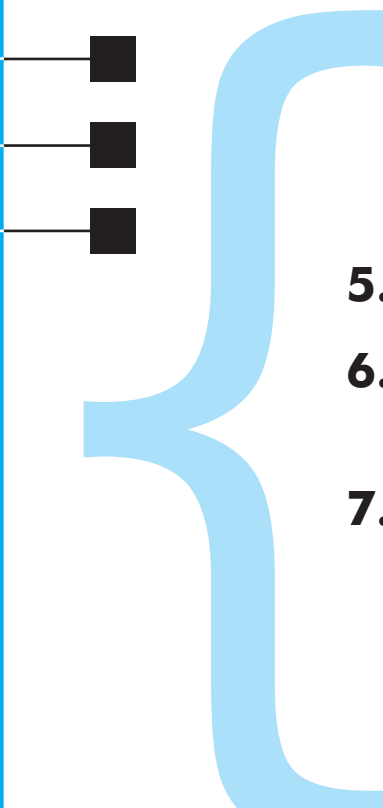


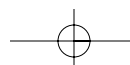
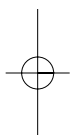
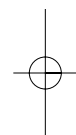
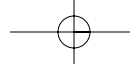
# Osnove programiranja



**5. POGLAVLJE:** TIPOVI PODATAKA

**6. POGLAVLJE:** OBJEKTNO-ORJENTIRANO  
PROGRAMIRANJE

**7. POGLAVLJE:** IZNIMKE



# 5. POGLAVLJE

## Tipovi podataka

### U ovom poglavlju:

- Tipovi podataka u .NET-u
- Pretvaranje tipova podataka
- Što su implicitne, a što eksplicitne pretvorbe
- Korištenje konstanti i pobrojanih članova
- Rad s poljima i kolekcijama
- Što su klase i strukture
- Korištenje delegata

**T**emelj su svakog programa, ma kako jednostavan bio, podaci s kojima radi. To mogu biti obični brojevi u programima koji obavljaju jednostavne kalkulacije, znakovni nizovi u programima koji komuniciraju s korisnikom ili pak neke složenije strukture, poput specifičnih objekata koji predstavljaju nekakav entitet ili polja, koja predstavljaju povezani niz podataka.

Da biste mogli u svojim programima koristiti različite vrste podataka, trebate upoznati tipove podataka koje vam se nude u .NET-u. Imate li pak nekog programerskog iskustva, brzo ćete se snaći i uočiti sve novitete i različitosti, a sve će vam to itekako trebati u programiranju aplikacija za .NET.

## II. DIO: OSNOVE PROGRAMIRANJA

# Korištenje tipova podataka

U .NET Frameworku postoji vrlo precizno definiran sustav tipova podataka. Svi .NET jezici su *strongly typed* – znači da su svi podaci čvrsto povezani sa svojim tipom i da im nije moguće slobodno mijenjati tip i razmjenjivati ih između objekata različitih tipova.

Naravno, zato postoji mogućnost konverzija podataka, koja se može odvijati implicitno (što znači da je nije potrebno navoditi, već se podrazumijeva) ili eksplicitno (tako da vi pozovete metodu za konverziju podataka). Tipičan primjer događa se pri spajanju brojeva i tekstova – želite li ih sve spojiti u jedan veći tekst, trebat ćete prvo eksplicitno pretvoriti sve brojeve u tekstualni tip podataka.

## Tipovi podataka

Dakle, tipovi podataka određuju vrstu podataka koju koristite i pohranjujete u svom programu. Tipovi podataka se prvenstveno dijele u reference i vrijednosti – reference su podaci kojima nije uvijek poznata veličina te zato varijable reference ukazuju na neku lokaciju u memoriji na kojoj je spremljen podatak, dok *vrijednosti* uvijek zauzimaju istu količinu memorije (ovisno o svom tipu, primjerice svaka vrijednost *int* zauzima 4 bajta) te njihove varijable direktno sadrže njihovu vrijednost.

Primjerice, znakovne nizove uvijek možete povećavati, čime dakako zauzimate veću količinu memorije, te su zato oni reference – varijable ukazuju na njihovu lokaciju u memoriji. S druge strane, brojevi uvijek zauzimaju određenu količinu memorije (koja ovisi o tipu broja, odnosno njegovu rasponu, pa mogu zauzimati od 1 do 8 byteova) te je moguće već prije rezervirati točnu količinu memorije za njih. Takve varijable uvijek sadrže vrijednost, a ne pokazivač na mjesto u memoriji otkud počinje neodređeno velik podatak (kao što je slučaj sa znakovnim tipovima).

Vrijednosni podaci se najjednostavnije mogu podijeliti na cjelobrojne, decimalne i logičke tipove te tipove čiji se podaci sastoje od samo jednog znaka (odn. podaci tipa *char*). Reference se dijele na znakovne nizove (tip *string*) i objekte (ili tip *Object*).

## Cjelobrojni tipovi

U tablici 5-1 nalazi se popis dostupnih cjelobrojnih tipova podataka i njihovih raspona.



Vjerojatno ste u tablici 5-1 uočili nepoznate pojmove – *signed* i *unsigned*. Ukoliko je neki broj *signed*, to znači da se jedan njegov bit koristi za predznak te stoga može poprimiti i pozitivne i negativne vrijednosti. Za razliku od njega, *unsigned* tip ne koristi jedan bit za predznak te stoga može poprimiti samo pozitivne vrijednosti, ali zato doseže višu najveću pozitivnu vrijednost.

5. POGLAVLJE: TIPOVI PODATAKA

**Tablica 5-1:**  
**Popis cjelobrojnih tipova podataka i njihovih raspona**

Tip	C# ime	Opis	Raspon
System.Byte	byte	8-bitna unsigned vrijednost (1 bajt)	0 do 255
System.Int16	short	16-bitna signed vrijednost (2 bajta)	-32768 do 32767
System.Int32	int	32-bitna signed vrijednost (4 bajta)	-2 <sup>31</sup> do 2 <sup>31</sup> -1
System.Int64	long	64-bitna signed vrijednost (8 bajtova)	-2 <sup>63</sup> do 2 <sup>63</sup> -1
System.Sbyte	sbyte	8-bitna signed vrijednost (1 bajt)	-128 do 127
System.UInt16	ushort	16-bitna unsigned vrijednost (2 bajta)	0 do 65535
System.UInt32	uint	32-bitna unsigned vrijednost (4 bajta)	0 do 2 <sup>32</sup> -1
System.UInt64	ulong	64-bitna unsigned vrijednost (8 bajtova)	0 do 2 <sup>64</sup> -1

Kako se cijela struktura .NET organizira u hijerarhiju klasa, tako i svaki podatak ima odgovarajuću klasu, koja je u tablici 5-1 navedena u stupcu *Tip*. Dakle, najočitije bi bilo na sljedeći način deklarirati varijable:

```
System.Int32 broj = new System.Int32();  
broj = 15;
```

No to je ipak malčice prekomplikirano za korištenje, pogotovo pri radu s nečim tako jednostavnim kao što su varijable. Zato u C#-u postoji odgovarajuća ključna riječ koja služi za deklaraciju varijabli, a navedena je u stupcu *C# ime*. Koristeći je možete mnogo lakše deklarirati varijable:

```
int broj;  
broj = 15;
```

Ako baš želite, cjelobrojnim tipovima podataka možete pridjeljivati i heksadecimalne vrijednosti tako da im dodate prefiks 0x. Primjerice, želite li kao u prethodnom primjeru postaviti varijablu broj na 15, napisat ćete:

```
broj = 0xF;
```



II. DIO: OSNOVE PROGRAMIRANJA

Tipovi s pomičnim zarezom

Želite li u svojim programima koristiti realne brojeve (ili decimalne, odnosno brojeve s pomičnim zarezom), koristit ćete neki od tipova iz tablice 5-2.

**Tablica 5-2:**  
**Popis tipova s pomičnim zarezom te njihove preciznosti i rasponi**

Tip	C# ime	Opis	Preciznost	Približni raspon
System.Single	float	32-bitna vrijednost	7 značajnih znamenki	-3.4*10 <sup>38</sup> do -1.4*10 <sup>-45</sup> i 1.4*10 <sup>-45</sup> do 3.4*10 <sup>38</sup>
System.Double	double	64-bitna vrijednost	15-16 značajnih znamenki	-1.7*10 <sup>308</sup> do -5.0*10 <sup>-324</sup> i 5.0*10 <sup>-324</sup> do 1.7*10 <sup>308</sup>
System.Decimal	decimal	128-bitna vrijednost	28 značajnih znamenki	-7.9*10 <sup>28</sup> do -1.0*10 <sup>-28</sup> i 1.0*10 <sup>-28</sup> do 7.9*10 <sup>28</sup>

Tip podataka *System.Single* prikladan je za jednostavne izračune koji ne zahtijevaju veliku preciznost rezultata, no ukoliko vam je ona relativno bitna, upotrijebit ćete tip *System.Double* koji, uz povećanu preciznost, može spremiti brojeve ogromnog raspona. Bavite li se izrazito preciznim izračunima, morat ćete ipak upotrijebiti vrijednost *System.Decimal*, koja pruža daleko najveću preciznost rezultata (čak 28 značajnih znamenki)

Logički tip

Želite u svojem programu koristiti jednostavni tip podataka, koji omogućava spremanje samo vrijednosti *istinito* ili *neistinito*, koristit ćete *System.Boolean* odnosno tip podataka *bool*.

```
bool logika; // deklaracija varijable "logika"
logika = true; // postavljanje na istinitu vrijednost

if (logika) // ili "if (logika == true)"
{
    Console.WriteLine("Istinito!");
}
```

## Preciznost ili nepreciznost, pitanje je sad

U tablici 5-2 nalazi se pojam preciznosti vrijednosti odnosno broj značajnih znamenki (engl. *significant digit*). Objasnimo sve na broju 4215.02474. Ukoliko se broj spremi s preciznošću od dvije značajne znamenke, dobit ćemo 4200; ukoliko se koristi preciznost od tri značajne znamenke, dobit ćemo 4220; ukoliko se pak koristi 5 značajnih znamenki, dobit će se 4215.0; ukoliko se koristi 7 značajnih znamenki, dobivamo 4215.025.

Ukoliko takve vrijednosti prikazete s jednom znamenkom prije točke (kako se i spremaju u ra-

čunalu), odnosno  $4.2 \cdot 10^3$ ,  $4.22 \cdot 10^3$ ,  $4.2150 \cdot 10^3$  i  $4.215025 \cdot 10^3$ , uočite primjenu i svrhu značajnih znamenki.

Dakle, dok *float* podaci mogu pamtit i brojeve s preciznošću od 7 znamenki (primjerice,  $4.215025 \cdot 10^3$ ), *decimal* je mnogo točniji i može spremiti broj poput  $4.215024749202391358192357189 \cdot 10^3$  (naravno, uočite da  $10^3$  uopće nije bitno za primjer i da tu može stajati bilo koja potencija, sve dok je konačni broj unutar odgovarajućeg raspona).

Naravno, osim istinite vrijednosti ili *true*, u C#-u možete definirati i neistinitu vrijednost odnosno *false*. Logički tip podataka je posebno koristan u *if*-naredbama i sličnim provjeravanjima ispunjenosti određenih uvjeta.

## Znakovni tipovi

U .NET-u imate mogućnost raditi s dva znakovna tipa – dok jedan služi za rad samo s jednim znakom (*char* tip), drugi koristi za spremanje znakovnih nizova (tip *string*). Vrijednosti varijabli tipa *char* upisujete korištenjem jednostrukih navodnika ('), a podatke tipa *string* upisujete korištenjem dvostrukih navodnika.

```
char mojZnak;
mojZnak = 'A';

string mojNiz;
mojNiz = "Dobar dan!";
```

Kako su znakovni nizovi podatak koji ćete najviše koristiti u komunikaciji s korisnikom za ispis podataka, nužno je poznavati njegove mogućnosti. Primjerice, spajanje dvaju nizova u jedan zove se konkatencija i obavlja se pomoću "+" operatora, baš kao i pri zbrajanju brojeva.

## II. DIO: OSNOVE PROGRAMIRANJA

```
string mojNiz1 = "Dobar ";
string mojNiz2 = "dan!";
string mojNiz3;

mojNiz3 = mojNiz1 + mojNiz2; // "Dobar dan!"
```

Naravno, tip podataka *string* ima ugrađen cijeli niz metoda koje možete koristiti pri radu, a važnije su opisane u tablici 5-3.

**Tablica 5-3:**  
**Važnije metode string tipa podataka**

Metoda	Opis
Insert	umeće neki znakovni niz na određenu poziciju trenutnog znakovnog niza
PadLeft, PadRight	dodaju znakove na lijevu stranu (početak) i desnu stranu (kraj) niza
Remove	briše određen broj znakova iz niza
Replace	zamjenjuje sva pojavljivanja nekog znaka u nizu s nekim drugim znakom
Substring	vraća podniz znakovnog niza
ToLower, ToUpper	pretvaraju sve znakove niza u mala ili velika slova
Trim	briše sve suvišne <i>whitespace</i> znakove (razmaci, tabovi, itd.) s početka i kraja niza

Sve opisane metode koriste se nad nekom varijablom tipa *string*, kao što je prikazano na slici 5-1. Primjerice, da biste ispisali prvih pet znakova nekog niza (zbog jednostavnosti, nastavit ćemo prethodni primjer), iskoristit ćete metodu *Substring*:

```
string mojNiz = mojNiz3.Substring(0, 5);
```

Metoda *Substring* prima dva parametra – prvi je znak od kojeg počinje podniz (0 označava prvi znak, 1 je drugi itd.), a drugi je broj znakova podniza. Stoga, primijenimo li prethodnu naredbu nad nizom "Dobar dan!", rezultat će biti "Dobar".

U svojim znakovnim nizovima možete koristiti i različite posebne znakove (tzv. *escape-znakove*). Primjerice, želite li u aplikaciji u konzoli izbrisati jedan znak s ekrana, ispisat ćete znak *backspace* koji će učiniti baš to. Želite li prijeći u novi red, ispisat ćete tzv. znak *newline*. Neki od tih znakova opisani su u tablici 5-4.



## 5. POGLAVLJE: TIPOVI PODATAKA

Znakovni niz	Opis
\'	jednostruki navodnik
\"	dvostruki navodnik
\\	znak backslash
\b	znak backspace za brisanje znaka lijevo od pokazivača
\n	znak newline za prelazak u naredni redak
\t	znak za pomak od nekoliko mjesta (znak tab)

**Tablica 5-4:**  
**Neki od posebnih znakova**  
**koje možete koristiti u svo-**  
**jim znakovnim nizovima**

Uočite da se u tablici 5-4 koristi znak *backslash* za označavanje svih posebnih znakovnih nizova (naprimjer: \', \t itd.). Stoga je potreban i poseban znakovni niz ukoliko se želi ispisati sam *backslash* – \\.



Primjerice, da biste unutar niza ispisali jednostruke i dvostruke navodnike, morat ćete iskoristiti sljedeću sintaksu:

```
// Uš'o medo u dućan i reče: "Dobar dan!"
mojNiz = "Uš\'o medo u dućan i reče: \"Dobar dan!\"";
```

## Objekt

Tip podataka *System.Object* je glavni tip podataka u .NET Frameworku i svi drugi tipovi su izvedeni iz njega. On ima 4 metode, koje zatim nasljeđuju svi drugi tipovi podataka i mogu se koristiti nad bilo kojom varijablom u .NET-u: *Equals* (provjerava jednakost između dvije instance), *GetHashCode* (vraća *hash* kôd), *GetType* (vraća objekt *type* koji govori o kojem se tipu podataka radi) i *ToString* (vraća tekstualni opis objekta).

Korištenje tipa *object* jednostavno je i u njega se mogu spremiti bilo kakve vrijednosti:

```
object mojObjekt;
mojObjekt = "Dobar dan!";
mojObjekt = 500;
mojObjekt = new System.Xml.XmlDocument();
```

No da biste mogli koristiti funkcionalnost pojedinog tipa podataka, morat ćete objekt pretvoriti u taj tip. Primjerice, da biste mogli u prvom primjeru raditi s podnizovima ili zamjenjivati znakove,

## II. DIO: OSNOVE PROGRAMIRANJA

morat ćete ga pretvoriti u objekt *string*. Slično ćete i drugi primjer morati pretvoriti u neki brojčani tip da biste ga mogli koristiti u računskim operacijama. Naravno, isto vrijedi i za treći primjer, koji biste trebali pretvoriti u tip *XmlDocument*.

Ključna stvar kod korištenja tipa podataka *object* je *boxing*. Radi se, naime, o implicitnoj konverziji vrijednosnih tipova podataka u reference. Primjerice, kako tip *object* predstavlja nadtip svim podacima u .NET-u, sve varijable možete pretvoriti u tip *object* i koristiti ih kao reference. Evo i primjera:

```
int Broj = 100;
object Objekt;
Objekt = Broj;
```

Dakle, cjelobrojnu varijablu smo pretvorili u objekt tako što smo njenu vrijednost pridružili objektu. Želimo li pak provjeriti što se nalazi u samoj *kutiji*, tj. objektu, možemo napisati:

```
if (Objekt is int)
{
    Console.WriteLine("Objekt sadrži broj!");
}
```

Suprotna operacija odnosno *unboxing* je pretvaranje natrag u originalni tip podataka. To je pak eksplicitna operacija odnosno morate u kôdu naznačiti da želite provesti pretvorbu podataka, za razliku od operacije *boxing* koja je implicitna, jer samo trebate pridiijeliti varijablu objektu i automatski će se dogoditi pretvorba.

## Pretvaranje tipova podataka

Mnogo puta ćete u svom kôdu koristiti podatke koje je potrebno pretvoriti u neki drugi tip. Recimo, radit ćete s brojevima, obavljati različite operacije, a zatim ćete htjeti ispisati poruku "Konačni rezultat je xyz". Očito ćete dobiveni broj morati pretvoriti u znakovni niz i zatim ga spojiti s tekстом "Konačni rezultat je".

To je samo jedna od primjena. Ponekad ćete obavljati i pretvorbe među sličnim tipovima podataka, koje nisu previše uočljive, a imaju velik utjecaj na kôd. Tako ćete ponekad pretvarati cijele brojeve u one s pomičnim zarezom da biste dobili točan rezultat pri, recimo, dijeljenju.

Konverzija podataka iz jednog tipa u drugi može biti obavljena na dva načina – *implicitno*, što znači da se konverzija obavlja automatski, i *eksplicitno*, što znači da se konverzija obavlja na vaš zahtjev odnosno korištenjem posebnih naredbi u kôdu.

## Implicitne pretvorbe

Implicitna pretvaranja između tipova podataka obavljaju se uvijek kad pri pretvaranju ne može doći do gubitka podataka.

5. POGLAVLJE: TIPOVI PODATAKA

```
int Broj1 = 50;
long Broj2;

Broj2 = Broj1;
```

Kako tip brojeva *long* ima mnogo veći raspon nego običan tip *int*, pri konverziji neće sigurno doći do gubitka podataka. Tad se konverzija obavlja na jednostavan način – na mjestu gdje je bio očekivan tip podatka *long* (u primjeru je to iza znaka jednakosti, no može biti i na mjestu parametra pri pozivu metode itd.), postavi se tip podatka *int*, a pretvorba iz *int* u *long* se obavlja automatski. U tablici 5-5 je popis mogućih implicitnih konverzija.

**Tablica 5-5:**  
**Popis konverzija koje će se obaviti implicitno**

Izvorni tip podataka	Implicitna konverzija moguća u tipove
byte	short, ushort, int, uint, long, ulong, float, double, decimal
short	int, long, float, double, decimal
int	long, float, double, decimal
long	float, double, decimal
float	double
char	int, uint, long, ulong, float, double, decimal
sbyte	short, int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
uint	long, ulong, float, double, decimal <ul style="list-style-type: none"><li>ulong float, double, decimal</li></ul>

Svako pridjeljivanje neke varijable tipa podatka iz lijevog stupca varijabli koja je tipa navedenog u desnom stupcu obaviti će se implicitno – bez ikakvog dodatno upozoravanja i bez gubitka podataka.

Eksplisitne pretvorbe

Ukoliko se neke pretvorbe ne mogu obaviti implicitno, trebat će ih obaviti korištenjem posebnih naredbi. Najjednostavnije je to napraviti *castanjem* neke varijable u drugi tip podataka. To se obavlja tako da se ispred same varijable u zagradama stavi novi tip podataka.

```
long Broj1 = 50;
int Broj2;

Broj2 = (int) Broj1;
```

## II. DIO: OSNOVE PROGRAMIRANJA

Primijetite da dok je pretvaranje iz *int* u *long* moguće, jer pritom nema gubitaka podataka, prethodni primjer je riskantan – iako i *long* i *int* mogu spremiti vrijednost 50 korištenu u primjeru, da se koristila neka vrijednost izvan opsega tipa *int* došlo bi do gubitka podataka. Pogledajte primjer:

```
int Broj1 = 10000;
byte Broj2;

Broj2 = (byte) Broj1;
```

Kako je opseg tipa podatka *byte* mnogo manji od vrijednosti koja se u njega pokušava spremiti (10000), doći će do gubitka podataka – ispišete li varijablu *Broj2*, uvidjet ćete da je u njoj spremljena vrijednost 16.



Kao što ste vidjeli u primjerima, lako se dogodi gubitak podataka pri eksplicitnim konverzijama. Budite vrlo pažljivi pri takvim pretvorbama i koristite ih samo kad ste sigurni u vrijednosti koje će biti spremljene. Također, preporučuje se i ugradnja mehanizma za hvatanje iznimaka, koje se mogu dogoditi ukoliko se pokuša izvršiti nedozvoljena eksplicitna konverzija. O tim ćete mehanizmima naučiti više u sljedećim poglavljima.

Ponekad ćete poželjeti obaviti pretvorbe za koje ne možete utvrditi hoće li pri njima doći do gubitka podataka, jer se to može znati samo za slične tipove. No što ako želite pretvoriti broj u znakovni niz ili obrnuto? Tad je nemoguće znati hoće li doći do gubitka i nužno je koristiti posebnu naredbu za konverziju.

Za to će vam poslužiti klasa *Convert* i njene metode. Njome vam je omogućeno pretvaranje između svih tipova:

```
int Broj = 100;
string Niz;

Niz = Convert.ToString(Broj);
```

Dakle, treba samo pozvati odgovarajuću metodu za pretvaranje i za parametar joj proslijediti varijablu koju želite pretvoriti.

## Konstante, polja i kolekcije

Dosad ste upoznali sve osnovne tipove podataka u .NET-u, no često će vaše aplikacije zahtijevati i složenije tipove, bilo zbog jednostavnosti i lakšeg korištenja ili zbog elegancije u programiranju.

## 5. POGLAVLJE: TIPOVI PODATAKA

### Slika 5-2:

**Pregriš metoda za pretvaranje u klasi Convert**



Za pretvaranje u znakovni niz mnogo je jednostavnije iskoristiti metodu `ToString()` koja je dostupna svakoj varijabli, primjerice:

```
int Broj = 100;
string Niz = Broj.ToString();
```

Nju možete koristiti i pri konkatenuiranju nekog broja sa znakovnim nizom. Da biste ih spojili u jedan znakovni niz, morat ćete broj pretvoriti u tip *string*. To ćete često raditi pri ispisivanju, primjerice:

```
Console.WriteLine("Broj je jednak " + Broj.ToString());
```



Na narednim stranicama upoznat ćete se s poljima i kolekcijama, dvjema sličnim strukturama čija je glavna namjena grupiranje srodnih podataka. Krenimo ipak za početak s jednostavnijim stvarima – s konstantama.

## Konstante

Na konstante se može gledati kao na varijable koje svoju vrijednost nikad ne mijenjaju. Njihova je primjena jasna – imate li vrijednosti koje su fiksne (primjerice, za računanje površine kruga, koristit ćete konstantnu varijablu u kojoj je spremljena vrijednosti  $\pi$ ), pridijelit ćete ih nekoj konstanti koju lako možete koristiti iz bilo kojeg mjesta u kôdu. Želite li negdje kasnije u razvoju programa promijeniti tu konstantu, nećete trebati prolaziti cijeli kôd i mijenjati njenu vrijednost, već ćete samo promijeniti deklaraciju konstante i pridijeliti joj novu vrijednost.

Konstante se definiraju s ključnom riječi *const*. Njihovim korištenjem mnogo ćete se lakše snalaziti u kôdu i lakše će vam biti pronaći greške.

## II. DIO: OSNOVE PROGRAMIRANJA

```
const double Pi = 3.14159265;
const int BrzinaSvjetlosti = 300000;
const string ImePrograma = "Moj program";
```

Kao što uočavate, konstante su obične varijable ispred kojih je napisano *const*.



**Konstantama ne možete mijenjati vrijednost u svom programu – pokušate li izvesti sljedeće naredbe, kompajler će vam javiti grešku:**

```
BrzinaSvjetlosti = 299999;
ImePrograma = "Najbolji program";
```

Konstante možete koristiti u svom programu kao i sve druge varijable, uz spomenuto ograničenje koje vam zabranjuje mijenjanje njihovih vrijednosti.



**Ukoliko stvarno želite koristiti konstantu PI, ne trebate pisati svoju verziju već možete iskoristiti predefiniranu u klasi *Math*, kojoj možete pristupiti s *Math.PI*.**

## Pobrojani članovi

Za pisanje čitkog i urednog kôda, osim korištenja konstanti, mogu vam pomoći i pobrojani članovi (engl. *enumerations*). Njihova je svrha vrlo slična konstantama – oni zapravo predstavljaju niz povezanih konstanti s logičnim imenima. Tipičan primjer su dani u tjednu – želite li u svom kôdu svakom danu pridijeliti redni broj, možete koristiti 7 različitih konstanti, a možete ih povezati u pobrojen niz.

Za definiranje pobrojanog niza ključna je riječ *enum*, koja zapravo predstavlja tip podatka. Unutar vitičastih zagrada navode se elementi niza s odgovarajućim vrijednostima.

```
enum Tjedan
{
    Ponedjeljak = 1,
    Utorak = 2,
    Srijeda = 3,
    Cetvrtak = 4,
```

## 5. POGLAVLJE: TIPOVI PODATAKA

```
Petak = 5,
Subota = 6,
Nedjelja = 7
}
```

Kao što primjećujete, radi se o konstantama koje se nalaze unutar pobrojanog niza *Tjedan*, a svaka ima svoju vrijednost.

Ukoliko ne navedete drugačije, sve će vrijednosti pobrojanog niza biti *int*. Naravno, vi imate na raspolaganju i definiranje drugačijeg tipa, pa pobrojani nizovi mogu biti i tipa *byte*, *short* ili *long*. Da biste definirali niz drugačijeg tipa, primjerice *short*, iskoristit ćete sljedeću sintaksu:

```
enum Tjedan : short
{
    // elementi niza
}
```

**Imajte na umu da su pobrojani nizovi posebni tipovi podataka i da njih ne možete definirati unutar neke metode. Oni mogu biti definirani isključivo unutar neke klase, dakle izvan svih njenih metoda.**



Naravno, sve pobrojane nizove možete jednostavno koristiti u svom kódu – umjesto da pišete vrijednost svake konstante, jednostavno ćete napisati ime pobrojenog člana. No budite oprezni – u C#-u ih morate prije korištenja obavezno pretvoriti u tip podataka koji sadržavaju.

```
int DanasnjiDan = 3;
// int DanasnjiDan = (int) Tjedan.Srijeda;

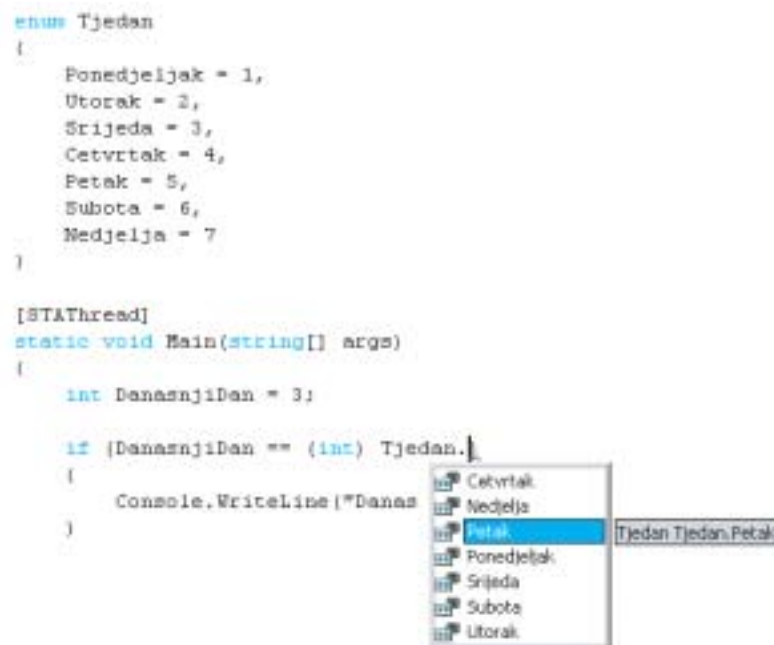
if (DanasnjiDan == (int) Tjedan.Srijeda)
{
    Console.WriteLine("Danas je srijeda!");
}
```

Vrlo je važno uočiti *castanje* u tip podatka *int* prije same usporedbe, jer takve podatke sadržava pobrojani niz. U drugoj liniji u komentarima je navedena naredba koja je po funkcionalnosti identična prvoj, no mnogo elegantnija i čitljivija.

Pobrojani nizovi su vrlo korisni i za stvaranje varijabli. Tako možete definirati varijablu koja je tipa *Tjedan*, a ona će moći sadržavati samo vrijednosti koje su definirane u pobrojanom nizu. Deklaracija takve varijable ni po čemu se ne razlikuje od drugih varijabli:

## II. DIO: OSNOVE PROGRAMIRANJA

**Slika 5-3:**  
*Visual Studio .NET će vam pri radu s pobrojanim nizovima ponuditi popis svih njihovih članova složenih po abecedi.*



```
Tjedan mojDanTjedna;
```

Takvim varijablama nećete moći pridijeliti vrijednost direktno pišući broj koji predstavlja vrijednost nekog dana, već ćete morati iskoristiti sam pobrojani niz.

```
mojDanTjedna = Tjedan.Petak;

if (mojDanTjedna == Tjedan.Utorak)
{
    // itd.
}
```

Već vjerojatno i sami uočavati koliko je čitljiviji ovakav kôd, za razliku od onog u kojem biste direktno koristili vrijednosti. Imate li u svom kôdu potrebu za ograničavanjem neke varijable samo na određen set mogućih vrijednosti (u našem primjeru, varijabla `mojDanTjedna` mogla bi poprimiti samo neku od vrijednosti iz pobrojanog niza `Tjedan`), pobrojani nizovi su jedini pravi izbor.

Kao što ćete vidjeti u narednim poglavljima, u baznim klasama .NET-a postoji mnoštvo pobrojanih nizova koji uglavnom spremaju vrijednosti o mogućim stanjima ili svojstvima nekog objekta, stoga je bitno znati o čemu se radi i kako sami možete načiniti pobrojane nizove.



## 5. POGLAVLJE: TIPOVI PODATAKA

Ponekad nije potrebno eksplicitno ispisati vrijednosti svakog člana pobrojanog niza. Ukoliko ne navedete drukčije, prvi član će imati vrijednost 0, drugi će imati vrijednost 1, treći 2 itd.

```
enum Brojevi
{
    Nula, // 0
    Jedan, // 1
    Dva // 2
}
```



## Polja

Za razliku od pobrojanih nizova u kojima su sve vrijednosti bile predefinirane, u programiranju možete koristiti i polja (u literaturi još možete pronaći i izraz *matrice*, od engl. *array*). Radi se o grupi srodnih vrijednosti istog tipa, kojima možete pristupiti korištenjem njihovog indeksa odnosno rednog broja.

Uzmimo jednostavan primjer: ukoliko želite u svojoj aplikaciji pobrojati sve učenike nekog razreda, iskoristit ćete polje. Definirat ćete da broj članova polja odgovara broju učenika u razredu, a zatim ćete za svaku vrijednost polja upisati ime pojedinog učenika u razredu. Tada ćete sve učenike razreda imati u samo jednoj varijabli, u polju, a svakom učeniku ćete moći pristupiti prema njegovom rednom broju u polju.

**Upamtite** – za polja u C#-u (kao i u većini drugih programskih jezika) kažemo da su *zero-based*. To znači da prvi član polja ima indeks 0, drugom se može pristupiti korištenjem indeksa 1 itd.



Deklaracija i inicijalizacija polja razlikuje se od deklaracije običnih varijabli. Tako ćete morati navesti broj članova polja unutar uglatih zagrada, a uglate zagrade ćete morati i staviti kod tipa polja. Primijetite da svako polje ima svoj tip koji određuje njegove članove. Tako će polje *int* moći sadržavati samo cjelobrojne vrijednosti, polje *string* samo znakovne nizove itd.

Nemojte da vas zbuni broj članova koji navodite pri deklaraciji polja. On označava ukupan broj članova, a zbog činjenice da prvi član polja ima indeks 0, zadnji će član imati indeks za jedan manji od ukupnog broja članova. Sljedeći primjer definira polje koje ima 10 članova, s indeksima od 0 do 9.

## II. DIO: OSNOVE PROGRAMIRANJA

```
int[] mojePolje = new int[10];
```

Prethodni je primjer tako identičan dužoj verziji po kojoj deklarirate polje u jednoj liniji, a inicijalizirate na određenu veličinu u narednoj liniji (naravno, te linije mogu biti međusobno udaljene u kódu):

```
int[] mojePolje;  
  
// ...  
  
mojePolje = new int[10];
```



**Veličinu nekog polja možete bilo kad u kódu promijeniti, no pripazite jer u tom slučaju gubite vrijednosti svih članova polja. Sintaksa promjene veličine polja jednaka je njegovoj početnoj inicijalizaciji:**

```
mojePolje = new int[20];
```

Vrijednosti polja pišete i čitate korištenjem njihovih indeksa.

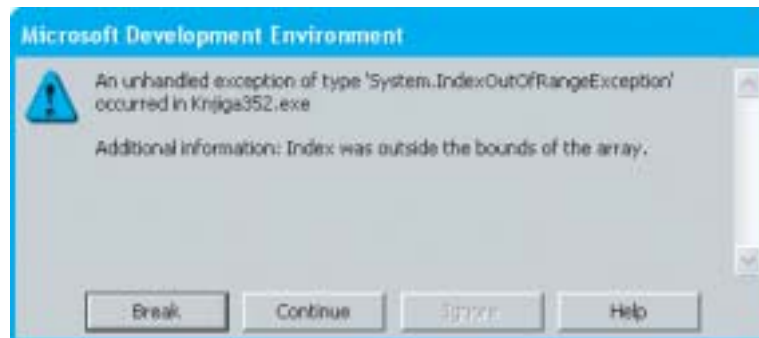
```
mojePolje[0] = 50;  
mojePolje[1] = 34;  
mojePolje[2] = 17;  
  
// Ispisat će "34"  
Console.WriteLine(mojePolje[1]);  
  
mojePolje[10] = 12; // Greška!
```

Ukoliko pokušate pristupiti nepostojećem članu polja korištenjem indeksa koji je izvan dopuštenih granica (u prethodnom primjeru indeks 10 je nevaljan – iako postoji 10 članova polja, njihovi se indeksi kreću od 0 do 9), kompajler vam neće javiti grešku, već će se ona dogoditi pri izvršavanju programa, stoga treba biti veoma oprezan!

Želite li pak ispisati veličinu polja, na raspolaganju vam stoji svojstvo *Length* varijable polja. Ono je tipa *int*, jer sadrži broj, pa ćete ga morati pretvoriti u tip *string*, ukoliko ga želite ispisati uz poruku:

```
Console.WriteLine(mojePolje.Length);  
Console.WriteLine("Polje ima " + mojePolje.Length.ToString() + " članova.");
```

## 5. POGLAVLJE: TIPOVI PODATAKA



**Slika 5-4:**  
**Greška koja se pojavljuje pri izvršavanju programa kad se pokuša pristupiti nepostojećem članu polja**

Svojstvo *Length* možete koristiti i za ispis svih članova polja. Jednostavno ćete proći od nultog člana do zadnjeg člana (kojem je indeks za jedan manji od broja članova polja) i ispisati članove polja korištenjem brojača za označavanje indeksa:

```
for (int i = 0; i <= mojePolje.Length - 1; i++)
{
    Console.WriteLine(mojePolje[i]);
}
```

Priča o podjeli tipova podataka na vrijednosne tipove koji sadržavaju vrijednost podataka (npr. *int*, *bool*, *long*) i reference koje sadrže samo pokazivač na vrijednost podatka smještenu negdje drugdje u memoriji (*object* i *string*) sad postaje važna. Ukoliko deklarirate da polje sadržava 10 podataka vrijednosnog tipa, u memoriji će se zauzeti 10 mjesta i oni će automatski biti postavljeni na svoju početnu vrijednost – primjerice, ukoliko definirate polje od 10 vrijednosti tipa *int*, zapravo ćete stvoriti polje koje za 10 članova ima vrijednost 0. No ukoliko definirate polje referenci (npr. *string*), podaci neće imati postavljenu početnu vrijednost. To je i očekivano, jer će svaki član polja zapravo biti referenca na neku lokaciju u memoriji koja sadržava samu vrijednost. Kako vi niste eksplicitno postavili vrijednost svakog člana, radit će se o *null*-pokazivačima, koji, kao što im i ime kaže, ne pokazuju nikamo.



Prethodni su primjeri radili samo s jednodimenzionalnim poljima, no vi na to niste ograničeni. U .NET-u imate podršku za pravokutna i zupčasta polja, a nazvana su po svom obliku. Radi se o višedimenzionalnim poljima – sve ćemo objasniti na lako shvatljivom primjeru od dvije dimenzije, no sve rečeno vrijedi i za više dimenzija.

## II. DIO: OSNOVE PROGRAMIRANJA

### Pravokutna polja

Na dvodimenzionalno polje se može gledati kao na tablicu koja se sastoji od redaka i stupaca. U pravokutnom polju svi će reci imati jednak broj stupaca, što i očekujemo od tablice. Evo kako biste definirali polje od 2 retka i 4 stupca:

```
int[ , ] mojaTablica = new int[2, 4];
```

Uočite da sad unutar uglatih zagrada postoje dva parametra – pri definiranju tipa oni su ostavljeni praznima, a pri inicijalizaciji su u njih upisane dimenzije. Članovima polja pristupate na očekivani način:

```
mojaTablica[0,0] = 0;
mojaTablica[1,3] = 5;
```



**Umjesto upisivanja dimenzija pri inicijalizaciji polja, možete odmah upisati same vrijednosti polja, čime direktno dajete do znanja dimenzije. Primjerice, jednodimenzionalno polje od 4 člana možete definirati na jedan od sljedeća dva načina:**

```
int[] mojePolje = new int[] {1, 2, 3, 4};
int[] mojePolje = {1, 2, 3, 4}
```

**Dakle, za definiranje članova polja pri inicijalizaciji koriste se vitičaste zagrade. Na sličan način možete definirati i dvodimenzionalno polje 3 x 3 (indeksi, dakle, idu od 0 do 2, a vrijednosti će biti izračunate kao tablica množenja):**

```
int[ , ] mojaTablica = { {1, 2, 3}, {2, 4, 6}, {3, 6, 9} };
```

Kad smo spomenuli da možete definirati višedimenzionalna polja, nismo se šalili – naime, sljedeći primjer definira polje od pet dimenzija, 5 x 3 x 8 x 1 x 9, no upitno je kako biste se u njemu snašli zbog teškog predočivanja više dimenzija.

```
int[ , , , , ] petDimenzija = new int[5, 3, 8, 1, 9];
```

### Zupčasta polja

Zupčasta polja mnogo su slobodnija što se tiče dimenzija. Zupčasto polje od dvije dimenzije nema isti broj elemenata u recima. Primjerice, da bi učenik prikazao svoje ocjene iz nekoliko predmeta, morat će koristiti zupčasto polje, jer neće svaki predmet imati isti broj ocjena. Prvi predmet (redak) će imati, recimo, 3 ocjene (stupca), drugi predmet će imati 6 ocjena, a treći samo 1 ocjenu.

5. POGLAVLJE: TIPOVI PODATAKA

Zbog takve specifičnosti, stvaranje zupčastih polja razlikuje se od stvaranja običnog polja. Na to se može gledati kao na stvaranje jednodimenzionalnog polja koje za svoje članove ima jednodimenzionalna polja. Tako će prvi redak sadržavati polje od 3 elementa, drugi polje od 6 elemenata, a treći polje od 1 elementa.

Evo kako biste definirali takvo polje:

```
int[] [] mojeOcjene = new int[3] [];  
mojeOcjene[0] = new int[] { 5, 4, 5 };  
mojeOcjene[1] = new int[] { 3, 4, 4, 5, 3, 5 };  
mojeOcjene[2] = new int[] { 4 };
```

Želite li saznati neku od dimenzija, jednostavno iskoristite *Length* svojstvo:

```
// Ispisat će "3"  
Console.WriteLine(mojeOcjene.Length);  
  
// Ispisat će "6"  
Console.WriteLine(mojeOcjene[1].Length);
```

# Kolekcije

Ukoliko ste počeli raditi s poljima, vrlo brzo ste naišli na njihov nedostatak. Naime, poljima niste mogli mijenjati veličinu tako da ostanu sačuvane sve vrijednosti koje sadržavaju (primjerice, povećati broj članova polja s 10 na 20 da ostanu zapisane vrijednosti prvih 10 članova). To je zapravo vrlo važno – često na početku izvršavanja ne znate koliko će članova imati neko polje, već želite dodavati članove po potrebi.

Kolekcija Opis	
BitArray	kolekcija bitova (0 ili 1)
Hashtable	kolekcija parova <i>ključ</i> i <i>vrijednost</i> organiziranih po <i>hash</i> -kódu ključa
Queue	red; kolekcija organizirana na FIFO principu ( <i>first-in, first-out</i> )
SortedList	kolekcija koja omogućava pristup elementima, uz standardno po ključu, i po njihovu indeksu (kolekcija je poredana)
Stack	stog; kolekcija organizirana na FILO principu ( <i>first-in, last-out</i> )

**Tablica 5-6:**  
**Korisni tipovi kolekcija**  
**dostupnih unutar**  
**System.Collections**

## II. DIO: OSNOVE PROGRAMIRANJA

Kolekcije vam pružaju baš to – omogućavaju vam stvaranje grupe objekata koja se dinamički puni i briše. Razlika se očituje i u tome što kolekcije neće posjedovati svojstvo *Length*, jer je njihova veličina promjenjiva, već će imati *Count*, jer on preciznije označava trenutnu količinu elemenata u kolekciji.

Kolekcije se nalaze unutar bazne klase *System.Collections*, koja pruža svu spomenutu funkcionalnost. U nastavku ćemo korištenje kolekcija objasniti na primjeru kolekcije *ArrayList*, no na raspolaganju vam stoji još nekoliko tipova kolekcija prikazanih u tablici 5-6, svaka sa svojom funkcionalnošću (osnovne metode korištene u sljedećim primjerima su iste), no njihovo proučavanje ostavljamo vama.

Sve ćemo objasniti na primjeru razreda. Napisat ćemo program koji će koristiti kolekciju *ArrayList* za praćenje učenika u razredu. Pretpostavka je, dakle, da na početku rada programa ne znamo koliko učenika ima razred – učenike ćemo upisivati i dodavati u kolekciju sve dok se ne upiše riječ "KRAJ".

```
System.Collections.ArrayList Razred = new System.Collections.ArrayList();
string ime;

while (true)
{
    Console.Write("Upišite ime učenika: ");
    ime = Console.ReadLine();
    if (ime.ToUpper() == "KRAJ") break;
    Razred.Add(ime);
}
```

Dakle, prvo smo deklarirali varijablu *Razred*, koja je tipa *System.Collections.ArrayList* odnosno kolekcija *ArrayList*. Koristimo i pomoćnu varijablu *ime*, u koju ćemo zapisivati upisana imena svih učenika razreda.



**Definiciju kolekcije možete i pojednostaviti, tako da na početku programa napišete:**

```
using System.Collections;
```

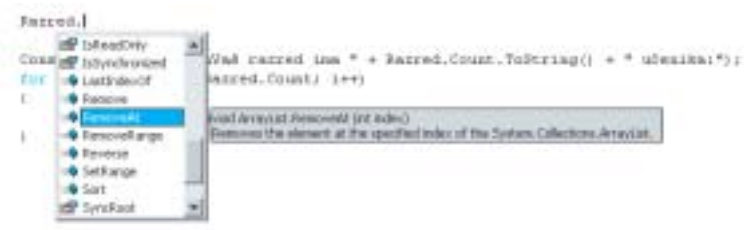
**Sad novu kolekciju možete stvoriti na sljedeći način, bez nepotrebnog ponavljanja:**

```
ArrayList Razred = new ArrayList();
```

Unos imena je riješen beskonačnom petljom. Unos se učitava u varijablu *ime*, a ukoliko je upisano "kraj" ili "KRAJ" (primijetite korištenje funkcije za pretvaranje u velika slova), petlja se prekida. U

## 5. POGLAVLJE: TIPOVI PODATAKA

početku programa kolekcija je prazna, a novi elementi se dodaju metodom *Add*, koja za parametar prima novi element.



**Slika 5-5:** *ArrayList* kolekcija skriva mnoštvo drugih metoda i svojstava, no ovdje smo objasnili samo najvažnije, koji je razlikuju od običnog polja.

Uočite da kolekcija *ArrayList* nigdje nema definiran svoj tip. Naime, ona može primiti bilo koji tip podataka zahvaljujući *boxingu* – proslijeđeni se podaci interno pretvaraju u tip *object*, pa je svejedno kojeg su tipa članovi kolekcije.



Da bismo saznali koliko članova ima kolekcija, iskoristit ćemo njeno svojstvo *Count*.

```
Console.WriteLine("Vaš razred ima " + Razred.Count.ToString() + " učenika.");
```

Pristup elementima kolekcije moguć je preko indeksa, baš kao i pri korištenju običnih polja. Sljedeći primjer ispisao bi sadržaj naše kolekcije.

```
for (int i = 0; i < Razred.Count; i++)
{
    Console.WriteLine(Razred[i]);
}
```

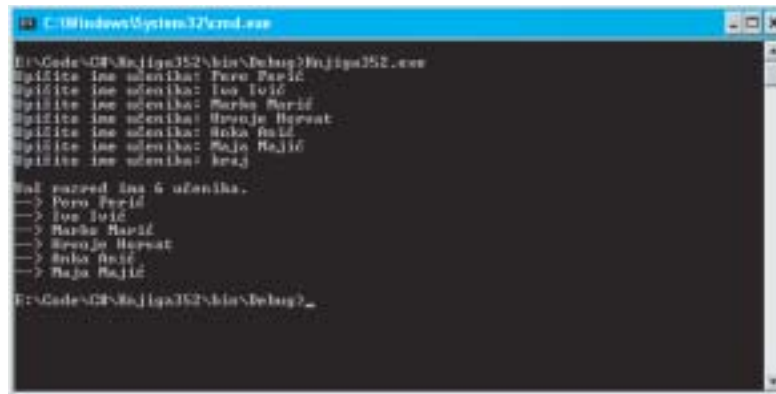
Pri korištenju kolekcija je najvažnije da uočite da kod njih ne trebamo znati koliko elemenata sadržavaju. Novi elementi dodaju se pomoću metode *Add*, a na isti način se mogu i micati. Želite li maknuti element s indeksom 0, napisat ćete:

```
Razred.RemoveAt(0);
```

Imajte na umu da će se cijela kolekcija tako smanjiti za jedan element: onaj koji je dosad imao indeks 1 sad će imati indeks 0, onaj s indeksom 2 dobit će indeks 1 i tako redom, jer je nulti izbrisan.

## II. DIO: OSNOVE PROGRAMIRANJA

**Slika 5-6:**  
**Primjer izvršavanja**  
**našeg programa koji**  
**radi s kolekcijom**  
**ArrayList**



### Za sve članove...

**D**a biste se kretali kroz sve članove kolekcije ili polja, umjesto dosadašnje obične *for* petlje koja ide od člana s nultim indeksom do onog s indeksom za jedan manje od ukupnog broja članova, na raspolaganju vam stoji još jedan oblik *for*-petlje. Radi se o petlji *foreach* koja, kao što joj i ime kaže, prolazi kroz sve članove polja ili kolekcije. Pritom vi ne trebate eksplicitno zadati kroz koje sve članove ona treba proći, jer će proći kroz sve njih, ma koliko ih bilo.

```
int[] mojePolje = new int[] {1,
2, 3};
foreach (int clan in mojePolje)
{
    Console.WriteLine(clan);
}
```

Dakle, petlja *foreach* u varijablu *clan* učitava sve elemente polja (isto vrijedi i za kolekciju). Vrlo je važno da je varijabla u koju se učitavaju

elementi istog tipa kao i svi članovi kolekcije kroz koju se prolazi. Ukoliko postoji mogućnost da se u kolekciji nalaze podaci različitih tipova, iskoristite dobri stari *boxing* – pretvorite sve članove u tip *object* i kasnije s njima radite što god želite. (Nemojte pretjerivati s korištenjem *boxinga*, jer to nije baš brza operacija.)

```
foreach (object clan in
mojaKolekcija)
{
    // ...
}
```

I na kraju, imajte na umu da petlja *foreach* ima nekoliko ograničenja. Prvo, unutar petlje *foreach* ne možete mijenjati vrijednost članova kolekcija ili polja, već ih možete samo čitati. Drugo, petlja *foreach* se ipak malo sporije izvršava od obične petlje *for*, zbog drugačijeg načina rada.



# Klase i strukture

Postepeno se krećemo prema sve složenijim oblicima podataka – evo nas na klasama i strukturama! Razumijevanje objektno orijentiranog programiranja (koje se temelji na klasama) bit će objašnjeno u narednom poglavlju i nužno je za razumijevanje rada .NET-a, jer, kao što već znate, funkcionalnost .NET-a organizirana je u hijerarhiju klasa.

U nastavku ćemo objasniti kako možete stvarati svoje klase i čemu one uopće služe, te kako se koriste strukture (tip podataka veoma sličan klasama).

## Klase

Na klase se može gledati kao na predloške za objekte. One opisuju funkcionalnost koje će pojedini objekti imati i podatke koje će sadržavati, no neće se raditi o konkretnom objektu. Primjer koji ćete susretati i u narednom poglavlju je *automobil* – klasa *Auto* može sadržavati opis objekta, njegove metode (primjerice, *Ubrzaj* i *Uspori*) te svojstva (*MaksimalnaBrzina*, *KolicinaGoriva* itd.).

No želite li stvoriti konkretan objekt koji će imati svoju vrijednost za maksimalnu brzinu i količinu goriva te koji će doista i obavljati akcije ubrzavanja i usporavanja, morat ćete stvoriti novi objekt iz klase *Auto*.

Evo kako biste definirali klasu *Auto*:

```
public class Auto
{
    // implementacija klase
}
```

Za definiranje klase koristi se ključna riječ *class* i ime klase, a unutar vitičastih zagrada se navodi njena implementacija. U implementaciji klase definiraju se članovi klase – radi se o svim metodama i svojstvima koje klasa ima.

**Ključna riječ *public* trenutno nije bitna za razumijevanje klasa, a bit će objašnjena u narednom poglavlju.**



Sve unutar vitičastih zagrada smatra se definicijom klase i tamo možete definirati njene članove. Na klase možete gledati kao na zasebne objekte koji u potpunosti sadržavaju svu svoju funkcionalnost i neovisni su o okolini. Tako svaka klasa može imati definirane metode i druge članove koji su nužni za njeno funkcioniranje. Definiranje članova unutar klasa ni po čemu se ne razlikuje od

## II. DIO: OSNOVE PROGRAMIRANJA

standardnog definiranja članova u programu iz jednostavnog razloga – i na vaš program se gleda kao na zasebnu klasu.

Stoga ćemo definirati nekoliko osnovnih metoda i svojstava klase *Auto*:

```
public class Auto
{
    public int TrenutnaBrzina;

    public void Ubrzaj()
    {
        TrenutnaBrzina += 10;
    }

    public void Uspori()
    {
        TrenutnaBrzina -= 10;
    }

    public void Kreni()
    {
        while (TrenutnaBrzina != 50) Ubrzaj();
    }

    public void Stani()
    {
        while (TrenutnaBrzina != 0) Uspori();
    }
}
```

Naša klasa će imati jedno cjelobrojno svojstvo, koje će služiti za praćenje trenutne brzine. Definiramo ga kao i svaku drugu varijablu – jednostavno navodimo njen tip i ime.

U klasi su definirane i 4 metode – za ubrzavanje, usporavanje, pokretanje i zaustavljanje automobila. One interno koriste varijablu *TrenutnaBrzina* i međusobno se pozivaju da bi ispunile svoju funkcionalnost. Tako će metoda za ubrzavanje povećati brzinu za 10 km/h, dok će je metoda za usporavanje smanjiti za 10 km/h. Metoda za pokretanje automobila će pozivati metodu za ubrzavanje sve dok brzina automobila ne dosegne 50 km/h. Slično će i metoda za usporavanje automobila usporavati sve dok brzina ne dođe na nulu.

## Stvaranje objekata iz klasa

Klase same za sebe ne vrijede mnogo. Kao što je rečeno, one služe kao predlošci za stvaranje objekata. Koristit ćemo ih poput bilo kojeg drugog tipa podataka, a za deklariranje ćemo koristiti samo ime klase.

## 5. POGLAVLJE: TIPOVI PODATAKA

```
Auto mojAuto;
```

Prethodnom naredbom se deklarira novi objekt klase *Auto*, no on se još ne stvara u memoriji. Još uvijek ga nije moguće koristiti, jer smo zasad tek *najavili* njegovo postojanje. Želite li ga stvoriti, iskoristit ćete naredbu *new*:

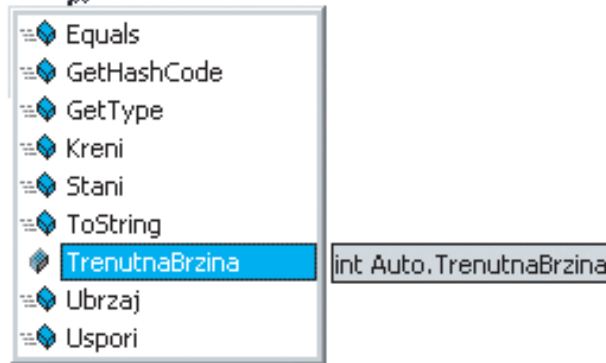
```
mojAuto = new Auto();
```

Tek sada je stvoren novi objekt klase *Auto*, koji sadrži svu njenu funkcionalnost (metode za ubrzavanje, usporavanje itd.). Naravno, kao i kod običnih varijabli, sve ste to mogli izvesti u jednoj naredbi:

```
Auto mojAuto = new Auto();
```

Sad možete pozivati sve naredbe i koristiti vrijednosti svih varijabli iz objekta. Na slici 5-7 prikazana je IntelliSense pomoć Visual Studia koja za objekt klase *Auto* ispisuje sve mu dostupne metode i članove.

```
Auto mojAuto = new Auto();
mojAuto.Kreni();
mojAuto.Ubrzaj();
mojAuto.Stani();
mojAuto. |
```



**Slika 5-7:**  
*IntelliSense će ispisati sve dostupne članove objekta.*

Da biste koristili članove nekog objekta, navedite ih odvojene točkom od imena objekta.



## II. DIO: OSNOVE PROGRAMIRANJA

Sve metode možete koristiti u svom programu. Slijedi jednostavan primjer u kojem simuliramo vožnju:

```
Auto mojAuto = new Auto();  
mojAuto.Kreni();  
mojAuto.Ubrzaj();  
Console.WriteLine(mojAuto.TrenutnaBrzina); // 60  
mojAuto.Stani();
```

Dakle, prvo smo stvorili novi objekt klase *Auto*, a zatim smo pozivali njegove metode. Sve te metode su definirane u samoj klasi, pa ih može koristiti svaki objekt te klase. Tako pokrećemo automobil, ubrzavamo ga, ispisujemo trenutnu brzinu (u tom trenutku će biti 60 km/h) i na kraju ga zaustavljamo.

## Konstruktori i destruktori

Klase imaju i još jednu veoma korisnu mogućnost – *konstrukture* i *destrukture*. Radi se o metodama koje se pozivaju pri stvaranju objekta iz neke klase (tome služi *konstruktor*) i pri uništavanju odnosno micanju objekta iz memorije (*destruktor*).

Da bismo uočili kako rade i u kojem se trenutku pozivaju konstruktori i destruktori, proširit ćemo prethodni primjer. Pri stvaranju objekta klase *Auto* želimo postaviti trenutnu brzinu na 5 km/h – recimo da želimo da novi automobili automatski voze malom brzinom. Tome će nam poslužiti konstruktor metoda.

Zbog jednostavnosti ćemo pri uništavanju objekta odnosno u destruktor metodi samo ispisati poruku – u složenijim primjerima destruktor će služiti za brisanje referenci, micanje nekih dodatnih objekata i slične operacije.

```
public class Auto  
{  
    public Auto()  
    {  
        Console.WriteLine("Stvaram objekt klase Auto.");  
        TrenutnaBrzina = 5;  
    }  
  
    ~Auto()  
    {  
        Console.WriteLine("Brišem objekt klase Auto.");  
    }  
  
    // ostatak implementacije...
```

## 5. POGLAVLJE: TIPOVI PODATAKA

```
}
```

Konstruktor i destruktor imaju isto ime kao i klasa u kojoj se nalaze. Dok konstruktor ima ispred sebe ključnu riječ *public* i nema oznaku tipa (jer ne vraća nijednu vrijednost, niti je moguće metodu konstruktora pozivati iz kôda – automatski se poziva pri stvaranju objekta), destruktor nema ni ključnu riječ *public* ni oznaku tipa, već samo znak “~” ispred imena.

U konstruktor tako možete ubaciti bilo koji kôd koji želite da se izvrši pri stvaranju objekta, a u destruktor sve što želite izvršiti po uništavanu samog objekta. Evo i detaljnije objašnjenog kôda primjera sa stvaranjem i korištenjem objekta:

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

    // Deklaracija objekta klase Auto
    Auto mojAuto;

    // Stvaranje novog objekta – pozivanje konstruktora!
    mojAuto = new Auto();

    // Korištenje metoda objekta
    mojAuto.Kreni();
    mojAuto.Stani();

} // Uništavanje objekta – pozivanje destruktora!
```

**Zbog Garbage Collectora – sustava u .NET-u koji prati korištenje objekata i sam automatski uništava nekoristene objekte – vrlo je teško odrediti točan trenutak kad se neki objekt uništava. U jednostavnim aplikacijama u konzoli to se najčešće dešava na kraju izvršavanja programa, no u složenijim aplikacijama je to vrlo teško odrediti, pa se stoga ne preporučuje oslanjanje na destruktor za izvršavanje kritičnih operacija.**



Ispis takvog programa bit će doista jednostavan te samo potvrđuje komentare iz prethodnog primjera i dokazuje da su se konstruktor i destruktor doista izvršili.

```
Stvaram objekt klase Auto.
Brišem objekt klase Auto.
```

## II. DIO: OSNOVE PROGRAMIRANJA

No konstruktorima se mogu prosljeđivati i parametri. Možda nećete biti zadovoljni postavljanjem početne brzine na 5 km/h, već ćete to htjeti odrediti za svaki objekt posebno. Izrada konstruktora koji prima parametar jednostavan je i svodi se na izradu metode koja prima parametar:

```
public class Auto
{
    public Auto(int PocetnaBrzina)
    {
        TrenutnaBrzina = PocetnaBrzina;
    }

    // ostatak implementacije...
}
```

Kao što vidite, konstruktor očekuje cjelobrojni parametar, koji se upisuje u varijablu *TrenutnaBrzina*. Sad je i stvaranje objekta klase *Auto* malo drugačije, no pruža veće mogućnosti:

```
Auto mojSporiAuto = new Auto(5);
Auto mojBrziAuto = new Auto(100);

// ispisuje 5
Console.WriteLine(mojSporiAuto.TrenutnaBrzina);

// ispisuje 100
Console.WriteLine(mojBrziAuto.TrenutnaBrzina);
```

Unutar zagrada pri stvaranju objekta navode se parametri za konstruktor. Uočite da u prethodnim primjerima unutar tih zagrada nije postojalo ništa, jer konstruktor nije ni primao parametre. Kako smo napisali konstruktor koji prima jedan parametar, nužno ga je navesti pri stvaranju objekta.

Vrlo se lako može provjeriti da li konstruktor doista radi – stvaramo dva objekta kojima su prosljeđene različite vrijednosti. Tako će objekt *mojSporiAuto* za trenutnu brzinu imati vrijednost 5 km/h, a *mojBrziAuto* 100 km/h, jer su to vrijednosti prosljeđene konstruktorima pri stvaranju.



Kad svladate osnove objektno orijentiranog programiranja (naredno poglavlje), naučit ćete napraviti tzv. preopterećene metode. Tada ćete moći napraviti objekte koji mogu imati dva konstruktora – jedan koji ne prima niti jedan parametar (i radi kao u prvim primjerima), i drugi koji prima parametar početne brzine (i radi kao u prethodnom primjeru). Takve objekte ćete zato moći stvarati na dva načina: s parametrom i bez njega.

## Strukture

Funkcionalnost struktura veoma je slična funkcionalnosti klasa, jer se i na strukture može gledati kao na zasebne elemente koji u sebi sadržavaju svu funkcionalnost. No razlika ipak postoji. Dok su klase zapravo reference (stvaranjem s *new* se tek zauzima mjesto u memoriji, a sama varijabla objekta samo ukazuje na tu memorijsku lokaciju), strukture su vrijednosni tip podataka. Usto, strukture, za razliku od klasa, nije moguće nasljeđivati.

Dakle, strukture mogu implementirati sve članove kao i klase, a za definiranje strukture se koristi ključna riječ *struct*.

```
public struct Tocka
{
    public int x, y;

    public void PomakniTocku(int PomakX, int PomakY)
    {
        x += PomakX;
        y += PomakY;
    }
}
```

Kao što vidite, načinjena je struktura koja predstavlja neku točku – koristi dvije cjelobrojne vrijednosti za pohranjivanje koordinata *x* i *y* te jednu metodu za pomak točke.

```
// Deklaracija strukture
Tocka A = new Tocka();

// Postavljanje varijabli
A.x = 50;
A.y = 100;

// Pozivanje metoda
A.PomakniTocku(-10, -50);
```

I strukture mogu imati konstruktore koji se definiraju isto kao i kod klasa (moraju biti istog imena kao i struktura) – u tom slučaju nove strukture stvarate baš kao i klase, a konstruktoru možete proslijediti i parametre.

```
Tocka A = new Tocka(3, 4);
```

**Pripazite:** strukture ne mogu imati definirane destruktore.



## II. DIO: OSNOVE PROGRAMIRANJA

### Strukture ili klase?

Na prvi pogled su strukture i klase identične – obje mogu sadržavati članove, obje mogu imati konstruktore i, kao i svi drugi tipovi podataka u .NET-u, izvedene su iz glavnog objekta *Object*. No razlika postoji: dok su klase reference, strukture su vrijednosni tip podataka i njihovi se podaci spremaju na stogu, u glavnom dijelu memorije.

Pristup stogu, gdje se spremaju strukture, veoma je brz i lagan, no ukoliko na njega spremite velike količine podataka, to će bitno utjecati na performanse aplikacije. To znači da su strukture idealne za male objekte koji sadržavaju male količine podataka. Klase su, s druge strane, prikladnije za veće objekte, za koje se očekuje da će duže postojati u memoriji (pa ih se stoga ne želi smještati na stog) i koji sadržavaju veće količine podataka.

Želite li tako stvoriti polje objekata, za primjer uzmimo polje od 10 točaka koje zajedno čine pravac. Pritom su definirane klasa i struktura za točku:

```
public struct TockaStruktura
{
    // implementacija kao u prethodnom primjeru
}

public class TockaKlasa
{
    // implementacija identična strukturi
}
```

Sljedeći program bi tako stvorio i inicijalizirao dva polja od 10 točaka, jedno koje koristi strukture za zapis pojedinih točaka, i drugo koje koristi klase. Primijetite da je stvaranje polja struktura ili klasa po sintaksi identično stvaranju polja s vrijednostima tipa *int* ili *string*.

```
TockaStruktura[] PravacStruktura = new TockaStruktura[10];
TockaKlasa[] PravacKlasa = new TockaKlasa[10];
```

Na ovom se primjeru može uočiti bitna razlika u načinu rada struktura i klasa. Ukoliko koristite polje klasa, u memoriji će se stvoriti 11 objekata – jedan objekt za polje i po jedan objekt za svaki od 10 članova polja. To je tako zato što će članovi polja biti zapravo reference na same objekte spremljene negdje drugdje u memoriji.

S druge strane, korištenjem polja struktura, u memoriji će se stvoriti samo jedan objekt, i to onaj za polje. Sve strukture točaka bit će spremljene unutar polja, jer se radi o vrijednosnom tipu podataka.

No takav način rada struktura nije nužno bolji. Već je prije spomenuto da korištenjem struktura možete usporiti aplikaciju – ukoliko svojim metodama prosljeđujete strukturu kao parametar, ona



## 5. POGLAVLJE: TIPOVI PODATAKA

će morati biti kopirana i metoda će koristiti novu instancu strukture. To je posljedica vrijednosnog tipa podatka – metodi se ne može proslijediti referenca na već postojeću instancu u memoriji, već se treba raditi nova kopija samih vrijednosti podataka. Time se, dakako, zauzima dodatna memorija.

Prije no što se odlučite na korištenje struktura ili klasa, dobro razmislite za što ćete takve objekte koristiti. Treba imati na umu i da strukture nije moguće nasljeđivati, pa one nisu izbor kad želite napraviti pravu objektno orijentiranu aplikaciju i razbiti svu funkcionalnost aplikacije na hijerarhiju objekata. (Više o tome u narednom poglavlju!) Pravilnim izborom i korištenjem možete ubrzati rad svoje aplikacije, a to je nešto oko čega se vrijedi potruditi.

# Delegati

U .NET-u postoji još jedan način pozivanja metoda. Radi se o *delegatima*, koji zapravo predstavljaju pokazivače na metodu. Pri deklaraciji delegata definirate tip metode koju može pozivati (tip podataka koju vraća) i parametre (sve to zajedno čini *potpis* metode).

**Važno je shvatiti osnovni koncept delegata. Na njih se može gledati kao na poseban tip podatka, koji u sebi sadržava poziv na neku metodu. Tako jedan delegat može biti iskorišten za pozivanje više različitih metoda s istim potpisom (isti povratni tip, parametri i prava pristupa).**



Dakle, nužno je definirati tip metoda koji se može pozivati korištenjem delegata. Sljedeći primjer omogućava pozivanje metoda koje vraćaju *double* vrijednost, a ujedno i primaju *double* vrijednost.

```
public delegate double mojDelegat (double x);
```

**Delegati predstavljaju dio klase i stoga moraju biti definirani izvan bilo koje metode.**



Da bismo mogli iskoristiti definiranog delegata, napraviti ćemo dvije metode s istim potpisom. One će biti krajnje jednostavne i služiti će isključivo za pokazivanje primjera.

```
public double kvadrat(double x) { return Math.Pow(x, 2); }
public double korijen(double x) { return Math.Sqrt(x); }
```

## II. DIO: OSNOVE PROGRAMIRANJA

Krenimo sad s uporabom delegata. Da bismo mogli pozvati metodu putem delegata, stvorit ćemo njegovu novu instancu, koja će sadržavati pokazivač na željenu metodu. Evo kako bi izgledali delegati za poziv prethodne dvije metode:

```
mojDelegat dKvadrat = new mojDelegat(kvadrat);  
mojDelegat dKorijen = new mojDelegat(korijen);
```

Dakle, definirali smo dvije varijable tipa *mojDelegat*. Njihove smo instance stvorili korištenjem operatora *new*, a unutar zagrada smo naveli metodu koju će svaki od delegata pozivati. Kako je delegat definiran tako da prima i vraća *double* vrijednost, na nove instance *dKvadrat* i *dKorijen* može se gledati kao na takve metode.

Korištenje delegata očito je na sljedećim primjerima. Iskoristit ćemo *dKvadrat* i *dKorijen* za poziv metoda.

```
double a = dKvadrat(5); // 25  
double b = dKorijen(49); // 7
```

Kao što vidite, korištenje delegata identično je pozivima običnih metoda. Tako u dvije varijable *double* tipa *a* i *b* spremamo kvadrat ili korijen brojeva. Imajte na umu da delegate možete koristiti za poziv bilo kakvih metoda, svedeno da li vraćaju neke podatke ili ne.



Iako je dosad pokazano da su delegati jednostavni za korištenje, možda ste si ipak postavili pitanje zašto ih uopće koristiti kad je jednostavnije direktno pozivati metode. Ne brinite, na vaše pitanje bit će odgovoreno kad se pozabavimo događajima u aplikacijama i asinkronim pozivima web-servisa. Zasad je važno da shvatite njihov koncept i način korištenja.