Artikli*. Na obrazec postavimo še dva gumba.

Izgled obrazca ko poženemo projekt



Odzivnemu dogodku *Click* gumba *Dodaj Skupino* bomo priredili transakcijo, ki bo v tabelo *Skupina_artiklov* dodala novo vrstico (skupini bomo dali ime *SkupinaXX*, pri čemer je XX trenutno število podatkov v tej tabeli).

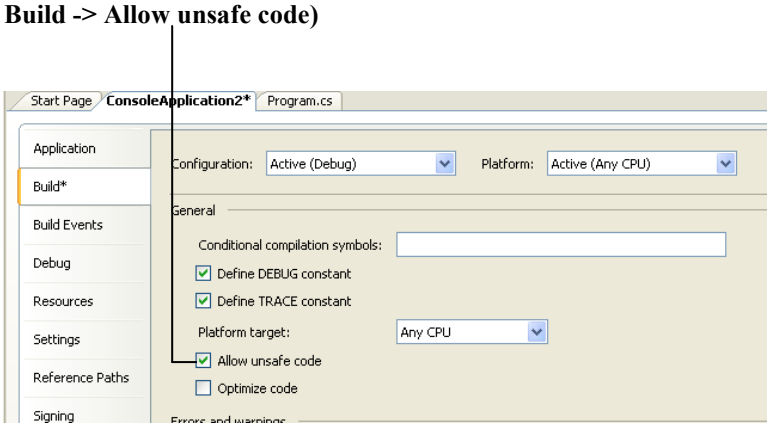
```
private void bDodaj Click(object sender, EventArgs e)
{
    SqlConnection dataConnection = new SqlConnection();
    dataConnection.ConnectionString = "Integrated Security=true;" +
                                      "Initial Catalog=nabavaSQL;" +
                                      "Data Source=mojRacunalnik\\SQLExpress";
    dataConnection.Open();
    //Kreiramo nov objekt za transakcijo
    SqlTransaction myTrans = dataConnection.BeginTransaction();
    //Kreiramo nov objekt za SQL ukaz
    SqlCommand dataCommand = new SqlCommand();
    dataCommand.Connection = dataConnection;
    dataCommand.Transaction = myTrans;
    try
    {
       int skupina = dataGridView1.Rows.Count;
       string novaSkupina = "Skupina" + skupina.ToString();
       dataCommand.CommandText = "Insert INTO Skupine_artiklov (Skupina) VALUES (@novaSk)";
       dataCommand.Parameters.AddWithValue("@novaSk", novaSkupina);
       dataCommand.CommandType = CommandType.Text;
       dataCommand.ExecuteNonQuery();
       //dokončno potrdimo spremembe v bazi
       myTrans.Commit();
    }
    catch (Exception ep)
    {
       //Če je prišlo do napake, vzpostavimo prvorno stanje v bazi podatkov
       myTrans.Rollback();
       MessageBox.Show("Napaka pri shranjevanju podatkov!");
    }
    finally
    {
       dataConnection.Close();
       //Ažuriramo vsebino gradnika dataGridView1 - tako da ažuriramo objekt dataSet1
       this.skupine_artiklovTableAdapter.Fill(this.dataSet1.Skupine_artiklov);
       this.artikliTableAdapter.Fill(this.dataSet1.Artikli);
    }
}
```

Odzivnemu dogodku *Click* gumba *Briši Skupino* pa bomo priredili transakcijo, ki bo v tabelo *Skupina_artiklov* izbrano vrstico pobrisala, nato pa v tabeli *Artikli* pobrisale še vse vrstice, v katerih nastopa podatek *Skupina*, ki smo pobrisali v tabeli *Skupina_artiklov*. V transakciji bomo imeli torej dva SQL ukaza in samo v primeru, da se bosta oba izvedla do konca, bo stanje v bazi podatkov spremenjeno, sicer pa ne.

```
private void bBrisi_Click(object sender, EventArgs e)
{
    SqlConnection dataConnection = new SqlConnection();
    dataConnection.ConnectionString = "Integrated Security=true;" +
                                      "Initial Catalog=nabavaSQL;" +
                                      "Data Source=sreco-PC\\SQLExpress";
    dataConnection.Open();
    //Kreiramo nov objekt za SQL transakcijo v SQL bazi podatkov
    SqlTransaction myTrans = dataConnection.BeginTransaction();
    //Kreiramo nov objekt za SQL stavek, ki ga bomo izveddli v SQL bazi podatkov
    SqlCommand dataCommand = new SqlCommand();
    dataCommand.Connection = dataConnection;
    dataCommand.Transaction = myTrans;
    try
    {
       //ugotovimo vrednost v celici ID_skupina izbrane vrstice
       int skupinaID;
       skupinaID = Convert.ToInt32(dataGridView1[0,dataGridView1.CurrentCell.RowIndex].Value);
       //oblikujemo SQL ukaz za brisanje te vrstice
       dataCommand.CommandText = "delete Skupine artiklov where ID skupina=(@novaSk)";
       dataCommand.Parameters.AddWithValue("@novaSk", skupinaID);
       //Poizkusimo izvesti dejansko brisanje izbrane skupine v bazi podatkov.
       dataCommand.ExecuteNonQuery();
       /*oblikujemo še SQL ukaz za brisanje vseh artiklov iz tabele Artikli, ki pripadajo
         izbrisani skupini*/
       dataCommand.CommandText = "delete Artikli where Skupina=(@novaSk)";
       //Poizkusimo izvesti dejansko brisanje vseh artiklov iz te skupine v bazi podatkov.
       dataCommand.ExecuteNonQuery();
       //dokončno shranimo narejene spremembe
       myTrans.Commit();
    }
    catch (Exception ep)
    {
       //Če transakcija ni uspela, vzpostavimo začetno stanje.
       myTrans.Rollback();
       MessageBox.Show("Napaka pri brisanju podatkov!");
    }
    finally
    {
       //na koncu zapremo našo transakcijo
       dataConnection.Close();
       this.skupine_artiklovTableAdapter.Fill(this.dataSet1.Skupine_artiklov);
    }
}
```

Kazalci v C#: Operator ->

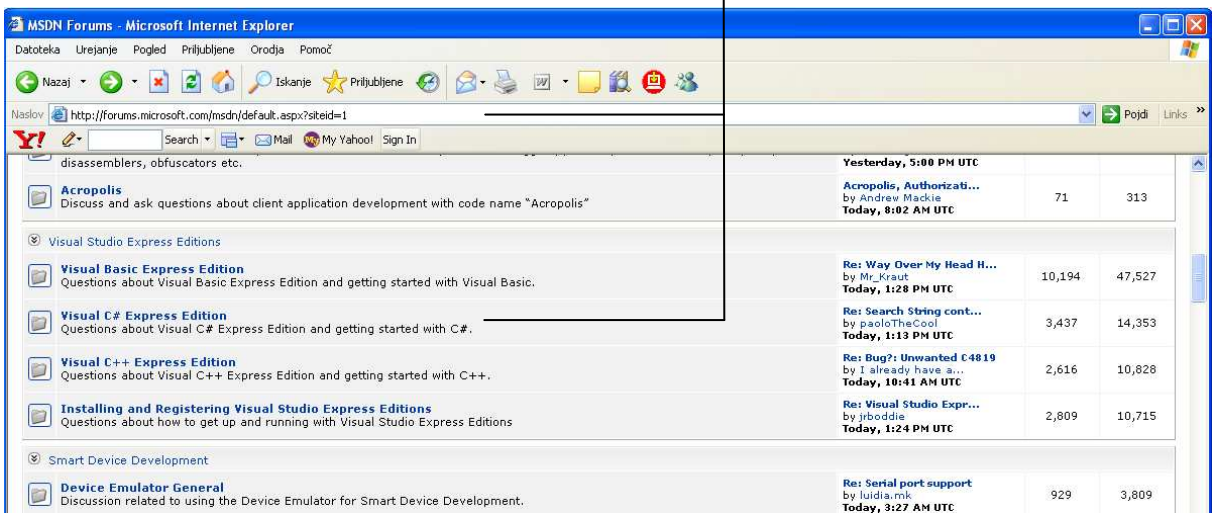
Tudi C# pozna kazalce, a jih uporabljamo redkeje kot v C++. Kodo, ki uporablja kazalce moramo obvezno zapreti v blok ***unsafe***, pri prevajanju pa odkljukati opcijo **Allow unsafe code (Meni Project -> Properties-> Build -> Allow unsafe code)**




```
struct Tocka
{
   public int x;
   public int y;
}

static void Main(string[] args)
{

    unsafe  // začetek bloka unsafe
    {
       Tocka pt = new Tocka();  // nov objekt izpeljan iz strukture Tocka
       Tocka* pp = &pt;   // kazalec pp (ki je tipa Tocka) kaže na objekt pt
       pp->x = 123;  // polje x objekta pt (na objekt kaže kazalec pp) dobi vrednost 123
       pp->y = 456;  // polje y objekta pt (na objekt kaže kazalec pp) dobi vrednost 456
       Console.WriteLine("{0} {1}", pt.x, pt.y); //izpis obeh polj objekta pt
    }  // konec bloka unsafe
}
```

Pomoč

Spoznali smo nekaj osnov pri delu s **C#.** Pri nadaljnjem delu bomo seveda rabili še veliko pomoči, ki nam jo okolje **C#** ponuja v meniju **Help**, ali pa pomoč poiščemo preprosto tako, da z miško kliknemo na besedo, za katero rabimo pomoč in nato na tipkovnici pritisnemo tipko **F1**. Če pa nam pomoč v meniju **Help** ali pa **F1** še ne zadoščata, imamo na voljo številne **msdn** forume!



Dodatek 1: Kode tipk (KeyCode) na tipkovnici

Ikona označuje tipke, ki jih podpira .NET Compact Framework. Zaradi enostavnosti razlaga posamezne tipke ni prevedena!

	Oznaka tipke	Razlaga
✓	**A**	**The A key.**
✓	**Add**	**The add key.**
✓	**Alt**	**The ALT modifier key.**
✓	**Apps**	**The application key (Microsoft Natural Keyboard).**
✓	**Attn**	**The ATTN key.**
✓	**B**	**The B key.**
✓	**Back**	**The BACKSPACE key.**
	BrowserBack	**The browser back key (Windows 2000 or later).**
	BrowserFavorites	**The browser favorites key (Windows 2000 or later).**
	BrowserForward	**The browser forward key (Windows 2000 or later).**
	BrowserHome	**The browser home key (Windows 2000 or later).**
	BrowserRefresh	**The browser refresh key (Windows 2000 or later).**
	BrowserSearch	**The browser search key (Windows 2000 or later).**
	BrowserStop	**The browser stop key (Windows 2000 or later).**
✓	**C**	**The C key.**
✓	**Cancel**	**The CANCEL key.**
✓	**Capital**	**The CAPS LOCK key.**
✓	**CapsLock**	**The CAPS LOCK key.**
✓	**Clear**	**The CLEAR key.**
✓	**Control**	**The CTRL modifier key.**

ControlKey	**The CTRL key.**
Crsel	**The CRSEL key.**
D	**The D key.**
D0	**The 0 key.**
D1	**The 1 key.**
D2	**The 2 key.**
D3	**The 3 key.**
D4	**The 4 key.**
D5	**The 5 key.**
D6	**The 6 key.**
D7	**The 7 key.**
D8	**The 8 key.**
D9	**The 9 key.**
Decimal	**The decimal key.**
Delete	**The DEL key.**
Divide	**The divide key.**
Down	**The DOWN ARROW key.**
E	**The E key.**
End	**The END key.**
Enter	**The ENTER key.**
EraseEof	**The ERASE EOF key.**
Escape	**The ESC key.**
Execute	**The EXECUTE key.**
Exsel	**The EXSEL key.**
F	**The F key.**

F1	**The F1 key.**
F10	**The F10 key.**
F11	**The F11 key.**
F12	**The F12 key.**
F13	**The F13 key.**
F14	**The F14 key.**
F15	**The F15 key.**
F16	**The F16 key.**
F17	**The F17 key.**
F18	**The F18 key.**
F19	**The F19 key.**
F2	**The F2 key.**
F20	**The F20 key.**
F21	**The F21 key.**
F22	**The F22 key.**
F23	**The F23 key.**
F24	**The F24 key.**
F3	**The F3 key.**
F4	**The F4 key.**
F5	**The F5 key.**
F6	**The F6 key.**
F7	**The F7 key.**
F8	**The F8 key.**
F9	**The F9 key.**
FinalMode	**The IME final mode key.**

G	The G key.
H	The H key.
HanguelMode	The IME Hanguel mode key. (maintained for compatibility; use HangulMode)
HangulMode	The IME Hangul mode key.
HanjaMode	The IME Hanja mode key.
Help	The HELP key.
Home	The HOME key.
I	The I key.
IMEAccept	The IME accept key, replaces IMEAceept.
IMEAceept	The IME accept key. Obsolete, use IMEAccept instead.
IMEConvert	The IME convert key.
IMEModeChange	The IME mode change key.
IMENonconvert	The IME nonconvert key.
Insert	The INS key.
J	The J key.
JunjaMode	The IME Junja mode key.
K	The K key.
KanaMode	The IME Kana mode key.
KanjiMode	The IME Kanji mode key.
KeyCode	The bitmask to extract a key code from a key value.
L	The L key.
LaunchApplication1	The start application one key (Windows 2000 or later).
LaunchApplication2	The start application two key (Windows 2000 or later).
LaunchMail	The launch mail key (Windows 2000 or later).
LButton	The left mouse button.

LControlKey	**The left CTRL key.**
Left	**The LEFT ARROW key.**
LineFeed	**The LINEFEED key.**
LMenu	**The left ALT key.**
LShiftKey	**The left SHIFT key.**
LWin	**The left Windows logo key (Microsoft Natural Keyboard).**
M	**The M key.**
MButton	**The middle mouse button (three-button mouse).**
MediaNextTrack	**The media next track key (Windows 2000 or later).**
MediaPlayPause	**The media play pause key (Windows 2000 or later).**
MediaPreviousTrack	**The media previous track key (Windows 2000 or later).**
MediaStop	**The media Stop key (Windows 2000 or later).**
Menu	**The ALT key.**
Modifiers	**The bitmask to extract modifiers from a key value.**
Multiply	**The multiply key.**
N	**The N key.**
Next	**The PAGE DOWN key.**
NoName	**A constant reserved for future use.**
None	**No key pressed.**
NumLock	**The NUM LOCK key.**
NumPad0	**The 0 key on the numeric keypad.**
NumPad1	**The 1 key on the numeric keypad.**
NumPad2	**The 2 key on the numeric keypad.**
NumPad3	**The 3 key on the numeric keypad.**
NumPad4	**The 4 key on the numeric keypad.**

	NumPad5	**The 5 key on the numeric keypad.**
	NumPad6	**The 6 key on the numeric keypad.**
	NumPad7	**The 7 key on the numeric keypad.**
	NumPad8	**The 8 key on the numeric keypad.**
	NumPad9	**The 9 key on the numeric keypad.**
	O	**The O key.**
	Oem1	**The OEM 1 key.**
	Oem102	**The OEM 102 key.**
	Oem2	**The OEM 2 key.**
	Oem3	**The OEM 3 key.**
	Oem4	**The OEM 4 key.**
	Oem5	**The OEM 5 key.**
	Oem6	**The OEM 6 key.**
	Oem7	**The OEM 7 key.**
	Oem8	**The OEM 8 key.**
	OemBackslash	**The OEM angle bracket or backslash key on the RT 102 key keyboard (Windows 2000 or later).**
	OemClear	**The CLEAR key.**
	OemCloseBrackets	**The OEM close bracket key on a US standard keyboard (Windows 2000 or later).**
	Oemcomma	**The OEM comma key on any country/region keyboard (Windows 2000 or later).**
	OemMinus	**The OEM minus key on any country/region keyboard (Windows 2000 or later).**
	OemOpenBrackets	**The OEM open bracket key on a US standard keyboard (Windows 2000 or later).**
	OemPeriod	**The OEM period key on any country/region keyboard (Windows 2000 or later).**
	OemPipe	**The OEM pipe key on a US standard keyboard (Windows 2000 or later).**

	Oemplus	**The OEM plus key on any country/region keyboard (Windows 2000 or later).**
	OemQuestion	**The OEM question mark key on a US standard keyboard (Windows 2000 or later).**
	OemQuotes	**The OEM singled/double quote key on a US standard keyboard (Windows 2000 or later).**
	OemSemicolon	**The OEM Semicolon key on a US standard keyboard (Windows 2000 or later).**
	Oemtilde	**The OEM tilde key on a US standard keyboard (Windows 2000 or later).**
	P	**The P key.**
	Pa1	**The PA1 key.**
	Packet	**Used to pass Unicode characters as if they were keystrokes. The Packet key value is the low word of a 32-bit virtual-key value used for non-keyboard input methods.**
	PageDown	**The PAGE DOWN key.**
	PageUp	**The PAGE UP key.**
	Pause	**The PAUSE key.**
	Play	**The PLAY key.**
	Print	**The PRINT key.**
	PrintScreen	**The PRINT SCREEN key.**
	Prior	**The PAGE UP key.**
	ProcessKey	**The PROCESS KEY key.**
	Q	**The Q key.**
	R	**The R key.**
	RButton	**The right mouse button.**
	RControlKey	**The right CTRL key.**
	Return	**The RETURN key.**
	Right	**The RIGHT ARROW key.**
	RMenu	**The right ALT key.**

RShiftKey	**The right SHIFT key.**
RWin	**The right Windows logo key (Microsoft Natural Keyboard).**
S	**The S key.**
Scroll	**The SCROLL LOCK key.**
Select	**The SELECT key.**
SelectMedia	**The select media key (Windows 2000 or later).**
Separator	**The separator key.**
Shift	**The SHIFT modifier key.**
ShiftKey	**The SHIFT key.**
Sleep	**The computer sleep key.**
Snapshot	**The PRINT SCREEN key.**
Space	**The SPACEBAR key.**
Subtract	**The subtract key.**
T	**The T key.**
Tab	**The TAB key.**
U	**The U key.**
Up	**The UP ARROW key.**
V	**The V key.**
VolumeDown	**The volume down key (Windows 2000 or later).**
VolumeMute	**The volume mute key (Windows 2000 or later).**
VolumeUp	**The volume up key (Windows 2000 or later).**
W	**The W key.**
X	**The X key.**
XButton1	**The first x mouse button (five-button mouse).**
XButton2	**The second x mouse button (five-button mouse).**

	Y	**The Y key.**
	Z	**The Z key.**
	Zoom	**The ZOOM key.**

Literatura:

Microsoft Visual C# 2005 Step by Step, John Sharp, izdano leta 2005
Microsoft Visual C# 2003 Step by Step, John Sharp, izdano leta 2003
Murach's C# 2005 Training & reference, Joel Murach, Mike Murach & Associates Inc. 2006
Microsoft Visual C# .NET, Mickey Williams, izdano leta 2002
http://www.functionx.com/csharp/
http://penelope.fmf.uni-lj.si/diri0607/index.php/Kategorija:Naloge

Vaje:

http://uranic.tsckr.si/C%23/Vaje%20C%23/
http://penelope.fmf.uni-lj.si/C_sharp/index.php/Razli%C4%8Dne_naloge