Uvod u C# 3.0 i LINQ

C# je u verziji 2.0 donio nekoliko novosti. Neke od njih prošle su gotovo nezapaženo i nisu podigle puno prašine, dok su neke u potpunosti opravdale svoje uvođenje.

Slična situacija može se očekivati i s verzijom 3.0 za koju sa sigurnošću možemo ustvrditi da će zbog LINQ-a (Language INtegrated Query) kao najvažnije novosti zauzeti vrlo istaknuto mjesto u svim budućim programima napisanim sa C# 3.0.

Generički tipovi su i najavljivani i pokazali se kao najvažnija novost verzije 2.0 i nitko, naravno, ne bi trebao ostati iznenađen što su i u verziji 3.0 zadržali svoju veliku primjenu. Ono što, međutim, iznenađuje je da su neke ne baš široko prihvaćene novosti verzije 2.0 kao npr. neimenovane metode, u verziji 3.0 dobile veliku i nezamijenjivu ulogu. Iako doduše u nešto izmijenjenom obliku. Neke pak novosti kao npr. implicitno dodjeljivanje tipa varijabli tek će se učestalim korištenjem pokazati vrlo upotrebljivim.

C# je s verzijom 3.0 dobio osobine programskog jezika koji omogućuje funkcionalno programiranje, odnosno koncept koji po definiciji *naglašava vrednovanje izraza u odnosu na izvršenje naredbi*, i čije su glavne osobine:

- Odgođeno izvršenje programskog kôda
- Pisanje čistih funkcija koje nemaju popratnih pojava (engl. side effects) kao npr. ispis na zaslon ili na objekt forme
- Prenošenje metoda kao argumenata metoda
- Stvaranje neimenovanih tipova za prijenos podataka unutar programa
- Deklarativan, a ne imperativni kôd (opisivanje rezultata izvršenja umjesto načina na koji dobiti te rezultate)

Nećemo ništa više matematički teoretizirati o funkcionalnom programiranju nego ćemo na konkretnim primjerima vidjeti kako to izgleda kad se primijeni (barem dijelom) u C# programskom jeziku, a primjena funkcionalnog programiranja u C# 3.0 isključivo je u cilju povećanja učinkovitosti pisanja programa.

Većina poboljšanja verzije 3.0 ima korijene u tzv. istraživačkim i eksperimentalnim programskim jezicima kao što su Cω i F# koji su razvijeni u posebnom Microsoftovom odjelu Microsoft Research.

Ipak, kao i uvijek dosad vrijeme će pokazati, koja poboljšanja su zaista poboljšanja, a koja nisu, više ili manje.

Svi primjeri u knjizi izrađeni su programskim jezikom C#, a preduvjet potreban za čitanje knjige je poznavanje C# 2.0 jer se opisuje samo ono što je novo u verziji 3.0, a na odmet neće biti ni poznavanje SQL-a i XML-a. Za aktivno sudjelovanje u praćenju sadržaja, odnosno za prevođenje primjera koji se navode, potrebno vam je jedno od troje:

- Visual Studio 2008,
- Visual Studio 2008 Express (besplatno),
- Visual Studio 2005 ili Express + LINQ Preview.

Svi oni sadrže .NET Framework 3.5, potreban za prevođenje programa napisanih u C# 3.0, a ukoliko .NET Framework 3.5 instalirate bez instalacije gore navedenih aplikacija, programe iz knjige moći ćete prevoditi i samo uz pomoć csc prevoditelja koji dolazi s instalacijom Frameworka. Ipak, korištenje IDE (engl. Integrated development environment) okruženja uvelike olakšava unos i prevođenje primjera, iako vam ono samo neće puno pomoći u svladavanju sadržaja knjige. .NET Framework 3.0, iako broj verzije upućuje na to, nije dovoljan za prevođenje C# 3.0 programa.

Pored ovog, za izvođenje primjera koji su vezani za SQL Server (treće poglavlje, LINQ to SQL) trebat ćete imati instalirano jedno od troje:

- SQL Server 2005 Express (besplatno),
- SQL Server 2005,
- SQL Server 2008.

Primjere iz knjige možete pronaći na adresi http://public.carnet.hr/~slsaric/LINQ.ZIP.

Poglavlje 1:
C# 3.0

Ovo poglavlje obrađuje:

- Implicitno dodjeljivanje tipa
- Neimenovane tipove
- Inicijalizaciju objekata i kolekcija
- Metode proširenja
- Lambda izraze
- Stablo izraza
- Automatski implementirana svojstva
- Parcijalne metode

1.1 Implicitno dodjeljivanje tipa (engl. Implicitly Typed Local Variables)

Još od programskog jezika C jedini je način deklariranja varijable bio navođenje tipa uz identifikator.

```
string s = “hello, world”;
int i = 0;
int j = i;
double d = 3.14;
Dog Hund = new Dog(“Oštar pas”);
List<double> constants = new List<double>();
```

Pritom se naravno trebalo paziti da tip bude odgovarajući s obzirom na dodijeljenu vrijednost, ali i očekivanu. Iako to uglavnom nije predstavljalo problem, C# 3.0 uvodi ključnu riječ *var* koja razvijatelje programa oslobađa potrebe dodjeljivanja tipa varijabli. Tako bi, koristeći ključnu riječ *var*, gore navedeni primjeri izgledali ovako:

```
var s = “hello, world”;
var i = 0;
var j = i;
var d = 3.14;
var Hund = new Dog(“Oštar pas”);
var constants = new List<double>();
```

Ipak nema razloga za strah ako pripadate svima onima koji ne vole tipsku neodređenost varijabli. C# je zadržao strogu tipsku određenost i ta vrlo cijenjena osobina ostaje prisutna u punoj mjeri. Ako pokušate varijabli *s* dodijeliti cijelobrojnu vrijednost ili npr. varijablu *Hund* definirati kao objekt tipa klase *Cat*:

```
s = 10;
Hund = new Cat(“Tom”);
```

ništa od toga nećete uspjeti i prevoditelj će javiti grešku. A upravo je prevoditelj zaslužan za automatsko dodjeljivanje tipa varijabli i on je taj koji pretvara tip *var* u odgovarajući tip. Zato i postoje neka ograničenja pri upotrebi tipa *var*:

- kod deklaracije je potrebno definirati varijablu
  `var counter;  // Greška`
- varijabla deklarirana kao *var* ne smije imati vrijednost *null*
  `var str = null;  // Greška`
- varijabla deklarirana kao *var* ne može biti inicijalizirana kolekcija
  `var arr = {1, 3, 5, 7, 9, 11};  // Greška`
- nije dozvoljeno izvesti višestruku deklaraciju ključnom riječju *var*
  `var x = 10, y = 11;  // Greška`
- varijabla deklarirana kao *var* ne može biti članica klase
  `private var x = 10;  // Greška`

- ključna riječ *var* ne može se koristiti za deklaraciju konstanti
```
const var e = 2.718;  // Greška
```

Za prva je dva slučaja jasno da prevoditelj ne može zaključiti kakvog tipa treba biti varijabla jer joj nije dodijeljena vrijednost na osnovu koje bi to mogao zaključiti, dok vrijednost *null* može imati bilo koji referentni, a od verzije 2.0 i vrijednosni tip.

Provjera dodjeljivanja vrijednosti null *var* varijabli odvija se u vremenu prevođenja što znači da je dozvoljeno napisati:

```
var str = “It is not null”;
str = null;
```

jer je to isto kao da smo napisali:

```
string str = “It is not null”;
str = null;
```

u što se možemo i uvjeriti ako u prvom primjeru izvedemo sljedeću naredbu (naredbu treba izvršiti prije dodjele vrijednosti *null*, inače će se generirati iznimka NullReferenceException):

```
Console.WriteLine(str.GetType());
```

koja će ispisati:
System.String

U trećem od navedenih nedozvoljenih slučajeva upotrebe ključne riječi *var*, potrebno je definirati objekt ključnom riječi *new*

```
var arr = new int[] {1, 3, 5, 7, 9, 11};
```

što je isto kao da smo napravili:

```
int [] arr = new int[] {1, 3, 5, 7, 9, 11};
```

Kako je riječ o zaključivanju (engl. infer) prevoditelja kojim tipom deklarirati varijablu, sasvim je očekivano da se *var* može koristiti i u *foreach* petljama:

```
int [] TenNumbers = new int[] {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
foreach(var i in TenNumbers)
    Console.WriteLine(i);
```

ili da ostanemo dosljedni upotrebi ključne riječi *var*, mogli smo to i ovako napraviti:

```
var TenNumbers = new int[] {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
foreach(var i in TenNumbers)
    Console.WriteLine(i);
```

Očito je, dakle, da nema nikakve razlike u deklaraciji varijable na standardan način ili korištenjem ključne riječi *var* (naravno uz navedena ograničenja koja su prisutna isključivo zbog nemogućnosti prevoditelja da zaključi tip zbog dvosmislenosti).

Zato je moguće varijabli *var* dodijeliti i povratnu vrijednost iz funkcije što može biti od koristi ako ne znamo kako je funkcija deklarirana, odnosno, kakav tip vraća (a nismo raspoloženi tražiti po *Helpu* i nije nam na raspolaganju Visual Studio koji takve informacije prikazuje u hodu). Kasnije ćemo kod neimenovanih klasa vidjeti da postoje situacije kada je jedino rješenje deklaracija varijable ključnom riječi *var*.

Evo prvog primjera:

▶ Primjer 1.1

```
using System;

public class Test
{
      public static void Main()
      {
            var a = Math.Sin(45);
            var b = Math.Cos(45);
            var c = a * a + b * b;
            Console.WriteLine("c = {0}, tip {1}", c, c.GetType());
      }
}
```

Ispis:

c = 1, tip System.Double

Ako nismo sigurni je li funkcije *Sin* i *Cos* vraćaju tip *float* ili *double*, koristit ćemo ključnu riječ *var* i u mnogo slučaja ne moramo čak niti znati u koji tip je prevoditelj pretvorio *var* varijablu. Ali poželimo li u gornjem primjeru napraviti:

```
float f = c;
```

prevoditelj će javiti grešku jer se tip *double* ne može implicitno pretvoriti u tip *float*, što samo potvrđuje da je C# ostao strogo tipski određen jezik.

Potvrdu da je to tako možemo dobiti i ako pozovemo MSIL Disassembler (ildasm.exe) nad izvršnom PE (engl. Portable Executive) datotekom i pogledamo što se nalazi u generiranom IL (engl. Intermediate Language) kôdu:

```
.method public hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       66 (0x42)
  .maxstack  3
```

```
  .locals init ([0] float64 a,
           [1] float64 b,
           [2] float64 c)
      . . .
```

Vidimo da su sve tri varijable, *a*, *b* i *c*, u IL kôdu deklarirane kao *double* jer je varijabla tipa *double* u IL kôdu deklarirana kao *float64*. U sve to se možete uvjeriti i ako u gornjem primjeru definirate još jednu *double* varijablu, ovaj put bez upotrebe ključne riječi *var*.

```
double d = 20;
```

i ako biste ponovo pokrenuli ildasm.exe, sve bi četiri varijable bile deklarirane kao *float64*.

Postoji još jedan slučaj kada nije dozvoljeno koristiti ključnu riječ *var*, a to je ako se u dosegu nalazi tip, odnosno klasa s imenom *var*. Pretpostavimo da imamo sljedeću definiranu klasu:

```
class var
{
    private int x;
    private int y;

    public var(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

Iako nije velika vjerojatnost da će netko u verziji 3.0 ovako nazvati svoju klasu, ovo može biti ranije definirana klasa koju još uvijek koristimo (ipak, ni prije nije bilo baš uobičajeno ovako nazvati klasu, ako ništa drugo ne udovoljava preporukama o korištenju notacije Pascal). Ako bismo sad pokušali deklarirati i definirati varijablu:

```
var x = 5;
```

program se neće prevesti jer će prevoditelj ključnu riječ *var* interpretirati kao naziv klase što opet znači da bi sljedeće deklaracija i definicija bile ispravne:

```
var x = new var(5, 10);
```

1.2 Neimenovani tipovi (engl. Anonymous Types)

Jeste li i koliko ste puta bili u situaciji da vam je trebalo neke podatke predočiti u obliku nekog objekta i onda vas je čekalo ono neminovno definiranje klase za takav jedan objekt. Sad je i tome došao kraj s neimenovanim tipovima (engl. anonymous types).

Kao i kod varijable *var*, i ovdje je prevoditelj taj koji izvodi skriveni posao, a to je stvaranje klase zajedno s pripadajućim varijablama članicama i svojstvima.

U knjizi C# u izlaganju o klasama stvorili smo klasu TV koja je imala nekoliko varijabli članica. S obzirom da su sve varijable članice deklarirane modifikatorom pristupa *private*, za svaku bi trebalo stvoriti svojstvo za pristup tim varijablama. Koristeći neimenovane tipove sad sve stane u jedan red:

```
var TV = new {Naziv = “Sharp”, Model = “C30”, Cijena = 6999.99};
```

Ako ste odmah pomislili da negdje moraju biti sve deklaracije i definicije varijabli i svojstava, pogodili ste, jer je opet prevoditelj generator kôda stvorenog u pozadini. A prevoditelj će za neimenovani tip stvoriti klasu koja će otprilike ovako izgledati:

```
class __Anonymous1
{
   private string naziv;
   private string model;
   private double cijena;

   public string Naziv
   {
      get { return naziv; }
      set { naziv = value; }
   }

   public string Model
   {
      get { return model; }
      set { model = value; }
   }

   public double Cijena
   {
      get { return cijena; }
      set { cijena = value; }
   }
}
```

dok će instanciranje objekta biti izvedeno s:

```
__Anonymous1 TV = new __Anonymous1();
TV.Naziv = “Sharp”;
TV.Model = “C30”;
TV.Cijena = 6999.99;
```

U svemu ovome nema ništa posebno što već ne znamo. Razlika je u tome što jedino prevoditelj može stvoriti objekt neimenovane klase navođenjem njezinog imena te dodijeliti vrijednosti varijablama članicama, odnosno, svi mi ostali to možemo jedino uz pomoć ključne riječi *var*. Također, možemo primijetiti da je objekt *TV* deklariran kao *var* što je jedino moguće kod neimenovanih klasa jer u trenutku instanciranja objekta ne znamo njegov

tip, odnosno klasu, a ako i znamo da se zove *__AnonymousX* opet nam to ništa ne pomaže, jer ne možemo instancirati objekt takve klase pozivom njezinog konstruktora.

Klasa *__Anonymous1* predefinira metodu *ToString*:
i ako pozovemo:

```
TV.ToString();
```

kao rezultat će se vratiti znakovni niz ovakvog sadržaja:
{ **Naziv=Sharp, Model=C30, Cijena=6999,99** }

▶ Primjer 1.2

Instanciranjem dvaju ili više objekata za koje po navedenim argumentima prevoditelj zaključi da mogu pripadati jednoj te istoj klasi, kreirat će se samo jedna klasa. Budući da su takvi objekti istog tipa između njih se može provesti implicitno razmjenjivanje vrijednosti:

```
using System;

public class Test
{
      public static void Main()
      {
            var TV1 = new {Naziv = “Sharp”, Model = “C30”,
                           Cijena = 6999.99};
            var TV2 = new {Naziv = “Sharp”, Model = “C40”,
                           Cijena = 7999.99};
            TV1 = TV2;
            Console.WriteLine(TV1.ToString());
      }
}
```

Ispis:

{ **Naziv=Sharp, Model=C40, Cijena=7999,99** }

Nažalost, prevoditelj nije u stanju poistovjetiti klase ako im zamijenimo redoslijed navedenih svojstava.

```
var TV3 = new {Model = “C50”, Cijena = 8999.99, Naziv = “Sharp”};
TV1 = TV3;   // GREŠKA
```

Iako i objekt *TV3* sadrži članove istog imena i tipa, zbog različitog redoslijeda njihovog navođenja prevoditelj će za njega stvoriti novu neimenovanu klasu, te neće biti moguće napraviti implicitnu konverziju iz jednog tipa neimenovane klase u drugi.

Lako je naći nedostatke neimenovanoj klasi kao npr.:

- ne može se instancirati
- ne mogu joj se predefinirati metode kao npr. *ToString*
- ne može se naslijediti
- ne može se dodijeliti vrijednost njezinom članu pozivom *set* svojstva

Zato ako vam sve ovo u vezi neimenovanih klasa i ne izgleda kao nešto veliko i osobito bitno, sigurno ćete biti razuvjereni u nastavku opisa verzije 3.0 kada budemo govorili o LINQ-u koji u velikoj mjeri ne bi mogao biti realiziran bez neimenovanih klasa.

1.3 Inicijalizacija objekata i kolekcija (engl. Object and Collection Initializations)

Korištenje neimenovanih klasa ima i druga ograničenja jer npr. ne možemo vratiti objekt takve klase kao povratnu vrijednost iz metode. Jasno je i zašto je to tako, ako ne znamo naziv klase, odnosno ako ga i znamo, nije nam dozvoljeno koristiti ga, ne možemo deklarirati takvu metodu.

Za taj i takve slučajeve, ako npr. želimo vratiti kao povratnu vrijednost objekt stvoren u hodu možemo koristiti C# 3.0 novost u vidu inicijalizacije objekata i kolekcija. Sljedeći će primjer koristeći takvu inicijalizaciju kao povratnu vrijednost vratiti objekt generiran bez poziva konstruktora i postavljanja svojstava.

▶ Primjer 1.3

```
using System;

public class MobilePhone
{
   private string name;
   private string model;
   private int g;

   public string Name
   {
      get { return name; }
```

```
      set { name = value; }
   }

   public string Model
   {
      get { return model; }
      set { model = value; }
   }

   public int G
   {
      get { return g; }
      set { g = value; }
   }
}
public class Test
{
      public static MobilePhone Upgrade(MobilePhone m)
      {
            return new MobilePhone{Name = m.Name,
                  Model = "M700", G = m.G + 1};
      }

      public static void Main()
      {
            MobilePhone se = Upgrade(new MobilePhone
                        {Name = "SonyEricsson",
                         Model = "M600", G = 3});
            Console.WriteLine(se.Name + " " + se.Model
                         + " " + se.G + "G");
      }
}
```

Ispis:

SonyEricsson M700 4G

Primjer ne samo što vraća objekt inicijaliziran bez poziva konstruktora i svojstava nego takav objekt i prosljeđuje metodi *Upgrade* koja mu, prije nego što ga vrati kao povratnu vrijednost, poveća vrijednost varijable *g* za 1.

Sve se radi u jednoj naredbi: deklaracija i instanciranje objekta, njegova inicijalizacija vrijednostima i povrat iz metode. Iako možemo primijetiti veliku sličnost s neimenovanim klasama, nismo ih u ovakvom primjeru mogli koristiti jer ne bismo mogli propisno deklarirati metodu.

Pri ovakvom načinu instanciranja objekta nismo obvezni navesti vrijednosti svih varijabli pozivom njihovih svojstava, ali onda će svim nenavedenim varijablama članicama klase biti dodijeljene podrazumijevane vrijednosti. Ako ne bismo naveli svojstvo *G*, varijabla *g* kojoj se pristupa preko tog svojstva inicijalno bi dobila vrijednost 0, pa bi metoda *Upgrade* tu 0 digla na vrijednost 1. Ipak, uvijek imate na raspolaganju klasični način inicijalizacije objekta prosljeđivanjem argumenata konstruktoru u kojem točno možete odrediti koje članice klase moraju biti eksplicitno inicijalizirane.

Ovo je bio primjer u kojem nismo definirali niti jedan konstruktor pa klasa koristi predefinirani konstruktor bez argumenata. U slučaju da je klasa *MobilePhone* imala definiran npr. konstruktor:

```
public MobilePhone(string n)
{
      name = n;
}
```

onda bismo ga morali i navesti kod inicijalizacije objekta:

```
MobilePhone se = Upgrade(new MobilePhone("SonyEricsson")
                              { Model = "M600", G = 3 });
```

a kod povratne vrijednosti iz metode *Upgrade*:

```
return new MobilePhone(m.Name){Model = "M700", G = m.G + 1};
```

Inicijalizacija kolekcija

Što se tiče inicijalizacije kolekcijskih klasa sve je vrlo slično i ako imamo generičku klasu *List<MobilePhone>* onda bismo to ovako napravili:

```
List<MobilePhone> smartphones = new List<MobilePhone>
{
      new MobilePhone {Name = "SonyEricsson", Model = "M600", G = 3},
      new MobilePhone {Name = "SonyEricsson", Model = "P990", G = 3},
      new MobilePhone {Name = "SonyEricsson", Model = "M700", G = 4}
};
```

Riječ je o najobičnijoj kolekciji koja se ni po čemu ne razlikuje od ostalih kolekcija što znači da joj možemo pridodati još jedan član na standardan način:

```
MobilePhone sams = new MobilePhone();
sams.Name = "Samsung";
sams.Model = "ZMXHW564";
sams.G = 3;
smartphones.Add(sams);
```

a kako je riječ o nizu objekata s 0 indeksacijom,

```
Console.WriteLine(smartphones[3].Model);
```

će ispisati:
ZMXHW564

Kreiranje objekta *sams* moglo je uz pomoć novosti C# 3.0 u vidu inicijalizacije objekta biti izvedeno i u jednoj naredbi:

```
smartphones.Add(new MobilePhone {
      Name = “Samsung”, Model = “ZMXHW564”, G = 3 });
```

1.4 Metode proširenja (engl. Extension Methods)

Ovo je poboljšanje verzije 2.0 u vidu proširenja klase, sučelja i strukture dodavanjem novih metoda toj istoj klasi, sučelju ili strukturi. Tu prije svega mislimo na predefinirane klase i sučelja iz .NET Frameworka, a i ostale koje ćemo eventualno koristiti, a nije nam dostupan njihov izvorni kod.

Iako isti učinak možemo postići i nasljeđivanjem klasa, ovaj način unosi određena poboljšanja jer je jednostavniji, a i omogućuje dodavanje novih metoda zapečaćenim klasama kao što je npr. klasa System.String.

Metoda proširenja implementira se kao statička metoda u statičkoj klasi i doseg vidljivosti klase ujedno je i doseg metode. Metodi se kao argument navede tip na koji se takva metoda primjenjuje dodajući ispred argumenta ključnu riječ *this*.

```
public static class MyExtMethods
{
    public static int PlusPlus(this int i)
    {
        return i+1;
    }
}
```

Ovo je primjer metode proširenja *PlusPlus* koja se može primijeniti na tip *int* i čiji će poziv povećati vrijednost varijabli za 1. Kad je metoda ovako određena, ona se može pozvati kao bilo koja druga metoda strukture System.Int32, odnosno metoda *PlusPlus* se ponaša kao da je i sama dio strukture System.Int32.

```
int ThisYear = 2007;
int NextYear = ThisYear.PlusPlus();
```

i u varijabli *NextYear* će naravno biti vrijednost 2008.

▶ Primjer 1.4

U jednoj statičkoj klasi može se definirati i više metoda proširenja. One se mogu odnositi na različite tipove, odnosno klase, a mogu biti i preopterećene:

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

public static class MyExtMethods
{
        public static int PlusPlus(this int i)
        {
            return i + 1;
        }

        public static int MinusMinus(this int i)
        {
            return i - 1;
        }

        public static string PlusPlus(this string s)
        {
            int i = 0;
            StringBuilder sb = new StringBuilder();
            while(i++ < s.Length)
                sb.Append((char)(s[i - 1] + 1));

            return sb.ToString();
        }

        public static string MinusMinus(this string s)
        {
            int i = 0;
            StringBuilder sb = new StringBuilder();
            while(i++ < s.Length)
                sb.Append((char)(s[i - 1] - 1));

            return sb.ToString();
```

```
        }
}

public class Test
{
      public static void Main()
      {
            int ThisYear = 2007;
            int NextYear = ThisYear.PlusPlus();
            int LastYear = ThisYear.MinusMinus();
            Console.WriteLine(NextYear);
            Console.WriteLine(LastYear);

            string s = “HAL”;
            Console.WriteLine(s.PlusPlus());
            s = “NJDSPTPGU”;
            Console.WriteLine(s.MinusMinus());
       }
}
```

Ispis:

2008
2006
IBM
MICROSOFT

Dvije su metode preopterećene *PlusPlus* i *MinusMinus*. Odabir metode *PlusPlus* koja će se pozvati ovisi o tipu varijable za koju je pozvana metoda. Ako je riječ o cijelom broju (engl. integer), poziva se

```
public static int PlusPlus(this int i)
```

koja će povećati vrijednost tog cijelog broja za jedan, a ako je riječ o znakovnom nizu (engl. string), poziva se

```
public static string PlusPlus(this string s)
```

koja će svaki znak iz znakovnog niza povećati za ASCII vrijednost 1.

Isto je i s preopterećenim metodama *MinusMinus* samo što one umanjuju vrijednosti za 1. Kod rada sa znakovnim nizovima zbog nepromjenjivosti (engl. immutable) objekta tipa *string* koristimo klasu *StringBuilder*.

Posebnost metoda proširenja je i u tome što, iako su deklarirane kao statičke, mogu biti pozvane i od klase i od objekta:

```
int x = 5;
int y = x.PlusPlus();    // Poziv od objekta
int z = MyExtMethods.PlusPlus(x);    // Poziv od klase
```

Varijabli *y* je vrijednost dodijeljena pozivom metode proširenja od strane objekta *x*, dok je varijabla *z* dobila vrijednost pozivom iste te metode, ali ovaj put od strane klase *MyExtMethods*, klase kojoj metoda i pripada. Jasno je da će u oba slučaja dodijeljena vrijednost biti ista, odnosno vrijednost 6.

Metode proširenja mogu se koristiti i izravno nad vrijednostima:

```
string SuperComputer = "IBM".MinusMinus() + 8999.PlusPlus();
```

i kao rezultat u varijabli *SuperComputer* bit će vrijednost:
HAL9000

Primjena metoda proširenja na generičke tipove

Nije neočekivano da metode proširenja možemo koristiti i na generičkim tipovima. U sljedećem primjeru kreirat ćemo klasu *Book* koja će imati 3 varijable, 2 znakovna niza i jedan cijeli broj *title*, *pages* i *isbn*, te ćemo implementirati metodu proširenja *ToCSV* nad generičkom klasom *List*. Metoda će imati zadatak pretvoriti članove kolekcije u znakovni niz u CSV formatu (vrijednosti odijeljene znakom kojeg ćemo doznati svojstvom *ListSeparator*).

▶ Primjer 1.5

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

public class Book
{
    private string title;
    private int pages;
    private string isbn;
    public Book(string title, int pages, string isbn)
    {
        this.title = title;
        this.pages = pages;
        this.isbn = isbn;
    }
```

```
    public override string ToString()
    {
        return title;
    }
}

public static class MyExtMethods
{
    public static string ToCSV<T>(this List<T> arr)
    {
        StringBuilder sb = new StringBuilder();
        for(int i = 0;i<arr.Count;i++)
            sb.Append(arr[i].ToString() +
                  System.Globalization.CultureInfo.
                  CurrentCulture.TextInfo.ListSeparator);

        return sb.ToString();
    }
}

public class Test
{
      public static void Main()
      {
            Book CSharp = new Book("C# na lak način", 300,
                                   "1000000000");
            Book CPlusPlus = new Book("C++ za 24 sata", 300,
                                   "2000000000");
            Book Java = new Book("Java bez po muke", 300,
                                   "3000000000");

            List<Book> arr = new List<Book>();
            arr.Add(CSharp);
            arr.Add(CPlusPlus);
            arr.Add(Java);

            Console.WriteLine(arr.ToCSV<Book>());
      }
}
```