Mario Bošnjak
0036371938

Seminar iz predmeta
Ergonomija računalne i programske opreme

Programski jezik C#

Sadržaj

SADRŽAJ **1**

UVOD **3**

STRUKTURA PROGRAMA **4**

HELLO WORLD 4
Komentari *4*
Main metoda *4*
Ulaz i izlaz *5*
Imenik *5*
KOMPAJLIRANJE I POKRETANJE 5

TIPOVI PODATAKA **6**

VRIJEDNOSNI TIPOVI 6
Integralni tipovi *6*
Tipovi s pomičnim zarezom *7*
Decimal *7*
Bool *7*
Struct *8*
Pobrojani tip *8*
REFERENTNI TIPOVI 9
Boxing i unboxing *9*

NIZOVI **11**

JEDNODIMENZIONALNI NIZ 11
Inicijalizacija niza *11*
Nizovi vrijednosnih i referentnih tipova *11*
VIŠEDIMENZIONALNI NIZ 12
NAZUBLJENI NIZ 12

STRINGOVI **14**

RAD SA STRINGOVIMA 14
Simbol @ *14*
ToString() *15*
Pristup pojedinim znakovima *15*
Velika i mala slova *15*
Usporedba *16*
Podjela stringa na podstringove *16*
STRINGBUILDER 17

KLASE **18**

POLIMORFIZAM 18
APSTRAKTNE I SEALED KLASE 21
SUČELJA 22
Eksplicitna implementacija sučelja *23*
ČLANOVI KLASE 24
Polja *24*
Konstante *25*
Ugniježđeni tip *25*
METODE 26
Parametri vrijednosnog tipa *27*
Parametri referentnog tipa *27*
KONSTRUKTORI 29

SVOJSTVA **31**

KORIŠTENJE SVOJSTAVA 31
Get pristupnik *32*

Set pristupnik *33*
ASIMETRIČNA SVOJSTVA 33
DELEGATI **35**
KOVARIJANCA I KONTRAVARIJANCA 36
DOGAĐAJI **38**
ZAKLJUČAK **39**
LITERATURA **40**

Uvod

Ovaj seminar je namijenjen osobama koje već imaju iskustva u programiranju, ponajprije sa C, C++ ili Java programskim jezicima. Ako ste programirali u nekom drugom jeziku (Visual Basic ili Pascal), možda ćete trebati malo više vremena da se naviknete na sintaksu, ali još uvijek bi trebali uz ovaj tutorijal naučiti C# jer se objašnjavaju stvari od početka.

C# ima za prethodnike C i C++ od kojih je uzeo sve stvari koje su bile dobre i koje nisu zahtijevale poboljšanja. Izrazi, naredbe i skoro cijela sintaksa, što čini većinu tipičnih programa, je ostala nepromijenjena. Zbog toga se C i C++ programeri kad počnu učiti C# osjećaju "kao doma".

Učinjen je veliki napredak i dodana su mnoga poboljšanja i nadogradnje koje ne bi imalo smisla sve nabrajati u uvodu. Saznat ćete velik broj ako pročitate cijeli tutorijal do kraja. Ipak za početak, da nabrojimo neke "sitne" **nadogradnje** koje su uklonile dosta česte i vremenski zahtjevne pogreške u C i C++ programima:

- Varijable se moraju inicijalizirati prije nego što se počnu koristiti - nema više grešaka da se koristila neinicijalizirana varijabla.
- Naredbe `if` i `while` zahtijevaju Boolean vrijednosti tako da programeri više ne mogu greškom koristiti operator `=` umjesto operatora `==.`
- Naredba `switch` više ne podržava prolazak na sljedeći `case` blok bez eksplicitnog navođenja, što je prije stvaralo pogreške ako programer zaboravi staviti `break` naredbu na kraju bloka.

Od bitnijih **novosti** možemo navesti sljedeće:

- Automatsko upravljanje memorijom oslobađa programere napornog posla oslobađanja memorije. "Viseći pointeri" i neoslobađanje memorije je jedan od najčešćih grešaka u C++ programima koji se u većini oslanjaju na rad sa pointerima. Pri tome se nije izgubilo na funkcionalnosti jer se još uvijek može raditi sa pointerima i adresama objekata u rijetkim slučajevima kad baš moramo ručno upravljati memorijom.
- Svi podaci su objekti! Kroz inovativne koncepte boxinga i unboxinga, premoštena je razlika između value i reference tipova, tako da se svi podaci mogu obrađivati kao objekti.
- U klasama se mogu definirati svojstva (properties), koja omogućavaju kontrolirani i intuitivniji pristup podacima koji se nalaze u objektu nego preko getter i setter metoda. Ono što se u C++-u možda moralo napisati `o.SetValue(o.GetValue() + 1)`, sada se može napisati puno jednostavnije kao `o.Value++.`
- C# podržava i atribute koji omogućavaju definiranje i korištenje deklarativnih informacija o komponentama. Mogućnost definiranja nove deklarativne informacije je moćan alat koji otvara programerima nove, i prije teško izvedive, mogućnosti.

Tutorijal je zamišljen da, osim što objasni osnovnu sintaksu, ponudi i odgovor na pitanje zašto se koristiti određena mogućnost. Ako do sada niste koristili sučelja ili delegate, puko navođenje sintakse vas neće nagovoriti da koristite mogućnosti koje vam oni nude. Zbog toga se u svakom poglavlju nalazi i kôd koji objašnjava koje probleme u programima ta mogućnost pokušava riješiti.

Struktura programa

U ovom poglavlju ćemo pogledati kako izgleda osnovna struktura C# programa. Započet ćemo sa standardnim "Hello, World!" primjerom, kroz koji ćemo opisati osnovne elemente od kojih se sastoji jedan normalan program.

Hello World

Skoro svaka knjiga i tutorijal o nekom programskom jeziku započinje sa najjednostavnijim primjerom koji samo ispisuje neki tekst na ekran, pa ni ovaj tutorijal neće biti iznimka.

Sljedeći programski odsječak prikazuje C# verziju Hello World programa:

```
using System;
// A "Hello World!" program in C#
namespace HelloWorld
{
    class Hello
    {
        static void Main()
        {
            System.Console.WriteLine("Hello World!");
        }
    }
}
```

Pogledajmo sada dijelove od kojih se sastoji ovaj program.

Komentari

Drugi redak u programu je komentar:

```
// A "Hello World!" program in C#
```

Dvije kose crte `//` pretvaraju ostatak reda u komentar. Neki blok možemo pretvoriti u komentar i tako da ga stavimo između `/*` i `*/` znakova, kao na primjer:

```
/* A "Hello World!" program in C#.
This program displays the string "Hello World!" on the screen. */
```

Main metoda

Svaki C# program mora sadržavati **Main** metodu u kojoj se program počinje izvršavati i sa kojom program završava. U Main metodi se stvaraju objekti i izvršavaju njihove metode.

Main metoda je `static` metoda koja se nalazi unutar neke klase ili strukture. U prethodnom primjeru, nalazi se unutar klase Hello. Metoda može za povratni tip imati `int` ili `void`, a može primiti i ulazne parametre:

```
static int Main(string[] args)
{
    //...
```

```
    return 0;
}
```

Parametar koji se predaje ovoj Main metodi je niz stringova koje predstavlja argumente koji su upisani prilikom poziva programa u komandnoj liniji. Za razliku od C++-a, ovaj niz ne uključuje ime exe datoteke.

Ulaz i izlaz

C# programi uglavnom koriste ulazno/izlazne servise koji su ponuđeni u .NET Framework biblioteci klasa. Izraz `System.Console.WriteLine("Hello World!");` koristi WriteLine metodu, jednu od izlaznih metoda u `Console` klase. Ona prikazuje string koji smo zadali kao parametar na standardni izlazni tok. Obzirom da smo na početku programa napisali direktivu `using System;` naša naredba za ispis teksta je mogla izgledati i ovako:

```
Console.WriteLine("Hello World!");
```

Sve klase unutar System imenika možemo koristiti bez prefiksa zahvaljujući **using** naredbi.

Imenik

Za one koji se nisu s njima prije susretali, imenik, prostor imena ili `namespace` je jednostavno sredstvo pomoću kojeg semantički grupiramo elemente (klase i druge imenike) i time rješavamo problem kolizije imena. Dvije klase mogu imati isto ime i možemo ih koristiti u svom programu ako se te klase nalaze u različitim imenicima. Razlikujemo ih naravno po prefiksu, tj. punom imenu koji uključuje i imenik u kojem se klasa nalazi. Imenici mogu sadržavati klase, strukture, sučelja, enumeracije, delegate, te druge imenike. Sa .NET Frameworkom dobivate par tisuća gotovih klasa i da nisu podijeljene i organizirane po imenicima, teško bi se njima koristili.

Kompajliranje i pokretanje

Ovaj, Hello World program, a i sve ostale primjere, možete isprobati na 2 načina. Jedan je pomoću Visual Studia ili nekog drugog razvojnog alata, a drugi je pomoću komandne linije i kompajlera koji dolazi zajedno sa .NET Framework SDK-om koji je besplatan.

Da biste kompajlirali i pokrenuli program iz komandne linije, potrebno je par koraka:

- Programski kod napišite u nekom uređivaču teksta (Notepad ili neki napredniji) i snimite u datoteku sa npr. imenom `Hello.cs`. C# datoteke sa izvornim kodom imaju nastavak `.cs`.
- Kompajler je program pod imenom `csc.exe`, pa bi naredba izgledala: `csc Hello.cs`
- Da pokrenete program, samo utipkajte naredbu: `Hello`

Tipovi podataka

C# spada u *strongly typed* jezike, tj. sve varijable i objekti moraju imati deklariran tip. Varijablama možemo zadati da imaju neki od ugrađenih tipova, poput `int` ili `char` tipa, ili neki korisnički-definirani tip poput klase ili sučelja. Drugi način podjele je na:

- vrijednosne tipove (*value type*)
- referentne tipove (*reference type*)

Vrijednosni tipovi

Vrijednosni tipovi se dijele su dvije glavne kategorije:

- Strukture
- Pobrojani tip

Strukture je dalje dijele na:

- Numeričke tipove
  - Integralne tipove
  - Tipove sa pomičnim zarezom
  - Decimal
- Bool
- Korisnički definirane strukture

Varijable koje imaju vrijednosni tip direktno sadržavaju vrijednost. Kad pridružujemo jednu varijablu vrijednosnog tipa drugoj varijabli, vrijednost se kopira. Ovo je različito od slučaja kad su varijable referentnog tipa kada se kopira samo referenca, ali ne i sami objekt.

Za razliku od referentnih tipova, tip nije moguće proširiti (naslijediti), ali strukture još uvijek mogu implementirati sučelja. Isto tako, za razliku od referentnih tipova, vrijednosni tipovi ne mogu sadržavati `null` kao vrijednost, ali sa dolaskom novog frameworka 2.0, dodan je *nullable type* koji nam omogućuje da vrijednosnom tipu pridružimo null.

Integralni tipovi

Sljedeća tablica prikazuje veličinu i raspon integralnih tipova:

Tip	Raspon	Veličina
sbyte	-128 to 127	Signed 8-bit integer
byte	0 to 255	Unsigned 8-bit integer
char	U+0000 to U+ffff	Unicode 16-bit character
short	-32,768 to 32,767	Signed 16-bit integer
ushort	0 to 65,535	Unsigned 16-bit integer
int	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer
uint	0 to 4,294,967,295	Unsigned 32-bit integer

long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer
ulong	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer

Tipovi s pomičnim zarezom

Sljedeća tablica prikazuje preciznost i procijenu raspona tipova s pomičnim zarezom.

Tip	Raspon	Preciznost
float	±1.5e−45 to ±3.4e38	7 digits
double	±5.0e−324 to ±1.7e308	15-16 digits

Decimal

Decimal je 128-bitni vrijednosni tip. U usporedbi sa tipom s pomičnim zarezom, decimalni tip ima veću preciznost i manji raspon, što ga čini pogodnijim za financijske i novčane proračune. Procijena raspona za decimalni tip se nalazi u sljedećoj tablici.

Tip	Raspon	Preciznost
decimal	±1.0 × 10e−28 to ±7.9 × 10e28	28-29 significant digits

Bool

Ključna riječ **`bool`** je alias za **`System.Boolean`** tip. Koristi se za deklariranje varijabli koje mogu pohranjivati Boolean vrijednosti, **`true`** i **`false`**. Nekoj bool varijabli se mogu pridružiti i izrazi koji kao rezultat vraćaju bool vrijednost, kao u sljedećem primjeru:

```
// keyword_bool.cs
using System;
public class MyClass
{
    static void Main()
    {
        bool i = true;
        char c = '0';
        Console.WriteLine(i);
        i = false;
        Console.WriteLine(i);

        bool Alphabetic = (c > 64 && c < 123);
        Console.WriteLine(Alphabetic);
    }
}
```

Kao rezultat se dobiju tri retka u kojima piše True, False, False.

Struct

Korisnički definirane strukture se koriste za enkapsulaciju manje grupe povezanih varijabli, poput koordinata pravokutnika ili karakteristika nekog predmeta iz skladišta. Sljedeći primjer prikazuje deklaraciju jednostavne strukture:

```
public struct Book
{
    public decimal price;
    public string title;
    public string author;
}
```

Strukture mogu sadržavati i konstruktore, konstante, atribute (*fields*), metode, svojstva, indeksere, operatore, događaje i ugniježđene tipove, premda, ako vam je potrebno više ovih elemenata, trebali biste razmislite da stvorite klasu umjesto strukture.

Pobrojani tip

Pobrojani tip ili enumeracija se stvara pomoću **`enum`** ključne riječi, a koristi se za definiranje seta konstanti. Svaka enumeracija ima ispod sebe neki integralni tip (osim **`char`** tipa). Defaultni tip je **`int`**. Po defaultu, prva konstanta ima vrijednost 0, a sve ostale po 1 više od prethodnog, pa tako u sljedećem primjeru

```
enum Days {Sat, Sun, Mon, Tue, Wed, Thu, Fri};
```

konstanta Sat ima vrijednost 0, Sun ima 1, Mon ima 2, itd. To možemo i promijeniti tako da konstantama eksplicitno zadamo vrijednost. Ona sljedeća konstanta, koja nema zadanu eksplicitnu vrijednost, ima vrijednost za 1 veću od prethodne konstante. U sljedećem primjeru

```
enum Days {Sat = 1, Sun, Mon, Tue = 2, Wed, Thu, Fri};
```

konstanta Sat ima vrijednost 1, Sun ima 2, Mon ima 3, Tue ima 2, Wed ima 3, itd.

Kod ovog zadnjeg primjera za enumeracije vidimo kako promijeniti tip koji se nalazi ispod konstanti u enumeraciji. Stavili smo da su tipa **`long`**. Primijetite također da, iako su tipa long, konstante iz enumeracije je potrebno eksplicitno pretvoriti u long tip prilikom pridruživanja long varijabli.

```
// keyword_enum2.cs
// Using long enumerators
using System;
public class EnumTest
{
    enum Range :long {Max = 2147483648L, Min = 255L};
    static void Main()
    {
        long x = (long)Range.Max;
        long y = (long)Range.Min;
        Console.WriteLine("Max = {0}", x);
        Console.WriteLine("Min = {0}", y);
    }
}
```

Referentni tipovi

Varijable referentnog tipa čuvaju samo reference na stvarne podatke. Razliku u odnosu na vrijednosne tipove smo već spomenuli kod opisa vrijednosnih tipova. Osnovna podjela referentnih tipova je na:

- Klase
- Sučelja
- Delegate

Postoje i dva već ugrađena referentna tipa:

- Object
- String

Sve ove referentne tipove ćemo detaljnije upoznati kroz sljedećih par poglavlja. Prije toga, pogledajmo što se događa kad vrijednosne tipove želimo koristiti kao objekte.

Boxing i unboxing

Boxing je postupak koji je obavlja kad vrijednosni tip "pakiramo" u referentni tip Objekt. To omogućuje vrijednosnom tipu da ga stavimo na gomilu (*heap*) kojom upravlja *garbage collector*. Boxing vrijednosnog tipa alocira instancu objekta na gomili i kopira vrijednost u novi objekt.

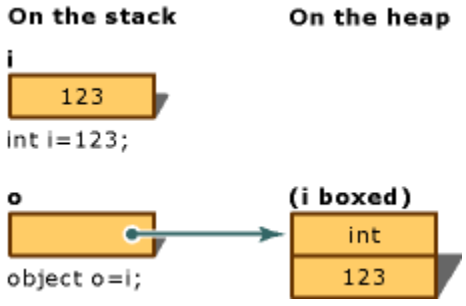
Pogledajmo sljedeću deklaraciju varijable vrijednosnog tipa:

```
int i = 123;
```

Sljedeća naredba implicitno primjenjuje boxing operaciju nad varijablom `i`:

```
object o = i;  // implicit boxing
```

Rezultat ove naredbe se može vidjeti na sljedećoj slici:



U sljedećem primjeru konvertiramo integer varijablu i u objekt o pomoću boxinga. Nakon toga, varijabli i promijenimo vrijednost iz 123 u 456. Primjer pokazuje kako originalni vrijednosni tip i novi *boxed* objekt koriste različite memorijske lokacije (ispišu se vrijednosti 456 i 123).

```
class TestBoxing
{
    static void Main()
    {
        int i = 123;
```

```
        object o = i;  // implicit boxing

        i = 456;  // change the contents of i

        System.Console.WriteLine("The value-type value = {0}", i);
        System.Console.WriteLine("The object-type value = {0}", o);
    }
}
```

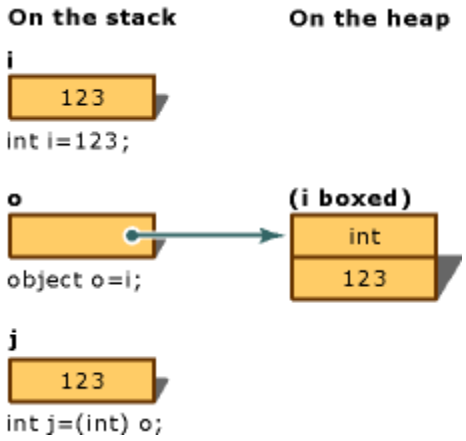
Unboxing je eksplicitna konverzija iz tipa objekt u vrijednosni tip. Unboxing operacija se sastoji od:

- provjere da je instanca objekta boxed vrijednost zadanog vrijednosnog tipa
- kopiranja vrijednosti iz instance u varijablu vrijednosnog tipa

Sljedeći primjer prikazuje i boxing i unboxing operacije:

```
int i = 123;      // a value type
object o = i;     // boxing
int j = (int) o;  // unboxing
```

Na slici je prikazano sto se događa u memoriji kad se izvrše ove tri naredbe.



Nizovi

Niz je podatkovna struktura koja sadrži više varijabli istog tipa. Osnovna svojstva nizova:

- Mogu biti jednodimenzionalni, višedimenzionalni ili nazubljeni nizovi.
- Default vrijednost elemenata numeričkog niza je 0, a referentni elementi su inicijalizirani sa null.
- Nazubljeni niz je niz nizova, pa obzirom da je niz referentni tip, sadrži inicijalno null vrijednosti.
- Prvi element niza ima indeks 0, a zadnji od n elemenata ima indeks n-1
- Elementi niza mogu biti bilo kojeg tipa, uključujući nizove
- Niz je referentni tip izveden iz apstraktnog tipa **`Array`**. Budući da ovaj tip implementira **`IEnumerable`** sučelje, može se koristiti u **`foreach`** iteraciji.

Jednodimenzionalni niz

Niz od pet integera možemo deklarirati ovako:

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

Ovaj niz sadrži elemente **`array[0]`** do **`array[4]`**. Operator new je korišten da se stvori niz i inicijaliziraju elementi niza na njihove default vrijednosti. U ovom slučaju, svi elementi su inicijalizirani na nulu.

Ako želimo imati niz stringova, on se deklarira na isti način.

```
string[] stringArray = new string[6];
```

Inicijalizacija niza

Niz je moguće inicijalizirati prilikom deklaracije i u tom slučaju nije potrebno navoditi broj elemenata jer se on može saznati iz inicijalizacijske liste.

```
string[] workDays = new string[] { "Mon", "Tue", "Wed", "Thu", "Fri" };
```

Ako inicijaliziramo niz zajedno sa deklaracijom, možemo koristiti i skraćeni zapis.

```
string[] workDays2 = { "Mon", "Tue", "Wed", "Thu", "Fri" };
```

Niz je moguće deklarirati i bez inicijalizacije, ali tada moramo koristiti new operator. Skraćeni zapis nije dozvoljen u tom slučaju.

```
int[] array3;
array3 = new int[] { 1, 3, 5, 7, 9 };   // OK
//array3 = {1, 3, 5, 7, 9};   // Error
```

Nizovi vrijednosnih i referentnih tipova

Pogledajte sljedeću deklaraciju niza:

```
SomeType[] array4 = new SomeType[10];
```

Rezultat ove naredbe ovisi da li je **SomeType** referentnog ili vrijednosnog tipa. U slučaju da je vrijednosnog tipa, rezultat naredbe je stvoreni niz od 10 instanci tipa SomeType. Ako je SomeType bio referentnog tip, naredba stvori niz od 10 elemenata i svakog inicijalizira na null referencu.

Višedimenzionalni niz

Nizovi mogu imati i više od jedne dimenzije. Na primjer, sljedeća deklaracija stvara dvodimenzionalni niz od 4 retka i 2 stupca.

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

Sljedeća deklaracija stvara niz sa tri dimenzije, 4, 2 i 3:

```
int[, ,] array1 = new int[4, 2, 3];
```

Inicijalizacija nizova prilikom deklaracije:

```
int[,] array2D = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
int[, ,] array3D = new int[,,] { { { 1, 2, 3 } }, { { 4, 5, 6 } } };
```

Skraćeni zapis inicijalizacije:

```
int[,] array4 = { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
```

Ako se niz ne inicalizira prilikom deklaracije mora se koristiti new operator:

```
int[,] array5;
array5 = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };   // OK
//array5 = {{1,2}, {3,4}, {5,6}, {7,8}};   // Error
```

Nazubljeni niz

Nazubljeni niz (*jagged array*) je niz čiji su elementi nizovi. Elementi nazubljenog niza mogu biti nizovi različitih dimenzija i veličina. Sljedeći primjeri prikazuju deklaraciju, inicijalizaciju i korištenje nazubljenih nizova.

Deklaracija jednodimenzionalnog niza koji ima tri elemenata, i svaki od njih je jednodimenzionalni niz integera:

```
int[][] jaggedArray = new int[3][];
```

Prije nego što se može koristiti **jaggedArray**, mora se inicijalizirati:

```
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
```

Mogli smo umjesto inicijalizacije default vrijednostima (nulama), elemente inicijalizirati na neke druge vrijednosti:

```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };
```

Deklaraciju i inicijalizaciju elemenata smo mogli napraviti i u jednom potezu:

```
int[][] jaggedArray2 = new int[][]
{
    new int[] {1,3,5,7,9},
    new int[] {0,2,4,6},
    new int[] {11,22}
};
```

Skraćeni zapis inicijalizacije je:

```
int[][] jaggedArray3 =
{
    new int[] {1,3,5,7,9},
    new int[] {0,2,4,6},
    new int[] {11,22}
};
```

Primijetite da nismo mogli izostaviti operator new za elemente nazubljenog niza.

Pristup pojedinim elementima izgleda ovako:

```
// Assign 77 to the second element ([1]) of the first array ([0]):
jaggedArray3[0][1] = 77;

// Assign 88 to the second element ([1]) of the third array ([2]):
jaggedArray3[2][1] = 88;
```

Moguće je i miksanje nazubljenih i višedimenzionalnih nizova. Sljedeći primjer deklarira i inicijalizira jednodimenzionalni nazubljeni niz koji sadrži dvodimenzionalne nizove različitih dimenzija:

```
int[][,] jaggedArray4 = new int[3][,]
{
    new int[,] { {1,3}, {5,7} },
    new int[,] { {0,2}, {4,6}, {8,10} },
    new int[,] { {11,22}, {99,88}, {0,9} }
};
```

Stringovi

Niz znakova se deklarira pomoću **`string`** ključne riječi. Vrijednost string varijable možemo zadati tako što joj pridružimo tekst u dvostrukim navodnicima:

```
string s = "Hello, World!";
```

Ako želimo izdvojiti dio string ili spojiti dva stringa, to možemo ovako napraviti:

```
string s1 = "orange";
string s2 = "red";

s1 += s2;
System.Console.WriteLine(s1);  // outputs "orangered"

s1 = s1.Substring(2, 5);
System.Console.WriteLine(s1);  // outputs "anger"
```

String objekti su nepromjenjivi! To znači da se ne mogu promijeniti nakon što su stvoreni. Metode koje rade sa stringovima zapravo vraćaju novi string objekt. U prethodnom primjeru, kad se **`s1`** i **`s2`** konkateniraju u jedan string, ta dva stringa koji sadrže vrijednosti **`"orange"`** i **`"red"`** su nepromijenjeni. Operator **`+=`** stvara novi string koji sadrži kombinirani rezultat. Nakon toga s1 ima referencu na drugi string. String koji sadrži **`"orange"`** još uvijek postoji, ali više nema reference na njega.

Zbog toga, radi poboljšanja performansi, ako radimo veliki broj manipulacija nad stringovima, trebali bi koristiti klasu **`StringBuilder`**, kao u primjeru:

```
System.Text.StringBuilder sb = new System.Text.StringBuilder();
sb.Append("one ");
sb.Append("two ");
sb.Append("three");
string str = sb.ToString();
```

Rad sa stringovima

Isto kao i u C++ jeziku, ako želimo ubaciti posebne znakove u string, koristimo *escape* znakove, poput **`"\n"`** za novi red ili **`"\t"`** za tab.

```
string hello = "Hello\nWorld!";
```

Prethodna naredba ispise riječi Hello i World u dva odvojena retka. Ako želimo ubaciti backslash u string, onda napišemo dvije kose crte:

```
string filePath = "\\\\My Documents\\";
```

Simbol @

Simbol @ govori konstruktoru stringa da ignorira *escape* znakove. Sljedeća dva stringa su identični:

```
string p1 = "\\\\My Documents\\My Files\\";
string p2 = @"\\My Documents\My Files\";
```

ToString()

Kao i svi objekti izvedeni iz klase Objekt, string ima **`ToString()`** metodu koja pretvara vrijednost u string. Ova metoda se koristi da se npr. numeričke vrijednosti pretvore u stringove:

```
int year = 1999;
string msg = "Eve was born in " + year.ToString();
System.Console.WriteLine(msg);  // outputs "Eve was born in 1999"
```

Pristup pojedinim znakovima

Pojedini znakovi koji su sadržani u stringu se mogu dobiti pomoću metoda **`SubString()`**, **`Replace()`**, **`Split()`** i **`Trim()`**.

```
using System;
string s3 = "Visual C#";
Console.WriteLine(s3.Substring(7, 2));        // outputs "C#"
Console.WriteLine(s3.Replace("C#", "Basic")); // outputs "Visual Basic"
```

Moguće je i kopiranje znakova u niz znakova, poput:

```
string s4 = "Hello, World";
char[] arr = s4.ToCharArray(0, s4.Length);

foreach (char c in arr)
{
    System.Console.Write(c);  // outputs "Hello, World"
}
```

Pojedini znakovi iz stringa se mogu dobiti i pomoću indeksa:

```
string s5 = "Backwards";

for (int i = 0; i < s5.Length; i++)
{
    System.Console.Write(s5[s5.Length - i - 1]);  // outputs "sdrawkcaB"
}
```

Velika i mala slova

Da promijenimo slova u stringu u mala ili velika, možemo koristiti ugrađene metode **`ToUpper()`** i **`ToLower():`**

```
string s6 = "Battle of Hastings, 1066";

System.Console.WriteLine(s6.ToUpper());  // outputs "BATTLE OF HASTINGS
1066"
System.Console.WriteLine(s6.ToLower());  // outputs "battle of hastings
1066"
```

Usporedba

Najjednostavniji način da se usporede dva stringa je pomoću **==** i **!=** operatora, koji obavljaju provjeru uzimajući u obzir i velika i mala slova (*case sensitive*).

```
string color1 = "red";
string color2 = "green";
string color3 = "red";

if (color1 == color3)
{
    System.Console.WriteLine("Equal");
}
if (color1 != color2)
{
    System.Console.WriteLine("Not equal");
}
```

String objekti imaju i **CompareTo()** metodu koja vraća integer vrijednost na osnovi da li je jedan string manji ili veći od drugog. Pri tome se misli na Unicode vrijednost znakova u stringu.

```
string s7 = "ABC";
string s8 = "abc";

if (s7.CompareTo(s8) > 0)
{
    System.Console.WriteLine("Greater-than");
}
else
{
    System.Console.WriteLine("Less-than");
}
```

Da bi pronašli neki string unutar stringa, koristimo **IndexOf()** metodu koja vraća indeks znaka u stringu gdje se prvi put pojavljuje zadani string. Ako ne pronađe zadani string, vraća vrijednost -1.

```
string s9 = "Battle of Hastings, 1066";

System.Console.WriteLine(s9.IndexOf("Hastings"));  // outputs 10
System.Console.WriteLine(s9.IndexOf("1967"));      // outputs -1
```

Podjela stringa na podstringove

Podjela stringa na podstringove, poput recimo podjela rečenice u pojedine riječi, je čest programski zadatak. **Split()** metoda uzima kao parametar niz znakova u kojem se nalaze delimiteri po kojima će se izvršavati podjela (npr. razmak), a vraća niz podstringova.

```
char[] delimit = new char[] { ' ' };
string s10 = "The cat sat on the mat.";
foreach (string substr in s10.Split(delimit))
{
    System.Console.WriteLine(substr);
}
```

StringBuilder

`StringBuilder` klasa stvara string buffer koji nudi bolje performanse od običnog stringa ako se u programu obavlja puno manipulacija sa string objektima. StringBuilder osim toga omogućuje i naknadnu promjenu pojedinih znakova. Primjer prikazuje kako promijeniti sadržaj stringa bez da stvorimo novi objekt:

```
System.Text.StringBuilder sb = new System.Text.StringBuilder("Rat: the
ideal pet");
sb[0] = 'C';
System.Console.WriteLine(sb.ToString());
System.Console.ReadLine();
```

U ovom primjeru se StringBuilder objekt koristi da bi se stvorio string iz seta numeričkih tipova:

```
class TestStringBuilder
{
    static void Main()
    {
        System.Text.StringBuilder sb = new System.Text.StringBuilder();

        // Create a string composed of numbers 0 - 9
        for (int i = 0; i < 10; i++)
        {
            sb.Append(i.ToString());
        }
        System.Console.WriteLine(sb);  // displays 0123456789

        // Copy one character of the string
        // (not possible with a System.String)
        sb[0] = sb[9];

        System.Console.WriteLine(sb);  // displays 9123456789
    }
}
```

Klase

Klasa je najmoćniji podatkovni oblik u C#-u. Poput struktura, klasa definira podatke i ponašanje. Programeri nakon toga mogu stvoriti objekte koji su instance tih klasa. Za razliku od struktura, klase podržavaju nasljeđivanje što je fundamentalni dio objektno-orijentiranog programiranja.

Klase se definiraju pomoću **`class`** ključne riječi, kao što je prikazanu u primjeru:

```
public class Customer
{
    //Fields, properties, methods and events go here...
}
```

Prije class ključne riječi se nalazi riječ **`public`** koja označava da svatko može stvoriti objekt iz ove klase. Ime se nalazi nakon class riječi. Ostatak definicije se nalazi unutar vitičastih zagrada gdje se obično nalaze definicije podataka i ponašanja objekta. Polja (*fields*), svojstva, metode, događaji i drugi dijelovi klase se zovu članovi klase.

Objekt se stvara pomoću ključne riječi **`new`** nakon koje trebamo navesti ime klase po kojoj će taj objekt biti baziran:

```
Customer object1 = new Customer();
```

Jednom kad je instanca klase stvorena, referenca na taj objekt je vraćena programeru. U ovom slučaju, **`object1`** je referenca na objekt baziran na **`Customer`** klasi. Ta referenca pokazuje na novi objekt, ali ne sadrži i same podatke. Štoviše, referenca se može stvoriti bez da se stvori objekt:

```
Customer object2;
```

Stvaranje reference bez da stvorimo i objekt može dovesti do pogreške ako pokušamo pristupiti objektu pomoću te reference. Ako smo već stvorili referencu možemo zadati da pokazuje na neki novi objekt ili na neki već postojeći.

```
Customer object3 = new Customer();
Customer object4 = object3;
```

U prethodnom slučaju obje reference pokazuju na isti objekt, pa ako promijenimo nešto u objektu pristupajući mu preko prve reference, promjene će biti vidljive ako nakon toga pristupimo i preko druge reference.

Polimorfizam

Klase mogu nasljeđivati druge klase. To možemo napraviti tako što stavimo dvotočku nakon imena klase prilikom njene deklaracije, i dodamo ime klase koju želimo naslijediti:

```
public class A
{
    public A() { }
}
```

```
public class B : A
{
    public B() { }
}
```

Nova, izvedena klasa dobiva sve ne-privatne podatke i ponašanje bazne klase, čemu doda još i podatke i ponašanje koje sama definira. Nova klasa efektivno ima dva tipa: tip bazne klase i tip nove klase. U prethodnom primjeru to znači da je klasa B efektivno i B i A tipa. Kad pristupamo B objektu, možemo ga koristiti kao da je to A objekt (npr. koristi ga kao parametar neke metode koja zahtijeva objekt klase A). Obrnuto ne vrijedi. Svojstvo da objekt može predstavljati više od jednog tipa podataka zove se polimorfizam.

Kroz nasljeđivanje, klasa se može koristiti kao više od jednog tipa. Može se koristiti kao svoj tip, svoj bazni tip ili kao tip sučelja ako je implementirano koje sučelje. U C#-u svaki tip je polimorfan jer je svaki tip izveden iz **`Object`** tipa.

Što da radimo ako ne želimo samo proširiti baznu klasu pomoću nasljeđivanja, nego i promijeniti njezine već definirane podatke i ponašanje? Imamo dvije opcije: možemo zamijeniti članove iz bazne klase sa novim članom ili možemo prospojiti (*override*) virtualni bazni član.

Zamjena člana bazne klase sa novim izvedenim članom zahtijeva **`new`** ključnu riječ. Ako je bazna klasa definirala metodu, polje ili svojstvo, new ključna riječ se stavlja prije povratnog tipa člana koji želimo zamijeniti u izvedenoj klasi:

```
public class BaseClass
{
    public void DoWork() { }
    public int WorkField;
    public int WorkProperty
    {
        get { return 0; }
    }
}

public class DerivedClass : BaseClass
{
    public new void DoWork() { }
    public new int WorkField;
    public new int WorkProperty
    {
        get { return 0; }
    }
}
```

Kad je korištena new ključna riječ, članovi izvedene klase se pozivaju umjesto članova bazne klase. Članovi bazne klase se zovu skriveni članovi. Skriveni članovi se još uvijek mogu pozvati ako izvedenu klasu *castamo* kao instancu bazne klase:

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.
BaseClass A = (BaseClass)B;
A.DoWork();  // Calls the old method.
```

Da bi instanca izvedene klase u potpunosti preuzela članove bazne klase, u baznoj klasi članovi trebaju biti deklarirani kao virtualni. To radimo tako što dodamo **`virtual`** ključnu

riječ ispred tipa povratne vrijednosti člana. Izvedena klasa onda ima opciju da iskoristi **override** ključnu riječ umjesto new da bi zamijenila implementaciju sa svojom:

```
public class BaseClass
{
    public virtual void DoWork() { }
    public virtual int WorkProperty
    {
        get { return 0; }
    }
}
public class DerivedClass : BaseClass
{
    public override void DoWork() { }
    public override int WorkProperty
    {
        get { return 0; }
    }
}
```

Polja ne mogu biti virtualni, samo metode, svojstva, događaji i indekseri. Kad izvedena klasa premosti virtualnog člana, taj član se zove čak i ako se instanci nove klase pristupa kao da je starog, baznog tipa:

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork();  // Also calls the new method.
```

Virtualni član ostane virtualni bez obzira koliko puta je naslijeđena bazna klasa. Ako klasa A deklarira virtualnog člana, klasa B naslijedi klasu A i klasa C naslijedi klasu B, klasa C je naslijedila virtualnog člana i ima opciju da ga premosti, bez obzira da li je klasa B premostila tog člana u svojoj definiciji:

```
public class A
{
    public virtual void DoWork() { }
}
public class B : A
{
    public override void DoWork() { }
}
public class C : B
{
    public override void DoWork() { }
}
```

Izvedena klasa može zaustaviti virtualno nasljeđivanje ako deklarira premoštenog člana kao **sealed**:

```
public class C : B
{
    public sealed override void DoWork() { }
}
```

Sada metoda **DoWork** više nije virtualna ni jednoj klasi koja se izvede iz klase C. Ona je još uvijek virtualna za instance klase C, čak i ako se castaju kao tip B ili A. Sealed metode se mogu zamijeniti u izvedenim klasama pomoću new ključne riječi:

```
public class D : C
{
    public new void DoWork() { }
}
```

U ovom slučaju, ako se metoda DoWork pozove na varijabli tipa D, poziva se nova metoda. Ako se varijable tipa A, B ili C koriste za pristup objektu tipa D i poziv metodi DoWork, prate se pravila virtualnog nasljeđivanja, te se poziv prosljeđuje do implementacije u klasi C.

Apstraktne i sealed klase

Ključna riječ **abstract** omogućuje nam da stvorimo klase i članove klase čije je svrha da definiraju svojstva koja moraju implementirati izvedene, ne-apstraktne klase.

```
public abstract class A
{
    // Class members here.
}
```

Apstraktne klase se ne mogu instancirati. Njihova svrha je da pruže zajedničku definiciju bazne klase koju nasljeđuje više klasa. Apstraktne klase mogu definirati i apstraktne metode:

```
public abstract class A
{
    public abstract void DoWork(int i);
}
```

Apstraktne metode nemaju implementaciju pa imaju samo točku-zarez umjesto normalne definicije u vitičastim zagradama. Izvedene klase apstraktne klase moraju implementirati sve apstraktne metode. Kad apstraktna klasa nasljeđuje virtualnu metodu iz bazne klase, ona može prospojiti virtualnu metodu sa apstraktnom metodom:

```
public class D
{
    public virtual void DoWork(int i)
    {
        // Original implementation.
    }
}

public abstract class E : D
{
    public abstract override void DoWork(int i);
}

public class F : E
{
    public override void DoWork(int i)
    {
        // New implementation.
    }
}
```

Ako je virtualna metoda deklarirana kao apstraktna, ona je još uvijek virtualna za klasu koja naslijedi apstraktnu klasu. Izvedena klasa ne može pristupiti implementaciji metode. U prethodnom primjeru DoWork klase F ne može pozvati DoWork klase D. Na taj način, pomoću apstraktne klase smo natjerali izvedenu klasu da stvori novu implementaciju za virtualnu metodu.

Ako ne želimo da se klasa može naslijediti definiramo je kao **sealed**. Obzirom da se sealed klase ne mogu koristiti kao bazne klase, neki run-time optimizatori mogu ubrzati poziv sealed članova. Primjer sealed klase:

```
public sealed class D
{
    // Class members here.
}
```

Sučelja

Sučelje se definira pomoću **interface** ključne riječi. Npr:

```
interface IComparable
{
    int CompareTo(object obj);
}
```

Sučelja opisuju skupinu povezanih ponašanja koji mogu pripadati bilo kojoj klasi ili strukturi. Sastoje se od metoda, svojstava, događaja i indeksera. Ne može sadržavati polja. Sučelja su automatski public.

Klase i strukture koje nasljeđuju sučelja slični su klasama koje nasljeđuju baznu klasu ili strukturu, sa dvije iznimke:

- Klasa ili struktura može naslijediti više od jednog sučelja
- Kad klasa ili struktura naslijedi sučelje, ona nasljeđuje definicije članova, ali ne i implementacije, npr:

```
public class Minivan : Car, IComparable
{
    public int CompareTo(object obj)
    {
        //implementation of CompareTo
        return 0;  //if the Minivans are equal
    }
}
```

Da bi implementirali člana sučelja, taj član mora biti public, non-static i imati isto ime i potpis kao član sučelja. Pod potpisom se misli na tipove parametara i povratne vrijednosti. Svojstva i indekseri mogu dodavati i ekstra načine pristupa. Ako npr. sučelje definira **get** dio svojstva, klasa koja implementira sučelje može definirati i get i **set** dio svojstva.

Sučelja mogu naslijediti druga sučelja Moguće je da klasa naslijedi neko sučelje više puta, kroz baznu klasu i sučelje koje je sam definirao. U tom slučaju, klasa implementira sučelje samo jednom ako je deklarirano kao dio nove klase. Ako naslijeđeno sučelje nije deklarirano kao dio nove klase, njegova implementacija se uzima iz bazne klase.

Eksplicitna implementacija sučelja

Ako klasa implementira dva sučelja koja sadrže članove sa istim potpisom, onda će implementacija tog člana u klasi prouzročiti da ova sučelja koriste tu istu implementaciju.

```
interface IControl
{
    void Paint();
}
interface ISurface
{
    void Paint();
}
class SampleClass : IControl, ISurface
{
    // Both ISurface.Paint and IControl.Paint call this method.
    public void Paint()
    {
    }
}
```

Ako dva člana sučelja ne bi trebali izvršavati istu funkciju, onda ovo može dovesti do krive implementacije za jednog ili oba člana. Moguće je implementirati člana sučelja eksplicitno tako da se član klase poziva samo preko sučelja. Ovo je izvedeno imenovanjem člana klase sa imenom sučelja i točkom:

```
public class SampleClass : IControl, ISurface
{
    void IControl.Paint()
    {
        System.Console.WriteLine("IControl.Paint");
    }
    void ISurface.Paint()
    {
        System.Console.WriteLine("ISurface.Paint");
    }
}
```

Član klase **`IControl.Paint`** je dostupan kroz sučelje **`IControl`** i **`ISurface.Paint`** je dostupan kroz sučelje **`ISurface`**. Obje implementacije metode su odvojene i ni jedna nije dostupna direktno preko klase :

```
SampleClass obj = new SampleClass();
//obj.Paint();  // Compiler error.

IControl c = (IControl)obj;
c.Paint();  // Calls IControl.Paint on SampleClass.

ISurface s = (ISurface)obj;
s.Paint(); // Calls ISurface.Paint on SampleClass.
```

Eksplicitna implementacija se koristi i za rješavanje slučaja kad dva sučelja deklariraju različite članove sa istim imenom poput svojstva i metode:

```
interface ILeft
{
    int P { get;}
}
interface IRight
```

```
{
    int P();
}
```

Da bi implementirali oba sučelje klasa mora koristiti eksplicitnu implementaciju za svojstvo P ili za metodu P, ili za oboje, kako bi se izbjegla pogreška pri kompilaciji. Npr:

```
class Middle : ILeft, IRight
{
    public int P() { return 0; }
    int ILeft.P { get { return 0; } }
}
```

Članovi klase

Klase i strukture imaju članove koji predstavljaju njihove podatke i ponašanje. Vrste članova:

- Polja
- Svojstva
- Metode
- Događaji
- Operatori
- Indekseri
- Konstruktori
- Destruktori
- Ugniježđeni tipovi

Polja

Polja omogućuju klasama i strukturama da enkapsuliraju podatke. Radi jednostavnosti, u sljedećim primjerima su polja deklarirana kao public, iako se to ne preporučuje u praksi. Polja bi trebala biti private, a pristup bi trebao biti indirektan, kroz svojstva, indeksere ili metode.

Polja sadrže podatke koji su potrebni klasi da obavi svoju zadaću. Tako na primjer, klasa koja predstavlja kalendar bi mogla imati tri integer polja: jedan za dan, drugi za mjesec i treći za godinu:

```
public class CalendarDate
{
    public int month;
    public int day;
    public int year;
}
```

Pristup pojedinom polju u objektu se ostvaruje imenovanjem objekta, zatim točka, pa ime polja:

```
CalendarDate birthday = new CalendarDate();
birthday.month = 7;
```

Polju možemo zadati inicijalnu vrijednost koristeći operator pridruživanja prilikom deklaracije. Ako želimo automatski dodijeliti vrijednost 7 polju za mjesec, to ćemo napravit ovako:

```
public class CalendarDateWithInitialization
{
    public int month = 7;
    //...
}
```

Polja se inicijaliziraju prije nego što se pozove konstruktor za objekt, pa tako, ako u konstruktoru zadamo neku drugu vrijednost polju, inicijalna vrijednost će se izgubiti.

Konstante

Klase i strukture mogu deklarirati konstante kao članove. Konstante su vrijednosti koje su poznate pri kompilaciji i ne mijenjaju se. (Ako želite stvoriti konstantnu vrijednost koja će se inicijalizirati u run-time, koristite **`readonly`** ključnu riječ). Konstante su deklarirane kao polja, koristeći **`const`** ključnu riječ prije tipa polja. Konstante moraju biti inicijalizirane pri deklaraciji:

```
class Calendar1
{
    public const int months = 12;
}
```

U ovom primjeru, polje sa mjesecom će uvijek biti 12 i ne može biti promijenjen, čak ni unutar klase. Konstante moraju biti integralnog tipa, pobrojani tip ili referenca na null.

Moguće je i deklarirati više konstanti odjednom:

```
class Calendar2
{
    const int months = 12, weeks = 52, days = 365;
}
```

Izraz koji se koristi za inicijalizaciju konstante može sadržavati neku drugu konstantu, sve dok nema kružnih referenci, npr:

```
class Calendar3
{
    const int months = 12;
    const int weeks = 52;
    const int days = 365;

    const double daysPerWeek = days / weeks;
    const double daysPerMonth = days / months;
}
```

Ugniježđeni tip

Tip definiran unutar klase ili strukture se zove ugniježđeni tip.

```
class Container
{
```

```
    class Nested
    {
        Nested() { }
    }
}
```

Unutarnji tip može pristupiti vanjskom tipu ako mu se konstruktoru preda referenca na vanjski tip:

```
public class Container
{
    public class Nested
    {
        private Container m_parent;

        public Nested()
        {
        }
        public Nested(Container parent)
        {
            m_parent = parent;
        }
    }
}
```

Metode

Metoda je blok koda koji sadrži niz naredbi. U C#-u, svaka izvršena instrukcija se nalazi u kontekstu metode.

Metode su deklarirane unutar klase ili strukture tako što su specificirani: nivo pristupa, povratna vrijednost, ime metode i parametri metode. Parametri su okruženi zagradama i razdvojeni zarezima. Prazne zagrade naznačuju da metoda ne zahtijeva nikakve parametre. Ova klasa ima tri metode:

```
class Motorcycle
{
    public void StartEngine() { }
    public void AddGas(int gallons) { }
    public int Drive(int miles, int speed) { return 0; }
}
```

Pozivanje metode nekog objekta je slično pristupanju polju objekta. Nakon imena objekta stavi se točka, pa ime metode, te zagrade sa eventualnim parametrima:

```
Motorcycle moto = new Motorcycle();

moto.StartEngine();
moto.AddGas(15);
moto.Drive(5, 20);
```

Parametri vrijednosnog tipa

Kad predajemo varijablu vrijednosnog tipa metodi, zapravo predajemo kopiju varijable metodi. svaka promjena izvršena nad parametrom unutar metode neće imati nikakvog utjecaja na originalne podatke pohranjene u varijabli izvan metode. Ako želite pozvati metodu koja će promijeniti vrijednost parametra, tada je moramo prenijeti po referenci koristeći **`ref`** ili **`out`** ključne riječi. Sljedeći primjeri koriste ref.

Prvo ćemo predati parametar vrijednosnog tipa po vrijednosti. Svaka promjena unutar metode **`SquareIt`** neće imati utjecaja na varijablu **`n`** izvan metode.

```
class PassingValByVal
{
    static void SquareIt(int x)
    // The parameter x is passed by value.
    // Changes to x will not affect the original value of x.
    {
        x *= x;
        System.Console.WriteLine("The value inside: {0}", x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("The value before: {0}", n);

        SquareIt(n);  // Passing the variable by value.
        System.Console.WriteLine("The value after: {0}", n);
    }
}
```

Rezultat je 5, 25, 5. Sljedeći primjer je skoro isti prethodnome, ali ovaj put varijablu predajemo po referenci:

```
class PassingValByRef
{
    static void SquareIt(ref int x)
    // The parameter x is passed by reference.
    // Changes to x will affect the original value of x.
    {
        x *= x;
        System.Console.WriteLine("The value inside: {0}", x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("The value before: {0}", n);

        SquareIt(ref n);  // Passing the variable by reference.
        System.Console.WriteLine("The value after: {0}", n);
    }
}
```

Ovaj put je rezultat 5, 25, 25.

Parametri referentnog tipa

Varijabla referentnog tipa ne sadrži podatke direktno, već sadrži referencu na podatke. Kada predamo parametar referentnog tipa po vrijednosti, moguće je promijeniti podatke na koje

pokazuje referenca, ali ne možemo promijeniti vrijednost reference same. Unutar metode možemo stvoriti novi objekt i referencu preusmjeriti na njega, ali to će trajati koliko i metoda. Izvan metode, referenca će opet pokazivati na stari objekt. Da bi promijenili moramo koristiti ref ili out ključne riječi.

Prvo ćemo demonstrirati predavanje parametra referentnog tipa po vrijednosti. Budući da je parametar referenca na niz, elemente niza je moguće mijenjati, ali ne možemo parametar postaviti da pokazuje na drugu memorijsku lokaciju, tj. barem ne izvan metode.

```
class PassingRefByVal
{
    static void Change(int[] pArray)
    {
        pArray[0] = 888;  // This change affects the original element.
        pArray = new int[5] {-3, -1, -2, -3, -4};   // Local change
        System.Console.WriteLine("Inside the method, the first element is:
                                  {0}", pArray[0]);
    }

    static void Main()
    {
        int[] arr = {1, 4, 5};
        System.Console.WriteLine("Inside Main, before calling the method,
                                  the first element is: {0}", arr [0]);

        Change(arr);
        System.Console.WriteLine("Inside Main, after calling the method,
                                  the first element is: {0}", arr [0]);
    }
}
```

Promjene nisu trajne zato što se zapravo metodi predaje kopija reference kad ne koristimo ključnu riječ ref. Kopija se mijenja unutar metode, ali original ostaje isti.

Kad koristimo ref, realokacija niza postaje trajna:

```
class PassingRefByRef
{
    static void Change(ref int[] pArray)
    {
        // Both of the following changes will affect
        // the original variables:
        pArray[0] = 888;
        pArray = new int[5] {-3, -1, -2, -3, -4};
        System.Console.WriteLine("Inside the method, the first element is:
                                  {0}", pArray[0]);
    }

    static void Main()
    {
        int[] arr = {1, 4, 5};
        System.Console.WriteLine("Inside Main, before calling the method,
                                  the first element is: {0}", arr[0]);

        Change(ref arr);
        System.Console.WriteLine("Inside Main, after calling the method,
                                  the first element is: {0}", arr[0]);
    }
}
```

Konstruktori

Konstruktori su metode klase koje se izvršavaju kada se neki objekt stvori. Konstruktori imaju isto ime kao i klasa i obično se koriste za inicijalizaciju podatkovnih članova novog objekta.

U sljedećem primjeru klasa ima definiran jednostavni konstruktor. Nakon toga, klasa se instancira pomoću new operatora. Konstruktor je pozvan od new operatora odmah nakon alokacije memorije za novi objekt.

```
public class Taxi
{
    public bool isInitialized;
    public Taxi()
    {
        isInitialized = true;
    }
}

class TestTaxi
{
    static void Main()
    {
        Taxi t = new Taxi();
        System.Console.WriteLine(t.isInitialized);
    }
}
```

Konstruktor koji ne prima nikakve parametre zove se default konstruktor. Oni se pozivaju kad god je objekt instanciran pomoću new operatora bez ikakvih proslijeđenih argumenata. Osim ako klasa nije static, klase bez konstruktora dobivaju public default konstruktor od C# kompajlera. On postavlja podatkovne članove na njihove default vrijednosti.

Ako ne želimo da se neka klasa instancira, možemo postaviti konstruktor da bude private. To nam može koristiti ako npr. klasa ima samo static članove i instanciranje takve klase nema smisla.

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = System.Math.E;  //2.71828...
}
```

Konstruktori mogu primati parametre i takvi se mogu pozivati pomoću new izraza ili **base** izraza. Klase mogu imati i više konstruktora i ne moraju obavezno imati default konstruktor:

```
public class Employee
{
    public int salary;

    public Employee(int annualSalary)
    {
        salary = annualSalary;
    }

    public Employee(int weeklySalary, int numberOfWeeks)
```

```
    {
        salary = weeklySalary * numberOfWeeks;
    }
}
```

Sada klasa može biti instancirana na dva načina:

```
Employee e1 = new Employee(30000);
Employee e2 = new Employee(500, 52);
```

Konstruktor može korstiti **base** ključnu riječ da pozove konstruktor bazne klase:

```
public class Manager : Employee
{
    public Manager(int annualSalary)
        : base(annualSalary)
    {
        //Add further instructions here.
    }
}
```

U ovom slučaju, konstruktor bazne klase se poziva prije nego što se počeo izvršavati blok naredbi ovog konstruktora. Bilo koji parametar koji je predan konstruktoru se može koristiti kao parametar za base.

U izvedenoj klasi, ako konstruktor bazne klase nije eksplicitno pozvan pomoću base, onda se automatski poziva default konstruktor, ako postoji. To znači da su ove dvije deklaracije konstruktora iste:

```
public Manager(int initialdata)
{
    //Add further instructions here.
}
```

```
public Manager(int initialdata) : base()
{
    //Add further instructions here.
}
```

Ako bazna klasa nema default konstruktor, izvedena klasa mora eksplicitno pozvati bazni konstruktor sa base.

Konstruktor može pozvati drugi konstruktor istog objekta pomoću ključne riječi **this**. Kao i base, this se može zvati sa i bez parametara, i svi parametri koji predani konstruktoru se mogu predati i drugom konstruktoru kojeg zovemo sa this. Prvi konstruktor zove drugoga u sljedećem primjeru:

```
public Employee(int weeklySalary, int numberOfWeeks)
    : this(weeklySalary * numberOfWeeks)
{
}
public Employee(int annualSalary)
{
    salary = annualSalary;
}
```

Svojstva

Svojstva su članovi klase koji omogućuju fleksibilnost pri čitanju, pisanju ili izračunu privatnih polja. Svojstva se mogu koristiti kao da su podatkovni članovi, ali u stvarnosti, to su posebne metode koje se zovu pristupnici (*accessors*). Pristupnik za čitanje je definiran u **`get`** bloku, a pristupnik za pisanje u **`set`** bloku. To omogućuje podacima da im se može lagano pristupiti, ali uz sigurnost i fleksibilnost koju pružaju metode.

U sljedećem primjeru klasa **`TimePeriod`** pohranjuje vremenski period. Interno, klasa pohranjuje vrijeme u sekundama, ali svojstvo pod imenom **`Hours`** omogućuje klijentu (korisniku klase) da koristi vrijednost vremenskog perioda u jedinici sat. Pristupnici svojstva Hours obavljaju konverziju između sati i sekundi:

```
class TimePeriod
{
    private double seconds;

    public double Hours
    {
        get { return seconds / 3600; }
        set { seconds = value * 3600; }
    }
}

class Program
{
    static void Main()
    {
        TimePeriod t = new TimePeriod();

        // Assigning the Hours property causes
        // the 'set' accessor to be called.
        t.Hours = 24;

        // Evaluating the Hours property causes
        // the 'get' accessor to be called.
        System.Console.WriteLine("Time in hours: " + t.Hours);
    }
}
```

Korištenje svojstava

Svojstva kombiniraju aspekte i polja i metoda. Korisniku objekta svojstvo izgleda isto kao polje. Sintaksa im je identična. Get pristupnik se izvršava kada se svojstvo čita, a set pristupnik kad se svojstvu pridružuje nova vrijednost. Svojstvo bez set pristupnika se smatra kao read-only, a svojstvo bez get pristupnika kao write-only. Ako ima oba pristupnika onda se zove read-write svojstvo.

Svojstva nam pružaju mnoge pogodnosti: mogu validirati podatke prije nego što dopuste promjenu vrijednosti, mogu transparentno prikazati podatke u klasi iako se možda podaci dobavljaju iz nekog drugog izvora, poput baze podataka, mogu obaviti neku akciju kad se podaci promijene, poput dizanja događaja ili promijene vrijednosti nekog dodatnog polja.

Deklaracija svojstva ima prvo ima način (nivo) pristupa, zatim tip svojstva, pa ime svojstva i na kraju get i/ili set pristupnike:

```
public class Date
{
    private int month = 7;  //"backing store"

    public int Month
    {
        get
        {
            return month;
        }
        set
        {
            if ((value > 0) && (value < 13))
            {
                month = value;
            }
        }
    }
}
```

Get pristupnik

Tijelo pristupnika je slično metodi. Mora vratiti vrijednost koja je tipa kao svojstvo. Izvođenje get pristupnika je ekvivalentno čitanju vrijednosti polja. Na primjer, kad vraćamo privatnu varijablu iz get pristupnika i optimizacije su uključene, poziv get pristupniku je *inlined* od strane kompajlera tako su performnse bolje nego da se poziva metoda. Ako je pristupnik virtualan, kompajler to ne može napraviti jer ne zna pri kompilaciji koja će se metoda zapravo zvati.

Kad referenciramo svojstvo, osim u slučaju da mu se nešto pridružuje, get pristupnik se poziva da se pročita vrijednost svojstva. Npr:

```
Person p1 = new Person();
//...

System.Console.Write(p1.Name);  // the get accessor is invoked here
```

Get pristupnik mora završiti sa **`return`** ili **`throw`** naredbom.

Loš programski stil je ako mijenjamo stanje objekta kroz get pristupnik (iako je moguće), kao u sljedećem primjeru:

```
private int number;
public int Number
{
    get
    {
        return number++;   // Don't do this
    }
}
```

Get pristupnik se može koristiti da vrati već postojeću vrijednost nekog polja ili da izračuna neku novu vrijednost i vrati je:

```
class Employee
{
    private string name;
    public string Name
    {
        get
        {
            return name != null ? name : "NA";
        }
    }
}
```

Set pristupnik

Set pristupnik je sličan metodi koja ima void povratni tip. On koristi implicitni parametar zvan **value**, čiji je tip isti kao u svojstva:

```
class Person
{
    private string name;  // the name field
    public string Name    // the Name property
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }
}
```

Kada pridružujemo vrijednost svojstvu, set pristupnik se poziva sa argumentom koji predstavlja novu vrijednost:

```
Person p1 = new Person();
p1.Name = "Joe";  // the set accessor is invoked here

System.Console.Write(p1.Name);  // the get accessor is invoked here
```

Nije dozvoljeno deklarirati lokalnu varijablu sa imenom value unutar set pristupnika.

Asimetrična svojstva

Po defaultu pristupnici imaju istu vidljivost (nivo pristupa), i to onu koja je zadana u definiciji svojstva. Međutim, ponekad je zgodno ograničiti pristup nekom od pristupnika. Obično je to set pristupnik koji se ograničava, dok get pristupnik ostaje public:

```
public string Name
{
    get
    {
        return name;
    }
    protected set
    {
        name = value;
```

```
    }
}
```

Pri tome postoji par ograničenja koji se moraju poštivati:

- Modifikatore pristupa se ne može koristiti pri implementaciji sučelja.
- Oba pristupnika moraju biti definirana i modificirati pristup možemo samo za jednog od pristupnika.
- Ako svojstvo ima override modifikator, modifikator pristupa za pristupnik mora biti isti kao u pristupniku kojeg prespajamo.
- Novo pristupa mora biti ograničeniji od onog koji je definiran za samo svojstvo

Delegati

Delegat je tip za referenciranje metoda. Kada se delegat pridruži metodi, ponaša se isto kao i ta metoda. Može se koristiti kao i svaka metoda, sa parametrima i povratnom vrijednosti.

```
public delegate void Del(string message);
```

Gora je navedena jedna definicija delegata. Svaka metoda koja odgovara potpisu delegata, koji se sastoji od povratnog tipa i parametara, se može pridružiti delegatu.

Ovo svojstvo da se metoda koristi kao parametar čini delegate idealnim za definiranje *callback* metoda. Na primjer, sort algoritam može primiti referencu na metodu koja uspoređuje dva objekta. Odvajanjem koda za usporedbu omogućuje algoritmu da se može iskoristiti za više tipova objekata.

Delegat objekt se obično gradi tako što se navede ime metode koji će delegat predstavljati. Kad se delegat instancira, poziv delegata se prosljeđuje metodi. Parametri koji su predani delegatu se daju metodi, isto tako se povratna vrijednost metode vraća delegatu. Delegat se u principu ponaša kao i metoda na koju je vezan. Primjer:

```
// Create a method for a delegate.
public static void DelegateMethod(string message)
{
    System.Console.WriteLine(message);
}
```

```
// Instantiate the delegate.
Del handler = DelegateMethod;

// Call the delegate.
handler("Hello World");
```

Delegat tipovi se izvode iz Delegate klase .NET Frameworka. Delegate tipovi su *sealed* (ne mogu se naslijediti) pa nije moguće izvesti *custom* klasu iz Delegate klase. Zato što su instancirani delegati objekti, mogu se proslijediti kao parametar ili pridružiti nekom svojstvu. Ovo omogućuje metodama da prime delegata kao parametar i da ga pozovu kasnije. Ovo se obično zove asinkroni povratni poziv i predstavlja tipičnu metodu kako obavijestiti pozivatelja kad neki dugotrajni proces završi. Kad je delegat korišten na ovaj način, kod koji koristi delegata ne mora znati koja je metoda povezana uz delegat. Funkcionalnost je slična enkapsulaciji koju sučelja pružaju.

Primjer kada predajemo delegat kao parametar nekoj metodi:

```
public void MethodWithCallback(int param1, int param2, Del callback)
{
    callback("The number is: " + (param1 + param2).ToString());
}
```

Sada možemo predati delegat koji smo stvorili u prethodnom primjeru:

```
MethodWithCallback(1, 2, handler);
```

Kada je delegat stvoren da obuhvati metodu instance, delegat referencira i metodu i referencu. Delegat ne zna koji je tip instance koju referencira, samo metodu koju sadrži ta instanca i da potpisom odgovara delegatu. Kada delegat predstavlja static metodu, onda referencira samo metodu. Pogledajte sljedeći primjer:

```
public class MethodClass
{
    public void Method1(string message) { }
    public void Method2(string message) { }
}
```

Sada sa static metodom **DelegateMethod** od prije, imamo tri metode koje se mogu obuhvatiti sa **Del** instancom.

Delegat može pozvati više od jedne metode. Ovo se zove *multicasting*. Da bi dodali ekstra metodu delegatovoj listi metoda koje poziva (*invocation list*), koristimo **+** ili **+=** operatore:

```
MethodClass obj = new MethodClass();
Del d1 = obj.Method1;
Del d2 = obj.Method2;
Del d3 = DelegateMethod;

//Both types of assignment are valid.
Del allMethodsDelegate = d1 + d2;
allMethodsDelegate += d3;
```

Sada **allMethodsDelegate** sadrži tri metode u svojoj liti - **Method1**, **Method2** i **DelegateMethod**. Originalna tri delegata **d1**, **d2**, i **d3** ostaju nepromijenjeni. Kad se pozove allMethodsDelegate, sve tri metode se pozivaju redom. Ako delegat koristi reference parametre, referenca se predaje redom svakoj metodi i svaka promjena koju napravi jedna metoda je vidljiva u sljedećoj. Ako neka metoda baci iznimku koja nije obrađena unutar metode, ta iznimka se predaje pozivatelju delegata i ostale metode u listi od delegata se ne pozivaju. Ako delegat ima povratnu vrijednost ili izlazne parametre, vraćaju se povratna vrijednost i parametri zadnje pozvane metode. Da bi neku metodu izbacili iz pozivne liste delegata, koristimo **-** ili **-=** operatore:

```
//remove Method1
allMethodsDelegate -= d1;

// copy AllMethodsDelegate while removing d2
Del oneMethodDelegate = allMethodsDelegate - d2;
```

Kovarijanca i kontravarijanca

Kovarijanca i kontravarijanca pružaju određen stupanj fleksibilnosti kada određujemo podudarnost potpisa metode i delegata. Kovarijanca omogućuje metodi da koristi izvedeni povratni tip kod delegata, a kontravarijanca omogućuje da se metoda koristi kao delegat čiji su parametri tipa koji je izveden od onih u potpisu metode.

Primjer kovarijance:

```
class Mammals
{
}
```

```
class Dogs : Mammals
{
}

class Program
{
    // Define the delegate.
    public delegate Mammals HandlerMethod();

    public static Mammals FirstHandler()
    {
        return null;
    }

    public static Dogs SecondHandler()
    {
        return null;
    }

    static void Main()
    {
        HandlerMethod handler1 = FirstHandler;

        // Covariance allows this delegate.
        HandlerMethod handler2 = SecondHandler;
    }
}
```

Primjer kontravarijance:

```
class Mammals
{
}

class Dogs : Mammals
{
}

class Program
{
    public delegate void HandlerMethod(Dogs sampleDog);

    public static void FirstHandler(Mammals elephant)
    {
    }

    public static void SecondHandler(Dogs sheepDog)
    {
    }

    static void Main(string[] args)
    {
        // Contravariance permits this delegate.
        HandlerMethod handler1 = FirstHandler;

        HandlerMethod handler2 = SecondHandler;
    }
}
```

Događaji

Događaji omogućuju nekoj klasi da pošalje obavijest da se nešto zanimljivo dogodilo. Na primjer, klasa koja predstavlja kontrolu korisničkog sučelja može definirati događaj koji se dogodi kad korisnik napravi klik s mišem na nju. Klasu ne zanima što će se dogoditi kad se klikne na gumb, ali želi obavijestiti izvedene klase da se klik događaj dogodio. Izvedene klase mogu odlučiti što će se dogoditi.

Događaji koriste delegate da ponude sigurnu enkapsulaciju metode koja će biti pozvana kad se događaj okine. U sljedećem primjeru klasa **`TestButton`** sadrži događaj **`OnClick`**:

```
// Declare the handler delegate for the event
public delegate void ButtonEventHandler();

class TestButton
{
    // OnClick is an event, implemented by a delegate ButtonEventHandler.
    public event ButtonEventHandler OnClick;

    // A method that triggers the event:
    public void Click()
    {
        OnClick();
    }
}
```

Klasa koja koristi objekt klase TestButton može odabrati da reagira na taj događaj i definirati metodu koja će biti pozvana da obradi događaj:

```
// Create an instance of the TestButton class.
TestButton mb = new TestButton();

// Specify the method that will be triggered by the OnClick event.
mb.OnClick += new ButtonEventHandler(TestHandler);

// Specify an additional anonymous method.
mb.OnClick += delegate { System.Console.WriteLine("Hello, World!"); };

// Trigger the event
mb.Click();
```

Ovakav način interakcije objekata je tipičan za korisnička sučelja, ali je izuzetno koristan i korišten i u mnogim drugim slučajevima.

Zaključak

Ovaj seminar je obuhvatio tek dio onoga što programski jezik C# i .NET Framework pružaju. Sam framework sadrži tisuće klasa koje možete iskoristiti u svojim programima. Za detaljan opis svih klasa, njihovih metoda te mogućih načina upotrebe, trebale bi biti napisane tisuće i deseci tisuća stranica. Toliko ih zapravo već i napisano, i puno više, a neke knjige možete pogledati i u navedenoj literaturi.

I sam jezik C# nudi još puno elemenata i metoda koje vam mogu olakšati posao programiranja. Ovaj seminar je obuhvatio samo ono osnovno za ulazak u svijet programiranja sa C# jezikom. Posao programera zahtijeva konstantno učenje i usavršavanje, razumijevanje kako određene mogućnosti mogu olakšati život, ali u nekim slučajevima i otežati. Nažalost, samo razumijevanje programerskih ideja, algoritama i principa, iako nužno, predstavlja samo početni korak prema razvoju kvalitetnih i optimiziranih sustava.

Literatura

- MSDN, November 2005
- Inside C# 2nd Edition, Tom Archer & Andrew Whitechapel, Microsoft Press 2002
- Practical Guidelines and Best Practices for Microsoft Visual Basic and Visual C# Developers, Francesco Balena & Giuseppe Dimauro, Microsoft Press 2005
- Code Complete 2nd Editon, Steve McConnell, Microsoft Press 2004
- Extreme Programming Adventures in C#, Ron Jeffries, Microsoft Press 2004
- Introduction to Design Patterns in C#, James W. Cooper, 2002