C#

PRO-MIL d.o.o.
Varaždin, travanj 2007.

Naslov knjige:	C#
Autor:	Slaven Sarić, dipl. ing. el.
Naklada:	PRO-MIL d.o.o. za nakladu, informatiku i edukaciju, Varaždin, R. Boškovića 20, 42000 Varaždin, tel: 042 / 203 981, 233 971, fax: 042 / 203 991, www.pro-mil.hr
Urednik:	Nenad Milijaš, dipl. inf.
Lektura:	Ružica Gregurić, dipl. uč.
Korektura:	PRO-MIL d.o.o.
Naslovnica:	Nenad Milijaš, dipl. inf.
Priprema za tisak:	Radek Perši
Tisak:	Tiskara Varteks, Varaždin
ISBN:	978-953-7156-22-0
Copyright:	© PRO-MIL d.o.o. za nakladu, informatiku i edukaciju, Varaždin

Sadržaj

Poglavlje 15: C#++

15.1. XML
15.2. ADO.Net
15.3. Web servisi

Dodaci:

Dodatak A: C# 2.0
Dodatak B: Ključne riječi
Dodatak C: Imenik (Namespaces)

Uvod u C#

C# (čita se kao C Sharp) je moderan, objektno orijentiran jezik namijenjen širokoj primjeni. C# je tipski siguran, objektno orijentiran s ugrađenim automatskim čišćenjem memorije (garbage collection). Također, C# ima ugrađenu obradu iznimaka što ga po pitanju sigurnosti čini vrlo dobrim odabirom naročito uzmemo li u obzir da je pisanje aplikacija za internet jedan od primarnih ciljeva korištenja ovog jezika.

C# osuđen je na uspjeh iz barem tri razloga.

- sintaksa naslijeđena iz C generacije jezika
- moćan i upotrebljiv .Net Framework Class Library (FCL) s velikim brojem klasa i objekata
- još moćniji Microsoft iza sebe, koji C# predstavlja kao glavni programski jezik za sve svoje buduće projekte

Ono prvo znači da će privući veliki broj programera koji poznati kao konzervativni ne vole mijenjati navike odnosno sintaksu, dok ovo drugo znači da se s vrlo malo naredbi može napraviti jako puno.

C# potpuno je nov jezik što znači da je mogao učiti iz grešaka svojih prethodnika, ali i uzeti ono najbolje od svakog od njih. Tu naravno mislimo na C, C++ i Javu.
Neki (mnogi) zlobnici reći će da je C# ipak kopija Jave, osobito ako se pogleda struktura programa. Bilo to točno ili ne, C# je tu, spreman za izradu programa, a tu je i onaj gore navedeni treći razlog.

.Net u kombinaciji sa C# nudi još jednu do sad nepostignutu osobinu, a to je da se na sličan način pišu aplikacije za potpuno različite platforme, kao npr. za mobilne uređaje, desktop računala ili za web servere.

C# dobio je ime koje bi značilo *povišeni C* kao analogija na tonske povisilice čime se htjelo ukazati na napredne mogućnosti u odnosu na C, a i na C++.
Znak '#' tako predstavlja znak za tonsku povisilicu, a ne znak 'Hash'. Neke interpretacije imena kažu da je to skriveni C++++ s tim da su 4 znaka + napisana u dva reda.
Kao što je C++ gledajući sintaktički povećanje C-a za 1, tako bi i C sa 4 plusa trebao biti povećanje C++ za 1.

Poglavlje 1:
Prvi C# program

Ovo poglavlje obrađuje:

- C# prevoditelj i način izvođenja programa
- strukturu programa u programskom jeziku C#
- korištenje argumenata naredbene linije

1.1. Prvi program

Najbolji način učenja programskog jezika je pisanje programa, a Dennis Ritchie i Brian Kernighan u svojoj knjizi *The C Programming Language* kažu da je to jedini način. Ne valja mijenjati dobre navike, pa pogledajmo kao i u toj knjizi kako bi izgledao program koji bi ispisao na zaslonu riječi *Svaki početak je težak.*

```
public class Test
{
      public static void Main()
      {
            System.Console.WriteLine("Svaki početak je težak");
      }
}
```

Nakon prevođenja programa u strojni oblik naredbom
csc Test.cs (ektenzija za datoteke sa C# programima je *cs*)
dobit ćemo izvršnu verziju programa čijim pokretanjem se ispisuju riječi *Svaki početak je težak.*

Naizgled ne baš velika stvar, međutim ovo ipak daje sigurnost u daljnjem proučavanju jer ako se program uspješno preveo, znači da je sve što se tiče programa prevoditelja (*compiler*) postavljeno kako treba. Ovdje je riječ o prevoditelju koji ne stvara klasični oblik izvršne datoteke, nego o prevoditelju koji naredbe programskog jezika pretvara u tzv. *Microsoft intermediate language (MSIL)* kôd koji je neovisan o procesoru. Prije pokretanja *MSIL* kôd mora biti preveden u izvršni kôd ovisan o procesoru, a to se postiže korištenjem tzv. *JIT (Just in time)* prevoditelja. Ovo *Just in time* znači da se *MSIL* kôd ne prevodi u cijelosti u izvršni kôd, nego se prevodi po pozivu, odnosno radi se neka vrsta keširanja. Npr. ako se u programu poziva neka funkcija, prije izvođenja će se kôd funkcije prevesti u izvršni oblik te će se taj izvršni oblik sačuvati za slučaj ponovnog poziva.
Izvršni kôd izvršava se unutar tzv. CLR (Common Language Runtime), a to je zapravo "izvršni stroj" (execute machine) i zato se takav izvršni kôd zove i upravljani (managed) kôd.

Na ovaj način moguće je isti izvorni (source) kôd prevesti u različiti izvršni kôd, ovisno o procesoru. To opet omogućuje ostvarenje onog Javinog sna "write once, run everywhere", odnosno da se isti programski kôd izvršava na različitim platformama. (To da se izvršava na različitim platformama lakši je dio posla. Izvršavanje s istim rezultatima izvođenja programa je teži dio. Tko god je pisao Java aplikacije za mobitele i ručna računala, složit će se da je to tako).

Program prevoditelj za C# dobije se instalacijom .NET Framework Software Development Kita (SDK) i besplatno se može skinuti s adrese *http://msdn.microsoft.com/downloads*.
U paketu je i .Net Runtime potreban za izvršavanje programa. Alat kao što je Visual Studio .Net nije besplatan, ali i nudi puno više od samog prevođenja u strojni oblik.
Knjiga obrađuje osnovnu verziju C# s *major* verzijom 1, dok ćemo u dodatku navesti najvažnije novosti koje donosi *major* verzija 2. Naravno, vertikalna kompatibilnost postoji, tako da ako primjere budete prevodili s *major* verzijom 2, ne bi smjelo biti razlika.

Svaki C# program mora imati *Main* funkciju koja označava početak izvršenja programa. U našem primjeru sadržaj *Main* funkcije je samo jedna naredba i to poziv funkcije *WriteLine* iz FCL-a. Kao što vidimo, potrebno je uključiti cijelu putanju do te funkcije o čemu će više

riječi biti kasnije. Za sada samo navedimo način kako se sintaksa pozivanja funkcije iz .Net Frameworka može skratiti:

```
using System;

public class Test
{
	public static void Main()
	{
		Console.WriteLine("Težak početak");
	}
}
```

Primjer 1.1

Ispis:
Težak početak

Sad je na početku programa uključen set klasa *System* tako da više nije potrebno navoditi cijelu putanju funkcije.
U C# programu sve je dio klase i nema globalnih varijabli, deklaracija i definicija funkcija izvan klase. Sve je dio neke klase i pristupanje članovima klase moguće je jedino preko instanci te klase odnosno objekata tipa te klase. Iznimka su statički članovi klasa koji pripadaju klasi, a ne objektu pa se tako i pozivaju.

1.2. Argumenti naredbene linije

Pogledajmo sada ovaj primjer:

```
using System;

public class Test
{
	public static void Main(string[] Args)
	{
		for(int i=0; i<Args.Length; i++)
		{
			Console.WriteLine(Args[i]);
		}
	}
}
```

Primjer 1.2

Ako program pokrenemo kao *Test Težak početak*, ispis će biti
Težak
početak

Ovako možemo, kombinirajući s izlaznim parametrima, dobiti različite rezultate izvršenja programa. I to sve bez ponovnog prevođenja programa. Primijetimo da unutar *for* petlje ne navodimo broj argumenata. Za to je zadužena funkcija *Length* klase *String*.

Ako i ne vidite neku veliku upotrebljivost navođenja argumenata u naredbenoj liniji, pogledajmo sljedeći primjer koji će izračunati ukupnu vrijednost otpora za dva paralelno spojena otpornika.

```
using System;
```

```
public class Test
{
	public static void Main(string[] Args)
	{
		if(Args.Length != 2)
			return;
		int R1 = System.Convert.ToInt32(Args[0]);
		int R2 = System.Convert.ToInt32(Args[1]);
		Console.WriteLine("{0} Ohm", R1 * R2 / (R1 + R2));
	}
}
```

Primjer 1.3

Ako program pokrenemo kao *Test 6 12*, ispis će biti
4 ohm

Jednostavnim navođenjem dviju vrijednosti otpornika program će izračunati ukupnu vrijednost otpora. Ako trebate izračunati mnogo različitih vrijednosti, ovo je vjerojatno najjednostavniji i najbrži način.

Argumenti naredbene linije su stringovi pa ih je prije izračuna potrebno pretvoriti u cjelobrojni tip i za to se koristi funkcija *ToInt32* klase *Convert*. Poziv ugrađenih funkcija za ovakve stvari nije ništa novo, C je za ovo imao standardnu funkciju *atoi* koja je radila isto. Novo je što je FCL puno bogatiji i opremljeniji od svih dosadašnjih biblioteka funkcija, klasa i objekata.

Funkcija *WriteLine*, kao i mnogo drugih funkcija, može se pozvati na nekoliko različitih načina. U ovom primjeru kao argument funkcije navodi se formatirani string koji u prvom dijelu ima parametar u vitičastim zagradama {0} što znači da se u drugom dijelu ovako formatiranog stringa očekuje vrijednost koja će biti ispisana. Broj parametara je proizvoljan s tim da se poštuje numeracija pa tako drugi parametar ima oznaku 1 itd. Tako se isti ispis mogao dobiti i ovako:

```
Console.WriteLine("{0} {1}", R1 * R2 / (R1 + R2), "Ohm");
```

ili ovako:

```
string otpor = "Ohm";
Console.WriteLine("{0} {1}", R1 * R2 / (R1 + R2), otpor);
```

Vidimo da na desnoj strani formatiranog stringa može biti izraz, statički tekst ili varijabla. Većinu funkcija iz FCL-a moguće je pozvati s različitim brojem i vrstom argumenata što je jedna od glavnih osobina objektno orijentiranog programiranja.

Logično, gdje je *WriteLine*, tu bi morao biti i *ReadLine* koji bi trebao učitati liniju s ulaza.

```
using System;

public class Test
{
	public static void Main()
	{
		Console.WriteLine("Unijeli ste: " + Console.ReadLine());
	}
}
```

Primjer 1.4

Ako nakon pokretanja programa upišemo
Težak početak
ispis će biti:
Unijeli ste: Težak početak

Console.ReadLine učitava liniju s ulaza i tu liniju vraća kao povratnu vrijednost tipa *string*.
Ovaj primjer pokazuje još dvije stvari.
Kao i kod C i C++ i ovdje možemo "pakirati" naredbe jednu unutar druge. Na taj način možda se gubi na čitljivosti, ali se zato dobija na performansama.

I drugo, funkcija *WriteLine* može kao argument primiti zbroj dvaju ili više stringova. To daje lakše razumijevanje onoga što će biti ispisano.
Usporedite:

```
Console.WriteLine("Unijeli ste: " + Console.ReadLine());
```

s

```
Console.Write("Unijeli ste: ");
Console.WriteLine(Console.ReadLine());
```

Očito je da je prvi način jednostavniji i razumljiviji, a i potreban je samo jedan poziv funkcije.
Funkcija *Write* razlikuje se od *WriteLine* jedino po tom što ne dodaje novi red na kraju.
Kao i kod C i C++ kraj naredbe označava se s ';'. Ovo se ne odnosi na naredbe uvjetnog izvođenja programa (`if-else`) i naredbe petlji (`for i while`) iza kojih ne dolazi ';'.

Poglavlje 2:
Varijable i tipovi podataka

Ovo poglavlje obrađuje:

- identifikatore i pravila za njihovo korištenje
- ključne riječi jezika
- pojam varijable
- tipove varijabli
- deklaraciju i definiciju varijable
- razliku između vrijednosnih i referentnih tipova
- konverziju između različitih tipova
- tip *string* i njegove metode
- tip *object*, boksiranje i odboksiranje
- tip *enum*

2.1. Identifikatori

Identifikatori su imena koja se daju varijablama, metodama, klasama itd. Kod davanja imena treba se pridržavati nekih, ne baš krutih, ali ipak pravila.

- Identifikator mora biti cijela riječ.
- Mora početi slovom ili donjom crtom (‘_’).
- Ne smije biti ključna riječ jezika.

Ključne riječi zaštićene su od strane programskog jezika i ako se baš želi koristiti kao identifikator, onda se ispred identifikatora kao prefix doda znak ‘@’. Taj znak nema ulogu u identificiranju identifikatora od strane prevoditelja tako da su identifikatori *@Counter* i *Counter* jednaki što se tiče prevoditelja i neće biti dozvoljeno deklarirati dvije varijable unutar istog bloka s imenima *@Counter* i *Counter*, ali će biti dozvoljeno deklarirati varijablu s imenom *@abstract* iako je *abstract* ključna riječ.

Primjeri dozvoljenih identifikatora:
counter5
Counter
_BrojDana
@double
Double (C# je case sensitive što znači da se razlikuju velika i mala slova)

Primjeri nedozvoljenih identifikatora:
2counter
-BrojDana
Broj Dana
double

Pregled ključnih riječi:

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
while				

Kod davanja imena identifikatorima preporuka je da se koristi tzv. Pascal notacija po kojoj svaka pojedina riječ u identifikatoru (riječi ne smiju biti odvojene razmakom!) počinje velikim slovom kao npr.

Counter
BrojDana
JakostStruje
SnagaMotora
ProsjekDohotka
itd…

Također, preporuka kod davanja imena identifikatorima jest da to budu smislena, opisna imena tako da ime identifikatora kazuje što radi ili kakve podatke sadrži. Nikako nije dobro radi brzine davati imena s raznim kraticama jer prije ili poslije netko drugi će morati listati po programu. A i samom autoru programa nakon nekog vremena puno je lakše intervenirati u kôdu ako su imena identifikatora smislena.

2.2. Vrijednosni tipovi

Jedan od temeljnih pojmova u programiranju je pojam varijable. Varijabla je memorijska lokacija koja može poprimiti određenu vrijednost. Vrsta podatka koji je sadržan u varijabli ovisi o tipu varijable i određuje se prilikom deklaracije varijable. Pored toga kod deklaracije varijable potrebno je i odrediti naziv varijable gdje vrijede pravila navedena kao i kod svih identifikatora. Programeri koji nisu radili sa C ili C++ često ne razlikuju dovoljno deklaraciju i definiciju varijable. Naravno, stvar je kao i obično vrlo jednostavna.

Deklaracija varijable određuje dvoje:

- tip varijable koji određuje kakvu vrstu vrijednosti varijabla može poprimiti (cjelobrojni, datum, string, decimalni, neka klasa itd..)
- naziv varijable

Definicija određuje vrijednost varijable i ne mora se dodijeliti kod deklaracije (ali se mora dodijeliti prije uporabe varijable)

Primjeri deklaracija i definicija:

```
int x = 5;                  // Deklaracija i definicija varijable x
long x = 5, y = 6;          // Deklaracija i definicija varijabli x i y
double x, y = 5;            // Deklaracija varijable x i deklaracija i definicija varijable y
x = 10;                     // Definicija varijable x
```

U C# programskom jeziku tip varijable može biti **vrijednosni** (value type) ili **referentni** (reference type).
Osnovna razlika između ovih dvaju tipova proizlazi iz načina njihovog spremanja u memoriji. Vrijednost varijable vrijednosnog tipa sadrži ujedno i vrijednost samog podatka u varijabli, dok vrijednost varijable referentnog tipa sadrži adresu memorijske lokacije na kojoj se nalazi stvarni podatak.
Pogledajmo primjer:

```
using System;
```

```
class TempClass
{
      public int Count = 0;
}

class Test
{
      static void Main()
      {
            int ValTip1 = 10;
            int ValTip2 = ValTip1;
            ValTip2 = 100;
            TempClass RefTip1 = new TempClass();
            TempClass RefTip2 = RefTip1;
            RefTip2.Count = 100;

            Console.WriteLine("{0}, {1}", ValTip1, ValTip2);
            Console.WriteLine("{0}, {1}", RefTip1.Count, RefTip2.Count);
      }
}
```

Primjer 2.1

Ispis:
10, 100
100, 100

U ovom primjeru su varijable *ValTip1* i *ValTip2* vrijednosne varijable, dok su *RefTip1* i *RefTip2* referentne varijable.

ValTip1 je kod deklaracije definirana i dodijeljena joj je vrijednost 10. Varijabli *ValTip2* je kod deklaracije dodijeljena vrijednost varijable *ValTip2* s naredbom ValTip2 = ValTip1 i nakon toga prestaje svaka veza između varijabli *ValTip1* i *ValTip2*.
Budući da su *ValTip1* i *ValTip2* smještene na različitim memorijskim lokacijama, promjena vrijednosti varijable *ValTip2* s naredbom ValTip2 = 100 nema nikakvog utjecaja na vrijednost *ValTip1*.

Kod referentnih varijabli *RefTip1* i *RefTip2* je drugačije jer one sadržavaju memorijske lokacije na kojima se nalazi podatak koji varijabla predstavlja.
S naredbom RefTip2 = RefTip1 sadržaj varijable *RefTip2* će biti isti kao i u *RefTip1* a to je memorijska lokacija na kojoj se podatak nalazi. Kako obje varijable sadrže istu vrijednost koja upućuje na istu memorijsku lokaciju, rezultat ispisa bit će isti.

C# podržava nekoliko ugrađenih vrijednosnih tipova:

Tip	**bytes**	**Raspon**	**Struktura**
sbyte	1	-127 do 128	System.SByte
byte	1	0 do 255	System.Byte
short	2	-2^{15} do $2^{15}-1$	System.Int16
ushort	2	0 do $2^{16}-1$	System.UInt16
int	4	-2^{31} do $2^{31}-1$	System.Int32
uint	4	0 do $2^{31}-1$	System.UInt32
long	8	-2^{63} do $2^{63}-1$	System.Int64
ulong	8	0 do $2^{63}-1$	System.UInt64

char	2	0 do 2^{16}-1	System.Char
float	4	1.5×10^{-45} do 3.4×10^{38}	System.Single
double	6	5×10^{-324} do 1.7×10^{308}	System.Double
bool	1	true ili false	System.Boolean
decimal	16	10^{-28} do 7.9×10^{28}	System.Decimal

Svi ovi ugrađeni vrijednosni tipovi zapravo su alijasi za strukture koje su dio FCL-a. Strukture su složeni vrijednosni tipovi o kojima će kasnije biti više riječi. Potpuno je ravnopravno napisati:

```
double d = 10.6;
i
System.Double d = 10.6;
```

Zbog čitljivosti i kompatibilnosti sa starim navikama i programima preporuka je da se koriste alijasi za ugrađene vrijednosne tipove.
Kod odabira tipa za određenu varijablu treba obratiti pozornost na raspon mogućih vrijednosti koje varijabla može poprimiti. Pritom je bolje ostaviti malo rezerve kod očekivanih vrijednosti, ali ne treba pretjerivati. Npr. pravo rasipanje memorije je rezervirati tip *long* za varijablu u kojoj će biti spremljen broj godina određene osobe.

Tipovi varijabli mogu se podijeliti u nekoliko grupa:

- **cjelobrojni** koji mogu poprimiti samo cijeli broj (`sbyte, short, int, long`)
- **pozitivni cjelobrojni** koji mogu poprimiti samo pozitivni cijeli broj (`byte, ushort, uint, ulong`)
- **racionalni** (`float, double, decimal`)
- **tipovi za prikaz znakova (characters)** (`char`)
- **true/false tip** (`bool`)

Cjelobrojni tipovi
Kao što je navedeno, vrijednosti cjelobrojnih tipova mogu biti *cijele* (označeni tip ili *signed*) ili *cijele pozitivne* (neoznačeni tip ili *unsigned*). Kod cijelih, prvi bit je potrošen kao oznaka je li broj pozitivan ili negativan (1 - negativan, 0 – pozitivan), a kod cijelih pozitivnih taj bit je iskorišten kao i svi ostali bitovi za vrijednost varijable pa je maksimalna vrijednost koju varijabla neoznačenog tipa može poprimiti dvostruko veća od one kod pripadajućeg označenog tipa.

Zato ako vrijednost varijable sigurno ne može biti negativna, bolje je odabrati neki od neoznačenih tipova. Ne samo što imamo na raspolaganju dvostruko veću vrijednost, nego smo i zaštićeni od neželjenog dodjeljivanja negativne vrijednosti varijabli gdje to nema smisla jer C# ne dozvoljava da neoznačena varijabla sadrži negativnu vrijednost.

```
uint x = -100;    // Prevoditelj javlja grešku
```

Ova zaštita postoji samo za vrijeme prevođenja, odnosno ako postoji mogućnost da neoznačena varijabla poprimi negativnu vrijednost za vrijeme **izvođenja** programa, rezultati mogu biti vrlo nepredvidljivi.

Pogledajmo ovaj primjer:

```
using System;

class Test
{
      static void Main()
      {
            for(uint Counter=10;Counter >= 0;Counter--)
                  Console.WriteLine(Counter);
      }
}
```

Primjer 2.2

Ovaj program će se prevesti bez ikakvih grešaka i upozorenja, međutim ako ga pokrenemo, program će ući u beskonačnu petlju. Ako pogledamo *for* petlju, u zadnjem prolazu varijabla *Counter* će doseći vrijednost 0 i nakon toga će se smanjiti za 1. Budući da je riječ o neoznačenoj varijabli koja ne može biti negativna, to će smanjenje značiti pretvaranje broja u najveći mogući pozitivni broj za ovaj tip. Uvjet za daljnje izvršenje bio bi ispunjen i program bi umjesto završetka nastavio s radom.
Zamjenom tipa `uint` s `int` problem je riješen i program bi ispisao brojeve od 10 do 0 te prestao s izvršenjem.

Racionalni brojevi

Za racionalne brojeve imamo na raspolaganju tri tipa: *float*, *double* i *decimal*.
Koji od njih ćemo upotrijebiti, ovisi o maksimalnoj očekivanoj vrijednosti te o preciznosti. Tip *float* ima preciznost od 7 znamenaka, *double* 15 ili 16, dok *decimal* ima preciznost od 27 ili 28 znamenaka, ali i zauzima 16 byte-ova.
Podrazumijevani (*default*) tip za racionalne brojeve je *double*, tako da kad dodjeljujemo vrijednost treba na to obratiti pozornost ili dodati oznaku na kraju broja koja će prevoditelju kazati da se broj tretira kao određeni tip. Te oznake su f ili F za *float*, m ili M za *decimal* te d ili D za *double*.

```
float f = 24.7;          // Prevoditelj javlja grešku
float f = 24.7f;         // OK
decimal d = 24.7;        // Prevoditelj javlja grešku
decimal d = 24.7m;       // OK
```

Karakter tip

Tip *char* se koristi za dodjeljivanje znakova iz Unicode set karaktera. npr.

```
using System;

class Test
{
      static void Main()
      {
            char c1 = 'A';
            char c2 = (char)66; // ASCII KOD ZNAKA B JE 66
            char c3 = '\x0043'; // ASCII KOD ZNAKA C JE 43 HEKSADECIMALNO
            char c4 = '\u0044';
            char plus = '+';
            Console.WriteLine("{0} {1} {2} {3} {4} {5} {6}",
                              c1, plus, c2, plus, c3, plus, c4);
      }
}
```

Primjer 2.3

Ispis:
A + B + C + D

Tip bool
Tip *bool* može imati samo dvije vrijednosti *true* i *false* i kao takav koristi se u situacijama koje mogu imati samo dvije vrijednosti.

```
bool x = true;
bool y = false;
```

Svaka varijabla ima područje u kojem vrijedi. To je područje unutar bloka koji se u C# označava vitičastim zagradma '{' i '}'.

```
using System;

class Test
{
	static void Main()
	{
		int x = 10;
		for(int y=0;y<10;y++)
		{
			int z = 20;
		}
		// Console.WriteLine("y = " + y);	// Ovo se neće prevesti
		// Console.WriteLine("z = " + z);	// Ovo se neće prevesti
		Console.WriteLine("x = " + x);		// OK
	}
}
```

Primjer 2.4

Varijable *y* i *z* nisu vidljive izvan bloka *for* petlje i zato će prevoditelj javiti grešku. To nije slučaj s varijablom *x* jer je ona deklarirana izvan tog bloka.
Međutim, ovo je ispravno napisano:

```
using System;

class Test
{
	static void Main()
	{
		int x = 10;
		for(int i=0;i<10;i++)
		{
			x++;
			Console.Write(x + ",");
		}
	}
}
```

Primjer 2.5

Ispis:
11,12,13,14,15,16,17,18,19,20,

S obzirom da je varijabla *x* deklarirana na početku *Main* funkcije, ona je vidljiva u cijeloj *Main* funkciji pa tako i u bloku *for* petlje.

2.3. Konverzija između tipova

Često se u jednoj naredbi koriste dva ili više različitih tipova. Tada dolazi do konverzije jednog tipa u drugi. Postoje dvije vrste konverzije, implicitna i eksplicitna.
Kod implicitne konverzije nema gubitka vrijednosti niti preciznosti i događa se automatski.
Kod eksplicitne konverzije potrebno je napraviti tzv. *cast* dodavanjem u zagradama tip varijable u koji se pretvara ispred tipa iz kojeg se pretvara vrijednost npr. `i = (int)l`.
Kod eksplicitne konverzije može biti gubitka vrijednosti ili preciznosti, ali i ne mora. Ako tip varijable u koji se pretvara može sadržavati vrijednost i(li) preciznost tipa iz kojeg se pretvara, onda nema gubitka. U suprotnom postoji gubitak vrijednosti i(li) preciznosti.

```
int i = 50;
long l = i;
```
Implicitna konverzija, nema gubitka vrijednosti.

```
float f = 35.6f;
double d = f;
```
Implicitna konverzija, nema gubitka ni vrijednosti ni preciznosti

```
int i;
long l = 50;
i = (int)l;
```
Eksplicitna konverzija, nema gubitka vrijednosti.

```
int i;
long l = 5000000000;
i = (int)l;
```
Eksplicitna konverzija, ima gubitka vrijednosti, u varijabli *i* neće biti broj 5000000000.

```
float f;
double d = 35.6;
f = (float)d;
```
Eksplicitna konverzija, nema gubitka ni vrijednosti ni preciznosti.

```
float f;
double d = 35.123456789;
f = (float)d;
```
Eksplicitna konverzija, ima gubitka preciznosti jer *float* nema preciznost na 9 decimala.

Ako se na jednoj strani izraza nalaze različiti tipovi, tada dolazi do implicitnog pretvaranja.

```
float f = 0.6f;
double d = 35.123456789;
double d1 = d + f;
```
U *d1* je vrijednost 35.723456789 jer je došlo do automatske pretvorbe *float f* u *double*.

Preciznost tipa *float* od 7 znamenaka dovoljna je za veliku većinu potreba, međutim velika većina nije i sve. Pogledajmo sljedeći primjer:

```
using System;

class Test
{
      static void Main()
      {
            float f = (float)Math.Cos(0.00001);
```

```
            Console.WriteLine("Float: " + f);

            double d = Math.Cos(0.00001);
            Console.WriteLine("Double:  " + d);

            decimal m = (decimal)Math.Cos(0.00001);
            Console.WriteLine("Decimal: " + m);
        }
}
```

Primjer 2.6

Ispis:
Float: 1
Double: 0,99999999995
Decimal: 0,99999999995

Cosinus vrlo malog kuta je vrlo blizu iznosu 1, tako da tip *float* to ne može prikazati.

Implicitna pretvaranja odvijaju se po sljedećim pravilima:

```
Iz sbyte u short, int, long, float, double, decimal
Iz byte u short, ushort, int, uint, long, ulong, float, double, decimal
Iz short u int, long, float, double, decimal
Iz ushort u int, uint, long, ulong, float, double, decimal
Iz int u long, float, double, decimal
Iz uint u long, ulong, float, double, decimal
Iz long u float, double, decimal
Iz ulong u float, double, decimal
Iz char u ushort, int, uint, long, ulong, float, double, decimal
Iz float u double
```

Eksplicitna pretvaranja odvijaju se po sljedećim pravilima:

```
Iz sbyte u byte, ushort, uint, ulong, char
Iz byte u sbyte, char
Iz short u sbyte, byte, ushort, uint, ulong, char
Iz ushort u sbyte, byte, short, char
Iz int u sbyte, byte, short, ushort, uint, ulong, char
Iz uint u sbyte, byte, short, ushort, int, char
Iz long u sbyte, byte, short, ushort, int, uint, ulong, char
Iz ulong u sbyte, byte, short, ushort, int, uint, long, char
Iz char u sbyte, byte, short
Iz float u sbyte, byte, short, ushort, int, uint, long, ulong, char,
decimal
Iz double u sbyte, byte, short, ushort, int, uint, long, ulong, char,
float, decimal
Iz decimal u sbyte, byte, short, ushort, int, uint, long, ulong, char,
float, double
```

2.4. Referentni tipovi

Referentni tip varijable kao vrijednost sadrži adresu memorijske lokacije na kojoj se nalazi podatak koji varijabla predstavlja. Najvažniji referentni tip je klasa. O klasama će više biti riječi kasnije, za sada samo to da je klasa korisnički tip podataka koji se sastoji između ostalog od varijabli vrijednosnog tipa i metoda (funkcija) koje uglavnom rade s tim varijablama. Npr. klasa može biti tip podataka *Zrakoplov* koji nije ugrađen u jezik, ali se sastoji od ugrađenih vrijednosnih tipova kao npr.

```
uint BrojPutnika
double Brzina
byte GodinaProizvodnje
```
itd…

Bogatstvo FCL-a upravo je u velikom broju tako definiranih klasa.
Deklaracija klase *Zrakoplov* izgledala bi ovako:

```
class Zrakoplov
{
      uint BrojPutnika;
      double Brzina;
      byte GodinaProizvodnje;
      float KolicinaGoriva() {…}    // Metoda koja vraća količinu goriva
      …
}
```

Varijable tako definirane klase zovu se **objekti** pa otud i naziv objektno orijentirano programiranje što i ukazuje na važnost takve varijable. Kreiranje objekta radi se pozivom:

`Zrakoplov Tupoljev = new Zrakoplov();` gdje je:
`new` - ključna riječ koja kreira objekt tipa *Zrakoplov*.
`Zrakoplov()` – tzv. konstruktor odnosno metoda koja rezervira memorijski prostor za objekt.

Referentne varijable se prilikom prosljeđivanja kao parametar u funkciji prenose **po referenci**, odnosno prenosi se adresa memorijske lokacije na kojoj se nalazi podatak koji objekt predstavlja, dok se vrijednosne varijable prenose **po vrijednosti**.
To znači da će se kod prenošenja referentnih varijabli prenijeti baš ta varijabla, dok će se kod prenošenja vrijednosnih varijabli prenijeti kopija varijable, odnosno neka druga varijabla koja ima istu vrijednost kao originalna. I tu prestaje svaka veza kopije vrijednosne varijable s originalom.

```
using System;

class TempClass
{
      public int Counter;

      public TempClass(int x)
      {
            Counter = x;
      }
}

class Test
{
      static void Main()
      {
            int Val = 5;
            Console.WriteLine("Prije poziva funkcije: " + Val);
            PromijeniVal(Val);
            Console.WriteLine("Poslije poziva funkcije: " + Val);
            Console.WriteLine();

            TempClass Ref = new TempClass(5);
            Console.WriteLine("Prije poziva funkcije: " + Ref.Counter);
            PromijeniRef(Ref);
```

```
            Console.WriteLine("Poslije poziva funkcije: " + Ref.Counter);
      }

      public static void PromijeniVal(int Value)
      {
            Value = 10;
      }

      public static void PromijeniRef(TempClass Ref)
      {
            Ref.Counter = 10;
      }
}
```

Primjer 2.7

Ispis:
Prije poziva funkcije: 5
Poslije poziva funkcije: 5

Prije poziva funkcije: 5
Poslije poziva funkcije: 10

Možemo primijetiti da se vrijednost varijable *Val* iz funkcije *Main* nije promijenila pozivom funkcije *PromijeniVal* zato što je *Val* varijabla vrijednosnog tipa i funkciji je prenesena kopija te vrijednosti. To što se varijabla i u funkciji *PromijeniVal* zove *Val* nema nikakva značaja, mogla se zvati bilo kako (naravno ne baš bilo kako, moraju se poštovati pravila za nazive identifikatora).

Kod referentne varijable odnosno objekta *Ref* (*ref* je ključna riječ i tako ne možemo nazvati varijablu, ali Ref nije ključna riječ!) funkciji *PromijeniRef* prenosi se adresa objekta što znači da svaka promjena na objektu *Ref* u funkciji *PromijeniRef* ima utjecaja na originalnu varijablu *Ref*. Kao i kod *PromijeniVal*, i varijabla *Ref* u *PromijeniRef* mogla se zvati bilo kako.

2.5. Tip String

String je klasa, dakle referentni tip koji se koristi za spremanje niza znakova, a zbog važnosti i čestog korištenja definiran je alijas *string* koji se može koristiti umjesto punog naziva *System.String*.
Deklariranje i definiranje stringa je jednostavno:

```
string Text = "Ovo je tekst";
string Text = "Ovo je " + "tekst";
```

C# donio je novost u kreiranju stringova koji sadrže neke specijalne znakove kao npr. znak '\'.
U C/C++, ako bismo željeli definirati string koji bi sadržavao putanju do neke mape npr. C:\ProgramFiles\VisualStudio, to bismo napravili ovako:

```
string Path = "C:\\ProgramFiles\\VisualStudio";
Console.WriteLine(Path);   // Ispisuje C:\ProgramFiles\VisualStudio
```

Umjesto jednog znaka '\' morali bismo staviti dva znaka.
C# to pojednostavljuje omogućujući postavljanje znaka '@' ispred stringa pa se znak '\' piše kao svaki drugi znak, dakle samo jednom:

```
string path = @"C:\ProgramFiles\VisualStudio";
```

Ipak, ovo nije obvezno, pa se za definiranje ovakvih nizova znakova može koristiti i C/C++ način.

Ako treba ispisati navodnike unutar stringa, treba ih podvostručiti:

```
string Text = @"Bond, ""James Bond""";
Console.WriteLine(Text);
```
Ispisuje: Bond, "James Bond"

Kao što smo rekli, string je klasa, dio FCL-a i kao takav sadrži dosta metoda koje se mogu koristiti u radu sa stringovima. Neke od metoda su statičke što znači da za njihovo pozivanje nije potrebno deklarirati instancu odnosno objekt tipa string, dok neke nisu statičke što znači da je za njihovo korištenje potrebno kreirati objekt tipa string.

Primjeri statičkih metoda:

Concat Spajanje dvaju stringova:
```
string Text = String.Concat("Internet ", "Explorer");
Console.WriteLine(Text);
```
Ispisuje: Internet Explorer

Compare Uspoređivanje dvaju stringova, funkcija *Compare* vraća

- negativni cijeli broj ako je prvi string manji od drugoga (dolazi prije po abecedi)
- 0 ako su stringovi jednaki
- pozitivni cijeli broj ako je prvi string veći od drugoga (dolazi kasnije po abecedi)

```
Console.WriteLine(String.Compare("Croatia", "Hrvatska"));
```
Ispisuje: -1
```
Console.WriteLine(String.Compare("Hrvatska", "Hrvatska"));
```
Ispisuje: 0
```
Console.WriteLine(String.Compare("Hrvatska", "Croatia"));
```
Ispisuje: 1

Primjeri nestatičkih metoda:

Substring Ekstrahiranje dijela stringa
```
string Text = "Hrvatska je prvak svijeta";
Console.WriteLine(Text.Substring(12, 5));
```
Ispisuje: prvak

Funkcija *Substring* u gornjem primjeru ekstrahira 5 znakova počevši od 13. znaka. C# radi indeksaciju počevši s 0 tako da prvi znak u nizu ima indeks 0, drugi ima indeks 1, 13. znak ima indeks 12, itd.
```
Console.WriteLine(Text.Substring(0, 8));
```
Ispisuje: Hrvatska

ToLower Pretvaranje svih znakova u male znakove
```
string Text = "This Is New Class For pRoteCted DELEGATE";
Console.WriteLine(Text.ToLower());
```
Ispisuje: this is new class for protected delegate

Gornji primjer ispisuje ni manje ni više nego 7 ključnih riječi jezika. Ključne riječi mogu biti sadržaj stringa!

Sve metode klase *String* su preopterećene što znači da se mogu pozvati s različitim brojem i tipom argumenata.

```
Console.WriteLine(String.Compare("csharp", "CSHARP"));        // -1
Console.WriteLine(String.Compare("csharp", "CSHARP", true));  //  0
Console.WriteLine(String.Compare("csharp", "CSHARP", false)); // -1
```

Funkciju *Compare* možemo pozvati s dva argumenta koji se uspoređuju i razlikuju se mala i velika slova, a može se pozvati i s tri argumenta gdje je treći argument tipa *bool* i određuje treba li razlikovati velika i mala slova (*true* - ne razlikuje, *false* – razlikuje). Vidimo da je treći primjer istovjetan prvom, odnosno da se pri uspoređivanju stringova po "defaultu" razlikuju mala i velika slova.

Budući da je ovo knjiga o C#, a ne o .Net Frameworku, za pregled ostalih metoda pogledajte dokumentaciju .Net Frameworka.

Objekt tipa *string* je nepromjenjiv (*immutable*) što znači da svaka radnja koja mijenja string stvara kopiju stringa ostavljajući prethodni sve dok se ne ukloni mehanizmom *čišćenja memorije (garbage collection)* koji sustav automatski provodi. Pogledajmo sljedeći primjer:

```
string Text = "";
for(int i=0;i<1000;i++)
      Text += "CSharp";
```

U varijabli *Text* bit će niz znakova koji se sastoji od 1000 puta riječ *CSharp.* Ali nakon što se to dogodi, u memoriji će ostati 999 stringova koji moraju čekati postupak čišćenja memorije da bi se uklonili iz memorije. Očito, vrlo neefikasan način za ovakav zadatak.

Srećom, postoji klasa *StringBuilder* koja omogućava ovo isto, ali bez suvišnog ostavljanja kopija objekata.

```
StringBuilder Text = new StringBuilder(10000);
for(int i=0; i<1000; i++)
      Text.Append("CSharp");
```

Kod definiranja objekta *Text* potrebno je odrediti početni kapacitet koji predstavlja maksimalan broj znakova koji mogu biti sadržani u objektu.

2.6. Formatiranje stringa

Format je jedna od vrlo upotrebljivih metoda klase *String*. To je funkcija koja omogućuje raznovrsno formatiranje podataka i koja vraća tako formatirani tip string.

```
String str = String.Format("{0} {1}", "New ", "York");
```
Varijabla *str* sadržavat će tekst *New York*.

Kao što je već napomenuto, kod poglavlja o argumentima naredbene linije, argument funkcije *Format* je formatirani string koji u prvom dijelu ima parametre u vitičastim zagradama {0} i {1} što znači da se u drugom dijelu ovako formatiranog stringa očekuju vrijednosti koje će

biti ispisane. Broj parametara je proizvoljan s tim da se poštuje numeracija pa tako drugi parametar ima oznaku 1, treći 2, itd.
Parametar može sadržavati i način ispisa, ovisno o tome želi li se ispisivati cijeli broj, decimalni broj, broj s oznakom valute, datum itd.

```
using System;

class Test
{
      public static void Main()
      {
            String str = String.Format("{0} {1}", "PDV", "uključen");
            Console.WriteLine(str);

            int n = 2;
            str = String.Format("{0} {1} {2:d}. {3}", "PDV", "uključen", n,
                                                      "put");
            Console.WriteLine(str);

            double CijenaKabela = 500.68;
            str = String.Format("{0,-20} {1,7:f}", "Cijena kabela je",
                                                CijenaKabela);
            Console.WriteLine(str);

            double CijenaUticnice = 15.68;
            str = String.Format("{0,-20} {1,7:f}", "Cijena utičnice je",
                                                CijenaUticnice);
            Console.WriteLine(str);

            str = String.Format("{0} {1:c}", "Cijena kabela je",
                                                CijenaKabela);
            Console.WriteLine(str);

            str = String.Format("{0} {1:d}", "Danas je", DateTime.Now);
            Console.WriteLine(str);
            str = String.Format("{0} {1:d}. {2}", "Danas je",

            DateTime.Now.DayOfYear, "dan u godini");
            Console.WriteLine(str);
      }
}
```

Primjer 2.8

Ispis:

```
PDV uključen
PDV uključen 2. put
Cijena kabela je      500,68
Cijena utičnice je     15,68
Cijena kabela je 500,68 kn
Danas je 4.12.2006
Danas je 338. dan u godini
```

Primijetite da se znak *d* kao oznaka u parametru može koristiti na različite načine. Ako je na desnoj strani *integer*, ispisuje se taj *integer*, a ako je na desnoj strani funkcija *DateTime.Now*, ispisuje se današnji datum. Funkcija *Format* zna što treba ispisati, ovisno o tome što se nalazi na desnoj strani.

Također obratite pozornost na poravnanje slijeva i zdesna u ispisu cijena kabela i utičnice. Za poravnanje slijeva koriste se negativni brojevi odnosno prefiks '-', dok se za poravnanje zdesna koriste pozitivni brojevi.

2.7 Tip object

Klase FCL-a izrađene su u obliku hijerarhijskih nivoa. Pojedina klasa može nasljeđivati drugu klasu, ali za razliku od C++ nema višestrukog nasljeđivanja. O tome će više biti govora u poglavlju o klasama.
Nasljeđivanje znači da klasa koja nasljeđuje preuzima neke varijable i metode iz klase koju nasljeđuje.
Tako svi tipovi varijabli nasljeđuju klasu *System.Object* i to je osnovna bazna klasa za sve tipove podataka. *System.Object* kao i *System.String* ima svoj alijas, pa je ravnopravno za tu klasu koristiti ključnu riječ `object`.

Budući da svi vrijednosni tipovi nasljeđuju klasu *object,* oni se mogu i implicitno konvertirati u tip *object.* Taj se postupak zove **boksiranje (boxing)**. Obrnuti postupak eksplicitnog konvertiranja varijable tipa *object* u vrijednosni tip zove se **odboksiranje (unboxing)**.

Primjer boksiranja:

```
using System;

class Test
{
      public static void Main()
      {
            int i = 100;
            object o = i;
            i = 200;
            Console.WriteLine("Tip int: {0}", i);
            Console.WriteLine("Tip object: {0}", o);
      }
}
```

Primjer 2.9

Ispis:
Tip int: 200
Tip object: 100

U primjeru je s naredbom *object o = i* provedeno implicitno pretvaranje tipa *integer* u tip *object*. Boksiranje omogućuje da se vrijednosni tip tretira kao referentni tip. Naravno, varijabla *i* ostaje vrijednosni tip i promjena vrijednosti varijable *i* ne utječe na varijablu *o*;

Primjer odboksiranja:

```
using System;

class Test
{
      public static void Main()
      {
            int i = 100;
            object o = i;
            int j = (int)o;
            Console.WriteLine("Tip int: {0}", j);
```

```
        }
}
```

Primjer 2.10

Ispis:
Tip int: 100

Varijabla *o* je s naredbom *int j = (int)o* eksplicitno konvertirana u integer.
Tip objekt mora se odboksirati u tip iz kojeg je boksiran. Ovo bi u gornjem primjeru bila greška:

```
long j = (long)o;
```

Tip objekt ima definiranu metodu *ToString* koju nasljeđuju svi tipovi varijabli. Metoda će kod vrijednosnih varijabli vratiti njihovu vrijednost pretvorenu u string, dok će kod klasa vratiti ime klase.

```
using System;

class TempClass
{
        private int x;
}


class Test
{
        public static void Main()
        {
                int i = 35;
                TempClass Tmp = new TempClass();
                Console.WriteLine(i.ToString());
                Console.WriteLine(Tmp.ToString());
        }
}
```

Primjer 2.11

Ispis:
35
TempClass

Console.WriteLine(i) će ispisati također 35, iako je argument *integer*. To je zato jer je funkcija *WriteLine* u klasi *Console* preopterećena i postoji više definiranih funkcija *WriteLine* s različitim argumentima. Ova koja ima *integer* kao argument ovako nekako je definirana:

```
public string WriteLine(int x)
{
        return x.ToString();
}
```

Ovo je definirano u sklopu FCL da bi se brže i lakše pisao kôd. Programer ne mora razmišljati kojeg je tipa argument koji se prosljeđuje funkciji, za svaki tip varijable postoji definirana funkcija *WriteLine*.

```
public string WriteLine(long f);
public string WriteLine(double d);
itd …
```

Kod klasa je iznimka ako klasa ima svoju definiranu funkciju *ToString* koja onda može vratiti neki drugi string različit od stringa koji predstavlja ime klase. O tome će više riječi biti u poglavlju o klasama.

2.8 Tip enum

Enum je poseban vrijednosni tip koji se sastoji od nekoliko tipova *integer* koji su predstavljeni simboličkim imenima.

```
enum Duga
{
      Red,
      Orange,
      Yellow,
      Green,
      Blue,
      Navy,
      Purple
}
```

U ovom primjeru definirano je sedam simboličkih naziva od kojih svaki predstavlja jedan *integer*. Indeksacija i ovdje počinje s 0 pa tako *Red* ima vrijednost 0, *Orange* 1 itd…

Ipak, bez obzira što su varijable koje čine tip *enum* cjelobrojne vrijednosti, potrebna je eksplicitna konverzija u tip *int*.

```
int i = Duga.Blue;      // Neispravno
int i = (int)Duga.Blue; // OK
```

Ako želimo dodijeliti različite vrijednosti od automatskog numeriranja od 0 do n, potrebno je to definirati u definiciji *enum* tipa:

```
enum Duga
{
      Red = 50,
      Orange,
      Yellow,
      Green,
      Blue,
      Navy,
      Purple
}
```

Sad *Red* ima vrijednost 50, *Orange* 51, itd.

```
enum Duga
{
      Red,
      Orange,
      Yellow = 50,
      Green,
      Blue,
      Navy = 100,
      Purple = 200
}
```

Sad *Red* ima vrijednost 0, *Orange* 1, *Yellow* 50, *Green* 51, *Blue* 52, *Navy* 100 i *Purple* 200.

Mogu se koristiti i negativne vrijednosti, pa ako dodijelimo za *Red* vrijednost –100, onda će *Orange* imati vrijednost –99 itd.

Varijable u *enum* mogu biti i drugog tipa različitog od *integer*. To se ovako definira:

```
enum Duga : uint
{
      . . .
}
```

Naravno, sad *Red* ne bi mogao imati negativnu vrijednost jer je riječ o neoznačenom tipu *uint*.

Poglavlje 3:
Operatori

Ovo poglavlje obrađuje:

- načine podjele operatora
- vrste operatora
- operatore za rad s bitovima
- prioritet operatora

3.1. Vrste operatora

Operatori su simboli koji se navode u naredbama programskog jezika i koji određuju koje operacije će biti izvedene nad varijablama.

Operatore možemo podijeliti na više različitih načina. Jedna od podjela je s obzirom na broj varijabli na koje se operator odnosi, pa se tako operatori dijele na:

- Unarne operatore koji utječu na samo jednu varijablu (operatori ++ i --)
- Binarne operatore za čije djelovanje su potrebne dvije varijable ili konstante (većina operatora su binarni)
- Ternarni operatori koji imaju djelovanje na tri varijable (postoji samo jedan takav operator)

Prirodnija i prihvatljivija je podjela s obzirom na kategoriju pa tako postoje sljedeće kategorije operatora:

- Aritmetički operatori (`+, -, *, /, %`)
- Operatori inkrementiranja i dekrementiranja (`++, --`)
- Operatori uspoređivanja vrijednosti (`==, !=, <, >, <=, >=`)
- Operatori dodjeljivanja (`=, +=, -=, *=, /=, %=`)
- Uvjetni operatori (`?:`)
- Logički operatori (`&, |, ^, !, ~, &&, ||, true, false`)
- Operatori za pomak bitova (`<<, >>`)

Postoji još nekoliko kategorija operatora koji će biti obrađeni u poglavlju o klasama.

Aritmetički operatori

Aritmetičke operatore najjednostavnije je shvatiti jer predstavljaju osnovne matematičke operacije.

```
x = y + z;
y = z * 2 – x;
s = "Dobar " + "dan";
```

Ako je u istoj naredbi više operatora, treba obratiti pozornost na redoslijed izvršenja operacija. Jednako kao i u matematici, operatori '`*`' i '`/`' su operatori s višim prioritetom nego operatori '`+`' i '`-`'.

```
int x = 5;
int y = 6;
int z = x + y * 2;  // U z će biti vrijednost 17, a ne 22
```

Opet, kao u matematici, prioritet se može promijeniti korištenjem zagrada:

```
int z = (x + y) * 2;  // U z će biti vrijednost 22
```

Poseban aritmetički operator je operator *mod* '`%`' koji vraća ostatak kod dijeljenja.

```
int x = 12;
```

```
int y = 5;
int z = x % y;  // U z će biti vrijednost 2
x = 10;
z = x % y;  // U z će biti vrijednost 0
```

Za razliku od C/C++ u C# operator *mod* možemo primijeniti i na tipove s pomičnim zarezom:

```
double x = 5.4;
double y = 2.6;
double z = x % y;  // U z će biti vrijednost 0.2
```

Operatori inkrementiranja i dekrementiranja

Operatori inkrement ++ i dekrement – su operatori koji povećavaju odnosno smanjuju vrijednost varijable za 1.

```
int C = 1972;
C++;   // U varijabli C će biti vrijednost 1973
C--;   // U varijabli C će ponovo biti vrijednost 1972
```

Treba obratiti pozornost na to dolaze li operatori inkrement i dekrement prije ili poslije naziva varijable. U oba slučaja će vrijednost varijable biti uvećana za 1, ali ipak postoji razlika.
Ako je operator ispred varijable, tada će se povećanje za 1 dogoditi prije uporabe vrijednosti varijable, a ako je operator iza varijable povećanje će uslijediti nakon uporabe vrijednosti varijable.

```
int x = 10;
int y = x++;    // U varijabli y će biti vrijednost 10

int x = 10;
int y = ++x;    // U varijabli y će biti vrijednost 11
```

Naročito treba obratiti pozornost ako se ovi operatori koriste u *if* naredbama.

```
int x = 10;
int y = 1;
if(x++ > 10)
      y--;
// U varijabli y će biti vrijednost 1

int x = 10;
int y = 1;
if(++x > 10)
      y--;
// U varijabli y će biti vrijednost 0
```

U prvom slučaju događa se najprije uspoređivanje vrijednosti varijable *x* s brojem 10, a tek nakon toga se varijabla *x* povećava za 1 korištenjem operatora ++.
U drugom slučaju najprije se varijabla *x* poveća za 1, a tek onda se vrši uspoređivanje s brojem 10.
Zato *if* uvjet prolazi u drugom slučaju, a u prvom ne.

Operatori uspoređivanja vrijednosti

Kao što im samo ime kaže, koriste se za uspoređivanje vrijednosti dviju varijabli ili konstanti.

==	Je li jednako
!=	Je li različito
<	Je li manje od
>	Je li veće od
<=	Je li manje ili jednako od
>=	Je li veće ili jednako od

Operator *je li jednako* == uspoređuje vrijednosti varijabli i ako su one jednake, operator vraća *true*.

```
int x = 10;
int y = 10;
if(x == y)
      Console.WriteLine("x i y su jednaki");
else
      Console.WriteLine("x i y nisu jednaki");
```

Ispis:
x i y su jednaki

Operator *je li različito* != uspoređuje vrijednosti varijabli i ako su vrijednosti varijabli različite, operator vraća *true*.

```
int x = 10;
int y = 10;
if(x != y)
      Console.WriteLine("x i y nisu jednaki");
else
      Console.WriteLine("x i y su jednaki");
```

Ispis:
x i y su jednaki

Ostali operatori za uspoređivanje vrijednosti su sami po sebi razumljivi:

```
int x = 5;
int y = 4;
if(x < ++y)
      x = 10;
```

U x će ostati vrijednost 5 jer *if uvjet* neće biti ispunjen budući da će se vrijednost varijable y povećati za 1 prije nego što se napravi uspoređivanje.

Operatori dodjeljivanja

Operator '=' je osnovni operator dodjeljivanja i njegovim korištenjem dodjeljuje se vrijednost s njegove desne strane varijabli s lijeve strane.

```
int x = 5;
int y = x + 10;
int z = x++ + ++y;       // z će biti 21 (5 + 16)

int a, b, c;
```

```
a = b = c = 3;          // a, b, c će biti 3
```

Posebni su operatori dodjeljivanja **+=, -=, *=, /=, %=** i oni su često zbunjujući za početnike.

```
x += y;
```
je ekvivalentno ovome:
```
x = x + y;
```
uz razliku da je u prvom slučaju *x* izračunat samo jednom, čime se dobiva na brzini izvršenja.

```
using System;

public class Test
{
      public static void Main()
      {
            int x = 7, y = 5;
            x += y;
            Console.WriteLine(x);
            x *= y;
            Console.WriteLine(x);
            x /= 4;
            Console.WriteLine(x);
            x -= 2;
            Console.WriteLine(x);
            x %= y;
            Console.WriteLine(x);
      }
}
```

Primjer 3.1

Ispis:
12
60
15
13
3

Tek da bismo pokazali koliko C# može pakirati naredbe, ovo isto mogli smo dobiti i ovako:

```
using System;

public class Test
{
      public static void Main()
      {
            int x = 7, y = 5;
            Console.WriteLine("{0}\n{1}\n{2}\n{3}\n{4}",
                        (x += y), (x *= y), (x /= 4), (x -= 2), (x % y));
      }
}
```

Primjer 3.2

Primijetite da su izrazi `x += 1` i `x -= 1` suvišni jer imamo operatore ++ i -- koji rade isto.

Uvjetni operatori

Možda bi bolji podnaslov bio u jednini jer uvjetni operator postoji samo jedan **?:**. Koristi se kada je u ovisnosti o vrijednosti varijable potrebno izvršiti jednu ili drugu radnju.

```
int x = 5, y;
y = x==5 ? 1 : 2; // U y će biti 1
```

Uvjetni operator sastoji se od dva dijela:
'**?**' s čije lijeve strane se nalazi uvjet koji se provjerava, te
'**:**' s čije lijeve strane se nalazi izraz koji se izvršava i vraća vrijednost ako je uvjet istinit, a s desne strane izraz koji se izvršava te vraća vrijednost ako je uvjet lažan.

U ovom primjeru uvjet koji se provjerava je "je li x jednak 5", a vrijednosti koje se vraćaju su 1 i 2. Budući da je *x* jednak 5, vraćena vrijednost će biti 1.
Ovaj operator zamjenjuje klasičnu *if-else* naredbu jer smo ovo isto mogli napisati:

```
if(i == 5)
      y = 1;
else
      y = 2;
```

ali je korištenje operatora za ovako nešto elegantnije i brže rješenje.
U jednoj naredbi možemo navesti više uvjetnih operatora koji zamjenjuju niz *if else* naredbi. Pogledajmo sljedeći primjer:

```
string Medalja;
int Plasman = 1;
Medalja =  Plasman == 1 ? "Zlatna" : Plasman == 2 ?
                          "Srebrna" : Plasman == 3 ?
                          "Brončana" : "Bez medalje";
```

U varijabli *Medalja* će biti "Zlatna" ako je vrijednost varijable *Plasman* jednaka 1, "Srebrna" ako je *Plasman* jednak 2, "Brončana" ako je *Plasman* jednak 3, te "Bez medalje" ako je u varijabli *Plasman* neka druga vrijednost.

Obratite pozornost da smo u definiciji uvjetnog izraza naveli da se s lijeve i desne strane znaka '**:**' nalazi izraz koji se izvršava te vraća vrijednost. To znači da u uvjetnom operatoru možemo koristiti i poziv funkcije koja će se izvršiti te vratiti neku vrijednost.

```
using System;

public class Test
{
      public static void Main()
      {
            double Result, Kut = Math.PI / 6;
            Result = Kut < Math.PI ? Math.Sin(Kut) : Math.Sin(Kut);
            Console.WriteLine(Result);
      }
}
```

Primjer 3.3

Ispis:
0.5

Ako je *Kut* manji od 180° (PI radijana, *Math.PI* je konstanta definirana u klasi *Math* s vrijednosti broja PI) izvršava se funkcija *Sin* iz klase *Math* te se rezultat vraća kao povratna vrijednost. Ako je *Kut* veći ili jednak 180°, izvršava se funkcija *Cos*.

Logički operatori

Logičke operatore možemo podijeliti u dvije grupe

- operatori za rad s varijablama (**&&, ||, !**)
- operatori za rad s bitovima (**&, |, …**)

Operator && izvodi logički AND (I), operaciju koja će vratiti istinu ako su oba uvjeta istinita.
Operator || izvodi logički OR (ILI), operaciju koja će vratiti istinu ako je barem jedan od uvjeta istinit.

```
int x = 2, y = 10;
if(x > 1  &&  y < 20)    // if uvjet je istinit jer su oba uvjeta istinita
      // ...
if(x > 1  &&  y < 10)    // if uvjet je lažan jer oba uvjeta nisu istinita
      // ...
if(x > 1  ||  y < 10)    // if uvjet je istinit jer je prvi uvjet istinit
      // ...
if(x > 4  ||  y < 10)    // if uvjet je lažan jer niti jedan uvjet nije istinit
```

Operatori && i || mogu se koristiti u istom izrazu, ali treba obratiti pozornost na prioritet (&& ima veći prioritet od ||) kao u sljedećem primjeru.
Program će ispisati za svaku godinu u prošlom i ovom stoljeću je li prijestupna ili ne. Ako niste znali, definicija prijestupne godine nije jednostavno ona koja je djeljiva sa 4.
Prijestupne godine su one koje su djeljive sa 400 plus one koje su djeljive sa 4, ali ne i sa 100.

```
using System;

public class Test
{
      public static void Main()
      {
            for(int i = 1900; i <= 2100; i++)
                  if((i % 400 == 0) || (i % 4 == 0  &&  i % 100 != 0))
                        Console.WriteLine(i + " je prijestupna");
                  else
                        Console.WriteLine(i + " nije prijestupna");
      }
}
```

Primjer 3.4

Ispis:
1900 nije prestupna
1901 nije prestupna
1902 nije prestupna
1903 nije prestupna
1904 je prestupna
...
2099 nije prestupna
2100 nije prestupna

Primijetite da godine 1900. i 2100. nisu prijestupne jer ne zadovoljavaju niti jedan od dva uvjeta, dok je godina 2000. prijestupna, a također ne zadovoljava drugi uvjet, ali zadovoljava prvi uvjet (djeljiva sa 400). Budući da su uvjeti povezani OR operatorom, taj jedan uvjet je dovoljan.

Drugi uvjet u *if* naredbi morao je biti stavljen u zagrade jer || ima niži prioritet od &&.

Operator ! (NOT) najjednostavnije rečeno pretvara *true* u *false*.

```
bool x = true;
bool y = !x;      // y je false
y = !y;           // y je true
```

Operator NOT možemo koristiti za pretvaranje cijelog istinitog izraza u lažni ili obrnuto. Pogledajmo sada onaj primjer s prijestupnim godinama korištenjem operatora NOT.

```
using System;

public class Test
{
      public static void Main()
      {
            for(int i = 1900; i <= 2100; i++)
                  if(!(i % 400 == 0) && !(i % 4 == 0  &&  i % 100 != 0))
                        Console.WriteLine(i + " nije prijestupna");
                  else
                        Console.WriteLine(i + " je prijestupna");
      }
}
```

Primjer 3.5

Rezultat ispisa je isti.

Ovo sve mogli smo riješiti jednostavnije koristeći funkciju iz .Net Frameworka *IsLeapYear*, međutim htjeli smo pokazati kako se izvršavaju izrazi s više operatora.

Funkcija IsLeapYear ne radi dobro za godine prije 1582. odnosno prije uvođenja gregorijanskog kalendara! Naime, za godine prije 1582. prijestupna godina je bila svaka koja je djeljiva sa 4. Općenito, funkcije datuma pogrešno rade za te godine jer ne rade automatsko prebacivanje na julijanski kalendar, pa u slučaju da vam to zatreba, imajte na umu.
Columbo je otkrio Ameriku u petak 12.10.1492. a ako zatražite od klase DateTime da vam to izračuna:

```
DateTime dt = new DateTime(1492, 10, 12);
Console.WriteLine(dt.DayOfWeek);
```

dobit ćete da je bila srijeda!

Kod operatora && i || dobro je znati da se provjera istinitosti prekida u slučaju:

- && - ako je prvi uvjet lažan jer će tada bez obzira na ostale uvjete cijeli izraz biti lažan
- || - ako je prvi uvjet istinit jer će tada bez obzira na ostale uvjete cijeli izraz biti istinit.

To je osobito bitno ako je drugi uvjet funkcija:

```
using System;

public class Test
{
	public static void Main()
	{
		int Flag = 0;
		bool Uvjet = false;
		if(Uvjet  &&  VeryImportant())
			Flag = 1;
	}

	public static bool VeryImportant()
	{
		int x = 0;
		// Obavljanje vrlo važnog posla ...
		return x == 0 ? true : false;
	}
}
```

Primjer 3.6

Funkcija *VeryImportant()* u ovom slučaju se neće izvršiti jer prva provjera u *if* naredbi daje *false*. Primijetite da nije potrebno navesti:

```
if(Uvjet == true)
```

Dovoljno je staviti:

```
if(Uvjet)
```

Isto tako ako biste stavili:

```
bool Uvjet = true;
if(Uvjet  ||  VeryImportant())
```

funkcija *VeryImportant()* ne bi se izvršila jer provjera prestaje nakon zadovoljenja prvog uvjeta.

Operatori za rad s bitovima & i | također obavljaju logičke funkcije AND i OR, ali nad bitovima.

```
int x = 5;  // 0101
int y = 7;  // 0111
int z = x & y;
```

U *z* će biti vrijednost 5 jer će se dogoditi sljedeće:

x	0	1	0	1
y	0	1	1	1
z	0	1	0	1

```
1 & 1 = 1
1 & 0 = 0
0 & 1 = 0
0 & 0 = 0
```

Sukladno tome za operator |

```
int x = 5;  // 0101
int y = 7;  // 0111
int z = x | y;
```

u *z* će biti vrijednost 7 jer će se dogoditi sljedeće:

x	0	1	0	1
y	0	1	1	1
z	0	1	1	1

```
1 | 1 = 1
1 | 0 = 1
0 | 1 = 1
0 | 0 = 0
```

Operatori za pomak bitova

C# ima operatore << i >> koji pomiču bitove ulijevo (<<) i udesno (>>) za određeni broj mjesta.

```
int x = 8;              // 1000
int y = x >> 1;         // 0100  pomak bitova za jedno mjesto udesno
int z = x >> 2;         // 0010  pomak bitova za dva mjesta udesno

int x = 8;              // 001000
int y = x << 1;         // 010000  pomak bitova za jedno mjesto ulijevo
int z = x << 2;         // 100000  pomak bitova za dva mjesta ulijevo
```

Vidimo da se kod pomicanja bitova upražnjena mjesta ispunjavaju s 0.

typeof operator

Ovaj operator će vratiti objekt tipa *SystemType* za određeni tip varijable.

```
using System;

class Test
{
      static void Main()
      {
            Type t1 = typeof(int);
            Type t2 = typeof(System.Int32);
            Type t3 = typeof(string);

            Console.WriteLine(t1.FullName);
            Console.WriteLine(t2.FullName);
            Console.WriteLine(t3.FullName);
            Console.WriteLine(t2.IsValueType);
            Console.WriteLine(t3.IsValueType);

      }
}
```

Primjer 3.7

Ispis:
System.Int32
System.Int32

```
System.String
True
False
```

Tipovi *int* i *System.Int32* vraćaju isti objekt jer, kao što smo već rekli, *int* je samo alijas za *System.Int32*. Kad se objekt dobije kao povratna vrijednost, onda se mogu pozvati razne metode koje pripadaju tom objektu kao što su:

FullName koja vraća puno ime tipa
IsValueType koja vraća *true* ako je tip vrijednosni odnosno *false* ako je referentni.

3.2. Prioritet operatora

Prioritet operatora određuje koji će se operator prije izvršiti u slučaju da u jednom izrazu imamo više različitih operatora.

Već smo vidjeli primjer s aritmetičkim operatorima +, -, * i /, a slično je i s ostalima. Pravilo je jednostavno, najprije se izvršavaju operatori s višim prioritetom, pa tek onda oni s nižim, a prioritet operatora može se promijeniti zagradama.

```
bool x = true, y = false;
return y  &&  !x ? true : false;
```

Ovdje je programer vjerojatno htio provjeriti s operatorom ?: je li *x* istinit, a tek onda napraviti *&&* između *y* i povratne vrijednosti operatora ?:.
Međutim zbog višeg prioriteta operatora && najprije bi se izvelo `y && !x`, a tek onda bi se tako dobiveni rezultat provjeravao u uvjetnom operatoru. Ako stavimo zagrade, onda je i prioritet promijenjen.

```
return y  &&  (!x ? true : false);
```

Tablica prioreta operatora:

Operator	Asocijativnost
(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked	S lijeva na desno
+(unary) -(unary) ~ ++x --x (T)x	S lijeva na desno
* / %	S lijeva na desno
+ -	S lijeva na desno
<< >>	S lijeva na desno
< > <= >= is as	S lijeva na desno
== !=	S lijeva na desno
&	S lijeva na desno
^	S lijeva na desno
|	S lijeva na desno
&&	S lijeva na desno
||	S lijeva na desno
?:	S desna na lijevo
= *= /= %= += -= <<= >>= &= ^= |=	S desna na lijevo

Asocijativnost određuje kako će operatori biti vrjednovani ako se u jednom izrazu nalazi više različitih operatora istog prioriteta.

```
int x = 4 * 12 % 5;
```

U *x* će biti vrijednost 3 jer će se najprije izračunati 4 * 12, a tek onda 48 % 5, dakle slijeva na desno.

Poglavlje 4:
Tijek programa

Ovo poglavlje obrađuje:

- naredbe za uvjetno izvođenje programa
- ugnježđenu *if else* naredbu
- naredbe petlji
- razliku između *while* i *do while* petlje
- naredbe prekida izvršenja (*break*)
- naredbe nastavka izvođenja (*continue*)
- naredbu iteracije kroz niz objekata (*foreach*)
- komentiranje programa
- pretprocesorske direktive

4.1. Naredba if-else

C# opremljen je svim standardnim naredbama za kontrolu tijeka programa. Jedna od njih je i naredba uvjetovanog izvršenja programa, standardna *if else* naredba koja postoji u svim programskim jezicima.

Sintaksa *if else* naredbe:

```
if(Uvjet ispunjen da ili ne)
{
      // Niz naredbi . . .
}
else
{
      // Niz naredbi . . .
}
```

Dakle *if else* naredba koristi se kada je u ovisnosti o tome je li neki izraz istinit ili lažan potrebno izvršiti određeni programski kôd.

```
using System;

public class Test
{
      public static void Main()
      {
            int x = 20, y, z;

            if(x == 20)
            {
                  y = 10;
                  z = 100;
            }
            else
            {
                  y = 11;
                  z = 101;
            }
            Console.WriteLine(y + z);
      }
}
```

Primjer 4.1

Ispis:
110

Ako je uvjet u *if* dijelu *if else* naredbe istinit, izvršavaju se naredbe koje pripadaju *if* bloku ograđenom vitičastim zagradama, a ako nije istinit, izvršava se *else* dio. Kao rezultat izvršenja bit će u varijabli *y* vrijednost 10, a u varijabli *z* 100.
Ako se u bloku koji se treba izvršiti nalazi samo jedna naredba, onda ih nije potrebno ograđivati vitičastim zagradama.

Kao uvjet u *if* dijelu, mora se navesti izraz koji vraća *bool* tip za razliku od C/C++ u kojima se moglo napisati ovako nešto:

```
int x = 20, y = -20;
if(x  &&  y)
```

```
  // ovo bi se izvršilo …
```

jer je u tim jezicima svaki broj različit od 0 vraćao istinu.

Dio *else* u naredbi (*if else* je jedna naredba programskog jezika) je proizvoljan ovisno o tome što zahtijeva programska logika.
Ipak pri tom pripazite, jer ako u gornjem primjeru izbacimo *else* dio, program se ne bi preveo jer bi postojala mogućnost da se varijable *y* i *z* koriste prije, odnosno bez definiranja.

If else naredba može se koristiti i u kontinuiranom obliku gdje postoji više različitih raspona vrijednosti. Pogledajmo primjer u kojem će se ispisati doseg reprezenatacije na Svjetskom prvenstvu u nogometu s obzirom na plasman.

```
using System;

public class Test
{
	public static void Main()
	{
		uint Plasman = 1;
		string Doseg;

		if(Plasman == 1)
			Doseg = "Svjetski prvak";
		else if(Plasman == 2)
			Doseg = "Finale";
		else if(Plasman >= 3  &&  Plasman <= 4)
			Doseg = "Polufinale";
		else if(Plasman >= 5  &&  Plasman <= 8)
			Doseg = "Četvrtfinale";
		else if(Plasman >= 9  &&  Plasman <= 16)
			Doseg = "Osmina finala";
		else if(Plasman >= 17  &&  Plasman <= 32)
			Doseg = "Ispadanje u skupini";
		else
			Doseg = "Nisu se kvalificirali za Svjetsko prvenstvo";

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

Primjer 4.2

Ispis:
Svjetski prvak

Vidimo niz *if else* naredbi koje provjeravaju uvjet i ako je uvjet zadovoljen, dodjeljuje se vrijednost varijabli *Doseg*. Posljednji *else* je bez *if* dijela što znači da će se izvršiti ako svi ostali *else if* i prvi *if* ne zadovolje uvjet.
Primijetite da se nakon prvog ispunjenja uvjeta više ništa ne provjerava i *if else* naredba prestaje s izvršavanjem. Zato ovakav primjer možemo napisati i ovako:

```
using System;

public class Test
{
	public static void Main()
	{
		uint Plasman = 1;
```

```
            string Doseg;

            if(Plasman == 1)
                  Doseg = "Svjetski prvak";
            else if(Plasman <= 2)
                  Doseg = "Finale";
            else if(Plasman <= 4)
                  Doseg = "Polufinale";
            else if(Plasman <= 8)
                  Doseg = "Četvrtfinale";
            else if(Plasman <= 16)
                  Doseg = "Osmina finala";
            else if(Plasman <= 32)
                  Doseg = "Ispadanje u skupini";
            else
                  Doseg = "Nisu se kvalificirali za Svjetsko prvenstvo";

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

Primjer 4.3

Ispis:
Svjetski prvak

Očito je da sad u *else if* dijelu imamo samo jedan uvjet za provjeru dok su u prethodnom primjeru bila dva što bi trebalo rezultirati bržim izvršenjem programa. (Dobro, zanemarimo osobine prevoditelja koji je u stanju sam optimizirati ovakve stvari).

Ako je npr. uvjet `else if(Plasman <= 4)` istinit, to automatski znači da je *Doseg* polufinale. Iako *Plasman = 1 ili 2* također zadovoljava ovaj uvjet, taj slučaj uopće ne može doći do provjere `else if(Plasman <= 4)` jer će već prije biti zadovoljen neki od prethodnih *if* što automatski znači prestanak izvršenja naredbe.

Naredbe *if else* mogu se ugnijezditi, ali tu treba biti jako oprezan da se ne bi dogodilo da se neki *else* odnosi na neki drugi *if*, a ne na onaj koji smo zamislili.

```
using System;

public class Test
{
      public static void Main()
      {
            int x = 1;
            string tekst;

            if(x >= 0)
            if(x == 0)
                  tekst = "x je 0";
            else
                  tekst = "x je veći od 0";
            else
                  tekst = "x je manji od 0";

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

Primjer 4.4

Ispis:

x je veći od 0

U ovom primjeru možemo postaviti pitanje na koji *if* se odnosi pojedini *else*. To se može izbjeći ispravnim nazubljivanjem i korištenjem vitičastih zagrada iako se u bloku nalazi samo jedna naredba.

```
using System;

public class Test
{
	public static void Main()
	{
		int x = 1;
		string tekst;

		if(x >= 0)
		{
			if (x == 0)
			{
				tekst = "x je 0";
			}
			else
			{
				tekst = "x je veći od 0";
			}
		}
		else
		{
			tekst = "x je manji od 0";
		}

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

Primjer 4.5

Ispis:
x je veći od 0

Sad je odmah vidljivo kojem *if* pripada koji *else*.

4.2. Naredba switch

Naredba *switch* koristi se za isto kao i *if else*, ali je oblik drugačiji. Naredbu *switch* trebalo bi koristiti kad imamo puno različitih mogućnosti koje je onda preglednije obraditi ovom naredbom.
Pogledajmo primjer u kojem za dati eksponent treba ispisati njegov naziv:

```
using System;

public class Test
{
	public static void Main()
	{
		int Eksponent = 6;
		string Tekst;

		switch(Eksponent)
```

```
            {
                  case -12:
                        Tekst = "piko";
                        break;
                  case -9:
                        Tekst = "nano";
                        break;
                  case -6:
                        Tekst = "mikro";
                        break;
                  case -3:
                        Tekst = "mili";
                        break;
                  case 3:
                        Tekst = "kilo";
                        break;
                  case 6:
                        Tekst = "mega";
                        break;
                  case 9:
                        Tekst = "giga";
                        break;
                  case 12:
                        Tekst = "tera";
                        break;
                  default:
                        Tekst = Eksponent < 0 ? "Premala veličina" :
                                                "Prevelika veličina";
                        break;
            }

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

Primjer 4.6

Ispis:
mega

Naredba *switch* provjerava vrijednost eksponenta te kada se pronađe vrijednost između niza *case* dijelova, prestaje s izvršenjem. Ako se vrijednost ne podudara niti u jednom *case* dijelu, izvršava se *default* dio koji ima ulogu onog zadnjeg *else* u slučaju kada bismo koristili niz *if else* naredbi.

Ako za dva ili više *case* treba izvršiti iste naredbe, može se ovako pisati:

```
case -13:
case -14:
case  13:
case  14:
      tekst = ". . .";
      break;
```

C# uvodi novost u *switch* naredbi jer se za razliku od C/C++ može provjeravati i tip *string*.

```
int Eksponent;
string Tekst = "nano";

switch(Tekst)
```

```
{
      case "piko":
            Eksponent = -12;
            break;
      case "nano":
            Eksponent = -9;
            break;
      . . .
}
```

Međutim, za razliku od C/C++, C# ne dozvoljava "padanje kroz case" (fall through case):

```
switch(Eksponent)
{
      case -12:
            Tekst = "piko";
            break;
      case -9:
            Tekst = "nano";
      case -6:
            Tekst = "mikro";  // Za Exponent = -9 u C/C++ bi se i ovo
                              // izvršilo
            break;
      . . .
}
```

Ovo se neće prevesti što je i logično, jer bi se u slučaju da je *Eksponent* jednak –9, *switch* naredba nastavila izvršavati, dalje provjeravajući je li *Eksponent* jednak –6 što je očito nemoguće.

4.3. Naredba for

For naredba koristi se za višestruko izvođenje jedne ili više naredbi programskog jezika. Sintaksa *for* naredbe:

```
for(inicijalizacija varijabli ; provjera uvjeta; operacije nad varijablama)
{
      // Niz naredbi . . .
}
```

Možemo reći da se *for* petlja sastoji od dva dijela:

- *for* naredbe koja se sastoji od 3 dijela
  - početna inicijalizacija varijabli (izvršava se na početku *for* petlje)
  - provjera uvjeta, kad uvjet postane lažan *for* petlja prekida s radom (izvršava se na početku svakog prolaza petlje)
  - razne operacije nad varijablama (izvršava se na kraju svakog prolaza petlje)
- bloka naredbi koje se izvršavaju u svakom prolazu petlje

Pogledajmo *for* petlju u primjeru. Potrebno je izračunati zbroj prvih 1000 brojeva. Očito je da je za ovakav zadatak *for* petlja idealna jer je potrebno 1000 puta napraviti istu operaciju, a to je da u neku varijablu dodamo vrijednost.

```
using System;
```

```
class Test
{
	static void Main()
	{
		int Zbroj = 0, Max = 1000;
		for(int i = 1;i <= Max;i++)
			Zbroj += i;
			Console.WriteLine("Zbroj prvih {0} brojeva je {1}", Max,
									Zbroj);
	}
}
```

Primjer 4.7

Ispis:
Zbroj prvih 1000 brojeva je 500500

Program će vrtjeti petlju 1000 puta od *i=1* do *i=1000* i pri svakom prolazu, varijabli *Zbroj* bit će pridodana vrijednost varijable *i* koja je u stvari brojač. Varijabla *Zbroj* zbog toga mora prije izvršenja petlje biti inicijalizirana na vrijednost 0.

Sva tri dijela u *for* naredbi mogu sadržavati više izraza koji se onda odjeljuju zarezom.
Npr.

```
for(int i=1, j=2; i <= Max; i++, j+=2)
```

Kao i kod *if* naredbe ako se u bloku nalazi samo jedna naredba, nije je potrebno ograđivati vitičastim zagradama, a u bloku može biti i nijedna naredba:

```
for(int i=1, j=2; i <= Max; i++, j+=2)
	;
```

iako je ';' naredba koja ne radi ništa, ali je ipak naredba.

Svaki od 3 dijela *for* naredbe može se izostaviti ako takvo što zahtijeva algoritam programa.

```
for(;;)
  ;
```

Ovo je potpuno ispravna *for* naredba koja će doduše ući u beskonačnu petlju i tako blokirati program, dok će prevoditelj javiti upozorenje, ali ne i grešku.

Kao uvjet u *for* naredbi može se staviti i cijeli izraz s logičkim operatorima:

```
using System;

class Test
{
	static void Main()
	{
		string s = "HyperTextMarkupLanguage";
		for(int i = 0;i < s.Length && s[i] != 'M'; i++)
			Console.Write(s[i]);
	}
}
```

Primjer 4.8

Ispis:
HyperText

Budući da je u uvjetu logički operator AND, *for* petlja će se prestati izvršavati kad se dogodi prvo od ovo dvoje:

- brojač *i* je dosegnuo vrijednost dužine stringa (za dužinu stringa koristi se već spomenuta funkcija *Length* iz klase *string*)
- na *i*-tom mjestu u stringu nalazi se znak 'M' (pristup pojedinim znakovima stringa dobije se korištenjem operatora [], naravno koristeći 0-indeksaciju)

Ako string s definiramo kao:

```
string s = "CascadingStyleSheets";
```

ispis će biti *CascadingStyleSheets* jer će *for* petlja prestati s izvršavanjem zbog ostvarenja prvog uvjeta (bolje reći neostvarenja).

For petlja može sadržavati u bloku drugu *for* petlju što znači da će se za svaki prolaz prve petlje izvršiti cijela druga petlja.
Pogledajmo primjer koji treba izračunati broj prim brojeva manjih od 1000. Ako ne znate, prim broj je broj koji nije djeljiv niti s jednim brojem osim s 1 i samim sobom.

```
using System;

class Test
{
	static void Main()
	{
		bool FlagPrim = true;
		const int Max = 1000;
		int BrojPrimova = 0;

		for(int i = 1;i<Max;i++)
		{
			for(int j=2;j<i-1;j++)
				if(i%j == 0)
				{
					FlagPrim = false;
					break;
				}
			if(FlagPrim)
				BrojPrimova++;
			FlagPrim = true;
		}
		Console.WriteLine("Broj prim brojeva do {0} je {1}", Max,
                                                        BrojPrimova);
	}
}
```

Primjer 4.9

Ispis:
Broj prim brojeva manjih od 1000 je 169

Za svaki broj manji od 1000 provjerava se pomoću druge *for* petlje je li broj djeljiv s nekim brojem osim s 1 i sa samim sobom. Zato brojač u drugoj *for* petlji ide od 2 do i-1. Čim se naiđe na neki broj s kojim je broj djeljiv, *for* petlja prestaje s radom koristeći naredbu *break,* te se varijabla *FlagPrim* postavi na *false* kao znak da se *BrojPrimova* ne treba povećati za 1. Ako *FlagPrim* ostane *true*, to je znak da je broj prim broj pa se *BrojPrimova* poveća za 1.

Kad bismo trebali izračunati broj prim brojeva manjih od npr. milijun sve što treba promijeniti u programu je definicija varijable *Max*.

4.4. Naredba while

Naredba *while* je vrlo slična naredbi *for*, sve što se može napraviti s *for* petljom može se napraviti i s *while* petljom i obrnuto.

Sintaksa *while* naredbe:

```
while(provjera uvjeta)
{
	// Niz naredbi . . .
}
```

Vidimo da *while* naredba ima samo jedan dio u zagradama, a to je provjera uvjeta. To znači da se onaj prvi i treći dio iz *for* petlje izvršava negdje drugo.

Sljedeći primjer ispisat će dati string tako da za svaki znak iz stringa ispiše onaj sljedeći po abecedi:

```
using System;

class Test
{
	static void Main()
	{
		string s = "HAL";
		int i = 0;

		while(i++ < s.Length)
			Console.Write((char)(s[i-1] + 1));
	}
}
```

Primjer 4.10

Ispis:
IBM

Početna inicijalizacija varijable navedena je neposredno prije petlje s naredbom `int i = 0`, a treći dio iz *for* petlje direktno je ugrađen u uvjet s naredbom `i++`.
Argument u funkciji *Write()* pretvoren je s *cast* operatorom `(char)` u tip *char*, ali je prije toga uvećan za jedan tako da bude ispisan znak sljedeći po abecedi.

(Iako autori Odiseje kažu da je to čista slučajnost, slavno i do sada nedostižno računalo HAL mora biti uvećano za 1 da bi se dobio IBM.)

Kao argument u *while* naredbi može biti i poziv funkcije.

```
using System;
using System.Threading;

public class Test
{
	public static void Main()
```

```
	{
		Random NekiBroj = new Random();

		while(NekiBroj.Next(100) != 50)
			;
	}
}
```

Primjer 4.11

U ovom primjeru *while* petlja će prestati s radom, a zajedno s njom i cijeli program, kad *Next* funkcija iz klase *Random* koja generira slučajni broj manji od 100, generira broj 50.

Pitanje: Kako biste ispisali sve generirane slučajne brojeve u gornjem primjeru?
Ako mislite ovako:

```
while(NekiBroj.Next(100) != 50)
	Console.WriteLine(NekiBroj.Next(100));
```

odgovor nije točan, jer bi ovo ispisalo svaki drugi generirani broj, budući da u funkciji *WriteLine()* navođenje `NekiBroj.Next(100)` znači i generiranje novog broja.

Rješenje je ovako:

```
using System;
using System.Threading;

public class Test
{
	public static void Main()
	{
		int x;
		Random NekiBroj = new Random();
		while((x = NekiBroj.Next(100)) != 50)
			Console.WriteLine(x);
	}
}
```

Primjer 4.12

Ispis:
Ispisat će se svi generirani brojevi dok se ne generira broj 50.

Sad je generirani broj u *while* naredbi spremljen u varijabli *x*, uspoređuje se je li različit od 50, te se ispisuje s *WriteLine()*.

4.5. Naredba do while

Pretpostavimo prethodni primjer uz uvjet da kod prvog ulaska u *while* petlju treba napraviti neki vrlo važan posao.

```
using System;
using System.Threading;

public class Test
{
	public static void Main()
	{
		bool FlagImportant = false;
```

```
            Random NekiBroj = new Random();

            while(NekiBroj.Next(100) != 50)
            {
                  if(!FlagImportant)
                  {
                        FlagImportant = true;
                        // . . . Vrlo važan kôd
                  }
                  // . . .
            }
      }
}
```

Primjer 4.13

Ponovo će funkcija *Next* generirati neki broj i sve dok taj broj nije jednak 50, petlja će se izvršavati. Vrlo važan kôd će se izvršiti samo kod prvog prolaza petlje što je osigurano setiranjem *bool* varijable *FlagImportant*.

Naizgled sve OK, međutim što ako funkcija *Next* odmah u prvom pozivu generira broj 50? Uvjet za izvršenje *while* petlje ne bi bio ispunjen i blok petlje zajedno s vrlo važnim kôdom ne bi se uopće izvršio.

Rješenje je *do while* petlja koja osigurava barem jedan prolaz petlje.

Sintaksa *do while* naredbe:

```
do
{
      // Niz naredbi . . .
} while(provjera uvjeta);
```

gdje je *while* dio isti kao i u *while* naredbi. Primijetite jedino da se dodaje ';' na kraj *do while* naredbe.

```
using System;
using System.Threading;

public class Test
{
      public static void Main()
      {
            bool FlagImportant = false;
            Random NekiBroj = new Random();

            do
            {
                  if(!FlagImportant)
                  {
                        FlagImportant = !FlagImportant;
                        // . . . Vrlo važan kôd
                  }
                  // . . .
            } while(NekiBroj.Next(100) != 50);
      }
}
```

Primjer 4.14

Sad je osigurano da će se vrlo važan kôd izvršiti jer će se blok petlje izvršiti najmanje jednom. Vrlo važan kôd će se izvršiti i najviše jednom jer će kod izvršenja vrlo važnog kôda varijabla *FlagImportant* postati *true* i onemogućiti ponovno izvršavanje tog kôda.

4.6. Naredba break

Naredbu *break* već smo susreli kod *switch* naredbe, a njena također velika primjena je u *for* i *while* petljama u kojima se koristi za trenutačni i bezuvjetni prestanak izvršenja petlje.

Pogledajmo primjer koji treba izračunati najveći broj čiji je faktorijel manji od milijun. Faktorijel je broj koji se dobije množenjem nekog broja sa svim brojevima manjim ili jednakim od tog broja, a većim od 0. Npr. faktorijel od 4 = 1 * 2 * 3 * 4 = 24.

```
using System;

public class Test
{
	public static void Main()
	{
		int i;
		long Faktorijel = 1;

		for(i = 1;i<100;i++)
		{
			if(Faktorijel > 1000000)
			{
				Faktorijel /= (i-1);
				break;
			}
			Faktorijel *= i;
		}
		Console.WriteLine("{0}! = {1}", i-2, Faktorijel);
	}
}
```

Primjer 4.15

Ispis:
9! = 362880

Koristimo *for* petlju u kojoj svaki put provjeravamo je li faktorijel premašio milijun. Onog trenutka kad premaši, izlazimo iz petlje naredbom *break*, ali prije toga moramo faktorijel smanjiti za prethodno pomnoženi broj koji je faktorijel povećao iznad milijuna.

Kod naredbe *break* treba obratiti pozornost kod dvostrukih *for petlji* jer *break* omogućuje izlazak samo iz petlje u kojoj je pozvan.

```
using System;

public class Test
{
	public static void Main()
	{
		int i, j;
		for(i = 0;i<100;i++)
		{
			for(j = 0;j<100;j++)
			{
```

```
                        if(i + j == 10)
                              break;
                  }
            }
      }
}
```

Primjer 4.16

U ovom primjeru *break* trenutačno prekida samo unutrašnju, ali ne i vanjsku petlju, što znači da će se nakon prekida izvršenja unutrašnje petlje vanjska nastaviti izvršavati te opet pozivati unutrašnju.

4.7. Naredba continue

Naredba *continue* također se koristi u bloku *for* i *while* petlje i uvjetuje nastavak izvršenja petlje bez izvršenja naredbi koje se nalaze u bloku.

```
using System;

class Test
{
      public static void Main()
      {
            for(int i = 1; i <= 100; i += 2)
            {
                  if(i < 90)
                        continue;
                  Console.Write(i + " ");
            }
      }
}
```

Primjer 4.17

Ispis:
91 93 95 97 99

Za sve brojeve manje od 90 neće se izvršiti naredba *Console.Write()*, ali petlja nastavlja s radom sve dok je uvjet zadovoljen (`i <= 100`).
Obratite pozornost da se ne izvršavaju jedino naredbe koje se nalaze iza naredbe *continue*. Sve one naredbe u bloku koje se nalaze ispred naredbe *continue* bi se izvršile, kao što se uostalom naredba `if(i < 90)` izvršava svih 100 puta.

4.8. Naredba goto

Goto je naredba koja omogućuje skok na bilo koju naredbu u programu i upravo zbog toga je najosporavanija naredba u svim programskim jezicima. Naime, korištenje *goto* naredbe narušava strukturu programa i stvara tzv. *špageti kôd*.
Iako je dokazano da se svaki programski algoritam može riješiti bez korištenja ove naredbe, C# kao i svi ostali programski jezici ima tu naredbu.

Kod naredbe *break* vidjeli smo primjer s ugnježđenim *for petljama* kod kojih *break* u unutrašnjoj *for petlji* ne izaziva prekid vanjske petlje.

```
using System;
```

```
public class Test
{
	public static void Main()
	{
		int i, j=0;
		for(i = 0;i<100;i++)
		{
			for(j = 0;j<100;j++)
			{
				if(i + j == 10)
					break;
			}
		}
		Console.WriteLine("{0} {1}", i, j);
	}
}
```

Primjer 4.18

Ispis:
100 100

Korištenjem *goto* naredbe rezultat je drugačiji.

```
using System;

public class Test
{
	public static void Main()
	{
		int i, j=0;
		for(i = 0;i<100;i++)
		{
			for(j = 0;j<100;j++)
			{
				if(i + j == 10)
					goto kraj;
			}
		}
		kraj:
		Console.WriteLine("{0} {1}", i, j);
	}
}
```

Primjer 4.19

Ispis:
0 10

Možemo primijetiti da se naredba *goto* sastoji od ključne riječi *goto* i labele koja označuje naredbu na koju se skače.

Tek da bismo pokazali da se ovo isto može dobiti i bez *goto* naredbe, evo takvog primjera.

```
using System;

public class Test
{
	public static void Main()
	{
		int i, j=0;
		bool Flag = false;
```

```
            for(i = 0;i<100 && !Flag;i++)
            {
                  for(j = 0;j<100;j++)
                  {
                        if(i + j == 10)
                        {
                              Flag = true;
                              break;
                        }
                  }
            }
            Console.WriteLine("{0} {1}", --i, j); // i će biti povećan
                  // nakon break, pa ga treba smanjiti za 1
      }
}
```

Primjer 4.20

Ispis:
0 10

4.9. Naredba foreach

Foreach je posebna vrsta petlje koja omogućava iteraciju kroz vrijednosti sadržane u objektu koji implementira *IEnumerable sučelje (interface)*. Za sada je dovoljno reći da je taj objekt niz podataka, a o sučeljima će kasnije biti više govora.

Sintaksa *foreach* naredbe:

```
foreach(tip identifikator in objekt)
{
      // Niz naredbi . . .
}
```

Primjer korištenja *foreach* petlje:

```
using System;

class Test
{
      static void Main()
      {
            int[] Arr = new int[] {1, 3, 5, 7, 9};

            foreach(int i in Arr)
                  Console.Write(i + " ");
      }
}
```

Primjer 4.21

Ispis:
1 3 5 7 9

Koristeći *foreach* naredbu, svaki integer u nizu *Arr* se čita i ispisuje.
Može se koristiti i neki drugi tip, npr. *string*:

```
using System;

class Test
```

```
{
	static void Main()
	{
		string [] Arr = new string[] {"To","be","or","not","to","be"};

		foreach(string s in Arr)
			Console.Write(s + " ");
	}
}
```

Primjer 4.22

Ispis:
To be or not to be

Ako želimo koristiti niz s različitim tipovima, moramo koristiti tip *object* za koji smo rekli da je tip koji svi ostali tipovi nasljeđuju.

```
using System;

class Test
{
	static void Main()
	{
		object[] Arr = new object[] {2001, " Oddisey", 9.5, " weeks"};

		foreach(object o in Arr)
		{
			Console.Write(o);
			if(o.GetType() == typeof(string))
				Console.WriteLine();
		}
	}
}
```

Primjer 4.23

Ispis:
2001 Oddisey
9.5 weeks

Funkcija *GetType()* koristi se da bi se pročitao tip objekta u nizu. Ako je objekt *string*, dodaje se novi red tako da bi ispis filmova bio ispravan.

Vidimo da u *foreach* naredbi nije potrebno navesti veličinu niza kao što bi bilo da se koristi *for* naredba.
Varijabla u koju se sprema pojedina vrijednost iz niza je *read only* (samo za čitanje), pa je ovako nešto nedozvoljeno:

```
foreach(int i in Arr)
	i++;
```

4.10. Komentari

Komentari su dijelovi programa koji se ne prevode u izvršni oblik, odnosno služe samo za pojašnjenje dijelova kôda.
C# podržava dvije vrste komentara:

- jednolinijski od znaka // do kraja reda

- višelinijski od znakova /* do znakova */

```
using System;

/* Višelinijski
   komentar */
public class Test
{
      public static void Main()
      {
            int i, x=0;
            for(i = 1;i <= 100;i++)
                  x += i;     // Jednolinijski komentar
      }
}
```

Primjer 4.24

Komentar treba biti kratak i jasan te treba pojasniti dijelove koda. Nikako ne treba kao komentar navoditi ono što je jasno iz naredbe programskog jezika.

Primjer lošeg komentara:

```
x++;  // x se povećava za 1
```

Primjer dobrog komentara:

```
if(Year % 4 == 0)  // Je li godina prijestupna
```

4.11. Pretprocesorske direktive

Pretprocesorske direktive koriste se kao uputa prevoditelju kako prevesti program i ne predstavljaju dio samog procesa prevođenja programa. Postoji desetak pretprocesorskih direktiva koje sve počinju znakom '#' (Hash).
Najprije ćemo napraviti jedan mali primjer, pa ćemo na njemu objasniti kako pretprocesorske direktive rade.

```
#define NAPON
#define STRUJA

using System;

public class Test
{
      public static void Main()
      {
            #if(NAPON && STRUJA)
                  Console.WriteLine("Otpor se može izračunati");
            #elif(NAPON && !STRUJA)
                  Console.WriteLine("Otpor se ne može izračunati");
            #elif(!NAPON && STRUJA)
                  Console.WriteLine("Otpor se ne može izračunati");
            #elif(!NAPON && !STRUJA)
                  Console.WriteLine("Otpor se ne može izračunati");
            #endif
      }
}
```

Primjer 4.25

Ispis:
Otpor se može izračunati

Pretprocesorske direktive navode se s prefiksom '#' i na početku su definirane dvije definirajuće direktive *#define* kojima se definiraju simboli NAPON i STRUJA. U funkciji *Main* se zatim direktivama *#if* i *#elif* provjerava jesu li simboli definirani ili ne. Princip rada ovakvih *#if* i *#elif* direktiva vrlo je sličan kao i kod standardne *if else* naredbe.
Međutim velika je razlika u tome što se ovakva provjera događa prije prevođenja programa. Zbog toga smo npr. mogli na kraju dodati još jedan ovako definiran *#elif*

```
#elif(OTPOR)
      R = U/I;
```

i sve bi se prevelo i izvršilo bez grešaka, iako nije napisano po pravilima programskog jezika. Razlog je što bi prevoditelj zbog toga što simbol OTPOR nije definiran, jednostavno ignorirao ovaj kôd.
Razlika je i kod direktive *#endif* koja označava kraj *#if #elif* bloka jer takve naredbe u običnom *if else* nema.

Suprotna direktiva od *#define* je *#undef* koja će poništiti definiciju (u engleskom jeziku puno je lakše imenovati ovakve stvari, oni jednostavno stave predmetak *un* i promjene značenje u suprotno). Sintaksa je ista kao i kod *#define*

```
#undef NAPON
```

Direktive *#warning* i *#error*
Direktivama se mogu i generirati upozorenja ili greške prevoditelja. Za tu svrhu koriste se direktive *#warning* i *#error* koje kao argument primaju tekst koji će javiti prevoditelj.

```
#if(DEBUG)
      #warning Debug je definiran
#endif
```

Ako je DEBUG definiran, prevoditelj će javiti upozorenje, dok bi isti primjer s direktivom *#error* javio grešku i program se ne bi preveo, makar bio sintaktički potpuno ispravan.

Direktive *#region* i *#endregion*
C# je uveo i neke nove direktive u odnosu na C i C++, a to su *#region* i *#endregion*. One omogućavaju označavanje bloka u programskom kôdu, kao npr. cijele jedne klase ili svih deklaracija varijabli u jednoj klasi. To onda omogućuje editorima kao što je onaj iz Visual Studia skrivanje i otkrivanje cijelog bloka naredbi. Slično kao što Windows Explorer radi s rastvaranjem mapa.

Poglavlje 5:
Klase i objekti

Ovo poglavlje obrađuje:

- pojam objekata i objektno orijentiranog programiranja
- pojam objekta kao tipa neke klase
- konstruktore i poziv konstruktora
- vidljivost članica klase
- razliku između statičkih i nestatičkih članova
- korištenje svojstava za pristup privatnim varijablama
- nasljeđivanje klasa
- apstraktne i zapečaćene klase
- sučelja i njihovu implementaciju kroz izvedene klase
- operatore *is* i *as*
- strukture
- ugnježđenje klasa
- organiziranje klasa u imenike

5.1. Klase i objekti

O objektno orijentiranom programiranju (OOP) intenzivno se počelo razmišljati početkom 80-ih godina kada se analizom utvrdilo da je izrada softvera "u krizi". S obzirom na veliki razvoj hardvera, ali i potrebu za sve složenijim programima, bilo je očito da proceduralni jezici kakvi su se do tada koristili ne mogu izaći na kraj s rastućim potrebama.
Objektno orijentirano programiranje uvelo je klase i objekte kao ključne pojmove u programiranju, za razliku od proceduralnog programiranja gdje je ključni pojam bila funkcija.

Klasa je korisnički tip podataka koji se sastoji od vrijednosnih i referentnih varijabli te metoda (funkcija) za rad s tim varijablama. Klase definira programer prema potrebama programa, a na raspolaganju mu je i veliki broj predefiniranih klasa iz FCL-a.
Klasa treba predstavljati neku "crnu kutiju" koja mora imati sposobnost da preko svojih metoda **sama** sobom upravlja.

Uglavnom, klasa treba biti cjelina čija eventualna promjena ne bi smjela imati puno utjecaja na ostale djelove programa što kao rezultat daje modularnost programa, a to opet povlači za sobom lakše održavanje programa.
U proceduralnim jezicima svaka je promjena u programu bila potencijalna bomba jer je neovisnost između dijelova programa bila mala.

Pogledajmo kako bi izgledala definicija klase *Televizor*, uz primjedbu da ovaj primjer više shvatimo kao način općenitog definiranja klase jer C# uvodi neke novine koje ćemo kasnije objasniti.

```
class Televizor
{
      private bool Ukljucen;
      private bool HDTVReady;
      private sbyte AktivniProgram;
      private sbyte Dijagonala;
      private int GodinaProizvodnje;
      private string SerijskiBroj;
      private string Tip;

      public Televizor(sbyte Diameter, int Year, bool Hdtv, string tip)
      {
            HDTVReady = Hdtv;
            Dijagonala = Diameter;
            GodinaProizvodnje = Year;
            Tip = tip;
      }

      public void PromijeniProgram(sbyte x)
      {
            AktivniProgram = x;
      }

      public void UkljuciTV()
      {
            Ukljucen = true;
      }

      public void IskljuciTV()
      {
            Ukljucen = false;
```

```
        }
}
```

Primijetite da na kraju definicije klase ne dolazi ';' kao u C++, iako se prevoditelj neće buniti ako se i doda ';' na kraj.
Klasa se definira ključnom riječi *class* nakon koje dolazi ime klase te vitičaste zagrade. U klasi *Televizor* imamo nekoliko definiranih varijabli od kojih svaka predstavlja neku osobinu televizora te 4 metode koje dodjeljuju vrijednosti tim varijablama.
Ključna riječ *private* označuje dostupnost člana klase o čemu će biti više rečeno u sljedećim poglavljima.
Ono što je najbitnije jest da te varijable ne bi trebao nitko drugi mijenjati osim samih metoda klase. Odnosno klasa mora **sama** imati sposobnost promijeniti vrijednosti svojih varijabli.

Uostalom, zar nije tako i u stvarnosti?
Tko mijenja aktivni program na TV? Gledatelj ili televizor. Odgovor je naravno da ga mijenja televizor, gledatelj samo daje naredbu televizoru preko daljinskog upravljača, a televizor sam mijenja aktivni program.

Objekt je varijabla tipa neke klase i kao što klasa treba predstavljati skupinu nečega što ima nešto zajedničko, tako i objekt kao instanca te klase treba predstavljati neki konkretan primjerak te skupine odnosno klase.

Tako bi dobar primjer objekta tipa gore definirane klase bio npr. neki konkretan tip televizora s točno određenim karakteristikama kao što su dijagonala, je li HDTV Ready, koje godine je proizveden i slično.

```
Televizor SharpC = new Televizor(29, 2006, true, "LCD");
```

Definirali smo objekt *SharpC* tipa *Televizor* sljedećim osobinama:

- Televizor ima dijagonalu 29 inča
- Godina proizvodnje je 2006.
- Televizor je HDTV Ready
- Tip televizora je LCD

Neki drugi televizor (objekt) definirao bi se ovako:

```
Televizor Philips2006 = new Televizor(25, 2005, false, "Plazma");
```

Objekt je referentna varijabla i definira se korištenjem ključne riječi *new* i posebne metode koja se zove *konstruktor* (constructor) koja mora imati isto ime kao i klasa.

Ako bismo željeli promijeniti na 2. program, pozvali bismo metodu *PromijeniProgram()*

```
SharpC.PromijeniProgram(2);
```

Metoda se poziva navođenjem imena objekta i metode odvojene točkom. To je tzv. *dot* operator.

Pozor:
Nemojte miješati objekt i object.

- objekt je hrvatski naziv za instancu klase (eng. object!)

- *object* je klasa i to je osnovna bazna klasa za sve tipove podataka

```
object object = "object";      // Neispravno, object je ključna riječ i ne
                               // može biti ime varijable
object objekt = "object";      // Ispravno, objekt nije ključna riječ
object Object = "object";      // Ispravno, Object (veliko O) nije ključna
                               // riječ
```

5.2. Konstruktori

Konstruktor je posebna metoda u klasi koja kreira objekt i ima isto ime kao i klasa. Konstruktor nije obvezno definirati u klasi i ako se ne definira, objekt će se kreirati sa standardnim konstruktorom koji se poziva bez argumenata.

```
Televizor = new Televizor();
```

Prava vrijednost i snaga upotrebe konstruktora vidi se u mogućnosti da je u klasi moguće definirati više različitih konstruktora koji će se razlikovati po broju proslijeđenih argumenata. Pogledajmo primjer klase *Planet* koja će imati nekoliko varijabli koje obilježavaju pojedini planet kao npr. ima li života, ima li atmosferu, brzine vrtnje itd., te tri različita konstruktora koji će kreirati objekte.

```
using System;

public class Planet
{
      private bool Life;
      private bool Atmosfera;
      private sbyte BrojSatelita;
      private int Promjer;
      private float BrzinaOkoOsi;
      private float BrzinaOkoSunca;
      private string NazivPlaneta;

      public Planet(string name, sbyte brojsat)
      {
            NazivPlaneta = name;
            BrojSatelita = brojsat;
      }

      public Planet(string name, bool life, bool atm)
      {
            NazivPlaneta = name;
            Life = life;
            Atmosfera = atm;
      }

      public Planet(string name, bool life, bool atm, sbyte brojsat,
                              int promjer, float brzina1, float brzina2)
      {
            NazivPlaneta = name;
            Life = life;
            Atmosfera = atm;
            BrojSatelita = brojsat;
            Promjer = promjer;
            BrzinaOkoOsi = brzina1;
            BrzinaOkoSunca = brzina2;
      }
```

```
}

public class Test
{
      static void Main()
      {
            Planet Earth = new Planet("Zemlja", true, true, 1, 6370,
                                                      0.463f, 29.7f);

            Planet Mercury = new Planet("Merkur", false, false);
            Planet Mars = new Planet("Mars", 2);
      }
}
```

Primjer 5.1

U primjeru imamo tri različita konstruktora:

1. konstruktor koji dodijeljuje vrijednosti za svih 7 varijabli (*objekt Earth*)
2. konstruktor koji definira samo ima li života i atmosfere na planetu (*objekt Mercury*)
3. konstruktor koji definira samo broj satelita (*objekt Mars*)

Iako drugi i treći konstruktor ne definiraju sve varijable, one su ipak definirane jer će im biti dodijeljene podrazumijevane (default) vrijednosti za određeni tip varijable.

Napomena: ako je definiran konstruktor s parametrima, ne može se koristiti standardni konstruktor bez parametara, osim ako ga sami ne definiramo.

Tablica podrazumijevanih vrijednosti:

Tip	Podrazumijevana vrijednost
`sbyte, byte, short, ushort, int, uint, long, i ulong`	`0`
`bool`	`false`
`char`	`'\0'`
`enum`	`0`
`reference`	`null`

Kod *reference* tipa *null* znači da ne upućuje ni na jedan objekt.

Varijable klase moguće je definirati i kod deklaracije s tzv. *inicijalizerom* (*initializer*). Da smo kod deklaracije varijable *Life* napisali npr.

```
bool Life = true;
```

onda bi za objekt *Mars* vrijedilo da na njemu ima života, odnosno varijabla *Life* objekta *Mars* imala bi vrijednost *true*.
To bi se naravno promijenilo kad bismo za definiciju objekta *Mars* koristili konstruktor koji dodjeljuje vrijednost varijabli *Life* na način:

```
Planet Mars = new Planet(false, false);
```

Dakle, konstruktor ima prednost u odnosu na inicijalizer, pa bi *false* iz konstruktora predefinirao *true* iz inicijalizera.

Destruktor

Destruktor je metoda koja uništava objekte i oslobađa memoriju. Za razliku od C++, u C# nema potrebe za definiranjem i pozivom *destruktora* jer je on u C# pozvan automatski prilikom izvršenja postupka čišćenja memorije (*garbage collection*) koji se također događa automatski.

Ipak, destruktor se može definirati na sljedeći način:

```
~ime_klase()
{
      . . .
}
```

Ako bismo u klasi *Planet* definirali destruktor:

```
~Planet()
{
      Console.WriteLine("Objekt " + NazivPlaneta + " je uništen");
}
```

onda bi rezultat izvršenja programa bio:

Objekt Mars je uništen
Objekt Zemlja je uništen (nadajmo se da destruktor u ovom slučaju ipak ne bi bio pozvan!)
Objekt Merkur je uništen

Destruktor će pak pozvati funkciju *System.Finalize* koja je ovako definirana:

```
protected override void Finalize()
{
      try
      {
            // . . .
      }
      finally
      {
            base.Finalize();
      }
}
```

gdje bi se u *try* dijelu pozvala ona funkcija *Console.WriteLine* iz destruktora.
Možemo vidjeti također da se objekti ne moraju uništavati u redoslijedu kako su i kreirani jer *garbage collection* sam odlučuje o tome.

5.3. Dostupnost klase i članova

Već smo naveli da dobro projektirana klasa treba (a često i mora) sama moći upravljati svojim podacima. To bi značilo da bi klasa morala imati neki instrument zaštite od neovlaštenog pristupa podacima od strane drugih klasa ili objekata. Takva zaštita omogućena je *modifikatorima pristupa (access modifiers).*

Postoji pet različitih modifikatora pristupa:

Modifikator	Ograničenje
`private`	Pristup članovima klase X omogućen je samo

	metodama klase X
`protected`	Pristup članovima klase X omogućen je samo metodama klase X te metodama klase Y koja nasljeđuje klasu X
`public`	Pristup članovima klase X omogućen je svim klasama bez ograničenja
`internal`	Pristup članovima klase X omogućen je samo metodama klasa koje se nalaze u istom skupu (*assembly*)
`protected internal`	Pristup članovima klase X omogućen je samo metodama klase X, metodama klase Y koja nasljeđuje klasu X te metodama klasa koje se nalaze u istom skupu (assembly). *Skup je sastavni dio svake .Net aplikacije, a sastoji se od manifesta koji opisuje sve što CLR treba znati o skupu, modula odnosno izvršnih datoteka te resursa odnosno neizvršnih datoteka.*

Pogledajmo sada malo jednostavnije definiranu klasu *Planet* koja ima samo jedan konstruktor:

```
using System;

public class Planet
{
	private bool Life;
	public bool Atmosfera;
	private string NazivPlaneta;

	public Planet(string name, bool life, bool atm)
	{
		NazivPlaneta = name;
		Life = life;
		Atmosfera = atm;
	}

	public void PromijeniLife(bool life)
	{
		Life = life;
	}

}

public class Test
{
	static void Main()
	{
		Planet Uranus = new Planet("Uran", false, true);
	}
}
```

Primjer 5.2

Kad bismo sada pokušali dodijeliti varijabli *Life* vrijednost *true* s izrazom
`Uranus.Life = true;`
prevoditelj bi javio grešku o nedozvoljenom pristupu varijabli *Life*,
dok bi

```
Uranus.Atmosfera = true;
```
prošlo bez greške.

Razlog je što je član klase *Life* označen modifikatorom *private* što znači da samo metode te klase imaju pristup članovima klase, dok je član *Atmosfera* označen kao *public* pa je dostupan bez ograničenja.
Naravno, netko mora imati pristup privatnim članovima, i taj netko su metode članice klase. Ovakav poziv metode će definirati (ne nužno i promijeniti) vrijednost člana *Life*:

```
Uranus.PromijeniLife(true);
```

Očito je da se to slaže s tvrdnjom da klasa sama mora upravljati svojim podacima jer uočite razliku kad napišemo:

```
Uranus.Life = true;
```
i
```
Uranus.PromijeniLife(true);
```

U prvom slučaju objekt pokušava direktno dodijeliti vrijednost članu klase (zapravo samom sebi!) i ne uspijeva jer je on (objekt *Uranus*) član klase *Test*, dakle iako tipa *Planet*, on je za klasu *Planet* vanjski član. U drugom pak slučaju sama klasa odnosno njena članica pokušava to isto i zato uspijeva.

Da bi to bilo moguće odnosno da bi sama metoda bila dostupna vanjskom objektu, metoda treba biti deklarirana kao *public*. Inače, da je deklarirana kao

```
private void PromijeniLife(bool life);
```

tada ni metoda ne bi bila dostupna vanjskom objektu. Ali metode su uglavnom deklarirane bez ograničenja pristupa, dakle kao *public*.
Podrazumijevani (*default*) modifikator pristupa je *private*, što znači da je ovo isto

```
int Life;
private int Life;
```

U oba slučaja varijabla *Life* bit će deklarirana kao *private*, ali ipak radi preglednosti i čitljivosti savjetuje se koristiti drugi način.

5.4. Statički članovi

Članovi klase mogu biti statički ili nestatički.
Razlika je što statički članovi pripadaju klasi i za njihovo pozivanje nije potrebno deklarirati instancu odnosno objekt tipa klase, dok nestatički članovi pripadaju objektu što automatski znači da je za njihovo korištenje potrebno kreirati objekt.
To možemo vidjeti na sljedećem primjeru:

```
using System;

class Test
{
      private short XCoord;
      private static short YCoord;

      private void FCoord()
```

```
        {
                XCoord = 300;
                YCoord = 400;
        }

        static void Main()
        {
                Test Tmp = new Test();
                Tmp.XCoord = 100;       // OK
                Tmp.YCoord = 200;       // Greška, neće se prevesti

                Test.XCoord = 100;      // Greška, neće se prevesti
                Test.YCoord = 100;      // OK

                FCoord();               // Greška, neće se prevesti
        }
}
```

Primjer 5.3

Tri su linije koje se neće prevesti:

`Tmp.YCoord = 200;` // statički član pozivamo objektom

`Test.XCoord = 100;` // nestatički član pozivamo klasom

Također se neće prevesti linija

`FCoord();`

jer statička funkcija, a *Main* je statička funkcija, ne može pozvati nestatičke članove klase. Ako promijenimo deklaraciju metode *FCoord* u *static*, program se opet neće prevesti, ali sad zbog toga što se u funkciji *FCoord* poziva

`XCoord = 300;`

dakle opet bi statička metoda pozivala nestatički član.

Budući da je statički član vezan za klasu, on je zajednički između svih objekata odnosno instanci te klase. To znači da takav jedan član možemo iskoristiti npr. kao brojač svih kreiranih instanci te klase. Pogledajmo to na sljedećem primjeru:

```
using System;

class TempClass
{
        private static int Counter;

        public TempClass()
        {
                Counter++;
        }

        public int BrojInstanci()
        {
                return Counter;
        }
}

class Test
{
        static void Main()
        {
                TempClass x = new TempClass();
                Console.WriteLine(x.BrojInstanci());
```

```
            TempClass y = new TempClass();
            Console.WriteLine(y.BrojInstanci());

            TempClass z = new TempClass();
            Console.WriteLine(z.BrojInstanci());
      }
}
```

Primjer 5.4

Ispis:
1
2
3

Kod kreiranja objekta poziva se konstruktor bez parametara koji povećava vrijednost varijable *Counter* za 1. Tako će kod svakog kreiranja objekta klase *TempClass* ova varijabla biti povećana za 1.
Zato, iako će objekt *z* povećati vrijednost varijable *Counter* samo za 1, njena vrijednost će nakon kreiranja objekta *z* biti 3 jer su objekti *x* i *y* povećali tu istu varijablu. Za dobivanje vrijednosti varijable *Counter* moramo koristiti metodu *BrojInstanci* jer je *Counter* deklariran kao *private*.

Statički konstruktor
Vidjeli smo da i metode mogu biti statičke, a poseban je slučaj definiranje statičkog konstruktora za kojeg vrijede neke posebnosti:

- statički konstruktor deklarira se bez modifikatora pristupa
- statički konstruktor izvršava se automatski prije kreiranja prve instance klase

```
using System;

class TempClass
{
      public int X;

      static TempClass()
      {
            Console.WriteLine("Statički konstruktor");
      }

      public TempClass(int x)
      {
            X = x;
            Console.WriteLine("Kreiranje objekta");
      }
}

class Test
{
      static void Main()
      {
            Console.WriteLine("Prije kreiranja objekta");
            TempClass x = new TempClass(5);
            Console.WriteLine("Poslije kreiranja objekta");
      }
}
```

Primjer 5.5

Ispis:
Prije kreiranja objekta
Statički konstruktor
Kreiranje objekta
Poslije kreiranja objekta

Vidimo da se statički konstruktor izvršio prije nestatičkog konstruktora, dakle prije kreiranja objekta. Statički konstruktor izvršava se samo jednom, prije kreiranja prvog objekta jer da smo u funkciji *Main* definirali još nekoliko objekata tipa *TempClass*, statički konstruktor ne bi se ponovo izvršio.

5.5. Svojstva (Properties)

Prisjetimo se primjera iz poglavlja 5.3 gdje smo naveli kako nije moguće *private* varijabli *Life* iz klase *Planet* direktno dodijeliti vrijednost:

```
Uranus.Life = true;
```
nego smo to morali napraviti korištenjem metode
```
Uranus.PromijeniLife(true);
```

Bilo bi lijepo kada bismo mogli koristiti najbolje od obojeg:
1. varijabla ostaje *private*, odnosno nedostupna vanjskim klasama i objektima
2. varijabla se može promijeniti
3. varijabli se može pristupiti direktno

Toga su također bili svjesni i projektanti C#-a, pa su u klase ugradili tzv. svojstvo (*property*). *Svojstvo* je u klasi implementirano kao metoda, a sintaksa poziva je ista kao i kod direktnog pristupa varijabli.

Vratimo se na primjer klase *Planet*, ali ovaj put s implementiranim svojstvima:

```
using System;

public class Planet
{
	private bool life;
	private bool atmosfera;
	private string NazivPlaneta;

	public Planet(string name, bool lf, bool atm)
	{
		NazivPlaneta = name;
		life = lf;
		atmosfera = atm;
	}

	public bool Life
	{
		get
		{
			return life;
		}

		set
		{
```

```
                life = value;
            }
        }
}

public class Test
{
        static void Main()
        {
                Planet Uranus = new Planet("Uran", false, true);
                Uranus.Life = true;
                Console.WriteLine("Na Uranu {0} života", Uranus.Life == true ?
                                                        "ima" : "nema");
        }
}
```

Primjer 5.6

Ispis:
Na Uranu ima života

Svojstvo se sastoji od dvaju *pristupnika* (*accessor*) **get** i **set**.

Pristupnik *get* poziva se kao da je varijabla članica klase (`Uranus.Life == true`), a tijelo *get* pristupnika je isto kao što bi bilo da je to metoda koja vraća privatnu varijablu. Zbog toga pristupnik *get* mora biti deklariran kao i varijabla čiju vrijednost vraća. U ovom primjeru to je *bool* jer je tako deklarirana varijabla *life*.

Pristupnik *set* postavlja vrijednost privatne varijable (`Uranus.Life = true`) koristeći posebnu internu varijablu *value* koja sadrži vrijednost navedenu u pozivu pristupnika (u ovom slučaju to je *true*).
Pristupnik *set* implementiran je u klasi kao metoda koja je deklarirana kao metoda koja ne vraća nikakvu vrijednost (*void*) i koja u svom tijelu dodijeljuje vrijednost *private* varijabli.

Pristupnik *set* možemo koristiti i ovako:
`Uranus.BrojSatelita += 2;`
pod uvjetom da je definirano svojstvo *BrojSatelita* cjelobrojnog tipa.

```
public int BrojSatelita
{
        get { return brojSatelita; }
        set { if(brojSatelita == 0)
                    brojSatelita = value; }
}
```

Svojstvo ne mora imati oba pristupnika. Svojstvo sa samo *get* pristupnikom omogućuje samo čitanje vrijednosti varijable (*read only*), dok svojstvo sa samo *set* pristupnikom omogućuje samo dodjeljivanje vrijednosti varijabli (*write only)*.

Pristupnik *get* nikako ne bi trebao mijenjati vrijednost varijabli čiju vrijednost vraća. Npr. ovo je vrlo loše napisano, iako dozvoljeno

```
public int Counter
{
        get { return counter++; }
}
```

jer ruši logiku programiranja. Netko tko čita naredbu `x = Counter` ne očekuje da ta naredba promijeni vrijednost ijednoj drugoj varijabli osim *x*.

Sa svojstvima smo dobili najbolje od obojeg:

- varijabla ostaje *private*
- možemo je promijeniti i direktno joj pristupiti

Ali oprezni čitatelj će primijetiti:
od dvije stvari smo napravili treću te uzeli samo ono najbolje, sigurno u tom trećemu (svojstvu) mora biti i nešto malo manje dobro.
I to je točno, u svojstvima postoji i nešto što se ne može zvati idealnim.

- Kad netko čita program on iz naredbe `Uranus.Life = true` ne može znati je li *Life* svojstvo ili varijabla, iako bi trebao pretpostaviti da je to svojstvo jer klasa uglavnom treba imati privatne članove. Ali zašto bi uopće taj netko morao razmišljati o dobro projektiranoj klasi?
- Ako pogledamo nazive svojstva i varijabli, vidjet ćemo da se svojstvo zove *Life*, a varijabla koju svojstvo mijenja *life*. Znači da ne možemo zadržati Pascal notaciju na oba mjesta jer svojstvo i varijabla ne mogu imati ista imena.

Ipak, savjet je da zanemarite ove eventualne primjedbe i koristite svojstva za pristup privatnim varijablama.

Svojstvo može biti deklarirano i kao *static* i tada za njega vrijedi sve što i za ostale statičke članove. Jedno od najkorištenijih statičkih svojstava je *readonly* svojstvo *Now* iz klase *DateTime* koje vraća trenutni datum i vrijeme.

`Console.WriteLine(DateTime.Now);` će ispisati
25.11.2006 17:28:11
(*Neke američke knjige u ovakvim slučajevima upozoravaju čitatelja da će na njegovom računalu možda biti ispisana neka druga vrijednost. Vjerujemo da naše čitatelje nije potrebno na to upozoriti!)*

5.6. Nasljeđivanje (Inheritance)

Nasljeđivane (*Inheritance*) je jedan od temeljnih pojmova u objektno orijentiranom programiranju. To je mogućnost da jedna klasa (izvedena klasa) naslijedi drugu (bazna klasa) odnosno da naslijedi neke njene članove (varijable, metode, svojstva…) i to samo one koje bazna klasa dozvoli. Izvedena klasa tako može naslijediti samo one članove koji nisu označeni kao *private* jer, kako smo rekli, ti članovi pripadaju samo baznoj klasi. Također izvedena klasa ne nasljeđuje konstruktore bazne klase.
Izvedena klasa deklarira se dodatkom znaka '**:**' i naziva klase koja se nasljeđuje.
Za razliku od C++, u C# nema višestrukog nasljeđivanja odnosno klasa ne može naslijediti više od jedne klase, ali se to može zaobići korištenjem sučelja (*interface*) što će biti kasnije obrađeno.

Jedan od primjera osnovnih i izvedenih klasa koji se može vidjeti u svakodnevnom radu s računalom su kontrolni objekti kao što su *ListBox, ComboBox, Button, CheckBox, Radio Button* itd… Svi oni su objekti svoje klase koja nasljeđuje osnovnu klasu *Control*. Klasa *Control* će zato definirati sve osobine zajedničke za sve kontrolne objekte kao što su

dimenzije, boja, tekst, je li objekt vidljiv ili ne itd… Svaka od tih klasa također će definirati svoje članice pa će npr. klasa *ListBox* imati metodu za sortiranje ili pretraživanje liste, dok će klasa *CheckBox* implementirati metodu koja će vraćati *true* ili *false* ovisno je li kvadratić označen ili ne.

Pogledajmo jedan primjer izvedene klase:

```
using System;

class Bazna
{
      private int xCoord = 100;
      protected int yCoord = 200;

      public Bazna()
      {
            Console.Write("Bazni konstruktor, ");
            Console.WriteLine("x = {0}, y = {1}", xCoord, yCoord);
      }

      public int XCoord
      {
            get
            {
                  return xCoord;
            }
      }

      public void BaseMethod()
      {
            Console.WriteLine("Bazna metoda pozvana od {0}", GetType());
      }
}

class Izvedena : Bazna
{
      private int zCoord = 300;

      public Izvedena()
      {
            Console.Write("Izvedeni konstruktor, ");
            Console.WriteLine("x = {0}, y = {1}, z = {2}", XCoord, yCoord,
                                                            zCoord);
      }
}

class Test
{
      static void Main()
      {
            Bazna Base = new Bazna();
            Izvedena Derived = new Izvedena();
            Base.BaseMethod();
            Derived.BaseMethod();
      }
}
```

Primjer 5.7

Ispis:

```
Bazni konstruktor, x = 100, y = 200
```

Bazni konstruktor, x = 100, y = 200
Izvedeni konstruktor, x = 100, y = 200, z = 300
Bazna metoda pozvana od Bazna
Bazna metoda pozvana od Izvedena

Primjer definira dvije klase, klasu *Bazna* kao baznu klasu i klasu *Izvedena* kao izvedenu. Treba obratiti pozornost na nekoliko stvari.

Definirana su dva objekta *Base* i *Derived*, a vidimo da je bazni konstruktor pozvan dva puta. Naime izvedena klasa **automatski** prije poziva svog konstruktora poziva konstruktor bazne klase po sljedećim pravilima:

- ako nije definiran konstruktor u baznoj klasi, poziva se podrazumijevani (default) konstruktor bez argumenata
- ako je definirano jedan ili više konstruktora u baznoj klasi, a među njima je i onaj bez argumenata (dakle predefinirani default konstruktor), poziva se taj konstruktor
- ako je definirano jedan ili više konstruktora u baznoj klasi, a među njima **nije** i onaj bez argumenata, potrebno je u deklaraciji izvedene klase naglasiti ključnom riječi *base* koji se konstruktor automatski poziva
  `public Izvedena() : base(10)`

Naravno ako u deklaraciji izvedene klase navedemo poziv točno određenog konstruktora, taj se konstruktor i poziva.

Dalje, možemo uočiti da izvedena klasa nasljeđuje varijablu *y* jer je ona deklarirana kao *protected*, metodu *BaznaMetoda* i svojstvo *XCoord* te ih izvedena klasa može koristiti kao svoje članice. Metoda *BaznaMetoda* poziva se dva puta od objekta klase *Bazna* i objekta klase *Izvedena*, a metodom *GetType()* možemo saznati tip objekta.
Primijetite u konstruktoru *Izvedena()* da se za ispis vrijednosti varijable *x* koristi naslijeđeno svojstvo *XCoord* iz bazne klase.

Izvedena klasa može pristupiti članicama bazne klase navođenjem ključne riječi *base* kao npr.
`base.BaseMethod();`
Kad klasa nasljeđuje članove, to znači da oni i dalje pripadaju baznoj klasi, a izvedena ih samo koristi. Ako bi u gornjem primjeru konstruktor izvedene klase izgledao ovako,

```
public Izvedena()
{
      yCoord = 700;
      Console.WriteLine("{0}, {1}", yCoord, base.yCoord);
}
```

program bi ispisao:
700, 700
jer je riječ o jednoj te istoj varijabli *yCoord*.

Ako bi neka treća klasa naslijedila klasu *Izvedena*, ona bi naslijedila sve nasljedive članove iz klase *Izvedena*, ali i iz klase *Bazna*.

Poseban slučaj kod nasljeđivanja je tzv. *polimorfizam*. To je sposobnost metoda naslijeđene klase da redefiniraju istoimene metode iz bazne klase. Pogledajmo to na primjeru klase *Office* i iz nje izvedenih klasa *Word*, *Excel* i *Access*.

```
using System;

class Office
{
	public void PokreniProgram()
	{
		Console.WriteLine("Office je pokrenut");
	}

	public virtual void SpremiDatoteku()
	{
		Console.WriteLine("Spremi Office datoteku");
	}

	public void ZatvoriProgram()
	{
		Console.WriteLine("Office je zatvoren");
	}
}

class Word: Office
{
	public override void SpremiDatoteku()
	{
		Console.WriteLine("Spremi doc datoteku");
	}
}

class Excel: Office
{
	public override void SpremiDatoteku()
	{
		Console.WriteLine("Spremi xls datoteku");
	}
}

class Access: Office
{
}


class Test
{
	static void Main()
	{
		Office[] Documents = new Office[3];
		Documents[0] = new Word();
		Documents[1] = new Excel();
		Documents[2] = new Access();
		for(int i=0;i<3;i++)
			Documents[i].SpremiDatoteku();
	}
}
```

Primjer 5.8

Ispis:

Spremi doc datoteku
Spremi xls datoteku
Spremi Office datoteku

Imamo baznu klasu *Office* i iz nje izvedene klase *Word*, *Excel* i *Access*. Klasa *Office* definira 3 metode od kojih je metoda *SpremiDatoteku* označena ključnom riječi *virtual*. To znači da su izvedene klase slobodne, ali ne i obvezne definirati vlastite metode *SpremiDatoteku*.
Klase *Word* i *Excel* to i čine, definiraju svoje metode *SpremiDatoteku* koje se pri tom moraju označiti ključnom riječi *override*, a izvedena klasa *Access* ne definira svoju metodu *SpremiDatoteku* te koristi metodu iz bazne klase.

Ono što je najzanimljivije, događa se u funkciji *Main* gdje smo definirali niz tipa *Office* s tri člana tako da je svaki član niza objekt tipa jedne od izvedenih klasa.
Prilikom poziva metode u *for* petlji, u vremenu izvođenja (runtime) se određuje koja će se implementacija metode pozvati, je li ona iz izvedene ili iz bazne klase, ovisno o tome je li izvedena klasa predefinirala (*override*) metodu *SpremiDatoteku* ili ne.
Zato prvi i drugi član niza (*Word* i *Excel*) pozivaju metode iz izvedenih klasa, a treći član (*Access*) poziva metodu iz bazne klase.

Pretpostavimo sada da se u izvedenoj klasi *Access* definira metoda *SpremiKaoXML*.

```
public virtual void SpremiKaoXML()
{
      Console.WriteLine("Spremi mdb kao xml datoteku");
}
```

To je sve u redu dok nakon nekog vremena autori klase *Office* (na što autori klase *Access* pretpostavimo nemaju utjecaja) ne odluče također implementirati metodu *SpremiKaoXML*.

```
public virtual void SpremiKaoXML()
{
      Console.WriteLine("Spremi Office kao xml datoteku");
}
```

Nakon prevođenja tako izmijenjene bazne klase i izvedenih klasa prevoditelj ne bi javio grešku, ali bi upozorio da metoda u izvedenoj klasi *Access* skriva naslijeđenu metodu iz bazne *Office* klase. I naravno ovakav poziv:

```
Access Acc = new Access();
Acc.SpremiKaoXML();
```

pozvao bi metodu izvedene *Access* klase.
Želimo li prevođenje bez upozorenja o skrivanju metode, potrebno je metodu u izvedenoj klasi deklarirati ključnom riječi *new*

```
public new virtual void SpremiKaoXML()
{
      Console.WriteLine("Spremi mdb kao xml datoteku");
}
```

Vratimo se sad na osnovni tip *object* koji smo već obradili u drugom poglavlju. Rekli smo da je *object* klasa koju sve ostale klase implicitno nasljeđuju. Tu se ne misli samo na klase sadržane u FCL-u, nego i na sve ostale pa tako i na klasu *Access* u gornjem primjeru. U to se najlakše možemo uvjeriti ako napišemo ovo:

```
Access Acc = new Access();
Console.WriteLine(Acc.ToString());
```

Pozvali smo metodu *ToString* koja nije definirana ni u klasi *Access*, a ni u klasi *Office*. Očito je dakle da je to virutalna metoda klase *object* koja u *Access* klasi nije predefinirana i zato ispisuje ime klase. Kad bismo metodu *ToString* predefinirali (*override*) u *Access* klasi npr. ovako

```
public override string ToString()
{
      return "SQL Server";
}
```

gornji primjer bi ispisao:
SQL Server

Tip *object* ima još neke virtualne metode kao što su:

Equals
Ova metoda vraća tip *bool* ovisno o tome jesu li dva objekta jednaka, odnosno upućuju li na istu vrijednost.

```
Access Acc1 = new Access();
Access Acc2 = Acc1;
Console.WriteLine(Acc1.Equals(Acc2));
```

Ispis:
True

GetType
Vraća runtime tip (*System.Type*) objekta.

```
int i = 1;
Console.WriteLine(i.GetType());
Console.WriteLine(((long)i).GetType());
```

Ispis:
System.Int32
System.Int64

5.7. Apstraktne (Abstract) klase

U gornjem primjeru klase *Office* i njoj izvedenih klasa deklarirana je virtualna metoda koja je predefinirana u izvedenim klasama. Međutim, nitko ne obvezuje izvedene klase da predefiniraju te metode. Ako se želi da se metoda mora predefinirati u izvedenoj klasi, tada se ta metoda mora deklarirati ključnom riječi *abstract*, a ako je u klasi barem jedna metoda deklarirana kao *abstract*, onda i cijela klasa mora biti deklarirana kao *abstract*. To opet znači da klasa ne može biti instancirana, odnosno da se ne može kreirati objekt tipa apstraktne klase. Kad kažemo "barem jedna metoda" to znači da apstraktna klasa može sadržavati i neapstraktne nasljedive metode, ali i isto takva svojstva i varijable.

```
using System;

abstract class Office
{
```

```
        abstract public void SpremiDatoteku();

}

class Excel: Office
{
        public override void SpremiDatoteku()
        {
                Console.WriteLine("Spremi xls datoteku");
        }
}

class Test
{
        static void Main()
        {
                Excel Exc = new Excel();
                Exc.SpremiDatoteku();
        }
}
```

Primjer 5.9

Ispis:
Spremi xls datoteku

Metoda *SpremiDatoteku* u apstraktnoj klasi *Office* nema tijela, nego je samo navedena deklaracija metode. To je i razumljivo jer budući da se metoda mora implementirati u izvedenoj klasi i budući da ne možemo kreirati objekt tipa apstraktne klase, tu metodu nema tko pozvati. Apstraktna metoda ne može biti označena kao *static* ako ste mislili da bi je onda mogla pozvati sama klasa.

Apstraktne klase možemo koristiti kao obvezu izvedenim klasama kako bi trebale izgledati, odnosno koje sve metode bi dobro projektirana izvedena klasa trebala sadržavati.
Ipak, ako iz izvedene klase *Excel* izvedemo neku drugu klasu npr.

```
class ExcelViewer : Excel
{
}
```

ona nije obvezna implementirati apstraktne metode iz apstraktne klase *Office*, iako nasljeđuje sve nasljedive članove takve klase *Office*.

5.8. Zapečaćene (Sealed) klase

Zapečaćene (*sealed*) klase su klase koje ne mogu biti naslijeđene i deklariraju se ključnom riječi *sealed*. Ali naravno zapečaćene klase mogu biti instancirane. Jedan od primjera zapečaćene klase je klasa *System.String* odnosno poznata klasa *string* i ako pokušate ovo:

```
class MyString : string
{
}
```

dobit ćete poruku da zapečaćena klasa ne može biti naslijeđena.

Zapečaćena može biti i metoda ako ne želimo da bude predefinirana. Da smo u primjeru s baznom klasom *Office* ovako definirali metodu u izvedenoj klasi *Excel*:

```
public sealed override void SpremiDatoteku()
```

```
{
}
```

onda klasa *ExcelViewer* koja bi naslijedila klasu *Excel* ne bi mogla predefinirati metodu *SpremiDatoteku*, ali bi je mogla koristiti, odnosno ovo bi bilo dozvoljeno:

```
ExcelViewer ev = new ExcelViewer();
ev.SpremiDatoteku();
```

5.9. Sučelja (Interfaces)

Sučelje (*interface*) je skup deklaracija metoda, svojstava, događaja i indeksa koje klasa koja nasljeđuje sučelje mora implementirati. Nasljeđivanje sučelja osigurava da će klasa udovoljiti svim uvjetima koje sučelje postavlja, a to je implementacija svih članica sučelja. Možemo reći da je sučelje apstraktna klasa sa svim apstraktnim članicama.
Kod sučelja se može koristiti višestruko nasljeđivanje, klasa kao i sučelje može naslijediti više različitih sučelja. Ako klasa nasljeđuje klasu i sučelje, onda se nakon znaka ':' mora najprije navesti ime klase, a tek onda sučelja.
Kao i apstraktna klasa, sučelje ne može biti instancirano, a deklarirani članovi ne mogu imati modifikatore pristupa. Po definiciji svi članovi imaju modifikator pristupa *public*, ali nije dozvoljeno navesti ni modifikator *public* ispred deklaracije člana. Budući da se sučelje ne može instancirati, ono ne može sadržavati konstruktore i varijable članice.

Napravimo sada jedan primjer višestrukog nasljeđivanja sučelja. Imat ćemo dva sučelja *IListBox* i *ITextBox* (sučeljima se obično daju imena s prvim velikim slovom 'I' što je posljednji trag tzv. mađarske notacije koja je dugo vremena bila preporučavana od strane Microsofta). Također, definirat ćemo sučelje *IComboBox* koje nasljeđuje ova dva sučelja i klasu *ComboXP* koja će naslijediti sučelje *IComboBox*.

```
using System;

interface IListBox
{
      int SelectedItem {get; set;}
      void Sort();
}

interface ITextBox
{
      string Tekst {get; set;}
      void SelectTekst();
}

interface IComboBox : IListBox, ITextBox
{
      void Search();
}

class ComboXP : IComboBox
{
      int selectedItem;
      string tekst;

      public ComboXP(int item, string tx)
      {
            selectedItem = item;
```

```
            tekst = tx;
        }

        public int SelectedItem
        {
            get { return selectedItem; }
            set { selectedItem = value; }
        }

        public string Tekst
        {
            get { return tekst; }
            set { tekst = value; }
        }

        public void Sort()
        {
            Console.WriteLine("Implementirana metoda Sort");
        }

        public void Search()
        {
            Console.WriteLine("Implementirana metoda Search");
        }

        public void SelectTekst()
        {
            Console.WriteLine("Implementirana metoda SelectTekst");
        }
}

class Test
{
        static void Main()
        {
            ComboXP Standard = new ComboXP(0, "");
            Standard.Sort();
        }
}
```

Primjer 5.10

Ispis:
Implementirana metoda Sort

Vidimo da klasa *ComboXP*, iako ne nasljeđuje direktno sučelja *IListBox* i *ITextBox*, nasljeđuje ih preko sučelja *IComboBox*, a s tim i obvezu implementiranja svih njihovih članica. To npr. u stvarnosti može značiti da proizvođač klase *ComboXP* jamči svim kupcima svoje klase da će ona imati implementirane sve članice naslijeđenih sučelja.

Objekt tipa sučelja ipak možemo kreirati i to eksplicitno korištenjem *cast* operatora.

```
ComboXP Standard = new ComboXP(0, "");
IComboBox Icb = (IComboBox)Standard;
Icb.Sort();
```

S obzirom da klasa *ComboXP* nasljeđuje sučelje *IComboBox*, objekt će se kreirati i bez *cast* operatora, pa smo mogli i ovako napisati

```
IComboBox Icb = Standard;
```

Ipak pazite, *GetType*(*Icb*) će vratiti *ComboXP* tip.

Poseban bi slučaj bio kad bismo u gornjem primjeru objektom *Icb* htjeli pozvati metodu npr. *SetFocus* koja bi bila deklarirana u oba sučelja *ITextBox* i *IListBox* koje klasa *ComboXP* nasljeđuje. Tada prevoditelj ne bi znao koju od ovih dviju metoda treba pozvati

```
ITextBox.SetFocus();
```
ili
```
IListBox.SetFocus();
```

Prevoditelj bi se mogao odlučiti jedino ako bi se one pozivale različitim brojem ili tipom argumenata, ali bi onda klasa *ComboXP* morala i definirati dvije različite metode *SetFocus*. Drugo rješenje je da se, ako bi metode bile s istim tipom i brojem argumenata, u definiciji metoda navede od kojeg sučelja se metoda implementira (*fully qualified name*):

```
void IListBox.SetFocus() {. . . }
void ITextBox.SetFocus() {. . . }
```

Kad bi i sučelje *IComboXP* imalo deklariranu metodu *SetFocus*
```
new void SetFocus()
```
onda opet nema dvosmislenosti i pozvala bi se *IComboBox.SetFocus*.
Ili ako bismo napisali

```
ComboXP Standard = new ComboXP(0, "");
IListBox Ilb = Standard;
Ilb.SetFocus();
```

opet nema dvosmislenosti i poziva se *IListBox.SetFocus()*.

Klasu možemo eksplicitno pretvoriti u bilo koje sučelje i prevoditelj neće javiti grešku. Međutim kod izvođenja dogodit će se iznimka o nepravilnoj *cast* operaciji i program će prestati s radom. Osim ako iznimka nije uhvaćena, ali o tome više u sljedećim poglavljima. Ovo će se ispravno prevesti, ali ne i izvršiti:

```
ComboXP Standard = new ComboXP(0, "");
IEnumerable Itb = (IEnumerable)Standard;
```

IEnumerable je ugrađeno sučelje iz FCL-a.

Vidjeli smo da su sučelja slična apstraktnim klasama, ali ipak razlike postoje pa ih navodimo u sljedećoj tablici:

Usporedba sučelja i apstraktne klase	
Sučelje	**Apstraktna klasa**
Ne može biti instancirano	Ne može biti instancirana, ali izvedena klasa može pozvati konstruktor apstraktne klase ključnom riječi *base*.
Sadrži **samo** deklaracije članova koji moraju biti definirani u izvedenim klasama	Sadrži deklaracije članova koji moraju biti definirani u izvedenim klasama, ali može sadržavati i implementacije metoda
Nije moguće dodati nove deklaracije članova, a da se ne naruši linija nasljeđivanja (*versioning*)	Moguće je dodati nove neapstraktne članove koji će moći biti naslijeđeni u izvedenim klasama, a da se ne naruši linija nasljeđivanja (*versioning*)

Ne može sadržavati definirane varijable kao članove	Može sadržavati definirane varijable kao članove
Klasa može naslijediti više od jednog sučelja te kombinaciju klase i jednog ili više sučelja	Klasa ne može naslijediti više od jedne apstraktne klase

5.10. Operator is

Sintaksa korištenja *is* operatora:
`izraz is tip`
gdje je *izraz* objekt za koji se provjerava je li podudaran (kompatibilan) s *tipom*. Operator vraća *true* ako je oboje od sljedećeg istinito:

- *izraz* je različit od *null*
- *izraz* može biti pretvoren u *tip*. Drugim rječima, *cast* naredba (*tip*)*izraz* neće generirati iznimku (exception)

Operator *is* uzima u obzir samo *cast* referentnih tipova, boksiranje (boxing) i odboksiranje (unboxing).

```
bool flag1 = false, flag2 = false;
object o = 10;
string s = "abcdef";
if(s is object)
      flag1 = true;
if(o is string)
      flag2 = true;
```

Nakon izvršenja ovog kôda u *flag1* će biti *true*, a u *flag2* će biti *false*.
To će biti zato jer bi u gornjem primjeru *cast* `(string)o` generirao iznimku, a *cast* `(object)s` ne.
Tipovi *object* i *string* su referentni tipovi i zato se operator *is* može koristiti. Ako bismo umjesto ovih referentnih tipova koristili vrijednosne npr. *int* i *long*, operator *is* bi u oba slučaja vratio *false*, pa bi u obje *flag* varijable bio *false*.
Ali boksiranje i odboksiranje radi, pa bi ovakav primjer također u *flag1* postavio *true*, a u *flag2* bi ostavio *false*.

```
bool flag1 = false, flag2 = false;
object o = 10;
long l = 20;
if(l is object)
      flag1 = true;
if(o is long)
      flag2 = true;
```

Prevoditelj će međutim za oba primjera javiti upozorenje da će `if(l is object)`, odnosno `if(s is object)` biti sigurno *true*.

5.11. Operator as

Sintaksa korištenja *as* operatora:
`izraz as tip`

Zamislimo da smo u gornjem primjeru s operatorom *is*, nakon što bi izraz `if(s is object)` vratio *true*, u tijelu *if* naredbe trebali izvršiti neku operaciju sa stringom pretvorenim u *object*. Očito je da bismo morali napisati ovako nešto

```
object o1 = (object)s;
```

a još je očitije da se u tom slučaju dvaput provjerava je li *izraz* kompatibilan s *tipom*. Prvi put u provjeri *if* uvjeta, a drugi put u tijelu *if* naredbe.
C# zato uvodi operator *as* koji će napraviti to dvoje odjednom. Operator *as* je implementiran u obliku uvjetnog izraza kao:

```
izraz is tip ? (tip)izraz : (tip)null
```

Iz ovoga je vidljivo da će operator *as* vratiti *izraz* pretvoren u *tip* ako je *izraz* kompatibilan s *tipom* ili *null* ako *izraz* nije kompatibilan s *tipom*. Zato smo gornji primjer s operatorom *is* mogli napisati ovako nekako koristeći operator *as*

```
object o1 = s as object;
if(o1 != null)
      . . . ;
```

Operatori *is* i *as* mogu primjenu naći i u sučeljima gdje njima možemo izvršiti provjeru je li objekt tipa sučelja odnosno može li se pretvoriti u tip sučelja.
Pogledajmo to na sljedećem primjeru.

```
using System;

interface IComboXP
{
      void Sort();
}

interface IComboVista
{
      void SortAdvanced();
}

class ComboXP : IComboXP
{
      public void Sort()
      {
            Console.WriteLine("Sortirano");
      }
}

class ComboVista : ComboXP, IComboVista
{
      public void SortAdvanced()
      {
            Console.WriteLine("Napredno sortirano");
      }
}

class Test
{
      static void Main()
      {
```

```
            ComboXP [] Arr = new ComboXP[4];
            Arr[0] = new ComboXP();
            Arr[1] = new ComboXP();
            Arr[2] = new ComboVista();
            Arr[3] = new ComboVista();

            foreach(ComboXP com in Arr)
            {
                  if(com is IComboVista)
                  {
                        IComboVista Icv = (IComboVista) com;
                        Icv.SortAdvanced();
                  }
                  else if(com is IComboXP)
                  {
                        IComboXP Ixp = (IComboXP) com;
                        Ixp.Sort();
                  }
            }

      }
}
```

Primjer 5.11

Ispis:
Sortirano
Sortirano
Napredno sortirano
Napredno sortirano

Imamo dva sučelja *IComboXP* i *IComboVista* te dvije klase:

- *ComboXP* koja nasljeđuje sučelje *IComboXP* i implementira metodu *Sort*
- *ComboVista* koja nasljeđuje klasu *ComboXP* i sučelje *IComboVista* te implementira metodu *AdvancedSort*

U funkciji *Main* definirali smo niz od 4 člana niza tipa *ComboXP*, s tim da smo za zadnja dva člana koristili konstruktor *ComboVista* klase pa smo tako dobili dva objekta tipa *ComboXP* (prva dva) i dva objekta *ComboVista* (zadnja dva).
Operatorom *is* provjeravamo je li objekt iz petlje tipa sučelja *IComboVista*, a to će biti ako je tipa klase koja nasljeđuje sučelje *IComboVista* (zadnja dva člana niza).
U slučaju ako je rezultat takve provjere *true* pozivamo metodu *SortAdvanced* klase *ComboVista*, a u slučaju da nije tipa sučelja *IComboVista*, to onda znači da je tipa *IComboXP* što i mora biti jer klasa *ComboXP* nasljeđuje sučelje *IComboXP*.

Zato nismo ni morali u *else* dijelu navesti provjeru
`else if(com is IComboXP)`
dovoljno je bilo napisati *else* bez uvjeta.
Također u tijelu *else* dijela nismo morali raditi eksplicitnu pretvorbu u *IComboXP*, dovoljno je bilo napisati `com.Sort()`.
Dakle, za prva dva člana niza izvršava se *else* dio naredbe a za druga dva člana izvršava se *if* dio.

Operator *as* možemo vidjeti na djelu ako bi *foreach* petlja iz gornjeg primjera izgledala ovako (rezultat izvršenja programa isti je u oba slučaja)

```
foreach(ComboXP com in Arr)
{
	IComboVista Combo = com as IComboVista;
	if(Combo == null)
	{
		IComboXP Combo1 = com as IComboXP;
		Combo1.Sort();
	}
	else
		Combo.SortAdvanced();
}
```

Vrlo slično kao i kod operatora *is*, s operatorom *as* provjeravamo može li se objekt *com* iz petlje pretvoriti u tip sučelja *IComboVista*. Ako se objekt ne može pretvoriti (*if* dio), pretvaramo ga opet operatorom *as* u tip sučelja *IComboXP* i pozivamo metodu *Sort*. Kao i kod primjera s *is* operatorom mogli smo koristiti i ovakav poziv

```
com.Sort();
```

U slučaju da se objekt može pretvoriti u *IComboVista* (*else* dio), poziva se metoda *SortAdvanced* klase *ComboVista*.

5.12. Strukture

Struktura je složeni vrijednosni tip sličan klasi, ali ipak s puno manje mogućnosti. Strukture se koriste za spremanje jednostavnih skupova podataka kao npr. koordinata, stranica pravokutnika itd… Deklaracija strukture slična je klasi, samo što umjesto ključne riječi *class* dolazi ključna riječ *struct*.

```
public struct Pravokutnik
{
	public int Sirina, Visina;

	public Pravokutnik(int a, int b)
	{
		Sirina = a;
		Visina = b;
	}
}
```

Struktura može sadržavati sljedeće članove:

- konstruktore
- metode
- varijable
- svojstva
- indekse
- konstante
- preopterećene operatore

Strukture se od klasa ipak dosta razlikuju, pa u sljedećoj tablici navodimo glavne razlike:

Tablica razlika između klasa i struktura:

Usporedba klasa i struktura	
Klasa	**Struktura**
Referentni tip podataka, kreiran na hrpi (heap)	Vrijednosni tip podataka, kreiran na stogu (stack)
Ne može biti instancirana bez operatora *new*	**Može** biti instancirana bez operatora *new*
Može naslijediti klasu i može biti naslijeđena	**Ne može** naslijediti klasu i ne može biti naslijeđena, ali može naslijediti sučelje. Implicitno nasljeđuje klasu *object*.
Može se inicijalizirati varijabla kod deklaracije	**Ne može** se inicijalizirati varijabla kod deklaracije
Može se redefinirati konstruktor bez parametara	**Ne može** se redefinirati konstruktor bez parametara

Glavna razlika između struktura i klasa je u tome što je struktura vrijednosni, a klasa referentni tip što sa sobom povlači sve one razlike obrađene u drugom poglavlju. Pogledajmo ovaj primjer:

```
using System;

struct SKoordinate
{
	public int XCoord, YCoord;

	public SKoordinate(int x, int y)
	{
		XCoord = x;
		YCoord = y;
	}
}

class CKoordinate
{
	public int XCoord, YCoord;

	public CKoordinate(int x, int y)
	{
		XCoord = x;
		YCoord = y;
	}
}

class MainClass
{
	public static void PromijeniXYs(SKoordinate sTmp)
	{
		sTmp.XCoord = 100;
		sTmp.YCoord = 200;
	}

	public static void PromijeniXYc(CKoordinate cTmp)
	{
		cTmp.XCoord = 100;
		cTmp.YCoord = 200;
	}

	public static void Main()
```

```
        {
                SKoordinate s = new SKoordinate(800, 600);
                CKoordinate c = new CKoordinate(1024, 768);

                PromijeniXYs(s);
                PromijeniXYc(c);

                Console.WriteLine(s.XCoord + ", " + s.YCoord);
                Console.WriteLine(c.XCoord + ", " + c.YCoord);
        }
}
```

Primjer 5.12

Ispis:
800, 600
100, 200

Definirali smo strukturu i klasu za isto, za spremanje koordinata. Zbog pojednostavljenja, varijable koordinata smo deklarirali *public*, inače u stvarnosti bi naravno trebale biti *private*, a pristupali bismo im preko svojstava.
S obzirom da je struktura vrijednosni tip, kod prosljeđivanja metodi *PromijeniXYs* prenosi se kopija strukture, dok se kod klase prenosi adresa memorijske lokacije na kojoj se nalazi podatak koji objekt predstavlja. Zato se koordinate strukture ne mijenjaju, za razliku od koordinata klase.
Kao što je navedeno u tablici, strukture se mogu kreirati i bez ključne riječi *new* pa smo mogli i ovako navesti:

```
SKoordinate s;
s.XCoord = 800;
s.YCoord = 600;
```

Kod struktura nema podrazumijevanih vrijednosti kao kod klasa. Da smo u našem primjeru tijela konstruktora ostavili prazna, kod klasa bi se varijable *XCoord* i *YCoord* inicijalizirale na podrazumijevane vrijednosti, a to je za tip *int* 0, dok bi za strukture prevoditelj javio grešku o nedodijeljenoj vrijednosti.

Strukture podatke spremaju na dijelu memorije zvanom stog (*stack*), za razliku od klasa koje podatke spremaju na hrpi (*heap*). Zato su strukture efikasnije kod rada s podacima spremljenim u nizovima. U sljedećem primjeru izračunat ćemo koliko vremena treba strukturama, a koliko klasama da kreiraju 10 milijuna parova koordinata.
Da ne bismo ponavljali cijeli program s deklaracijama strukture i klase, navest ćemo samo *Main* funkciju, a ostalo ostaje kao u prethodnom primjeru.

```
using System;
// . . .
// . . .
public static void Main()
{
        const int Num = 10000000;
        DateTime start = DateTime.Now;

        SKoordinate[] s = new SKoordinate[Num];
        for(int i = 0;i<Num;i++)
                s[i] = new SKoordinate(i, i);

        DateTime end1 = DateTime.Now;
```

```
        TimeSpan Int1 = end1 - start;
        Console.WriteLine("Strukture kreirane za " + Int1.Seconds + "."
                                                + Int1.Milliseconds + " sec");
        CKoordinate[] c = new CKoordinate[Num];
        for(int i = 0;i<Num;i++)
              c[i] = new CKoordinate(i, i);

        DateTime end2 = DateTime.Now;
        TimeSpan Int2 = end2 - end1;
        Console.WriteLine("Klase kreirane za " + Int2.Seconds + "." +
                                                Int2.Milliseconds + " sec");
}
```

Primjer 5.13

Ispis:
Strukture kreirane za 0.171 sec
Klase kreirane za 4.453 sec

Prolazna vremena računamo već spomenutom funkcijom *Now* iz klase *DateTime*, a njihovu razliku koristeći objekt klase *TimeSpan*. Vidimo da je velika razlika u potrebnom vremenu za kreiranje nizova, u ovom primjeru čak 26 puta u korist struktura. Naravno, ovo je ekstremni primjer i pitanje je koliko će netko u rješavanju stvarnih problema moći profitirati iz ovakve razlike u brzini izvršenja.

5.13. Ugnježđene klase (Nested classes)

Ugnježđenje klasa je slučaj kada je jedna klasa (unutarnja) članica druge klase (vanjska). Ugnježđene klase koriste se uglavnom kada takvu klasu nema smisla koristiti osim u sklopu klase kojoj ona pripada.

```
using System;

class Vanjska
{
      private int x = 100;

      public class Unutarnja
      {
            public void fIn()
            {
                  Vanjska v = new Vanjska();
                  Console.WriteLine(v.x);
            }
      }
}

class Test
{
      static void Main()
      {
            Vanjska.Unutarnja u = new Vanjska.Unutarnja();
            u.fIn();
      }
}
```

Primjer 5.14

Ispis:
100

S obzirom da je klasa *Unutarnja* članica klase *Vanjska*, njoj se pristupa kao i svakom drugom članu klase s *dot* operatorom (.).
Zanimljivo, iako ne i neočekivano je što unutarnja klasa ima pristup privatnoj varijabli *x* vanjske klase. Kažemo ne i neočekivano jer s obzirom da je unutarnja klasa članica vanjske klase, ona treba imati sva prava pristupa članovima svoje klase kao i sve druge članice kao što su metode, svojstva itd.

Unutarnja klasa može imati i druge modifikatore pristupa osim *public*. Pretpostavimo da sada unutarnju klasu označimo modifikatorom *private*. To automatski znači da ona ne može biti vidljiva izvan klase kojoj pripada, odnosno da se ovo u funkciji *Main* neće prevesti:

```
Vanjska.Unutarnja u = new Vanjska.Unutarnja();
```

Jedno od rješenja jest da se definira metoda *fOut*, članica klase *Vanjska* koja će onda imati pristup privatnom članu klase kojoj i sama pripada, te se tako preko instance vanjske klase poziva metoda unutarnje klase.

```
using System;

class Vanjska
{
	private int x = 100;

	private class Unutarnja
	{
		public void fIn()
		{
			Vanjska v = new Vanjska();
			Console.WriteLine(v.x);
		}
	}

	public void fOut()
	{
		Unutarnja u = new Unutarnja();
		u.fIn();
	}
}

class Test
{
	static void Main()
	{
		Vanjska v = new Vanjska();
		v.fOut();
	}
}
```

Primjer 5.15

Ispis:
100

Obratno ne vrijedi, vanjska klasa nema pristup privatnim članicama unutarnje klase. Ako u unutarnjoj klasi definiramo varijablu

```
private int y = 5;
```

ovo se neće prevesti u metodi *fOut*:

```
Unutarnja u = new Unutarnja();
u.y = 6;
```

5.14. Imenik (Namespaces)

Zamislimo npr. jednu hidrocentralu koja ima Westinghousov generator te klasu *Generator* u sklopu software-a za upravljanje generatorom. Sada zamislimo da ista ta hidrocentrala kupi još jedan generator, ali ovaj put od Končara koji u software-u ima svoju klasu *Generator*. Cilj hidrocentrale je, naravno, imati jedan software za upravljanje koji bi trebao koristiti obje klase *Generator*.
Ali što ako bismo napisali:

```
Generator g = new Generator(100);
```

objekt koje klase *Generator* bi se kreirao? Westinghousove ili Končarove?
Ne bi se kreirao niti jedan jer prevoditelj uopće ne bi preveo program zbog toga što dvije klase imaju isto ime.
Rješenje ovakvih situacija je da se klase organiziraju u tzv. imenike (*namespace*) odnosno skupove klasa.

```
using System;

namespace Westinghouse
{
	class Generator
	{
		private float PowerMW;

		public Generator(float f)
		{
			PowerMW = f;
		}
	}
}

namespace Koncar
{
	class Generator
	{
		private float SnagaMW;

		public Generator(float f)
		{
			SnagaMW = f;
		}
	}
}

class Test
{
	static void Main()
	{
		Westinghouse.Generator w = new Westinghouse.Generator(100);
		Koncar.Generator s = new Koncar.Generator(120);
	}
}
```

Primjer 5.16

Imenik može imati i više klasa i svima se pristupa navođenjem punog imena (fully qualified name). Preporuka je da se imenicima daju imena tvrtke koja je izradila klasu. To omogućuje dovoljno sigurnu zaštitu od jednakih punih imena klasa.
Imenik po definiciji ima implicitni modifikator pristupa *public* i nije dozvoljeno navođenje modifikatora za imenik.

Imenici mogu biti hijerarhijski ugnježđeni radi bolje organizacije kao npr.

```
namespace Westinghouse
{
      namespace Machines
      {
            namespace Large
            {
                  class Generator
                  {
```

Klasi *Generator* bismo tada pristupili ovako:
```
Westinghouse.Machines.Large.Generator w = new
                  Westinghouse.Machines.Large.Generator(100);
```
Očito je ovakvu putanju teško i zapisati, a kamoli zapamtiti. C# zato uvodi ključnu riječ *using* koja omogućuje referenciranje cijelog punog imena bez navođenja istog.

```
using WestGenerator = Westinghouse.Machines.Large.Generator;
```

Sada možemo napisati
```
WestGenerator w = new WestGenerator(100);
```
i kreirat će se objekt tipa Westinghousovog generatora.

FCL je također organiziran u obliku imenika i kada navedemo npr.
```
System.IO.File
```
referencirali smo se na klasu *File* u imeniku *IO* koji je dio imenika *System*.

Izrada biblioteka
Klase se uglavnom isporučuju u prevedenom obliku, odnosno u sklopu biblioteke (library). Ako bi klase potrebne za rad Westingousovih strojeva bile spremljenih u datoteci *Machines.cs*, onda bi se biblioteka pod imenom *WestinghouseMachines.dll* napravila pozivom programa za prevođenje na ovakav način:

```
csc /target:library /out:WestinghouseMachines.dll Machines.cs
```

dok bi neki klijent koji želi koristiti te klase morao u prevođenju navesti ime biblioteke:

```
csc /out:Test.exe /R:WestinghouseMachines.dll Test.cs
```

Ako koristite Visual Studio, onda je postupak jednostavniji i svodi se na par klikova mišem:

za izradu biblioteke:
File -> New -> Project -> C# Class Library -> Build

a za klijentski program koji koristi biblioteku:
View -> Solution Explorer -> Add Reference -> Browse -> Datoteka s bibliotekom

Poglavlje 6:
Metode

Ovo poglavlje obrađuje:

- deklaraciju i definiciju metoda
- poziv metoda s i bez argumenata
- povratnu vrijednost iz metoda
- preopterećenje metoda
- parametre *ref*, *out* i *param*
- rekurzivni poziv metoda

6.1. Deklaracija i povratna vrijednost

Metoda, funkcija ili procedura, tri naziva za jedno te isto, u proceduralnim jezicima je bila temeljna stvar u svakom programu što se vidi po tome da je po njoj proceduralno programiranje i dobilo ime. Kod objektnog programiranja u središte svega postavljene su klase i objekti, ali daleko od toga da su metode postale nebitne ili manje bitne. Na kraju krajeva svaka klasa sadrži metode, a i svojstva kao malo drugačiju implementaciju standardnih oblika metoda. Najbolji pokazatelj koliko su metode bitne i u objektnom programiranju su svi do sad obrađeni primjeri u knjizi koji su u velikoj mjeri koristili metode.

Svaka metoda, slično kao i svaka varijabla sastoji se od deklaracije i definicije:

```
Modifikator Tip NazivMetode(argumenti)
{
      // . . .
}
```

- Modifikator metode uglavnom je *public*, zato što se korištenje objekata uglavnom svodi na pozivanje metoda i svojstava pa one moraju biti vidljive izvan klase.
- Tip metode može biti bilo koji od tipova jezika (*int, long, double*, neka klasa…) i on određuje povratnu vrijednost koju metoda vraća s ključnom riječi *return*. Poseban tip metode je tip *void* koji znači da metoda ne vraća nikakvu vrijednost.
- Kod naziva metode vrijede pravila kao i za sve identifikatore, a preporučljivo je i tu koristiti Pascal notaciju.
- Argumenti mogu biti: niti jedna, jedna ili više varijabli proslijeđenih metodi od pozivatelja. Bitno je da tipovi varijabli kod poziva metode odgovaraju tipovima u definiciji metode.
- Definicija metode ograđena je vitičastim zagradama (mnogi programeri navikli na C sintaksu kažu da bi se najteže odrekli upravo vitičastih zagrada) unutar kojih se navode naredbe s tim da se mora osigurati povratak vrijednosti naredbom *return*.

S obzirom da je u C# sve dio klase, deklaracija i definicija metode više nisu razdvojene kao u C/C++, odnosno deklaracija je sadržana u samoj definiciji metode.

Sljedeći primjer je metoda koja će vratiti vrijednost kompleksnog broja.

```
using System;

class Test
{
      public static double KompleksnaVrijednost(double real, double im)
      {
            double d = Math.Sqrt(real * real + im * im);
            return d;
      }

      static void Main()
      {
            double Real = 6.7, Imagine = 5.3;
            double Absolute = KompleksnaVrijednost(Real, Imagine);
            Console.WriteLine(Absolute);
      }
}
```

Primjer 6.1

Ispis:
8,54283325366942

Metoda *KompleksnaVrijednost* računa vrijednost kompleksnog broja koji se sastoji od realnog i imaginarnog dijela. Oni se prosljeđuju metodi koja iz njih računa vrijednost te je vraća naredbom *return*.
Mogli smo proći i bez nepotrebnog kreiranja varijable *Absolute* u funkciji *Main* te pozvati metodu ovako:

```
Console.WriteLine(KompleksnaVrijednost(Real, Imagine));
```

a isto tako nije nam trebala ni varijabla *d* u metodi *KompleksnaVrijednost* koja je mogla vratiti vrijednost ovako:

```
return Math.Sqrt(real * real + im * im);
```

Obratite pozornost da je u gornjem primjeru metoda *KompleksnaVrijednost* mogla biti označena i modifikatorom *private* i program bi se izvršio na isti način jer metodu pozivamo iz funkcije *Main*, dakle iz iste klase kojoj i metoda *KompleksnaVrijednost* pripada. Također primijetite da je metoda označena kao *static* da bi se mogla pozvati iz statičke *Main* funkcije.

Kod prosljeđivanja argumenata radi se implicitna konverzija ako varijable nisu istog tipa. Da smo u funkciji *Main* ovako definirali varijable

```
float Real = 6.7f, Imagine = 5.3f;
```

one bi se u metodi automatski pretvorile u *double*. Obrnuto ne vrijedi, nismo mogli u deklaraciji metode *KompleksnaVrijednost* staviti

```
private static double KompleksnaVrijednost(float real, float im)
```

i očekivati automatsku konverziju *double* u *float*.

Ako je metoda tipa *void*, to znači da ne vraća nikakvu vrijednost. Tada u metodi nije obvezno navoditi naredbu *return*, a ako se navodi, treba se navesti bez argumenata.
Naredba *return* ne samo što vraća vrijednost iz metode, nego i trenutačno prekida izvršenje metode. Ako bismo metodu ovako definirali:

```
public static double KompleksnaVrijednost(double real, double im)
{
      return 1;
      double d = Math.Sqrt(real * real + im * im);
}
```

ona bi uvijek vraćala vrijednost 1, iako bi prevoditelj javio upozorenje o dijelu kôda koji nikad ne može biti dosegnut.
Također, *return* može vratiti i cijeli izraz kao npr.

```
return 1 + 2 + 3;
```

Metoda može vratiti i objekt neke klase, a povratna vrijednost funkcije ne može biti više od jedne varijable. To ograničenje lako se može zaobići tako da metoda vrati niz.
Napravimo sada jedan primjer u kojem će metoda vratiti niz objekata.

```
using System;

class MobileDevice
{
      private string opSustav;
      private float marketShare;

      public MobileDevice(string os)
      {
            opSustav = os;
      }

      public float MarketShare
      {
            get   {return marketShare;}
            set   {marketShare = value;}
      }

      public string OPSustav
      {
            get   {return opSustav;}
            set   {opSustav = value;}
      }

}

class Test
{
      public static MobileDevice[] SetMarketShare(MobileDevice[] Arr)
      {
            Arr[0].MarketShare = 45;
            Arr[1].MarketShare = 35;
            Arr[2].MarketShare = 20;

            return Arr;
      }

      static void Main()
      {
            MobileDevice[] Gadgets = new MobileDevice[3];
            Gadgets[0] = new MobileDevice("Palm OS");
            Gadgets[1] = new MobileDevice("Pocket PC");
            Gadgets[2] = new MobileDevice("Symbian");

            Gadgets = SetMarketShare(Gadgets);

            for(int i = 0;i<3;i++)
                  Console.WriteLine(Gadgets[i].OPSustav + " - " +
                                    Gadgets[i].MarketShare + " %");
      }
}
```

Primjer 6.2

Ispis:
Palm OS - 45 %
Pocket PC - 35 %
Symbian - 20 %

Definirali smo klasu *MobileDevice* koja ima samo dva člana, operacijski sustav *opSustav* i postotak udjela na tržištu *marketShare*. U funkciji *Main* kreirali smo niz od 3 objekta ove klase, ali bez definiranja *marketShare* u konstruktoru što znači da će varijabla inicijalno imati vrijednost 0. Iako imamo niz objekata, riječ je o samo jednoj varijabli koju prosljeđujemo metodi *SetMarketShare* koja će svakom od tih objekata dodijeliti udio na svjetskom tržištu definirajući varijablu *marketShare*.

Ono što je ovaj primjer htio naglasiti, jest da metoda vraća niz objekata naredbom
`return Arr;`
i zato je morala biti propisno deklarirana kao tip `MobileDevice[]`.

6.2. Preopterećene metode

Preopterećenje metoda (*Method overloading*) je slučaj kada klasa ima više metoda istog imena, ali različitog broja argumenata. S ovakvim metodama već smo se susretali kod konstruktora kojih u jednoj klasi može biti više, a isto je i s drugim metodama.

Napravit ćemo jedan primjer u kojem ćemo imati preopterećenu metodu koja će imati 4 različite definicije:

```
using System;

class TempClass
{
	public void MethodOne(int i)
	{
		Console.WriteLine(i);
	}

	public void MethodOne(double d)
	{
		Console.WriteLine(d);
	}

	public void MethodOne(double d1, double d2)
	{
		Console.WriteLine(d1 * d1 + d2 * d2);
	}

	public void MethodOne(string s)
	{
		Console.WriteLine(s);
	}
}

class Test
{
	static void Main()
	{
		int i = 10;
		TempClass x = new TempClass();
		x.MethodOne(i);
		x.MethodOne(40.67);
		x.MethodOne(6, 8);
		x.MethodOne("Broj 1");
	}
}
```

Primjer 6.3

Ispis:
10
40,67
100
Broj 1

Metoda *MethodOne* poziva se na 4 različita načina argumentom *int*, *double*, dva puta *double* i *string*, a prevoditelj prema tipu i broju argumenata određuje koju od njih će pozvati.
Kod poziva metode može se iskoristiti implicitna konverzija koja će se automatski dogoditi kod prenošenja argumenata.

Da smo umjesto
`public void MethodOne(int i)`
imali definirano
`public void MethodOne(long i)`
program bi se jednako izvršio.

Iako nije dozvoljeno prenošenje argumenata kod kojih je potrebna eksplicitna konverzija, ako u gornjem primjeru varijablu *i* definiramo kao
`long i = 10;`
program će se i prevesti i izvršiti bez grešaka iako se iz tipa *long* u tip *int* događa eksplicitna konverzija. Zašto?

Odgovor:
Zato što nema potrebe za eksplicitnom konverzijom iz *long* u *int* jer prevoditelj vidi moguću implicitnu konverziju iz *long* u *double*. I zato će za varijablu
`long i = 10;`
prevoditelj pozvati metodu *MethodOne*(*double*).

6.3. Parametar ref

Kod prenošenja argumenata treba obratiti pozornost je li argument koji se prenosi vrijednosnog ili referentnog tipa. Ovo je osobito bitno ako metoda mijenja vrijednost dobivenog argumenta. To je već objašnjeno detaljno u poglavlju 2.4 o referentnim tipovima.

```
using System;

class Test
{
      public static void Promijeni(int i)
      {
            i = 11;
            return;
      }

      static void Main()
      {
            int x = 10;
            Console.WriteLine(x);
            Promijeni(x);
            Console.WriteLine(x);
      }
}
```

Primjer 6.4

Ispis:
10
10

Iako metoda *Promijeni* mijenja vrijednost svog argumenta, to se događa na kopiji originalne varijable i zato povećanje za jedan nema utjecaja na izvornu vrijednost varijable *x* u funkciji *Main*. Ako bi iz nekog razloga bilo potrebno da metoda promijeni vrijednost prosljeđenog vrijednosnog tipa, to se može napraviti da se argument označi ključnom riječi *ref* i to u pozivu metode kao i u deklaraciji metode.

```
using System;

class Test
{
	public static void Promijeni(ref int i)
	{
		i = 11;
		return;
	}

	static void Main()
	{
		int x = 10;
		Console.WriteLine(x);
		Promijeni(ref x);
		Console.WriteLine(x);
	}
}
```

Primjer 6.5

Ispis:
10
11

Navođenjem parametra *ref* uz varijablu prevoditelj prenosi vrijednosnu varijablu kao referentni tip i zato će metoda *Promijeni* povećati vrijednost originalnoj varijabli *x*, a ne njenoj kopiji. Parametar *ref* se odnosi samo na onu varijablu uz koju je naveden. Ako je metoda ovako deklarirana:

```
public static void Promijeni(ref int i, int j)
```

i ako bismo je pozvali kao:

```
Promijeni(ref x, y);
```

onda bi metoda mogla promijeniti originalnu vrijednost samo varijable *x*, ali ne i *y*.

Ako ste mislili da parametar *ref* nema smisla navoditi uz referentne tipove, jer zašto nešto što je već referentni tip pretvarati u referentni tip, pogledajte sljedeći primjer.

```
using System;
using System.Text;

class Test
{
	public static void Promijeni1(StringBuilder sb)
	{
		sb = new StringBuilder("Linux");
```

```
            return;
        }

        public static void Promijeni2(ref StringBuilder sb)
        {
            sb = new StringBuilder("Linux");
            return;
        }

        static void Main()
        {
            StringBuilder s1 = new StringBuilder("Windows");
            Promijeni1(s1);
            Console.WriteLine(s1);

            StringBuilder s2 = new StringBuilder("Windows ");
            Promijeni2(ref s2);
            Console.WriteLine(s2);
        }
}
```

Primjer 6.6

Ispis:
Windows
Linux

Iako u obje metode *Promijeni1* i *Promijeni2* imamo isti kôd, samo druga metoda mijenja varijablu tipa *StringBuilder*. Argument *sb* u metodi *Promijeni1* je varijabla različita od varijable *s1* iz funkcije *Main* samo što upućuju na istu vrijednost, dok je argument *sb* u *Promijeni2* upravo varijabla *s2* iz funkcije *Main*.

Ako bismo u metodi *Promijeni1* zamijenili liniju

```
sb = new StringBuilder("Linux");
```

s

```
sb.Remove(0, sb.Length);
sb.Append("Linux");
```

tada bi i metoda *Promijeni1* promijenila string Windows u Linux jer se ne bi dogodilo kreiranje novog objekta i varijabla *sb* u ove dvije naredbe bila bi ona koja upućuje na istu vrijednost kao i varijabla *s1* iz funkcije *Main*.

6.4. Parametar out

U primjeru s *ref* parametrom metoda *Promijeni* dodjeljuje vrijednost 11 proslijeđenoj varijabli bez obzira s kojom vrijednosti je varijabla `"ušla"` u metodu. Međutim, kad koristimo parametar *ref*, varijabla koja se prosljeđuje mora biti definirana, iako u ovom slučaju to nema nikakvog utjecaja na njenu konačnu vrijednost nakon izvođenja metode *Promijeni*.

Ako želimo izbjeći tu obvezu prethodnog definiranja varijable, možemo koristiti parametar *out* koji ne zahtijeva definiranje varijable prije prosljeđivanja, ali zato zahtijeva da se varijabli u metodi obvezno dodijeli vrijednost.

```
using System;
```

```
class Test
{
	public static void Promijeni(out int i)
	{
		i = 11;
		return;
	}

	static void Main()
	{
		int x;
		Promijeni(out x);
		Console.WriteLine(x);
	}
}
```

Primjer 6.7

Ispis:
11

Kao što primjer pokazuje, sintaksa korištenja parametra *out* ista je kao i kod parametra *ref*.

Usporedba parametara ref i out	
ref	**out**
Varijabla prije poziva **mora** biti definirana	Varijabla prije poziva **ne mora**, ali **može** biti definirana
Varijabli se u metodi **može**, ali i **ne mora** dodijeliti vrijednost	Varijabli se u metodi **mora** dodijeliti vrijednost
Varijabli se u metodi **ne mora** direktno dodijeliti određena vrijednost, već se **može** promijeniti npr. operatorom ++	Varijabli se u metodi **mora** direktno dodijeliti određena vrijednost s operatorom =, i **ne može** se promijeniti (samo s) npr. operatorom ++

Parametre *ref* i *out* ne možemo koristiti da bismo s njima razlikovali listu argumenata za preopterećenje metoda jer se njihovo razlikovanje događa u vremenu izvođenja, a ne u vremenu prevođenja. Zato ovakve dvije definicije metoda prevoditelj neće pustiti:

```
public static int Promijeni(ref int i)
{
	//. . .
}

public static int Promijeni(out int i)
{
	//. . .
}
```

ali će ove dvije proći:

```
public static int Promijeni(ref int i)
{
	//. . .
}

public static int Promijeni(int i)
{
```

```
        //. . .
}
```

6.5. Parametar param

Parametar *param* omogućuje da se metodi prenese varijabilan broj argumenata. Sjetite se primjera u prvom poglavlju s paralelnim spojem otpornika. Program je računao ukupni otpor za dva paralelno spojena otpornika. Iako to u tom primjeru nije riješeno pomoću metode, rješenje bi bilo vrlo slično. Imali bismo metodu s dva argumenta koja bi vratila izračunatu vrijednost.

Ali paralelnih otpornika može biti i više od dva što znači da bismo trebali za svaki broj otpornika imati posebnu metodu. Tu dolazi do izražaja parametar *param* što će pokazati sljedeći primjer:

```
using System;

public class Test
{
      public static double UseParams(params double[] Otpori)
      {
            if(Otpori.Length == 0)
                  return 0;
            else if(Otpori.Length == 1)
                  return Otpori[0];

            double UkupniOtpor = Otpori[0] * Otpori[1] / (Otpori[0] +
                                                      Otpori[1]);
            for(int i = 2; i < Otpori.Length; i++ )
                  UkupniOtpor = UkupniOtpor * Otpori[i] / (UkupniOtpor +
                                                      Otpori[i]);
            return UkupniOtpor;
      }

      public static void Main()
      {
            double R = UseParams(10);
            Console.WriteLine("{0} Ohm", R);

            R = UseParams(10, 10);
            Console.WriteLine("{0} Ohm", R);

            R = UseParams(10, 20, 30, 40, 50, 60, 70, 80, 90, 100);
            Console.WriteLine("{0} Ohm", R);
      }
}
```

Primjer 6.8

Ispis:
10 Ohm
5 Ohm
3,41417152147406 Ohm

Način na koji C# radi ovo s varijabilnim brojem argumenata vrlo je jednostavan. Vidimo da je u deklaraciji metode dovoljno navesti ključnu riječ *param* i nakon toga niz u kojem su sadržani podaci koji se prenose.

Parametar *param* može biti samo jedan i mora doći kao posljednji navedeni argument, kao npr. ovako:

```
public static double UseParams(int x, int y, params double[] Otpori)
```

ali ne i ovako:

```
public static double UseParams(params double[] Otpori, int x, int y)
```

Prevoditelj će jednostavno prenositi parametre jedan po jedan sve dok ne dođe do argumenta s parametrom *param*. U tom trenutku će sve ostale neprenesene argumente proslijediti kao niz.

Paralelni spoj otpornika računa se po formuli
`1/R = 1/R1 + 1/R2 + . . .`
i na početku metode provjeravamo je li možda proslijeđen nijedan (vraća se 0) ili samo jedan otpornik (vraća se njegova vrijednost). Za broj otpornika 2 ili više, ukupna vrijednost računa se u *for* petlji od vrijednosti 2 do vrijednosti *Otpori.Length* koja predstavlja ukupan broj otpornika.

6.6. Rekurzivne metode

Rekurzivne metode su metode koje pozivaju same sebe, i jedino u tome je njihova posebnost. Sve drugo za njih vrijedi kao i za ostale metode. Rekurzivne metode su vrlo elegantan način za rješavanje nekih zadataka, ali treba biti oprezan i osigurati napuštanje metode, kako metoda ne bi otišla u beskonačno pozivanje same sebe.

U četvrtom poglavlju napravili smo jedan primjer koji je računao faktorijel broja. Sada ćemo to isto napraviti korištenjem rekurzivne metode.

```
using System;

public class Test
{
	public static long Faktorijel(int n)
	{
		long result;

		if(n == 1)
			return 1;
		result = Faktorijel(n-1)* n;

		return result;
	}

	public static void Main()
	{
		int i = 10;
		Console.WriteLine("{0}! = {1}", i, Faktorijel(i));
	}
}
```

Primjer 6.9

Ispis:
10! = 3628800

Metoda *Faktorijel* poziva samu sebe i neće vratiti vrijednost sve dok ne izvrti pozive same sebe za sve vrijednosti od *n* do 1.

Svaki poziv metode, međutim troši i resurse i zato je nekad bolje koristiti standardne metode i problem riješiti korištenjem petlji. Pogledajmo jedan primjer koji se često navodi kod rekurzivnih metoda, a to je izračun Fibonaccijevog niza. Kod Fibonaccijevog niza svaki sljedeći broj jednak je zbroju prethodna dva.

Izračunat ćemo četrdeseti član niza na dva načina:

- rekurzivnom metodom *FiboRec* koristeći rekurzivne pozive metode
- standardnom metodom *Fibo* koristeći *while* petlju

Metode ćemo deklarirati kao *double* iako broj u Fibonaccijevom nizu može biti samo cijeli. Vrijednost brojeva u nizu brzo raste tako da smo ovako osigurali više brojeva koji se mogu izračunati.

```
using System;

public class Test
{
	public static double FiboRec(int n)
	{
		return (n == 0 || n == 1) ?  n : FiboRec(n-1) + FiboRec(n-2);
	}

	public static double Fibo(int n)
	{
		int PrviBroj = 0, DrugiBroj = 1;

		while(n-- != 0)
		{
			int Tmp = PrviBroj + DrugiBroj;
			PrviBroj = DrugiBroj;
			DrugiBroj = Tmp;
		}

		return PrviBroj;
	}

	public static void Main()
	{
		int i = 40;

		DateTime start = DateTime.Now;
		Console.Write("{0}. član = {1} ", i, FiboRec(i));
		DateTime end1 = DateTime.Now;
		TimeSpan Int1 = end1 - start;
		Console.WriteLine("izračunat za " + Int1.Seconds + "." +
					Int1.Milliseconds + " sec");

		DateTime end2 = DateTime.Now;
		Console.Write("{0}. član = {1} ", i, Fibo(i));
		TimeSpan Int2 = end2 - end1;
		Console.WriteLine("izračunat za " + Int2.Seconds + "." +
					Int2.Milliseconds + " sec");
	}
```

```
}
```

Primjer 6.10

Ispis:

```
40. član = 102334155 izračunat za 8.859 sec
40. član = 102334155 izračunat za 0.15 sec
```

U metodi *FiboRec* koristi se uvjetni izraz u kojem se provjerava je li *n* jednak 1 ili 2. Ako je jednak, onda se ta vrijednost i vraća, a ako ne, poziva se zbroj *FiboRec(n-1) + FiboRec(n-2)*. Obratite pozornost da će već prvi poziv metode *FiboRec* s desne strane uvjetnog izraza odmah ponovo pozvati metodu *FiboRec*.

Očita je razlika u vremenu izvršavanja, zato prije uporabe rekurzivnih metoda treba razmisliti je li možda bolje ipak koristiti standardni način i algoritam programa riješiti eventualno nekom petljom.

Poglavlje 7:
Ostali članovi klase

Ovo poglavlje obrađuje:

- konstante
- *readonly* član klase
- razliku između konstante i *readonly* člana
- objekt *this* i njegovu primjenu
- indekse
- preopterećenje operatora

7.1. Konstante

Konstanta je članica klase kojoj je kod deklaracije dodijeljena vrijednost i ne može se više promijeniti. Konstante mogu biti svi vrijednosni tipovi osim struktura plus *string* i *null* od referentnih tipova.
Konstante su implicitno *static* tako da se pozivaju navođenjem imena klase.

```
using System;

public class Planet
{
      public const double Gravity = 9.80665;
}

public class Test
{
      public static void Main()
      {
            double Masa = 100;
            double Tezina = Masa * Planet.Gravity;

            Console.WriteLine("Težina = " + Tezina + " N");
      }
}
```

Primjer 7.1

Ispis:
Težina = 980,665 N

Konstante se vrednuju u vremenu prevođenja, a mogu biti definirane drugom konstantom kao npr.

```
public const double x = 10;
public const double y = z + 5;
public const double z = x + 7;
```

ali naravno, ne i ovako da se uđe u krug:

```
public const double x = y;
public const double y = x;
```

Konstante mogu biti definirane i nekim izrazom:

```
public const double x = 10 * 10;
public const double y = 10 * Math.PI;
```

a može se kao i kod varijabli definirati više konstanti odjednom:

```
public const int z1 = 10, z2 = 20, z3 = 30;
```

Konstante imaju veliku primjenu u slučajevima kada neku konstantnu vrijednost koristimo na mnogo mjesta u programu. U slučaju potrebe za promjenom vrijednosti konstante, potrebno je promijeniti vrijednost na samo jednom mjestu i ta promjena će se iskazati na svim mjestima u programu gdje se ta konstanta koristi.
Pored toga, osigurani smo da se vrijednost konstante neće promijeniti tijekom izvođenja programa.

7.2. Readonly član

Konstante su dobre ako znamo njihovu vrijednost unaprijed, prije prevođenja. Ali neće uvijek biti takav slučaj, odnosno bit će situacija kada njihovu vrijednost moramo odrediti u vremenu izvođenja programa. Tada ćemo također željeti da njihova vrijednost ostane nepromijenjena nakon definiranja.

Za ovakve slučajeve postoji član klase *readonly*, kojem se vrijednost može dodijeliti u deklaraciji, ali i u izvođenju konstruktora i nakon toga ostaje nepromijenjena. Pogledajmo primjer sličan onom s gravitacijom kao konstantom, ali ovaj put koristeći *readonly* člana klase.

```
using System;

public class Planet
{
	private readonly double gravity;

	public Planet(double d)
	{
		gravity = d;
	}

	public double Gravity
	{
		get
		{
			return gravity;
		}
	}
}

public class Test
{
	public static void Main()
	{
		double Masa = 100;
		Planet Earth = new Planet(9.80665);
		Planet Jupiter = new Planet(22.94756);

		double TezinaZemlja = Masa * Earth.Gravity;
		double TezinaJupiter = Masa * Jupiter.Gravity;

		Console.WriteLine("Težina na Zemlji = {0} N",
		                                  TezinaZemlja);
		Console.WriteLine("Težina na Jupiteru = {0} N",
		                                  TezinaJupiter);
	}
}
```

Primjer 7.2

Ispis:
Težina na Zemlji = 980,67 N
Težina na Jupiteru = 2294,76 N

Gravitaciju određujemo tek kod kreiranja objekata jer je do tada i ne znamo budući da je različita za različite planete, te za razliku od konstante koja je pripadala klasi, *readonly* član pripada objektu, ali može biti i *static*.
Primijetili ste da izbjegavamo pojam "*readonly varijabla*" jer samim tim što se vrijednost ne može promijeniti, možda riječ varijabilan nije prikladna. Međutim, *readonly* članovi su po

svemu varijable kao i sve ostale, samo što im se vrijednost ne može promijeniti nakon definiranja.

7.3. Objekt this

Objekt *this* koji se poziva ključnom riječju *this* predstavlja trenutni objekt s kojim se radi. Jedan od razloga korištenja ključne riječi *this* je situacija kad je u konstruktoru potrebno dodijeliti vrijednosti članicama klase i to tako da im se dodjeljuju vrijednosti proslijeđene konstruktoru kao argumenti. Često je tu riječ o istim nazivima varijabli, pa *this* omogućuje lakše razaznavanje varijabli.

Kao npr. u sljedećem primjeru:

```
using System;

class Vrijeme
{
	private byte sat = 0, minuta = 0, sekunda = 0;

	public Vrijeme(byte sat, byte minuta, byte sekunda)
	{
		sat = sat;
		minuta = minuta;
		sekunda = sekunda;
	}

	public byte Sat
	{
		get {return sat;}
	}

	public byte Minuta
	{
		get {return minuta;}
	}

	public byte Sekunda
	{
		get {return sekunda;}
	}
}

public class Test
{
	public static void Main()
	{
		Vrijeme PetDoDvanaest = new Vrijeme(11, 59, 55);
		Console.WriteLine("{0}:{1}:{2}", PetDoDvanaest.Sat,
				PetDoDvanaest.Minuta, PetDoDvanaest.Sekunda);
	}
}
```

Primjer 7.3

Ispis:
0:0:0

Imamo klasu *Vrijeme* s tri varijable i 3 *readonly* svojstva. U funkciji *Main* kreirali smo objekt *PetDoDvanaest* te konstruktoru, kao što i samo ime objekta kaže, proslijedili vrijednosti

```
sat = 11
minuta = 59
sekunda = 55
```

ali kao što vidimo, pozivom svojstava ne ispisuje se to vrijeme nego vrijeme s inicijaliziranim vrijednostima.
Što se dogodilo?

Pogledamo li konstruktor, vidjet ćemo naredbe:

```
sat = sat;
minuta = minuta;
sekunda = sekunda;
```

S desne strane izraza očito je da su to argumenti proslijeđeni konstruktoru od pozivatelja, ali koje su varijable s lijeve strane, isti ti argumenti ili članice klase ?
Odgovor je, kao što vidimo iz rezultata ispisa, da su i s lijeve i desne strane izraza proslijeđeni argumenti tako da članice klase *sat, minuta* i *sekunda* ostaju nepromijenjeni. Rješenje ovakve situacije je korištenje ključne riječi *this* ispred varijabli s lijeve strane izraza:

```
this.sat = sat;
this.minuta = minuta;
this.sekunda = sekunda;
```

Sad će ispis biti onaj očekivani:
11:59:55

Ovakav slučaj može se izbjeći davanjem različitih imena argumentima konstruktora i članicama klase, ali s korištenjem *this* sve je potpuno jasno i čitljivo, a kao što vidimo i izvršava se na očekivani način.

Poziv članica klase preko ključne riječi *this* moguć je samo u konstrukturu, metodama i svojstvima. Ovo nije dozvoljeno:

```
private byte milisekunda = this.sekunda * 1000;
```

Isto tako, *this* se ne može koristiti unutar statičkih metoda i svojstava.

Drugi primjer korištenja *this* je u slučaju ako želimo proslijediti trenutni objekt kojeg *this* predstavlja, npr. metodi neke druge klase.

```
using System;

class Vrijeme
{
	private byte sat = 0, minuta = 0, sekunda = 0;

	public Vrijeme(byte sat, byte minuta, byte sekunda)
	{
		this.sat = sat;
		this.minuta = minuta;
		this.sekunda = sekunda;
	}
```

```
        public void SetTime(Time t)
        {
                t.ChangeTime(this);
        }

        public byte Sat {get {return sat;}}
        public byte Minuta {get {return minuta;}}
        public byte Sekunda {get {return sekunda;}}
}

class Time
{
        private byte sat = 0, minuta = 0, sekunda = 0;

        public void ChangeTime(Vrijeme v)
        {
                sat = v.Sat;
                minuta = v.Minuta;
                sekunda = v.Sekunda;
                Console.WriteLine("{0}:{1}:{2}", Sat, Minuta, Sekunda);
        }

        public byte Sat {get {return sat;}}
        public byte Minuta {get {return minuta;}}
        public byte Sekunda {get {return sekunda;}}
}

public class Test
{
        public static void Main()
        {
                Vrijeme PetDoDvanaest = new Vrijeme(11, 59, 55);
                Time NoTime = new Time();
                PetDoDvanaest.SetTime(NoTime);
        }
}
```

Primjer 7.4

Ispis:
11:59:55

Imamo dvije klase: *Vrijeme* i *Time*. U funkciji *Main* definiramo dva objekta *PetDoDvanaest* tipa *Vrijeme* i *NoTime* klase *Time*. Zbog uštede u prostoru svojstva su definirana malo nepreglednije, ali riječ je o istim onim kao i u prethodnom primjeru.

Prenošenje objekta događa se dvaput, prvi put se prenosi objekt *NoTime* iz funkcije *Main* metodi *SetTime* klase *Vrijeme*, a drugi put ista ta metoda prenosi preko ključne riječi *this* objekt tipa *Vrijeme* kojemu i sama pripada. To je zapravo objekt *PetDoDvanaest* kreiran u funkciji *Main*.

Metoda *ChangeTime* taj preneseni objekt koristi za definiranje svojih članica *sat*, *minuta* i *sekunda*. Ovakvim prenošenjem objekta po referenci možemo jednako efikasno prenositi objekte bez obzira na njihovu veličinu. Metoda *ChangeTime*, budući da je dobila *this* kao referencu na objekt, mogla bi promijeniti i objekt *this*, odnosno njegovo vrijeme. U ovom slučaju ipak ne bi mogla, jer bi za to trebalo definirati i *set* svojstva u klasi *Vrijeme*.

7.4. Indeksi (Indexers)

Treća primjena ključne riječi *this* je u korištenju *indeksa*. *Indeksi* omogućuju interpretaciju objekta kao niz. Pogledajmo klasu *ComboBox* koja bi ovako bila definirana:

```
class ComboBox
{
	private int boja, visina, sirina;
	private string [] stavke = new string [100];

	public ComboBox(string [] stavke)
	{
		int i = 0;
		foreach(string s in stavke)
			this.stavke[i++] = s;
	}
}
```

i instancu klase

```
ComboBox cb = new ComboBox(str);
```

Ono što nas najviše zanima u *ComboBoxu* su njegove stavke i ako bismo trebali pristupiti trećoj stavci, to bismo trebali raditi ovako:

```
cb.stavke[2]
```

Bilo bi puno bolje i intuitivnije s obzirom da su stavke ono što je glavno u *ComboBoxu*, da im se pristupa na način da se cijeli objekt *cb* interpretira kao niz. Tako bismo trećoj stavci odnosno stringu *ComboBoxa* pristupili jednostavno kao:

```
cb[2]
```

Indeksi kao članovi klase služe upravo ovakvoj svrsi, a sintaksa im je slična kao i svojstvima. Sličan je i zadatak koji obavljaju, vraćanje ili postavljanje vrijednosti određenog člana klase. Samo što će svojstva to raditi na varijabli koja nije tipa niz, dok će indeksi to raditi uglavnom na nekom nizu koji je član klase.
Napravimo sada jedan primjer u kojem ćemo pojedinim stavkama *ComboBoxa* pristupati preko indeksa.

```
using System;

class ComboBox
{
	private string [] stavke = new string [100];

	public ComboBox(string [] stavke)
	{
		int i = 0;
		foreach(string s in stavke)
			this.stavke[i++] = s;
	}

	public string this[int ind]
	{
		get
		{
			return (ind >= 0 || ind <= 100) ?  stavke[ind] : null;
		}
```

```
            set
            {
                  if(ind >= 0 || ind <= 100)
                        stavke[ind] = value;
            }
      }
}

public class Test
{
      public static void Main()
      {
            string[] str = new string[] {"First", "Second", "Third",
                                         "Fourth", "Fifth"};
            ComboBox cb = new ComboBox(str);
            Console.WriteLine(cb[4]);
            cb[4] = "Peti";
            Console.WriteLine(cb[4]);
      }
}
```

Primjer 7.5

Ispis:
Fifth
Peti

Indeksi se definiraju ključnom riječju *this*, a kao svojstva mogu biti *readonly*, *writeonly* ili *read/write* ovisno koji od pristupnika *get* i *set* su definirani.
Naredba

```
cb[4] = "Peti";
```

će koristeći *set* pristupnik postaviti vrijednost petog člana od člana klase *stavke*, a

```
Console.WriteLine(cb[4]);
```

će koristeći *get* pristupnik dati njegovu vrijednost.

Budući da indeksi predstavljaju točno određeni objekt, oni ne mogu biti deklarirani kao *static*. Indeksi se mogu koristiti i kao članice sučelja te mogu biti virtualni i apstraktni ako se koristi nasljeđivanje. S obzirom da je takva izvedba slična kao i kod metoda, pogledajte u poglavlju o klasama više o tome.

Vidjeli smo jednostavnost pristupa pojedinom stringu *ComboBoxa*, ali što ako imamo niz objekata tipa *ComboBox*. Što bi bio rezultat izvršenja ovog kôda?

```
ComboBox [] cb = new ComboBox [3];
. . .
Console.WriteLine(cb[2]);
```

Što ovdje predstavlja *cb[2]*?
Treći objekt u nizu ili treći string u objektu?

Odgovor je ono prvo, treći objekt u nizu. Stoga bi ovdje bila primijenjena virtualna metoda *ToString* i bilo bi ispisano *ComboBox*. Pogledajmo cijeli primjer:

```
using System;

class ComboBox
{
```

```
        private string [] stavke = new string [100];

        public ComboBox(string [] stavke)
        {
                int i = 0;
                foreach(string s in stavke)
                        this.stavke[i++] = s;
        }

        public string this[int ind]
        {
                get
                {
                        return (ind >= 0 || ind <= 100) ?  stavke[ind] : null;
                }

                set
                {
                        if(ind >= 0 || ind <= 100)
                                stavke[ind] = value;
                }
        }
}

public class Test
{
        public static void Main()
        {
                string[] str1 = new string[] {"First", "Second", "Third"};
                string[] str2 = new string[] {"Fourth", "Fifth", "Sixth"};
                string[] str3 = new string[] {"Seventh", "Eight", "Ninth"};

                ComboBox [] cb = new ComboBox [3];
                cb[0] = new ComboBox(str1);
                cb[1] = new ComboBox(str2);
                cb[2] = new ComboBox(str3);

                Console.WriteLine(cb[1][2]);
                cb[1][2] = "Šesti";
                Console.WriteLine(cb[1][2]);
        }
}
```

Primjer 7.6

Ispis:
Sixth
Šesti

Definirali smo tri objekta tipa *ComboBox* i svaki od njih ima svoje indekse. Konstruktorima objekata proslijeđeni su stringovi *str1*, *str2*, i *str3*.
Primjećujemo da prvi operator [] uz objekt označava redni broj objekta u nizu objekata, a drugi [] predstavlja indeks.
Zato kad napišemo:

```
cb[1][2]
```

to se odnosi se na treću (3.) stavku od drugog (2.) člana niza objekata. Ne treba naglašavati da se i ovdje koristi 0 indeksacija gdje prvi član ima oznaku 0.

Indeksi sa stringovima

Indeksi ne moraju nužno biti ograničeni samo na cjelobrojne varijable, iako se kao takvi najviše koriste. Kao indeks možemo definirati i *string* što ponekad i ima smisla. Sada ćemo napraviti klasu koja bi sadržavala između ostalog niz stringova engleskih i hrvatskih riječi i koja bi mogla poslužiti za izradu nekakve vrste e-rječnika.
Naravno, u klasi ćemo staviti naglasak na indekse, a ne na izradu e-rječnika.

```
using System;

class EnglishToCroatian
{
	private string [] English = new string [100];
	private string [] Hrvatski = new string [100];

	public EnglishToCroatian(string [] English, string [] Hrvatski)
	{
		int i = 0;
		foreach(string s in English)
			this.English[i++] = s;

		i = 0;
		foreach(string s in Hrvatski)
			this.Hrvatski[i++] = s;
	}

	public string this[string str]
	{
		get
		{
			int i = 0;
			foreach(string s in English)
				if(s == str)
					return(Hrvatski[i]);
				else
					i++;

			return "riječ nije pronađena";
		}
	}
}

public class Test
{
	public static void Main()
	{
		string[] str1 = new string[] {"capacity", "current",
					"inductance", "resistance", "voltage",};
		string[] str2 = new string[] {"kapacitet", "struja",
					"induktivitet", "otpor", "napon"};

		EnglishToCroatian Dict = new EnglishToCroatian(str1, str2);
		Console.WriteLine("voltage" + " - " + Dict["voltage"]);
	}
}
```

Primjer 7.7

Ispis:
voltage – napon

Kreiran je objekt tipa *EnglishToCroatian* i preko konstruktora definiraju se nizovi stringova s engleskim i hrvatskim riječima.
Indeks je definiran kao string što je jedino ispravno za ovakav primjer. Očita je razlika između ovih dvaju poziva:

```
Dict["voltage"]
```
i
```
Dict[4]
```

Prvi način je jasniji i predstavlja upravo ono kako bismo riječ tražili u tiskanom rječniku. Što se tiče implementacije indeksa, nije se puno promijenilo u deklaraciji, umjesto *int* pišemo *string*, a povratna vrijednost u oba slučaja bila bi *string*. Najveća je razlika u definiciji pristupnika *get*, prolazi se s *foreach* petljom kroz sve riječi i kad se pronađe proslijeđena engleska riječ, vraća se odgovarajuća hrvatska.

Pristupnik *set* nije definiran, ali on bi u svojoj implementaciji također kao i pristupnik *get* sekvencijalno prošao kroz niz koristeći *foreach* petlju, te u slučaju pronalaska riječi redefinirao odgovarajuću hrvatsku:

```
Dict["voltage"] = "razlika potencijala";
```

Zanemarimo da je za ovakav slučaj puno efikasnije koristiti algoritam binarnog pretraživanja umjesto ovakvog sekvencijalnog.
Ipak, evo i takvog rješenja s binarnim pretraživanjem.

```
public string this[string str]
{
	get
	{
		int Low = 0, High = Num - 1, Middle;

		while(Low <= High)
		{
			Middle = (Low + High) / 2;
			if(String.Compare(str, English[Middle]) > 0)
				Low = Middle + 1;
			else if(String.Compare(str, English[Middle]) < 0)
				High = Middle-1;
			else
				return Hrvatski[Middle];
		}
		return "Riječ nije pronađena";
	}
}
```

Uvjet za binarno pretraživanje su abecedno poredani stringovi. Algoritam je izveden tako da se varijabla *Middle* pozicionira po listi, i to najprije na polovicu liste. Ako riječ nije pronađena varijable *Low* i *High* se povećavaju i smanjuju ovisno je li riječ pronađena na prethodnom mjestu manja ili veća od tražene. I tako sve dok se riječ ne pronađe ili ako se uopće ne pronađe vraća se string da riječ nije pronađena.

Pretraživanje se radi već opisanom metodom *Compare* klase string. Varijabla *Num* predstavlja ukupan broj riječi u stringovima *English* i *Hrvatski* i najjednostavnije ju je definirati u konstruktoru u jednoj od *foreach* petlji. (Pazite, u ovakvom primjeru *English.Length* će vratiti

100!) Ovakav način pretraživanja puno je brži od sekvencijalnog, ali razlika se vidi tek kod većeg broja elemenata.

Na kraju da zaključimo, indeksi su dobra stvar, ali ne treba ih implementirati pod svaku cijenu, odnosno treba ih koristiti samo kad njihovo korištenje "ima smisla".

7.5. Preopterećeni operatori (Overloaded operators)

Za ugrađene vrijednosne tipove kao što su *int*, *long*, *double* itd. sasvim je razumljivo i od strane programera očekivano da može napisati ovako nešto:

```
int x = 5, y = 6, z;
z = x + y;
```

ili

```
if (x < y)
. . .
```

Na žalost ovako nešto ne možemo napraviti i za klase za koje bi možda također imalo smisla primijeniti operatore na ovakav način.
Na sreću C# nudi način za takvo nešto s tzv. preopterećenjem operatora (*operator overloading*). Preopterećeni operatori su članovi klase koji predefiniraju korištenje standardnih operatora na način da budu smisleno primijenjeni na objekte te klase. Ako imamo npr. definiranu klasu kompleksnih brojeva koja bi imala dvije glavne članice (naravno bile bi bi tu i druge članice, ali ove dvije su najvažnije)

```
private double real;
private double imagine;
```

bilo bi poželjno da za objekte te klase možemo primijeniti standardne operatore za zbrajanje, oduzimanje ili uspoređivanje kompleksnih brojeva.

Napravimo jedan takav primjer klase *KompleksniBrojevi* i nekoliko preopterećenih operatora kao članova klase:

```
using System;

class KompleksniBroj
{
      private double Real;
      private double Imagine;

      public KompleksniBroj(double Real, double Imagine)
      {
            this.Real = Real;
            this.Imagine = Imagine;
      }

      public static KompleksniBroj operator +(KompleksniBroj i1,
                              KompleksniBroj i2)
      {
            KompleksniBroj x = new KompleksniBroj(i1.Real + i2.Real,
                              i1.Imagine + i2.Imagine);
            return x;
```

```
        }

        public static KompleksniBroj operator -(KompleksniBroj i1,
                                KompleksniBroj i2)
        {
              KompleksniBroj x = new KompleksniBroj(i1.Real - i2.Real,
                                i1.Imagine - i2.Imagine);
              return x;
        }

        public static bool operator ==(KompleksniBroj i1, KompleksniBroj i2)
        {
              return ((i1.Real * i1.Real + i1.Imagine * i1.Imagine) ==
                    (i2.Real * i2.Real + i2.Imagine * i2.Imagine)
                          ? true : false);
        }

        public static bool operator !=(KompleksniBroj i1, KompleksniBroj i2)
        {
              return ((i1.Real * i1.Real + i1.Imagine * i1.Imagine) !=
                    (i2.Real * i2.Real + i2.Imagine * i2.Imagine)
                          ? true : false);
        }

        public static bool operator <(KompleksniBroj i1, KompleksniBroj i2)
        {
              return ((i1.Real * i1.Real + i1.Imagine * i1.Imagine) <
                    (i2.Real * i2.Real + i2.Imagine * i2.Imagine)
                          ? true : false);
        }

        public static bool operator >(KompleksniBroj i1, KompleksniBroj i2)
        {
              return ((i1.Real * i1.Real + i1.Imagine * i1.Imagine) >
                    (i2.Real * i2.Real + i2.Imagine * i2.Imagine)
                          ? true : false);
        }

        public static KompleksniBroj operator ++(KompleksniBroj x)
        {
              return (new KompleksniBroj(x.Real+1, x.Imagine+1));
        }

        public static KompleksniBroj operator --(KompleksniBroj x)
        {
              return (new KompleksniBroj(x.Real-1, x.Imagine-1));
        }

        public override string ToString()
        {
              return Real.ToString() + " + " + Imagine.ToString() + "i";
        }
}

public class Test
{
        public static void Main()
        {
              KompleksniBroj KB1 = new KompleksniBroj(30, 40);
              KompleksniBroj KB2 = new KompleksniBroj(10, 25);
              KompleksniBroj KB3 = KB1 + KB2;
```

```
            Console.WriteLine(KB3);
            KB3++;
            Console.WriteLine(KB3);
            Console.WriteLine(KB1 - KB2);
            Console.WriteLine(KB1 == KB2);
        }
}
```

Primjer 7.8

Ispis:
40 + 65i
41 + 66i
20 + 15i
False

Klasa *KompleksniBroj* ima dvije varijable, *Real* koja predstavlja realni dio kompleksnog broja i *Imagine* koja predstavlja imaginarni dio. Klasa definira nekoliko operatora za rad s kompleksnim brojevima `+, -, ==, !=, < , >, ++,` i `--`.
Implementacija svakog od njih izvedena je kao što se i radi s kompleksnim brojevima. Zato zbog jednostavnosti nisu definirani operatori množenja i dijeljenja, jer množenje kompleksnih brojeva nije jednostavno množenje dvaju realnih i dvaju imaginarnih brojeva.

Operator + tako definira zbrajanje kompleksnih brojeva kao zasebno zbrajanje realnih i imaginarnih dijelova. S tako dobivena dva zbroja kreira se objekt tipa *KompleksniBroj* i vraća kao rezultat.
Operatori < i > vraćaju *bool* tip *true* ili *false* ovisno koji je broj veći, a kao veličina kompleksnog broja uzima se njegova apsolutna vrijednost.
Operatori == i `!=` provjeravaju jesu li kompleksni brojevi jednaki usporedbom i realnog i imaginarnog dijela.
Operatori ++ i – su unarni operatori pa kao argument uzimaju samo jedan objekt na kojemu je povećanje odnosno smanjenje za jedan izvedeno povećanjem i smanjenjem i realne i imaginarne vrijednosti.

U klasi je predefinirana i virtualna metoda *ToString* da bi ispisala kompleksni broj u uobičajenom obliku:
`real + i * imag`

Vidimo da je u deklaraciji preopterećenog operatora potrebno navesti ključnu riječ *operator* te simbol tog operatora. Iako se većina operatora može preopteretiti (predefiniranje se odnosi samo za tu klasu), svi operatori se ipak ne mogu preopteretiti.

Sljedeći operatori ne mogu se preopteretiti:
`=  &&  ||  ?:  new  typeof  sizeof  is`

`[] i ()` se također ne mogu preopteretiti ali se isti efekt može dobiti indeksima, odnosno preopterećenjem operatora *implicit* i *explicit*

Preopterećeni operatori moraju se deklarirati kao *static*, ne mijenja im se prioritet s preopterećenjem i strogo se **ne preporučuje** preopterećenje operatora na način koji nema smisla. Ako ovako napišemo:

```
KompleksniBroj KB = new KompleksniBroj(20, 30);
```

```
KB++
```

onda je za očekivati da ovo poveća vrijednost realnog dijela na 21, a imaginarnog na 31, a ne da ih eventualno smanji, iako bi se operator ++ moglo natjerati da radi smanjivanje.
Jednako tako ne treba preopterećivati operatore ako klasa po svojoj definiciji nema potrebe ni smisla za takvom operacijom. Jednako kao što smo rekli i kod indeksa, to treba napraviti samo ako "ima smisla".

C# prevoditelj obvezuje na uparivanje preopterećenih operatora. Ako je preopterećen operator <, onda mora biti i operator >, odnosno ako je preopterećen == onda mora biti i !=.
Iako to nije obveza, ako su operatori == i != preopterećeni, preporučuje se i predefiniranje virtualnih metoda *Equals* i *GetHashCode*. U slučaju da se to ne napravi, prevoditelj će javiti upozorenje.

S obzirom da vrijednost kompleksnog broja predstavlja neki broj i to je njegova apsolutna vrijednost, bilo bi zgodno da tu vrijednost možemo implicitno dodijeliti nekoj *double* varijabli. Ali ako ovako napišemo:

```
KompleksniBroj KB = new KompleksniBroj(60, 80);
double d = KB;
```

to se neće prevesti i prevoditelj će javiti da ne postoji implicitna pretvorba tipa *KompleksniBroj* u *double*. Rješenje je definirati operator implicitne pretvorbe koji će to napraviti:

```
public static implicit operator double (KompleksniBroj x)
{
      return Math.Sqrt(x.Real * x.Real + x.Imagine * x.Imagine);
}
```

Napišemo li:

```
KompleksniBroj KB = new KompleksniBroj(60, 80);
double d = KB;
Console.WriteLine(d);
```

program će ispisati *100* jer je to vrijednost broja dobivena kao drugi korijen zbroja kvadrata realnog i imaginarnog dijela.

Pitanje:
Što bi se ispisalo nakon izvršenja ovoga?

```
KompleksniBroj KB = new KompleksniBroj(60, 80);
Console.WriteLine(KB);
```

a) u slučaju da **je** definiran operator za implicitnu pretvorbu iz klase *KompleksniBroj* u *double*
b) u slučaju da **nije** definiran operator za implicitnu pretvorbu iz klase *KompleksniBroj* u *double*

Odgovor:

a) **100** jer bi se dogodila implicitna pretvorba *KB* u *double* i metoda *WriteLine* bi pozvala predefiniranu virtualnu metodu *ToString* iz *System.Double* koja *double* pretvara u string prezentaciju broja
b) **60 + 80i** jer bi se također pozvala virtualna metoda *ToString*, ali ovaj put ona iz klase *KompleksniBroj*

Poglavlje 8:
Nizovi

Ovo poglavlje obrađuje:

- deklaraciju i definiciju nizova
- vrste nizova i razliku među njima
- klasu *Array*
- klasu *ArrayList*
- implementaciju naredbe *foreach* u nizovima
- sučelja *IEnumerable* i *IEnumerator*

8.1. Jednodimenzionalni nizovi

Nizove (*Arrays*) smo već koristili u našim primjerima, a sada evo i definicije. Dakle, niz je referentni tip podataka izveden od apstraktne klase *System.Array*, a predstavlja skup (niz) podataka odnosno varijabli istog tipa.
Nizovi mogu biti:

- Jednodimenzionalni
- Višedimenzionalni
- Nazubljeni (*jagged*) ili nizovi nizova

Sintaksa deklaracije najjednostavnijeg jednodimenzionalnog niza je:

```
tip [] identifikator
```

Možemo vidjeti da je C# promijenio raspored uglatih zagrada u odnosu na C/C++ kod kojih su zagrade dolazile iza identifikatora.

Sama definicija jednodimenzionalnog niza može se izvesti na nekoliko različitih načina:

```
int [] Arr = new int[] {1, 2, 3, 4, 5, 6, 7};

int [] Arr = {1, 2, 3, 4, 5, 6, 7};

int [] Arr = new int[7];
for(int i=0;i<7;i++)
      Arr[i] = i+1;
```

Sva tri načina će kreirati referentni tip *Arr* koji sadrži 7 *int* varijabli, dakle 7 vrijednosnih tipova. Ovdje treba razlikovati varijablu *Arr* koja je objekt tipa `System.Int32[]` i čiji je sadržaj referenca na ovih 7 vrijednosnih tipova koji su članovi niza.
Zbog ovakve razdvojenosti niza i njegovih članova moguće je dinamički promijeniti veličinu niza.

```
int [] Arr = new int[7];
Arr = new int[14];
```

Također, može se vidjeti da se članovima niza pristupa koristeći nultu indeksaciju:

```
Arr[0] = Arr[1];
```

a može se koristiti i *foreach* petlja

```
foreach(int i in Arr)
      Console.WriteLine(i);
```

Hoće li niz sadržavati vrijednosne ili referentne tipove kao svoje članove ovisi o tipu niza jer niz može biti i ovako deklariran:

```
ClassX [] Arr = new ClassX[7];
```

Budući da su članovi niza objekti klase *ClassX* potrebno ih je i kreirati, svakog posebno s ključnom riječi *new*:

```
for(int i = 0;i<7;i++)
      Arr[i] = new ClassX();
```

Kod kreiranja nizova članovima niza se dodjeljuju njihove podrazumijevane vrijednosti, a to je za tip *int* 0, a za *ClassX null*. Za podrazumijevane vrijednosti svih tipova pogledajte tablicu u poglavlju o konstruktorima. Zato će se nakon izvršenja ovoga:

```
int [] Arr1 = new int[7];
ClassX [] Arr2 = new ClassX[7];

for(int i=0;i<7;i++)
      if(Arr2[i] == null)
            Console.WriteLine(Arr1[i]);
```

ispisati sedam puta 0 jer je to vrijednost članova *Arr1*, dok je *null* vrijednost svih članova *Arr2*. Ako prije *for* petlje umetnemo ovaj kôd:

```
for(int i = 0;i<7;i++)
      Arr2[i] = new ClassX();
```

neće se ništa ispisati jer će sada pozivom konstruktora biti kreirani objekti i u *Arr2* više niti jedan član neće biti *null*.

Klasa System.Array

Kao što smo rekli, nizovi nasljeđuju apstraktnu klasu *System.Array*, a s njom i njene metode i svojstva. Jedna od njih je i statička metoda *BinarySearch* čiju smo vlastitu implementaciju napravili u poglavlju o indeksima. Koristeći tu metodu, sad sve može stati u jednu naredbu:

```
string[] str1 = new string[] {"capacity", "current",
                        "inductance", "resistance", "voltage",};
string[] str2 = new string[] {"kapacitet", "struja",
                        "induktivitet", "otpor", "napon"};

string Hrv = str2[Array.BinarySearch(str1, "voltage")];
```

U stringu *Hrv* bit će string “napon”.
Metoda *BinarySearch* vraća indeks pronađenog stringa ili negativnu vrijednost ako string nije pronađen. Drugi argument u deklaraciji te metode je *object*.

```
public static int BinarySearch(Array, object)
```

pa se ona može koristiti i za ostale tipove podataka.

```
int [] Arr = new int[] {1, 3, 5, 7, 9, 11};
Arr[Array.BinarySearch(Arr, 5)] = 50;
```

Nakon izvršenja ovog kôda u *Arr* bili bi članovi
1, 3, 50, 7, 9, 11,
jer bi metoda *BinarySearch* vratila broj 2 kao indeks na kojem se nalazi traženi broj 5.

Pogledajmo jedan primjer u kojem ćemo kreirati dva niza s istim podacima koristeći jednodimenzionalni niz i klasu *Array*:

```
using System;

public class Test
{
```

```
        public static void Main()
        {
              string[] RedniBrojevi1 = new string[] {"First", "Second",
                          "Third", "Fourth", "Fifth"};
              Array.Sort(RedniBrojevi1);

              for(int i=0; i < RedniBrojevi1.Length;i++)
                    Console.Write(RedniBrojevi1[i] + "  ");
              Console.WriteLine();

              Array RedniBrojevi2 = Array.CreateInstance(typeof(String), 5);
              RedniBrojevi2.SetValue("First", 0);
              RedniBrojevi2.SetValue("Second", 1);
              RedniBrojevi2.SetValue("Third", 2);
              RedniBrojevi2.SetValue("Fourth", 3);
              RedniBrojevi2.SetValue("Fifth", 4);
              Array.Reverse(RedniBrojevi2);

              for(int i=0; i < RedniBrojevi2.Length;i++)
                    Console.Write(RedniBrojevi2.GetValue(i) + "  ");
        }
}
```

Primjer 8.1

Ispis:
Fifth First Fourth Second Third
Fifth Fourth Third Second First

U prvom slučaju kreirali smo jednodimenzionalni niz *RedniBrojevi1* koji sadrži pet stringova. Koristeći statičku metodu *Sort* klase *Array* koju ovaj niz nasljeđuje, stringovi su sortirani, a svojstvo *Length* iste klase *Array* koristi se u *for* petlji da bi se dobio broj elemenata u nizu.

U drugom slučaju niz *RedniBrojevi2* kreirali smo metodom *Array.CreateInstance* kojoj kao argument prosljeđujemo tip niza te njegovu veličinu. Nakon toga metodom *SetValue* definiraju se članovi niza, a prije ispisa metoda *Reverse* ih okrene tako da posljednji string postane prvi itd…

Sljedeća tablica navodi članice klase *System.Array*

Metoda	**Opis**
`BinarySearch`	Pretražuje jednodimenzionalni niz koristeći *BinarySearch* algoritam
`Clear`	Briše sve članove niza i postavlja granice na 0
`Clone`	Kreira kopiju niza
`Copy`	Kopira dio članova niza u drugi niz izvodeći *cast* i *boxing*
`CopyTo`	Kopira sve članove jednodimenzionalnog niza u drugi startavši od određenog indeksa
`CreateInstance`	Kreira novu instancu *Array* klase
`GetEnumerator`	Vraća tip sučelja *IEnumerator* za niz
`GetLength`	Vraća broj članova niza
`GetLowerBound`	Vraća donju granicu niza

`GetUpperBound`	Vraća gornju granicu niza
`GetValue`	Vraća vrijednost određenog člana niza
`IndexOf`	Vraća indeks prvog pronađenog člana s određenom vrijednošću
`Initialize`	Inicijalizira sve vrijednosne članove niza pozivom podrazumijevanog (default) konstruktora
`LastIndexOf`	Vraća indeks posljednjeg pronađenog člana s određenom vrijednošću
`Reverse`	Okreće string tako da posljednji član postaje prvi itd…
`SetValue`	Postavlja vrijednost za određeni član niza
`Sort`	Sortira članove niza
Svojstvo	**Opis**
`IsFixedSize`	Vraća tip *bool* koji određuje je li niz fiksne veličine
`IsReadOnly`	Vraća tip *bool* koji određuje je li se niz može samo čitati
`IsSynchronized`	Vraća tip *bool* koji određuje je li pristup nizu siguran po pitanju izvršavanju u niti (*Thread*)
`Length`	Vraća broj članova niza u svim dimenzijama
`Rank`	Vraća broj dimenzija u nizu
`SyncRoot`	Vraća objekt koji se može iskoristiti za sinkroniziran pristup nizu

8.2. Višedimenzionalni nizovi

Nizovi mogu biti i višedimenzionalni. Kreiranje dvodimenzionalnog niza s npr. 5 redova i 2 kolone napravilo bi se ovako:

```
int[,] TwoDimensions = new int[5, 2];
```

a trodimenzionalnog s dimenzijama 5, 4 i 3 u svakoj od tri dimenzije:

```
int[,,] ThreeDimensions = new int[5, 4, 3];
```

Kao i kod jednodimenzionalnih nizova i kod višedimenzionalnih članovi niza mogu se definirati na različite načine:

```
int[,] Arr2 = new  int[,] {{1,3}, {2,4}, {5,7}, {6, 8}, {9, 11}};
int[,] Arr2 = {{1,3}, {2,4}, {5,7}, {6, 8}, {9, 11}};
```

Dvodimenzionalnom nizu pristupa se tako da se najprije navede indeks reda, pa onda indeks kolone željenog podatka:

`Arr2[3,1]` je 8,
`Arr2[4,0]` je 9

Napravimo sad jedan primjer dvodimenzionalnog niza koji će predstavljati tablicu množenja.

```
using System;

class TablicaM
{
	private int[,] Tab;
	private int DimX, DimY;

	public TablicaM(int x, int y)
	{
		DimX = x;
		DimY = y;
		Tab = new int[x, y];

		for(int i=0;i<x;i++)
			for(int j=0;j<y;j++)
				Tab[i, j] = (i+1) * (j+1);
	}

	public int Pomnozi(int x, int y)
	{
		if(x >= 1  &&  x <= DimX  && y >= 1  &&  y <= DimY)
			return Tab[x-1, y-1];
		else
			return 0;
	}
}

public class Test
{
	public static void Main()
	{
		TablicaM x = new TablicaM(100, 100);
		Console.WriteLine(x.Pomnozi(47, 74));
	}
}
```

Primjer 8.2

Ispis:
3478

Definirali smo klasu *TablicaM* koja sadrži varijable:

- dimenzije niza: *DimX*, *DimY*
- dvodimenzionalni niz: *int[,] Tab*

S obzirom da u vremenu prevođenja ne znamo dimenzije niza, one se određuju prosljeđivanjem argumenata konstruktoru u kojem se onda kreira objekt *Tab*.

```
Tab = new int[x, y];
```

Koristeći dvostruku *for* petlju niz inicijaliziramo s vrijednostima koje će biti umnožak kolone i reda, ali zbog 0 indeksacije kolona i red će biti uvećani za jedan. Zato metoda *Pomnozi* koja

vraća umnožak dva broja vraća član niza s indeksima manjim za jedan od brojeva koje treba pomnožiti.

```
return Tab[x-1, y-1];
```

Dvodimenzionalni niz također nasljeđuje neka svojstva i metode od klase *Array* kao npr.

```
Tab.Length, broj članova u obje dimenzije (10000)
Tab.GetLength(0), veličina prve dimenzije (100)
Tab.GetLength(1), veličina druge dimenzije (100)
```

Niz se može proslijediti metodi kao argument i tu nema nikakve razlike u odnosu na ostale varijable. Metoda koja prosljeđuje niz, navodi kao argument samo identifikator, a metoda koja prima niz kao argument mora u svojoj deklaraciji imati i ispravnu deklaraciju tog niza: Definiramo li u klasi *TablicaM* ove dvije metode

```
public void F1()
{
      F2(Tab);
}

public void F2(int[,] t)
{
      Console.WriteLine(t[20, 30]);
}
```

onda bi nakon poziva *F1* ta metoda proslijedila cijeli niz zajedno sa svim njegovim članovima pa bi nakon takvog poziva ispis bio 651 (zato što je 21 x 31 = 651, a to su 20. red i 30. kolona.)

8.3. Nazubljeni nizovi (Jagged array)

Posebna vrsta višedimenzionalnih nizova su nazubljeni (*jagged*) nizovi. Obične višedimenzionale nizove možemo predstaviti u obliku pravokutnika jer svaki "red" u nizu ima jednaki broj članova. Za razliku od njih, nazubljeni nizovi imaju različit broj članova u svakom redu i često se zovu i nizovi nizova.

Deklaracija i definicija nazubljenog niza od 3 niza:

```
int[][] Arr = new int[3][];
```

Ova naredba je kreirala 3 jednodimenzionalna niza s različitim brojem članova i svakog od njih moramo posebno definirati:

```
Arr[0] = new int[7];
Arr[1] = new int[5];
Arr[2] = new int[9];
```

Za svaki od ova tri niza sada vrijedi sve ono što i za svaki jednodimenzionalni niz. Sljedeći primjer pokazuje korištenje nazubljenih nizova.

```
using System;

public class Test
```

```
{
	public static void Main()
	{
		sbyte[][] Arr = new sbyte[3][];
		Arr[0] = new sbyte[7];
		Arr[1] = new sbyte[5];
		Arr[2] = new sbyte[9];

		sbyte x = 1;
		for(int i = 0; i<Arr.Length; i++)
			for(int j=0; j<Arr[i].Length; j++)
				Arr[i][j] = x++;

		for(int i = 0; i<Arr.Length; i++)
			Array.Reverse(Arr[i]);

		for(int i = 0; i<Arr.Length; i++)
		{
			for(int j=0; j<Arr[i].Length; j++)
				Console.Write(Arr[i][j] + " ");
			Console.WriteLine();
		}
	}
}
```

Primjer 8.3

Ispis:

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

Primjer kreira nazubljeni niz koji se sastoji od tri jednodimenzionalna niza veličina 7, 5 i 9 članova. U svaki od tih članova dodajemo za jedan veću vrijednost što je osigurano varijablom *x* i njenim povećavanjem za 1 nakon svakog dodjeljivanja. Tako će prvi član drugog niza (niz *Arr[1]*) imati za jedan veću vrijednost od posljednjeg člana prvog niza (niz *Arr[0]*), a isto vrijedi i za treći niz (niz *Arr[2]*).

Pojedinom članu niza pristupamo sintaksom:

```
Arr[i][j]
```

gdje je:

- *i* - indeks niza u nazubljenom nizu
- *j* – indeks člana u *i*-tom nizu

Nakon definiranja nizovima se okrenu vrijednosti koristeći statičku metodu *Reverse* te se na kraju te vrijednosti ispišu.
Vidimo da sva 4 niza (nazubljeni i 3 jednodimenzionalna) koriste nasljeđeno svojstvo *Length*, ali kod nazubljenog to svojstvo vraća broj nizova (3), a kod njegovih članova, a to su jednodimenzionalni nizovi, svojstvo vraća broj članova u pojedinom nizu (7, 5 i 9).

8.4. Klasa ArrayList

Jedan od nedostataka klase Array i nizova koji je nasljeđuju je nemogućnost dinamičkog dodavanja članova. U poglavlju o jednodimenzionalnim nizovima spomenuli smo da se ovako nešto može napraviti:

```
int [] Arr = new int[7];
```

```
Arr = new int[14];
```

odnosno da se nakon definiranja niza koji sadrži 7 članova isti taj niz može definirati tako da sadrži 14 članova. Ali što bi u tom slučaju bilo s vrijednostima prvih 7 članova ako bismo ih u međuvremenu definirali, odnosno ako bismo napravili ovo:

```
int [] Arr = new int[7];
for(int i=0;i<7;i++)
      Arr[i] = 1;
Arr = new int[14];
```

Što bi sada bio sadržaj prvih 7 članova niza, 7 jedinica ili 7 nula?
Odgovor je: 7 nula jer bi operator *new* svih 14 članova doveo na njihove podrazumijevane vrijednosti, a to je 0.

Rješenje ovakvih situacija je klasa *ArrayList* iz imenika *System.Collections*.

```
using System;
using System.Collections;

public class Test
{
      public static void Main()
      {
            ArrayList Arr = new ArrayList();

            for(int i=0;i<7;i++)
                  Arr.Add(1);

            for(int i=0;i<7;i++)
                  Arr.Add(0);

            for(int i=0;i<14;i++)
                  Console.Write(Arr[i] + " ");
      }
}
```

Primjer 8.4

Ispis:
1 1 1 1 1 1 1 0 0 0 0 0 0 0

Definirali smo objekt *Arr* klase *ArrayList* te mu koristeći metodu članicu *Add* dodijelili za prvih 7 članova vrijednost 1. Nakon toga ponovo koristeći metodu *Add* dodajemo novih 7 članova s vrijednošću 0. Ono što se razlikuje u odnosu na prethodni primjer je da prvih 7 članova ostaje nepromijenjeno, odnosno dinamičko (u *runtime* vremenu) dodavanje novih članova nema utjecaja na postojeće članove.

Ako želimo ubrzati rad s *ArrayList*, možemo pozvati konstruktor s argumentom koji će kazati koliko memorijskog prostora treba rezervirati za objekt. To možemo samo ako unaprijed znamo približan broj elemenata. Vidjeli smo da nikakav problem nije dodati nove elemente u niz pa je onda bolje kod kreiranja objekta odmah mu omogućiti i rezerviranje memorijskog prostora.

Klasa *ArrayList* ima dosta vrlo upotrebljivih metoda, a jedna od njih je i metoda *Insert* koja insertira član u niz, dinamički mu povećavajući veličinu za 1. Slično tome postoji i metoda *RemoveAt* koja uklanja član niza s određenog indeksa. Obje ove metode mijenjaju indekse

nekih članova i to onih koji dolaze nakon insertiranog odnosno uklonjenog člana niza, pa je na to potrebno eventualno obratiti pozornost.

Za ostale metode i svojstva klase *ArrayList* pogledajte FCL dokumentaciju.

Implementacija *foreach* petlje

ArrayList je jedna od kolekcijskih klasa (*Collection Classes*), a njihova je osobina da su im članovi niza referentni tipovi odnosno objekti.
U prethodnom primjeru članove objekta *Arr* klase *ArrayList* smo mogli ispisati i ovako, koristeći *foreach* petlju:

```
foreach(int i in Arr)
      Console.Write(i);
```

Da bi kolekcijska klasa mogla koristiti *foreach* petlju sa sintaksom

```
foreach(tipvar ident in colec)
```

gdje je:
tipvar - tip varijable
ident – naziv varijable
in – ključna riječ
colec - objekt kolekcijske klase

mora implementirati metodu *GetEnumerator* koja će vratiti objekt tipa neke klase.

Ta klasa čijeg tipa objekt se vraća mora sadržavati:

- svojstvo *Current* koje će vratiti član tipa *tipvar* iz sintakse *foreach* petlje koji predstavlja aktivni (trenutni) element u kolekciji
- metodu *MoveNext* koja se pozicionira na sljedeći element kao aktivni i vraća *false* ako više nema elemenata u kolekciji

Sada ćemo definirati klasu *CollectionClass* čije ćemo članove niza tipa *double* moći proći kroz *foreach* petlju.

```
using System;
using System.Collections;

public class CollectionClass
{
      double [] Elementi;

      public CollectionClass()
      {
            Elementi = new double[] { 1.1, 2.2, 3.3, 4.4 };
      }

      public EnumeratorClass GetEnumerator()
      {
            return new EnumeratorClass(this);
      }

      public class EnumeratorClass
      {
            int Ind = -1;
```

```
            CollectionClass Kolekcija;

            public EnumeratorClass(CollectionClass Kolekcija)
            {
                  this.Kolekcija = Kolekcija;
            }

            public double Current
            {
                  get
                  {
                        return(Kolekcija.Elementi[Ind]);
                  }
            }

            public bool MoveNext()
            {
                  Ind++;
                  return Ind < Kolekcija.Elementi.Length;
            }
      }
}

public class Test
{
      public static void Main()
      {
            CollectionClass col = new CollectionClass();

            foreach(double d in col)
                  Console.WriteLine(d);
      }
}
```

Primjer 8.5

Ispis:
1,1
2,2
3,3
4,4

Klasa *CollectionClass* implementira, od *foreach* petlje zahtijevanu, metodu *GetEnumerator* koja vraća objekt tipa *EnumeratorClass*, klase koja je članica klase *CollectionClass*. Klasi *EnumeratorClass* je proslijeđen objekt *this* da bi njene metode *Current* i *MoveNext* mogle izvesti pozicioniranje i provjeru ima li još elemenata.

Kad je ovo sve ovako izvedeno, u funkciji *Main* može se pozvati *foreach* petlja koja će proći kroz sve *double* članove objekta *col*.

Što bi bilo kad niz *Elementi* ne bi imao niti jednog člana? Koji bi element vratilo svojstvo *Current*?
Očito je da bi u tom slučaju indeks kod poziva svojstva *Current* bio –1, dakle nedozvoljena vrijednost, međutim svojstvo se u tom slučaju uopće neće pozvati jer će se prije izvršiti metoda *MoveNext* koja će naredbom

```
Ind++
```

postaviti vrijednost varijable *Ind* na 0 i povratna vrijednost iz metode će u tom slučaju biti *false*.

Ovo je jednostavniji način implementacije *foreach* petlje. Onaj nešto složeniji zahtijevao bi da naša klasa *CollectionClass* naslijedi sučelje *IEnumerable* te da klasa *EnumeratorClass* naslijedi sučelje *IEnumerator*. To bi onda zahtijevalo i implementaciju metoda

```
IEnumerator IEnumerable.GetEnumerator()
{
	return GetEnumerator();
}
```

u klasi *CollectionClass* te u klasi *EnumeratorClass* implementaciju metode

```
public void Reset()
{
	Ind = -1;
}
```

te zamjenu postojećeg svojstva *Current* s

```
object IEnumerator.Current
{
	get
	{
		return(Kolekcija.Elementi[Ind]);
	}
}
```

Možemo primijetiti da svojstvo *Current* vraća tip *object* što znači i neophodno odboksiranje, ali prednost ovakvog drugog rješenja s nasljeđivanjem sučelja *IEnumerable* i *IEnumerator* je u slučaju poziva ovakve prevedene kolekcijske klase iz drugih jezika različitih od C# jer će za razliku od prvog slučaja drugi jezici ovo vidjeti kao pravu kolekciju.

Ipak, evo i rješenja koje će uzeti najbolje iz oba slučaja:

- nasljeđena sučelja *IEnumerable* i *IEnumerator*, a s tim i vidljivost iz drugih jezika
- svojstvo *Current* koje ne vraća tip *object* nego *double*

Preopteretit ćemo svojstvo *Current* tako da imamo oba definirana, *Current* kao i u prvom primjeru i *IEnumerator.Current* izvedeno tako da poziva prvo svojstvo.

```
public double Current
{
	get
	{
		return(Kolekcija.Elementi[Ind]);
	}
}

object IEnumerator.Current
{
	get
	{
		return this.Current;
	}
}
```

Poglavlje 9:
Delegati i događaji

Ovo poglavlje obrađuje:

- delegate i njihovu vezu s metodama
- višemetodne delegate
- događaje
- događaje u GUI-u

9.1. Delegati (Delegates)

U svim dosadašnjim primjerima poziv neke metode ostvarivao se direktnim pozivom ili preko klase ili preko objekta, ali svakako navođenjem imena metode. Međutim postoje situacije kada odluku koju metodu treba pozvati treba donijeti u vremenu izvođenja. Odnosno, u prevođenju znamo da ćemo pozvati neku metodu, ali ne znamo koju.

Za ovakvo dinamičko odlučivanje koju metodu treba pozvati koristi se posebni referentni tip podataka tzv. delegat (*delegate*) čija će instanca sadržavati referencu na metodu koja će biti pozvana s pozivom delegata.
Sintaksa deklaracije delegata izgleda ovako:

```
Modifikator delegate TipPovratneVrijednosti Naziv(TipArgumenata)
```

ili u nekom konkretnom primjeru:

```
public delegate bool Del(int i, int j);
```

U ovom primjeru deklariran je delegat *Del* čijim će se pozivom pozvati neka metoda koja vraća tip *bool* te koja kao argumente prima dva tipa *int*. Ovakva sintaksa mora se strogo poštovati kod metoda koje će biti indirektno pozvane preko delegata te argumenti metode i njena povratna vrijednost moraju točno odgovarati onima navedenim u deklaraciji delegata.

Iako se uvođenje delegata prikazuje kao velika novost u C#, mnogi poznavatelji C/C++ će naći veliku sličnost s tzv. pointerima na funkcije kojima se u principu moglo napraviti isto. Ipak postoje neke bitne razlike u korist delegata:

- delegati su tipski sigurni
- delegati mogu sadržavati reference na više metoda (od kojih sve moraju imati isti tip povratne vrijednosti te tip i broj argumenata)
- delegati su osnova za uvođenje događaja (*events*) koji će biti obrađeni u sljedećem poglavlju.

Sljedeći primjer će nam pokazati primjenu delegata. Definirat ćemo klasu *Girl* koja će imati samo dvije varijable članice: *visina* i *datumR*.

```
using System;

public class Girl
{
      private int visina;
      private DateTime datumR;

      public Girl(int i, DateTime d)
      {
            visina = i;
            datumR = d;
      }

      public int Visina
      {
            get
            {
                  return visina;
            }
      }
```

```
        public DateTime DatumR
        {
                get
                {
                        return datumR;
                }
        }
}

public class Test
{
        public delegate int Sortiranje(Girl o1, Girl o2);

        public static void Main()
        {
                Sortiranje d1 = new Sortiranje(PoVisini);
                Sortiranje d2 = new Sortiranje(PoDatumu);

                PozivPoDelegatu(d1);
                PozivPoDelegatu(d2);
        }

        public static int PoVisini(Girl o1, Girl o2)
        {
                return o1.Visina <  o2.Visina ? -1 :
                        o1.Visina == o2.Visina ?  0 : 1;
        }

        public static int PoDatumu(Girl o1, Girl o2)
        {
                return o1.DatumR <  o2.DatumR ? -1 :
                        o1.DatumR == o2.DatumR ?  0 : 1;
        }

        public static void PozivPoDelegatu(Sortiranje Met)
        {
                Girl Ivana = new Girl(125, new DateTime(2000, 6, 24));
                Girl Jelena = new Girl(110, new DateTime(2002, 11, 21));

                int Ret = Met(Ivana, Jelena);
                Console.WriteLine(Ret < 0 ? "Ivana < Jelena" :
                        Ret == 0 ? "Ivana == Jelena" : "Ivana > Jelena");
        }
}
```

Primjer 9.1

Ispis:
Ivana > Jelena
Ivana < Jelena

Deklariran je delegat *Sortiranje* te njegove dvije instance:

- *d1* koji sadrži referencu na metodu *PoVisini*
- *d2* koji sadrži referencu na metodu *PoDatumu*

Metoda *PoVisini* sortira dva objekta tipa *Girl* po visini odnosno po varijabli *visina*, dok ih metoda *PoDatumu* sortira po datumu rođenja, odnosno po varijabli *datumR*. Najvažniji dio priče događa se u naredbama:

```
PozivPoDelegatu(d1);
PozivPoDelegatu(d2);
```

gdje dinamički odlučujemo hoćemo li djevojke sortirati po visini (metoda *PoVisini*) ili po datumu rođenja (metoda *PoDatumu*), odnosno koja će se metoda pozvati. Mnogi će s pravom primijetiti čemu tolika priča oko delegata, zar nije bilo jednostavnije pozvati direktno te dvije metode.

To je točno za ovakav primjer, međutim zamislite da je kôd ovako izgledao

```
Sortiranje d = d1;
if(. . .)
   d = d2;
PozivPoDelegatu(d);
```

Sad je nezamjenjivost delegata očita jer sve do provjere u *if* uvjetu ne znamo koju metodu treba pozvati.
Poziv metode *PozivPoDelegatu* prosljeđuje instancu delegata samoj metodi koja onda s naredbom:

```
Met(Ivana, Jelena);
```

poziva samu metodu, čija je referenca sadržana u delegatu *Met*. Opet isto kao i kod svih objekata, *Met* je kopija prenesene instance delegata (*d1* i *d2*) koja ukazuje na isti sadržaj.
Pravila lijepog ponašanja još bi nalagala da se prije poziva delegata provjeri upućuje li uopće na neku metodu.

```
if(d1 != null)
   . . .
```

Program kao rezultat izvršenja ispisuje:
Ivana > Jelena
Ivana < Jelena

Za prvi slučaj sortiranja po visini logično je da je Ivana veća od Jelene jer je sortiranje provedeno po visini djevojčica, ali za drugi slučaj sortiranja po datumu baš i nije logično da je Ivana manja od Jelene. To je zato što kod uspoređivanja datuma vrijedi obrnuta logika (veći je noviji datum). Tu ništa ne možemo jer klasa *DateTime* preopterećuje operatore uspoređivanja na taj način.
Ipak, uvijek (gotovo uvijek) postoji rješenje, a to je u ovom slučaju da sami napravimo uspoređivanje objekata tipa *DateTime*.

```
public static int PoDatumu(Girl o1, Girl o2)
{
      if( (o1.DatumR.Year < o2.DatumR.Year)
                        ||
          ((o1.DatumR.Year == o2.DatumR.Year) &&
           (o1.DatumR.Month < o2.DatumR.Month))
                        ||
          ((o1.DatumR.Year == o2.DatumR.Year) &&
           (o1.DatumR.Month == o2.DatumR.Month) &&
           (o1.DatumR.Day < o2.DatumR.Day)))
             return 1;

      else if((o1.DatumR.Year == o2.DatumR.Year)
                        &&
```

```
                (o1.DatumR.Month == o2.DatumR.Month)
                          &&
                (o1.DatumR.Day == o2.DatumR.Day))
              return 0;

      else
              return -1;
}
```

Možda metoda *PoDatumu* izgleda složenije nego što jest, ako malo analizirate uvjete u *if else* naredbi u to ćete se i sami uvjeriti. Sad će ispis biti onaj očekivani i logičan te će Ivana biti veća od Jelene i po visini i po datumu rođenja.

Višemetodni delegati

Kod usporedbe sa C/C++ pointerima na funkciju naveli smo da delegati imaju tu prednost da jedan delegat može referencirati više metoda (*multicast delegate*). Najbolje u tome je da će se izvesti sve metode na koje referencira instanca delegata. Druga, treća itd… metoda dodaju se korištenjem operatora +=, a kao što se metode mogu dodavati, tako se mogu i oduzimati od instance delegata. Evo jednog jednostavnog primjera koji pokazuje višemetodne delegate.

```
using System;

public class Test
{
      public delegate void Del(double d);

      public static void Main()
      {
            Del d1 = new Del(F1);
            d1 += new Del(F2);
            d1(5);
            d1 -= new Del(F2);
            d1(6);
      }

      public static void F1(double d)
      {
            Console.WriteLine(d);
      }

      public static void F2(double d)
      {
            Console.WriteLine(d * 2);
      }
}
```

Primjer 9.2

Ispis:

```
5
10
6
```

Instanci *d1* delegata *Del* dodjeljuju se dvije reference, najprije na metodu *F1*, a zatim korištenjem operatora += i na metodu *F2*. To znači da će poziv instance *d1* pozvati obje metode, *F1* i nakon toga *F2*. Metode kao što su dodane, mogu biti i oduzete iz delegata što je i napravljeno naredbom

```
d1 -= new Del(F2);
```

Pritom nije nužno metode uklanjati obrnutim redoslijedom kojim su dodavane. Ako bismo napravili ovo:

```
d1 -= new Del(F1);
```

onda bi *d1* sadržavao referencu samo na metodu *F2*.
Delegati se mogu zbrajati i oduzimati i na ovakav način koristeći najobičnije + i – operatore.

```
Del d1 = new Del(F1);
Del d2 = new Del(F2);
Del d3 = d1 + d2;
Del d4 = d3 - d1;
```

Delegati su kao i stringovi nepromjenjivi (*immutable*) što znači da svaka radnja koja mijenja delegat stvara kopiju delegata ostavljajući prethodnog sve dok se ne ukloni mehanizmom čišćenja memorije *(garbage collection).* Međutim to nije problem kao kod stringova jer je tu obično riječ o vrlo malom broju ovakvih operacija nad delegatima. Sigurno nećete doći u situaciju da za jedan delegat dodijelite više od nekoliko metoda.

9.2. Događaji (Events)

Pojam događaja prisutan je u svim objektnim sustavima. To je potpuno promijenilo koncepciju rada s programima, jer dok je u neobjektnim programima program vodio korisnika, u objektnim sustavima korisnik je glavni, odnosno on odlučuje što će napraviti u određenom trenutku.
Korisnik će možda kliknuti mišem, pa će nešto napisati, pa će pokrenuti naredbu iz izbornika, pa će opet kliknuti mišem, ali ovaj put desnom tipkom itd. Nemoguće bi bilo predvidjeti sve te kombinacije unaprijed i eventualno ih obraditi neobjektnim sustavima.

Svaka takva radnja generira tzv. događaj (*event*) koji se onda obrađuje. Obradu takvog događaja nije nužno napraviti trenutačno, odnosno sam sustav može procijeniti na osnovi prioriteta, obradu kojeg događaja će izvršiti trenutačno, a obradu kojeg će staviti u listu čekanja i naknadno je izvesti.

Programski jezici različito obrađuju ove događaje, C# za to koristi upravo delegate.
Realizacija je jednostavna:
Klasa se prijavljuje klasi da je obavijesti kad se dogodi neki događaj koji je važan za taj njen objekt kao npr.

- Klasa *Form* (Forma je u .Netu naziv za standardni prozor programa Window) se prijavljuje klasi *Button* da želi obavijest kad se dogodi klik na *Button* jer njen objekt (klase *Form*) tada mora izvesti neku radnju ili više njih, kao npr. zatvoriti formu.
- Klasa *Form* se prijavljuje klasi *ListBox* da želi obavijest kad se dogodi dvostruki klik na neku stavku u listi jer njen objekt tada mora izvesti neku radnju, npr. prikazati sve podatke o toj stavci u nekom drugom prozoru.

Obavijest znači pozivanje metode klase koju se obavještava da se događaj dogodio.
Događaj je višemetodni delegat i ako je ovo deklaracija takvog delegata

```
public delegate void Del(int, int);
```

onda je ovo deklaracija događaja *Ev*

```
public event Del Ev;
```

Napravimo sada jedan primjer s događajima:

```
using System;

public delegate void Signal(int v, long b);

class Operater
{
	public event Signal Signalizirano;

	public void DolazniSignal(int vrsta, long broj)
	{
		Signalizirano(vrsta, broj);
	}
}

class MobilePhone
{
	public MobilePhone(Operater Sig)
	{
		Sig.Signalizirano += new Signal(SignalStigao);
	}

	private void SignalStigao(int vrsta, long broj)
	{
		if(vrsta == 0)
			Console.WriteLine("Stigao poziv od broja " + broj);
		else
			Console.WriteLine("Stigao SMS od broja " + broj);
	}
}

public class Test
{
	public static void Main()
	{
		Operater r = new Operater();
		MobilePhone SE = new MobilePhone(r);

		r.DolazniSignal(0, 38521100000);
		r.DolazniSignal(1, 38521200000);
	}
}
```

Primjer 9.3

Ispis:
Stigao poziv od broja 38521100000
Stigao SMS od broja 38521200000

Prvo što vidimo jest deklaracija delegata *Signal* izvan klase, iako je kao što smo rekli u C# sve dio klase. Ništa ne izlazi izvan dozvoljenih okvira jer je svaki delegat implementiran kao klasa izvedena iz klase *System.MulticastDelegate*.
Tako će za delegat iz ovog primjera

```
public delegate void Signal(int v, long b);
```

prevoditelj kreirati klasu:

```
class Signal : System.MulticastDelegate
{
	// Konstruktor . . .
	// Metoda BeginInvoke() . . .
```

```
      // Metoda EndInvoke() . . .

      public void Invoke(int v, long b);
}
```

Ako imamo:

```
Signal Del = new Signal(NekaMetoda);
Del(10, 1000);
```

onda je to zapravo:

```
Del.Invoke(10, 1000);
```

odnosno objekt *Del* klase *Signal* poziva metodu *Invoke*.

Ostaje još pitanje: Kako metoda *Invoke* zna da treba pozvati metodu *NekaMetoda?*
Odgovor je u deklaraciji konstruktora kojeg prevoditelj generira

```
public Signal(object target, int methodPtr)
```

u kojem drugi argument predstavlja metodu koju treba pozvati, zapravo predstavlja njenu adresu. Prvi argument je sam objekt kojemu će se pozvati metoda i za statičke metode ima vrijednost *null*.

Vratimo se sad na naš primjer.
Primjer definira dvije klase *Operater* i *MobilePhone*. Događaj *Signalizirano* definiran je naredbom:

```
public event Signal Signalizirano;
```

Klasa *MobilePhone* u svom se konstruktoru prijavljuje klasi *Operater* za obavijest o događaju s naredbom:

```
Sig.Signalizirano += new Signal(SignalStigao);
```

Ovo znači otprilike da klasa *MobilePhone* kaže klasi *Operater*:
obavijesti me kad dođe signal na način da pokreneš moju metodu *SignalStigao* te mi kao parametre pošalji vrstu poziva i broj.

Sve što *Operater* treba znati o *MobilePhone* je da treba pozvati njenu metodu *SignalStigao* te joj proslijediti odgovarajuće parametre. Kad poziv stigne do operatera, pozivom njegove metode *DolazniSignal* u funkciji *Main* on poziva metodu *SignalStigao* klase *MobilePhone*. Zato objekt *SE* klase *MobilePhone* ne mora poduzimati nikakvu akciju, dovoljno je da bude u stanju pripravnosti (standby), a metoda *SignalStigao* bit će pokrenuta od strane klase *Operater*.
Kao što je i u stvarnosti, vaš mobilni uređaj će primiti signal, obavijestiti vas te prikazati na zaslonu detalje o pozivu, sve to bez ikakve vlastito inicirane akcije.

Događaj je višemetodni delegat što onda znači da bismo mogli napraviti ovo

```
Sig.Signalizirano += new Signal(SignalStigao1);
Sig.Signalizirano += new Signal(SignalStigao2);
```

odnosno, dogodio bi se poziv dviju metoda *SignalStigao1* i *SignalStigao2* nakon ostvarenja događaja. Ako bi u našem primjeru htjeli ovo implementirati kao

- pozovi metodu *SignalStigao1* kao obavijest o vrsti poziva
- pozovi metodu *SignalStigao2* kao obavijest o broju koji zove ili šalje SMS

onda bi i same metode bile drugačije izvedene:

```
private void SignalStigao1(int vrsta, long broj)
{
      Console.WriteLine(vrsta == 0 ? "Stigao poziv" : "Stigao SMS");
}

private void SignalStigao2(int vrsta, long broj)
{
      Console.WriteLine(broj);
}
```

Vidimo da u svakoj od metoda imamo višak informacija, u prvoj je to varijabla broj, a u drugoj varijabla vrsta. To je u ovom slučaju neizbježno zbog izričitog zahtjeva da metode čije se reference dodaju jednom događaju moraju imati isti tip argumenata i povratne vrijednosti.

Nasljeđivanje klase koja generira događaj
Što bi se dogodilo ako bismo definirali klasu *OperaterNew* koja bi naslijedila klasu *Operater*

```
class OperaterNew : Operater
{
}
```

i ako bismo u funkciji *Main* kreirali objekt te nove klase i proslijedili ga objektu *SE*?

```
OperaterNew r = new OperaterNew();
MobilePhone SE = new MobilePhone(r);
```

Dogodilo bi se isto što i u prvom slučaju jer bi nova klasa naslijedila metodu *DolazniSignal* koja bi pozvala naslijeđeni događaj. Ako to ne bismo željeli, odnosno ako bismo željeli da klasa *MobilePhone* reagira samo na događaje iz klase *Operater*, ali ne i iz klase *OperaterNew*, trebali bismo metodu *DolazniSignal* u klasi *Operater* deklarirati kao virtualnu, a u klasi *OperaterNew* predefinirati je ovako:

```
public override void DolazniSignal(int vrsta, long broj)
{
      // Radi nešto sasvim drugo
}
```

Događaji u GUI-u
Događaji se često koriste u grafičkom korisničkom okruženju (GUI). Jedan od takvih primjera je prethodno spomenuti slučaj kada se klasa *Form* prijavljuje klasi *Button* za obavijest o nekom događaju nad objektom klase *Button*. Za tu svrhu koristi se predefinirani delegat iz imenika (namespace) *System*:

```
public delegate void EventHandler(object sender, System.EventArgs e);
```

Zato svaka metoda iz forme koju objekt klase *Button* poziva ima dva argumenta:

- **`object sender`** koji sadrži informacije o samom pošiljatelju odnosno objektu klase *Button*
- **`System.EventArgs e`** koji sadrži dodatne informacije o događaju

Zbog podudarnosti s preporukama .Net Frameworka to su argumenti koji bi se trebali koristiti kod svih metoda pozivanih od strane bilo kojeg delegata. Ako trebamo dodatne informacije proslijediti sa svojim argumentima, možemo naslijediti klasu *EventArgs* i proslijediti objekt takve klase.

Klasa *Button* ovako je definirana:

```
public class Button
{
      public event EventHandler Click;
      protected void OnClick(Event e)
      {
            // . . .
            Click(this, e);
      }
      // . . .
}
```

Ako forma želi da joj metoda bude pozvana kad se dogodi klik na *Button*, ona će se prijaviti za to definiranjem događaja:

```
TipkaOK.Click += new System.EventHandler(TipkaOK_Click);
```

gdje je *TipkaOK* objekt tipa klase *Button* koji generira događaj, a *TipkaOK_Click* metoda u formi koja će biti pozvana od događaja *Click* i čija će definicija ovako izgledati:

```
private void TipkaOK_Click(object sender, System.EventArgs e)
{
      // . . .
}
```

Poglavlje 10:
Iznimke

Ovo poglavlje obrađuje:

- razloge uvođenja iznimaka kao sastavnog dijela jezika i njihovo korištenje
- hvatanje iznimaka
- bacanje iznimaka
- ponovno bacanje iznimaka
- naredbu *finally*
- kreiranje vlastitih klasa iznimaka
- klasu *Exception* i njoj izvedene klase
- operatore *checked* i *unchecked*

10.1. Hvatanje iznimaka

Kad su autori Jave odlučili povećati sigurnost izvođenja programa te promijeniti C/C++ parolu "vjeruj programeru" u "zaštiti programera", kao da su predskazali što će se dogoditi u bliskoj budućnosti. I to ne samo u IT industriji, nego globalno u cijelom svijetu jer je sigurnost postala glavna tema svih dijelova života.

Programer je imao veliku slobodu u pisanju programa, međutim pokazalo se da su se zbog brzorastuće potražnje za programima isti ti programi morali sve brže i brže pisati tako da se programeru više nije moglo vjerovati da neće napisati neki program čiji će pojedini djelovi biti nesigurni.

Naravno da je i C# morao pratiti takve trendove, a jedna od najefikasnijih zaštita programera, a samim tim i korisnika programa od neželjenog izvođenja programa, je sustav iznimki (*exceptions*).
Iznimka se događa u slučaju nedozvoljenog izvršenja programa. To se može dogoditi zbog različitih razloga i svaki od njih će generirati iznimku koju je potrebno obraditi u kôdu. Razlozi mogu biti zbog loše napisanog programa, ali mogu biti prouzrokovani i sustavom. Jedan od razloga može biti npr. pokušaj dijeljenja s nulom. Pretpostavimo izvršenje ovog kôda:

```
int x = 1;
Random r = new Random();
for(int i = 0;i<100;i++)
{
	int y = r.Next(100);
	int z = x/y;
}
```

Postoje dva moguća ishoda ovog kôda ovisno o tome hoće li metoda *Next* iz 100 pokušaja vratiti broj 0 ili ne. Ako ne, neće biti dijeljenja s nulom i sve je OK. Ali ako nula bude generirana, doći će do dijeljenja s nulom odnosno nedozvoljene računske operacije pa će sustav prekinuti izvršenje programa i generirati iznimku. Napravimo sad program koji će uhvatiti tu iznimku i na taj način preduhitriti sustav.

```
using System;
using System.Threading;

public class Test
{
	public static void Main()
	{
		int x = 1;
		Random r = new Random();

		for(int i = 0;i<100;i++)
		{
			int y = r.Next(100);

			try
			{
				int z = x/y;
			}
			catch(Exception e)
			{
				Console.WriteLine("Pokušaj dijeljenja s nulom");
```

```
                    }
                }
        }
}
```

Primjer 10.1

Ispis:

Ispisat će se onoliko puta "Pokušaj dijeljenja s nulom" koliko puta bude generiran broj 0

Ipak je uvijek bolje osigurati da se iznimka uopće ne dogodi, koliko se to može jer izvršenje programa ne ovisi jedino o programeru. Hvatanje iznimke treba biti krajnja mjera i izvodi se kao što vidimo u primjeru s naredbom *try catch* koja se sastoji od dva dijela:

- *try* dijela u čijem bloku se trebaju nalaziti naredbe koje mogu uzrokovati iznimku
- *catch* dijela u kojem se nalaze naredbe koje će se izvršiti u slučaju ostvarenja iznimke

Iznimka je objekt kao i sve drugo i instanca je osnovne klase iznimki *System.Exception*. Postoji više različitih klasa za obradu iznimki sve izvedenih iz klase *Exception*. Zato je kod hvatanja iznimki poželjno uhvatiti pravu iznimku, a ne općenitu kao u našem primjeru. Naš primjer će generirati iznimku dijeljenja s nulom *DivideByZeroException* i zato bi *catch* dio trebao ovako izgledati:

```
catch(DivideByZeroException e)
{
}
```

Nakon hvatanja iznimke program ne prestaje s radom, nego nastavlja s izvođenjem prve naredbe nakon *catch* dijela, a to je u našem primjeru nastavak izvršenja *for* petlje.
U jednoj *try catch* naredbi može se hvatati (ali ne i odjednom uhvatiti!) i više iznimki tako da se *catch* dijelovi navedu jedan ispod drugoga. Ako bismo u gornjem primjeru imali još i definiran cjelobrojni niz od 90 članova:

```
int [] Arr = new int[90];
```

te ako bi *try catch* naredba ovako izgledala:

```
try
{
      int z = x/y;
      Arr[y] = 0;
}
catch(DivideByZeroException e)
{
      Console.WriteLine("Pokušaj dijeljenja s nulom");
}
catch(IndexOutOfRangeException e)
{
      Console.WriteLine("Indeks veći od dozvoljenog");
}
```

ovisno o generiranom broju dogodilo bi se sljedeće:

- ne bi se dogodila nikakva iznimka za `0 < y < 90`
- dogodila bi se iznimka `DivideByZeroException za y = 0`
- dogodila bi se iznimka `IndexOutOfRangeException za y > 89`

Pritom se prije provjerava prvonavedena *catch* naredba i ako bismo stavili kao prvi *catch* hvatanje generalne iznimke

```
catch(Exception e)
```

program se ne bi preveo jer ona obuhvaća sve ostale iznimke, odnosno ako bi se izvršila bilo koja iznimka onda bi se izvršila i generalna iznimka *Exception*. Zato treba najprije navesti preciznije iznimke pa tek onda one općenitije kao npr:

```
catch(DivideByZeroException Exception e)
catch(ArithmeticException e)
catch(Exception e)
```

Obratite pozornost da se iznimke događaju nad sintaktički potpuno ispravnim programom, napisanom po pravilima programskog jezika te da iznimke ne sprječavaju bugove u programu kada algoritam nije izveden kako treba.

Tablica iznimaka:

Iznimka	**Opis**
`System.OutOfMemoryException`	Generirana kada ne uspije pokušaj dodjeljivanja memorije
`System.StackOverflowException`	Generirana kada dođe do prekoračenja stoga s velikim brojem poziva metoda
`System.NullReferenceException`	Generirana kada se referencira *null* objekt tamo gdje je zahtijevan *not null* objekt
`System.TypeInitializationException`	Generirana kada statički konstruktor generira iznimku, a ne postoji *catch* dio da obradi iznimku
`System.InvalidCastException`	Generirana kada u vremenu izvođenja ne uspije eksplicitna konverzija iz osnovnog tipa ili sučelja u izvedeni tip
`System.ArrayTypeMismatchException`	Generira se kada spremanje podataka u niz ne uspije jer je podatak nepodudaran s tipom niza
`System.IndexOutOfRangeException`	Generira se kada se pokuša pristupiti članu niza s indeksom manjim od nule ili većim od indeksa posljednjeg člana u nizu
`System.MulticastNotSupportedException`	Generira se kada se pokuša napraviti višemetodni delegat od delegata koji nemaju *void* kao povratnu vrijednost
`System.ArithmeticException`	Generira se u slučaju neuspjeha izvođenja aritmetičkih operacija
`System.DivideByZeroException`	Generira se kada dođe do dijeljenja s nulom
`System.OverflowException`	Generira se kod premašenja vrijednosti tipa varijabli u slučaju izvođenja aritmetičkih operacija

Naredba *finally*

Nekad je potrebno i u slučaju generiranja iznimke izvršiti određene naredbe kao npr. zatvoriti otvorenu bazu podataka, odspojiti vezu s internetom itd. Ako bismo to radili u *try* dijelu, postoji mogućnost da se ne bi izvršilo jer ostvarena iznimka prekida izvršenje *try* dijela.

```
try
{
      int z = x/y;
      // Zatvori bazu podataka . . .
}
catch(DivideByZeroException e)
{
      Console.WriteLine("Pokušaj dijeljenja s nulom");
}
```

U ovom primjeru baza podataka ne bi se zatvorila u slučaju da je *y* jednak 0. Ako želimo bezuvjetno zatvoriti bazu podataka (a želimo i moramo), u *try catch* naredbi mora se dodati *finally* dio koji će se izvršiti u svakom slučaju bez obzira na ostvarenu iznimku.

```
try
{
      int z = x/y;
}
catch(DivideByZeroException e)
{
      Console.WriteLine("Pokušaj dijeljenja s nulom");
}
finally
{
      // Zatvori bazu podataka . . .
}
```

Sada smo sigurni da će baza podataka biti zatvorena. *Finally* blok ne zahtijeva nužno da bude naveden *catch* dio, ovo sljedeće će se prevesti bez grešaka i izvršiti će se *finally* dio, ali će iznimku obraditi sustav, odnosno CLR (Common Language Runtime) što će izgledati ružno korisniku programa.

```
try
{
      int z = x/y;
}
finally
{
      // Zatvori bazu podataka . . .
}
```

Finally blok ne smijemo napustiti pozivom naredbe *return* i tu nas štiti prevoditelj, ali nitko nas ne štiti ako se dogodi iznimka u *finally* dijelu jer će tada doći do prekida izvršenja *finally* bloka. Zato ako je već moguće da se dogodi iznimka u *finally* bloku, treba sve ono što je najbitnije navesti na početku bloka. U tom slučaju bi se takve naredbe kao zatvaranje baze podataka ipak izvršile bez obzira na eventualno ostvarenu iznimku.
Drugačiji je slučaj ako se iznimka dogodi u *catch* dijelu, netko će obraditi tu iznimku (CLR ili neki drugi *try catch*), ali će se nakon toga *finally* dio ipak izvesti.

Exception objekt

Svaka generirana iznimka kreira objekt tipa klase ostvarene iznimke. A gdje je objekt, tu su i njegove metode i svojstva koja možemo iskoristiti za dobivanje dodatnih informacija o ostvarenoj iznimki. Taj objekt naveden je u *catch* dijelu u kojem možemo i pozvati metode i svojstva objekta iznimke.

Jedno od svojstava je *Message* koje vraća opis iznimke. Obično ako imamo točno određenu iznimku, svojstvo *Message* i ne koristi puno jer ako smo uhvatili iznimku

```
catch(IndexOutOfRangeException e)
```

onda nam *Message* koji će vratiti tekst *"Index was outside the bounds of the array"* i ne govori ono što nismo već mogli znati iz imena iznimke. Ovo svojstvo može biti vrlo korisno ako je uhvaćena generalna iznimka *System.Exception* jer će ono i tada vratiti točan opis ostvarene iznimke.

```
using System;
using System.Threading;

public class Test
{
	public static void Main()
	{
		int x = 1, y = 0, z;
		string [] Desetoboj = new string [] {"100 m", "Skok u dalj",
				"Bacanje kugle", "Skok u vis", "400 m",
				"110 m prepone", "Bacanje diska", "Skok s motkom",
				"Bacanje koplja", "1500 m"};

		Random r = new Random();

		try
		{
			if(r.Next(2) == 0)
				Desetoboj[10] = "Maraton";
			else
				z = x / y;
		}
		catch(Exception e)
		{
			Console.WriteLine(e.Message);
		}
	}
}
```

Primjer 10.2

Ispis:
Index was outside the bounds of the array. (ili Attempted to divide by zero.)

Imamo *try* blok u kojem će se ovisno o vrijednosti slučajnog broja koji može biti 0 ili 1, dogoditi jedna od dviju mogućih iznimki, prekoračenje indeksa ili dijeljenje s nulom. Međutim iznimke su obuhvaćene generalnom iznimkom *Exception* i budući da će objekt generirane iznimke imati definirano svojstvo *Message* s opisom iznimke, isti taj opis možemo lako saznati pozivom svojstva.

Ostali članovi objekta iznimke su:

- *HelpLink* - link na help datoteku vezanu za iznimku
- *InnerException* - instanca objekta koji je uzrokovao iznimku
- *Source* - ime objekta ili aplikacije koji su uzrokovali iznimku
- *StackTrace* - string prezentacija stoga u trenutku ostvarenja iznimke
- *TargetSite* - metoda u kojoj se dogodila iznimka

10.2. Bacanje iznimaka (Throwing)

Osim sustava, iznimke se mogu generirati i u programu. Možda za takvo nešto baš i nije prikladan prijevod bacanje, ali bilo kako bilo, riječ je o generiranju iznimke pozivom ključne riječi *throw* što će uzrokovati trenutačno ostvarenje određene iznimke.
Sintaksa naredbe *throw* je:

```
throw izraz
```

gdje je
izraz - objekt tipa klase *Exception* ili neke od nje izvedene klase.

Sljedeći primjer pokazat će nam bacanje iznimke. Riječ će biti o situaciji kad osoba zadužena za daljinsko dodatno upravljanje bolidom Formule 1 (vozač je ipak glavni) unosi parametar koji će podesiti motor. Imat ćemo klasu *F1* te u njoj metode *PodesiParametar* u kojoj osoba unosi *double* vrijednost i *PodesiMotor* koja će prije obavljanja posla provjeriti je li parametar odgovarajući te ako nije generirati iznimku.

```
using System;

class F1
{
	public void UnesiParametar()
	{
		try
		{
			double d = Convert.ToDouble(Console.ReadLine());
			PodesiMotor(d);
		}
		catch
		{
			Console.WriteLine("Nedozvoljena vrijednost");
		}
	}

	public static void PodesiMotor(double d)
	{
		if(d < 50  ||  d > 60)
			throw(new Exception());
		else
			Console.WriteLine(d + " je OK");
	}
}

class Test
{
	public static void Main()
	{
		F1 Renault = new F1();
		Renault.UnesiParametar();
```

```
        }
}
```

Primjer 10.3

Ispis:
Za unesenu vrijednost npr. 55,6 ispisuje se 55,6 je OK,
a za unesenu vrijednost npr. 5,6 ispisuje se Nedozvoljena vrijednost

Može se vidjeti naredba *catch* bez navođenja objekta. To je moguće ako se želi uhvatiti općenita iznimka *System.Exception*. Dakle, ovo je isto:

```
catch(Exception e) {}
catch {}
```

Osoba unosi parametar u metodi *UnesiParametar* koji se prosljeđuje metodi *PodesiMotor* koja će prije bilo kojeg posla provjeriti je li parametar možda izvan dopuštenih vrijednosti. Ako je izvan tih vrijednosti, sama metoda će generirati iznimku koja će se obraditi u metodi *UnesiParametar*, znači u metodi koja je pozvala metodu *PodesiMotor*.
Kod odabira *try catch* naredbe u kojoj će se obraditi generirana iznimka vrijedi sljedeći redoslijed traženja odgovarajuće obrade iznimke:

1. Metoda u kojoj je iznimka generirana
2. Metoda koja je pozvala metodu u kojoj je je iznimka generirana itd.
3. Funkcija *Main*
4. CLR

Treba naglasiti da se traži **odgovarajuća** obrada iznimke što znači da se će se pozvati obrada samo točno one iznimke koja je bačena ili iznimke koju nasljeđuje bačena iznimka. Ako se u koracima od 1 do 3 ne nađe takva iznimka, CLR iz koraka 4 će obraditi iznimku. Kad bismo bacili npr. iznimku *DivideByZeroException* i ako bismo u koraku 3 imali obradu baš te iznimke, a u koraku 2 obradu iznimke *ArithmeticException* koju iznimka *DivideByZeroException* nasljeđuje, pozvala bi se iznimka u koraku 2, dakle iznimka *ArithmeticException*.
Obrnuto ne vrijedi, ako bismo bacili iznimku *ArithmeticException,* ona se ne bi mogla obraditi sa *catch(DivideByZeroException).*

Ponovno bacanje (Rethrowing)
Ponekad *catch* blok neće moći potpuno obraditi iznimku i željet će poslati iznimku na daljnju obradu npr. metodi iz koje je pozvana. To je tzv. ponovno bacanje iznimke (*rethrowing*) i navodi se s naredbom *throw* bez argumenata.

```
using System;

public class Test
{
      public static void Main()
      {
            try
            {
                  Overflow();
            }
            catch(Exception e)
            {
                  Console.WriteLine("Dodatna obrada iznimke ...");
            }
```

```
        }

        public static void Overflow()
        {
                int [] Arr = new int [10];

                try
                {
                        Arr[10] = 1;
                }
                catch(IndexOutOfRangeException e)
                {
                        Console.WriteLine("Obrada iznimke ...");
                        throw;
                }
        }
}
```

Primjer 10.4

Ispis:
Obrada iznimke ...
Dodatna obrada iznimke ...

U metodi *Overflow* izvedeno je nedozvoljeno dodjeljivanje vrijednosti nizu *Arr*. S obzirom da je indeks premašen, generirana je iznimka *IndexOutOfRangeException* i u *catch* bloku je iznimka obrađena. Budući da *catch* blok nije uspio potpuno riješiti situaciju, naredbom *throw* bez argumenata iznimka je ponovo generirana i vraćena na obradu funkciji *Main* iz koje je pozvana metoda *Overflow*. U funkciji *Main* pronalazi se *catch* s generalnom iznimkom te se iznimka dodatno obrađuje.

Naravno da je u funkciji *Main* iznimku mogla obraditi samo ista iznimka ili iznimka iz koje je izvedena klasa iznimke *IndexOutOfRangeException*. Da je u *Main* funkciji umjesto *Exception* bila *ArithmeticException*, CLR bi preuzeo konačnu obradu iznimke.

10.3. Kreiranje vlastitih iznimaka

Uz sve ugrađene iznimke nekad je potrebno imati i posebnu klasu iznimke koja će obraditi poseban slučaj u programu. Vratimo li se na primjer s Formulom 1, vidjet ćemo da je unos nedozvoljenog parametra aktivirao generalnu iznimku *System.Exception*. Bilo bi bolje kada bi taj nedozvoljeni unos aktivirao što je moguće specijaliziraniju iznimku koja ne bi obrađivala ništa drugo nego samo takav jedan slučaj.

To je moguće nasljeđivanjem klase *ApplicationException*. Pogledajmo sada onaj primjer s Formulom 1 u kojem ćemo definirati klasu iznimke *F1Exception*.

```
using System;

class F1Exception : ApplicationException
{
        public F1Exception(string Message) : base(Message)
        {
        }
}

class F1
{
        public void UnesiParametar()
```

```
        {
                try
                {
                        double d = Convert.ToDouble(Console.ReadLine());
                        PodesiMotor(d);
                }
                catch (F1Exception e)
                {
                        Console.WriteLine(e.Message);
                }
        }

        public static void PodesiMotor(double d)
        {
                if(d < 50  ||  d > 60)
                {
                        F1Exception f1 = new F1Exception("Štedi gorivo, ne mogu
                                                        podesiti motor");
                        throw(f1);
                }
                else
                        Console.WriteLine(d + " je OK");
        }
}

class Test
{
        public static void Main()
        {
                F1 Renault = new F1();
                Renault.UnesiParametar();
        }
}
```

Primjer 10.5

Ispis:
Za unesenu nedozvoljenu vrijednost ispisuje se poruka
Štedi gorivo, ne mogu podesiti motor

Razlika u odnosu na prethodni primjer je što se sada naredbom *throw* baca već kreirani objekt *f1* klase *F1Exception* kojem prije bacanja možemo postaviti neke vrijednosti. U ovom slučaju se njegovom konstruktoru prosljeđuje string, koji će nakon poziva konstruktora osnovne klase s tim istim stringom kao argumentom, biti poruka u svojstvu *Message*.

Svojstvo *Message* je *readonly* i zato ga ne možemo definirati nakon kreiranja objekta, ali možemo u konstruktoru, što se upravo i radi u ovom slučaju.
Sad lako možemo vidjeti bolju upotrebljivost ovakve jedne strogo specijalizirane klase jer ako pretpostavimo da je svojstvo *Message* ono što će biti ispisano vozaču na njegovom ekranu, taj string koji će biti u svojstvu *Message* može se dinamički kreirati ovisno o situaciji i kao takav proslijediti vozaču.
(Dobro, reći ćete da oni imaju radio vezu što je puno jednostavniji način komunikacije, ali radio veza može i zatajiti, a može se i prisluškivati… Objektno orijentirano programiranje je uvedeno, između ostalog, zato što cijeli svijet funkcionira na tom principu i zato nam je cilj naći primjere koji bi mogli opisati stvarne situacije, ali naravno da je implementacija stvarnosti dosta drugačija. Ipak, princip ostaje isti, sve je objekt neke klase.)

U nekoj drugoj situaciji možda ćemo željeti definirati svojstvo *HelpLink* što je URL help datoteke vezane za iznimku. Ovo svojstvo za razliku od svojstva *Message* nije *readonly* i može mu se dodijeliti vrijednost nakon kreiranja objekta.
Podrazumijeva se da ova naslijeđena klasa može definirati nove članove pored onih naslijeđenih i na taj način još bolje omogućiti obradu iznimke.

Zaključak
Iznimke su postale neizostavni dio svakog novog programskog jezika isto kao što trebaju biti ugrađene u svaki ozbiljniji program. Ipak kod korištenja iznimaka trebalo bi se držati nekih preporuka:

- Treba nastojati uhvatiti što precizniju iznimku, dakle treba izbjegavati hvatanje općenitih iznimki tipa *System.Exception*. Ipak je bolje uhvatiti i takvu opću iznimku nego prepustiti CLR-u da to riješi na svoj način. Način na koji CLR to rješava je, ako ništa drugo, korisniku programa puno neljubazniji.

- Ako se iznimka uhvatila, onda bi se trebala nastojati i riješiti u *catch* dijelu. Nije dovoljno samo preduhitriti CLR i njegovu poruku o iznimci koja se dogodila te umjesto nje staviti neku u kojoj ćemo se ispričati korisniku zbog toga što se dogodilo. Npr. ako se dogodila iznimka zbog pokušaja otvaranja nepostojeće datoteke, nije dovoljno samo ispisati poruku da datoteka ne postoji (pa makar to ispisali ne znam kako lijepim fontom), nego je potrebno i ponuditi korisniku da eventualno otvori neku drugu datoteku.

- Ne treba pretjerivati s korištenjem iznimaka. Tamo gdje treba, OK neka budu, ali veliki broj iznimaka u programu troši resurse, a i gomila programski kôd.

10.5. Operatori checked i unchecked

C# po podrazumijevanim (default) postavkama prevoditelja omogućuje tzv. tiho prekoračenje vrijednosti (silent overflow). Ako izvedemo ovaj kôd:

```
sbyte x = 127;
sbyte y = ++x;
```

prevoditelj neće javljati nikakvo upozorenje, ali u varijabli *y* neće biti 128 nego –128 jer će doći do promjene vrijednosti vodećeg bita (vidi poglavlje br. 2). Ako ipak želimo provjeru ovakvog jednog izraza, imamo dva rješenja:

- promijeniti postavke prevoditelja (*Za Visual Studio to je Project -> Properties-> Configuration Properties -> Check for arithmetic overflow/underflow -> true*)
- koristiti *checked* operator

Operator *checked* ima prioritet u odnosu na postavke prevoditelja.

Sljedeći primjer pokazat će korištenje ovog operatora.

```
using System;

public class Test
{
      public static void Main()
      {
            sbyte x = 127, y;
```

```
            try
            {
                  y = checked(++x);
            }
            catch(OverflowException e)
            {
                  y = x;
            }
            Console.WriteLine(y);
      }
}
```

Primjer 10.6

Ispis:
127

U gornjem primjeru, prije nego što se varijabli *y* dodijeli vrijednosti varijable *x* uvećane za jedan, provjerava se operatorom *checked* je li došlo do prekoračenja dozvoljene pozitivne vrijednosti za *sbyte* (127). Kako je 127 + 1 = 128, vrijednost je prekoračena i operator *checked* je generirao iznimku *OverflowException*. Da je *x* imao neku manju vrijednost, iznimka se ne bi dogodila i *y* bi imao vrijednost *x* uvećano za jedan.

Za razliku od *checked*, operator *unchecked* dozvoljava ovakve operacije, ali rezultat je izvođenje operacije s netočnim rezultatom. Kao i kod operatora *checked* koji neće generirati iznimku ako ne dođe do premašenja vrijednosti, tako će i *unchecked* dati točan rezultat ako izračunata vrijednost ne premašuje dozvoljenu vrijednost.

```
using System;

public class Test
{
      public static void Main()
      {
            int x = 2000000000, y = 2, z;
            try
            {
                  z = checked(x * y);
            }
            catch(OverflowException e)
            {
                  z = unchecked(x * y);
                  Console.WriteLine(z);
            }
      }
}
```

Primjer 10.7

Ispis:
-294967296

Vrijednost varijable *x* pomnožene s *y* premašuje dozvoljenu *int* vrijednost i zato će *checked* operator generirati iznimku u kojoj će se napraviti ista operacija *x* * *y* ali uz pomoć operatora *unchecked*, te će rezultat biti netočan jer je došlo do premašenja vrijednosti.
Da smo *x* i *y* definirali kao konstante:

```
const int x = 2000000000, y = 2;
```

program se ne bi preveo jer bi prevoditelj u vremenu prevođenja izračunao da je riječ o premašenju vrijednosti. Međutim ako bismo za ovakve konstantne *x* i *y* stavili samo

```
z = unchecked(x * y);
```

to bi se prevelo, iako u *z* ne bi bila ispravna vrijednost.

S jednim *checked* ili *unchecked* operatorom možemo provjeriti i više izraza tako da ih stavimo u blok:

```
checked
{
      y1 = x * 2;
      y2 = x * 4;
      y3 = x * 6;
}
```

Jedno ograničenje koje imaju ovi operatori je da ne mogu ići u "dubinu".

```
static void F()
{
      int i = checked(NaKvadrat(50000));
}

static int NaKvadrat(int x)
{
      return x * x;
}
```

U ovom primjeru *checked* ne može provjeriti hoće li biti premašenja vrijednosti u metodi *NaKvadrat*. Međutim, ovo sljedeće uspješno bi provjerio i generirao iznimku o premašenju dozvoljene vrijednosti:

```
int x = 50000, y = 50000;
int i = checked(NaKvadrat(x * y));
```

Poglavlje 11:
Atributi i refleksija

Ovo poglavlje obrađuje:

- predefinirane atribute
- korisničke atribute
- *AttributeUsage* klasu
- pozicijske i imenovane parametre
- refleksiju
- posredno pozivanje metoda

11.1. Atributi

Atributi (*Attributes*) su jedna od novosti u C# koja omogućuje da klasa na neki način sama sebe opiše. Osnovna klasa *Atribut* ili od nje izvedene klase tako mogu opisati klase, metode, varijable, događaje itd, a takve informacije koje se još nazivaju deklarativni opisi, mogu se čitati i u vremenu prevođenja i u vremenu izvođenja programa. Atributi se navode unutar uglatih zagrada ispred člana na koji se odnose.
Postoji nekoliko definiranih klasa za rad s atributima u sklopu .Net Frameworka, a sljedeći primjer će nam pokazati korištenje jedne od njih gdje ćemo pokazati kako prevoditelj može čitati atribute. Riječ će biti o klasi *System.Attribute.Obsolete*.

```
using System;

public class Test
{
      [Obsolete("Metoda FCPlusPlus je zastarjela, koristite FCSharp",
                                                      true)]
      static void FCPlusPlus()
      {
            //int *Arr = (int *) malloc(100 * sizeof(int));
      }

      static void FCSharp()
      {
            int [] Arr = new int[100];
      }

      public static void Main()
      {
            FCPlusPlus();
      }
}
```

Primjer 11.1

Primjer se neće dati prevesti jer će prevoditelj ispred metode *FCPlusPlus* u uglatim zagradama pronaći atribut *Obsolete*, te će javiti poruku:
'Test.FCPlusPlus ()' is obsolete: 'Metoda FCPlusPlus je zastarjela, koristite FCSharp'

Ono što vidimo ispred metode *FCPlusPlus* je poziv konstruktora objekta klase *ObsoleteAttribute* s dva argumenta:

- prvi argument je string koji predstavlja poruku
- drugi argument je tip *bool* koji određuje hoće li poziv ovakve metode predstavljati grešku (*true*) ili upozorenje (*false*) prevoditelja.

Da smo u funkciji *Main* pozvali metodu *FCSharp* program bi se preveo i izveo bez grešaka i upozorenja. Možete vidjeti da se kao ime klase u atributu navodi samo *Obsolete*, a ne puno ime klase *ObsoleteAttribute*. C# prevoditelj omogućuje izostavljanje riječi *Attribute* u imenu klase, ali naravno da se može staviti i puno ime klase *ObsoleteAttribute*.
Ova klasa je inače izvedena iz osnovne klase *Attribute* iz imenika *System*.

Zanimljiva je i klasa *DLLImportAttribute* koja omogućuje poziv Win32 API funkcija. Napravit ćemo primjer u kojem će se pozvati poznata API funkcija *MessageBox*.

```
using System;
```

```
using System.Runtime.InteropServices;

class Test
{
      [DllImportAttribute("user32.dll")]
      public static extern int MessageBoxA(
                  int i, string s1, string s2, int    t);

      public static int Main()
      {
            return MessageBoxA(0, "API je pozvan", "Win32", 0);
      }
}
```

Primjer 11.2

Konstruktoru klase *DLLImportAttribute* prosljeđuje se naziv dll (dynamic link library) datoteke u kojoj se nalazi funkcija koja se želi pozvati. Program će pozvati API funkciju *MessageBox* koja će ispisati poruku "*Api je pozvan*" s naslovom "*Win32*".

U ova dva primjera atribut smo pridijelili metodama, ali poveći je popis tipova koji mogu imati pridijeljene atribute:

- Sve – bilo koji element aplikacije
- Klasa
- Konstruktor
- Metoda
- Svojstvo (Property)
- Varijabla
- Povratna vrijednost
- Delegat
- Događaj (Event)
- Tip enum
- Struktura
- Sučelje (Interface)
- Modul
- Parametar
- Skup (Assembly)

Korisnički atributi

Osim predefiniranih, atributi mogu biti i korisnički definirani (*Custom attributes*). Jedini je uvjet da naslijede apstraktnu klasu *Attribute*. Korisnički definirana klasa atributa može se npr. koristiti da pridijelimo klasi ime autora koji ju je izradio te njenu trenutnu verziju što ćemo pokazati na sljedećem primjeru. Definirat ćemo korisnički definiran atribut *AutorAttribute*:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
       AllowMultiple = false, Inherited = false)]
public class AutorAttribute : Attribute
{
      protected string autor;
      private double verzija;
      private bool vrstaPosla;

      public AutorAttribute(string autor, double verzija)
      {
```

```
        }
        . . .
}
```

Primjer deklaracije pokazuje da sama *AutorAttribute* klasa ima pridijeljeni atribut i to tipa klase *AttributeUsage*. *AttributeUsage* klasa je predefinirana klasa, dio Framework-a, koja ima tri svojstva koja se mogu definirati.

- ValidOn
- AllowMultiple
- Inherited

Svojstvo *ValidOn*
Ovo svojstvo omogućuje da se odredi na koje tipove varijabli klase će se primijeniti atribut tipa *AttributeUsage*. Već smo naveli sve dozvoljene tipove na koje se atributi mogu primijeniti, a to su manje-više svi tipovi koje inače klasa može sadržavati kao svoje članove. Svi ti tipovi su definirani u sklopu klase *AttributeUsage* kao tip *enum*:

```
public enum AttributeTargets
{
      Assembly    = 0x0001,
      Module      = 0x0002,
      Class       = 0x0004,
      Struct      = 0x0008,
      Enum        = 0x0010,
      Constructor = 0x0020,
      Method      = 0x0040,
      Property    = 0x0080,
      Field       = 0x0100,
      Event       = 0x0200,
      Interface   = 0x0400,
      Parameter   = 0x0800,
      Delegate    = 0x1000,

      All = Assembly | Module | Class | Struct | Enum | Constructor |
            Method | Property | Field | Event | Interface | Parameter |
            Delegate,

      ClassMembers  =  Class | Struct | Enum | Constructor | Method |
            Property | Field | Event | Delegate | Interface
}
```

Ako bismo vrijednosti svakog člana iz *AttributeTargets* pretvorili u binarni brojevni sustav mogli bismo primijetiti dvoje:

- svaki od njih ima jedinicu samo na jednom mjestu
- nijedna dva člana nemaju jedinicu na istom mjestu

To znači da primjenom binarnog operatora OR (|) na bilo koja dva ili više članova možemo dobiti vrijednost koja će u sebi sadržavati vrijednosti svih članova, odnosno vrijednost koja će imati preslikanu jedinicu od svakog od svojih operanada.
Npr. ako napravimo ovo:

```
int x = (int)(AttributeTargets.Method | AttributeTargets.Property);
```

U varijabli *x* će biti vrijednost 192 odnosno binarno 11000000, dakle obje jedinice su sadržane i obje su na svojim mjestima.
Tako prevoditelj iz samo jedne varijable može pročitati na koga se sve atribut može odnositi i u slučaju nedopuštenog dodjeljivanja atributa neće dozvoliti prevođenje.

Svojstvo *AllowMultiple*
Ovo svojstvo tipa *bool* omogućuje ili onemogućuje pridjeljivanje više od jednog atributa jednom elementu. Ako je definiran kao *true*, onda omogućuje da se, ako ćemo uzeti klasu *AutorAttribute* kao primjer, jednoj klasi dodijele dva različita atributa, odnosno dvije instance te klase pri čemu se oni navode jedan ispod drugog.

```
[Autor("John Smith", 1.0, VrstaPosla="Programiranje")]
[Autor("Smith Jones", 1.0, VrstaPosla="Testiranje")]
public void ToNumber()
{
}
```

Kao što ćemo vidjeti u nastavku, ovo omogućuje da se programski prođe kroz cijelu klasu i ustanovi tko je koju metodu programirao, a tko testirao.

Svojstvo *Inherited*
Svojstvo *Inherited* je također tipa *bool* i ono određuje hoće li klase koje nasljeđuju klasu s primijenjenim atributima naslijediti iste te atribute. Logično, *true* označava nasljedivost atributa, dok *false* označuje nenasljedivost.

Pozicijski i imenovani parametri
Ako pogledamo primjer klase *AutorAttribute*, uočit ćemo da klasa ima tri varijable: *autor*, *verzija* i *opis*, ali samo prve dvije se prosljeđuju konstruktoru u kojem se i definiraju. To su tzv. pozicijski (*positional*) parametri i njih je obvezno navesti kod svakog definiranja instance atributa. Treći parametar je tzv. imenovani (*named*) parametar i proizvoljan je te njega možemo, ali i ne moramo definirati kod pridjeljivanja atributa. U slučaju ako ga želimo definirati, moramo osigurati i *set* svojstvo za tu varijablu koje se onda pozove u samoj dodjeli atributa:

```
VrstaPosla="Programiranje"
```

Ako se imenovani parametri ne definiraju, oni se inicijaliziraju na podrazumijevane (default) vrijednosti za određeni tip varijable. Za podrazumijevane vrijednosti pogledajte tablicu u poglavlju 5.2.

11.2. Refleksija (Reflection)

Zamjena za ovakvo pridjeljivanje atributa klasama i metodama bilo bi eventualno komentiranje za svaku klasu, metodu i ostale dijelove kôda, tko je programirao pojedinu klasu ili tko je testirao pojedinu metodu. Više je nego jasno da bi izrada dokumentacije o autorima jednog velikog projekta iziskivala puno ručnog rada ili bi u najboljem slučaju tražila izradu nekog programa za parsiranje, odnosno ekstrahiranje informacija, tako da bi taj program u konačnici bio krajnje nepouzdan.

Zato, ako dosad i niste vidjeli neku veliku upotrebljivost atributa, jer lako je naći protuargument za ona dva primjera koja smo pokazali:

- primjer s zastarjelim kôdom (zašto uopće ostavljati zastarjeli kôd u programu, teško da će se ijedna nova verzija programa ikada vraćati na stare alate?)
- primjer s pozivom funkcije *MessageBox* (što će nam Win32 funkcija kad .Net Framework ima istu takvu, ako ne i bolju funkciju? Iako ovdje može biti riječ o vrlo složenim, upotrebljivim i što je osobito važno, već napisanim funkcijama koje se ne isplati preprogramirati za .Net)

korištenje refleksije će vas sigurno razuvjeriti. Refleksija je postupak koji omogućuje iteraciju kroz sve elemente klase kojima su pridijeljeni atributi te im se pristupa kao svim drugim objektima. A atributi i jesu objekti, kao što smo već naglasili.

Najbolji način pokazivanja primjenjivosti refleksije jest pokazati je na primjeru. Koristit ćemo korisnički definiranu klasu *AutorAtribute* koju ćemo pridijeliti pojedinim metodama i svojstvima. Klasa *AutorAtribute* imat će 3 varijable: *autor* i *verzija* kao pozicijske varijable i *vrstaPosla* kao imenovanu varijablu.

```
using System;
using System.Reflection;

[AttributeUsage(AttributeTargets.Method | AttributeTargets.Property,
       AllowMultiple = true, Inherited = false)]
public class AutorAttribute : Attribute
{
      protected string autor;
      protected double verzija;
      protected string vrstaPosla;

      public AutorAttribute(string autor, double verzija)
      {
            this.autor = autor;
            this.verzija = verzija;
      }

      public string Autor
      {
            get {return autor;}
      }

      public double Verzija
      {
            get {return verzija;}
      }

      public string VrstaPosla
      {
            get {return vrstaPosla;}
            set {vrstaPosla = value;}
      }
}

public class OKCorral
{
      private int numberOfBullets = 100;
      private int numberOfClantons = 4;

      [Autor("Wyatt Earp", 1.0, VrstaPosla = "Testiranje")]
      public void ShotByGun()
      {
```

```
            numberOfBullets--;
            if((numberOfBullets % 2) == 0)
                  numberOfClantons--;
      }

      [Autor("Doc Holliday", 1.0, VrstaPosla = "Programiranje")]
      public void ShotByColt()
      {
            numberOfBullets--;
            numberOfClantons--;
      }

      [Autor("Wyatt Earp", 2.0)]
      public int NumberOfBullets
      {
            get {return numberOfBullets;}
      }

      [Autor("Wyatt Earp", 2.0)]
      [Autor("Doc Holliday", 2.1, VrstaPosla = "Programiranje")]
      public int NumberOfClantons
      {
            get {return numberOfClantons;}
      }
}


public class Test
{
      public static void Main()
      {
            Type t = typeof(OKCorral);
            AutorAttribute Aut;

            foreach(MethodInfo metoda in t.GetMethods())
            {
                  bool Flag = false;
                  foreach(Attribute attr in
                        metoda.GetCustomAttributes(true))
                  {
                        Aut = attr as AutorAttribute;
                        if(Aut != null)
                        {
                              if(!Flag)
                                    Console.WriteLine(metoda.MemberType +
                                                " " + metoda.Name);
                              Console.WriteLine("Autor: {0};  Verzija:
                                          {1:f};  {2}", Aut.Autor,
                                          Aut.Verzija, Aut.VrstaPosla);
                              Flag = true;
                        }
                  }
                  if(Flag)
                        Console.WriteLine();
            }

            foreach(PropertyInfo svojstvo in t.GetProperties())
            {
                  bool Flag = false;
                  foreach(Attribute attr in
                              svojstvo.GetCustomAttributes(true))
```

```
                {
                        Aut = attr as AutorAttribute;
                        if(Aut != null)
                        {
                                if(!Flag)
                                        Console.WriteLine(svojstvo.MemberType +
                                                        " " + svojstvo.Name);
                                Console.WriteLine("Autor: {0};  Verzija:
                                                {1:f};  {2}", Aut.Autor,
                                                Aut.Verzija, Aut.VrstaPosla);
                                Flag = true;
                        }
                }
                if(Flag)
                        Console.WriteLine();
        }
    }
}
```

Primjer 11.3

Ispis:

```
Method ShotByGun
Autor: Wyatt Earp;  Verzija: 1,00;  Testiranje

Method ShotByColt
Autor: Doc Holliday;  Verzija: 1,00;  Programiranje

Property NumberOfBullets
Autor: Wyatt Earp;  Verzija: 2,00;

Property NumberOfClantons
Autor: Wyatt Earp;  Verzija: 2,00;
Autor: Doc Holliday;  Verzija: 2,10;  Programiranje
```

Korisnički definiranu klasu pridijelili smo klasi *OKCorral* koja ima dvije definirane varijable, dva svojstva koji omogućuju čitanje njihovih vrijednosti te dvije metode kojima će se kao i svojstvima pridijeliti atributi.

Refleksija će nam omogućiti pronalazak svih atributa pridijeljenih metodama i svojstvima klase *OKCorral*. Najprije će varijabla *t* postati objekt tipa *System.Type* za klasu *OKCorral*. Ovaj objekt se kasnije koristi u *foreach* petlji tako što će poziv njegove metode *GetMethods* vratiti niz tipa *MethodInfo*, klase koja kao što joj i ime kaže, daje informacije o pojedinoj metodi.

Nakon toga kreće se u drugu *foreach* petlju u kojoj se metodom *GetCustomAttributes* dobiva niz atributa pridijeljenih toj metodi. Je li pojedini atribut tipa klase *AutorAttribute* koja nas u ovom slučaju zanima, dobivamo korištenjem operatora *as* koji će biti različit od *null* ako je atribut tipa *AutorAttribute*. Ako je objekt tipa koji tražimo, vrijednosti njegovih varijabli dobivaju se jednostavnim pozivom *get* svojstava.

Refleksija sve podatke čita zahvaljujući *Metadati*, a to je binarna datoteka koja između ostalog sadrži sve podatke o klasi, njenim članicama, vidljivosti, metodama i argumentima itd.
Varijabla *Flag* je uvedena da bi se naziv metode ispisao samo jednom u slučaju da je metodi pridijeljeno više od jednog atributa.

Što se tiče atributa vezanih za svojstva postupak je potpuno isti samo što se koristi svojstvo *PropertyInfo* za dobivanje informacija o pojedinom svojstvu, a niz takvih objekata osigurava metoda *GetProperties*. Kod svojstava imamo slučaj da su svojstvu *NumberOfClantons* dodijeljena dva autora, Wyatt Earp i Doc Holliday što je omogućeno zbog *true* vrijednosti svojstva *AllowMultiple* klase *AutorAttribute*.

Sad je lako zamisliti upotrebljivost, ali i laku realizaciju refleksije koja će pronaći sve metode koje je Doc Holliday programirao ili npr. sva svojstva koja je Wyatt Earp testirao. Isto tako bi lako bilo doznati bez traženja po kôdu tko je sve bio zadužen za metodu *ShotByColt*, a tko za *ShotByGun*.

Jedan atribut koji nismo spomenuli, a koji ćete često susresti u programima naveden ispred funkcije *Main* je atribut *STAThreadAttribute* ili kraće *STAThread* koji je naznaka da se eventualni poziv neupravljanog kôda izvodi preko tzv. modela *Single Thraeading Apartment* koji osigurava da se međusobno usklade upravljani i neupravljani kôd. Do poziva takvog neupravljanog koda može doći a da i ne znate, ako metoda neke klase iz biblioteke klasa koju koristite, pozove takav neupravljani kôd. O neupravljanom kôdu će više biti riječi u sljedećim poglavljima.

Posredno (indirekno) pozivanje metode

Refleksija se može iskoristiti i za posredno pozivanje neke metode. Iako metodu možemo, naravno, pozvati direktno navođenjem njenog imena i argumenata, to isto možemo postići i korištenjem refleksije. Kad se pronađe željena metoda u nekoj klasi pozivom *GetMethod*, ista ta pronađena metoda može se i pozvati, pozivom metode *Invoke* iz klase *MethodInfo*.

```
using System;
using System.Reflection;

public class Test
{
      public static void Main()
      {
            Type t = Type.GetType("System.Console");
            Type [] ArgArr = new Type[1];
            ArgArr[0] = Type.GetType("System.String");

            MethodInfo Cout = t.GetMethod("WriteLine", ArgArr);
            Object [] Argumenti = new Object[1];

            Argumenti[0] = "Obračun kod OK Korala";
            Cout.Invoke(t, Argumenti);
      }
}
```

Primjer 11.4

Ispis:
Obračun kod OK Korala

Poziva se metoda *Console.WriteLine* bez navođenja iste. Najprije se u varijablu *t* sprema *SystemType* klase *System.Console*. Sve što treba nakon toga napraviti je pozvati metodu *GetMethod* prethodno definiranog objekta *t* navodeći metodi ime metode koja se želi pozvati te argumente u pozivu. Prvi argument je string, dok drugi predstavlja broj i tip argumenata koji se pozivaju s metodom *WriteLine*.

Isti ti argumenti definiraju se kao niz objekata te se prosljeđuju metodi *Invoke* koja u konačnici poziva metodu *WriteLine*.
Ako vam se čini puno posla za jedan običan poziv metode, znajte da bi ga bilo još i više da metoda *WriteLine* nije statička metoda. Želite li pozvati nestatičku metodu morate još i kreirati objekt te klase pozivom metode *Activator.CreateInstance*. Sada ćemo napraviti primjer koji će pozvati metodu *Add* klase *ArrayList* iz imenika *System.Collections*. Primjer će imati manje linija kôda samo zato što ćemo neke naredbe spakirati.

```
using System;
using System.Reflection;
using System.Collections;

public class Test
{
	public static void Main()
	{
		Type t = Type.GetType("System.Collections.ArrayList");
		Object o = Activator.CreateInstance(t);

		t.InvokeMember("Add",
				System.Reflection.BindingFlags.InvokeMethod,
				null, o, new object[] {"Sve je relativno"});
		t.InvokeMember("Add",
				System.Reflection.BindingFlags.InvokeMethod,
				null, o, new object[] {"Ništa nije apsolutno"});

		ArrayList a = (ArrayList)o;
		foreach(string s in a)
			Console.WriteLine(s);
	}
}
```

Primjer 11.5

Ispis:
Sve je relativno
Ništa nije apsolutno

Nakon što je objekt kreiran, pozivom metode *InvokeMember* dva puta pozivamo metodu *Add*. Da bismo provjerili jesmo li je zaista dvaput pozvali, pretvorit ćemo dinamički kreirani objekt *o* u objekt *a* klase *ArrayList* te pozivom *foreach* petlje ispisati sve stringove iz niza.

Kao zaključak ponovimo: naravno da je puno lakše pozvati metode direktno nego na ovaj način, međutim dobro je znati i ovakav način jer ako vam ikada zatreba pretraživanje nekog skupa (*assembly*) te dinamičko pozivanje metoda sadržanih u tom skupu, ovo je jedini način.

Skup je osnovna jedinica svake .Net aplikacije. Svi primjeri u knjizi nakon prevođenja će kreirati skup u obliku tzv. prenosive izvršne (Portable Executive, PE) datoteke. To je u našim primjerima bila EXE datoteka, ali može biti i DLL datoteka ako je rezultat prevođenja izrada biblioteke klasa. Malo prije spomenuta *metadata* je također dio skupa i ako bismo željeli pročitati npr. sve konstruktore iz nekog skupa, koristeći refleksiju, to bismo napravili na sljedeći način:

```
using System;
using System.Reflection;
```

```
public class Test
{
	public static void Main()
	{
		Assembly Skup = Assembly.Load("Test");
		Type[] Tipovi = Skup.GetTypes();

		foreach(Type t in Tipovi)
		{
			Console.WriteLine(t.Name);
			ConstructorInfo[] Arr = t.GetConstructors();

			foreach(ConstructorInfo ci in Arr)
				Console.WriteLine(ci);
		}
	}
}
```

Primjer 11.6

Ispis:

```
Planet
Void .ctor(System.String, SByte)
Void .ctor(System.String, Boolean, Boolean)
Void .ctor(System.String, Boolean, Boolean, SByte, Int32, Single, Single)
Test
Void .ctor()
```

Sa statičkom metodom *Load* učitan je skup *Test* koji je prevedni oblik primjera 5.1 iz 5. poglavlja. Zato osigurajte u putanji tu datoteku *Test.exe* prije pokretanja ovog primjera. Vidimo da su pronađeni svi konstruktori iz skupa, zajedno s tipovima argumenata. Klasa *Planet* ima tri konstruktora, dok klasa *Test* ima samo jedan, onaj podrazumijevani bez argumenata. Tog konstruktora bez argumenata klasa *Planet* nema što se slaže s definicijom da klasa ima podrazumijevani konstruktor sve dok se neki konstruktor ne definira.

Kao što ste mogli primijetiti, zadnjih nekoliko poglavlja skrenuli smo u .Net vode, što nije neočekivano jer vlada međusobna simbioza, čak i sinergija između C# kao programskog jezika i .Net platforme. Tako ćemo nastaviti i u sljedećim poglavljima.

Poglavlje 12:
Višenitnost

Ovo poglavlje obrađuje:

- definiciju niti i višenitnog izvođenja programa
- klasu *Thread*
- prioritet niti
- spajanje niti
- uspavljivanje niti
- uništavanje niti
- sinkronizaciju niti

12.1. Niti (Threads)

Prema definiciji nit (*thread*) je osnovna jedinica uzimanja procesorskog vremena od strane operacijskog sustava. Ako za proces pojednostavljeno kažemo da je to aplikacija koja se trenutno izvodi, onda je nit sastavni dio procesa, odnosno proces se sastoji od jedne ili više niti. Ono gdje niti najviše dolaze do izražaja je slučaj istovremenog izvršavanja više različitih niti, što onda vodi do višezadaćnosti (multitasking), odnosno istovremenog izvršavanja više različitih poslova. (Nećemo sada ulaziti u srž problema pa dokazivati da višezadaćnost u jednoprocesorskim sustavima nije višezadaćnost, nego zbog načina raspodjele procesorskog vremena različitim nitima postoji samo dojam višezadaćnosti).

Ima mnogo primjera svima poznatog izvršavanja programa u nitima. Npr. ako pošaljete neki veliki dokument na ispis na pisač koji nema baš veliku vlastitu memoriju, sigurno ne biste željeli da je to jedina nit programa, odnosno ne biste željeli da se ništa drugo ne može raditi dok se cijeli dokument ne ispiše. (Toga su problema bili svjesni još i programeri u DOS-u koji su za tu svrhu koristili tzv. rezidentne programe).
Ili drugi primjer, radite učitavanje (download) s neke internetske stranice i surfate po istoj toj stranici, naravno istovremeno. Sve to i još mnogo drugih slučajeva istovremenog izvršavanja, omogućuje sposobnost programa da se podijeli u niti koje se onda paralelno izvršavaju.

Klasa *System.Threading.Thread* je ta koja omogućuje izvođenje više različitih niti. Njen konstruktor kao parametar prima instancu predefiniranog delegata *ThreadStart*. Ovaj delegat će sadržavati referencu na metodu koja će biti pozvana pokretanjem niti, s tim da metoda mora imati *void* povratnu vrijednost te ne smije imati argumente jer je tako delegat definiran:

```
public delegate void ThreadStart();
```

pa je deklaracija konstruktora klase *Thread*:

```
public Thread(ThreadStart t)
```

Na sljedećem primjeru vidjet ćemo kako dvije niti rade u istom procesu, prva nit će ispisivati parne brojeve, dok će druga ispisivati neparne brojeve.

```
using System;
using System.Threading;

class NitBrojeva
{
      private Thread t1;
      private Thread t2;

      public void PozoviNiti()
      {
            ThreadStart Odds = new ThreadStart(NeparniBrojevi);
            ThreadStart Evens = new ThreadStart(ParniBrojevi);

            t1 = new Thread(Odds);
            t2 = new Thread(Evens);

            t1.Start();
            t2.Start();
      }

      public void NeparniBrojevi()
```

```
        {
                for(int i=1;i < 100000;i+=2)
                        Console.WriteLine(i);
        }

        public void ParniBrojevi()
        {
                for(int i=2;i <= 100000;i+=2)
                        Console.WriteLine(i);
        }
}



class Test
{
        static void Main()
        {
                NitBrojeva n = new NitBrojeva();
                n.PozoviNiti();
        }
}
```

Primjer 12.1

Definirane su dvije instance delegata: *Odds* koji sadrži referencu na metodu *NeparniBrojevi* i *Evens* koji sadrži referencu na metodu *ParniBrojevi*. Također, definirane su dvije niti kao objekti klase *Thread*: *t1* koja će pozvati metodu za ispis neparnih brojeva i *t2* za ispis parnih brojeva. Niti se pokreću pozivom neopterećene metode *Start* bez argumenata.
Rezultat izvršenja programa će biti naizmjeničan ispis parnih i neparnih brojeva, odnosno metode *ParniBrojevi* i *NeparniBrojevi* će se naizmjenično izvršavati. Kojom će se frekvencijom ta izmjena odvijati i koliko će trajati izvršenje pojedine niti, odlučuju CLR i operacijski sustav.

Prioritet niti

Ako želimo utjecati na njihovu odluku o trenutku pokretanja i trajanju izvršavanja pojedine niti, onda toj niti možemo dodijeliti viši prioritet s enum tipom *ThreadPriority* koji ima 5 različitih vrijednosti:

- Lowest
- BelowNormal
- Normal
- AboveNormal
- Highest

Ako npr. drugoj niti dodijelimo najveći mogući prioritet naredbom:

```
t2.Priority = ThreadPriority.Highest;
```

rezultat izvršenja programa bi vjerojatno bio ispis svih parnih brojeva, pa tek onda neparnih. Kažemo vjerojatno, jer to je ipak samo preporuka CLR-u i operacijskom sustavu. Oni su ipak ti koji u konačnici odlučuju o raspodjeli procesorskog vremena. Očekivano, podrazumijevani prioritet je *ThreadPriority.Normal*.

Spajanje (Join) niti

Spajanje niti je slučaj kada neka nit u metodi koju je pozvala, pozove *Join* metodu druge niti. To je onda znak sustavu da prekine izvršenje prve niti te da je više ne izvršava sve dok ona druga ne završi. Ako bismo u gornjem primjeru u metodu *ParniBrojevi* dodali sljedeće naredbe:

```
if(i == 5000)
      t1.Join();
```

čak ako bi i ostao najviši prioritet izvođenja druge niti (parni brojevi), u trenutku kad varijabla *i* dosegne vrijednost 5000, ona bi prestala s izvođenjem te bi se pozvala prva nit (neparni brojevi) i tek nakon ispisa svih neparnih brojeva, ispisali bi se preostali parni brojevi.

Uspavljivanje (Sleep) niti

Izvršavanje niti može se trenutačno zaustaviti pozivom statičke metode *Thread.Sleep* koja će zaustaviti nit za onoliko milisekundi koliko iznosi vrijednost argumenta u metodi.

```
using System;
using System.Threading;

class Test
{
      static void Main()
      {
            for(int i=0;i<60;i++)
            {
                  Thread.Sleep(1000);
                  Console.WriteLine(i+1);
            }
      }
}
```

Primjer 12.2

Rezultat ispisa će biti da se svake sekunde ispiše jedan broj. Ako se pitate koja se nit zaustavlja ako nismo definirali niti jednu, ne zaboravite da se svaki proces odnosno aplikacija sastoji od najmanje jedne niti. Zato, ako i ne definiramo niti jednu nit, opet se jedna nit izvodi i to je ta čije se izvršenje u ovom slučaju zaustavlja.

Metoda *Sleep* je preopterećena i osim argumenta koji označava broj milisekundi koliko će nit biti suspendirana, argument može biti i objekt klase *TimeSpan*.

```
TimeSpan ts = new TimeSpan(0, 1, 0);
Thread.Sleep(ts);
```

I sama klasa *TimeSpan* ima više konstruktora. Ovaj *ts*, proslijeđen metodi *Sleep* će zaustaviti izvršenje niti na jednu minutu. Argumenti su poredani ovim redoslijedom: broj sati, minuta i sekundi koliko će nit biti zaustavljena.

Poziv statičke metode *Sleep* zaustavlja samo onu nit u kojoj je pozvana. Želimo li da jedna nit zaustavi drugu, onda se to radi s pozivom metode *Suspend*. Nit ostaje zaustavljena sve dok se ne pozove njena metoda *Resume*.

```
public void NeparniBrojevi()
{
      t2.Suspend();
```

```
        for(int i=1;i < 100000;i+=2)
              Console.WriteLine(i);
        t2.Resume();
}
```

Ovaj primjer bi uz pretpostavku istog prioriteta niti *t1* i *t2* zaustavio izvršavanje niti *t2* odmah na početku izvođenja metode *NeparniBrojevi* odnosno niti *t1*. Nit *t2* bila bi zaustavljena sve do kraja izvršenja niti *t1* kada bi poziv metode *Resume* objekta *t2* ponovo pokrenuo nit *t2*.

Uništavanje niti

Normalno, nit se uništava nakon izvršenja metode čija je referenca proslijeđena njenom konstruktoru. Ali nit se može prekinuti i pozivom metode *Abort*.
Ako bismo u prvom primjeru iz ovog poglavlja u metodi *NeparniBrojevi* petlju izveli ovako:

```
for(int i=1;i < 100000;i+=2)
{
        if(i == 11)
              t1.Abort();
        Console.WriteLine(i);
}
```

nit koja ispisuje neparne brojeve prekinula bi se kad vrijednost varijable *i* dosegne 11. Treba naglasiti da će se to izvršiti tek kada izvršenje niti *t1* dođe na red. To će uz visoki prioritet druge niti biti tek kad se ispišu svi parni brojevi.
Zato, ako bi prva nit pokušala ovo:

```
t2.Abort();
```

odnosno zaustaviti drugu nit, to joj ne bi pošlo za rukom jer opet zbog visokog prioriteta niti *t2*, kad bi naredba za poziv metode *Abort* došla na red za izvršavanje, druga nit bi već završila s radom. Drugačije bi bilo kada bi prioriteti bili jednaki, iako i tada trenutak kad će se to dogoditi ovisi, kao što smo rekli, o CLR-u i operacijskom sustavu.

Pitanje: kako bismo izveli da se obje niti zaustave ako korisnik upiše riječ “stop” ?
Prvo što nam pada na pamet jest da bi trebala biti neka treća nit *t3* s ovakvim delegatom:

```
ThreadStart StopBoth = new ThreadStart(StopMethod);
```

dok bi metoda *StopMethod* bila ovako implementirana:

```
public void StopMethod()
{
        if(Console.ReadLine() == "stop")
        {
              t1.Abort();
              t2.Abort();
        }
}
```

Nakon pokretanja svih triju niti, prve dvije bi ispisivale neparne i parne brojeve, a treća bi čekala na unos korisnika. Kada korisnik unese riječ “stop”, pozivom metode *Abort* bi se zaustavile niti *t1* i *t2*.
Ali što ako korisnik pogrešno unese riječ? Novi unos ne bi se prihvatio jer bi nit *t3* već prestala s radom. Očito je da bi se nit *t3* trebala izvršavati u nekoj beskonačnoj petlji:

```
public void StopMethod()
{
      for(;;)
         if(Console.ReadLine() == "stop")
         {
             t1.Abort();
             t2.Abort();
         }
}
```

Sad je taj problem riješen i korisnik može ponovo unositi riječi. Ali još nešto nedostaje, kad korisnik unese riječ "stop", program ne završava s radom jer je ostao u beskonačnoj *for* petlji niti *t3*. Zato nit *t3* mora nakon uništavanja niti *t1* i *t2*, uništiti i samu sebe pozivom metode *Abort*.

```
public void StopMethod()
{
      for(;;)
         if(Console.ReadLine() == "stop")
         {
            t1.Abort();
            t2.Abort();
            t3.Abort();  // može i break
         }
}
```

Stanja niti

Vidjeli smo da nit može biti u nekoliko različitih stanja. Stanje niti može se dobiti svojstvom *ThreadState*, a sljedeća tablica prikazuje objedinjeno sva moguća stanja niti.

Stanje niti	**Opis**
`Initialized (Inicijalizirano)`	Nit je inicijalizirana (kreiran je objekt s proslijeđenim delegatom), ali još nije startala.
`Ready (Spremna)`	Nit čeka na slobodno procesorsko vrijeme da bi počela ili nastavila s radom.
`Running (U izvršavanju)`	Nit trenutno koristi procesorsko vrijeme, odnosno izvršava se.
`Standby (U pripravi)`	Nit se sprema na izvođenje.
`Terminated (Zaustavljena)`	Nit je završila s radom.
`Transition (U tranziciji)`	Nit čeka na korištenje nekog drugog resursa, različitog od procesora, da bi mogla nastaviti s izvođenjem.
`Unknown (Nepoznato stanje)`	Stanje niti je nepoznato.
`Wait (U čekanju)`	Nit nije spremna za korištenje procesorskog vremena jer čeka da se neka periferna operacija završi ili da postane slobodna.

Na kraju da zaključimo, niti su definitivno pozitivna i dobra stvar, ali ipak ne treba uletjeti u zamku pa ih koristiti i tamo gdje nema potrebe, da ne bismo došli u situaciju da procesor više vremena troši na prebacivanje s niti na nit (i pritom mora zapamtiti gdje je stao u pojedinoj

niti!) nego na samo izvršavanje niti. Osim toga, korištenje niti bez obzira na sve, ipak komplicira program.
Ali ako se neki posao svodi na dugotrajno računanje ili ako čekate na korisnikovu reakciju, praktički nemate izbora nego izvesti to kao zasebnu nit.

Ostaje na kraju ona općenita preporuka koje se vjerojatno najteže držati. Koristite niti tamo gdje to ima smisla i potrebe, odnosno gdje njihovim korištenjem više dobivate nego gubite.

12.2. Sinkronizacija niti

Problemi s korištenjem niti pojave se brzo, neočekivano i nepredvidivo. To je dovoljno razloga da s velikim oprezom pristupite izradi niti i njihovoj sinkronizaciji. Osobito su problemi na vidiku u slučaju kada niti pristupaju istim resursima kao npr. datotekama. Zamislite situaciju da dvije niti trebaju pročitati neki podatak iz datoteke te ga povećati za jedan i tako povećanog ponovo spremiti u datoteku. Sljedeće su operacije potrebne:

1. Pročitaj varijablu iz datoteke
2. Povećaj varijablu za jedan
3. Vrati varijablu natrag u datoteku

I najpovršniji pogled na ovo uočava potencijalni problem. Zamislite ovakav slijed događaja:

1. Prva nit čita varijablu iz datoteke
2. Prva nit povećava varijablu za jedan
3. Druga nit čita varijablu iz datoteke
4. Prva nit vraća varijablu u datoteku
5. Druga nit povećava varijablu za jedan
6. Druga nit vraća varijablu u datoteku

Očekivano bi bilo da ovakav niz operacija poveća varijablu iz datoteke za dva. Međutim, rezultat će biti povećanje samo za jedan jer će druga nit povećati za jedan varijablu koja je bila u datoteci prije nego što ju je prva nit povećala za jedan. Dakle, problem je što se korak 3 dogodio prije koraka 4.

Sad ćemo simulirati jedan sličan slučaj s tim da će varijablu iz datoteke predstavljati statička varijabla neke klase, dok ćemo vrijeme potrebno da bi se obavilo čitanje i pisanje iz i u datoteku zamijeniti metodom *Sleep*.

```
using System;
using System.Threading;

class Count
{
      public static int Counter = 100;

      public void Counting1()
      {
            while(Count.Counter > 0)
            {
                  int x = Count.Counter;
                  x--;
                  Thread.Sleep(5);
                  Count.Counter = x;
```

```
                    Console.WriteLine(Count.Counter);
                }
        }

        public void Counting2()
        {
                while(Count.Counter > 0)
                {
                        int x = Count.Counter;
                        x--;
                        Thread.Sleep(5);
                        Count.Counter = x;
                        Console.WriteLine(Count.Counter);
                }
        }

}

class Test
{
        static void Main()
        {
                Count Odbrojavanje = new Count();
                Thread t1 = new Thread(new
                                ThreadStart(Odbrojavanje.Counting1));
                t1.Start();

                Thread t2 = new Thread(new
                                ThreadStart(Odbrojavanje.Counting2));
                t2.Start();
        }
}
```

Primjer 12.3

Definirane su dvije niti *t1* i *t2* na metode objekta *Odbrojavanje* klase *Count*. I jedna i druga nit čitaju statičku varijablu *Counter* te joj smanjuju vrijednost za jedan pomoću varijable *x*. Između uzimanja vrijednosti varijable *Counter* i njenog smanjivanja stavili smo metodu *Sleep(5)* koja bi trebala simulirati vrijeme koje bi se utrošilo ako to radimo s datotekom. Budući da je riječ o statičkoj varijabli koja pripada klasi, očekivali bismo da rezultat ispisa budu brojevi od 99 do 0.

Međutim događa se onaj gore spomenuti problem sinkronizacije niti, gdje se korak 3 izvršava prije koraka 4. Naime između čitanja vrijednosti varijable *Counter* i njenog smanjivanja, događa se da druga nit napravi to isto, pa je rezultat ispisa onaj očekivani od 99 do 0, ali s dupliranim ispisom brojeva.

Srećom, C# nudi relativno jednostavan izlazak iz ovakve situacije s ključnom riječi *lock*, kojom se označi takav kritični dio koda. To je znak da nijedna druga nit neće narušiti izvršenje naredbi u bloku dok nit u kojoj je blok tako označen ne završi sa svim naredbama u bloku. U primjeru s čitanjem i upisom u datoteku to bi značilo da nakon što nit pristupi datoteci za čitanje varijable, nijedna druga nit ne može pročitati istu tu vrijednost sve dok prva nit ne vrati povećanu vrijednost varijable u datoteku.

U primjeru s odbrojavanjem to bi ovako izgledalo. Metode *Counting1* i *Counting2* ovako bi izgledale:

```
public void Counting1()
{
	while(Count.Counter > 0)
	{
		lock(this)
		{
			int x = Count.Counter;
			x--;
			Thread.Sleep(5);
			Count.Counter = x;
			Console.WriteLine(Count.Counter);
		}
	}
}
```

U samom primjeru nije bilo nužno definirati dvije metode *Counting1* i *Counting2* budući da rade isto. Delegati dviju različitih niti mogu imati referencu na jednu te istu metodu.
Kao argument od *lock* obično se navodi objekt *this*, ako je riječ o instanci klase, a ako je riječ o statičkoj metodi onda se kao argument navodi *typeof(ime klase)*.

U rezultatima ispisa može se primijetiti jedna zanimljivost. U slučaju kada se *lock* ne koristi, ispišu se po dva puta vrijednosti od 99 do 0, a ako se *lock* koristi ispišu se po jednom vrijednosti od 99 do –1.
Zašto sada do –1?

Malo bolja analiza i to otkriva. Kada druga nit (ili prva, nije bitno) uđe s vrijednosti varijable *Counter* = 1 u *while* petlju, uvjet za izvršenje petlje je zadovoljen i nit krene u izvršenje naredbi u bloku petlje. Ali tu je odmah čeka naredba *lock* koja joj kaže, čekaj dok prva nit ne završi taj blok. Kad prva nit završi blok, počinje se izvršavati *lock* blok druge niti, ali sada je *Counter* s prvom niti već smanjen na 0 (a uvjet u *while* petlji je prošao i više ga se u ovom prolazu ne provjerava!), te ga blok u drugoj niti dodatno smanji za jedan.
Ako bismo u *lock* blok stavili cijelu *while* petlju onda se to ne bi dogodilo i ispis bi bio onaj očekivani od 99 do 0.

Na računalu na kojem je ovo testirano *Counter* je otišao do –1. Ipak, sjetite se da raspodjelu procesorskog vremena dodjeljuju CLR i operacijski sustav tako da ovakav slučaj i ne mora biti u nekom drugom sustavu.

Poglavlje 13:
Tokovi

Ovo poglavlje obrađuje:

- rad s datotekama i mapama
- tokove
- klase *Directory* i *DirectoryInfo*
- klase *File* i *FileInfo*
- razliku između tekstne i binarne datoteke
- pisanje podataka u datoteku
- čitanje podataka iz datoteke
- klasu *Environment*
- vrste kodiranja

13.1. Rad s datotekama i mapama

Ako vam je ovih zadnjih nekoliko poglavlja djelovalo apstraktno, ovo bi trebalo izgledati "ovozemaljskije" jer bit će riječ o datotekama i mapama, dakle o onome što svatko gotovo svaki dan radi s raznim explorerima. Naravno, mi ćemo koristiti C#, klase i objekte što uostalom koriste i svi exploreri skriveno iza GUI-a.
Najprije ćemo obraditi rad s mapama, a nakon toga ćemo vidjeti kako sve to izgleda u kombinaciji s datotekama.

Rad s mapama

Dvije su klase za rad s mapama u .Net Frameworku

- Directory
- DirectoryInfo

Zanimljivo, iako su Windowsi u korisničkom okruženju promijenili staro ime *directory* u *folder*, u nazivima klasa i metoda je ostalo ime *directory*. Iako tu ne možemo govoriti o kompatibilnosti sa starim API funkcijama na razini izvršenja metoda, vjerojatno se iz sintaktičkih razloga (čitaj navike) zadržalo ime *directory*.
Klasa *Directory* je zapečaćena klasa sa samo statičkim članovima, dok je *DirectoryInfo* također zapečaćena, ali sa samo nestatičkim članovima.

Sljedeće tablice pokazuju neke metode i svojstva ovih dviju klasa:

Klasa Directory	
Metoda	**Opis**
`CreateDirectory`	Kreira mapu navedenu kao argument
`Delete`	Briše mapu, a ako je drugi argument *true* briše i sve podmape zajedno s njihovim sadržajima.
`Exists`	Vraća *true* ako mapa postoji, inače vraća *false*
`GetCreationTime`	Vraća kao objekt tipa *DateTime* datum i vrijeme kad je mapa kreirana
`GetCurrentDirectory`	Vraća naziv trenutne mape
`GetDirectories`	Vraća kao niz stringova mape unutar navedene mape koje udovoljavaju navedenom kriteriju
`GetFiles`	Vraća kao niz stringova nazive datoteka koje se nalaze u navedenoj mapi, a koje udovoljavaju navedenom kriteriju
`GetLogicalDrives`	Vraća nazive logičkih diskova kao niz stringova u obliku X:\
`Move`	Seli mapu i cijeli njen sadržaj na drugo odredište
`SetCurrentDirectory`	Postavlja mapu kao trenutnu.

Klasa DirectoryInfo	
Metoda	**Opis**
`Create`	Kreira mapu koju predstavlja objekt tipa

	ove klase
`CreateSubdirectory`	Kreira podmapu u mapi koju predstavlja objekt
`Delete`	Briše mapu koju predstavlja objekt, ako je argument *true* briše i podmape sa svim sadržajem
`GetDirectories`	Vraća kao niz stringova mape unutar mape koju predstavlja objekt, a koje udovoljavaju navedenom kriteriju
`GetFiles`	Vraća kao niz stringova nazive datoteka koje se nalaze u mapi koju objekt predstavlja, a koje udovoljavaju navedenom kriteriju
Svojstvo	**Opis**
`CreationTime`	Vraća kao objekt tipa *DateTime* datum i vrijeme kad je mapa kreirana
`Exists`	Vraća *true* ako mapa postoji
`FullName`	Vraća puni naziv mape koju objekt predstavlja
`Name`	Vraća naziv mape koju objekt predstavlja
`Parent`	Vraća naziv roditeljske mape
`Root`	Vraća naziv korijenske mape za mapu koju objekt predstavlja

Vidimo da se dosta metoda preklapa u obje klase, ali uz osnovnu razliku da se kod klase *DirectoryInfo* mora kreirati objekt za određenu mapu, pa se tek onda s tom mapom može raditi. Mapa neće biti kreirana instanciranjem objekta, nego se to mora napraviti pozivom metode *Create*.

```
DirectoryInfo dir = new DirectoryInfo("c:\\NewFolder");
dir.Create();     // Tek sad je mapa kreirana
```

Sljedeći primjer će kreirati mapu i nekoliko podmapa koristeći klasu *DirectoryInfo*:

```
using System;
using System.Threading;
using System.IO;

class Test
{
      public static void Main()
      {
            string Spices = "Spice Girls";
            string [] SpiceGirls = {"Melanie", "Geri", "Victoria", "Emma",
                                                        "Melanie"};
            DirectoryInfo dir = new DirectoryInfo(Spices);
            if(dir.Exists == false)
                  dir.Create();
            foreach(string s in SpiceGirls)
                  dir.CreateSubdirectory(s);

            Console.WriteLine("Provjerite jesu li mape kreirane prije nego
                                    što se izbrišu.");
            Thread.Sleep(20000);
```

```
            dir.Delete(true);
        }
}
```

Primjer 13.1

Ispis:
Provjerite jesu li mape kreirane prije nego što se izbrišu.

Na početku je potrebno uključiti imenik *System.IO* jer se u njemu nalaze klase za rad s mapama i datotekama. Primjer će kreirati mapu *Spice Girls* sa 4 podmape s imenima članica grupe. Da bismo izbjegli "hardcoding", imena podmapa definirali smo kao niz stringova. Najprije se kreira osnovna mapa, a onda unutar nje podmape. Prednost ovakvog pristupa kreiranju mapa u odnosu na klasu *Directory* i njene statičke metode je u tome što u ovom slučaju objekt predstavlja mapu. Zato nema onog šetanja po mapama i podmapama prije kreiranja pojedine mape. Sve što se radi odnosi se na mapu koju predstavlja objekt.

Prije kreiranja osnovne mape *Spice Girls* provjeravamo postoji li mapa već, iako nije nikakva greška pokušati kreirati mapu koja već postoji. Zato, iako imamo pet djevojaka u grupi, budući da dvije imaju isto ime, bit će kreirane samo 4 podmape.
Na kraju se mape i brišu pozivom metode *Delete* kojoj se kao argument prosljeđuje vrijednost *true* što je znak da se izbrišu sve podmape zajedno sa svim datotekama koje bi one sadržavale. Metoda *Sleep* osigurava 20 sekundi stanke prije brisanja tek da se možete uvjeriti da su datoteke kreirane.

Napomena:
Mape izbrisane na ovaj način metodom *Delete* brišu se nepovratno, kao i sadržaji mapa, odnosno **izbrisane mape ne šalju se u koš za smeće (Recycle Bin).**

Rad s datotekama
Slično kao kod mapa, i kod datoteka postoje dvije klase za rad s njima,

- File
- FileInfo

a isto kao i kod mapa, obje su klase zapečaćene. Klasa *File* sadrži samo statičke članove, dok klasa *FileInfo* samo nestatičke što vodi do svih onih razlika koje iz toga proizlaze. Najlakši način kreiranja niza objekata klase *FileInfo* je poziv metode *GetFiles* iz klase *DirectoryInfo*. Prije nego što napravimo jedan takav primjer pogledajmo kakve sve metode i svojstva postoje u klasama *File* i *FileInfo*.

Klasa File	
Metoda	**Opis**
`Copy`	Kopira datoteku s jednog mjesta na drugo
`Create`	Kreira praznu datoteku
`Delete`	Briše datoteku
`Exists`	Vraća *true* ako datoteka postoji
`GetAttributes`	Vraća enum tip *FileAttributes* s atributima datoteke
`GetCreationTime`	Vraća *DateTime* objekt s vremenom kreiranja datoteke
`Open`	Otvara datoteku te vraća objekt *FileStream*

	za tu datoteku
`SetAttributes`	Postavlja atribute za datoteku

Klasa FileInfo	
Metoda	**Opis**
`Create`	Kreira datoteku
`Delete`	Briše datoteku
`Open`	Otvara datoteku te vraća objekt *FileStream* za tu datoteku
`OpenRead`	Otvara datoteku za čitanje
`OpenWrite`	Otvara datoteku za pisanje
Svojstvo	**Opis**
`Attributes`	Vraća ili postavlja atribute datoteke
`CreationTime`	Vraća ili postavlja datum i vrijeme kad je datoteka kreirana
`Directory`	Vraća instancu *DirectoryInfo* mape u kojoj se datoteka nalazi
`DirectoryName`	Vraća string s imenom mape u kojoj se datoteka nalazi
`Exists`	Vraća *true* ako datoteka postoji
`FullName`	Vraća puni naziv datoteke koju objekt predstavlja
`Length`	Vraća veličinu datoteke u byte-ovima

Primjer koji ćemo pokazati imat će veliki zadatak pred sobom. Trebati će pronaći mapu s najviše datoteka. Naravno, na cijelom disku. Iako bi to bio stvarno veliki zadatak ako bismo morali koristiti Windows Explorer, uz pomoć C# i klasa *DirectoryInfo* i *FileInfo* sve će izgledati puno lakše.

```
using System;
using System.IO;

class FilesAndFolders
{
	private DirectoryInfo Dir;
	private int maxFiles = 0;
	private string hasMostFiles = "";
	private string SystemFolder =
		Environment.GetEnvironmentVariable("SystemRoot");

	public FilesAndFolders(string root)
	{
		Dir = new DirectoryInfo(root);
	}

	public void OpenDirectory()
	{
		DirectoryInfo[] Directories = null;
		FileInfo [] Files = null;

		try
		{
			Directories = Dir.GetDirectories();
			Files = Dir.GetFiles();
```

```
            }
            catch(Exception)
            {
                  Console.WriteLine("Nemam pristup za: " + Dir.FullName);
                  return;
            }

            if(String.Compare(Dir.FullName, 0, SystemFolder, 0,
                        SystemFolder.Length, true) != 0
                              &&
                        Files.Length > MaxFiles)
            {
                  MaxFiles = Files.Length;
                  HasMostFiles = Dir.FullName;
            }

            foreach(DirectoryInfo NextDir in Directories)
            {
                  Dir = NextDir;
                  OpenDirectory();
            }
      }

      public int MaxFiles
      {
            get
            {
                  return maxFiles;
            }

            set
            {
                  maxFiles = value;
            }
      }

      public string HasMostFiles
      {
            get
            {
                  return hasMostFiles;
            }

            set
            {
                  hasMostFiles = value;
            }
      }
}

class Test
{
      public static void Main()
      {
            FilesAndFolders ff = new FilesAndFolders("c:\\");
            ff.OpenDirectory();

            Console.WriteLine();
            Console.WriteLine("{0} datoteka u mapi {1}", ff.MaxFiles,
                  ff.HasMostFiles);
```

```
        }
}
```

Primjer 13.2

Sve se događa u klasi *FilesAndFolders*. Pretraživanje na način ”ako nađeš mapu u mapi, pretraži i tu mapu na isti način” je idealno za poziv jedne rekurzivne metode kao što je *OpenDirectory*. Kad se mapa pronađe, čitaju se sve datoteke iz nje u niz stringova *Files*, tipa *FileInfo*. Ako je veličina tog niza, odnosno broj datoteka veći od varijable *maxFiles*, onda vrijednost *maxFiles* postaje jednaka tom broju datoteka u mapi te je od tog trenutka to najveći broj datoteka u jednoj mapi. Tako je na kraju vrijednost koja ostane u *maxFiles* najveći broj datoteka u jednoj mapi. Isto se događa i sa stringom *hasMostFiles* u kojem će biti zapisan naziv mape s najviše datoteka.

Iz pretrage smo isključili mapu s operacijskim sustavom isključivo iz razloga što bi vrlo vjerojatno neka od *service pack* mapa iz te sistemske mape dala najveći broj datoteka. To je postignuto korištenjem statičke metode *StringCompare*, dok se sistemska mapa dobije pozivom također statičke metode:

```
Environment.GetEnvironmentVariable("SystemRoot");
```

Konstruktor klase prima kao argument naziv mape koja će biti korijenska u pretraživanju, što znači da se pretražuju mape iz te mape i svih mapa ispod nje.

Program relativno brzo prijeđe cijeli disk, a iz pretrage su isključene mape kojima kao korisnik nemamo prava pristupa. To su svi ostali korisnici na računalu s administratorskim ovlastima. To je postignuto tako što se uhvati iznimka koja se generira u slučaju pokušaja pristupa takvim mapama, te se ne dopušta prekid traženja što bi inače bio slučaj kad iznimku ne bismo uhvatili.

Možete se pitati kako provjeriti je li pronađena baš mapa s najviše datoteka. Imate dvije mogućnosti:

- dobro analizirati programski kôd i uvjeriti se da radi upravo ono što bi trebao
- pokrenuti Windows Explorer i provjeravati i provjeravati. . .

Sigurni smo da ovaj drugi način nećete koristiti, pa evo i jedan treći, a to je da onu mapu koja vam je pronađena kao ona s najviše datoteka jednostavno kopirate na neko drugo mjesto. Dodajte joj još jednu datoteku i pokrenite novo traženje. Sada bi mapa s najviše datoteka trebala biti ta novoformirana.

Ovakav jedan program uz malu doradu mogao bi napraviti još mnogo toga kao npr.

- pronaći najveću datoteku na disku ili možda 10 najvećih
- pronaći sve datoteke s nastavkom (ekstenzijom) “bak” i izbrisati ih
- pronaći sve Word ili Excel dokumente na disku itd.

U prvom slučaju trebali bismo proći svaki put kroz cijeli niz *Files* i ispitivati veličinu svake datoteke te koristiti postupak kao s varijablom *maxFiles* za broj datoteka u mapi.
U drugom slučaju opet bismo koristili niz *Files* i uz pomoć *foreach* petlje to bi ovako izgledalo:

```
foreach(FileInfo f in Files)
```

```
if(f.Extension.ToLower() == ".bak")
      f.Delete();
```

Metoda *ToLower* je tu da osigura uspoređivanje malih slova s malim slovima. Primijetite također da je točka (‘.’) dio ekstenzije datoteke.
OK, drugo i treće može i *Search* u Windows OS-u, ali ovakav pristup omogućava dinamičko izvođenje u sklopu izvršenja nekog programa.

Naš primjer pretražuje logički disk, a ne fizički što znači da ako imate više particija na disku morate ponoviti postupak za svaku particiju. Ali i to se može automatizirati uz pomoć statičke metode *Directory.GetLogicalDrives* koja će vratiti nazive logičkih diskova kao niz stringova u obliku X:\\.

Klasa Environment
Gore smo spomenuli klasu *Environment* za dobivanje naziva sistemske mape. Klasa *Environment* sadrži još neke zanimljive članove kao što je *SpecialFolder* tipa enum. Prosljeđivanjem ovog člana metodi *GetFolderPath* možemo dobiti nazive nekih mapa s posebnim značenjem.

Tako će npr. ovaj poziv:
`Environment.GetFolderPath(Environment.SpecialFolder.DesktopDirectory)`
dati naziv mape radne površine,

`Environment.GetFolderPath(Environment.SpecialFolder.MyMusic)`
će dati naziv mape “Moja glazba”, dok će

`Environment.GetFolderPath(Environment.SpecialFolder.StartUp)`
dati naziv mape s programima iz startne grupe.

Kada ovakvim pozivom dobijemo naziv mapa, onda taj naziv možemo proslijediti konstruktoru klase *DirectoryInfo*. Kreirat će se objekt za tu mapu te njoj i njenim datotekama možemo pristupati kao i svim ostalim mapama.

Očito je da su velike mogućnosti u ovakvom pristupu mapama i datotekama, ali je moguća i potencijalna šteta. Zato na kraju još jednom ponavljamo: pri korištenju metoda *Delete* budite vrlo oprezni, osobito u petljama.

13.2. Čitanje i pisanje tekstnih datoteka

Ne vjerujemo da se može napisati knjiga o nekom programskom jeziku, a da se barem ukratko ne obradi pisanje i čitanje datoteka. Odmah na početku treba razlikovati dva pojma:

- datoteka
- tok (stream)

Datoteka je skup podataka u nekoj mapi koja ima svoje ime, putanju (path), veličinu, atribute itd. Onog trenutka kad se datoteka otvori za pisanje ili čitanje njeni podaci postaju tok. Tok je širi pojam od podataka koji se pišu ili čitaju iz datoteke, jer npr. tok može obuhvaćati i prijenos podataka preko mreže ili prijenos blokova memorije.

Datoteka u koju se spremaju podaci može biti tekstna ili binarna datoteka. Tekstne datoteke su jednostavnije za razumijevanje, ali je rad s takvim datotekama sporiji i nezgrapniji nego s binarnim datotekama. Osnovna razlika među njima je što se kod tekstnih datoteka podaci

spremaju kao čisti tekst i ako u tekstnu datoteku upišemo tekst ‘milijun’, onda će taj tekst točno tako biti spremljen u datoteci i ona će biti veličine 7 byte-ova. Ako u tekstnu datoteku upišemo broj 1000000 i on će biti točno tako upisan u datoteku i ona će također biti veličine 7 byte-ova, ali ne zato što se milijun s brojevima piše kao 1000000, nego zato što igrom slučaja oba stringa imaju po 7 znakova. To znači da se u tekstnu datoteku sve upisuje u obliku niza znakova.
Kod binarnih datoteka je drugačije, ali to ćemo obraditi u sljedećem poglavlju.

Sljedeći primjer će pokazati upis u tekstnu datoteku i čitanje iz nje. Koristit ćemo klase *StreamWriter* i *StreamReader* koje nasljeđuju apstraktne klase *TextWriter* i *TextReader* iz imenika *System.IO*.

```
using System;
using System.IO;

class Test
{
	private const string Big = "bignumber.txt";

	static void Main()
	{
		StreamWriter FWrite = null;
		try
		{
			FWrite = new StreamWriter(Big, false,
					System.Text.Encoding.ASCII);

			FWrite.WriteLine("milijun");
			int i = 1000000;
			FWrite.WriteLine(i);

			FWrite.WriteLine("milijarda");
			i = 1000000000;
			FWrite.WriteLine(i);
		}
		catch (Exception e)
		{
			Console.WriteLine(e.Message);
		}
		finally
		{
			if(FWrite != null)
				FWrite.Close();
		}

		StreamReader FRead = null;
		try
		{
			FRead = new StreamReader(Big,
					System.Text.Encoding.ASCII);
			string s;
			while((s = FRead.ReadLine()) != null)
				Console.WriteLine(s);

		}
		catch (Exception e)
		{
			Console.WriteLine(e.Message);
		}
```

```
            finally
            {
                  if(FWrite != null)
                        FWrite.Close();
            }

            FileInfo f = new FileInfo(Big);
            Console.WriteLine(f.Length + " bytes");
      }
}
```

Primjer 13.3

Ispis:

```
milijun
1000000
milijarda
1000000000
41 bytes
```

Konstruktor (jedan od 7) *StreamWriter* klase prima tri argumenta:

- naziv datoteke
- *bool* tip koji ako je *false*, onda će datoteka, u slučaju da već postoji, biti prebrisana, a ako je *true* onda će se podaci dodati na kraj datoteke
- tip *System.Text.Encoding* koji određuje vrstu kodiranja

Podaci se upisuju u datoteku metodama *Write* i *WriteLine* koje se razlikuju jedino po tome što ova druga dodaje znak za novi red. Obje ove datoteke kao argument mogu primiti različite varijable kao *int*, *long*, *double*, *string* itd., ali će se podatak uvijek upisati kao niz znakova kao što je objašnjeno na početku teksta. Datoteku je na kraju potrebno i zatvoriti metodom *Close*, što se osigurava u *finally* bloku u slučaju iznimke.

Kod klase *StreamReader* je slično, može se vidjeti da će se čitanje svih linija iz datoteke izvesti *while* petljom zahvaljujući činjenici da *ReadLine* čita sekvencijalno linije iz datoteke te da vraća *null* string ako u datoteci više nema linija.

Na kraju, koristeći svojstvo *Length* klase *FileInfo* dobije se veličina datoteke. Veličina je 41 byte jer je to:

- 7 byte-ova za “milijun”
- 7 byte-ova za “1000000”
- 9 byte-ova za “milijarda”
- 10 byte-ova za “1000000000”
- 8 byte-a za 4 znaka za novi red

Klasa *StreamReader* ima dosta metoda i svojstava, u sljedećoj tablici ćemo navesti samo one iz grupe *read*. Za ostale pogledajte dokumentaciju .Net Frameworka.

Klasa StreamReader	
Metoda	**Opis**
`Read`	Čita sljedeći karakter iz toka
`ReadBlock`	Čita maksimalno onoliko znakova iz toka koliko je navedeno u argumentu
`ReadLine`	Čita liniju iz toka

`ReadToEnd`	Čita sve podatke od trenutne pozicije do kraja datoteke

Vrsta kodiranja

Treći parametar u klasi *StreamWriter* određuje vrstu kodiranja. Naš primjer radi s osnovnim ASCII setom od 256 znakova, ali se može koristiti i Unicode set znakova kod kojeg je svaki znak veličine 2 byte-a pa je i broj mogućih znakova puno veći, 2^{32}. Zato korištenje Unicode seta rezultira dvostruko većom datotekom nego u slučaju ASCII seta.
Međutim zbog kompatibilnosti prvih 256 znakova u Unicode setu je isto kao i kod ASCII seta znakova s Latin 1 proširenim setom (od znaka 128 - 255).

Ako se argument za vrstu kodiranja uopće ne navede, onda se koristi podrazumijevana vrijednost, a to nije ni ASCII ni Unicode set nego tzv. UTF-8 set karaktera. Kod UTF-8 seta svakom Unicode znaku se izbace vodeći byte-ovi s vrijednošću nula, dok se znakovi s vrijednošću iz prvog dijela ASCII tablice pretvore u ASCII set. Zato će ovaj naš primjer kreirati iste datoteke ako odaberemo ASCII kodiranje ili ako po "defaultu" ostavimo UTF-8 kodiranje jer smo koristili samo znakove iz osnovnog ASCII seta.

Klasa FileStream

Datoteke se mogu čitati i pisati i metodama klase *FileStream*. Njena posebnost je u tome što pisanje i čitanje radi isključivo tipom *byte*. Tako bismo čitanje datoteke iz gornjeg primjera mogli i ovako izvesti:

```
int i;
FileStream fs = new FileStream(Big, FileMode.Open);
while((i = fs.ReadByte()) > 0)
      Console.Write((char)i);
fs.Close();
```

Metoda *ReadByte* vraća pročitani *byte* iz datoteke pretvoren u *int* tako da *byte* 0x67 ili veliko slovo 'C' postaje *int* 0x00000067 i zato nema gubitka vrijednosti u konverziji. Za čitanje više byte-ova odjednom može se koristiti metoda:

```
ReadByte(byte [] buffer, int offset, int count)
```

gdje je:

- *buffer* – niz tipa *byte* u koji će se spremiti pročitani podaci iz datoteke
- *offset* – indeks u nizu *buffer* od kojeg će se spremiti podaci
- *count* – maksimalan broj byte-ova koji će se pročitati

13.3. Čitanje i pisanje binarnih datoteka

Ako kažemo da su binarne datoteke sve one koje nisu tekstne, nećemo pogriješiti. Ipak to nikako ne možemo prihvatiti kao imalo ozbiljniju definiciju binarne datoteke. Zato ćemo reći ovako:
Binarna datoteka je datoteka u koju se varijabla zapisuje tako da zauzima isto onoliko byte-ova kao i u memoriji računala. To ćemo najbolje vidjeti na primjeru u kojem ćemo upisati podatke kao i u prethodnom primjeru. Koristit ćemo klase *BinaryWriter* i *BinaryReader*.

```
using System;
using System.IO;
```

```
class Test
{
	private const string Big = "bignumber.bin";

	static void Main()
	{
		Stream IzlazniTok = File.OpenWrite(Big);
		BinaryWriter FWrite = null;

		try
		{
			FWrite = new BinaryWriter(IzlazniTok);

			FWrite.Write("milijun");
			int i = 1000000;
			FWrite.Write(i);

			FWrite.Write("milijarda");
			i = 1000000000;
			FWrite.Write(i);
		}
		catch (Exception e)
		{
			Console.WriteLine(e.Message);
		}
		finally
		{
			if(FWrite != null)
				FWrite.Close();
		}

		Stream UlazniTok = File.OpenRead(Big);
		BinaryReader FRead = null;

		try
		{
			FRead = new BinaryReader(UlazniTok);

			Console.WriteLine(FRead.ReadString());
			Console.WriteLine(FRead.ReadInt32());
			Console.WriteLine(FRead.ReadString());
			Console.WriteLine(FRead.ReadInt32());
		}
		catch (Exception e)
		{
			Console.WriteLine(e.Message);
		}
		finally
		{
			if(FWrite != null)
				FWrite.Close();
		}

		FileInfo f = new FileInfo(Big);
		Console.WriteLine(f.Length + " bytes");
	}
}
```

Primjer 13.4

Ispis:

```
milijun
1000000
```

milijarda
1000000000
26 bytes

Klase *BinaryWriter* i *BinaryReader* kao prvi argument imaju tokove *IzlazniTok* i *UlazniTok* koji se dobivaju kao povratne vrijednosti od *OpenWrite* i *OpenRead*, koje su obje statičke metode klase *File*.

Odmah vidimo razliku u veličini datoteke. Iako smo zapisali istu ”količinu” podataka, veličina u slučaju s binarnom datotekom je osjetno manja. Dok kod stringova nema razlike jer je veličina stringa jednostavno broj znakova, kod tipa *int* (a i kod ostalih vrijednosnih tipova) se bez obzira na vrijednost, zapisuje uvijek onoliko byte-ova koliko varijabla tog tipa zauzima u memoriji. Zato će se i 1000000 i 1000000000 zapisati sa 4 byte-a.
Da bi se podaci iz binarne datoteke mogli čitati, mora se znati kako su zapisani i zato se datoteka čita u redoslijedu kako su podaci i upisivani: *string, int, string, int*.

Binarne datoteke veliku primjenu imaju kod zapisivanja niza varijabli. To ćemo pokazati na sljedećem primjeru gdje ćemo u binarnu datoteku zapisati niz *double* varijabli:

```
using System;
using System.IO;

class Test
{
	private const string Big = "bignumber.bin";

	static void Main()
	{
		double [] Constants = new double[] {1.60217653, 3.1415926,
				2.718, 9.80665, 6.6742};
		Stream IzlazniTok = File.OpenWrite(Big);
		BinaryWriter FWrite = null;

		try
		{
			FWrite = new BinaryWriter(IzlazniTok);
			foreach(double d in Constants)
				FWrite.Write(d);
		}
		catch (Exception e)
		{
			Console.WriteLine(e.Message);
		}
		finally
		{
			if(FWrite != null)
				FWrite.Close();
		}

		Stream UlazniTok = File.OpenRead(Big);
		BinaryReader FRead = null;

		try
		{
			FRead = new BinaryReader(UlazniTok);
			while(UlazniTok.Position < UlazniTok.Length)
				Console.WriteLine(FRead.ReadDouble());
```

```
            }
            catch (Exception e)
            {
                  Console.WriteLine(e.Message);
            }
            finally
            {
                  if(FWrite != null)
                        FWrite.Close();
            }

            FileInfo f = new FileInfo(Big);
            Console.WriteLine(f.Length + " bytes");
      }
}
```

Primjer 13.5

Ispis:
1,60217653
3,1415926
2,718
9,80665
6,6742
40 bytes

Definirali smo niz *Constants* s 5 decimalnih brojeva s različitim brojem decimala koje se zapisuju u binarnu datoteku koristeći metodu *Write* klase *BinaryWriter*. Odmah vidimo da je veličina kreirane datoteke 40 byte-ova, dakle točno 5 * veličina tipa *double*, odnosno svaki element niza zapisuje se s 8 byte-ova. Da smo ove podatke zapisali u tekstnu datoteku, svaki član niza bi se zapisao s različitim brojem byte-ova ovisno o broju decimala.

S obzirom da znamo da su u datoteci samo vrijednosti tipa *double*, možemo ih lako pročitati u petlji koristeći metodu *ReadDouble* klase *BinaryReader*. *While* petlja izvršava se sve dok je trenutna pozicija u ulaznom toku manja od ukupne veličine datoteke. To svojstvo *Position* možemo iskoristiti i za klizanje po datoteci kod čitanja. Ako napišemo ovo:

```
int DoubleSize = SizeOf(typeof(double));
UlazniTok.Position = 3 * DoubleSize;
double d = FRead.ReadDouble();
```

u varijabli *d* će biti vrijednost 9.80665, dakle četvrti član niza jer će se setiranjem svojstva *Position* ulazni tok pomaknuti na početak četvrtog člana u datoteci. Za određivanje veličine tipa *double* koristi se metoda *SizeOf* klase *Marshal*. Bez obzira što znamo da je 8 byte-ova trenutno važeća vrijednost za veličinu tipa *double*, ipak je bolji ovakav način određivanja vrijednosti. Ako se jednog dana u nekoj novoj verziji .Net Frameworka ta vrijednost i promijeni, metoda *SizeOf* će uvijek dati ispravnu vrijednost.

Ipak najbolje što se tiče binarnih datoteka ostaje za kraj. U binarne datoteke može se zapisivati i niz struktura i za takvu binarnu datoteku će vrijediti sve kao i za ovu prethodnu s nizom *double* vrijednosti. Ako bismo ovako definirali strukturu *Record*:

```
struct Record
{
      private int i;
      private long l;
      private double d;
```

```
            . . .
      public void Write(BinaryWriter w)
      {
            w.Write(i);
            w.Write(l);
            w.Write(d);
      }
}
```

onda bismo koristeći njenu metodu *Write* mogli u datoteku zapisivati strukturu po strukturu, a ako bismo htjeli pročitati npr. 31. strukturu u datoteci, to bismo napravili na isti način kao i u prethodnom primjeru:

```
UlazniTok.Position = 30 * StructSize;
```

gdje bi *StructSize* bio:

```
StructSize = Marshal.SizeOf(typeof(int)) + Marshal.SizeOf(typeof(long)) +
             Marshal.SizeOf(typeof(double));
```

Naravno da bismo i tada morali pročitati podatke iz strukture u ispravnom redoslijedu, dakle u redoslijedu *ReadInt32(), ReadInt64(), ReadDouble()*.

Na kraju da zaključimo, niti tekstne datoteke isključuju binarne, niti je obrnuto. Svaka od njih ima svoju primjenu. Ako je riječ o čistom tekstu, nemamo što razmišljati, koristit ćemo tekstne datoteke. Ali ako podatke treba zapisati u nekom nizu vrijednosti i još ako se te vrijednosti mogu predstaviti nekim nizom kao varijablom, onda opet nemamo što razmišljati i koristit ćemo binarne datoteke.

13.4. Web tokovi

Na početku poglavlja rekli smo da tok ne mora biti nužno iz i u datoteku, nego da može predstavljati i transfer podataka preko interneta. Sljedeći primjer će učitati cijeli sadržaj stranice *www.eurosport.com*, spremiti ga na disk te prikazati tu istu stranicu izvanmrežno (offline) pozivom Internet Explorera. Za to ćemo koristiti klase *WebRequest* za slanje zahtjeva web serveru, te *WebResponse* za prihvat odgovora s web servera, odnosno samog sadržaja stranice.

```
using System;
using System.IO;
using System.Net;
using System.Diagnostics;

class Test
{
      static void Main()
      {
            string Url = "http://www.eurosport.com";
            string HTMLSource = null;
            StreamReader Reader = null;

            try
            {
                  WebRequest WebZahtjev = WebRequest.Create(Url);
                  WebResponse WebOdgovor = WebZahtjev.GetResponse();
                  Reader = new
                        StreamReader(WebOdgovor.GetResponseStream());
```

```
                HTMLSource = Reader.ReadToEnd();
            }
            catch(System.Net.WebException e)
            {
                Console.WriteLine("Ne mogu se spojiti na
                                  www.eurosport.com");
                return;
            }
            finally
            {
                if(Reader != null)
                    Reader.Close ();
            }

            StreamWriter FWrite = null;
            try
            {
                string CurrFolder = Directory.GetCurrentDirectory();
                FWrite = new StreamWriter(CurrFolder +
                "\\eurosport.html", false, System.Text.Encoding.ASCII);
                FWrite.Write(HTMLSource);

                Process IE = new Process();
                IE.StartInfo.FileName = "iexplore.exe";
                IE.StartInfo.Arguments = CurrFolder + "\\eurosport.html";
                IE.Start();
            }
            catch(Exception e)
            {
                Console.WriteLine("Ne mogu zapisati u datoteku");
            }
            finally
            {
                if(FWrite != null)
                    FWrite.Close();
            }
        }
}
```

Primjer 13.6

Postupak učitavanja stranice radi se u dva koraka. Najprije se statičkom metodom *Create* klase *WebRequest* zatraži sadržaj od web servera. Metoda *Create* vraća objekt klase *WebRequest* (točnije, vratit će objekt njoj izvedene klase ovisno o protokolu koji se koristi kod dohvata web sadržaja, a kako je u našem primjeru riječ o *http* protokolu kreirat će se objekt tipa *HttpWebRequest*) i taj objekt u drugom koraku poziva metodu *GetResponse* za prihvat sadržaja stranice. Sadržaj se čita kao tok, metodom *ReadToEnd* koja će cijeli sadržaj pročitati odjednom.

Nakon što je sadržaj stranice pročitan i spremljen u varijablu *HTMLSource* sprema se u datoteku, u istu mapu u kojoj se nalazi i sam program.
Pokretanje Internet Explorera s imenom datoteke sa spremljenim sadržajem kao argumentom, izvodi se pomoću klase *Process* iz imenika *System.Diagnostic*.
Stranica se iz više različitih razloga nekad ne može dohvatiti (nedostupnost servera, nepostojanje konekcije, pogrešna URL adresa) i oni su obuhvaćeni iznimkom *System.Net.WebException*.

Dohvat stranice u nitima

Učitavanje stranice ponekad može i potrajati, opet iz više različitih razloga (spora veza, preopterećenje servera itd.) i nije dobro da je program blokiran samo tim zadatkom. Vratimo se trenutak na poglavlje o nitima kada smo spomenuli upravo ovakvu situaciju i naglasili da je izvršavanje programa u nitima idealno za takvu situaciju. Program ćemo izvesti tako da se odvija u dvije niti, jedna će biti osnovna u kojoj ćemo prihvaćati web stranicu, a druga može biti bilo što, čak i obični brojač, tek da se vidi da se nešto događa. Takav bi program trebao ovako nekako izgledati:

```
class TwoThreads
{
      private Thread t1;
      private Thread t2;

      public void LaunchThreads()
      {
            ThreadStart Web = new ThreadStart(GetWebPage);
            ThreadStart OtherJob = new ThreadStart(DoSomething);

            t1 = new Thread(Web);
            t2 = new Thread(OtherJob);

            t1.Start();
            t2.Start();
      }

      public void GetWebPage()
      {
            // . . .
      }

      public void DoSomething()
      {
            // . . .
      }

}
```

a u funkciji *Main* bismo definirali objekt ove klase i pozvali njegovu metodu *LaunchThreads*

```
TwoThreads n = new TwoThreads();
n.LaunchThreads();
```

Ako bi u metodi *DoSomething* bio običan, nebitni brojač kao npr. ovaj:

```
for(long i=0;;i++)
      if((i % 10000000) == 0)
            Console.WriteLine(i);
```

onda bismo na kraju metode *GetWebPage*, nakon što se stranica prikaže u prozoru Internet Explorera, trebali pozvati *t2.Abort()* i uništiti drugu nit.

WebRequest i *WebResponse* su apstraktne klase i ovaj isti zadatak možemo obaviti i koristeći od njih izvedene klase *HttpWebRequest* i *HttpWebResponse*.

```
HttpWebRequest WebZahtjev = (HttpWebRequest)WebRequest.Create(Url);
HttpWebResponse WebOdgovor = (HttpWebResponse)WebZahtjev.GetResponse();
```

HttpWebRequest i *HttpWebResponse* redefiniraju neke metode svojih baznih apstraktnih klasa, a uvode i neke nove pa zato pružaju više mogućnosti.
Sve četiri klase nude niz metoda i svojstava vezanih za sadržaj na internetu koji učitavamo, kao što su :

- svojstvo *ContentType* koje će vratiti upravo to, dakle Content Type odnosno vrstu dohvaćenog dokumenta što bismo u našem primjeru mogli iskoristiti kod odluke kojim programom na klijentskoj strani treba otvoriti dokument
- svojstvo *ContentLength* koje vraća veličinu dokumenta u byte-ovima
- svojstvo *Headers* koje daje još neke dodatne informacije, kao npr. o serveru na kojem se nalazi stranica, datumu zadnjeg ažuriranja itd.

Poziv skripti na web serveru
Ovim klasama možemo pozvati i skripte na web serveru (PHP, ASP, ASP.Net), a razlozi za njihovo pokretanje mogu biti npr.

- poziv skripte da parsira neku web stranicu, da izvuče samo podatke koji nas zanimaju te da ih pošalje klijentu
- poziv skripte da napravi nešto na web serveru na kojem klijent nema pravo pristupa, kao npr. upis u bazu podataka.

Prvi razlog je manje bitan jer parsiranje stranice možemo napraviti i na klijentskoj strani gdje nam je na raspolaganju za to vrlo moćna klasa *Regex*, dok je drugi razlog bitniji jer na jednostavan način možemo npr. s klijentske strane upisivati u bazu podataka na web serveru. Tako klijent bez ikakvih prava na web serveru, jednostavnim pozivom skripte koja ima ta prava ostvaruje pravo upisa u bazu podataka.

Poglavlje 14:
Unsafe kôd

Ovo poglavlje obrađuje:

- pokazivače
- unsafe kôd
- prevođenje sa switch-om *unsafe*
- razloge za korištenje unsafe kôda
- naredbu *fixed*

14.1. Pokazivači

Jedna od najvažnijih stvari koja je C i C++ donijela popularnost (ali i odbojnost), bili su pokazivači (*pointers*). Najjednostavnije rečeno pokazivači su varijable koje sadrže adrese nekih drugih varijabli. Pokazivači su imali veliku primjenu kod nizova budući da su se adrese članova niza spremale u kontinuiranom bloku u memoriji, pa se preko pokazivača vrlo jednostavno i brzo moglo pristupati članovima niza. I ne samo to nego se, s obzirom da je pokazivač na prvog člana bio i pokazivač na cijeli niz, prijenosom pokazivača mogao prenositi cijeli niz. Znači, prijenosom nekoliko byte-ova koliko je pokazivač zauzimao, mogli su se prenositi cijeli blokovi memorije.

Uz svu tu slobodu i fleksibilnost, pokazivači su unosili i neizbježnu nesigurnost jer definiranjem pokazivača u njega ste jednostavno postavljali adresu neke memorijske lokacije, a svaka promjena vrijednosti pokazivača direktno je mijenjala sadržaj te memorijske lokacije. To je u principu mogla biti bilo koja memorijska lokacija. Sve se odvijalo uz potpunu slobodu programera, bez nadzora nečega kao što je CLR.

Evo jednog primjera s pokazivačima:

```
#include "iostream.h"

void PromijeniNiz(double *p)
{
	for(int i=0;i<100;i++)
		*(p + i) *= 2;
}

void main()
{
	double *p = new double [100];
	for(int i=0;i<100;i++)
		*p++ = i + 1;
	p -= 100;

	PromijeniNiz(p);

	for(i=0;i<100;i++)
	{
		cout << *(p + i);
		cout << "\n";
	}
}
```

Primjer 14.1

Ovo je jedini primjer u knjizi koji nije napisan u C#. Napisan je u C++ i zadatak mu je kreirati niz od 100 *double* članova, dodijeliti im vrijednosti od 1 do 100, taj niz proslijediti u metodu *PromijeniNiz* koja će svaku vrijednost iz niza povećati 2 puta.
Niz je definiran uz pomoć pokazivača s naredbom:

```
double *p = new double [100];
```

koja je napravila dvije stvari:

- deklariran je pokazivač tipa *double* što znači da će sadržavati adresu ili adrese na kojima će se nalaziti *double* vrijednosti.
- rezerviran je memorijski prostor za 100 *double* članova

Fleksibilnost pokazivača najvidljivija je u tome što se na njemu mogu jednostavno primijeniti operatori ++, --, += i -=. Tako ako pokazivač *p* trenutno pokazuje na 50. član niza, onda će nakon naredbe *p++*, isti taj pokazivač pokazivati na 51. član, odnosno njegova vrijednost će biti adresa 51. člana niza. Također naredbom *p -= 10* pokazivač vraćamo za 10 članova unatrag tako da, ako je pokazivao na 20. član, sad će pokazivati na 10. član.
Ako treba adresirati neki član niza bez promjene vrijednosti pokazivača, onda se taj član može dobiti ovako:

```
*(p+1);      // 2. član niza
*(p+10);     // 11. član niza
*p;          // prvi član niza
```

Ovaj način je sigurniji nego kada s operatorima ++, --, += i -= pomičemo pokazivač po nizu jer ne moramo voditi računa o tome koja je trenutno vrijednost u pokazivaču. Bit će uvijek adresa prvog čana niza jer gornje tri naredbe ne mijanjaju sadržaj pokazivača. Usporedite to s ovim:

```
p++;
p += 6;
p -= 2;
p--;
```

Na koji član niza sada pokazuje pokazivač? Moramo ipak malo zastati i otkriti da pokazivač nakon ovih naredbi pokazuje na četvrti član niza. (1 + 6 – 2 – 1). Očito je da ovo drugo zahtijeva veći oprez, ali mi smo u gornjem primjeru koristili oba načina pa usporedite.

Metodi *PromijeniNiz* prosljeđuje se samo pokazivač (dakle, samo nekoliko byte-ova) i ona može promijeniti vrijednosti cijelog niza jer ako zna adresu prvog člana, onda zna i adrese ostalih članova niza od kojih se svaka nalazi u memorijskoj lokaciji za 8 (jer je to veličina tipa *double*) byte-ova većoj od prethodne.

Pokazivaču se može dodijeliti direktno adresa neke varijable korištenjem operatora &

```
int i = 100;
int *p = &i;
Console.WriteLine(*p);
```

Rezultat ispisa bit će 100 jer će u pokazivaču *p* biti adresa na kojoj se nalazi vrijednost varijable *i*, a s izrazom **p* dobije se vrijednost memorijske lokacije čiju adresu sadrži pokazivač. Zato će ovaj sljedeći primjer ispisati 101:

```
int i = 100;
int *p = &i;
*p = 101;
Console.WriteLine(i);
```

jer će se naredbom *p = 101 postaviti vrijednost 101 u memorijsku lokaciju na koju pokazuje pokazivač, a to je zapravo memorijska lokacija varijable *i*.

Pokazivači mogu pokazivati i na niz struktura:

```
struct Managed
```

```
{
      public int x;
      public int y;
}

. . .

Managed *p = stackalloc Managed [100];
for(int i=0;i<100;i++)
{
      (p + i)->x = i + 1;
      (p + i)->y = 2 *(i + 1);
}
```

Pokazivač *p* sad pokazuje na niz od 100 struktura, a kako vidimo, članovima struktura se pristupa operatorom ->.

14.2. Naredba unsafe

C# je odlučio prekinuti s pokazivačima, barem na razini upravljanog (managed) kôda. U C# možete koristiti pokazivače, ali onda takav kôd morate označiti ključnom riječju *unsafe*. To bi u prijevodu značilo kao nesiguran, što je možda još jedan pokušaj da vas .Net obeshrabri za korištenje pokazivača. Osim ako nisu potrebne superkritične performanse jer će se neupravljani kôd brže izvršavati od upravljanog.

Sama riječ neupravljani puno nam se više sviđa od riječi nesigurni jer bismo onda mogli reći da su svi dosadašnji programi napisani u C i C++ nesigurni. A to, složiti ćete se, nikako ne odgovara istini.

Ipak, tko voli, neka izvoli. Kao što smo rekli C# omogućuje korištenje pokazivača, potrebno je samo takav kôd označiti kao *unsafe* i u prevoditelju odabrati *unsafe* način prevođenja:

- ako koristite poziv *csc* naredbe za prevođenje, dodajte switch */unsafe*
  npr. **csc test.cs /unsafe**
- ako koristite Visual Studio uključite *Project -> Properties -> Configuration Properties -> Build -> Allow Unsafe Code Blocks -> True*

Evo sad onog prethodnog primjera, ovaj put u C#.

```
using System;

class Test
{
      unsafe static void Main()
      {
            double *p = stackalloc double [100];
            for(int i=0;i<100;i++)
                  *p++ = i+1;
            p -= 100;

            PromijeniNiz(p);

            for(int i=0;i<100;i++)
                  Console.WriteLine(*(p+i));
      }
```

```
	unsafe public static void PromijeniNiz(double *p)
	{
		for(int i=0;i<100;i++)
			*p++ *= 2;
	}
}
```

Primjer 14.2

Ispis:
Progam će ispisati brojeve od 1 – 100 uvećane 2 puta.

Ključna riječ *unsafe* dodaje se ispred klase, članice klase ili izraza koji se treba izvršiti bez nadzora CLR-a, a vidimo još i ključnu riječ *stackalloc* koja je zamijenila *new* iz C++ primjera, a koja također rezervira memorijski prostor za 100 članova tipa *double*.

Ako koristite *unsafe*, zaboravite na mnoge dobre stvari na koje smo se uz pomoć CLR-a mogli osloniti. Pogledajte ovaj kôd:

```
double *p = stackalloc double [100];
for(int i=0;i<100;i++)
	Console.WriteLine(*(p+i));
```

Što će se ispisati?
To nitko ne zna i potpuno je nepredvidljivo jer u pokazivačima koji nisu definirani može biti bilo što, tu ne vrijede podrazumijevane vrijednosti kao kod vrijednosnih i referentnih tipova. I ne samo to, što ako bismo napravili ovo?

```
double *p = stackalloc double [100];
for(int i=100;i<200;i++)	// Za ovih 100 nije rezervirana memorija
	*(p + i) = i+1;
```

Nemojte ovo pokušavati jer ako su u pokazivačima nepredvidljive vrijednosti, onda njihov sadržaj, odnosno adrese koje upućuju na memorijske lokacije, mogu upućivati na bilo što i dodijeljivanjem vrijednosti tim memorijskim lokacijama prebrisat će se vrijednost koja se tu nalazi. Ponavljamo, ako ne znamo što je na određenoj memorijskoj lokaciji, kako možemo biti sigurni da na njoj nije nešto jako važno.

Ako vam je ovaj silan oprez bio dovoljan da ne koristite *unsafe*, nećete pogriješiti, ali ako ste dovoljno oprezni, onda opet nema razloga da ne koristite *unsafe*. Naravno, ako imate dobrih razloga za to.

14.3. Naredba fixed

Pokazivaču možemo dodijeliti i adresu neke varijable koja je pod kontrolom automatskog čišćenja memorije (Garbage Collection) kao npr. člana neke klase ili strukture. Tada je potrebno ključnom riječju *fixed* označiti dodjelu te vrijednosti te blok u kojem se pokazivač koristi, kao znak GC-u da ne vrši nikakve radnje s memorijskom lokacijom na kojoj se nalazi ta varijabla.

```
using System;

unsafe class Test
{
	private int x = 50;
```

```
        static void Main()
        {
                Test t = new Test();
                fixed(int *p = &t.x)
                {
                        Console.WriteLine(*p);
                        GC.Collect();
                        Console.WriteLine(*p);
                }
        }
}
```

Primjer 14.3

Ispis:
50
50

Naredbom *fixed* označili smo dodjelu adrese varijable *x* pokazivaču *p*. Tako smo se osigurali da mehanizam za čišćenje memorije neće pomaknuti memorijsku lokaciju na kojoj se nalazi varijabla *x* i to niti nakon automatskog čišćenja, niti nakon forsiranog poziva mehanizma za čišćenje metodom *Collect*.
U *fixed* izrazu može se navesti i više dodjela vrijednosti pokazivaču:

```
fixed(int *p = &t.x, point = &t.y)
```

point je kao i *p* pokazivač na *int* varijablu, a ako su pokazivači različitih tipova, onda se moraju navesti dvije *fixed* naredbe:

```
fixed(int *p = &t.x)
fixed(float *point = &t.y)
```

Poglavlje 15:
C#++

Ovo poglavlje obrađuje:

- općenito o XML-u
- čitanje i kreiranje XML dokumenata
- čitanje rss dokumenata
- izradu XML dokumentacije programskog kôda
- ADO.NET za pristup bazama podataka
- parametriziranje SQL naredbi
- adaptere podataka
- web servise
- klijentske programe koji koriste web servise
- Google web servise

Posljednje poglavlje nazvali smo ovako kao znak da ćemo tu obraditi neke nove i vrlo upotrebljive klase. Iako smo vidjeli cijele knjige napisane o primjeni XML-a u .Netu (o primjeni, ne o XML-u!), te o tehnologiji za pristup bazama podataka ADO.Net, kao i o web servisima, pokušat ćemo s nekoliko primjera obuhvatiti ono najosnovnije.

15.1. XML

Na početku najkraći mogući uvod u XML.

XML (Extensible Markup Language) je jezik nastao u drugoj polovici devedesetih i namijenjen je za razmjenu podataka izmedu različitih IT sustava, uglavnom preko interneta. I .Net je također prihvatio XML kao svoj standard za razmjenu podataka.
XML opisuje **što** podaci prikazuju za razliku od HTML-a koji opisuje **kako** se podaci prikazuju. Tagovi u XML-u definiraju se ovisno o tome što se želi prikazati u dokumentu i strukturirani su u obliku nivoa odnosno stabla.

Nećemo pogriješiti ako kažemo da je XML dokument baza podataka u tekstnom formatu. Zato XML dokument brzo naraste s većom količinom slogova, ali budući da je riječ o tekstnoj datoteci, čitači XML dokumenata lako izađu na kraj s takvim velikim dokumentima. Kada bi na svijetu postojala samo jedna baza podataka, onda bi vjerojatno bila manja potreba za XML-om, ali kako nema takve baze podataka, XML je preuzeo tu ulogu koju su prihvatili svi veliki igrači u IT industriji.

XML dokument mora biti dobro oblikovan (well formed) što znači da mora biti napisan u potpunosti po pravilima jezika. Ako i jedan tag nije ispravno napisan, XML dokument neće se prikazati i dobit ćemo poruku o loše oblikovanom dokumentu. Nije slučaj kao kod HTML-a gdje i ne mora biti sve ispravno napisano, ono što je ispravno to je u redu, a ono što nije ili će biti ignorirano od strane pretraživača ili će čak pretraživač sam pokušati ispraviti greške. Također, za razliku od HTML-a, XML je *case sensitive* što znači da se razlikuju velika i mala slova.

Sad ćemo na primjeru vidjeti kako izgleda jedan XML dokument. Bit će to kraći oblik tečajne liste.

```
<?xml version="1.0" encoding="windows-1250" ?>
<TecajnaLista>
  <Valuta>
    <Drzava>Australija</Drzava>
    <Oznaka>AUD</Oznaka>
    <Sfr>036</Sfr>
    <Jedinica>1</Jedinica>
    <Iznos>4.385692</Iznos>
  </Valuta>
  <Valuta>
    <Drzava>Japan</Drzava>
    <Oznaka>YPY</Oznaka>
    <Sfr>392</Sfr>
    <Jedinica>100</Jedinica>
    <Iznos>4.704524</Iznos>
  </Valuta>
  <Valuta>
    <Drzava>Velika Britanija</Drzava>
    <Oznaka>GBP</Oznaka>
    <Sfr>826</Sfr>
```

```
    <Jedinica>1</Jedinica>
    <Iznos>10.960638</Iznos>
  </Valuta>
  <Valuta>
    <Drzava>USA</Drzava>
    <Oznaka>USD</Oznaka>
    <Sfr>840</Sfr>
    <Jedinica>1</Jedinica>
    <Iznos>5.570351</Iznos>
  </Valuta>
  <Valuta>
    <Drzava>EU</Drzava>
    <Oznaka>EUR</Oznaka>
    <Sfr>978</Sfr>
    <Jedinica>1</Jedinica>
    <Iznos>7.357876</Iznos>
  </Valuta>
</TecajnaLista>
```

XML je jezik za strojeve, zato ako i ne izgleda pregledno, kasnije ćete vidjeti kako klase .Net Frameworka brzo riješe ovakve dokumente. Vidimo da su tagovi potpuno drugačiji nego u HTML-u, tagovi koji svojim imenom opisuju sebe.

Prva linija u dokumentu opisuje verziju XML-a i način kodiranja, proizvoljna je, ali se uglavnom navodi. Sve drugo u dokumentu su tzv. ugnježđeni elementi (*nested elements*) koji se sastoje od dva taga, startnog (*start tag*) i završnog taga (*ended tag*). Sve između startnog i završnog taga je sadržaj (*content*) elementa, a to uključuje tekst između startnog i završnog taga kao i druge elemente (*child elements*).
Svaki XML dokument mora imati korijenski (*root*) element i to je u ovom primjeru element *TecajnaLista*.

Elementi mogu imati i atribute (*attributes*) koje nismo koristili u gornjem primjeru, ali bi se mogli ovako definirali:

```
<TecajnaLista Datum="27032006">
```

Definiran je atribut *Datum* i dodijeljena mu je vrijednost 27032006.

Nekad ćete možda doći u nedoumicu je li bolje nešto staviti kao atribut ili kao element jer mogli smo u dokumentu s tečajnom listom staviti *Oznaku* kao atribut elementa *Valuta*, a ne kao zaseban element.

```
<Valuta Oznaka="USD">
```

Svi bi podaci bili opet dostupni, ali trebalo bi se držati generalnog pravila da ako je nešto dio neke definicije onda to treba biti *child* element, a ne atribut. A oznaka valute sigurno spada u definiciju same valute.

Čitanje XML dokumenata

Čitači XML dokumenata tzv. parseri imaju upravo tu namjenu, da prođu sekvencijalno cijeli XML dokument te pročitaju sadržaj elemenata. U sljedećem primjeru ćemo, koristeći klase iz imenika *System.XML*, pročitati malo prije navedeni dokument s tečajnom listom. Prije

izvođenja programa osigurajte u putanji datoteku *TecajnaLista.xml* u kojoj će sadržaj biti gore navedeni XML dokument.

```
using System;
using System.Xml;

class Test
{
	static void Main()
	{
		XmlDocument Doc = new XmlDocument();

		try
		{
			Doc.Load("TecajnaLista.xml");
		}
		catch(XmlException e)
		{
			Console.WriteLine("Dokument nije dobro oblikovan");
		}
		catch(System.IO.IOException e)
		{
			Console.WriteLine("Ne postoji datoteka
					TecajnaLista.xml");
		}
		XmlNodeList Nodes = Doc.GetElementsByTagName("Valuta");

		foreach(XmlNode Node in Nodes)
			Console.WriteLine("{0} {1} = {2} kn",
			Node["Jedinica"].InnerText, Node["Oznaka"].InnerText,
			Node["Iznos"].InnerText);
	}
}
```

Primjer 15.1

Ispis:

```
1 AUD = 4.385692 kn
100 YPY = 4.704524 kn
1 GBP = 10.960638 kn
1 USD = 5.570351 kn
1 EUR = 7.357876 kn
```

Praktički sa samo jednom naredbom, s pozivom metode *GetElementsByTagName* rješavamo cijeli XML dokument. Ova metoda vraća kolekciju čvorova, tipa klase *XmlNodeList* koja nasljeđuje sučelje *IEnumerable* pa možemo koristiti *foreach* petlju za prolaz kroz sve elemente cijelog dokumenta. Čvor može biti sami element ili tekst elementa, a tipovi čvorova su sadržani u enum tipu *XmlNodeType*.

Hvataju se i dvije iznimke, *IOException* u slučaju da datoteka ne postoji i *XmlException* ako dokument nije dobro oblikovan. Dovoljno je da samo u jednom tagu nekog elementa zamijenite veliko i malo slovo i generirat će se iznimka *XmlException*.

Kreiranje XML dokumenata

Kreiranje XML dokumenata jednako je jednostavno kao i njihovo čitanje. Kreirat ćemo onaj isti dokument s tečajnom listom, ali pod imenom *TL.xml*.

```
using System;
using System.Collections;
using System.Xml;

class XMLTLista
{
	string[,] Arr = new  string[,]
	{
		{"Australija", "Japan", "Velika Britanija", "USA", "EU"},
		{"AUD", "YPY", "GBP", "USD", "EUR"},
		{"036", "392", "826", "840", "978"},
		{"1", "100", "1", "1", "1"},
		{"4.385692", "4.704524", "10.960638", "5.570351", "7.357876"}
	};

	public void CreateXMLFile()
	{
		XmlTextWriter Writer = null;

		try
		{
			Writer = new XmlTextWriter("TL.xml",
				System.Text.Encoding.Default);
			Writer.Formatting = Formatting.Indented;

			Writer.WriteStartDocument();
			Writer.WriteStartElement("TecajnaLista");

			for(int i=0;i<Arr.GetLength(1);i++)
			{
				Writer.WriteStartElement("Valuta");

				Writer.WriteElementString("Drzava", Arr[0, i]);
				Writer.WriteElementString("Oznaka", Arr[1, i]);
				Writer.WriteElementString("Sfr", Arr[2, i]);
				Writer.WriteElementString("Jedinica", Arr[3, i]);
				Writer.WriteElementString("Iznos", Arr[4, i]);

				Writer.WriteEndElement();
			}
			Writer.WriteEndElement();
		}
		catch(Exception e)
		{
			Console.WriteLine(e.Message);
		}
		finally
		{
			if(Writer != null)
				Writer.Close();
		}
	}
}

class Test
{
	static void Main()
	{
		XMLTLista XmlDynamic = new XMLTLista();
		XmlDynamic.CreateXMLFile();
	}
```

```
}
```

Primjer 15.2

Podaci za XML dokument definirani su u višedimenzionalnom nizu, a dokument se jednostavno "slaže" s metodama objekta *Writer* klase *XmlTextWriter*. Metodom *WriteStartElement* kreira se element *Valuta*, a onda se metodom *WriteElementString* dodaju njegovi *child* elementi. Argumenti metode *WriteElementString* su naziv elementa i njegov tekst koji se dobije iz višedimenzionalnog niza. Elemente treba na kraju zatvoriti i za to je zadužena metoda *WriteEndElement*.

Čitanje rss dokumenata

XML jezik ima i svoje tzv. dijalekte. Jedan od njih je i *rss* (Really Simple Syndication) koji se koristi uglavnom kod davatelja raznih vijesti na internetu. Davatelji vijesti kreiraju *rss* dokumente na način da su elementi u dokumentu naslov vijesti (element *title*) URL adresa vijesti (element *link*), datum objave vijesti (element *pubDate*) itd. Budući da svi *rss* dokumenti moraju udovoljavati XML 1.0 standardu, mi ćemo sada koristeći klase iz imenika *System.Xml* napraviti program koji će:

- učitati sa stranice BBC Tehnology *rss* dokument
- parsirati ga i izvući sve naslove, linkove i datume objave vijesti
- spojiti se na prve i zadnje dvije vijesti iz dokumenta

```
using System;
using System.Xml;
using System.Diagnostics;

class Test
{
      static void Main()
      {
            string Url =
"http://newsrss.bbc.co.uk/rss/newsonline_uk_edition/technology/rss.xml";
            XmlDocument Doc = new XmlDocument();
            Doc.Load(Url);
            XmlNodeList Nodes = Doc.GetElementsByTagName("item");

            int i = 0;
            foreach(XmlNode Node in Nodes)
            {
                  Console.WriteLine("{0}\n{1}\n{2}\n\n",
                        Node["title"].InnerText, Node["link"].InnerText,
                        Node["pubDate"].InnerText);
                  if(i == 0  ||  i == 1  ||  i == Nodes.Count - 2  ||
                                             i == Nodes.Count - 1)
                  {
                        Process IE = new Process();
                        IE.StartInfo.FileName = "iexplore.exe";
                        IE.StartInfo.Arguments = Node["link"].InnerText;
                        IE.Start();
                  }
                  i++;
            }
      }
}
```

Primjer 15.3

Program će prije nego što pokrene linkove, prikazati sve naslove, linkove i datume objave vijesti iz *rss* dokumenta. Prikazuju se prve dvije i zadnje dvije vijesti. Prve dvije jer su to dvije najnovije, a zadnje dvije da ih vidimo prije nego što nestanu jer se vijesti dodaju FIFO (First In First Out) metodom.
Dokument se učitava ponovo metodom *Load* kojoj kao argument sada umjesto datoteke prosljeđujemo link (link također upućuje na datoteku, ali na web serveru) i sve radi na isti način. Opet vidimo primjer preopterećenosti koji umnogome olakšava rad s objektima i klasama te njihovim metodama. Prvu i drugu vijest pronalazimo pomoću brojača *i*, dok zadnje dvije pronalazimo pomoću svojstva *Count* koje vraća ukupan broj *item* elemenata, a to je u ovom slučaju ukupan broj vijesti.

Otvaranje stranice čiji smo link pročitali iz *rss* dokumenta vršimo kao i u primjeru s web tokovima, samo što ovaj put radimo *online* i kao argument prosljeđujemo link web stranice. Još jednom preopterećenost, ali ovaj put na razini pokretanja Internet Explorera. Očito, nije to bez razloga jedna od najvažnijih osobina svakog objektno orijentiranog sustava.

XML dokumentacija

Pisanje komentara i dokumentacije za programski kôd uvijek su bile stvari koje su programeri nerado radili, iako su uvijek bili svjesni njihove važnosti. Vjerojatno je razlog što pisanje dokumentacije zahtijeva angažman sasvim drugačijeg tipa nego za pisanje software-a. .Net kao nova platforma ponudio je i tu nešto sasvim novo što omogućuje neku vrstu programskog upravljanja dokumentacijom i komentarima.

U poglavlju o komentarima naveli smo da je C# preuzeo od C++ i Jave dvije vrste komentara, linijski (od // do kraja reda) i komentar bloka (od /* do */), ali je uveo i treću vrstu, tzv. komentar u XML formatu. Koristeći strukturu XML jezika, programer piše komentare i opise klasa, metoda, svojstava i ostalih dijelova kôda na način da koristi XML elemente te komentar unosi kao vrijednost pojedinog elementa.
Nakon toga prevoditelj kreira od komentara XML dokument koji onda predstavlja dokumentaciju programa u XML formatu.

Ovakvi XML komentari unose se nakon navođenja tri znaka za kosu crtu ///, tako da se prevoditelj kada naiđe na dva znaka kose crte ne zaustavlja, nego nastavlja provjeravati je li treći znak također kosa crta što mu je onda znak da je to XML komentar.
Postoji nekoliko predefiniranih tagova za XML komentiranje, a na sljedećem primjeru vidjet ćemo kako to sve izgleda u praktičnoj izvedbi.

```
using System;

/// <summary>
/// Klasa za rad s temperaturama
/// </summary>
/// <remarks>
/// Klasa Celzius nema definiran konstruktor.
/// Temperatura se zadaje preko svojstva
/// </remarks>

class Celzius
{
      /// <summary>
      /// Varijabla sadržava trenutnu temperaturu
```

```
        /// koja se zadaje preko svojstva
        /// </summary>
        private double trenutnaTemperatura;

        /// <summary>
        /// Get/Set svojstvo za varijablu trenutnaTemperatura
        /// </summary>
        /// <value>
        /// Vrijednost koja se zadaje mora biti tipa double
        /// </value>
        public double TrenutnaTemperatura
        {
              get
              {
                    return trenutnaTemperatura;
              }

              set
              {
                    trenutnaTemperatura = value;
              }
        }

        /// <summary>
        /// Metoda vraća vrijednost temperature u Kelvinima
        /// za zadanu vrijednost u Celziusima
        /// </summary>
        /// <param name="TempCelzius"> Temperatura koja se
        /// pretvara u Kelvine
        /// </param>
        /// <returns>
        /// Vraća se double vrijednost temperature u Kelvinima
        /// </returns>

        public double Kelvins(double TempCelzius)
        {
              return TempCelzius + 273.15;
        }
}

class Test
{
        public static int Main()
        {
              Celzius t = new Celzius();
              t.TrenutnaTemperatura = 25;
              Console.WriteLine(t.Kelvins(26.85) + " K");
              return 0;
        }
}
```

Primjer 15.4

Ispis:
300 K

Kao rezultat dobili bismo ovakvu XML datoteku (ovo je kondenzirani prikaz):

```
<?xml version="1.0" ?>
<doc>
  <assembly>
      <name>Test</name>
```

```
  </assembly>
  <members>
    <member name="T:Celzius">
    <member name="F:Celzius.trenutnaTemperatura">
    <member name="M:Celzius.Kelvins(System.Double)">
    <member name="P:Celzius.TrenutnaTemperatura">
    </members>
</doc>
```

Da bi prevoditelj kreirao ovakav XML dokument, potrebno je program prevesti na sljedeći način:

```
csc Test.cs /doc:XMLKomentar.xml
```

gdje je *XMLKomentar.xml* datoteka u XML formatu koja će biti kreirana. Ako koristite Visual Studio, odaberite: *Project -> Properties-> Configuration Properties ->XML Documentation File*.

Sam program vrlo je jednostavan i tu je tek da pokaže XML komentiranje. Imamo klasu *Celzius* koja ima po jedno svojstvo, metodu i varijablu.

Ono što nas ovdje zanima su tagovi. Naveli smo ih 5:

1. **<summary>** koji treba dati općeniti opis tipa ili člana klase
2. **<remarks>** za nešto konkretniji opis tipa ili člana klase
3. **<param>** za opis parametara koji se prosljeđuju metodama
4. **<returns>** za komentar povratne vrijednosti iz metode
5. **<value>** za komentar vrijednosti koja se dodjeljuje sa *set* svojstvom

Ovo nisu svi predefinirani elementi, postoje još neki kao npr. <exception> za opis iznimki ili <example> za navođenje primjera nekog kôda, a programer može definirati i svoje vlastite. Neki elementi imaju i atribute kao <param> kojem se kao atribut, navodi naziv varijable koja je argument u deklaraciji metode. Ako već stavimo atribut onda moramo navesti točno njegovo ime, inače će nas prevoditelj upozoriti na neusklađenost imena. Isto tako, ako metoda ima više od jednog argumenta, za svakog moramo navesti element <param>.

S ovih nekoliko primjera pokušali smo vas uvesti u svijet XML-a u .Net-u. Za nešto više nemamo dovoljno stranica, a kao što smo već nekoliko puta rekli, ovo je ipak knjiga o C#. Svejedno, nadamo se da je i ovo bilo dovoljno da iskoristite sve prednosti koje XML pruža kao otvoreni, jednostavan za prijenos preko mreže, općeprihvaćeni standard za razmjenu podataka. Ako tome pridodamo jednostavno pisanje i čitanje XML dokumenata uz korištenje klasa .Net-a i C#, onda je pozitivan dojam o XML-u potpun.

15.2. ADO.Net

Opet jedna stvar koju nikako nismo mogli ne spomenuti. Manje više svaki program ima potrebu za spremanjem podataka. Jedan način, onaj s tokovima, već smo obradili, a sad nam preostaje onaj puno moćniji, rad s pravim bazama podataka. Riječ je o **ADO.Net** (Active Data Objects), cijelom jednom sustavu za rad s bazama podataka koji također donosi niz novina u odnosu na dosadašnje pristupe. Nove stvari ne moraju uvijek biti i bolje, ali ovaj pristup bazama podataka je bez ikakve sumnje efikasniji što se tiče sustava i jednostavniji što se tiče izrade programa.

Pristup podacima odvija se preko posebnih programskih modula tzv. upravitelja podataka (data providers). *ADO.Net* koristi dva različita upravitelja, odnosno dvije skupine klasa iz imenika *System.Data*:

- klase za pristup MS-SQL bazi podataka
- klase za pristup ostalim bazama podataka

Prvi pristup odvija se u potpuno upravljanom okruženju, a drugi koristi neupravljane, dobro poznate OLE DB upravitelje.

Mi ćemo u našim primjerima koristiti OLE DB upravitelj za pristup Access bazi podataka. U tu svrhu kreirali smo bazu podataka *Brodovi* s tablicom *PoznatiBrodovi* koju smo napunili s 15 poznatih brodova iz povijesti. Svaki se slog sastoji od 5 polja:

1. Naziv broda (tip *string)*
2. Država kojoj je brod pripadao ili pripada (tip *string*)
3. Dužina broda (tip *number*)
4. Datum prvog porinuća (tip *datum*)
5. Je li brod trenutno na dnu mora (tip *bool*)

Htjeli smo imati različite tipove podataka, a podaci uglavnom odgovaraju stvarnim podacima. Ako i postoje neke greške, nemojte zamjeriti.

Prvi primjer će iz baze podataka o brodovima ispisati sve brodove koji se nalaze na dnu mora (naravno, samo od onih iz naše baze). Ideja je očigledna, proći ćemo sve brodove iz baze i ispisati samo one koji imaju vrijednost polja *NaDnuMora == true*.

```
using System;
using System.Data.OleDb;

class Test
{
	static void Main()
	{
		string Database = "Brodovi.mdb";
		OleDbConnection Conn = new OleDbConnection
		("Provider=Microsoft.Jet.OLEDB.4.0;" + "Data Source=" +
			Database + ";Persist Security Info=False");

		try
		{
			Conn.Open();
			OleDbCommand Command = new OleDbCommand("SELECT * FROM
```

```
                        PoznatiBrodovi", Conn);
                OleDbDataReader Reader = Command.ExecuteReader();

                while(Reader.Read())
                    if((bool)Reader["NaDnuMora"])
                        Console.WriteLine(Reader["Naziv"] + " - " +
                                          Reader["Drzava"]);
                Reader.Close();
            }
            catch(OleDbException e)
            {
                Console.WriteLine(e.Message);
            }
            finally
            {
                Conn.Close();
            }
        }
}
```

Primjer 15.5

Ispis:
Flying Dutchman - Nizozemska
Bounty - Engleska
Titanic - Engleska
Bismarck – Njemačka

Sve klase koje koristimo nalaze se u imeniku *System.Data.OleDB*, pa je taj imenik potrebno uključiti na početku programa. Prvo što treba napraviti u programu jest uspostaviti fizičku konekciju s bazom podataka za što je zadužen konekcijski objekt (*Connection object*), a to je u našem slučaju objekt *Conn* klase *OleDbConnection*. Konstruktor klase prima kao argument konekcijski string u kojem se nalazi ime upravitelja podataka i naziv same datoteke baze podataka. Ako je konekcijski string sintaktički neispravan, npr. stavite li *Provide* umjesto *Provider* generirat će se iznimka *System.ArgumentException*.

Sljedeći i praktički jedini koraci prije čitanja baze su otvaranje same konekcije metodom *Open* i kreiranje komandnog objekta klase *OleDbCommand* koji kao argumente dobiva SQL naredbu koju treba izvršiti i prethodno kreirani konekcijski objekt.
Sad je sve spremno za čitanje baze podataka i to se izvodi pozivom metode *ExecuteReader* objekta *Command* čiju povratnu vrijednost koristimo za čitanje slogova iz baze podataka. Čitanje se radi metodom *Read* koja će vratiti *false* ako u bazi više nema podataka za pročitati.

Objekt *Reader* ima preopterećeni indeks pa ga možemo koristiti tako da mu kao argument navedemo redni broj polja u slogu ili string, a to je naziv polja u bazi podataka. Tako bi ovo dalo isti rezultat:

```
if((bool)Reader["NaDnuMora"])
```
i
```
if((bool)Reader[5])
```

Ovdje trebamo pohvaliti autore .Net Frameworka koji prije usporedbe navedenog indeksa s nazivom polja pretvore jedan i drugi u mala (ili možda velika) slova (usporedba *case insensitive*), tako da možemo navesti i ovako:

```
if((bool)Reader["nadnumora"])
```

i sve će biti uredno izvedeno. Promašimo li neko slovo ili koristimo brojeve kao indekse pa navedemo

```
if((bool)Reader[6])
```

i premašimo indeks, bit će generirana iznimka *IndexOutOfBoundException*, pa i na to treba računati u izradi prave komercijalne aplikacije.

Rekli smo da objekt *Command* klase *OleDbCommand* prima kao argument SQL naredbu koja će se izvršiti. Zato smo mogli izbjeći provjeravanje je li *Reader["NaDnuMora"] true* ili *false* tako da smo provjeru mogli ugraditi u samu SQL naredbu:

```
select * from PoznatiBrodovi where NaDnuMora=true
```

Na kraju konekciju nikako ne treba ostaviti otvorenom, zato je radi svake sigurnosti zatvaramo u bloku *finally*.

Objekt tipa *Reader* može pročitati i ostale tipove podataka kao npr. datum. Potrebno je samo napraviti odgovarajući *cast*, odnosno konverziju. Ako navedemo:

```
DateTime dt;
while(Reader.Read())
      dt = (DateTime)Reader["datumporinuca"];
```

u objektu *dt* bit će datum porinuća za pojedini brod.

Standardno vrijeme dozvoljeno za izvršenje metode *ExecuteCommand* je 30 sekundi i sadržano je u get/set svojstvu *CommandTimeout*, a u slučaju vremenskog prekoračenja generirat će se *OleDbException* iznimka.

Sigurno ste se već mogli uvjeriti u fleksibilnost i upotrebljivost tehnologije ADO.Net. Ali tu nije kraj. Na početku smo spomenuli dvije skupine klasa: *OleDB* koje smo koristili u našem primjeru i klase za rad s MS SQL serverom. Što ako trebate cijeli program prebaciti na MS SQL Server bazu podataka?
Gotovo da trebate napraviti samo dvije (2) stvari:

1. Zamijeniti konekcijski string
2. Zamijeniti "*OleDb"* u *"Sql"* u imenima klasa i članica klasa i stvar će vrlo vjerojatno sve raditi na isti način samo s drugom bazom podataka. Naime, gotovo sve klase imaju OleDb* i Sql* verzije i obični Find Replace jednostavno riješava prebacivanje s jedne baze na drugu.

Promjena (Update) slogova u bazi podataka

Što bi trebalo napraviti da se Titanik izvuče s dna mora?
Trebali biste troje:

1. naći nekog bogataša koji ne zna što će s novcem
2. izvući Titanik s dna mora
3. promijeniti u našoj bazi polje *NaDnuMora* za Titanik iz *true* u *false*

Svatko odmah vidi da je treći korak daleko najlakši, a mi ćemo pokazati u sljedećem primjeru da je koristeći ADO.Net i lakši nego što ste možda očekivali.

```
using System;
using System.Data.OleDb;

class Test
{
      static void Main()
      {
            string Database = "Brodovi.mdb";
            OleDbConnection Conn = new OleDbConnection
            ("Provider=Microsoft.Jet.OLEDB.4.0;" + "Data Source=" +
            Database + ";Persist Security Info=False");

            try
            {
                  Conn.Open();
                  string sSQL = "UPDATE PoznatiBrodovi SET " +
                          "NaDnuMora = '0'" + " WHERE Naziv = 'Titanic'";

                  OleDbCommand Command = new OleDbCommand(sSQL, Conn);
                  Command.ExecuteNonQuery();
            }
            catch(OleDbException e)
            {
                Console.WriteLine(e.Message);
            }
            finally
            {
                Conn.Close();
            }
      }
}
```

Primjer 15.6

U odnosu na prethodni primjer samo su dvije razlike:

- sintaksa SQL naredbe
- poziv metode objekta *Command*

U SQL naredbi navodi se uvjet za koji brod treba promijeniti varijablu *NaDnuMora*, dok objekt *Command* ovaj put poziva metodu *ExecuteNonQuery*. Ipak, ovakvo direktno navođenje u SQL naredbi imena broda kojem treba promijeniti vrijednost nekog polja rijetko se koristi. Uglavnom će u sklopu nekog programa naziv tog broda ili njegov ID biti sadržaj neke varijable koji se dobije iz nekog od GUI objekata. Tada bi to ovako trebalo izgledati:

```
string Ship = "Titanic";
string sSQL = "UPDATE PoznatiBrodovi SET " + "NaDnuMora = '0'" +
              " WHERE Naziv = " + "\'" + Ship + "\'";
```

a ako bi u uvjetu bio ID broda:

```
int IDShip = 7;
string sSQL = "UPDATE PoznatiBrodovi SET " + "NaDnuMora = '0'" +
              " WHERE ID = " + IDShip;
```

Obratite pozornost na razliku ako je u uvjetu WHERE string i ako je broj. U prvom slučaju potrebno je varijablu staviti u navodnike što se napravilo slaganjem stringa, a u slučaju brojeva navodnici nisu potrebni. Ispred navodnika stavili smo znak '\' koji ništa neće promijeniti, ali povećava preglednost. Bez njega bismo u nekim fontovima vidjeli 5 istih navodnika što prevoditelju ne bi smetalo, ali nekome tko čita program možda i bi.

Brisanje (Delete) slogova u bazi podataka

Mnogi poznati brodovi slavu su stekli samo zato što se nalaze na dnu mora. Ako bi ih izvukli s dna mora, odnosno ako bi varijabla *NaDnuMora* postala *false*, možda bi ih netko htio izbrisati iz ovakve jedne baze podataka. To ćemo opet napraviti kao i u prethodnom primjeru samo što ćemo sad u SQL naredbi ime broda navesti kao varijablu.

```
using System;
using System.Data.OleDb;

class Test
{
	static void Main()
	{
		string Database = "Brodovi.mdb";
		OleDbConnection Conn = new OleDbConnection
		("Provider=Microsoft.Jet.OLEDB.4.0;" + "Data Source=" +
		Database + ";Persist Security Info=False");
		string LegendaryShip = "Bismarck";

		try
		{
			Conn.Open();
			string sSQL = "DELETE FROM PoznatiBrodovi
				WHERE Naziv = " + "\'" + LegendaryShip + "\'";

			OleDbCommand Command = new OleDbCommand(sSQL, Conn);
			Command.ExecuteNonQuery();
		}
		catch(OleDbException e)
		{
			Console.WriteLine(e.Message);
		}
		finally
		{
			Conn.Close ();
		}
	}
}
```

Primjer 15.7

Nema potrebe pojašnjavati kôd jer je jedina razlika u SQL naredbi. Kad smo govorili o kompatibilnosti ovakvih klasa .Net Frameworka za sve baze podataka, to se ne odnosi na SQL sintaksu. Zato obratite pozornost na eventualnu razliku u SQL sintaksi između pojedinih baza. Iako je uglavnom ista, neke manje razlike ipak postoje.

Insertiranje (Insert) slogova u bazi podataka

Ako treba dodati brod u bazu podataka, koristit ćemo naredbu INSERT:

```
using System;
using System.Data.OleDb;

class Test
{
	static void Main()
	{
		string Database = "Brodovi.mdb";
		OleDbConnection Conn = new OleDbConnection
		("Provider=Microsoft.Jet.OLEDB.4.0;" + "Data Source=" +
		Database + ";Persist Security Info=False");

		int ID = 100;
		string Name = "Rimska Galija";
		string Drzava = "Rimsko Carstvo";
		double Duzina = 25;
		DateTime DatumPorinuca = new DateTime(200, 1, 1);
		bool NaDnuMora = true;

		string sSQL = "INSERT INTO PoznatiBrodovi VALUES(" +
			"\'" + ID + "\'" + ", " +

			"\'" + Name + "\'" + ", " +

			"\'" + Drzava + "\'" + ", " +

			"\'" + Duzina + "\'" + ", " +

			"\'" + DatumPorinuca.Day + "." +
			DatumPorinuca.Month + "." +
			DatumPorinuca.Year + "\'" + ", " +

			"\'" + (NaDnuMora == false ?  "0" : "1") + "\'" + ")";

		try
		{
			Conn.Open();
			OleDbCommand Command = new OleDbCommand(sSQL, Conn);
			Command.ExecuteNonQuery();
		}
		catch(OleDbException e)
		{
			Console.WriteLine(e.Message);
		}
		finally
		{
			Conn.Close ();
		}
	}
}
```

Primjer 15.8

Kod insertiranja novih slogova treba navesti vrijednosti svih polja pa je kreiranje stringa *sSQL* radi preglednosti razvučeno u nekoliko redova. Vidimo da kao i kod primjera s UPDATE naredbom, varijable tipa string navodimo u navodnicima, dok number tipove navodimo bez navodnika. Također, sintaksa kod *bool* naredbi je
1 je *true*, a 0 je *false*,
dok se datum, kao i string, navodi unutar navodnika u propisanom formatu *dd.mm.gggg* pa i tu obratite pozornost na eventualnu razliku između pojedinih baza podataka.

Parametriziranje SQL naredbi

Često je potrebno neku SQL naredbu ponoviti više puta. To traži svaki put redefiniranje SQL stringa te ponovno definiranje objekta klase *OleDbCommand*:

```
sSQL = " . . .";
Command = new OleDbCommand(sSQL, Conn);
Command.ExecuteNonQuery();
```

Za drugu SQL naredbu sve ovo je potrebno ponoviti s izmijenjenim SQL stringom. Slaganje SQL stringa zna biti kao što smo vidjeli nezgrapno i podložno greškama. Zato ADO.Net uvodi tzv. parametriziranje naredbi kod kojeg se SQL naredbe višestruko pozivaju na puno jednostavniji način.

Sada ćemo "potopiti" dva jedrenjaka tako da ćemo im postaviti vrijednost varijable *NaDnuMora* u *true* bez ponovnog definiranja SQL stringa.

```
using System;
using System.Data.OleDb;

class Test
{
      static void Main()
      {
            string Database = "Brodovi.mdb";
            OleDbConnection Conn = new OleDbConnection
                  ("Provider=Microsoft.Jet.OLEDB.4.0;" + "Data Source=" +
                  Database + ";Persist Security Info=False");

            string Sailboat1 = "Black Magic";
            string Sailboat2 = "Alinghi";

            try
            {
                  Conn.Open();
                  string sSQL = "UPDATE PoznatiBrodovi SET " +
                        "NaDnuMora = ?" + " WHERE Naziv = ?";

                  OleDbCommand Command = new OleDbCommand(sSQL, Conn);

                  Command.Parameters.Add("NaDnuMora", OleDbType.Boolean);
                  Command.Parameters.Add("Naziv", OleDbType.Char);
                  Command.Parameters["NaDnuMora"].Value = true;
                  Command.Parameters["Naziv"].Value = Sailboat1;
                  Command.ExecuteNonQuery();

                  Command.Parameters.Add("NaDnuMora", OleDbType.Boolean);
                  Command.Parameters.Add("Naziv", OleDbType.Char);
                  Command.Parameters["NaDnuMora"].Value = true;
                  Command.Parameters["Naziv"].Value = Sailboat2;
                  Command.ExecuteNonQuery();
            }
            catch(OleDbException e)
            {
                  Console.WriteLine(e.Message);
            }
            finally
```

```
            {
                  Conn.Close();
            }
      }
}
```

Primjer 15.9

U SQL string stavimo upitnike na mjestima gdje trebaju doći varijable koje se nakon toga definiraju svojstvom *Parameters*. Prije nego što se varijablama dodijeli vrijednost potrebno im je odrediti i tip pomoću enum tipa *OleDbType*.
Ovdje također postoje razlike između različitih baza podataka tako da npr. MS SQL Server umjesto upitnika traži da se navede naziv varijable, ali za razliku od upravitelja koji mi koristimo u našim primjerima, ne zahtijeva da se varijable dodaju metodom *Add* u kolekciju *Parameters* redoslijedom kojim su navedeni u SQL stringu.
Ako bismo u gornjem primjeru zamijenili naredbe s metodama *Add* za pojedini slog, baza podataka ne bi se promijenila. Ali pripazite na to da se u slučaju ako UPDATE ne promijeni niti jedan slog, ne generira nikakva iznimka, jednostavno baza ostaje nepromijenjena.

Možda vam se ovo čini kao gomilanje kôda, međutim za više naredbi od dvije sigurno bismo sve stavili u neku petlju koja bi osjetno smanjila veličinu programskog kôda. Pogledajmo to na sljedećem primjeru:

```
int [] ShipIDs = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
int [] ShipLengths = {2, 4, 6, 8, 10, 2, 4, 6, 8, 10, 2, 4, 6, 8, 10};
try
{
      Conn.Open();
      string sSQL = "UPDATE PoznatiBrodovi SET Duzina =
                        Duzina + ? WHERE ID = ?";
      OleDbCommand Command = new OleDbCommand(sSQL, Conn);

      for(int i=0;i<ShipIDs.Length;i++)
      {
            Command.Parameters.Add("Length", OleDbType.Numeric);
            Command.Parameters.Add("PrimaryID", OleDbType.Numeric);
            Command.Parameters["Length"].Value = ShipLengths[i];
            Command.Parameters["PrimaryID"].Value = ShipIDs[i];
            Command.ExecuteNonQuery();
      }
}
```

Sad već ne izgleda kao glomazni kôd. Ovaj programski kôd će svakom brodu čiji su ID-ovi u nizu *ShipIDs* povećati dužinu za vrijednost u nizu *ShipLengths*.
Može se vidjeti da naziv argumenta u kolekciji *Parameters* ne mora biti isti kao i naziv polja u bazi podatataka, bitno je samo pratiti navedeni redoslijed u SQL naredbi.

Adapteri podataka (Data adapters)

Korištenje parametara za izmjene u bazi podataka pojednostavljuje stvar, ali ipak smo ostali vezani za SQL sintaksu. Bilo bi puno bolje kad bismo izmjene nad podacima u bazi mogli raditi koristeći C# naredbe. Još kad bi to sve moglo biti u *offline* modu te da se na kraju jednim potezom sve promjene vrate u bazu, gdje bi ADO.Net-u bio kraj.
Kao što ste i očekivali, ADO.Net može sve ovo nabrojeno i to se postiže korištenjem adaptera podataka (Data adapters). Oni omogućuju da se cijela baza podataka kopira u memoriju kao objekt klase *DataSet*. Nakon toga sve što radimo s podacima radimo na tom objektu, u

offline-u naravno i kad su promjene gotove, podaci se jednostavno vrate u bazu. Adapter podataka ćemo na djelu vidjeti u sljedećem primjeru u kojem ćemo svim brodovima iz naše baze promijeniti vrijednost varijable *NaDnuMora*. Ako je *true*, postat će *false* i obrnuto.

```
using System;
using System.Data;
using System.Data.OleDb;

class Test
{
      static void Main()
      {
            string Database = "Brodovi.mdb";

            OleDbDataAdapter Adapter = new OleDbDataAdapter("SELECT * FROM
                        PoznatiBrodovi",
                  "Provider=Microsoft.Jet.OLEDB.4.0;" + "Data Source=" +
                  Database + ";Persist Security Info=False");

            OleDbCommandBuilder Builder = new OleDbCommandBuilder(Adapter);

            DataSet ds = new DataSet();
            Adapter.Fill(ds, "PoznatiBrodovi");
            DataTable table = ds.Tables["PoznatiBrodovi"];

            foreach(DataRow row in table.Rows)
                  row["NaDnuMora"] = !(bool)row["NaDnuMora"];

            Adapter.Update(ds, "PoznatiBrodovi");
            ds.AcceptChanges();
      }
}
```

Primjer 15.10

Nemamo više objekta klase *OleDbConnection*, nego sve radimo s objektom *Adapter* klase *OleDbDataAdapter* kojemu kao argument prosljeđujemo SQL naredbu i konekcijski string. *Adapter* otvara, ali i zatvara bazu. Nakon toga kreira se objekt *ds* klase *DataSet* koji predstavlja bazu podataka u memoriji. Metodom *Fill* objekta *Adapter* jednostavno se kopira tablica *PoznatiBrodovi* u bazu podataka u memoriji *ds* i njeno svojstvo *Tables*. Sve je isto kao i u pravoj bazi podataka, objekt klase *DataTable* predstavlja tablicu, objekt klase *DataRow* predstavlja slogove u tablici, a onaj klase *DataColumn* predstavlja polja.

Nakon toga sve što radimo s podacima, radimo na poznati način, koristeći C# naredbe i klase .Net Frameworka. Kad je s promjenama gotovo, potrebno je sve to potvrditi i vratiti natrag u pravu bazu podataka metodama *Adapter.Update* i *ds.AcceptChanges*.

Argument svojstva *Tables* možemo navesti preko brojčanog indeksa ili indeksa sa stringom. Svakako preporučujemo korištenje *Tables["PoznatiBrodovi"]* umjesto *Tables[0]* jer je u prvom načinu odmah vidljivo o kojoj se tablici radi. Isto tako, kod objekta *row* u *foreach* petlji preporučuje se navođenje stringa, odnosno imena polja kao argumenta, dakle *row["NaDnuMora"]*, a ne *row[5]*.

Ipak, promjene na bazi podataka može napraviti samo SQL naredba (INSERT, UPDATE ili DELETE). U našem programu nigdje se ne vidi naredba UPDATE koja mijenja vrijednost polja *NaDnuMora* u slogovima tablice. Kako se onda promjene događaju?

Odgovor je u klasi *OleDbCommandBuilder* i svojstvima klase *OleDbDataAdapter*: *InsertCommand*, *UpdateCommand* i *DeleteCommand* koji su inicijalno postavljeni na vrijednost *null*, ali će ih kreirati kreiranje objekta klase *OleDbCommandBuilder*. Dakle, svojstvo *UpdateCommand* zaslužno je za generiranje SQL naredbe koja će izmijeniti vrijednosti u bazi podataka.

ADO.Net je veliki zadatak za desetak stranica, ali se nadamo, kao i kod XML-a, da je bilo dovoljno da vas upozna s osnovnim i najvažnijim stvarima.
Na početku knjige napomenuli smo veliku stvar koju je .Net donio u svijet izrade software-a, a to je da se praktički na isti način pišu aplikacije za razne vrste računala. To je tako i s platformom ADO.Net. Sve primjere koje smo naveli moći ćete upotrijebiti i u nekoj web aplikaciji za pristup bazi podataka na web serveru. Dorada će biti potrebna, ali sve klase i objekti su jednako na raspolaganju kod obiju platformi. Isto vrijedi i za sami jezik C#, sve što smo u cijeloj knjizi naučili, možete primijeniti u različitim aplikacijama, za web servere, mobitele, ručna računala ili desktop aplikacije. Sintaksa je ista, klase su manje više također iste, a gotovo da je ista i struktura programa.

15.3. Web servisi

Za kraj smo ostavili web servise (*web services*), posebne programske module koji nisu privilegija ni C# ni .Neta iako smo mišljenja da je upravo to dobitna kombinacija za izradu web servisa.
Web servisi su programski moduli koji su smješteni na web serveru, sastoje se od web metoda koje se pozivaju od strane korisnika servisa i savršeno se uklapaju u zamišljeni mrežni svijet međusobno povezanih računala i aplikacija. Koristeći dobro poznate protokole HTTP (Hyper Text Transfer Protocol) za komunikaciju i SOAP (Simple Object Access Protocol) za prijenos podataka, web servis napisan npr. u C# smješten na Windows web serveru može biti pozvan od nekog klijenta, odnosno desktop aplikacije napisane također u C#, ali i od strane PHP skripte na Linux web serveru kao i od Java aplikacije na mobitelu.
Sve to moguće je zbog otvorenosti i opće prihvaćenosti SOAP protokola za razmjenu podataka koji je, pogađate, baziran na XML-u.

Kakve metode mogu biti sadržaj web servisa?
Praktički sve, kako bi se reklo, od igle do lokomotive.
U našim primjerima web servisa koristit ćemo web server na lokalnom računalu tzv. IIS (Internet Information Server) koji se instalira na sljedeći način:

1. *Start -> Control Panel -> Add or Remove Programs*
2. *Add/Remove Windows Components*. Pojavit će se *Windows Components Wizard*
3. Slijedite upute na zaslonu za instalaciju, uklanjanje ili dodavanje komponenti

Za detaljnija uputstva o instalaciji IIS-a pogledajte na:
http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/iiiisin2.mspx?mfr=true

Sad ćemo napraviti jedan web servis. Bit će to servis *AlphabetString* s dvije metode za tip *string*.

- metoda koja će vratiti ASCII vrijednost znaka iz stringa koji je najmanji po abecedi
- metoda koja će vratiti ASCII vrijednost znaka iz stringa koji je najveći po abecedi

```
<%@ WebService Language="C#" Class="AlphabetString" %>

using System;
using System.Web.Services;

[WebService(Name="AlphabetStringWebService", Description="Dodatne metoda za
                        klasu string")]
class AlphabetString
{
      [WebMethod(Description="Vraća prvi znak po abecedi iz prosljeđenog
                              stringa")]
      public int FirstByAlphabet(string s)
      {
            char c = 'z';
            s = s.ToLower();

            for(int i=0;i<s.Length;i++)
            {
                  if(s[i] < c  &&  s[i] >= 97  &&  s[i] <= 122)
                        c = s[i];
            }
```

```
            return (int)c;
      }

      [WebMethod(Description="Vraća zadnji znak po abecedi iz prosljeđenog
                              stringa")]
      public int LastByAlphabet(string s)
      {
            char c = 'a';
            s = s.ToLower();

            for(int i=0;i<s.Length;i++)
            {
                  if(s[i] > c  &&  s[i] >= 97  &&  s[i] <= 122)
                        c = s[i];
            }
            return (int)c;
      }
}
```

Primjer 15.11

Kao što vidimo, web servis nema nikakve funkcije *Main* koja bi označavala početak izvođenja. Također, web servisi nemaju nikakvo korisničko okruženje, oni su tu jednostavno da bi bili pozvani od strane drugih programskih modula (klijentski programi, web aplikacije, web stranice, drugi web servisi itd.)
Web servisi spremaju se u datoteke s nastavkom *asmx* i na početku je potrebno navesti direktivu koja najmanje mora sadržavati atribut *Class* koji će označavati klasu web servisa. Svaka metoda u klasi web servisa mora imati atribut *WebMethod*, a može imati i parametre kao npr. opis (Description).

Ovako napisan web servis, spremljen na web serveru pod imenom *AlphabetString.asmx* je spreman za pružanje usluga i svakome tko u svom pretraživaču napiše URL ove datoteke (u obliku *http://localhost/. . ./AlphabetString.asmx*) prikazat će se sve metode koje web servis pruža zajedno s kratkim opisom. Nakon poziva neke od njih metoda web servisa će tražiti od vas da upišete argument te jednostavno kliknete na tipku Invoke koja će pozvati metodu na izvršenje.
Ako odaberete metodu *LastByAlphabet*, upišete npr. tekst “Happy New Year”, web servis će vratiti ovakav tekst u XML formatu:

```
<?xml version="1.0" encoding="utf-8" ?>
  <int xmlns="http://tempuri.org/">121</int>
```

Ono što je rezultat izvršenja metode je ispis broja 121 koji je ASCII vrijednost znaka ‘y’ kao najvećeg po abecedi u proslijeđenom tekstu (metoda prije usporedbe cijeli string pretvara u mala slova). Vidimo da XML dokument definira i tag <int> kao oznaku tipa varijable koji vraća metoda.

Poziv web servisa iz klijentskog programa

Već smo naveli da web servise mogu pozvati razne vrsta software-a, pa tako i klijentska aplikacija. Koristeći već napisani web servis *AlphabetString*, napravit ćemo program koji će koristeći metode web servisa pronaći najmanji i najveći znak u stringu.

```
using System;
```

```
class Test
{
    public static void Main()
    {
        AlphabetStringWebService s = new AlphabetStringWebService();

        Console.WriteLine(s.FirstByAlphabet("Merry Christmas"));
        Console.WriteLine(s.LastByAlphabet("Happy New Year"));
    }
}
```

Primjer 15.12

Ispis:
97
121

Program je jednostavan da jednostavniji ne može biti. Ali odakle klasa *AlphabetStringWebService*? U imeniku *System* sigurno nije, a niti jedan drugi imenik nismo uključili. Kako onda možemo kreirati objekt te klase?
Kod navođenja URL adrese web servisa u prethodnom primjeru ovakva pitanja su suvišna, jasno je da se sve događa na serveru na kojem je i smješten web servis, ali ovo je ipak klijentska aplikacija na našem računalu, dok je web servis na tko zna kojem računalu. (Dobro, u ovom slučaju je i on našem računalu jer smo kreirali lokalni web server, ali je ipak odvojen od naše klijentske aplikacije jer ga ona vidi kao virtualni URL. Kao što će i svi web servisi koje ćemo pozivati biti odvojeni u svakom pogledu od aplikacija koje ih pozivaju).

Mnogo pitanja, ali dovoljan je samo jedan odgovor.
Klasa *AlphabetStringWebService* nalazi se u web servis proksiju (Web Service Proxy), objektu koji predstavlja lokalnu prezentaciju udaljenog web servisa. Proksi tako predstavlja vezu između klijentske aplikacije i web servisa na web serveru.
Proksi se kreira pozivom programa *wsdl.exe* koji dolazi s instalacijom .NET Framework Software Development Kita (SDK), a kojemu se kao argument doda URL *asmx* datoteke s web servisom.

wsdl http://localhost/Samples/WebServices/AlphabetString.asmx

Ovo je URL datoteke *AlphabetString.asmx* na našem računalu, a ako vam se putanja razlikuje, promijenite je u argumentu programa *wsdl*. Nakon izvršenja ove naredbe na računalu će vam se kreirati proksi web servis u obliku datoteke *AlphabetStringWebService.cs*. Tu datoteku moramo navesti i kod prevođenja samog klijentskog programa:

```
csc Test.cs AlphabetStringWebService.cs
```

Ako pogledamo što se nalazi u datoteci, vidjeti ćemo između ostalog i ovo:

```
public class AlphabetStringWebService :
             System.Web.Services.Protocols.SoapHttpClientProtocol
{
      /// <remarks/>
      public AlphabetStringWebService()
      {
            this.Url =
            http://localhost/Samples/WebServices/AlphabetString.asmx";
      }
      . . .
}
```

Sad vidimo kako naš klijent vidi klasu *AlphabetStringWebService*. Ako web servis nije napravljen s .Net Framework-om ili ako nam nije poznat URL *asmx* datoteke, da bismo kreirali proksi, moramo imati datoteku koja predstavlja tzv. wsdl ugovor *(wsdl contract)* web servisa i opet pozvati program *wsdl,* ali s datotekom *wsdl ugovora* kao parametrom:

wsdl AlphabetString.wsdl

To ćemo napraviti u sljedećem poglavlju u kojem ćemo napraviti primjer s Google web servisom za pretraživanje interneta.

Kako će netko znati da vaša stranica ima neke, njemu zanimljive web servise?
On bi čak i platio za njihovo korištenje, ali jednostavno ne zna da oni uopće postoje. Zato je uveden sustav DISCO (kao kratica od discovery, odnosno otkrivanje) datoteka koji svaka stranica koja nudi web servise postavi na svoj web server. DISCO datoteka je datoteka u XML formatu koja opisuje sve web servise koji se nalaze na serveru, tako da pogled u tu datoteku omogućuje uvid u sve ponuđene web servise.

Google web servisi

I veliki Google spoznao je korisnost web servisa pa vam daje mogućnost da korištenjem njegovih web servisa uključite moćni Google sustav pretraživanja u svoje web stranice, aplikacije, vlastite web servise itd. Korištenje Google web servisa je besplatno (zasad je tako) uz sljedeće uvjete:

- trebate imati *Licence key* koji besplatno možete dobiti na adresi *http://www.google.ca/apis*
- broj poziva servisa za pretraživanje interneta ne smije premašiti 1000 na dan

Za komercijalizaciju Google web servisa te za ostale servise koje možete koristiti pogledajte uvjete na *www.google.com*.

S obzirom da nemamo pristup *asmx* datoteci Google web servisa, učitat ćemo datoteku *http://api.google.com/GoogleSearch.wsdl* i spremiti je na disk pod imenom *GoogleSearch.wsdl*. Nakon toga ćemo pokrenuti program *wsdl* i kao argument mu proslijediti tu datoteku:

wsdl GoogleSearch.wsdl

Ovo će kreirati proksi objekt pod imenom *GoogleSearchService.cs* i nakon toga je sve spremno za izradu klijenta koji će koristiti metode web servisa za pretraživanje.

Mi ćemo u sljedećem primjeru napraviti klijentski program koji će simulirati pritisak na tipku *I'm feeling lucky* (*Prati me sreća*) na *www.google.com*. Pritisak na tu tipku automatski otvara prvu pronađenu stranicu.
Program će iz naše baze podataka *Brodovi* slučajnim odabirom uzeti naziv jednog broda i otvoriti stranicu koju bi Google izlistao kao prvu za slučaj pretraživanja interneta svega što ima veze s tim brodom. Također, podaci o 10 prvih pronađenih stranica (link, naslov, kratki opis) će se spremiti u datoteku pod imenom traženog broda.

```
using System;
using System.IO;
using System.Threading;
using System.Diagnostics;
using System.Data.OleDb;

class DoIFeelLucky
{
	private string FileToSave = "About";

	public DoIFeelLucky()
	{
	}

	public string FindShip(int ind)
	{
		string Database = "Brodovi.mdb";
		OleDbConnection Conn = new OleDbConnection
			("Provider=Microsoft.Jet.OLEDB.4.0;" + "Data Source=" +
			Database + ";Persist Security Info=False");

		try
		{
			Conn.Open();
			OleDbCommand Command = new OleDbCommand(
				"SELECT * FROM PoznatiBrodovi", Conn);
			OleDbDataReader Reader = Command.ExecuteReader();

			int i = 0;
			while(Reader.Read())
				if(i++ == ind)
				{
					string s = (string)Reader["Naziv"];
					Reader.Close();
					FileToSave += s;
					FileToSave += ".txt";
					return s;
				}
			Reader.Close();
		}
		catch(OleDbException e)
		{
			Console.WriteLine(e.Message);
		}
		finally
		{
			Conn.Close();
		}
		return null;
	}

	public void CallGoogleSearch(string toSearch)
	{
		GoogleSearchService Search = null;
		GoogleSearchResult Results = null;

		try
		{
			Search = new GoogleSearchService();
			Results = Search.doGoogleSearch(
				"Vaš Licence Key",
```

```
                        toSearch, 0, 10, false, "", false, "", "", "");
            }
            catch(System.Net.WebException e)
            {
                  Console.WriteLine(e.Message);
                  return;
            }

            Process IE = new Process();
            IE.StartInfo.FileName = "iexplore.exe";
            IE.StartInfo.Arguments = Results.resultElements[0].URL;
            IE.Start();

            SaveToDisk(Results);
      }

      public void SaveToDisk(GoogleSearchResult Results)
      {
            StreamWriter FWrite = null;
            try
            {
                  FWrite = new StreamWriter(FileToSave, false,
                        System.Text.Encoding.ASCII);
                  foreach(ResultElement r in Results.resultElements)
                  {
                        FWrite.WriteLine(r.URL);
                        FWrite.WriteLine(r.title);
                        FWrite.WriteLine(r.snippet);
                        FWrite.WriteLine();
                        FWrite.WriteLine();
                  }
            }
            catch (Exception e)
            {
                  Console.WriteLine(e.Message);
            }
            finally
            {
                  if(FWrite != null)
                        FWrite.Close();
            }
      }
}

class Test
{
      static void Main()
      {
            DoIFeelLucky fl = new DoIFeelLucky();
            Random x = new Random();
            string ShipForSearch = fl.FindShip(x.Next(15));

            ShipForSearch += " + ship";
            fl.CallGoogleSearch(ShipForSearch);
      }
}
```

Primjer 15.13

Napomena:

1. Obratite pozornost da stranica koja se otvara ne mora biti ona koju biste otvorili preko *www.google.com* jer ako idete direktno preko *www.google.com,* onda Google radi prilagodbe za Hrvatsku.
2. U metodi *doGoogleSearch* kao prvi argument morate upisati vaš *Licence key*

Definirana je klasa *DoIFeelLucky* koja radi većinu zadanog posla. Najprije se metodom *FindShip* pronalazi naziv broda koristeći klase i metode ADO.Net. Nakon što je naziv broda pronađen, poziva se metoda *CallGoogleSearch* koja će napraviti najveći posao i kojoj se kao argument prosljeđuje naziv broda. Nazivu broda dodaje se string " + ship" tek da bi pretraživanje bilo točnije.
Glavna stvar u metodi *CallGoogleSearch* je poziv Google web servisa, odnosno kreiranje objekta klase *GoogleSearchService* te poziv njegove metode *doGoogleSearch*. Ova metoda ima nekoliko argumenata:

- **Key** – Google License key
- **QueryTerm** – Izraz koji se pretražuje. Može biti sve kao i kod standardnog pretraživanja, riječ, više riječi, fraza itd.
- **Start** – Indeks prvog vraćenog rezultata pretraživanja
- **MaxResults** – Maksimalan broj vraćenih rezultata pretraživanja. Može biti između 1 i 10
- **Filter** – Ako je *true,* eliminira dvostruke rezultate, odnosno rezultate s istim naslovom.
- **Restricts** – Omogućuje traženje u samo nekim državama ili područjima
- **SafeSearch** – Ako je *true,* eliminira neprimjerne sadržaje
- **LanguageRestrict** – Omogućuje traženje u samo nekim jezičnim područjima
- **InputEncoding** – Trenutno se ignorira
- **OutputEncoding** – Trenutno se ignorira

Metoda će vratiti rezultate pretraživanja kao objekt klase *GoogleSearchResult* koji sadrži niz objekata tipa *ResultElement* od kojih svaki predstavlja jedan vraćeni rezultat. Mi koristimo u našem primjeru tri člana objekta tipa *ResultElement*: URL za otvaranje rezultata pretraživanja, te pored toga još i članove *title* (naslov) i *snippet* (kratki opis) koji se preko *foreach* petlje spremaju u datoteku. Datoteci se ime dinamički kreira u formatu *About + Naziv broda + .txt*.

Ovo je bio zadnji primjer u knjizi i kao takav zaslužio je biti onaj s najviše programskog kôda. Nadamo se da ste u knjizi pronašli barem nešto što bi moglo unaprijediti vaše znanje, a ako ćete od toga steći i nekakvu korist, onda je zadovoljstvo obostrano. Svaka knjiga mora imati nedostataka, ako ništa drugo čitatelji nisu isti, pa što je nekome dobro, nekome može biti manje dobro. Ili što je nekome prejednostavno, nekome može biti suviše složeno. Naš cilj bio je napraviti knjigu koja će pružiti cijelovit prikaz ne samo programskog jezika C#, nego i osnova programiranja.
Zato ako imate pohvale, pohvalite nas, a isto tako ako mislite da nešto nije dobro urađeno u knjizi, javite nam. Trudili smo se da se svi programi prevode i izvode bez grešaka, osim u 1-2 primjera gdje se htjelo pokazati kako prevoditelj neće prevesti neke stvari, ali je to onda naglašeno. Ipak, greške su uvijek moguće, pa i tu vrijedi isto kao i za pohvale i kritike, upozorite nas na njih.

Dodaci:

Sadržaj:

- Verzija 2.0 donijela je nekoliko novosti koje navodimo u ovom dodatku

Dodatak A: C# 2.0

Verzija 2.0 donijela je nekoliko novosti koje navodimo u ovom dodatku.

Parcijalne (Partial) klase

Sustav parcijalnih klasa (ali i struktura i sučelja) omogućuje da se definicija jedne klase razdvoji u dva dijela od kojih svaki može biti u posebnoj datoteci. Na taj način omogućena je izrada jedne klase od strane više programera. Parcijalne klase označuju se ključnom riječi *partial*.

```
using System;

partial class NekaKlasa
{
      private int xVar = 10;
      public int XVar {get {return xVar;}}
}

partial class NekaKlasa
{
      private int yVar = 20;
      public int YVar {get {return yVar;}}
}

class Test
{
      public static void Main()
      {
            NekaKlasa tmp = new NekaKlasa();
            int z = tmp.XVar + tmp.YVar;
            Console.WriteLine(z);
      }
}
```

Primjer 16.1

Ispis:
30

Iako u dva dijela, riječ je o jednoj klasi *NekaKlasa* i rad s njenim članicama je isti kao i kod klasa koje nisu razdijeljene u više dijelova. Zbog jednostavnosti prikaza oba dijela smo stavili u jednu datoteku, ali kao što smo naglasili, dijelovi klase mogu biti razdvojeni i u više datoteka.

Neimenovane (Anonymous) metode

Delegati su referentni tipovi čija instanca sadržava referencu na metodu koja će biti pozvana s pozivom delegata. U tu je svrhu trebalo definirati posebnu metodu pa C# u veziji 2.0 uvodi pojam neimenovane metode koja će biti pozvana s pozivom delegata.

```
using System;

public class Test
{
      private static int y = 20;
```

```
        public delegate void Del(double x);

        public static void Main()
        {
                int z = 30;

                Del d = delegate(double x)
                {
                        Console.WriteLine(x + y + z);
                };

                F(d);
        }

        public static void F(Del d)
        {
                d(10);
        }
}
```

Primjer 16.2

Ispis:
60

Neimenovana metoda nema imena jer je implementirana unutar metode *Main*. U implementaciji neimenovane metode su vidljive, odnosno dostupne i članice klase (varijabla *y*) i lokalne varijable (varijabla *z*), a prednost ovakvog korištenja delegata je u efikasnijem izvršenju jer je implementacija delegata realizirana direktno unutar postojeće metode. Ipak neimenovane metode bi trebalo koristiti samo ako je riječ o manjem kôdu jer inače narušavaju preglednost same metode u kojoj su definirane.

Generički tipovi (Generics)

Jedan od problema koji se uočio kod kolekcijskih klasa bila je njihova slaba tipska određenost. Tako je klasa *ArrayList* sve spremala kao tip *object* što je značilo boksiranje i odboksiranje u vremenu izvođenja što je prilično neefikasno. Zato su u C# 2.0 uvedeni tzv. generički tipovi (*Generics*) koji su ojačali tipsku određenost kolekcijskih klasa. S verzijom 2.0 došlo je i nekoliko predefiniranih klasa koje su podržavale generičke tipove kao što je npr. klasa *System.Collections.Generic.List*

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

public class MyClass
{
        private string element = "";

        public MyClass(string s)
        {
                element = s;
        }

        public string Element
        {
                get {return element;}
                set {element = value;}
        }
```

```
}

class Test
{
	static void Main()
	{
		List<MyClass> Arr = new List<MyClass>();
		Arr.Add(new MyClass("Zlato"));
		Arr.Add(new MyClass("Srebro"));
		Arr.Add(new MyClass("Bronca"));

		foreach(MyClass x in Arr)
			Console.WriteLine(x.Element);
	}
}
```

Primjer 16.3

Ispis:
Zlato
Srebro
Bronca

Sintaksa korištenja generičkih tipova ista je kao i kod *template* iz C++ što znači da se tip navodi unutar znakova < i >. Kod instanciranja objekta pomoću generičkog tipa <*MyClass*> definirali smo da klasa radi samo s tipom *MyClass*, što znači da ovako nešto prevoditelj ne bi dopustio:

```
Arr.Add("Četvrto mjesto");
```

jer nije moguće implicitno pretvoriti tip *string* u tip *MyClass*.
Klasa *ArrayList* ovo bi dopustila, ali bi se kod izvršenja *foreach* petlje generirala iznimka zbog nemogućnosti pretvaranja tipa *string* u *MyClass*.

Generički tip može biti bilo koji tip i vrijede pravila implicitne pretvorbe:

```
List<double> Arr = new List<double>();
Arr.Add(3.14);
Arr.Add(2.718);
Arr.Add(5);
```

U ovom primjeru treći član je *integer* koji će se implicitno pretvoriti u *double*.

Svaka klasa koja nije zapečaćena može se naslijediti, pa tako i predefinirana klasa *List* koja podržava generičke tipove:

```
class MyList<T> : List<T>
{
	public new void Add(T x)
	{
	}
	// . . .
}
```

Metoda *Add* nije označena kao virtualna u klasi *List* pa smo morali navesti ključnu riječ *new* kao znak da klasa *MyList* želi definirati svoju metodu za dodavanje novih članova. Da nismo naslijedili klasu *List*, ni *new* ne bi bio potreban.

Klasa Dictionary

U imeniku *System.Collections.Generic* nalazi se i klasa *Dictionary*, još jedna kolekcijska klasa koja podržava generičke tipove. Ona nam daje mogućnost da dvije vrijednosti spremamo kao ključ/vrijednost par (*key/value pair*). Naravno, kako je riječ o klasi koja podržava generičke tipove, ključ i vrijednost mogu biti varijable različitog tipa što se navodi kod definicije objekta.

```
Dictionary<string, int> Prefix = new Dictionary<string, int>();
string[] name = new string[] {"kilo", "mega", "giga", "tera", "peta"};
int exp = 3;

for(int i = 0; i<name.Length; i++, exp += 3)
      Prefix.Add(name[i], exp);

Console.WriteLine("Mega = 10^{0}", Prefix["mega"]);
```

U ovom primjeru ključ je tipa *string*, a vrijednost je tipa *int*. Objekt tipa *Dictionary* ne može sadržavati dva ista ključa, dok vrijednosti mogu biti iste.

Kreiranje generičke klase

Da, možemo kreirati i vlastitu klasu koja će podržavati generičke tipove:

```
public class GenClass<T>
{
  T datum;

  public T Datum
  {
      get
      {
            return datum;
      }
      set
      {
            datum = value;
      }
  }
}
```

Kreirali smo klasu *GenClass* koja ima samo jednu varijablu kojoj pristupamo preko svojstva. Tip varijable se određuje kod kreiranja objekta tako da se tzv. *placeholder* T zamijeni s željenim tipom:

```
GenClass<string> ByString = new GenClass<string>();
ByString.Datum = "Jedanaesti lipanj dvije tisuće i sedme";

GenClass<long> ByLong = new GenClass<long>();
ByLong.Datum = 11062007;

GenClass<DateTime> ByDate = new GenClass<DateTime>();
ByDate.Datum = new DateTime(2007, 6, 11);
```

Koristeći jednu te istu klasu definirali smo tri objekta s različitim tipom varijable članice *datum*. Ako sad pozovemo:

```
Console.WriteLine(ByString.Datum);
Console.WriteLine(ByLong.Datum);
```

```
Console.WriteLine(ByDate.Datum.ToShortDateString());
```

ispisat će se:
Jedanaesti lipanj dvije tisuće i sedme
11062007
11.6.2007

Generičke tipove možemo koristiti i kod poziva metoda:

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

class Test
{
   static void Main()
   {
      int[] ArrInt = {1, 2, 3, 4, 5};
      double[] ArrDouble = {1.5, 2.5, 3.5, 4.5, 5.5};
      string[] ArrString = {"Jedan", "Dva", "Tri", "Četiri", "Pet"};

      UseGenerics(ArrInt);
      UseGenerics(ArrDouble);
      UseGenerics(ArrString);
   }

   static void UseGenerics<T> (T[] Arr)
   {
      foreach(T el in Arr)
         Console.Write(el + " ");
      Console.WriteLine();

   }
}
```

Primjer 16.4

Ispis
1 2 3 4 5
1,5 2,5 3,5 4,5 5,5
Jedan Dva Tri Četiri Pet

Parametar T zamjenjuje bilo koji od tri tipa, a alternativa ovakvom korištenju generičkih tipova bile bi tri preopterećene metode *UseGenerics* za svaki od tipova *int, double* i *string*.

Dakle, da zaključimo, generički tipovi kao najvažnija novost u verziji 2.0 poboljšavaju:

- tipsku sigurnost
- ponovnu upotrebljivost napisanog kôda
- efikasnost izvršenja kôda

Iteratori (Iterators)

U poglavlju o *ArrayList* klasi smo vidjeli koliko toga moramo napraviti da bi neka klasa mogla implementirati *foreach* petlju. Ponovit ćemo:

Klasa mora implementirati metodu *GetEnumerator* koja će vratiti objekt tipa neke klase. Ta klasa čijeg tipa objekt se vraća mora sadržavati:

- svojstvo *Current* koje će vratiti član koji predstavlja aktivni (trenutni) element u kolekciji
- metodu *MoveNext* koja se pozicionira na sljedeći element kao aktivni i vraća *false* ako više nema elemenata u kolekciji

C# 2.0 opet pojednostavljuje stvar uvodeći iteratore. Sad je dovoljno kreirati metodu *GetEnumerator* koja će vratiti vrijednost člana niza kolekcijske klase koristeći novu ključnu riječ *yield*. Sad ćemo ponovo napraviti primjer 8.5 koristeći iteratore.

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

public class CollectionClass
{
      double [] Elementi;

      public CollectionClass()
      {
            Elementi = new double[] { 1.1, 2.2, 3.3, 4.4 };
      }

      public IEnumerator<double> GetEnumerator()
      {
            foreach(double d in Elementi)
                  yield return d;
      }
}

class Test
{
      public static void Main()
      {
            CollectionClass Arr = new CollectionClass();

            foreach(double d in Arr)
                  Console.WriteLine(d);
      }
}
```

Primjer 16.5

Ispis:
1,1
2,2
3,3
4,4

Očita je razlika u jednostavnosti, a ona je postignuta zahvaljujući prevoditelju koji automatski generira potrebne metode za iteraciju kao što su *MoveNext* i *Current*.
Izraz *yield return* vraća trenutni član niza kolekcijske klase dok će *yield break* prekinuti iteraciju.

```
foreach(double d in Elementi)
{
      if(d > 3)
            yield break;
      yield return d;
}
```

Ovo bi ispisalo samo prva dva člana niza jer bi prije vraćanja trećeg člana naredba *yield break* prekinula iteraciju.

Statičke klase

Statičke klase su klase koje imaju samo statičke članove i označavaju se ključnom riječju *static* ispred deklaracije klase:

```
static class MyClass
{
      // . . .
}
```

S obzirom da statička klasa ne može imati niti jedan nestatički član, onda nema nikakvog razloga za instanciranjem objekta statičke klase, pa se ni ne može kreirati takav objekt. Statičke klase su i zapečaćene i ne mogu imati definiran konstruktor, ali mogu implementirati statički konstruktor.

Vrijednosni tipovi koji mogu imati vrijednost *null* (nullable types)

Opet vidimo koliko hrvatskih riječi moramo upotrijebiti da bismo preveli neki engleski izraz. Za referentne tipove i prije je bilo moguće da poprime vrijednost *null*, a sad je to moguće i za vrijednosne tipove. *Nullable* tip je osobito koristan kada se podaci iz baze podataka dohvaćaju u vrijednosne tipove jer vrijednosti pojedinih takvih polja mogu biti i nedefinirane kao npr. polje tipa *bool* koje može imati vrijednost *true, false*, ali može biti i nedefinirano. Ako bismo kod čitanja takvih nedefiniranih podataka koristili obične vrijednosne varijable, generirala bi se iznimka.

Sintaksa za deklaraciju *nullable* tipa je:

```
tip? identifikator
```

ili na primjeru:

```
int? x = 4;
x = null;
```

Dakle deklaracija je standardna osim što se dodaje znak ? iza tipa varijable. *Nullable* tipovi su zapravo instance strukture *System.Nullable* koja podržava generičke tipove i koja je ovako nekako definirana:

```
struct Nullable<T>
{
    public bool HasValue {}
    public T Value {}
    // . . .
}
```

pa smo gornji primjer mogli i ovako navesti:

```
Nullable<int> x = new Nullable<int>(4);
x = null;
```

Struktura *Nullable* ima dva *readonly* svojstva:

- *HasValue* koje vraća *true* ako varijabla ima vrijednost, odnosno *false* ako joj je vrijednost *null*.
- *Value* koje vraća vrijednost varijable ako varijabla ima vrijednost, inače generira iznimku *System.InvalidOperationException*

```
using System;

class Test
{
	static void Main()
	{
		int? x = 4;
		if(x.HasValue)
		{
			Console.WriteLine(x);
			x = null;
		}

		try
		{
			int y = x.Value;
		}
		catch (System.InvalidOperationException)
		{
			Console.WriteLine("Varijabla x ima null vrijednost");
		}
	}
}
```

Primjer 16.6

Ispis:

```
4
Varijabla x ima null vrijednost
```

Definirana je *nullable* varijabla *x* kojoj je kod deklaracije dodijeljena vrijednost. Zatim se svojstvom *HasValue* provjerava ima li varijabla vrijednost i ako ima, dodjeljuje joj se *null* vrijednost. To onda znači da će svojstvo *Value* kod definiranja varijable *y* generirati iznimku *System.InvalidOperationException* koja se hvata te se ispisuje poruka o *null* vrijednosti varijable *x*.

Pretvaranje *nullable* tipa u vrijednosni tip

Kako pretvoriti varijablu *nullable* tipa u odgovarajuću vrijednosnu varijablu?
Ako napravite ovo:

```
int? x = 4;
int y = (int)x;
```

u *y* će biti vrijednost 4, međutim da je u *x* bila vrijednost *null* generirala bi se iznimka *InvalidOperationException*. C# 2.0 uvodi operator ?? koji rješava ovaj problem:

```
int? x = null;
int y = x ?? 0;
```

Sad će u *y* biti vrijednost navedena iza operatora ??. To je u ovom slučaju 0, ali može biti bilo koja druga vrijednost (naravno, u rasponu dozvoljenih vrijednosti za taj tip).

Operator ::

Pogledajmo ovaj primjer:

```
using System;

namespace Konflikt
{
	class System
	{
		public static void Main()
		{
			System.Console.WriteLine("abcdef");
		}
	}
	// . . .
}
```

Ovo se neće prevesti i prevoditelj će javiti grešku da klasa *Konflikt.System* ne sadrži definiciju za *Console*. Prevoditelj naravno ne zna da smo mi željeli pozvati klasu *Console* iz globalnog imenika *System* i zato je potrebno to navesti koristeći operator ::

```
global :: System.Console.WriteLine("abcdef");
```

Sad će se program prevesti i ispisati *abcdef*. Iako je malo vjerojatno da bi netko mogao nazvati neku svoju klasu *System*, u većim projektima mogu se dogoditi i dupliranja putanja pojedinih članova klase, pa je uveden operator :: koji precizira putanju člana klase.

#pragma warning

Ova direktiva onemogućuje pojedino upozorenje prevoditelja. Sintaksa poziva je:

```
#pragma warning disable BrojUpozorenja
```

Ako npr. navedete naziv objekta tipa *Exception* u *catch* naredbi, a ne upotrijebite ga u tijelu naredbe, prevoditelj će javiti upozorenje broj 168 da je objekt deklariran, ali ne i upotrijebljen.

```
catch(Exception e)
{
}
```

Naredba *#pragma warning disable 168* će onemogućiti prikaz takvog upozorenja. Ako treba onemogućiti više različitih upozorenja, onda se njihovi brojevi navode jedan iza drugog odvojeni zarezom.

Ako nakon isključenja upozorenja u daljnjem kôdu treba vratiti prikaz upozorenja, koristi se ista sintaksa samo što se *disable* zamijeni s *restore*.

Dodatak B: Ključne riječi (Keywords)

`abstract`
Modifikator koji označuje da izvedena klasa mora implementirati metode označene kao *abstract*

`as`
Binarni operator koji pretvara (s operatorom *cast*) lijevi član u tip na desnoj strani

`base`
Varijabla s istim značenjem kao i *this* osim što se operator *base* odnosi na baznu klasu iz koje je klasa izvedena

`bool`
Logički tip varijable koji može biti *true* or *false* (istina ili laž)

`break`
Naredba koja vrši trenutačni prekid *switch* naredbe te *for* ili *while* petlje

`byte`
Neoznačeni tip varijable veličine 1 byte koji može primiti samo cjelobrojene vrijednosti

`case`
Oznaka u *switch* naredbi kod koje se provjerava je li parametar koji se navodi uz *case* isti kao argument u *switch* naredbi

`catch`
Dio *try catch* naredbe koji omogućuje hvatanje iznimaka navedenih kao parametar u *catch* dijelu naredbe

`char`
Znakovni tip podataka veličine 2 byte-a koji može primiti znakove iz Unicode seta znakova

`checked`
Operator koji prisiljava na provjeru premašenja dopuštenih vrijednosti

`class`
Referentni tip podataka koji se sastoji od podataka i metoda za rad s tim podacima, odnosno koji integira podatke i njihovu funkcionalnost

`const`
Modifikator varijable koji označava da je njena vrijednost konstantna

`continue`
Naredba u petljama koja uvjetuje prekid daljnjeg izvršenja naredbi u bloku, ali omogućuje nastavak izvođenja petlje

`decimal`
Decimalni tip podataka veličine 16 byte-ova i preciznosti na 27 ili 28 decimala

`default`
Oznaka u *switch* naredbi čije se naredbe izvršavaju ako nijedan parametar iz *case* dijela nije isti kao argument u *switch* naredbi

`delegate`
Referentni tip podataka čija instanca sadržava referencu na metodu koja će biti pozvana s pozivom delegata.

`do`
Prvi dio *do while* naredbe koja osigurava najmanje jedan prolaz petlje

`double`
Tip podataka s pomičnim zarezom veličine 8 byte-ova i preciznosti na 15 ili 16 decimala

`else`
Dio *if else* naredbe čiji se blok izvršava ako uvjet u *if* dijelu nije istinit

`enum`
Vrijednosni tip koji definira grupu numeričkih konstanti

`event`
Višemetodni delegat koji će pozvati metodu neke klase nakon ostvarenja događaja

`explicit`
Operator koji definira eksplicitnu konverziju (ne događa se automatski)

`extern`
Modifikator metode koji naznačuje da je metoda implementirana s neupravljanim kôdom (unmanaged code)

`false`
Oznaka za laž kod tipa *bool*

`finally`
Dio u *try catch* naredbi koji označava naredbe koje se moraju izvršiti bez obzira na ostvarenje iznimke

`fixed`
Naredba koja fiksira referentni tip tako da ne bude očišćen čišćenjem memorije *(garbage collection)*

`float`
Tip podataka s pomičnim zarezom veličine 4 byte-a i preciznosti na 7 decimala

`for`
Naredba za izvršenje jedne ili više naredbi više puta sve dok je uvjet u *for* naredbi istinit

`foreach`
Posebna vrsta petlje koja omogućava iteraciju kroz vrijednosti sadržane u objektu koji implementira *IEnumerable* sučelje *(interface)*

`get`
Modifikator koji vraća vrijednost svojstva (*property*)

`goto`
Naredba koja omogućuje bezuvjetan skok na naredbu označenu labelom

`if`
Naredba koja omogućuje uvjetovano izvršenje programa. Kao argument mora se navesti izraz koji vraća tip *bool*

`implicit`
Operator koji definira implicitnu konverziju (događa se automatski)

`in`
Operator između tipa i niza koji implementira *IEnumerable* sučelje *(interface*)

`int`
Cjelobrojni označeni tip veličine 4 byte-a

`interface`
Skup metoda, svojstava, događaja i indeksa koje klasa koja nasljeđuje sučelje (*interface)* mora implementirati.

`internal`
Modifikator pristupa koji naznačava da je pristup članu klase moguć samo unutar skupa (*assembly*)

`is`
Operator koji uspoređuje je li lijevi član istog tipa kao desni, je li izveden od člana s desne strane ili implementira član s desne strane

`lock`
Naredba koja omogućuje zaključavanje referentnog tipa da bi se više različitih niti međusobno uskladilo

`long`
Cjelobrojni označeni tip veličine 8 byte-a

`namespace`
Definira set tipova (klasa) pod zajedničkim imenom

`new`
Operator koji poziva konstruktor kreirajući novi objekt

`null`
Oznaka da niti jedan objekt nije referenciran

`object`
Osnovni tip koji nasljeđuju svi ostali tipovi

operator
Modifikator koji naznačuje preopterećenje operatora

out
Modifikator koji naznačuje da je parametar prenesen po referenci te da mora biti definiran u pozvanoj metodi

override
Modifikator metode koji naznačuje da metoda klase predefinira virtualnu metodu klase ili sučelja

params
Modifikator koji naznačuje da posljedni parametar u metodi može primiti više različitih parametara istog tipa

private
Modifikator pristupa koji naznačuje da samo taj objekt ima pravo pristupa članovima klase

protected
Modifikator pristupa koji naznačuje da samo taj objekt ili objekt izvedene klase ima pravo pristupa članovima klase

public
Modifikator pristupa koji naznačuje da sve klase odnosno objekti imaju pravo pristupa članovima klase

readonly
Modifikator koji naznačuje da vrijednost varijabli može biti dodijeljena samo jednom i to u deklaraciji ili u konstruktoru

ref
Modifikator koji naznačuje da je parametar prenesen po referenci te da mora biti definiran prije poziva metode

return
Naredba koja označava kraj izvršenja naredbi u metodi te vraća vrijednost tipa metode ili ne vraća ništa ako je metoda deklarirana kao *void*

sbyte
Označeni tip varijable veličine 1 byte koji može primiti samo cjelobrojene vrijednosti

sealed
Modifikator klase koji naznačuje da se klasa ne može naslijediti

set
Modifikator koji postavlja vrijednost svojstva (*property*)

short
Cjelobrojni označeni tip veličine 2 byte-a

`sizeof`
Operator koji vraća veličinu tipa u byte-ovima

`stackalloc`
Operator koji vraća pokazivač na određeni broj vrijednosnih tipova alociranih na *stogu* (*stack*)

`static`
Modifikator koji naznačuje da se članovima pristupa preko tipa (klase), a ne preko instance (objekta)

`string`
Predefiniran referentni tip koji sadrži niz nepromjenjivih Unicode znakova

`struct`
Vrijednosni tip podataka koji se sastoji od podataka i metoda za rad s tim podacima, odnosno koji integrira podatke i njihovu funkcionalnost

`switch`
Naredba koja omogućuje pronalaženje određene vrijednosti između više različitih. Kao parametar u *switch* naredbi može biti vrijednosni tip, ali i *string*

`this`
Varijabla koja predstavlja trenutnu (baš tu) instancu (objekt) klase ili strukture

`throw`
Naredba koja generira iznimku kad se dogodi neka nedozvoljena operacija

`true`
Oznaka za istinu kod tipa *bool*

`try`
Naredba koja omogućuje sigurno izvođenje dijelova programa te će se u slučaju nedozvoljene operacije izvršiti dio naveden u *catch* dijelu naredbe

`typeof`
Operator koji vraća tip objekta i to kao *System.Type* objekt

`uint`
Cjelobrojni neoznačeni tip veličine 4 byte-a

`ulong`
Cjelobrojni neoznačeni tip veličine 8 byte-a

`unchecked`
Operator koji onemogućava provjeru premašenja dopuštenih vrijednosti

`unsafe`
Modifikator koji naznačuje da će naredbe biti izvršene u tzv. neupravljanom (*unmanaged*) načinu rada

`ushort`
Cjelobrojni neoznačeni tip veličine 2 byte-a

`using`
Modifikator koji specificira da određeni imenik (*namespace*) može biti referenciran bez potpuno navedenog imena

`value`
Ime implicitne varijable u *set* modifikatoru svojstva

`virtual`
Modifikator koji naznačuje da metoda može biti predefinirana u izvedenoj klasi

`void`
Oznaka da metoda ne vraća nikakvu vrijednost

`volatile`
Oznaka da varijabla može biti promijenjena od strane operacijskog sustava ili druge niti

`while`
Naredba za izvršenje jedne ili više naredbi više puta sve dok je uvjet u *while* naredbi istinit

Dodatak C: Imenici (Namespaces)

Namespace
Accessibility
EnvDTE
IEHost.Execute
Microsoft.CLRAdmin
Microsoft.CSharp
Microsoft.IE
Microsoft.JScript
Microsoft.JScript.Vsa
Microsoft.Office.Core
Microsoft.VisualBasic
Microsoft.VisualBasic.Compatibility.VB6
Microsoft.VisualBasic.CompilerServices
Microsoft.VisualBasic.Vsa
Microsoft.VisualC
Microsoft.Vsa
Microsoft.Vsa.Vb.CodeDOM
Microsoft.Win32
Microsoft_VsaVb
RegCode
System
System.CodeDom
System.CodeDom.Compiler
System.Collections
System.Collections.Specialized
System.ComponentModel
System.ComponentModel.Design
System.ComponentModel.Design.Serialization
System.Configuration
System.Configuration.Assemblies
System.Configuration.Install
System.Data
System.Data.Common
System.Data.Odbc
System.Data.OleDb
System.Data.OracleClient
System.Data.SqlClient
System.Data.SqlTypes
System.Diagnostics
System.Diagnostics.Design
System.Diagnostics.SymbolStore
System.DirectoryServices
System.Drawing
System.Drawing.Design
System.Drawing.Drawing2D

Namespace
System.Drawing.Imaging
System.Drawing.Printing
System.Drawing.Text
System.EnterpriseServices
System.EnterpriseServices.CompensatingResourceManager
System.EnterpriseServices.Internal
System.Globalization
System.IO
System.IO.IsolatedStorage
System.Management
System.Management.Instrumentation
System.Messaging
System.Messaging.Design
System.Net
System.Net.Sockets
System.Reflection
System.Reflection.Emit
System.Resources
System.Runtime.CompilerServices
System.Runtime.InteropServices
System.Runtime.InteropServices.CustomMarshalers
System.Runtime.InteropServices.Expando
System.Runtime.Remoting
System.Runtime.Remoting.Activation
System.Runtime.Remoting.Channels
System.Runtime.Remoting.Channels.Http
System.Runtime.Remoting.Channels.Tcp
System.Runtime.Remoting.Contexts
System.Runtime.Remoting.Lifetime
System.Runtime.Remoting.Messaging
System.Runtime.Remoting.Metadata
System.Runtime.Remoting.Metadata.W3cXsd2001
System.Runtime.Remoting.MetadataServices
System.Runtime.Remoting.Proxies
System.Runtime.Remoting.Services
System.Runtime.Serialization
System.Runtime.Serialization.Formatters
System.Runtime.Serialization.Formatters.Binary
System.Runtime.Serialization.Formatters.Soap
System.Security
System.Security.Cryptography
System.Security.Cryptography.X509Certificates
System.Security.Cryptography.Xml
System.Security.Permissions
System.Security.Policy
System.Security.Principal

Namespace
System.ServiceProcess
System.ServiceProcess.Design
System.Text
System.Text.RegularExpressions
System.Threading
System.Timers
System.Web
System.Web.Caching
System.Web.Configuration
System.Web.Handlers
System.Web.Hosting
System.Web.Mobile
System.Web.Mail
System.Web.RegularExpressions
System.Web.Security
System.Web.Services
System.Web.Services.Configuration
System.Web.Services.Description
System.Web.Services.Discovery
System.Web.Services.Protocols
System.Web.SessionState
System.Web.UI
System.Web.UI.Design
System.Web.UI.Design.MobileControls
System.Web.UI.Design.WebControls
System.Web.UI.HtmlControls
System.Web.UI.MobileControls
System.Web.UI.WebControls
System.Web.Util
System.Windows.Forms
System.Windows.Forms.ComponentModel.Com2Interop
System.Windows.Forms.Design
System.Windows.Forms.PropertyGridInternal
System.Xml
System.Xml.Schema
System.Xml.Serialization
System.Xml.XPath
System.Xml.Xsl