Bevezetés a C# programozásba

1. Ismerkedés a Visual Studio fejlesztőrendszerrel és a C# nyelvvel 1

1.1. Mit használtunk a fenti programban? 4

2. A C# programok alapelemei 6

2.1 A C# nyelv egyszerű típusai 6

2.2 Változók és konstansok 7

2.3 Az enum és a string típusok 8

2.4 Értékadás 9

2.5 C# függvények (metódusok) hívása 10

2.6 Típus-átalakítások 10

2.6 Matematikai függvények - a System.Math osztály 11

2.7 Véletlen számok - a System.Random osztály 12

3. Operátorok és kifejezések 13

3.1 Aritmetikai operátorok 13

3.2 Precedencia és asszociativitás 14

3.3 Értékadó operátorok 15

3.4 Léptető (inkrementáló/dekrementáló) operátorok 15

4. A C# nyelv alapvető utasításai 16

4.1 Egyes és összetett utasítások 16

4.2 Szelekciós utasítások 16

4.3 Ciklusutasítások 19

4.4 Kivételkezelő utasítások 24

5. Tömbök használata 26

5.1 Egydimenziós tömbök 26

5.2 Két- és többdimenziós folytonos tömbök 27

5.3 A System.Array osztály 28

6. Metódusok 30

6.1 A metódusok definiálása 29

6.2 Különböző fajtájú paraméterek használata 32

6.3 Metódusok túlterhelése (átdefiniálása) 36

7. Felhasználói adattípusok(struct és class) 37

7.1 Osztályok definiálása 38

7.2 Objektumok létrehozása 38

7.3 Adatmezők az osztályokban 41

7.4 Az adatmezők inicializálása a konstruktorokban 44

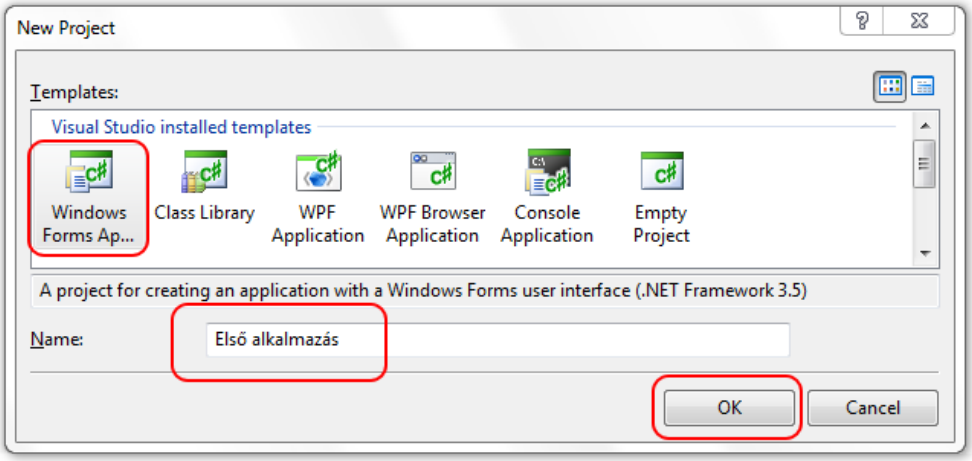
7.5 További metódusok készítése 45

7.6 Tulajdonságok és indexelők 47
7.7 Statikus osztálytagok 52
8. Objektum-orientált programozás 56
8.1 Osztályok származtatása, öröklés 56
8.2. Többalakúság (polimorfizmus) 61
8.3 Interfészek (interface) 67
F1. Bevezetés az objektum-orientált programozásba 68
F1.1 Bevezetés az objektum-orientált világba 69
F1.2 Alapelemek 69
F1.3 Objektum-orientált C# programpélda 72
F2. A C# nyelvű programokban gyakran használt osztályok 73

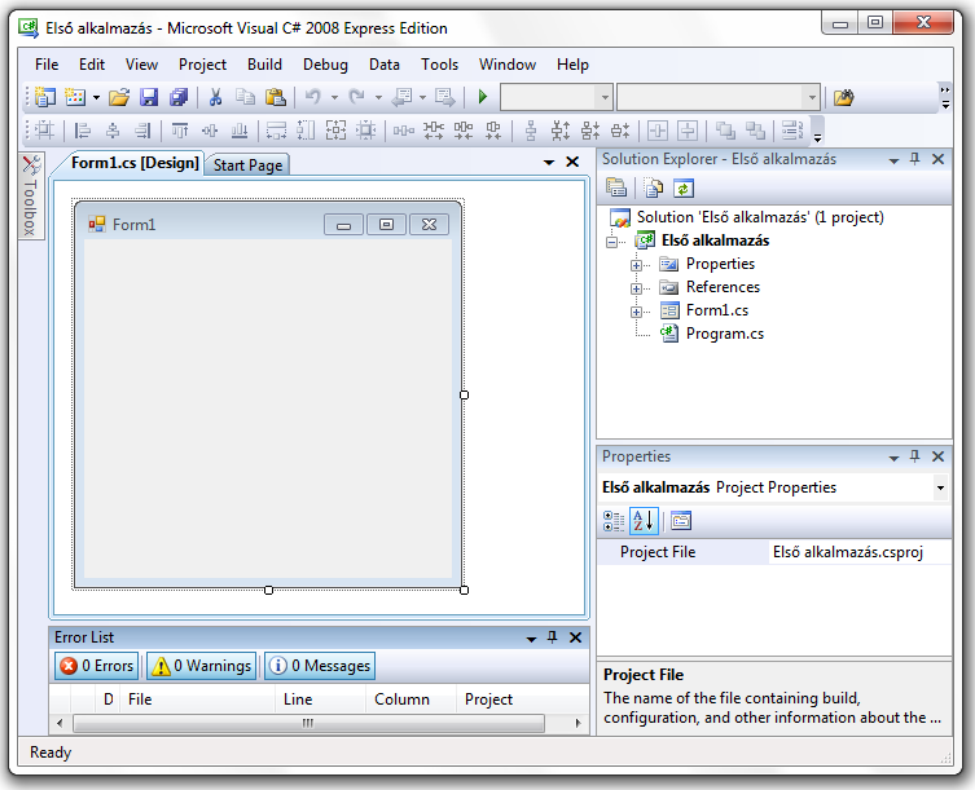
1. Ismerkedés a Visual Studio fejlesztőrendszerrel és a C# nyelvvel

Készítsünk programot, amely egy szövegmezőben vár egy nevet, majd egy gomb megnyomásakor megjelenő üzenetablakban üdvözli a felhasználót. Egy másik gomb megnyomásakor az alkalmazás befejezi a működését.

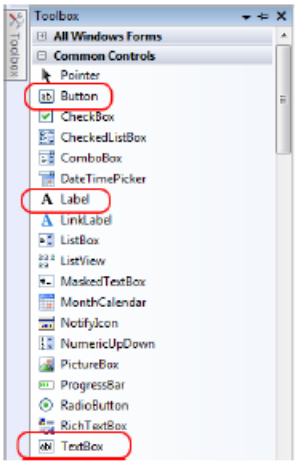
A programkészítés első lépése az alkalmazás elemeit összefogó projekt előállítása. Ehhez a File/New Project … menüválasztás hatására megjelenő ablakban megadjuk a projekt nevét (Name) és típusát (Windows Forms Application).



Az OK gomb megnyomása után létrejön a készítendő Windows Forms alkalmazás üres ablaka.

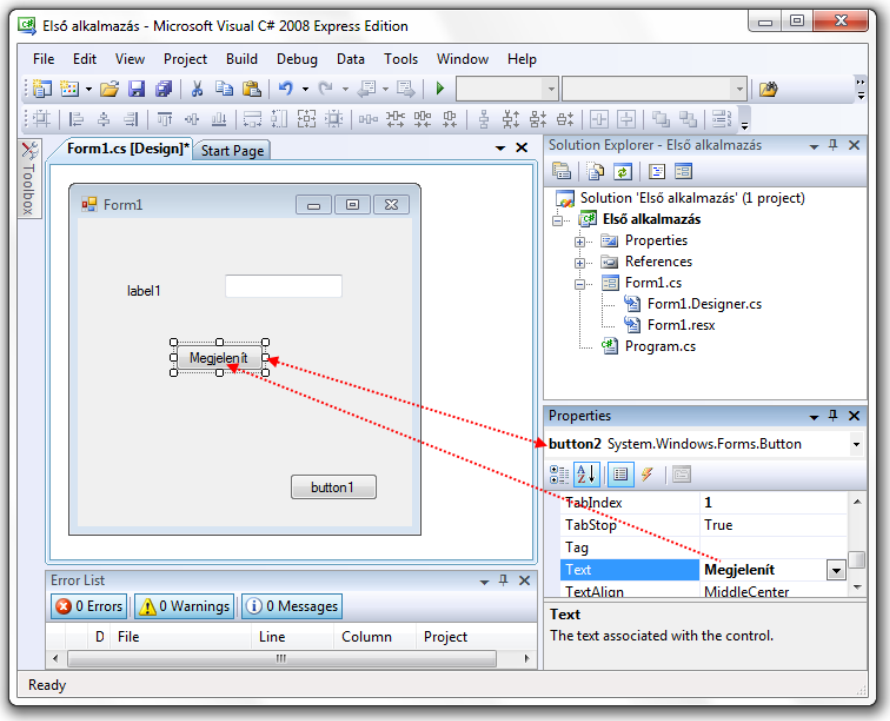


A formra kell helyeznünk a szükséges vezérlőelemeket, egy szövegmezőt (TextBox), egy címkét (Label) és két nyomógombot (Button) az eszköztárból (Toolbox). Ha a Toolbox nem látszik a képernyő bal oldalán, a View/Toolbox menüponttal behozhatjuk. A vezérlőket többféleképpen is felrakhatjuk a formra. A legegyszerűbb talán, ha kétszer kattintunk a vezérlő eszköztárban szereplő ikonján.



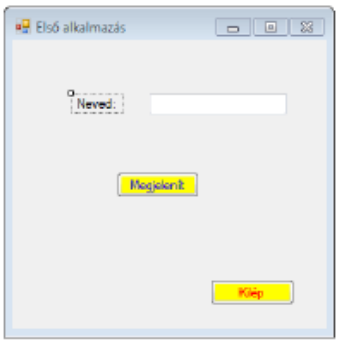
1. Az egér segítségével elrendezzük a formon a vezérlőket, majd pedig beállítjuk a tulajdonságaikat a Tulajdonságok ablakban (View/Properties Window). (Itt a form tulajdonságai is megadhatók.)

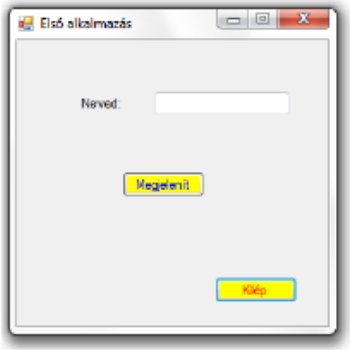
2.



Vezérlőtípus (osztály)	Vezérlő (objektum)	Text tulajdonság	BackColor tulajdonság	ForeColor tulajdonság	Font tulajdonság
Form1	this	Első alkalmazás			
TextBox	textBox1				
Label	label1	Neved:			
Button	button1	Megjelenít	Yellow	Blue	
Button	button2	Kilép	Yellow	Red	

A beállítások után az alkalmazás megtervezett ablaka:

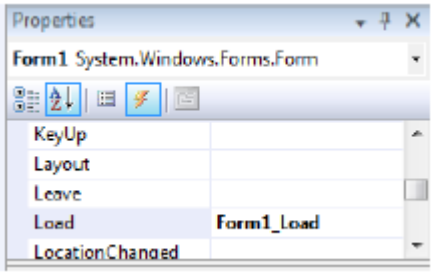




Az alkalmazás felülete futás közben is letesztelhető a Debug/Start Debugging menüválasztással, vagy a Ctrl+F5 billentyűkom-binációval.

3. A program kódját az eseménykezelő metódusokban kell megadnunk. Minden komponensnek (vezérlők és form) van egy ún. alapértelmezés szerinti eseménye (a form esetén a betöltés, nyomógombnál a kattintás, a szövegmezőnél a tartalom megváltozása stb.), amelybe bejutunk, ha a fejlesztés folyamán kétszer kattintunk a komponensen.

4. Más eseménykezelő is elkészíthető a vezérlő kiválasztása után, Properties ablak villám ikonjára kattintva, majd pedig kétszer kattintva a megfelelő esemény sorának jobb oldalára.



A form betöltésekor további tulajdonságokat is beállíthatunk:

```
private void Form1_Load(object sender, EventArgs e)
{
    textBox1.Text = "";
    this.BackColor = Color.LightGoldenrodYellow;
}
```

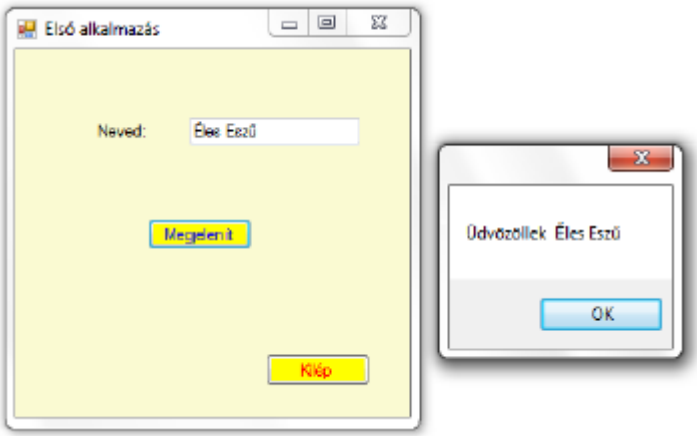
A Megjelenít gomb megnyomásakor üzenetablakban üdvözöljük a szövegmezőben megadott nevű egyént:

```
private void button2_Click(object sender, EventArgs e)
{
    string nev = textBox1.Text;
    MessageBox.Show("Üdvözöllek " + nev);
}
```

A Kilép gomb megnyomásakor befejezzük a program futását:

```
private void button1_Click(object sender, EventArgs e)
{
    Application.Exit();
}
```

A program futtatásakor először beírjuk a nevet a szövegmezőbe, majd pedig megnyomjuk a Megjelenít gombot:



Az elkészült programot a Form1.cs állományba íródik, ha használjuk a File/Save All és a File/Close Solution menüpontokat. A Form1.cs fájl tartalma:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace Első_alkalmazás
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            textBox1.Text = "";
            this.BackColor = Color.LightGoldenrodYellow;
        }

        private void button2_Click(object sender, EventArgs e)
        {
            string nev = textBox1.Text;
            MessageBox.Show("Üdvözöllek  " + nev);
        }

        private void button1_Click(object sender, EventArgs e)
        {
            Application.Exit();
        }
    }
}
```

Az alkalmazás megnyitásához a File/Open/Project/Solution menüpont használatával megjelenő ablakból a megoldás (solution, .sln) fájlt kell kiválasztanunk. Megnyitás után a munka az elmondottaknak megfelelően folytatható.

1.1. Mit használtunk a fenti programban?

A C# nyelv teljesen objektum-orientált nyelv. Ez azt jelenti, hogy a legegyszerűbb program készítése során is objektumokkal, illetve az objektumokat leíró típusokkal (osztályokkal - class) dolgozunk. Ilyen osztályok a fenti példában a Form1, a Color, a MessageBox stb., az objektumok pedig a nev, a this, a textBox1 nevekkel hivatkozott elemek. A Visual Studio rendszerben az osztályok neve nagybetűvel kezdődik és cián színűek, míg az objektumok neve kisbetűs és nem színezett.

Az objektumok rendelkeznek tulajdonságokkal (properties), melyekkel megváltoztathatjuk megjelenésüket, kinézetüket vagy a tárolt adataikat. A vizuális objektumok tulajdonságai a fejlesztői környezet Properties ablakában is módosíthatók, illetve megnézhetők. A C# programból egy értékadó utasítás bal oldalán megadva az objektum nevét és ponttal elválasztva a tulajdonság nevét, módosíthatjuk annak értékét. Az értékadás jobb oldalán használva ezt a kifejezést, lekérdezhetjük a tulajdonság értékét. (A tulajdonságok általában nagybetűvel kezdődnek - a példában aláhúzva szerepelnek.)

```
this.BackColor = Color.LightGoldenrodYellow
string nev = textBox1.Text;
```

Az objektumok viselkedését, működését, képességeit a kívülről (is) hívható metódusok (függvények) határozzák meg. A metódushívás utasításban az objektum nevét szintén ponttal elválasztva követi a metódus (nagybetűs) neve, azonban a sort üres kerek zárójelpár, vagy kerek zárójelek között megadott argumentumlista zárja.

```
MessageBox.Show("Üdvözöllek  " + nev);      // üzenetablak megjelenítése a megadott szöveggel
Application.Exit();                         // kilépés az alkalmazásból
```

Vannak olyan metódusok, amelyek magához az osztályhoz tartoznak, így őket az osztály nevét használva érjük el. A fenti példában csak ilyen, ún. statikus (static) metódusokat hívtunk.

Az eddigiekben a számunkra készen elérhető programelemekkel foglalkoztunk, azonban a legegyszerűbb program készítése során is el kell készítenünk egy ablakobjektum típusát, vagyis osztályát. Szerencsére a Visual Studio gondoskodik az osztály helyes felépítéséről, azonban már a C# programozással való ismerkedés legelején érdemes megismerkedni az osztály részeivel. A C# program főbb részeit kapcsos zárójelek közé kell zárni – így van ez az általunk készített osztályok és a metódusok esetén is. A kapcsos zárójelpárt megelőzi egy fejléc, ami megmondja, hogy mis is következik a kapcsos zárójelek között, vagyis mi lesz az adott definíció törzsében.

```
public partial class Form1 : Form
{
        public Form1()
        {
            // metódus, ami automatikusan lefut a Form1 típusú objektum létrehozásakor (konsturktor)
        }
        private void Form1_Load(object sender, EventArgs e)
        {
            // eseménykezelő metódus, a Form1 típusú ablak betöltődésekor (load) hívódik meg
        }
        private void button1_Click(object sender, EventArgs e)
        {
            // eseménykezelő metódus, ami a button1 gomb megnyomásakor (click) aktiválódik
        }
}
```

Az eseménykezelő metódusokat általában a Windows rendszer hívja, amikor a program használója a grafikus felületen bizonyos műveleteket végez (begépel egy szöveget, mozgatja az egeret, kattint az egérrel stb.)

Nagyon fontos megjegyeznünk, hogy végrehajtható utasításokat (pontosvesszővel lezárt értékadásokat és metódushívásokat stb.) csak metódusokban helyezhetünk el. A nem eseménykezelő metódusok készítésével és használatával a későbbiekben ismerkedünk meg.

A C# nyelven készített programok futtató rendszere a .NET, több ezer előre elkészített osztályt biztosít számunkra. Szerencsére ezeket csoportosítva érhetjük el, mintha egy lemez mappáiban lennének eltárolva. Egy osztály teljes nevéhez az elérési útvonala is hozzátartozik, amit névtérnek (namespace) nevezünk. Például, a Form osztály teljes neve a System.Windows.Forms.Form, így a névtere a System.Windows.Forms. Hogy ne kelljen a teljes nevet használnunk, a programunkat tároló fájl (modul) elején kijelölhetjük azokat a névtereket, amelyben keresheti a megadott osztályt a fordító, például:

```
using System.Windows.Forms;
```

Ezzel természetesen az összes itt tárolt osztály (Button, TextBox stb.) elérését is biztosítjuk.

Amikor egy új Windows Forms alkalmazást készítünk, a fejlesztőrendszer az ablakunk osztályát egy új névtérben hozza létre, melynek neve megegyezik megadott projektnévvel:

```
namespace Első_alkalmazás
{
    // típusdefiníciók
}
```

2. A C# programok alapelemei

A C# nyelvű programban a használni kívánt neveket mindig deklarálnunk kell, ezáltal adjuk meg a fordítónak a név tulajdonságait. A C# betűérzékeny, így a kis- és a nagybetűvel írt, azonos nevek különböző programelemeket azonosítanak. A névképzés szabályai megegyeznek az általánosan használtakkal, vannak foglalt szavak, a nevek betűvel kezdőnek stb. A leggyakrabban használt foglalt szavak:

bool, break, byte, case, catch, char, class, const, continue, decimal, default, do, double, else, enum, false, finally, float, for, foreach, if, int, long, namespace, new, null, object, out, override, params, private, protected, public, readonly, ref, return, sbyte, short, sizeof, static, string, struct, switch, this, throw, true, try, uint, ulong, ushort, using, virtual, void, while

(Érdekes megemlíteni, hogy saját célra akár foglalt szót is használhatunk, ha a szó elé tesszük a @ karaktert, például: @class, valamint azt, hogy betűk alatt unicode betűket kell értenünk, vagyis magyar ékezetes magánhangzókat is alkalmazhatunk, például: GyökKeresés().)

2.1 A C# nyelv egyszerű típusai

A C# nyelvben a típusokat két csoportra, az érték- és hivatkozás-típusok csoportjára oszthatjuk. Az első csoport esetén elemeivel deklarált (definiált) változók tárolják az értéküket, míg a referencia-változók csupán hivatkoznak az értéket tároló memóriaterületre. Az értéktípusokhoz tartoznak az egyszerű, felsorolt (enum) és a struktúra (struct) típusok, míg a referencia-típusokat többek között az osztályok (class) és a tömbök alkotják. Az alaptípusokat táblázatban csoportosítottuk:

Előjeles egészek	sbyte, short, int, long
Előjel nélküli egész típusok	byte, ushort, uint, ulong
Unicode karakterek típusa	char
Lebegőpontos típusok	float, double
Nagypontosságú decimális típus	decimal
Logikai típus	bool

Az alábbiakban részletes információk találhatók az előre definiált egyszerű és referencia típusokról:

Egész típusok

Típus	***.NET típus***	Ábrázolás	Értékkészlet	Értékutótag
sbyte	***System.Sbyte***	8-bites, előjeles	-128 … 127	
byte	***System.Byte***	8-bites, előjel nélküli	0 … 255	
short	***System.Int16***	16-bites, előjeles	-32768 … 32767	
ushort	***System.Uint16***	16-bites, előjel nélküli	0 … 65535	
char	***System.Char***	16-bites, előjel nélküli, Unicode	'\u0000' … '\uffff'	
int	***System.Int32***	32-bites, előjeles	-2147483648 … 2147483647	
uint	***System.Uint32***	32-bites, előjel nélküli	0 … 4294967295	u, U
long	***System.Int64***	64-bites, előjeles	-9223372036854775808 … 9223372036854775807	l, L
ulong	***System.Uint64***	64-bites, előjel nélküli	0 … 18446744073709551615	ul, UL, lu, LU

Lebegőpontos típusok

Típus	.NET típus	Ábrázolás	Értékkészlet	Pontosság (jegy)	Érték-utótag
float	System.Single	32-bites	±1.5 × 10−45 … ±3.4 × 1038	7	f, F
double	System.Double	64-bites	±5.0 × 10−324 … ±1.7 × 10308	15-16	(d, D)
decimal	System.Decimal	128-bites	±1.0 × 10−28 … ±7.9 × 1028	28-29	m, M

Logikai típus

Típus	.NET típus	Ábrázolás	Értékkészlet
bool	System.Boolean	8-bites	true, false

Referencia típusok

Típus	.NET típus	Leírás
object	System.Object	Minden előre definiált és felhasználó által készített típus a System.Object osztálytól származik.
string	System.String	Nem módosítható unicode karaktersorozatok típusa.

Megjegyzések a típusokhoz:

bool — Logikai értéket tárol, mely igaz (true), vagy hamis (false) lehet.

char — A char típus a 16-bites Unicode kódolási szabványt támogatja.

decimal — A decimális típus 28-29 jegy pontossággal tárol valós számokat, elsősorban pénzügyi számításokhoz.

2.2 Változók és konstansok

Más nyelvekhez hasonlóan adatainkat típusos memóriaterületeken, ún. változókban tárolhatjuk, melyek értéke módosítható. A megváltoztatni nem kívánt adatainkhoz is nevet rendelhetünk, ezek a konstansok.

2.2.1 Változók deklarálása

Az értéktípusú változók egyedi névvel rendelkeznek, míg másokra névvel ellátott referenciákkal hivatkozunk.

Értéktípusú változó létrehozása:

```
int a=12;
double b, c=23.1;
```

Hivatkozott változó készítése:

```
object d; // csak egy referencia jön létre
d = new object(); // elkészül a hivatkozott objektum (változó)

// ugyanez egyetlen lépésben

object d = new object();
```

A C# programok osztálydefiníciókból állnak (Javasolt, hogy az osztályok nevét nagybetűvel kezdjük.) Az osztályok változókat (mezőket) és metódusokat tartalmaznak. A metódusok (azaz függvények) tárolják azokat az utasításokat, melyek végrehajtásával különböző feladatokat oldhatunk meg. A C# programszerkezetének megfelelően a változókat alapvetően két csoportra oszthatjuk, a lokális változók és az adatmezők csoportjára:

namespace WindowsFormsApplication

```
namespace WindowsFormsApplication
{
    public partial class Form1 : Form
    {
        int mezo1, mezo2 = 12;

        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            int v1, v2 = 23;
            {
                double v3 = 0;
            }
        }

    }
}
```

Az adatmező-definíciók (mezo1, mezo2) az osztályon belül, azonban a metódusokon kívül helyezkednek el, míg a lokális változókat a metódus törzse (v1, v2), vagy egy belső összetett utasítás (blokk) (v3) tartalmazza. Fontos megjegyeznünk, hogy valamely külső szinten definiált lokális változó nevével újabb változót nem hozhatunk létre belsőbb szinteken. Így a példában nem készíthetünk változót v1 és v2 nevekkel legbelső blokkban.

A C# nyelv lehetővé teszi a változók kezdőértékkel való ellátását a deklarációban. A kezdőérték nélküli mezőváltozók területe törlődik, nullaértékű bájtokkal töltődik fel, míg a kezdőérték nélküli lokális változók tartalma definiálatlan lesz. (Az ilyen változók értékének használata fordítási hibához vezet.)

2.2.2 Konstansok használata

A kezdőértékkel ellátott adatmezőket és a lokális változókat a const szóval konstanssá tehetjük, azonban a mezőnevek esetén a readonly módosítóval is célt érhetünk.

const double pi = 3.14259265;

readonly float verzio = 1.23;

A kétféle megoldás között a legfontosabb különbség, hogy a **const** konstansokat a fordító értékeli ki, míg a **readonly** mezők a program futása során kapnak értéket, így akár a konstruktorban is inicializálhatók.

2.3 Az enum és a string típusok

Mielőtt továbbhaladnánk, meg kell ismerkednünk néhány olyan típussal, melyekkel már a legegyszerűbb programokban is találkozhatunk.

2.3.1 Az enum típus

Egész típusú konstansainkat nemcsak egyesével, hanem csoportosan is megadhatjuk, amennyiben felsorolásokat készítünk. Az enum típusok definíciói a névtérben és az osztályban egyaránt elhelyezkedhetnek.

Az enum kulcsszó után a felsorolástípus azonosítóját kell megadnunk, melyet kettősponttal elválasztva a tagok típusa követhet. A definícióban kapcsos zárójelek között, vesszővel elválasztva adjuk meg a konstansneveket. A felsorolt nevekhez 0-tól egyesével sorszámok rendelődnek, azonban magunk is adhatunk értéket a felsorolás tagjainak.

```
// Halk=0, Normal=1, Hangos=2
enum Hangero { Halk, Normal, Hangos };
enum Hangero { Halk = 1, Normal, Hangos }; // Normal=2, Hangos=3
enum Intervallum : int { Max = 122334, Min = 234 };
```

A felsorolástípus nevével változókat deklarálhatunk, míg a típus elemeit a típusnév után ponttal elválasztva érjük el:

```
Hangero Zene = Hangero.Hangos;
```

A felsorolás tagjaihoz rendelt egész értékeket típus-átalakítás segítségével kapjuk meg:

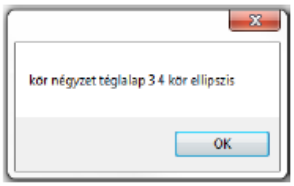
```
int Enek = (int)Hangero.Halk;
```

Ha a felsorolás elemei között logikai műveleteket szeretnénk végezni, akkor az enum elé a [Flags] attribútumot kell helyezni – ekkor a felsorolás elemei nem egész értékekhez, hanem bitekhez kötődnek.

```
[Flags] enum alakzat {kicsi = 1, négyzet = 2, közepes = 4, téglalap = 5, nagy = 7, kör = 9};
// ...
alakzat k = alakzat.kicsi | alakzat.négyzet;
```

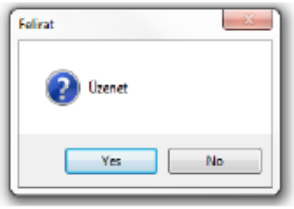
Az alábbi példában a felsorolás első elemétől az utolsó eleméig a for ciklus segítségével haladunk végig, egyesével lépkedve. A nyomógomb megnyomásakor az eredmény egy üzenetablakban jelenik meg:

```
enum alakzat { négyzet = 1, téglalap, kör = 5, ellipszis };
private void button1_Click(object sender, EventArgs e)
{
    string s = alakzat.kör + " ";
    for (alakzat i = alakzat.négyzet; i <= alakzat.ellipszis; i++)
        s += i.ToString() + " ";
    MessageBox.Show(s);
}
```



Általában saját felsorolást csak nagyobb programok esetén készítünk, azonban a .NET rendszer által definiált konstans értékek egy részét felsorolások elemeként érhetjük el. Példaként tekintsük az üzenetablak megjelenítésének kibővített formáját!

```
DialogResult valasztas = MessageBox.Show("Üzenet", "Felirat", MessageBoxButtons.YesNo,
                                                               MessageBoxIcon.Question);
if (valasztas == DialogResult.Yes)
    MessageBox.Show("Igen!");
```



A példában szereplő MessageBoxButtons, MessageBoxIcon és DialogResult típusnevek felsorolásokat takarnak.

2.3.2 A string típus

A alaptípusok elsősorban numerikus adatok tárolását segítik, azonban a programok nagy része szövegeket tárol és kezel. A C# a .NET rendszer System.String osztályára definiálja a string típust, melynek objektumait nem módosítható szövegek tárolására használhatjuk. A programban a szöveges konstansainkat kettős idézőjelek között kell megadnunk:

```
System.String s1 = "Programozás C# nyelven";
string nev1 = "C:\\Munka\\ZHGyakorló";
string nev2 = @"C:\Munka\ZHGyakorló";
```

A string típus lehetővé teszi a szövegek között a + és a += szövegösszefűzés operátorok használatát. A szöveg adott indexű karakterét a [ ] operátor adja vissza, az első karakter indexe 0. A @ karakter segítségével kiküszöbölhetjük a szövegen belüli „érzékeny” karakterek, mint például az Escape-szekvenciák hatását. Néhány Escape-szekvencia:

Escape-szekvencia	*Jelentés*
\a	veszélyjelző hang
\n	új sor
\t	vízszintes tabulátor
\\	fordított perjel
\'	aposztróf
\"	idézőjel

2.4 Értékadás

A változók általában értékadás során kapnak értéket, melynek általános alakja:

```
változó = érték;
```

Az értékadás operátorának (=) bal és jobb oldalán egyaránt szerepelhetnek kifejezések, melyek azonban különböznek egymástól. A bal oldalon álló kifejezés azt a területet jelöli ki a memóriában, ahová a jobb oldalon megadott kifejezés értékét be kell tölteni. Nézzünk néhány példát az értékadásra!

```
int x;
x = 7;
x = x + 5;
x += 11;
```

x = 7 Az x változó kijelöli azt a tárterületet, ahová a jobb oldalon megadott konstansértéket (7) be kell másolni.

x = x + 5; Az értékadás során az x változó az egyenlőségjel mindkét oldalán szerepel. A bal oldalon álló x változó kijelöli azt a memóriaterületet, ahová a futtató rendszer behelyezi a jobb oldalon álló kifejezés értékét. Vagyis az értékadás előtt az x pillanatnyi értékét (7) összeadja az 5 konstansértékkel, majd az eredményt az x-ben tárolja. Ezáltal az x korábbi 7 értéke felülíródik 12-vel.

x += 11; Amennyiben az értékadás célja az adott változó értékének művelettel való módosítása, használhatjuk összetett értékadásokat. A kifejezés jelentése, „növeld meg x értékét 11-gyel”. A művelet után x 23 lesz.

2.5 C# függvények (metódusok) hívása

Néhány általános szabályt meg kell ismernünk a függvényekkel (metódusokkal) kapcsolatosan, mielőtt rátérnénk a különböző osztályok használatának bemutatására.

Minden metódusnak van neve, és vannak paraméterei, melyeken keresztül megkapja azokat az adatokat, melyekkel dolgozik a függvény törzsében. A paraméterek kerek zárójelben vannak, a kerek nyitó és záró zárójelet még akkor is ki kell tenni, ha a metódus nem rendelkezik paraméterrel. A metódus törzsét a nyitó és záró kapcsos zárójel fogja közre.

A metódus jellemzője a visszatérési értéke, melynek típusát a metódusfejben, a neve előtt kell megadni. A visszatérési értéket a return utasításban adjuk meg. Az elmondottakat jól szemlélteti az alábbi kis metódus:

```
int fgnev(int a)
{
int b = a;
. . .
return b;
}
```

A metódus hívásához a nevét, valamint az egyes paramétereknek megfelelő típusú argumentumok listáját használjuk. A metódusnév után a kerek zárójelpár megadása akkor is kötelező, ha a függvény nem rendelkezik paraméterrel. A fenti függvény hívását az alábbiak szerint végezhetjük el:

```
int x, y = 12;
x = fgnev(y);
```

Bizonyos metódusok az osztályokhoz tartoznak, vagyis objektumok létrehozása nélkül is hívhatók, Osztálynév.Metódus(); formában. Ezek a metódusok az ún. statikus (static) tagfüggvények. Más metódusokat azonban csak az objektumpéldány létrehozását követően érhetünk el objektumnév.Metódus(); alakban. Szükség esetén az ob-jektumpéldányok egyszerűen elkészíthetők:

Osztály objektum = new Osztály();

ahol a new műveleti jel után egy speciális metódust, az ún. konstruktort hívjuk.

2.6 Típus-átalakítások

Már a legegyszerűbb programokban is találkozhatunk a különböző típusok közötti konverzió igényével.

2.6.1 Numerikus adatok konverziója

Mint ismert, a C# erősen típusos nyelv, ezért a fordító semmilyen olyan műveletet nem fordít le, amely adatvesztéssel járhat. Tipikus példa erre a valós adatok egész változóba való töltése.

A fordító bizonyos átalakításokat azonban automatikus elvégez – ezeknél garantált az előjel és az érték megőrzése. Ilyen, implicit (automatikusan végrehajtódó) numerikus típus-átalakításokat láthatunk az alábbi példában:

```
byte xx = 12;
int yy = xx;
float ff = 23.2F; // A 23.2 konstans típusa alaphelyzetben double, ezért kell a típusjelző F utótag
double dd = ff;
ulong nn = 0x1234567890UL;
double hh = nn;
```

Amennyiben az implicit konverzió nem hajtható végre, explicit típus-átalakítással – (típus) – kérhetjük a konverziót a fordítótól:

```
int yy = -123;
byte xx = (byte)yy; // 133
double dd = 19.791223800729E-1; // 1.9791223800729
float ff = (float)dd; // 1.979122
ulong nn = (ulong)dd; // 1
```

2.6.2 Numerikus adatok és karaktersorozatok közötti típus-átalakítások

Mivel a .NET az operációs rendszerrel szöveges adatok segítségével kommunikál (beolvasás, kiírás, komponensek tartalma stb.), fontos megismernünk az átalakítással kapcsolatos lehetőségeket.

A szöveggé (sztringgé) alakítás minden .NET osztály jellegzetessége, hiszen mindegyik örökli az object osztálytól a ToString() metódust. Ez automatikusan meghívódik minden olyan esetben, amikor string típusú operandusra, argumentumra van szükség:

```
int eredmeny = 122334;
string szoveg = "" + eredmeny;

int eredmeny = 122334;
MessageBox.Show(eredmeny.ToString());
```

Ennél több ismeretre csak akkor van szükségünk, ha formázottan kívánjuk az adatainkat szöveggé alakítani. Nézzünk néhány, a String osztály Format() és az osztályok ToString() metódusaiban egyaránt alkalmazható formátumelemet!

```
string s = String.Format("{0:000.000}; {1:###.###}", 12.23, 12.23); // "012,230; 12,23"
s = String.Format("{0,10:F2}", 123);   // "    123,00"
s = String.Format("{0,-10:F2}", 123);  // "123,00    "

int a = 123; s = a.ToString("00###");  // "00123"
s = a.ToString("X5");                  // "0007B"
a = 123456; s = a.ToString("N");       // "123 456,00"
s = a.ToString("N0");                  // "123 456"
double d = 12.3; s = d.ToString("F3"); // "12.300"
s = d.ToString("E3");                  // "1,230E+001"
```

Sokkal több figyelmet igényel a különböző típusú adatok szövegből való kinyerése. A továbbiakban teljesség igénye nélkül, csak néhány jól használható megoldás bemutatására szorítkozunk.

Általános megoldást biztosítanak a Convert osztály ToTípus() formájú, statikus metódusai:

```
double sugar = Convert.ToDouble("123,34");
uint sortav = Convert.ToUInt32("46");
```

Felhívjuk a figyelmet, hogy alaphelyzetben, a Windows országfüggő beállításai határozzák meg, hogy tizedes pontot vagy tizedes vesszőt kell használnunk a szövegben tárolt valós számok esetén.

További megoldáshoz jutunk, ha a párhuzamos típusosztályok statikus Parse() metódusát használjuk:

```
double sugar = Double.Parse("123,34");
uint sortav = UInt32.Parse("46");
```

2.6 Matematikai függvények - a System.Math osztály

Az alábbiakban felsorolunk néhány fontos matematikai függvényt, a visszatérési érték típusának jelzésével.

Használat	*Típusa*	*Metódus*
abszolút érték képzése	**double**	Abs(**double** x)
abszolút érték képzése	**int**	Abs(**int** x)
szög koszinusza	**double**	Cos(**double** x)
szög szinusza	**double**	Sin(**double** x)
szög tangense	**double**	Tan(**double** x)
természetes alapú logaritmus	**double**	Log(**double** x)
e^x	**double**	Exp(**double** x)
hatványozás (x^y)	**double**	Pow(**double** x, **double** y)
négyzetgyök	**double**	Sqrt(**double** x)
kerekít a legközelebbi egész számra	**double**	Round(**double** x)
kerekít a legközelebbi valós értékre, az adott tizedes jegyig.	**double**	Round(**double** x, **int** jegy)

A **Math.E** az e értékét, míg a **Math.PI** pedig π értékét adja meg konstansként.

Néhány példa a matematikai függvények hívására:

gyökvonás	`y = Math.Sqrt(x); vagy  y = Math.Pow(x,0.5);`
koszinusz	`y = Math.Cos(x);`
szinusz	`y = Math.Sin(x);`
tangens	`y = Math.Tan(x);`
hatványozás a^n	`y = Math.Pow(a,n); vagy y = Math.Exp(Math.Log(x)*n));`
exp függvény e^x	`y = Math.Exp(x);`
természetes logaritmus ln x	`y = Math.Log(x);`

2.7 Véletlen számok - a System.Random osztály

Programok készítése során gyakran lehet szükségünk automatikusan előállított egész vagy valós számokra, például teszteléshez. A .NET alatt külön osztály (Random) áll rendelkezésünkre, mellyel ún. véletlen számokat készíthetünk. A nehézséget csupán az okozza, hogy a Random osztály metódusai nem statikusak, így elérésükhöz objektumpéldányt kell létrehoznunk:

```
Random veletlen1 = new Random();
Random veletlen2 = new Random(123);
```

Az első esetben a véletlen számok sora véletlenszerűen indul, míg a második formát használva mindig ugyanazt a számsort kapjuk.

A **Random** osztály **Next()** metódusainak hívásával mindig újabb véletlen egész számot kapunk. A különbség az egyes hívási formák között, hogy milyen intervallumban helyezkedik el az eredmény:

```
int rn;
rn = veletlen1.Next();          // intervallum: [0, 2147483647)
rn = veletlen1.Next(123);       // intervallum: [0, 123)
rn = veletlen1.Next(12, 23);    // intervallum: [12, 23)
```

Szükség esetén a [0.0, 1.0) intervallumba eső valós véletlen számot is előállíthatunk, a NextDouble() metódus hívásával. Megjegyezzük, hogy a NextBytes() metódus segítségével bájttömböt tölthetünk fel véletlen számokkal.

3. Operátorok és kifejezések

A C# nyelvben a legnépesebb utasításcsoportot a pontosvesszővel lezárt kifejezések (értékadás, metódushívás, léptetés és új objektum létrehozása) alkotják. A kifejezések egyetlen operandusból, vagy operandusok és műveleti jelek (operátorok) kombinációjából épülnek fel.

Az operandusok a C# nyelv azon elemei, amelyeken az operátorok kifejtik a hatásukat. Az operandus lehet konstans érték, azonosító, sztring, metódushívás, tömbindex kifejezés, tagkiválasztó kifejezés (ezek az ún. elsődleges kifejezések) és tetszőleges összetett kifejezés, amely zárójelezett vagy zárójel nélküli, operátorokkal összekapcsolt további operandusokból áll.

Az operátorokat több szempont szerint csoportosíthatjuk. Az egyoperandusú (unary) operátorok esetén a kifejezés általános alakja:

op operandus vagy operandus op

Az első esetben, amikor az operátor (op) megelőzi az operandust, előrevetett (prefix), míg a második esetben hátravetett (postfix) alakról beszélünk.

int n = 7;

n++;

n értékének növelése (postfix),

--n;

n értékének csökkentése (prefix),

(double)n

n értékének valóssá alakítása.

Az operátorok többsége két operandussal rendelkezik, ezek a kétoperandusú (binary) operátorok:

operandus1 op operandus2

int n = 7;

n + 2

n + 2 kiszámítása,

n << 3

n bitjeinek eltolása 3 pozícióval balra,

n += 5;

n értékének növelése 5-tel.

A kifejezések kiértékelése a bennük szereplő műveletek precedenciája alapján történik. Amennyiben több azonos elsőbbségű operátorral van dolgunk, akkor általában balról-jobbra haladva mennek végbe a műveletek, kivételt csak az értékadó operátorok képeznek. A leggyakrabban használt műveletek:

Műveletek

Asszociativitás

Operátorok

elsődleges

balról-jobbra

x.y, f(x), a[x], x++, x--, new
egyoperandusú
balról-jobbra
+, -, !, ~, ++x, --x, (T)x
multiplikatív
balról-jobbra
*, /, %
additív
balról-jobbra
+, -
biteltoló
balról-jobbra
<<, >>
összehasonlító
balról-jobbra
<, >, <=, >=
azonosságvizsgálat
balról-jobbra
==, !=
logikai ÉS
balról-jobbra
&
logikai kizáró VAGY
balról-jobbra
^
logikai VAGY
balról-jobbra
|
feltételes ÉS
balról-jobbra
&&
feltételes VAGY
balról-jobbra
||
feltételes
jobbról-balra
?:

értékadó

jobbról-balra

=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=

A táblázatban a műveletek precedenciája (elsőbbsége) fentről lefele haladva csökken.

3.1 Aritmetikai operátorok

14

A műveletek csoportosítását a mindenki által jól ismert aritmetikai operátorokkal kezdjük. Az aritmetikai operátorokat a műveletek elsőbbsége (precedenciája) szerint soroltuk fel, az azonos szintű precedencia azonos sorszám alatt jelenik meg. A négy alapműveletet kiegészítik az előjelek, valamint az egészosztás maradék operátora:

1.

+

+ előjel (identitás)

-

- előjel (negálás)

2.

*

szorzás

/

osztás

%

maradékképzés

3.

+

összeadás

-

kivonás

3.2 Precedencia és asszociativitás

A kifejezések az elsőbbségi (precedencia) szabályok szerint kerülnek kiértékelésre. Ezek a szabályok meghatározzák a kifejezésekben szereplő, különböző elsőbbséggel rendelkező műveletek végrehajtási sorrendjét. Az azonos elsőbbségi operátorok esetén a balról-jobbra, illetve a jobbról-balra asszociativitás ad útmutatást.

3.2.1 A precedencia-szabály

Ha egy kifejezésben különböző precedenciájú műveletek szerepelnek, akkor mindig a magasabb precedenciával rendelkező operátort tartalmazó részkifejezés értékelődik ki először.

Zárójelezéssel az elsőbbséget megváltoztathatjuk. Az

edcba**□□

és az

□□)*(*edcba□□

kifejezések kiértékelési sorrendje megegyezik, amennyiben a változók deklarációja:

int a = 3, b = 5, c = 6, d = 9, e = 2;

1. b * c -> 30

2. d * e -> 18

3. a + b * c = a + 30 -> 33

4. a + b * c - d * e = 33 - 18 -> 15

A kiértékelés sorrendje a matematikából ismert zárójelek segítségével megváltoztatható. Megjegyezzük, hogy a C# nyelvben csak a kerek zárójel () használható, bármilyen mélységű zárójelezést is hajtunk végre. Az (a+b)*(c-d)*e kifejezés kiértékelésének lépései:

int a = 3, b = 5, c = 6, d = 9, e = 2;

1. (a + b) -> 8

2. (c - d) -> -3

3. 8 * -3 -> -24

4. -24 * 2 -> -48

3.2.2 Az asszociativitás szabály

Az asszociativitás határozza meg, hogy az adott precedenciaszinten található műveleteket balról-jobbra vagy jobbról-balra haladva kell elvégezni.

Az értékadó utasítások csoportjában a kiértékelés jobbról-balra halad, ami lehetővé teszi a több változó együttes értékadását:

a = b = c = 0; azonos az a = (b = (c = 0));

Amennyiben azonos precedenciájú műveletek szerepelnek egy aritmetikai kifejezésben, akkor a balról-jobbra szabály lép életbe. Az alábbi kifejezés kiértékelése három azonos elsőbbségű művelet végrehajtásával indul.

a + b * c / d * e

Az asszociativitás miatt a kiértékelési sorrend:

double a = 3, b = 5, c = 12, d = 10, e = 2;

1. b * c -> 60

2. b * c / d =60 / d -> 6

3. b * c / d * e = 6 * e -> 12

4. a + b * c / d * e = a+ 12 -> 15

Matematikai képletre átírva jól látszik a műveletek sorrendje:

edcba□□□

Ha a feladat az

edcba□□□

képlet programozása, akkor ezt kétféleképpen is megoldhatjuk:

1. Zárójelbe tesszük a nevezőt, így mindenképpen a szorzattal osztunk:

a + b * c / (d * e)

2. Az is jó megoldás, ha a nevezőben szereplő szorzat mindkét operandusával osztunk:

a + b * c / d / e

3.3 Értékadó operátorok

Az értékadó műveletek az operátorok leggyakrabban használt csoportját képezik. A művelet során az operátor jobb oldalán álló kifejezés értéke kerül kapcsolatba a bal oldalon álló változóval. Egyszerű értékadás esetén a változó felveszi a kifejezés értékét, míg összetett értékadás során a változó aktuális értéke és a kifejezés értéke között egy művelet is végbemegy, melynek eredménye kerül a változóba.

double a = 3, b, c, e1, e2;

c = 3; // c <- 3

a = (b = 6) / 4 * c; // a <- 4.5, b <- 6

e1 = e2 = 2; // e1 <- 2, e2 <- 2

e1 /= a + 2 - b; // e1 <- 4

e2 += 4 * c - 3; // e2 <- 11

Megjegyzések:

- Az a = (b = 6) / 4 * c; kifejezésben az a és a b is értéket kap.
- A jobbról-balra szabály lehetővé teszi több változó ugyanazon értékre való állítását: e1 = e2 = 2;
- Ha egy változó a jobb és a bal oldalon is szerepel, akkor a kifejezés tömörebben is felírható úgy, hogy a műveleti jelet az egyenlőség elé tesszük (szóköz nélkül).

e1 = e1 / (a + 2 - b); e1 /= a + 2 - b;

Az összetett értékadás műveletei: +=, -=, *=, /=, %=, &=, |=, ^=, <<=,>>=

3.4 Léptető (inkrementáló/dekrementáló) operátorok

Valamely numerikus változó eggyel való növelése, illetve csökkentése hagyományos formája helyett érdemes a léptető operátort alkalmazni:

i = i+1; és i += 1; helyett i++; vagy ++i;

i = i-1; és i -= 1; helyett i--; vagy --i;

++i; --i; prefixes alakok

i++; i--; postfixes alakok

Amennyiben az előrevetett (prefixes) operátort kifejezésben használjuk, akkor a léptetés a kifejezés kiértékelése előtt megy végbe! A hátravetett (postfixes) forma esetén a léptetés a kifejezés kiértékelése után következik be, így a kifejezésben az operandus az eredeti értékével szerepel.

int k = 1, m = 7, n;

n = ++k - m--; // k 2, m 6 és n -5 lesz

4. A C# nyelv alapvető utasításai

A C# nyelv utasításai a program futása során hajtódnak végre. A legnépesebb utasításcsoportot a pontosvesszővel lezárt kifejezések (értékadás, metódushívás, léptetés és új objektum létrehozása) alkotják. További utasításcsoportok a szelekciós, az iterációs, a vezérlésátadó és a kivételkezelő utasítások csoportja.

A szelekciós utasítások az egyszerű, illetve többszörös elágazást lehetővé tevő if és switch utasításokat jelentik. Az iterációs utasítások csoportja a C# nyelv ciklusutasításait (while, for, foreach, do while) tartalmazza. A vezérlésátadó utasítások egy része a fenti két csoport elemeivel együtt használható (break, continue, goto case xxx, goto default), azonban néhány önállóan is alkalmazható (goto, return).

4.1 Egyes és összetett utasítások

A metódusok lokális változói és konstansai utasításokban jönnek létre. A mellékhatással rendelkező kifejezések utasítássá válnak, ha pontosvesszőt (;) helyezünk mögéjük – ezek a kifejezés-utasítás. Felhívjuk a figyelmet, hogy ebbe a csoportba tartoznak a metódushívások is:

```
private void button1_Click(object sender, EventArgs e)
{
const int ezer = 1000; // konstans-deklaráció
int k = 12, i = 23; // változó-deklarációk
k++; // kifejezés-utasítás
i += 7; // kifejezés-utasítás
MessageBox.Show("" + (k * i + ezer)); // kifejezés-utasítás
}
```

Egyetlenegy pontosvesszőből (;) áll az ún. üres utasítás, amely utasítást helyettesít, amennyiben mást nem akarunk végrehajtani azon a helyen.

Kapcsos zárójelekkel { } több utasítást egyetlen utasítássá vonhatunk össze. A szelekciós és iterációs utasítások alaphelyzetben csupán egyetlen tevékenységet hajtanak végre. A kapcsos zárójelekkel összefogott utasításblokk mindenhol elhelyezhető, ahol utasítás szerepel a C# nyelv leírásában:

```
{
int a = 7;
a += 12;
;
}
```

4.2 Szelekciós utasítások

A szelekciós utasításokkal (if, switch) feltételtől (kifejezés értékétől) függően jelölhetjük ki a program további futásának lépéseit. Megvalósíthatunk leágazást, kettéágaztatást vagy többirányú elágaztatást. A szelekciókat egymásba is ágyazhatjuk. A feltételek megfogalmazásánál a matematikából jól ismert összehasonlító (relációs) és feltételes műveleteket használjuk.

4.2.1 Relációs és feltételes logikai műveletek

Az összehasonlításokban két kifejezést relációs műveleti jellel kapcsolunk össze. A művelet elvégzésének eredménye bool típusú true (igaz) vagy false (hamis).

kifejezés1 > kifejezés2

akkor igaz, ha a kifejezés1 nagyobb, mint a kifejezés2,

kifejezés1 >= kifejezés2

akkor igaz, ha a kifejezés1 nagyobb, vagy egyenlő, mint a kifejezés2,

kifejezés1 < kifejezés2

akkor igaz, ha a kifejezés1 kisebb, mint a kifejezés2

kifejezés1 <= kifejezés2

akkor igaz, ha a kifejezés1 kisebb, vagy egyenlő, mint a kifejezés2,

kifejezés1 == kifejezés2

akkor igaz, ha a kifejezés1 egyenlő a kifejezés2-vel,

kifejezés1 != kifejezés2

akkor igaz, ha a kifejezés1 nem egyenlő a kifejezés2-vel.

A feltételes logikai operátorokat bool típusú kifejezéseket összekapcsoló feltételekben használjuk. Az operátorokat a precedencia csökkenő sorrendjében mutatjuk be:

tagadás negáció

! kifejezés

A kifejezés logikai értékét ellentettjére változtatja.

feltételes ÉS konjunkció

kifejezés1 && kifejezés2

Csak akkor ad igaz eredményt, ha mindkét kifejezés értéke igaz.

feltételes VAGY diszjunkció

kifejezés1 || kifejezés2

Akkor igaz, ha bármelyik kifejezés igaz értékű.

Példák a relációs műveletekre és a feltételes logikai operátorok használatára:

x % 2 == 0

akkor igaz, ha az x egész változó 2-es osztási maradéka (modulusa) zérus, azaz 2-vel osztható, tehát az x páros szám,

x % 3 == 0

akkor igaz, ha az x egész változó 3-as modulusa zérus, azaz az x hárommal osztható,

x1 > 0 && x2 > 0 && x3 > 0

igaz, ha x1, x2 és x3 mindegyike nagyobb nullánál, vagyis mind a három szám pozitív,

x > 0 && x % 2 == 1

akkor igaz, ha az x nagyobb nullánál és az x változó 2-es modulusa 1, azaz az x pozitív, páratlan szám.

A logikai operátorok működését ún. igazságtáblával írhatjuk le:

a

!a

a

b

a && b

a

b

a || b

false

true

false

false

false

false

false

false

true

false

false

true

false

false

true

true

true

false

false

true

false

true

true

true

true

true

true

true

logikai tagadás

logikai ÉS művelet

logika VAGY művelet

4.2.2 A feltételes operátor

A feltételes operátor (?:) három operandussal rendelkezik:

feltétel_kif ? igaz_kif : hamis_kif

Ha a feltétel_kif igaz (true), akkor az igaz_kif értéke adja a feltételes kifejezés értékét, ellenkező esetben a kettőspont (:) után álló hamis_kif. Felhívjuk a figyelmet arra, hogy

a feltételes művelet elsőbbsége elég alacsony, épphogy megelőzi az értékadását, ezért összetettebb kifejezésekben zárójelet kell használnunk:

```
(1 > 2 ? 12 : 23) + (2 < 3 ? 12 : 23) // értéke 35
```

4.2.3 Az if utasítás

A feltételes utasításnak három formája van.

- Az egyszerű if utasítás:

```
if (kifejezés)
utasítás
```

Ha a feltétel igaz (true), akkor az utasítás végrehajtódik.

if kifejezés igaz ág kifejezés true utasítás

- Az if-else utasításnál a feltétel teljesülése esetén az utasítás1 hajtódik végre, különben pedig az utasítás2.

```
if (kifejezés)
utasítás1
else
utasítás2
```

- A láncolt if-else-if utasításnál tetszőleges számú feltételt sorba láncolhatunk, mely sorozatot az else ág zárja. Ha egyik feltétel sem teljesül, akkor az else ág hajtódik végre:

```
if (kifejezés1)
utasítás1
else if (kifejezés2)
utasítás2
. . .
else
utasításn
```

Az alábbi programrészlet egy szövegmezőből beolvasott számról eldönti, hogy pozitív, negatív vagy zérus:

```
private void button1_Click(object sender, EventArgs e)
{
double szam;
string valasz;
if (double.TryParse(textBox1.Text, out szam))
{
valasz = szam.ToString();
if (szam > 0)
valasz += " pozitív szám";
else if (szam < 0)
valasz += " negatív szám";
```

```
else
valasz += " zérus";
MessageBox.Show(valasz);
}
else
{
MessageBox.Show("Hibás adat!");
}
}
```

4.2.4 A switch utasítás

A switch utasításban egy (bool, enum, egész vagy string típusú) kifejezés értékétől függően több lehetséges utasítás (eset) valamelyike hajtódik végre. Az egyes eseteket kötelezően valamilyen vezérlésátadó utasításnak (break, continue – ha a switch ciklusban helyezkedik el, goto, return, throw) kell zárnia. Az utasítás általános alakja:

```
switch (kifejezés) {
case konstans_kifejezés1:
utasítások
break;
case konstans_kifejezés2:
case konstans_kifejezés3:
utasítások
break;
. . .
default:
utasítások
break;
}
```

hamis ág kifejezés false utasítás2 igaz ág kifejezés true utasítás1 if kifejezés

A switch utasítás kiértékeli a kifejezést, és arra a case címkére adja át a vezérlést, amelyben a konstans_kifejezés értéke megegyezik a kifejezés értékével. A nem kötelező default címkével megjelölt utasítással folytatódik a program futása, amennyiben egyikkel sem egyezik meg. Az adott címkéhez tartozó programrészlet végrehajtása után általában a break utasítással lépünk ki a switch utasításból.

Az alábbi programrészlet egy karakteres változó értéke (+ vagy -) alapján összeget vagy különbséget számol:

```
private void button1_Click(object sender, EventArgs e)
{
char op = '+';
double a = 12, b = 23, ered;
```

```
switch (op)
{
case '+':
ered = a + b;
break;
case '-':
ered = a - b;
break;
default:
MessageBox.Show("Hibás műveleti jel: " + op);
return;
}
MessageBox.Show(string.Format(" {0} {1} {2} = {3} ", a, op, b, ered));
}
```

4.3 Ciklusutasítások

A programozási nyelvekben bizonyos utasítások automatikus ismétlését biztosító programszerkezetet iterációnak vagy ciklusnak (loop) nevezzük. Ez az ismétlés mindaddig tart, amíg az ismétlési feltétel igaznak bizonyul (while, for, do), vagy ha az adatkészlet végére érünk (foreach).

4.3.1 A while utasítás

A while ciklus mindaddig ismétli a törzsében lévő utasításokat, míg a vizsgált kifejezés (vezérlőfeltétel) értéke igaz (true). Az utasítás általános alakja:

```
while (feltétel)
utasítás; // a ciklus törzse
```

Ez egy ún. elöl tesztelt ciklus, vagyis nem biztos, hogy a ciklusmag egyáltalán lefut.

feltétel hamis igaz utasítás a ciklus utáni utasítás

Az alábbi programrészlet a 0-tól 9-ig futó while ciklusban kiszámítja a páratlan számok szorzatát és a páros számok összegét:

```
private void button1_Click(object sender, EventArgs e)
{
int osszeg = 0, szorzat = 1, i = 0;
while (i < 10)
{
if (i % 2 == 0)
osszeg += i;
else
szorzat *= i;
i++;
```

```
}
MessageBox.Show("Páros számok összege : " + osszeg + "\n" +
"Páratlan számok szorzata : " + szorzat);
}
```

4.3.2 A for utasítás

A for ciklust általában akkor használjuk, ha a ciklus törzsében megadott utasításokat egy számsorozat elemein végigfuttatva kívánjuk végrehajtani. A for ciklus valójában a while ciklus speciális átfogalmazása, mint ahogy az alábbi példákból jól látható. (Az első példában számokat összegzünk, míg a másodikban egy adott szám véletlenszerű előállítására várunk.):

```
int i = 0, s = 0;
while (i < 10)
{
s += i;
i++;
}
int i, s = 0;
for (i = 0; i < 10; i++)
{
s += i;
}
Random m = new Random();
int huzas = 0;
while (huzas != 23)
{
huzas = m.Next(123);
}
Random m = new Random();
int huzas = 0;
for (; huzas != 23; )
{
huzas = m.Next(123);
}
```


Fontos különbség azonban a kétféle ciklusmegoldás között, hogy a for ciklus esetén változók a ciklusfejben is definiálhatók, így azok elérhetősége ezáltal a törzsre korlátozódik:

```
int s = 0;
for (int i = 0; i < 10; i++)
```

```
{
s += i;
}
// Itt az i már nem érhető el!
```

A for utasítás általános formája:

```
for(inicializáció; feltétel; léptetés)
utasítás; // a ciklus törzse
```

4.3.3 A do-while utasítás

A do-while ciklus törzse egyszer, a feltétel tesztelése nélkül, mindig végrehajtódik (hátul tesztelő ciklus). A ciklus addig működik, amíg a feltétel igaz (true) értékű. Az utasítás általános alakja:

```
do
utasítás // a ciklus törzse
while (feltétel_kif);
```

Írjuk át a while ciklusnál bemutatott programot do-while használatával:

```
private void button1_Click(object sender, EventArgs e)
{
int osszeg = 0, szorzat = 1, i = 0;
do
{
if (i % 2 == 0)
osszeg += i;
else
szorzat *= i;
i++;
} while (i < 10);
MessageBox.Show("Páros számok összege : " + osszeg + "\n" +
"Páratlan számok szorzata : " + szorzat);
}
```

Igen gyakori programozási hiba, amikor a ciklusok fejlécét pontosvesszővel zárjuk. Nézzük az 1-től 10-ig a páratlan egész számok összegzésére tett kísérleteket!

```
int i = 1, s=0;
while(i<=10);
{
s += i;
i+=2;
}
int i, s=0;
```

```
for (i=1; i<10; i+=2);
{
s += i;
}
int i = 1, s=0;
do
{
s += i;
i+=2;
} while(i<10);
```

A while esetén az üres utasítást ismétli vég nélkül a ciklus (végtelen ciklus). A for ciklus lefut az üres utasítás ismétlésével (a fordító figyelmeztet!), majd pedig kiírja az s változó kilépés utáni értékét, a 11-et. A do-while ciklus esetén a fordító hibát jelez, ha a do után pontosvesszőt teszünk, így a közölt kód teljesen jó megoldást takar

igaz feltétel utasítás a ciklus utáni utasítás inicializáció léptetés hamis feltétel hamis igaz utasítás a ciklus utáni utasítás

4.3.4 A foreach utasítás

A foreach ciklust adatlisták elemeinek bejárására használjuk. Az adatlista általában tömb vagy gyűjtemény (mint például az ArrayList). A ciklus általános alakja:

```
foreach(típus elemnév in adatlista)
utasítás;
```

Az adatlista a megadott típusú elemekből áll. A ciklus segítségével az adatelemeket csak olvashatjuk, a megváltoztatásukra nincs mód. (A tömbök leírásánál látunk példákat az utasítás alkalmazására.)

4.3.5 A break, a continue és goto utasítások

Bizonyos esetekben szükséges, hogy az összetett utasításokból kilépjünk, erre szolgál a break utasítás, melynek segítségével az aktuális switch vagy ciklusblokkból (for, while, do-while, foreach) ugorhatunk ki, azaz a vezérlés a blokk utáni első utasításra kerül.

A goto utasítás feltétel nélküli ugrást valósít meg, melynek során a megadott címkére kerül a vezérlés:

```
{
goto ide;
. . .
ide:
}
```

A program áttekinthetősége, olvashatósága érdekében, amennyiben lehet, inkább a break utasítást használjuk. Általános programozási elvárás a goto utasítás használatának kerülése!

4.3.5.1 A brake utasítás a ciklusokban

Vannak esetek, amikor egy ciklus szokásos működésébe közvetlenül be kell avatkoznunk. Ilyen feladat például, amikor adott feltétel teljesülése esetén ki kell ugrani egy ciklusból. Erre ad egyszerű megoldást a break utasítás, melynek hatására az utasítást tartalmazó

legközelebbi while, for, foreach és do-while utasítások működése megszakad, és a vezérlés a megszakított ciklus utáni első utasításra kerül.

Az alábbi while ciklus kilép, ha megtalálja a megadott két egész szám legkisebb közös többszörösét:

```
int a=8, b=12, lkt;
lkt = Math.Min(a, b);
while (lkt <= a * b)
{
if (lkt % a == 0 && lkt % b == 0)
break;
lkt++;
}
MessageBox.Show("Legkisebb közös többszörös: " + lkt);
```

A break utasítás használata kiküszöbölhető, ha a hozzá kapcsolódó if utasítás feltételét beépítjük a ciklus feltételébe, ami ezáltal sokkal bonyolultabb lesz:

```
while (lkt<=a*b && !(lkt % a == 0 && lkt % b == 0)) {
lkt++;
}
```

Ha a break utasítást egymásba ágyazott ciklusok közül a belsőben használjuk, akkor csak a belső ciklusból lépünk ki vele.

4.3.5.2 A continue utasítás

A continue utasítás elindítja a while, a for, a foreach és a do-while ciklusok soron következő iterációját. Ekkor a ciklustörzsben a continue után elhelyezkedő utasítások nem hajtódnak végre. Így a continue utasítással feltételtől függően iterációs lépéseket hagyhatunk ki.

A while és a do-while utasítások esetén a következő iteráció a ciklus feltételének ismételt kiértékelésével kezdődik. A for ciklus esetében azonban, a feltétel feldolgozását megelőzi a léptetés elvégzése.

A következő példában a continue segítségével értük el, hogy a 1-től maxn-ig egyesével lépkedő ciklusban csak a 7-tel vagy 12-vel osztható számok jelenjenek meg:

```
const int maxn = 123;
string eredmeny = "";
for (int i = 1; i <= maxn; i++)
{
if ((i % 7 != 0) && (i % 12 != 0))
continue;
eredmeny += i.ToString() + " ";
}
MessageBox.Show(eredmeny);
```

A break és a continue utasítások gyakori használata rossz programozói gyakorlat. Mindig érdemes átgondolnunk, hogy meg lehet-e valósítani az adott programszerkezetet vezérlésátadó utasítások nélkül. Az előző példában ezt könnyedén megtehetjük az if feltételének megfordításával:

```
for (int i = 1; i <= maxn; i++)
{
if ((i % 7 == 0) || (i % 12 == 0))
eredmeny += i.ToString() + " ";
}
```


4.4 Kivételkezelő utasítások

A C# alkalmazás futása közben fellépő hibákat a kivételkezelés (exception-handling) segítségével tarthatjuk kézben. Amennyiben a programunkból nem készülünk fel a hibás esetekre, a .NET alapértelmezett mechanizmusa érvényesül, vagyis megszakad a program végrehajtása.

Mivel a C#-ban a kivételek is objektumok, minden egyes kivételhez saját kivételosztály tartozik. Példaként tekintsünk a leggyakrabban előforduló kivételek közül néhányat a System névtérből!

Kivételosztály	Leírás
ArithmeticException	hiba az aritmetikai kifejezésben, például 0-val osztás, túlcsordulás,
Exception	a kivételek ősosztálya,
FormatException	szöveg számmá való átalakításakor léphet fel,
IndexOutOfRangeException	tömb indexhatárainak túllépése,
NullReferenceException	objektum-referencia helyett null értéket talál a futtató rendszer,
System.IO.IOException	valamilyen input/output hiba lépett fel.

A C# a try, a catch, a finally és a throw alapszavakkal támogatja a kivételkezelést. A kezelésre valójában csak az első három szó szolgál, hisz a throw utasítást kivételes helyzetek programból történő előállítására használhatjuk, például:

```
throw new Exception();
```

A C# alkalmazásban a try-catch, a try-catch-finally, illetve a try-finally utasításokkal felkészülhetünk az esetleges hibás állapotok kezelésére. A kivétel feldolgozásához egy try (jelentése „próbáld”) blokkba kell tenni azokat az utasításokat, amelyek kivételt okozhatnak. Ha a try-blokk végrehajtása közben kivétel jön létre, a blokk végén álló catch segítségével „elkaphatjuk” azt, és megadhatjuk, hogy mely utasításokat kell a

kivétel fellépése esetén végrehajtani. Amennyiben semmilyen kivétel sem keletkezik, vagy ha a kivétel típusa nem egyezik a catch-ban szereplővel, akkor a catch-blokk utasításai nem futnak le.

```
private void button1_Click(object sender, EventArgs e)
{
try
{
int adat = Convert.ToInt32(textBox1.Text);
MessageBox.Show("A megadott szám 11-szerese: " + (11 * adat));
}
catch (FormatException ex)
{
MessageBox.Show("Hibás adatbevitel!");
}
MessageBox.Show("A program fut tovább...");
}
```

Egy try-blokkhoz több catch-blokk is tartozhat, hiszen a catch után zárójelben álló, egyetlen paraméter típusa választja ki a kezelendő kivételt. Az aktuális catch-blokk végrehajtása után az utolsó catch-blokkot követő utasítással folytatódik a program futása. Ha a fellépő kivételt egyik catch sem azonosította, az a futtatórendszerhez továbbítódik, és a program futása hibaüzenettel megszakad. A catch blokkok sorozatát egy paraméter nélküli catch zárhatja, mellyel minden, a fentiekben nem azonosított kivételt elfoghatunk:


```
private void button1_Click(object sender, EventArgs e)
{
try
{
// figyelt utasítások
}
catch(IndexOutOfRangeException ex)
{
MessageBox.Show("Hibás tömbindex.");
}
catch(ArithmeticException ex)
{
MessageBox.Show("Számítási hiba.");
}
catch
```

```
{
MessageBox.Show("Valami egyéb hiba történt.");
}
MessageBox.Show("A program fut tovább...");
}
```

Ha egy megkezdett tevékenységet mindenképpen be szeretnénk fejezni, akkor a try-blokk, illetve az őt követő catch szerkezet után álló finally (jelentése „végül”) blokkban megadhatjuk az ehhez szükséges utasításokat. A finally-blokk utasításai mindig végrehajtódnak, akár fellép kivétel, akár nem. Kezelt kivétel esetén a catch utasításait követően, kezeletlen kivétel esetén pedig a program futásának hibaüzenettel történő befejezése előtt. A finally-blokk utasításai akkor is lefutnak, ha a return, a break vagy a continue utasítással kilépünk a try-blokkból.

```
private void button1_Click(object sender, EventArgs e)
{
try
{
// figyelt utasítások
MessageBox.Show("Figyelés indul...");
}
catch(ArithmeticException ex)
{
MessageBox.Show("Számítási hiba.");
}
finally
{
MessageBox.Show("Ez mindeképp lefut!");
}
MessageBox.Show("A program fut tovább...");
}
```

Ha nem lép fel kivétel a try-blokkban, akkor a metódus hívásakor az alábbi szövegek jelennek meg:

Figyelés indul...

Ez mindeképp lefut!

A program fut tovább...

A kezelt ArithmeticException kivétel keletkezése esetén ezt láthatjuk:

Figyelés indul...

Számítási hiba.

Ez mindenképp lefut!

A program fut tovább...

A kezeletlen FormatException kivétel fellépése pedig ezt eredményezi:

Figyelés indul...

Ez mindenképp lefut!

...futás közbeni hibaüzenet ...

Érdemes megjegyezni, hogy a catch-blokkok nélküli try-finally utasítást is használhatunk, valamint azt, hogy a try-catch-finally blokkokat egymásba is ágyazhatjuk.

5. Tömbök használata

Amennyiben nagyobb mennyiségű, azonos típusú adatot kell tárolnunk a memóriában, a tömböket hívhatjuk segítségül. Különbséget teszünk a folytonos területen elhelyezkedő egy- és többkiterjedésű (-dimenziós), valamint az ún. feldarabolt (jagged) tömbök között. (Ez utóbbival nem foglalkozunk a jegyzetben.)

A C# nyelvben a tömböket dinamikusan, a new operátor segítségével kell létrehozni, és a tömbök mérete lekérdezhető a Length tulajdonsággal. A C# teljeskörű indexhatár-ellenőrzést végez, és kivétellel jelzi, ha az index kívül esik az index-határon. A tömb területének felszabadításáról a garbage collection (szemétgyűjtő) folyamat gondoskodik, melynek átadhatjuk a tömb területét, ha a tömb nevéhez a null értéket rendeljük. A tömb elemeit, más nyelvekhez hasonlóan, az indexelés operátorával [ ] érjük el. A tömb elemeinek indexe (minden dimenzióban) 0-val kezdődik.

5.1 Egydimenziós tömbök

Az egyetlen kiterjedéssel rendelkező tömböket szokás vektoroknak nevezni. Például, az egész elemeket tartalmazó, egydimenziós tömbhivatkozás definíciója:

```
int [] tömbnév;
```

Ötelemű vektor létrehozása:

```
tömbnév = new int[5];
```

A fenti két lépés egyben is elvégezhető:

```
int[] tömbnév= new int[5];
```

Szükség esetén az elemeknek kezdőértéket is adhatunk, melyek száma meg kell egyezzen a tömb elemeinek számával:

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

A létrejövő tömb méretét az iniciliazációs lista elemeinek számával is meghatározhatjuk:

```
int[] tömbnév = {1,2,3,4,5};
```

illetve

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

A tömb elemeinek írható/olvasható eléréséhez az indexelés műveletét használjuk a for cikluson belül. Nem szabad megfeledkeznünk arról, hogy az első elem indexe mindig 0, míg az utolsó elem indexe méret-1, esetünkben tömbnév.Length-1:

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

tömbnév[i] = i+1;
```

A tömbelemek kényelmes elérését segíti a foreach ciklus, azonban ekkor az elemek nem módosíthatók:

```
int s=0;
foreach (int elem in tömbnév)
s += elem;
```

Amennyiben nincs szükségünk többé a tömb elemeire, ezt az alábbi értékadással jelezhetjük a futtatórendszernek:

```
tömbnév = null;
```

Az alábbi példában gombnyomásra feltöltünk egy tízelemű egész tömböt, majd pedig meghatározzunk az elemek összegét és átlagát:


```
private void button1_Click(object sender, EventArgs e)
{
const int elemszam = 10;
int osszeg = 0;
string eredmeny = "";
Random m = new Random();
int[] x = new int[elemszam];
eredmeny = "A tömb elemei: ";
for (int i = 0; i < x.Length; i++)
{
x[i] = (int)(m.Next(10) + 1);
eredmeny += x[i] + ", "; ;
}
foreach (int elem in x)
osszeg += elem;
double atlag = (double)osszeg / x.Length;
eredmeny += "\nA tömb elemeinek összege: " + osszeg;
eredmeny += "\nA tömb elemeinek átlaga : " + atlag;
MessageBox.Show(eredmeny);
}
```

5.2 Két- és többdimenziós folytonos tömbök

A C# nyelvben két- és többkiterjedésű tömböket is készthetünk, amelyeknél két, illetve több indexszel jelöljük ki a kívánt elemet. Könyvünkben csak a kétdimenziós tömbökkel foglalkozunk, azonban az elmondottak további dimenziókra is továbbvihetők. Nézzük a kétdimenziós tömbök általános definícióját:

```
elemtípus [,] tömbnév;
tömbnév = new elemtípus[méret1, méret2];
```

ahol az első dimenzió (méret1) a tömb sorainak, míg a második (méret2) a tömb oszlopainak számát határozza meg. Mátrixok tárolására a kétdimenziós tömböket használjuk.

Példaként tekintsük a 2 sort és három oszlopot tartalmazó tömb (mátrix) kezelését! Az egydimenziós esethez hasonlóan a tömbhivatkozás és a tömb létrehozása külön, illetve egyetlen utasításban egyaránt megadható:

```
int [,] mátrix;

mátrix = new int[2, 3];

int[,] mátrix = new int[2, 3];
```

A kétdimenziós tömböknek kezdőértéket is adhatunk. Az inicializációs listában az elemek számának dimenziónként egyeznie kell a tömb kiterjedéseivel:

```
int[,] mátrix = new int[2, 3] {{1,2,3},{4,5,6}};
```

A létrejövő tömb méretét az iniciliazációs lista elemeinek számával is meghatározhatjuk:

```
int[,] mátrix = {{1,2,3},{4,5,6}};
```

illetve

```
int[,] mátrix = new int[,] {{1,2,3},{4,5,6}};
```

Felhívjuk a figyelmet, hogy a Length tulajdonság többdimenziós esetben is a tömb elemeinek számát tartalmazza, a példánkban 6-ot. Amennyiben dimenziónként van szükségünk az elemek számára, a tömb GetLength(dim) metódusát kell hívnunk, argumentumként átadva a dimenzió sorszámát (0, 1, ...). A mátrix tömb elemeit téglalapos elrendezésben, üzenetablakban megjelenítő programrészlet:

```
string eredmeny = "";

for (int i = 0; i < mátrix.GetLength(0); i++)

{

for (int j = 0; j < mátrix.GetLength(1); j++)

eredmeny += mátrix[i, j] + "\t";

eredmeny += "\n"; ;

}

MessageBox.Show(eredmeny);
```


A teljes tömb bejárását, és elmeinek kiolvasását az alábbi kódrészlet szemlélteti:

```
int sum = 0;

foreach (int elem in mátrix)

sum += elem;
```

Amennyiben nincs szükségünk többé a mátrix elemeire, az alábbi értékadással jelezhetjük ezt a futtatórendszernek:

```
mátrix = null;
```

5.3 A System.Array osztály

A System névtérben definiált Array (tömb) osztály minden folytonos helyfoglalású tömb statikus metódusai segítik a tömbök hatékony kezelését. Bizonyos megoldások a tömbobjektumok tulajdonságai és metódusai révén is megvalósíthatók.

Az alábbiakban először összefoglaljuk a tulajdonságokat, majd néhány gyakran alkalmazott művelet elvégzésére mutatunk példát. Induljunk ki az alábbi két deklarációból:

int[] v = new int[35];

int[,] m = new int[12, 23];

A Rank tulajdonság a dimenziók számát, míg a Length tulajdonság az elemek számát szolgáltatja:

v.Rank

1

m.Rank

2

v.Length

35

m.Length

276

Az IsReadOnly tulajdonság mindkét esetben false értékkel jelzi, hogy a tömbök módosíthatók.

Az Array osztály statikus metódusainak többsége egydimenziós tömbökkel működik helyesen. (Bizonyos műveletek a többdimenziós tömböket, mint Length hosszú vektorokat kezelik.). Például, a Clear() metódus a tömb elemeinek nullázását végzi, kijelölve a kezdőelemet és a nullázandó elemek számát. A v és m tömbök teljes nullázása a következőképpen végezhető el:

Array.Clear(v, 0, v.Length);

Array.Clear(m, 0, m.Length);

Az elmondottak figyelembevételével tekintsünk néhány hasznos megoldást! Minden műveletet az alábbi elemkészlettel feltöltött x vektoron végzünk el:

int[] x ={0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};

int[] y = new int[x.Length];

Művelet

Leírás

A tömb tartalma a művelet után

Array.Clear(x, 5, x.Length - 5);

Adott indextől adott számú elem nullázása.

0 1 2 3 4 0 0 0 0 0 0 0

Array.Copy(x, 7, x, 0, 5);

A forrástömb 7. indextől 5 elemének másolása a cél tömbbe a 0. indextől kezdve.

7 8 9 10 11 5 6 7 8 9 10 11

Array.Resize(ref x, x.Length/2);

Vektor átméretezése.

0 1 2 3 4

Array.Reverse(x);

A teljes vektor elemsorrendjének megfordítása.

11 10 9 8 7 6 5 4 3 2 1 0

Array.Reverse(x,3,5);

A részvektor elemsorrendjének megfordítása.

0 1 2 7 6 5 4 3 8 9 10 11

Array.Sort(x);

Az x vektor elemeinek rendezése növekvő sorrendben.

Array.Sort(x);

Array.Reverse(x);

Az x vektor elemeinek rendezése csökkenő sorrendben.

29

Array.Copy(x, y, x.Length);

Az x tömb tartalmának átmásolása az y tömbbe.

x.CopyTo(y,0);

Az x vektor tartalmának átmásolása az y vektorba a 0. indextől.

Az adatfeldolgozás szempontjából talán legfontosabb a rendezett vektorban való gyors keresést biztosító BinarySearch() metódus. A használatához szükséges rendezettséget a Sort() metódus hívásával állíthatjuk elő. Sikeres esetben a keresés eredménye a megtalált elem index, illetve -1 ellenkező esetben. Rendezetlen tömbben is kereshetjük egy adott elem első (IndexOf()), illetve annak utolsó előfordulását (LastIndexOf()).

30

6. Metódusok

Az eddigi példaprogramokban csak eseménykezelő metódusokban végeztünk műveleteket. Ezek típusa, paraméterezése a .NET rendszerben rögzített, sőt a hívásáról is a Windows rendszer gondoskodik. Nagyobb, jól strukturált programok esetén általában több alprogramot (függvényt, metódust) használunk, hisz általuk az alkalmazás forráskódja olvashatóbb, érthetőbb lesz, sőt hatékonyabban javítható és a későbbiekben kényelmesen tovább is fejleszthető. Ugyancsak metódusokkal járunk jól, ha ugyanazt a számítást, tevékenységet többször kell elvégeznünk a programban.

Mint a bevezetőben is láttunk, a metódusok képezik az osztályok működő részét, hiszen csak itt adhatunk meg végrehajtható utasításokat.

6.1 A metódusok definiálása

A metódus definiálásának általános formája (a □□ karakterek között szereplő elemek opcionálisak, vagyis, vannak esetek, amikor nem szerepelnek):

□módosítók□ típus metódusnév(□paraméterlista□)

{

// a metódus törzse

lokális deklarációk

egyéb utasítások

□return □kifejezés□;□

}

A leggyakrabban előforduló módosítók:

 public – a metódus nyilvános, vagyis bármely más metódusból hívható,

 private – saját metódus, azaz csak a metódust tartalmazó osztály metódusaiból hívható, vagy más osztály metódusaiból nem,

 static – statikus, objektumpéldány nélkül is hívható az osztály nevét használva.

A metódus (visszatérési) típusa az alábbiak közül kerülhet ki:

 tetszőleges értéktípus, az egyszerű típusoktól, a felsorol típuson át egészen a struktúrákig,

 a referenciatípusok (tömb, osztály stb.) valamelyike,

 ha nincs visszatérési érték, akkor ezt a void típussal jelezzük.

A return utasításnak két fontos szerepe van:

 hatására a hívott metódus visszatér a hívás helyére, és a vezérlés a hívást követő utasításra kerül,

 az utasításban szereplő kifejezés értéke, mint függvényérték (visszatérési érték) jelenik meg.

6.1.1 A C# nyelv metódusainak osztályozása:

Ha a metódus rendelkezik visszatérési típussal (például double), akkor (más nyelvekben) függvénynek hívjuk. A függvény egyetlen visszatérési értékének típusa megegyezik a metódus neve előtt álló típussal. Ekkor a metódus a paraméterlistán átadott paraméterekkel műveleteket végez, melyek eredményét egyetlen értékként adja vissza a return utasítás segítségével.

public double MertaniKozep(double x, double y){ return Math.Sqrt(x * y);}a metódus törzsemódosítóvisszatérési típusmetódusnéva paraméterek típusavisszatérési értéka formális paraméterek neve

Az elmondottaknak megfelelő metódus (függvény) aktiválható az értékadó utasítás jobb oldalán, vagy például a MessageBox.Show() metódus argumentumaként. Ekkor a metódushívás, mint kifejezés jelenik meg a programban.

Tipikusan ilyen metódusok például a Math osztályban összefogott matematikai függvények és a ToString() metódus:

double a = 1, b = -4, c = 4;

double x1 = (-b + Math.Sqrt(b * b - 4 * a * c)) / (2 * a);

MessageBox.Show(Math.Sin(Math.PI / Math.E).ToString());

A visszatérési értékkel nem rendelkező (void típusú) metódust más nyelveken eljárásnak nevezzük. Az eljárás metódusokat általában valamilyen jól meghatározott tevékenység elvégzésére használjuk. Az ilyen metódusok hívása önálló utasításként jelenik meg a programban.

a metódus törzseprivate void Megjelenit(string szoveg, double adat){ MessageBox.Show(szoveg + ": " + adat, "Megjelenít");}a paraméterek típusamódosítónincs visszatérési értékmetódusnéva formális paraméterek neve

Az eddigiekben a megjelenítő metódusokat használtuk ily módon:

Console.WriteLine("Eredmények:");

MessageBox.Show("Eredmények:");

6.1.2 Formális és aktuális paraméterek

A metódusfejben szereplő paramétereket formális paramétereknek nevezzük. Minden formális paraméternek van típusa, és vesszővel választjuk el őket egymástól. A metódusok tetszőleges számú paraméterrel rendelkezhetnek.

A metódusok hívása során használt argumentumokat aktuális paramétereknek nevezzük. Mint említettük, az aktuális paraméterek típusának és sorrendjének összhangban kell lennie a formális paraméterlista elemeivel. A paraméterek használatával a későbbiekben részletesen foglalkozunk.

6.1.3 A metódusok hívása

Egy metódus hívásakor a metódus nevét és azt követően kerek zárójelek között az aktuális paraméterek (argumentumok) vesszővel tagolt listáját kell megadnunk. A zárójelek használata akkor is kötelező, ha semmilyen paraméterrel sem rendelkezik a metódus. Az aktuális paraméterek típusának és számának egyeznie kell a metódus-fejben szereplő megfelelő formális paraméterek típusával és számával. A fenti metódusok hívása:

```
double mk = MertaniKozep(7, 27);

Megjelenit("12 és 23 mértani közepe", MertaniKozep(12, 23));

Random rnd = new Random();

int rszam = rnd.Next();
```

6.1.4 A lokális és az osztályszintű változók használata

A metódusokban deklarált változók korlátozott, helyi (lokális) elérhetőséggel rendelkeznek, és csak az őket deklaráló blokk (utasításblokk) végéig élnek. Minden alkalommal a blokk elején megszületnek, és a memóriaterületük felszabadul, ha a vezérlés kikerül a blokkból. Természetesen ezt követően már nem lehet rájuk hivatkozni.

A lokális változók nem rendelkeznek automatikus kezdőértékkel, ezért ügyelnünk kell arra, hogy az első felhasználás helyéig kapjanak értéket!

A metódusokban általában akkor használunk lokális változókat, ha segítségükkel egy bonyolultabb algoritmus programozását egyszerűbbé tehetjük. Felhívjuk a figyelmet arra, hogy egy adott nevű lokális változót csak egyszer deklarálhatunk a metódusban, függetlenül attól, hogy az egymásba ágyazott blokkok melyik szintén szerepel a deklarációja.


```
// Nem fordítható le!
private void button1_Click(object sender, EventArgs e)
{
double a = 1, b;
b = 2;
if (b > 20)
{
int c = 0;
for (int a = 0; a < b; a++)
c += a;
}
```

```
}
// Lefordítható
private void button1_Click(object sender, EventArgs e)
{
double a = 1, b;
b = 2;
if (b > 20)
{
int c = 0;
for (int x = 0; x < b; x++)
c += x;
}
}
```

Az azonos osztályban definiált metódusok sem „látják” egymás lokális változóit. Amennyiben több metódusból szeretnénk ugyanazt a változót elérni, akkor annak definícióját az osztályon belül, azonban a metódusokon kívül kell elhelyeznünk. Az ilyen változók az osztályszintű változók, vagy adatmezők (fields).

Az adatmezők, ellentétben a lokális változókkal, a típusuknak megfelelő kezdőértéket (0, "", null stb.) kapnak a program indításakor. Ezek az osztályszintű változók statikussá (static) tehetők, így az osztály példányai megosztva használják azt.

Az alábbi példában szövegmezőből beolvassuk egy tömb méretét, létrehozzuk és feltöltjük a tömböt. Egy másik gomb megnyomásakor összegezzük a tömb elemeit. A megoldáshoz a tömböt adatmezőként kell deklarálnunk, míg többi változót helyi eléréssel használhatjuk:

```
public partial class Form1 : Form
{
int[] tomb; // adatmező
private void button1_Click(object sender, EventArgs e)
{
int meret = Convert.ToInt32(textBox1.Text); // lokális változó
tomb = new int[meret];
Random rnd = new Random(); // lokális változó
for (int i = 0; i < tomb.Length; i++) // i lokális változó a ciklusban
{
tomb[i] = rnd.Next(100);
}
}
private void button2_Click(object sender, EventArgs e)
{
```

```
int osszeg = 0; // lokális változó
for (int i = 0; i < tomb.Length; i++) // i lokális változó a ciklusban
{
osszeg += tomb[i];
}
textBox2.Text = osszeg.ToString();
}
}
```

6.2 Különböző fajtájú paraméterek használata

Paraméterek definiálásával a metódusok felhasználhatósága nagyban megnövekszik, hiszen ugyanazt a műveletsort, akár különböző adatokon is végrehajthatjuk a programon belül. A paramétereket alapvetően két csoportba soroljuk,

A C# nyelvben négy különböző paraméterfajta létezik, az értékparaméterek (alapértelmezés), a referencia-paraméterek (a ref módosítóval), a kimenő paraméterek (az out módosítóval) és a paramétertömbök (a params módosítóval). A fenti paraméterek bármelyikét egyaránt deklarálhatjuk érték- és referenciatípusokkal. Amikor azt a kifejezést halljuk, hogy „referencia” vagy „érték”, tisztában kell lennünk azzal, hogy most típusról vagy paraméterről beszélünk.

Értéktípus esetén a változó közvetlenül a neki megfeleltetett értéket tárolja. Hivatkozástípus esetén az általunk létrehozott változó csupán egy hivatkozást tárol, amely hivatkozással közvetett módon érhetjük el az adatokat.

Értéktípus

Hivatkozástípus

```
double na, nb;
na = 12;
nb = na; // érték másolódik
nb -= 5;
```

na □ 12, nb □ 7

```
int[] va, vb;
va = new int[] {12, 23};
vb = va; // hivatkozás másolódik
vb[0] -= 5;
```

va[0] □ 7, vb[0] □7

A fenti táblázat jól szemlélteti az elmondottakat. A definiált változókat egy ábrával tehetjük még szemléletesebbé, ahol a táblázatban szereplő változók utolsó állapotát láthatjuk a memóriában. (A piros nyíl a hivatkozást /referenciát jelöli.)

6.2.1 Értékparaméterek

Értékparaméterek esetén ugyanaz megy végbe, mint normál értékadásnál, hiszen a metódusban létrejövő lokális paraméter-változóba átmásolódik a híváskor megadott adat vagy hivatkozás.

Érték szerinti paraméterátadásnál tetszőleges olyan kifejezés is megadható argumentumként, amellyel a megfelelő paraméter értéke előállítható. Az alábbi példában az Osztályoz() metódus a paraméterként átadott pontszám alapján függvényértékként visszaadja az osztályzatot.

```
int Osztalyoz(int pontszam)
{
int erdemjegy = 1;
if (pontszam >= 86) erdemjegy = 5;
else if (pontszam >= 76) erdemjegy = 4;
else if (pontszam >= 66) erdemjegy = 3;
else if (pontszam >= 51) erdemjegy = 2;
return erdemjegy;
}
```

Az Osztalyoz() metódus többféleképpen is hívható valamely más metódusából az osztálynak:

```
int jegy, pontok = 83;
jegy = Osztalyoz(59);
jegy = Osztalyoz(20 + 2 * 20 + 15);
jegy = Osztalyoz(pontok);
```

A híváskor megadott változó argumentumok és a deklarált paraméterek nevei között nincs semmiféle kapcsolat, a fordító csupán a típusok egyezésére ügyel.

Amennyiben egy referenciatípusú paramétert fogadunk értékként, a hivatkozott tömb vagy objektum minden további nélkül módosítható a híváskor átadott referencia felhasználásával.

A következő példában egydimenziós egész típusú tömbökkel műveleteket végző metódusok szerepelnek. Felhívjuk a figyelmet arra, hogy a függvények hívásakor a tömb hivatkozását értékként adjuk át.

```
void Feltolt(int[] a, int max) {
Random rnd = new Random();
for (int i = 0; i < a.Length; i++)
a[i] = (rnd.Next(max) + 1);
}
int MinKeres(int[] a) {
int minAdat = a[0];
for (int i = 1; i < a.Length; i++)
if (a[i] < minAdat) minAdat = a[i];
return minAdat;
```

```
}
int MaxKeres(int[] a) {
int maxAdat = a[0];
for (int i = 1; i < a.Length; i++)
if (a[i] > maxAdat) maxAdat = a[i];
return maxAdat;
}
```

A metódusok tetszőleges méretű int típusú elemeket tartalmazó tömbargumentumokkal hívható. A függvények hívásakor csak a tömb nevét kell megadnunk, a szögletes zárójeleket nem.

```
private void button1_Click(object sender, EventArgs e)
{
int[] vektor = new int[12];
Feltolt(vektor, 123);
int minElem = MinKeres(vektor);
int maxElem = MaxKeres(vektor);
// …
}
```

Az utolsó példánkban a paraméterként átadott listaablakot töltjük fel adott intervallumba eső formázott valós számokkal:

```
void Listaz(ListBox lista, double min, double max, double lepes)
{
lista.Items.Clear();
for (double adat = min; adat<=max; adat+=lepes)
lista.Items.Add(adat.ToString("0###.##"));
}
```

Amennyiben a formon két listaablak található, a fenti metódus lehetséges hívásai:

```
private void button1_Click(object sender, EventArgs e)
{
Listaz(listBox1, 0, Math.PI, Math.PI / 30);
Listaz(listBox2, -5, 3, 0.1);
}
```

6.2.2 Referencia-paraméterek

A referencia-paraméter a híváskor az argumentumként megadott változót azonosítja, mintegy második nevet adva a változónak. Így a paraméteren keresztül magát a változót érjük el, legyen az akár érték- vagy hivatkozástípusú.

A referencia-paramétert a ref módosítóval jelöljük, mind a deklarációban, mind pedig a metódus hívásakor a megfelelő aktuális paraméter előtt. A hivatkozás-paramétereket

szokás változóparaméternek is nevezni, hiszen hívási argumentumként csak megfelelő típusú, inicializált változót adhatunk meg.

Három egész változó értékét a következők metódusok segítségével állíthatjuk növekvő sorrendbe. A referencia paraméterek használata azért szükséges, mivel a metódusokkal külső változók értékét kívánjuk megváltoztatni.

```
void Csere(ref int x, ref int y)
{
int z = x;
35
x = y;
y = z;
}
void RendezNovekvo(ref int a, ref int b, ref int c)
{
if (a > b)
Csere(ref a, ref b);
if (a > c)
Csere(ref a, ref c);
if (b > c)
Csere(ref b, ref c);
}
```

A RendezNovekvo metódus hívása:

```
private void button1_Click(object sender, EventArgs e)
{
int x = 7, y = 5, z = 21;
RendezNovekvo(ref x, ref y, ref z);
MessageBox.Show("x=" + x + ", y=" + y + ", z= " + z);
}
```

Tömböt és objektumot csak akkor kell referenciával átadni, ha a metóduson belül meg kívánjuk változtatni az átadott változókban tárolt referenciák értékét. Ilyen eset, amikor például a metódusban foglalunk helyet egy tömb számára:

```
void TombotFeltolt(ref int[] tomb)
{
tomb = new int[7] { 3, 1, 2, -1, 4, 5 , 0};
}
```

A metódus lehetséges hívásai egy másik metódusból. (Nem szabad megfeledkeznünk a referenciával átadni kívánt változók kezdőértékének beállításáról!)

```
int[] aTomb = null, bTomb = null;
```

```
TombotFeltolt(ref aTomb);
TombotFeltolt(ref bTomb);
```

6.2.3 Kimenő paraméterek

A referencia-paraméterekhez hasonló módon ún. kimenő paramétereket is használhatunk a metódusokban, melyeket a metódus deklarációjában és a hívásakor egyaránt az out módosítóval kell megjelölni.

Alapvető eltérés a ref és az out paraméterek között, hogy kimenő formális paraméterként (argumentumként) inicializálatlan változó is megadható. A kimenő paramétereknek azonban mindenképpen értéket kell adni a függvényből való kilépés előtt.

Az alábbi példában MinMax metódus segítségével egyszerre kapjuk meg egy tömb legkisebb és legnagyobb elemének értékét és pozícióját:

```
void MinMax(int[] v, out int min, out int minIndex, out int max, out int maxIndex)
{
min = v.Min();
minIndex = Array.IndexOf(v, min);
max = v.Max();
maxIndex = Array.IndexOf(v, max);
}
```

A metódus lehetséges alkalmazása:

```
private void button1_Click(object sender, EventArgs e)
{
int[] a = { 23, 29, -7, 0, 12, 7, 102 };
int min, minpoz, max, maxpoz;
MinMax(a, out min, out minpoz, out max, out maxpoz);
MessageBox.Show(string.Format("min: a[{0}]->{1}, max: a[{2}]->{3}", minpoz, min,
    maxpoz, max));
}
```


6.2.4 Paramétertömbök

A metódusok hívásánál változó darabszámú, azonos típusú argumentumot is megadhatunk. A változó hosszúságú argumentumlista feldolgozását paramétertömb deklarálásával végezhetjük el. A paraméterlista végén elhelyezkedő, egydimenziós paramétertömböt a params módosító vezeti be.

A példában szereplő Atlag metódus meghatározza tetszőleges számú valós argumentum számtani közepét:

```
double Atlag(params double[] szamok)
{
return szamok.Sum() / szamok.Length;
}
```

A metódus valós tömbbel, illetve számok vesszővel tagolt listájával egyaránt hívható:

```
double[] a = { 1.23, 2.34, 3.45 };
double a1 = Atlag(a); // 2.34
double a2 = Atlag(12.34, 3.12); // 7.73
double a3 = Atlag(123); // 123.0
double a4 = Atlag(); // NaN - nem egy szám
```

6.3 Metódusok túlterhelése (átdefiniálása)

A C# nyelvben a különböző paraméterlistájú metódusokat azonos névvel is elláthatjuk. Ezt a metódusok túlterhelése (overloading) teszi lehetővé. Híváskor az a metódus aktiválódik, mely aktuális paraméterlistájának lenyomata (típussora) megegyezik a formális paraméterekével.

Példánkban két szám összegzésére különböző függvényváltozatokat készítünk double, int és string típusú argumentumokhoz:

```
double Osszeg(double a, double b)
{
return a + b;
}
int Osszeg(int a, int b)
{
return a + b;
}
double Osszeg(string a, string b)
{
return Convert.ToDouble(a) + Convert.ToDouble(b);
}
```

Az Osszeg metódus hívásakor a fordító dönti el az argumentumok típusa alapján, hogy melyik változatot kell hívni:

```
double sum1 = Osszeg(1.2, 2.3);
int sum2 = Osszeg(12, 23);
double sum3 = Osszeg("1.2", "2.3");
```

Amennyiben a teljes típusegyezés nem áll fenn, azok a típusok is szóba jönnek, melyekre automatikusan konvertál a fordítóprogram:

```
double sum1 = Osszeg(1.2F, 23); // float, int -> double, double
```

Amennyiben nem talál a fordító megfelelő függvényváltozatot, hibajelzést (The best overloaded method match for …) kapunk.

A .NET rendszer osztályai nagyon sok túlterhelt metódust tartalmaznak, melyek közül a megfelelő kiválasztását a fejlesztőrendszer is segíti. Például, a véletlen szám osztály használatakor 2 konstruktor metódusok közül választhatunk, ha kattintunk a nyilakra:

Még több lehetőség közül választhatunk a Messagebox osztály Show metódusa esetén:

7. Felhasználói adattípusok

A programozási feladatok egy részében a megfelelően kialakított, megválasztott adatstruktúra sok kódírástól kíméli meg a program készítőjét. C# nyelven a struct és a class felhasználó adattípusok segítségével tetszőleges típusú, logikailag összetartozó adatokat egyetlen egységben kezelhetünk, sőt hozzájuk saját műveleteket (metódusok formájában) is definiálhatunk.

Mindkét említett kulcsszó segítségével definiált típusokat osztálynak nevezzük, bár a struct esetében a struktúra kifejezést is használhatjuk, ami valójában egy korlátozott lehetőségekkel rendelkező osztály. Valamely osztálytípussal deklarált változókat objektumoknak, vagy az osztály példányainak (instance) nevezzük.

Ezek az objektumok képezik az objektum-orientált programozás alapját. Az adatok és az adatokat kezelő függvények (metódusok) összeépítése (encapsulation) nagyon fontos sajátossága minden objektum-orientált nyelvnek.

Az osztályok rendelkeznek az adatrejtés (data hiding) képességével, amely elengedhetetlen a nagyobb programok kialakításához. Ez azt jelenti, hogy az osztály tagjainak elérhetősége a private (saját) és a public (nyilvános) kulcsszavak felhasználásával szabályozható. Megjegyezzük, hogy a C# programokban további elérési kulcsszavakat (internal, protected stb.) is alkalmazhatunk.

Az osztálytagok alapértelmezett elérési módja a private, amely megakadályozza a tagok más osztályok metódusaiból történő elérését.

Az objektum-orientált programozás további eszközeivel, az örökléssel (inheritance) és a többalakúsággal (polymorphism) a jegyzet későbbi fejezeteiben foglalkozunk. Felhívjuk a figyelmet arra, hogy ezek a megoldások csak a class kulcsszóval definiált osztályok esetén működnek.

A következőkben sorra vesszük azokat az alapvető programelemeket (adatmezőket, metódusokat, tulajdonságokat és indexelőket), amelyekből felépíthetjük az osztályainkat. A tárgyalás során egyaránt foglalkozunk a struct és a class típusú osztályokkal. A két megoldás között a legfontosabb különbség, hogy a class osztályok referenciatípusok, míg a struct osztályok (struktúrák) értéktípusokat képviselnek.

Az osztályok tagjai az alábbiak közül kerülhetnek ki (többekkel már találkoztunk a korábbi fejezetekben):

Tag

Leírás

Konstansok (const, readonly)

Az osztályhoz kapcsolódó szimbolikus konstansok.

Adatmezők

Az osztály változói.

Metódusok

Az osztály által elvégezhető műveleteket megvalósító függvények.

Tulajdonságok

Adatmezők olvasásához és írásához kapcsolódó műveletek.

Indexelők

Az osztálypéldányok indexeléséhez kapcsolódó műveletek.

Események (event)

Az osztály által létrehozható értesítések.

Operátorok (operator)

Az osztály által biztosított konverziós és egyéb műveleteket megvalósító függvények.

Konstruktorok

Függvények, melyek inicializálják az osztály példányait, illetve magát az osztályt.

Típusok

Az osztályba ágyazott típusdeklarációk.

A tagokat két csoportra, a statikus (static) és a példánytagok csoportjára oszthatjuk. A statikus tagok magához az osztályhoz tartoznak, és példányok nélkül is használhatók, míg a példánytagokat csak az objektum létrehozása után érhetjük el.

7.1 Osztályok definiálása

Első megközelítésben olyan osztályokat készítünk, amelyek csak adatokat (konstansokat és változókat) tartalmaznak. Ebben az esetben az adatokhoz nyilvános (public) hozzáférést kell biztosítanunk. Az osztályok készítésének általános formáiban a □ □ karakterek között szereplő elemek opcionálisak.

Felhívjuk a figyelmet arra, hogy az osztályok leírásában az osztály szintjén deklarált változóknak (adatmezőknek) és readonly konstansoknak kezdőértéket csak a class definícióban adhatunk.

□elérési előírás□ class Osztálynév

{

// konstansok

□elérési előírás□ const típus konstansnév1 = konstans érték1;

□elérési előírás□readonly típus konstansnév2 = konstans érték2;

// adatmező változók

□elérési előírás□ típus mezőnév1 □ = kezdőérték1 □;

□elérési előírás□ típus mezőnév2 □ = kezdőérték2 □;

}

□elérési előírás□ struct Struktúranév

{

// konstansok

□elérési előírás□ const típus konstansnév1 = konstans érték1;

□elérési előírás□readonly típus konstansnév2;

// adatmező változók

□elérési előírás□ típus mezőnév1;

□elérési előírás□ típus mezőnév2;

}

Az osztály típusok elérési előírását általában nem adjuk meg, elfogadjuk az internal hozzáférést, amely biztosítja az adott programegységen belüli elérését az osztálynak. (A másik lehetőség az osztály nyilvánossá tétele minden más programegység számára.)

Az osztálytagok esetén a saját (private) elérés az alapértelmezett, ami az osztály saját metódusaira korlátozza a tagok elérését. Ahhoz, hogy az adott programegységen belül bárhonnan elérjük a tagokat, az internal vagy a public hozzáférést kell használnunk.

7.2 Objektumok létrehozása

Mivel a class osztályok referenciatípusok, a velük definiált változók csupán egy hivatkozást tárolnak arra az objektumra, amit a new felhasználásával hozunk létre a program halomterületén (heap). Így a példánytagok eléréséhez először létre kell hoznunk egy referencia változót, majd a new operátor segítségével elkészíthetjük és inicializálhatjuk az objektumot:

A C#-ban az objektumpéldányok létrehozásának első lépése a megfelelő típusú objektumhivatkozás változó deklarálása:

Osztálynév objektumhivatkozás = null; // létrejön egy hivatkozás változó

A következő lépés az Osztálynév típusú objektum létrehozása, és hivatkozásának bemásolása az objektumhivatkozás változóba:

objektumhivatkozás = new Osztálynév(); // létrejön és inicializálódik az objektum

A fenti két lépést egyben is elvégezhetjük:

Osztálynév objektumhivatkozás = new Osztálynév(); // létrejön és inicializálódik

// az objektum

A new után üres zárójelpárral lezárt osztálynév valójában egy speciális metódus, a konstruktor hívását jelenti, amelynek feladata az objektumpéldány inicializálása. Amíg más konstruktort nem definiálunk, addig a fordító egy paraméter nélküli, alapértelmezett (default) konstruktort biztosít számunkra, amely nulla értékekkel látja el az objektum inicializálatlan adatmezőit.

Amennyiben nincs szükségünk többé az objektumra, a referencia nullázásával a szemétgyűjtő rendszer (Garbage Collection, GC) tudomására hozhatjuk ezt:

objektumhivatkozás = null; // a hivatkozás változó nullázódik

A programok készítése során egy objektumhivatkozásra általában úgy tekintünk, mintha az maga az objektum lenne. Ez a szemlélet mindaddig használható, amíg a referencián keresztül érjük el az objektum tagjait. Amennyiben a referencia változót értékadásban vagy metódus argumentumaként szerepeltetjük, tudnunk kell, hogy eközben az objektummal semmi sem történik, hiszen csak a hivatkozása másolódik, vagy a hivatkozás változó adódik át.

Struktúrák esetén a fenti lépések némileg másképp, egyszerűbben használhatók.

Struktúranév struktúra; // létrejön a struktúrapéldány, azonban inicializálatlan

// inicializálás tagonként

struktúra.tag = érték;

// inicializálás konstruktorhívással

struktúra = new Struktúranév(); // nullázás

struktúra.tag = érték;

Példaként definiáljunk síkbeli pontok koordinátái tárolására alkalmas class és struct osztálytípust! (Egyszerűség kedvéért a későbbiekben az osztály kifejezés alatt a class típust értjük, míg struktúraként a struct típusokra hivatkozunk. A példákban a világoskék háttérszínű kódrészletek osztályokkal kapcsolatosak, míg struktúrákra vonatkozó megoldások világoszöld háttérrel rendelkeznek.) Megjegyezzük, hogy a saját típusok nevét nagybetűvel kezdjük, míg az adattagok nevét kisbetűvel.

class Pont { public double xPoz; public double yPoz; } struct Pont { public double xPoz; public double yPoz; }

Ha a programban egy új osztály leírását a már meglévő osztályokon kívül kívánjuk elhelyezni, akkor ezt ajánlott a meglévő osztályok névterével azonos (nevű) névtérben megtenni. (Ezzel elkerülhetjük az osztályra való hivatkozás körüli bonyodalmakat.) Felhívjuk a figyelmet arra, hogy Windows Forms alkalmazások esetén a Visual Studio megköveteli, hogy a form osztálya legyen az első osztály a fájlban.

Másik megoldásként az új osztályt abban az osztályban is megadhatjuk, amelynek metódusaiból használni kívánjuk azt. Az ilyen ún. beágyazott (nested) osztályokat a metódusokon kívül kell elhelyeznünk.

namespace WindowsFormsAlkalmazás { public partial class Form1 : Form { public Form1() { InitializeComponent(); namespace WindowsFormsAlkalmazás { public partial class Form1 : Form { class Pont { public double xPoz;

} } class Pont { public double xPoz; public double yPoz; } } public double yPoz; } public Form1() { InitializeComponent(); } } }

A fenti típusokat csak akkor tudjuk használni a koordináták tárolására, ha változókat hozunk létre segítségükkel a metódusokban.

{ Pont cp1 = null; // hivatkozás sehova cp1 = new Pont(); // (0,0) Pont cp2 = new Pont(); // (0,0) } { Pont sp1; // (?, ?) sp1 = new Pont(); // (0,0) Pont sp2 = new Pont(); // (0,0) }

7.3 Adatmezők az osztályokban

Az objektum- és a struktúratagok elérésére a pont (.) operátort használjuk. Közvetlenül az pont jobb oldalán áll a típus (statikus tagok esetén), illetve a változó (nem statikus tagoknál) neve, míg a bal oldalán az elérni kívánt tag (konstans, adatmező, metódus, tulajdonság) azonosítója.

Példaként használjuk az előző részben elkészített pont típusokat és a segítségükkel deklarált objektumot valamint struktúrákat! A változóknak történő értékadás felhasználásával szemléltetjük a referencia- és az értéktípusok közötti alapvető különbségeket.

Mivel az osztályok (class) referenciatípusok, a velük definiált változók (cp1, cp2) csupán egy hivatkozást tárolnak arra az objektumra, amit a new felhasználásával hozunk létre a program halom területén (heap).

Amennyiben egy objektumra már nincs szükségünk, annak hivatkozását lenullázva (null) átadjuk annak adatterületét az automatikus szemétgyűjtőnek (GC), ami valamikor megszünteti az objektumot.

Ezzel ellentétben a struktúrák (struct) értéktípusok, így az általuk létrehozott változók (sp1, sp2) – hasonlóan az int, double stb. típusú változókhoz – közvetlenül értéket tárolnak. A metódusokon belül definiált struktúra változók a program veremterületén (stack) jönnek létre, és a programblokkból való kilépéskor megszűnnek.

{ Pont cp1 = new Pont(); // (0,0) cp1.xPoz = 12; // (12, 0) cp1.yPoz = 23; // (12,23) { Pont sp1 = new Pont(); // (0,0) sp1.xPoz = 12; // (12, 0) sp1.yPoz = 23; // (12,23)

42

Pont cp2 = null; // csak hivatkozás cp2 = cp1; cp2.xPoz += 7; // (19,23) // a cp1, cp2 objektumok értéke (19,23) cp1 = cp2 = null; // eldobjuk az objektumot } Pont sp2; // (?,?) sp2 = sp1; // (12,23) sp2.xPoz += 7; // (19,23) // az sp1 értéke (12,23) }

A fenti műveletek eredménye jól illusztrálja a referencia- és az értéktípusok közötti különbségeket. Mivel referencia változók közötti értékadás során a hivatkozások másolódnak, a cp2 = cp1; értékadást követően cp1 és cp2 ugyanarra az objektumra hivatkoznak.

sp21923cp2cp1sp11223VeremterületHalomterület1923

Ezzel ellentétben az sp2 = sp1; értékadás során az sp1 struktúrában tárolt adatok másolódnak át az sp2 struktúrába.

A csak adatmezőket tartalmazó osztályokkal általában struktúrákat készítünk, azonban metódusok hozzáadásával a programozási munka ekkor is kényelmesebbé tehető. Struktúráknál további szempont a kis adatmennyiség (≤16 bájt), és az, hogy ezek az adatok lehetőleg ne változzanak túl gyakran.

A következő példában a Pont típusú struktúrákat valamint objektumokat tömbben tárolva síkbeli háromszögeket készítünk véletlen koordinátákkal:

public partial class Form1 : Form { public Form1() { InitializeComponent(); } private void button1_Click(object sender, EventArgs e) {

43

Random rnd = new Random(); Pont[] sháromszög = new Pont[3]; // Pont struktúrák tömbje for (int i = 0; i < sháromszög.Length; i++) { sháromszög[i].xPoz = rnd.NextDouble() * 2014; sháromszög[i].yPoz = rnd.NextDouble() * 2014; } } } struct Pont { public double xPoz; public double yPoz; }

public partial class Form1 : Form { public Form1() { InitializeComponent(); } private void button1_Click(object sender, EventArgs e) { Random rnd = new Random(); Pont[] cháromszög = new Pont[3]; // hivatkozások tömbje for (int i = 0; i < cháromszög.Length; i++) { cháromszög[i] = new Pont(); // a Pont objektum létrehozása cháromszög[i].xPoz = rnd.NextDouble() * 2014; cháromszög[i].yPoz = rnd.NextDouble() * 2014; }

44

} } class Pont { public double xPoz; public double yPoz; }

Minden szónál szemléletesebb, ha megnézzük a kétféle megoldás során létrejövő tömbök elhelyezkedését a memóriában:

56.2339.5cháromszög345.31654.7VeremterületHalomterület1223.342345.671223.342345.67345.31654.756.2339.5sháromszög

7.4 Az adatmezők inicializálása a konstruktorokban

Mint láttuk, az objektumok létrehozásakor kötelező, míg a struktúrák esetén opcionális a paraméter nélküli, inicializáló metódus, az alapértelmezett (default) konstruktor (constructor) meghívása. Ez a metódus alapértelmezés szerinti értékekkel (0, 0.0, false, null stb.) tölti fel az objektum/struktúra adatmezőit. Amennyiben valamely objektumtag rendelkezik kezdőértékkel, a fordító által készített alapértelmezett konstruktor nem nullázza a kezdőértékkel ellátott adatmezőket.

A túlterhelés mechanizmusát alkalmazva, különböző paraméterezésű konstruktorokkal láthatjuk el az osztályainkat és a struktúráinkat. Konstruktorok készítésénél az alábbi szabályokat kell figyelembe vennünk:

 Nevének meg kell egyeznie az osztály/struktúra nevével.

 Nincs visszatérési típusa, a void sem használható.

 Egy osztály alapértelmezett konstruktorát csak akkor készíti el a fordító, ha nincs az osztálynak általunk definiált konstruktora.

 Ezzel szemben a struktúrák alapértelmezett konstruktora mindig létezik, így a struktúrákhoz nem készíthetünk default konstruktort.

 Ha a paraméterlistán a paraméterek neve megegyezik a mezőnevekkel, akkor a konstruktoron belül a this kulcsszó segítségével különböztetjük meg az osztály mezőit az azonos nevű paraméterektől.

 A konstruktorok egymást is hívhatják, ha a konstruktorfejben a kettőspont és this szó után zárójelek között felsoroljuk a hívási argumentumokat.

 A struktúrák konstruktoraiban minden adattagot (a readonly tagot is) kötelező inicializálni, míg az osztályok esetén az inicializálatlan adattagok lenullázódnak vagy megtartják a kezdőértéküket, amennyiben rendel-keznek ilyennel.

Az elmondottak alapján a Pont típusainkat paraméteres konstruktorokkal bővítjük:

class Pont { public double xPoz ; public double yPoz ; public Pont() : this(0, 0) { } public Pont(double xy) : this(xy, xy) { } public Pont(double xPoz, double yPoz) { this.xPoz = xPoz; this.yPoz = yPoz; } } struct Pont { public double xPoz; public double yPoz; public Pont(double xy) : this(xy, xy) { } public Pont(double xPoz, double yPoz) { this.xPoz = xPoz; this.yPoz = yPoz; } }

A konstruktorok hívását az alábbi utasításokban alkalmazzuk:

{ Pont cp1 = new Pont(); // (0,0) Pont cp2 = new Pont(7); // (7,7) Pont cp3 = new Pont(12,23); // 12,23) } { Pont sp1 = new Pont(); // (0,0) Pont sp2 = new Pont(7); // (7,7) Pont sp3 = new Pont(12, 23); // 12,23) }

A programban bárhol használhatjuk a new Pont(12,23) formájú konstruktorhívásokat, amelyek osztályok esetén objektumok születésével és inicializálásával járnak, a kifejezések értéke pedig objektumreferencia. Struktúra esetében a konstruktorhívás kifejezés értéke egy struktúraérték.

7.5 További metódusok készítése

Általánosságban elmondhatjuk, hogy az osztályok (a struktúrák is) névvel ellátott mezőkben tárolt adatok, és névvel azonosított metódusok gyűjteménye, mely metódusok az adatokon végeznek műveleteket. A metódusok készítésével és hívásával kapcsolatos ismereteket a 6. fejezetben foglaltuk össze, most csak néhány gondolattal és példával bővítjük az elmondottakat.

A metódusok fontos jellemzője, hogy csak a működésükhöz szükséges adatok egy részét kapják meg paraméterként, mivel az adatok többsége az objektum adatmezőiben tárolódnak. A metódusok nevét az osztálynevekhez hasonlóan nagybetűvel kezdjük.

A metódusokat több szempont szerint is csoportosíthatjuk.

 konstruktorok – inicializáló metódusok,

 destruktor – az objektum megszűnése előtti pillanatokban meghívódó metódus (amit ritkán használunk),

 új értéket beállító – (gyakran a Set szóval kezdődik a nevük),

 értéklekérdező – (általában a Get szócska vezeti be a nevüket),

 az algoritmusok megvalósításához szükséges általános metódusok.

46

Az elérési (Set/Get) metódusokra azért van szükség, mivel az objektum-orientált elveknek megfelelően egy objektum adattagjaihoz nem engedünk közvetlen hozzáférést, így azokat private (vagy protected) kulcsszóval definiáljuk. Ezzel szemben a metódusok többségét nyilvánossá tesszük (public).

Az osztályok minden (nem statikus) metódusának van egy nem látható paramétere – a this objektumreferencia. Mivel egy adott osztály metódusait megosztva használják az osztály objektumai, a this paraméter mondja meg, hogy melyik példányhoz tartozó adatmezőkkel kell dolgoznia a metódusnak. Az egymásból hívott metódusok esetén a this automatikusan továbbadódik. (Megjegyezzük, hogy a this paraméter a struktúrák metódusaiban is elérhető.)

Mivel az osztályok és a struktúrák esetén teljesen megegyeznek az általános célú metódusok használatának szabályai, csak a Pont osztályt bővítjük metódusokkal:

class Pont { // Adatmezők private double xPoz; private double yPoz; // Konstruktorok public Pont() : this(0, 0) { } public Pont(double xy) : this(xy, xy) { } public Pont(double xPoz, double yPoz) { this.xPoz = xPoz; this.yPoz = yPoz; } // Elérési metódusok public double GetX() { return xPoz; } public double GetY() { return yPoz; } public void SetX(double x)

47

{ xPoz = x; } public void SetY(double y) { yPoz = y; } // Általános metódusok public double Távolság(Pont p2) { return Math.Sqrt(Math.Pow(xPoz - p2.xPoz, 2) + Math.Pow(yPoz - p2.yPoz, 2)); } public void Elmozgat(double dx, double dy) { xPoz += dx; yPoz += dy; } }

A Pont osztály használatát egy nyomógomb eseménykezelő metódusában mutatjuk be:

private void button1_Click(object sender, EventArgs e) { Pont p1 = new Pont(12, 23); Pont origo = new Pont(0, 0); p1.SetX(p1.GetX() / 4); p1.Elmozgat(0, -19); MessageBox.Show(p1.Távolság(origo).ToString()); }

7.6 Tulajdonságok és indexelők

48

A fenti példában ún. elérési metódusokat alkalmaztunk a private adatmezők kezelésére. Ez a megoldás teljes egészében megfelel az objektum-orientált felfogásnak, hisz az adatok csak ellenőrzött módon változhatnak meg. A C#-ban a tulajdonságok (properties) nyelvi szinten történő támogatásával, automatikussá vált az elérési metódusok létrehozása és hívása:

elérési előírás típus Tulajdonságnév

{

get

{

return adatmező;

}

set

{

adatmező = value;

}

}

Amikor a tulajdonság nevére értékadást hajtunk végre, a fordító meghívja a set metódust, melynek value paraméterében átadja a beállítandó értéket. A tulajdonság értékének lekérdezésekor pedig a get metódus aktivizálódik. A get metódus elhagyásával ún. csak írható, míg a set elhagyásával csak olvasható tulajdonság készíthető. A tulajdonságok nevét a metódusokhoz hasonlóan nagybetűvel kezdjük.

A tulajdonságok bevezetésével az előző rész példaprogramja sokkal egyszerűbbé, olvashatóbbá válik:

class Pont { // Adatmezők private double xPoz; private double yPoz; // Konstruktorok public Pont() : this(0, 0) { } public Pont(double xy) : this(xy, xy) { } public Pont(double xPoz, double yPoz) { this.xPoz = xPoz; this.yPoz = yPoz; } // Tulajdonságok public double X { get { return xPoz; } set { xPoz = value; } }

public double Y { get { return yPoz; } set { yPoz = value; } } // Általános metódusok public double Távolság(Pont p2) { return Math.Sqrt(Math.Pow(xPoz - p2.xPoz, 2) + Math.Pow(yPoz - p2.yPoz, 2)); } public void Elmozgat(double dx, double dy) { xPoz += dx; yPoz += dy; } }

Az Pont osztály objektumának kezelését végző kódrészlet valóban sokkal érthetőbb lett:

private void button1_Click(object sender, EventArgs e) { Pont p1 = new Pont(12, 23); Pont origo = new Pont(0, 0); p1.X = p1.X / 4; // vagy p1.X /= 4; p1.Elmozgat(0, -19); MessageBox.Show(p1.Távolság(origo).ToString()); }

A C# 3.0 (Visual Studio 2008) változatában vezették be az automatikusan implementált tulajdonságokat. Ekkor a private adatmezők létrehozását és közvetlen elérését a fordítóra bízzuk, valamint az osztály metódusaiból is ezekre a tulajdonságokra hivatkozunk. Az ilyen tulajdonságok definíciója:

□elérési előírás□ típus Tulajdonságnév { get; set; }

A get és a set kulcsszavak egyike itt is elhagyható. A megoldás hátránya, hogy elveszítjük az adatok ellenőrzésének, esetleges konverziójának lehetőségét, és az adatmezők közvetlen elérésének elvesztésével az osztály metódusai is lassabban működnek.

A megoldás nagy előnye, hogy az objektum-orientáltság előírásainak megtartása mellet, egyszerűbb lesz az osztályunk forráskódja, mint ahogy az a Pont osztály esetén látható. (A Pont osztály használatában nem indokolt semmilyen változtatás.)

class Pont { // Konstruktorok public Pont() : this(0, 0) { } public Pont(double xy) : this(xy, xy) { } public Pont(double xPoz, double yPoz) { X = xPoz; Y = yPoz; } // Automatikusan implementált tulajdonságok public double X { set; get; } public double Y { set; get;} // Általános metódusok public double Távolság(Pont p2) { return Math.Sqrt(Math.Pow(X - p2.X, 2) + Math.Pow(Y - p2.Y, 2)); } public void Elmozgat(double dx, double dy) { X += dx; Y += dy; } }

A C# nyelv további érdekes lehetősége az indexelők (indexers) használata, amikor az objektumokat a tömbökhöz hasonlóan indexeljük. Indexelők esetén index(ek) gyanánt tetszőleges típusú adatokat használhatunk:

elérési előírás típus this[indexlista]

{

get { }

set { }

}

Egy osztályhoz több indexelőt is készthetünk a this névvel, amennyiben az indexlista típusai különböznek egymástól (túlterhelés).

51

Bővítsük a Pont osztályt egy indexelővel, amely lehetővé teszi az x-koordináta 0, és az y-koordináta 1 indexszel való elérését! (Rossz indexérték esetén kiváltunk egy indexhatár-túllépés kivételt.) A teljes osztály helyett csak az előző alfejezet osztályába behelyezett indexelőt, valamint a használatát tartalmazó eseménykezelő metódust mutatjuk be.

class Pont { // … // Indexelő public double this[int n] { get { if (n == 0) return X; else if (n == 1) return Y; else throw new IndexOutOfRangeException(); } set { if (n == 0) X = value; else if (n == 1) Y = value; else throw new IndexOutOfRangeException(); } } // … }

private void button1_Click(object sender, EventArgs e) {

52

Pont p1 = new Pont(12, 23); Pont origo = new Pont(0, 0); p1.X /= 4; // vagy p1.X /= 4; p1.Elmozgat(0, -19); p1[0] = p1[1] + 2; p1[1] *= 2; MessageBox.Show(p1.Távolság(origo).ToString()); }

7.7 Statikus osztálytagok

A static kulcsszóval olyan osztálytagokat készíthetünk, amelyek magához az osztályhoz tartoznak az objektumok helyett. (Az eddig bemutatott tagok közül csak a const konstansok, a paraméteres konstruktorok és az indexelők nem lehetnek statikusak. A const konstansok a static előírás nélkül is statikus tagjai az osztályoknak.) A statikus konstruktor kivételével a static tagok a szokásos elérési előírásokkal rendelkezhetnek.

Ha a static módosító az osztályfejben szerepel, akkor az osztály minden tagját statikusnak kell definiálni, és az ilyen osztályt nem lehet példányosítani. (Struktúrák esetén ez a megoldás nem használható.)

A statikus tagok eléréséhez a pont operátor bal oldalán a tagot tartalmazó osztály nevét kell megadnunk. Objektum-hivatkozások és struktúraváltozók neveivel a statikus tagok nem érthetők el!

A statikus metódusok (az ún. osztály szintű metódusok) nem rendelkeznek a this paraméterrel, így nem hivatkozhatnak az osztály nem statikus tagjaira.

A static elérésű adatmezők (az osztálymezők) az osztály betöltésekor, egyetlen példányban jönnek létre, így azokat az objektumok megosztva használják. A statikus adatmezők automatikus inicializálására statikus konstruktort készíthetünk, mely nem rendelkezik sem elérhetőséggel, sem típussal, sem paraméterekkel, neve pedig megegyezik az osztály nevével.

A statikus adattagok az osztályokban és a struktúrákban egyaránt kaphatnak kezdőértéket, azonban e nélkül nulla kezdőértékekkel mindig inicializálódnak.

A statikus tagokkal ellátott osztályokat gyakran használjuk a hagyományos programozási nyelvek globális függvényeit tartalmazó könyvtárak helyett, lásd a Math osztályt, továbbá jól használhatók globális változók tárolóhelyeként.

Az alábbi Trigon osztály Sin() és Cos() metódusai fokokkal és radiánokkal egyaránt működnek, amennyiben a hívás előtt beállítjuk a megfelelő Mértékegység tulajdonságot.

public partial class Form1 : Form { public Form1() { InitializeComponent(); }

53

private void button1_Click(object sender, EventArgs e) { double y = Trigon.Sin(Trigon.pi / 6); // radiánban számol Trigon.MértékEgység = Trigon.Egyseg.fok; // innentől fokokban számol y = Trigon.Sin(30); Trigon.MértékEgység = Trigon.Egyseg.radián; // innentől radiánban számol y = Trigon.Sin(Math.PI / 6); } } static class Trigon { private static double fok2Rad; public enum Egyseg { fok, radián }; public static Egyseg MértékEgység { get; set; } public static readonly double pi; // Statikus konstruktor static Trigon() { pi = Math.PI; fok2Rad = pi / 180; MértékEgység = Egyseg.radián; } public static double Sin(double x) { return Math.Sin(MértékEgység == Egyseg.radián ? x : fok2Rad * x); } public static double Cos(double x) { return Math.Cos(MértékEgység == Egyseg.radián ? x : fok2Rad * x); } }

54

A .NET rendszer egy sor struktúrát biztosít a fejlesztők számára, amelyek nem statikus és statikus metódusokkal, tulajdonságokkal egyaránt rendelkeznek: System.Drawing.Color, System.Drawing.Rectangle stb. Gyakori megoldás, hogy ugyanaz a művelet - nem statikus metódus hívásával - elvégezhető a struktúra belső változóin, illetve statikus metódussal az azonos típusú külső struktúrák adatain.

Az alábbi Pont struktúra jól szemlélteti, hogy a távolságszámítás művelet elérhető a belső és egy külső pont, illetve két külső pont adatai között is. Ez utóbbi metódus statikusként definiált. Érdekességként egy statikus számláló bevezetésével megtudhatjuk, hogy hány struktúrapéldány jött létre a paraméteres konstruktor hívásával.

public partial class Form1 : Form { public Form1() { InitializeComponent(); } private void button1_Click(object sender, EventArgs e) { Pont pont1 = new Pont(6, 8); Pont pont2 = new Pont(3, 4); double távolság1 = pont1.Távolság(pont2); double távolság2 = Pont.Távolság(pont1, pont2); MessageBox.Show("" + távolság1 + ", " + távolság2); MessageBox.Show("Pontok száma: " + Pont.GetSzámláló()); } } struct Pont { public double X { set; get; } public double Y { set; get; } static public int számláló; public Pont(double xPoz, double yPoz) : this() { X = xPoz; Y = yPoz; számláló++;

55

} public double Távolság(Pont p2) { return Math.Sqrt(Math.Pow(X - p2.X, 2) + Math.Pow(Y - p2.Y, 2)); } // Statikus metódusok static public double Távolság(Pont p1, Pont p2) { return Math.Sqrt(Math.Pow(p1.X - p2.X, 2) + Math.Pow(p1.Y - p2.Y, 2)); } public static int GetSzámláló() { return számláló; } }

Egy struktúra konstruktorában csak akkor hivatkozhatunk tulajdonságra, illetve hívhatunk metódust, ha a struktúra adatterülete már teljes egészében inicializált. Ezt egyszerűen biztosíthatjuk, ha a konstruktorfejben megadjuk az alapértelmezett konstruktor hívását.

56

8. Objektum-orientált programozás

Az előző fejezetben megismerkedtünk az osztályok készítésének alapvető szabályaival, az adatok és a rajtuk végzendő műveletek egybeépítésével (encapsulation), valamint az adatrejtés (data hiding) eszközeivel. Áttekintettük, hogyan tudunk feladatokat egymástól független osztályokkal megoldani. Az objektum-orientált programozás adta lehetőségek azonban nem merülnek ki ebben.

A problémák objektum-orientált feldolgozása során általában egy új programépítési módszert, a származtatást (öröklés, inheritance) alkalmazzuk. A származtatás lehetővé teszi, hogy már meglévő osztályok adatait és műveleteit új megközelítésben alkalmazzuk, illetve a feladat igényeinek megfelelően módosítsuk, bővítsük azokat. A problémákat így nem egyetlen (nagy) osztállyal, hanem osztályok egymásra épülő rendszerével (hierarchiájával) oldjuk meg.

Az öröklés az objektum-orientált C# nyelv egyik legfontosabb sajátossága. Ez a mechanizmus lehetővé teszi, hogy meglévő osztályból kiindulva, új osztályt hozzunk létre (származtassunk). A származtatás során az új osztály örökli a meglévő (ős)osztály nyilvános (public, internal) és védett (protected, protected internal) tagjait, amelyeket aztán az új osztály sajátjaként használhatunk. Azonban az új osztállyal bővíthetjük is a meglévő osztályt, új adatmezőket, tulajdonságokat és metódusokat definiálhatunk, illetve újraértelmezhetjük (lecserélhetjük) az öröklött, de működésükben elavult metódusokat (polimorfizmus, polymorphism).

A továbbiakban csak a class kulcsszóval létrehozott osztályokkal foglalkozunk, hiszen az értéktípusú struktúrák, legfeljebb objektum-alapú programépítést tesznek lehetővé.

8.1 Osztályok származtatása, öröklés

Az öröklés során a származtatott osztály (utódosztály) örökli (megkapja) egy előzőleg definiált osztály adatmezőit és metódusait. Azt az osztályt, amelytől a származtatott osztály örököl, alaposztálynak (ősosztály) nevezzük. A származtatott osztály szintén lehet további osztályok alaposztálya, lehetővé téve ezzel osztályhierarchia kialakítását.

Az alábbi példában az AOsztály az ősosztály, a BOsztály pedig a származtatott osztály, amely teljes egészében magában foglalja az AOsztályt.

AOsztályAOsztályBOsztály

```
class AOsztály { // ... } class BOsztály : AOsztály { // ... }
```

A származtatott osztály az alaposztály minden tagját örökli, azonban az alaposztályból csak a public (internal) és protected (védett) (protected internal) tagokat éri el sajátjaként.

A védett (protected) elérés kettős viselkedést takar. Privát hozzáférést jelent az adott osztály felhasználójának, aki objektumokat hoz létre vele, azonban nyilvános elérést biztosít az osztály továbbfejlesztőjének, aki új osztályt származtat belőle. A metódusokat általában public vagy protected hozzáféréssel adjuk meg, míg az adatmezők esetén a protected vagy a private elérést alkalmazzuk. (A privát hozzáféréssel a származtatott osztály metódusai elől is elrejtjük a tagokat.) A származtatott osztályban az öröklött tagokat új adatmezőkkel és metódusokkal is kiegészíthetjük.

A származtatás általános formájában a származtatott utódosztály neve után kettősponttal elválasztva az ősosztály (alaposztály) azonosítója következik:

```
elérési előírás class UtódOsztály : ŐsOsztály
{
// az osztály törzse
```

}

A .NET rendszerben minden osztály rendelkezik közvetlen őssel, még akkor is, ha ezt nem adjuk meg programunkban. Ez a közös ős a System.Object (object) osztály, amely egy sor hasznos metódust (ToString(), Equals(), GetType(), …) biztosít az osztályoknak:

class AOsztály : System.Object // vagy object { // ... } class BOsztály : AOsztály { // ... }

Felhívjuk a figyelmet arra, hogy az alaposztálybeli elérhetőségüktől függetlenül a konstruktorok nem öröklődnek.

Az alábbi – síkbeli és térbeli pontokat definiáló – példában bemutatjuk a származtatás alkalmazását. A kialakított osztály-hierarchia igen egyszerű (a piros nyíl a közvetlen alaposztályra mutat):

Pont2DPont3D

class Pont2D { protected double xPoz; protected double yPoz;

58

public Pont2D() : this(0, 0) { } public Pont2D(Pont2D p) { xPoz = p.xPoz; yPoz = p.yPoz; } public Pont2D(double x, double y) { xPoz = x; yPoz = y; } public double Távolság(Pont2D p2) { return Math.Sqrt(Math.Pow(xPoz - p2.xPoz, 2) + Math.Pow(yPoz - p2.yPoz, 2)); } public void Elmozgat(double dx, double dy) { xPoz += dx; yPoz += dy; } public string Lekérdez2D() { return "(" + xPoz + ", " + yPoz+")"; } } class Pont3D : Pont2D { protected double zPoz;

59

public Pont3D() : this(0, 0, 0) { } public Pont3D(Pont3D p) : base() { xPoz = p.xPoz; yPoz = p.yPoz; zPoz = p.zPoz; } public Pont3D(double x, double y, double z) : base(x, y) { zPoz = z; } public double Távolság(Pont3D p2) { return Math.Sqrt(Math.Pow(xPoz - p2.xPoz, 2) + Math.Pow(yPoz - p2.yPoz, 2) + Math.Pow(zPoz - p2.zPoz, 2)); } public void Elmozgat(double dx, double dy, double dz) { Elmozgat(dx, dy); zPoz += dz; } public string Lekérdez3D() { return "(" + xPoz + ", " + yPoz + ", " + zPoz + ")"; } }

Az osztályok felhasználását egy eseménykezelő metódusban mutatjuk be:

private void button1_Click(object sender, EventArgs e)

60

{ Pont2D p1 = new Pont2D(12,23); Pont2D p2 = new Pont2D (p1); Pont2D p3 = new Pont2D(); Pont3D q1 = new Pont3D(7,29,80); Pont3D q2 = new Pont3D (q1); Pont3D q3 = new Pont3D(); MessageBox.Show(p1.Lekérdez2D() + "\t" + q1.Lekérdez3D()); p2.Elmozgat(-2, 7); q2.Elmozgat(1, 2, 3); double d1 = p2.Távolság(p1); double d2 = q2.Távolság(q1); MessageBox.Show("" + d1 + "\t" + d2); }

A C# nyelv közvetlenül csak egyetlen osztálytól való származtatást, azaz az egyszeres öröklést (single inheritance) támogatja. Egyszeres öröklés során valamely osztálynak legfeljebb egy közvetlen őse, és tetszőleges számú utódja lehet. Az öröklés több lépésben való alkalmazásával igazi fastruktúra (osztály-hierarchia) alakítható ki.

AlakzatHáromszögSzabályos háromszögTéglalapNégyzetOválisEllipszisKörNégyoldalú

8.1.1 Az alaposztály inicializálása és az öröklött tagok elérése

Amikor egy osztályt egy másik osztálytól származtatunk, akkor az utód osztályból a base referenciával mindig hivatkozhatunk a közvetlen ős elérhető tagjaira. (Emlékeztetőül, a this hivatkozás az aktuális objektumpéldányt jelöli). A base hivatkozást ritkán

használjuk, hiszen az öröklés során, a védett és a nyilvános tagok az utód osztály saját tagjai lesznek.

A base hivatkozás fontos alkalmazási területe a közvetlen ős konstruktorának hívása az utód konstruktorából, például:

class A {

61

public A() { } public A(int a) { } } class B : A { public B() : base() { } // mint: public B() { } public B(int a, int b) : base(a) { } }

Ha a konstruktorfejben nem szerepel a base(argumentumok) hívás, akkor a fordító beszúr oda egy base() hivatkozást, ami aktivizálja az ősosztály paraméter nélküli konstruktorát, amennyiben az létezik.

A származtatott osztályban deklarált tagok elfedhetik a közvetlen őstől örökölt azonos nevű tagokat. Ekkor a base hivatkozás segítségével érhetjük el az elfedett (hided) adatmezőket, tulajdonságokat, illetve hívhatjuk az ős nem látható metódusait (base.a, base.Kiir()).

class A { protected int a; public A() { } public A(int a) { } public string Kiir() { return "" + a; } } class B : A { protected int a; public B() : base() { } // mint: public B() { } public B(int a, int b) : base(a) { } public int Számol() { return base.a * a; } public new string Kiir() { return base.Kiir() + "\t" + a; } }

Metódusok elfedésekor a fordító figyelmeztet, és javasolja a new módosító használatát, amivel megerősíthetjük, hogy tudatosan alkalmaztuk ezt a megoldást, mint a B osztály Kiir() metódusa esetén.

8.2.Többalakúság (polimorfizmus)

62

A polimorfizmus (polymorphism) megmondja, hogy az C# fordítója, illetve a futtató rendszer miként kezeli az azonos nevű műveleteket (metódusokat) egy osztályon belül, illetve az osztályhierarchiában. Kétféle többalakúságot különböztetünk meg:

 Ha ugyanban az osztályban több metódust deklarálunk azonos névvel, azonban különböző paraméterlistával futás idejű (compile time) polimorfizmust valósítunk meg. Ennek további nevei a korai vagy statikus kötés (early binding), illetve a túlterhelés (overloading), amikor a hivatkozásokat az aktuális osztály elemeivel oldja fel a fordító. (A korai kötés működésére a 6.3. és a 8.1 fejezetekben láthatunk példákat.)

 Amikor az osztályhierarchia különböző szintjein deklarálunk azonos nevű és azonos paraméterezésű metódusokat, lehetőségünk adódik a futás idejű (run time) polimorfizmus alkalmazására. Ennek is több elnevezése használatos: késői vagy dinamikus kötés (late binding), illetve felülbírálat (overriding).

Mivel öröklés során gyakran specializáljuk az leszármazott osztályt, szükséges lehet, hogy bizonyos örökölt műveletek másképp működjenek. A futás idejű polimorfizmusnak köszönhetően egy objektum attól függően, hogy az osztály-hierarchia mely szintjén lévő osztály példánya, ugyanarra az üzenetre másképp reagálhat. Az pedig, hogy az üzenet hatására melyik metódus hívódik meg az öröklési láncból, csak a program futása közben derül ki.

8.2.1 Virtuális öröklés

Korábban már láttuk, hogy az öröklés során a leszármazott osztály örökli az őse minden adatát és műveleteit. Ezek az öröklött metódusok minden további nélkül használhatók a származtatott osztály objektumaival is, hiszen azok magukban foglalják az ősüket.

Példaként tekintsük a különböző alakzatok területét számító osztályok hierarchiáját! A hierarchia mindhárom szintjén deklaráljuk a területszámító metódust.

public class Alakzat { protected double sugár; public Alakzat(double r) { sugár = r; } public double Terület() { return 0; } public double R { set { sugár = value; } } } public class Kör : Alakzat { public Kör(double r) : base(r) { } public new double Terület() { return Math.PI * sugár * sugár; }

63

} public class Gömb : Kör { public Gömb(double r) : base(r) { } public new double Terület() { return 4 * base.Terület(); } }

Egyetlen Gömb típusú objektumot hozunk létre, amelyre aztán Alakzat és Kör típusú referenciákkal is hivatkozunk. A felhasználás bemutató kódrészletből látszik, hogy mindig az adott hivatkozásnak megfelelő területszámító metódus fut le, amit a korai kötés következménye.

64

private void button1_Click(object sender, EventArgs e) { Gömb g = new Gömb(10); Kör k = g; Alakzat a = g; MessageBox.Show("" + a.Terület() + "\t" + k.Terület() + "\t" + g.Terület()); }

Mivel a g objektum egy gömb, azt szeretnénk elérni, hogy mindhárom esetben a gömbnek megfelelő számítás hajtódjon végre. Ehhez az Alakzat osztály Terület() metódusát virtuálissá kell tennünk a virtual módosítóval, továbbá a Kör és a Gömb osztályok Terület() metódusait pedig felül kell bírálnunk az override módosítóval. Ezekkel a lépésekkel hagyjuk érvényesülni a késő kötést.

public class Alakzat { protected double sugár; public Alakzat(double r) { sugár = r; } public virtual double Terület() { return 0; } public double R { set { sugár = value; } } } public class Kör : Alakzat { public Kör(double r) : base(r) { } public override double Terület() { return Math.PI * sugár * sugár; } } public class Gömb : Kör { public Gömb(double r) : base(r) { }

65

public override double Terület() { return 4 * base.Terület(); } }

Az üzenetkezelőben semmit sem változtatunk, mégis a várt eredményt kapjuk:

(Megjegyezzük, hogy a base.Terület() hivatkozás esetén továbbra is a korai kötés érvényesül.)

Összefoglalva elmondhatjuk, hogy a dinamikus kötés segítségével az örökölt metódusokat újakkal helyettesíthetjük, függetlenül attól, hogy milyen ősosztály referenciával is hivatkozunk az objektumra. A lecserélhetőséget az ősben a virtual (virtuális) módosítóval kell jeleznünk, míg a csere az utódban az override (hatástalanít) módosító hatására megy végbe. A polimorfizmus működéséhez elengedhetetlen, hogy a virtuális metódus és a cserélő metódus fejléce teljesen azonos legyen.

Nézzünk most egy olyan alkalmazását az Alakzat, Kör és Gömb osztályoknak, ahol a helyes működés egyetlen feltétele a késői kötés! Tömbben szeretnénk tárolni a különböző típusú alakzatok objektumhivatkozásait, majd a tömb bejárásával szeretnénk megjeleníteni az egyes alakzatok területét (például egy CAD rendszer modelljének tárolásakor). A tömb elemeinek a közös ős típusával kell deklarálnunk, hiszen így mindhárom típusú objektumra hivatkozhatnak az elemek:

Csak a nyomógomb eseménykezelőjének tartalmát kell a feladat megoldásához megváltoztatnunk:

private void button1_Click(object sender, EventArgs e) { Alakzat[] modell = new Alakzat[3]; modell[0] = new Gömb(10); modell[1] = new Alakzat(23); modell[2] = new Kör(5); string s = string.Empty; foreach (Alakzat elem in modell) s += elem.ToString() + ": " + elem.Terület() + "\n"; MessageBox.Show(s); }

8.2.2. Absztrakt metódusok és osztályok

66

Az objektum-orientált fejlesztés során olyan osztályokat is kialakíthatunk, melyeket csak továbbfejlesztésre, származtatásra lehet használni, és vele objektumpéldány nem készíthető, azonban objektum-referencia igen. (Ilyen lehet például az előző rész Alakzat osztálya.) Ehhez egyszerűen az abstract módosítót kell az osztályfejben elhelyezni, és az osztály absztrakttá válik.

Az absztrakt osztályok további jellegzetessége, hogy bizonyos műveletek (metódusok), amelyek szükségesek az osztály működéséhez általában nincsenek kidolgozva – a fejsorukat pontosvessző zárja és hiányzik a törzsük. Az ilyen metódusok fejsorában szintén az abstract kulcsszót kell alkalmazni (a virtual helyett). Az absztrakt metódusokat az utódosztályban meg kell valósítanunk, hisz ellenkező esetben az is absztrakttá válik. A megvalósító metódusokat override módosítóval kell ellátnunk.

Ha egy osztályban definiáltunk absztrakt metódust, akkor ezt az osztályfejben is jelezni kell az abstract módosítóval. Az absztrakt osztályban természetesen védett (belső és nyilvános) elérésű adatmezőket is elhelyezhetünk. Privát hozzáférésű adatmezők is szerepelhetnek benne, hiszen nem szükséges, hogy az abstract osztály minden metódusa absztrakt legyen.

Nézzük, hogyan tehető absztrakttá az előző példa Alakzat osztálya! Az átalakításokat követően a new Alakzat(5) kifejezések többé már nem használhatók.

public abstract class Alakzat { protected double sugár; public Alakzat(double r) { sugár = r; } public abstract double Terület(); public double R { set { sugár = value; } } }

8.2.3 Az öröklési lánc lezárása

Valamely osztály fejlécében a sealed (lepecsételt) módosítót megadva, megakadályozhatjuk, hogy abból az osztályból új osztályt lehessen származtatni:

sealed class LezártOsztály

{

//…

}

Megjegyezzük, hogy a struktúrák alapértelmezés szerint sealed osztályok.

A sealed módosító másik alkalmazásával megakadályozhatjuk egy felülbírált (override) metódus további felülbírálását az öröklési láncban. Amennyiben a fenti példában a sealed módosítót szerepeltetjük a Kör osztály Terület() metódusánál az override mellett, a Gömb osztály Terület() metódusa többé nem lehet felülbírált.

67

8.3 Interfészek (interface)

Mint említettük a C# nyelv az egyszeres öröklést támogatja, vagyis minden osztálynak pontosan egy közvetlen őse lehet. A C# nyelv fejlesztői a többszörös öröklés bizonyos

hiányzó lehetőségeinek pótlására, lehetővé tették egy sokkal egyszerűbb és biztonságosabb megoldás, az interfészek (interface) használatát.

Az interfész egy előíráshalmaz. Amelyik osztály vagy struktúra megvalósítja azt, annak meg kell felelnie az előírásoknak. Az interfész deklarálhat metódusokat, tulajdonságokat, indexelőket és eseményeket, de azokat nem valósítja meg. Nem tartalmazhat azonban adatokat, még konstansokat sem. Egy interfésznek több más interfész is lehet őse, valamely osztály illetve struktúra pedig több interfészt is megvalósíthat.

Az interfész deklarációja sokban emlékeztet az osztályra, azonban csak nyilvános absztrakt metódusokat, tulajdonságokat, indexelőket és eseményeket tartalmazhat. Az abstract szó használata nem kötelező, hisz maga az interfész, illetve annak minden metódusa hallgatólagosan absztrakt. Az interfészek névét általában nagy I betűvel kezdjük:

interface InterfészNév

{

típus metódus1(paraméterlista1);

típus metódus2(paraméterlista2);

// …

}

Az interfészeket az osztálydefiníciókban az alaposztállyal együtt, illetve az ősosztály nélkül egyaránt megadhatjuk:

class Osztály : AlapOsztály, Interfész1, Interfész2,...

{

// az Osztály tagjai

}

class Osztály : Interfész1, Interfész2,...

{

// az Osztály tagjai

}

Példaként alakítsuk át az előző alfejezet alakzatok területét számító példájában az Alakzat osztályt IAlakzat interfésszé, amely kötelezővé teszi a Terület() metódus és az csak írható R tulajdonság megvalósítását!

public interface IAlakzat { double Terület(); double R { set; } } public class Kör : IAlakzat { protected double sugár; public Kör(double r) { sugár = r; } public virtual double Terület() {

return Math.PI * sugár * sugár; } public double R { set { sugár = value; } } } public class Gömb : Kör { public Gömb(double r) : base(r) { } public override double Terület() { return 4 * base.Terület(); } }

A felhasználást bemutató programrész is módosul, mivel az interfész segítségével fogjuk csokorba az alakzatokat. Minden olyan alakzat hivatkozása eltárolható a modell tömbben, amely megvalósítja az IAlakzat interfész előírásait.

private void button1_Click(object sender, EventArgs e) { IAlakzat[] modell = new IAlakzat[3]; modell[0] = new Gömb(7); modell[1] = new Kör(23); modell[2] = new

Kör(12); string s = string.Empty; foreach (IAlakzat elem in modell) s += elem.ToString() + ": " + elem.Terület() + "\n"; MessageBox.Show(s); }

F1. Bevezetés az objektum-orientált programozásba

Az objektum-orientált programozás (OOP) olyan modern programozási módszertan (paradigma), amely a program egészét egyedi jellemzőkkel rendelkező, önmagukban is működőképes, zárt programegységek (objektumok) halmazából építi fel. Az objektum-orientált programozás a klasszikus strukturált programozásnál jóval hatékonyabb megoldást nyújt a legtöbb problémára, és az absztrakt műveletvégző objektumok kialakításának és újrafelhasználásának támogatásával nagymértékben tudja csökkenteni a szoftverek fejlesztéséhez szükséges időt.

Ez az anyag valóban csak egy bevezetés, és nem tér ki az OOP speciális megoldásaira. A Hallgatóra bízzuk a téma folytatását, hiszen még meg kell ismerkednie az objektum-orientált C# nyelv egyik legfőbb sajátosságával, az örökléssel. Ez a mechanizmus lehetővé teszi, hogy meglévő osztályokból kiindulva, új osztályt hozzunk létre (származtassunk). Ugyancsak önálló feldolgozásra vár az öröklés során megvalósított többalakúság (polimorfizmus), mely segítségével az öröklött, de elavult tagfüggvények lecserélhetők a származtatás során.

F1.1 Bevezetés az objektum-orientált világba

Az objektum-orientált programozás a „dolgokat” („objektumokat”) és köztük fennálló kölcsönhatásokat használja alkalmazások és számítógépes programok tervezéséhez. Ez a módszertan olyan megoldásokat foglal magában, mint a bezárás (encapsulation), a modularitás (modularity), a többalakúság (polymorphism) valamint az öröklés (inheritance).

Felhívjuk a figyelmet arra, hogy az OOP nyelvek általában csak eszközöket és támogatást nyújtanak az objektum-orientáltság elveinek megvalósításához.

F1.2 Alapelemek

Először ismerkedjünk meg az objektum-orientált témakör alapelemeivel! A megértéshez nem szükséges mély programozási ismeretek megléte.

Osztály (class)

Az osztály (class) meghatározza egy dolog (objektum) elvont leírását, beleértve a dolog jellemvonásait (attribútumok, mezők, tulajdonságok) és a dolog viselkedését (amit a dolog meg tud tenni, metódusok (módszerek), műveletek, funkciók).

Azt mondhatjuk, hogy az osztály egy tervrajz, amely leírja valaminek a természetét. Például, egy Teherautó osztálynak tartalmazni kell a teherautók közös jellemzőit (gyártó, motor, fékrendszer, maximális terhelés stb.), valamint a fékezés, a balra fordulás stb. képességeket (viselkedés).

Osztályok önmagukban biztosítják a modularitást és a strukturáltságot az objektum-orientált számítógépes programok számára. Az osztálynak értelmezhetőnek kell lennie a probléma területén jártas, nem programozó emberek számára is, vagyis az osztály jellemzőinek „beszédesnek” kell lenniük. Az osztály kódjának viszonylag önállónak kell lennie (bezárás – encapsulation). Az osztály beépített tulajdonságait és metódusait egyaránt az osztály tagjainak nevezzük (A C# nyelvben az adatmező, a tulajdonság, illetve a metódus kifejezéseket használjuk).

Objektum (object)

Az objektum az osztály megvalósítása (mintapéldája). A Teherautó osztály segítségével minden lehetséges teherautót megadhatunk, a tulajdonságok és a viselkedési formák felsorolásával. Például, a Teherautó osztály rendelkezik fékrendszerrel, azonban az énAutóm (objektum) fékrendszere elektronikus vezérlésű (EBS) vagy egyszerű légfékes is lehet.

Példány (instance)

Az objektum szinonimájaként az osztály egy adott példányáról is szokás beszélni. A példány alatt a futásidőben létrejövő aktuális objektumot értjük. Így elmondhatjuk, hogy az énAutóm a Teherautó osztály egy példánya. Az aktuális objektum tulajdonságértékeinek halmazát az objektum állapotának (state) nevezzük. Ezáltal minden objektumot az osztályban definiált állapot és viselkedés jellemez.

Az énAutóm objektum (a Teherautó osztály példánya)

Metódus (method)

Metódusok felelősek az objektumok képességeiért. A beszélt nyelvben az igéket hívhatjuk metódusoknak. Mivel az énAutóm egy Teherautó, rendelkezik a fékezés képességével, így a Fékez() ez énAutóm metódusainak egyike. Természetesen további metódusai is lehetnek, mint például az Indít(), a GáztAd(), a BalraFordul() vagy a JobbraFordul(). A programon belül egy metódus használata általában csak egy adott objektumra van hatással. Bár minden teherautó tud fékezni, a Fékez() metódus aktiválásával (hívásával) csak egy adott járművet szeretnénk lassítani.

A hagyományos autókban a sebességfokozatot (ami a kocsi egy jellemzője, adata) közvetlenül állítjuk be, a sebességváltó felhasználásával. Ez azonban magában hordozza annak lehetőségét, hogy rosszul váltva tönkretesszük a sebességváltó fogaskerekeit. Ehelyett elektronikus sebességváltást alkalmazva, az autó ellenőrzötten a megfelelő sebességfokozattal fog haladni.

Az ESebesség egy olyan viselkedés (metódus), ami a sebességfokozat beállítását biztonságossá teszi. Bizonyos nyelvekben az ilyen jellegű megoldásokat tulajdonságnak (property) hívjuk, és az adatmezőkhöz hasonlóan használjuk.

Üzenetküldés (message passing)

Az üzenetküldés az a folyamat, amelynek során egy objektum adatokat küld egy másik objektumnak, vagy “megkéri” a másik objektumot valamely metódusának végrehajtására. Az üzenetküldés szerepét jobban megértjük, ha egy teherautó szimulációjára gondolunk. Ebben egy sofőr objektum a „fékezz” üzenet küldésével aktiválhatja az énAutóm Fékez() metódusát, lefékezve ezzel a járművet. Az üzenetküldés szintaxisa igen eltérő a különböző programozási nyelvekben. C# nyelven a kódszintű üzenetküldést a metódushívás valósítja meg.

F1.2 Alapvető elvek

Egy rendszer, egy programnyelv objektum-orientáltságát az alábbi elvek támogatásával lehet mérni. Amennyiben csak néhány elv valósul meg, objektum-alapú rendszerről beszélünk, míg mind a négy elv támogatása az objektum-orientált rendszerek sajátja.

Bezárás, adatrejtés (encapsulation , data hiding)

A fentiekben láttuk, hogy az osztályok alapvetően jellemzőkből (állapot) és metódusokból (viselkedés) épülnek fel. Azonban az objektumok állapotát és viselkedését két csoportba osztjuk. Lehetnek olyan jellemzők és metódusok, melyeket elfedünk más objektumok elől, mintegy belső, privát (private, védett - protected) állapotot és viselkedést

létrehozva. Másokat azonban nyilvánossá (public) teszünk. Az OOP alapelveinek megfelelően az állapotjellemzőket privát eléréssel kell megadnunk, míg a metódusok többsége nyilvános lehet. Szükség esetén a privát jellemzők ellenőrzött elérésére nyilvános elérési metódusokat, C# nyelven tulajdonságokat (properties) készíthetünk.

Általában is elmondhatjuk, hogy egy objektum belső világának ismeretére nincs szüksége annak az objektumnak, amelyik üzenetet küld. Például, a Teherautó rendelkezik a Fékez() metódussal, amely pontosan definiálja, miként megy végbe a fékezés. Az énAutóm vezetőjének azonban nem kell ismernie, hogyan is fékez a kocsi.

Minden objektum egy jól meghatározott interfészt biztosít a külvilág számára, amely megadja, hogy kívülről mi érhető el az objektumból. Az interfész rögzítésével az objektumot használó, ügyfél alkalmazások számára semmilyen problémát sem jelent az osztály belső világának jövőbeni megváltoztatása. Így például egy interfészen keresztül biztosíthatjuk, hogy pótkocsikat csak a Kamion osztály objektumaihoz kapcsoljunk.

Öröklés (inheritance)

Öröklés során egy osztály specializált változatait hozzuk létre, amelyek öröklik a szülőosztály (alaposztály) jellemzőit és viselkedését, majd pedig sajátként használják azokat. Az így keletkező osztályokat szokás alosztályoknak (subclass), vagy származtatott (derived) osztályoknak hívni.

Például, a Teherautó osztályból származtathatjuk a Kisteherautó és a Kamion alosztályokat. Az énAutóm ezentúl legyen a Kamion osztály példánya! Tegyük fel továbbá, hogy a Teherautó osztály definiálja a Fékez() metódust és az fékrendszer tulajdonságot! Minden ebből származtatott osztály (Kisteherautó és a Kamion) örökli ezeket a tagokat, így a programozónak csak egyszer kell megírnia a hozzájuk tartozó kódot.

TeherautóKisteherautóKamion

Az öröklés menete

Az alosztályok meg is változtathatják az öröklött tulajdonságokat. Például, a Kisteherautó osztály előírhatja, hogy a maximális terhelése 20 tonna. A Kamion alosztály pedig az EBS fékezést teheti alapértelmezetté a Fékez() metódusa számára.

A származtatott osztályokat új tagokkal is bővíthetjük. A Kamion osztályhoz adhatunk egy Navigál() metódust. Az elmondottak alapján egy adott Kamion példány Fékez() metódusa EBS alapú fékezést alkalmaz, annak ellenére, hogy a Teherautó osztálytól egy hagyományos Fékez() metódust örökölt; valamint rendelkezik egy új Navigál() metódussal, ami azonban nem található meg a Kisteherautó osztályban.

Az öröklés valójában „egy” (is-a) kapcsolat: az énAutóm egy Kamion, a Kamion pedig egy Teherautó. Így az énAutóm egyaránt rendelkezik a Kamion és a Teherautó metódusaival.

A fentiekben mindkét származtatott osztálynak pontosan egy közvetlen szülő ősosztálya volt, a Teherautó. Ezt, a C# által is támogatott öröklési módot egyszeres öröklésnek (single inheritance) nevezzük.

Absztrakció (abstraction)

Az elvonatkoztatás a probléma megfelelő osztályokkal való modellezésével egyszerűsíti az összetett valóságot, valamint a probléma - adott szempontból - legmegfelelőbb öröklési szintjén fejti ki hatását. Például, az énAutóm az esetek nagy többségében Teherautóként kezelhető, azonban lehet Kamion is, ha szükségünk van a Kamion specifikus

jellemzőkre és viselkedésre, de tekinthetünk rá Járműként is, ha egy flotta elemeként vesszük számba. (A Jármű a példában a Teherautó szülő osztálya.)

Az absztrakcióhoz a kompozíción keresztül is eljuthatunk. Például, egy Autó osztálynak tartalmaznia kell egy motor, sebességváltó, kormánymű és egy sor más komponenst. Ahhoz, hogy egy Autót felépítsünk, nem kell tudnunk, hogyan működnek a különböző komponensek, csak azt kell ismernünk, miként kapcsolódhatunk hozzájuk (interfész). Az interfész megmondja, miként küldhetünk nekik, illetve fogadhatunk tőlük üzenetet, valamint információt ad arról, hogy az osztályt alkotó komponensek milyen kölcsönhatásban vannak egymással.

Interfészek (interface) alkalmazása

A járműiparban törvényi előírások határozzák meg azokat a képességeket, amelyekkel a járműveknek rendelkezniük kell a biztonságos közlekedés érdekében. Azok a járművek, amelyek megvalósítják ezeket az előírásokat, viselkedési formákat (metódusokat), biztonságosnak nevezhetők. Természetesen a régebbi gyártású autók nem rendelkeznek ezekkel a képességekkel. A törvény nem szól arról, hogy miként kell megvalósítani az előírásokat, ezt teljes mértékben a gyártókra bízza.

Az objektum-orientált fejlesztés során az interfészekkel adhatunk egy adott osztályhoz ilyen előírásokat. Mint láttuk, egy osztálynak pontosan egy ősosztálya lehet, azonban tetszőleges számú interfészt megvalósíthat.

Példaként tekintsük a BiztonságosJármű interfészt, amely két képesség előírását tartalmazza, az ABS (blokkolásgátló) és az ESP (elektronikus menet-stabilizátor) rendszerekét. Amennyiben ezek az előírások csak a kamionokra vonatkoznak, a fenti öröklési fa Kamion szintjén kell azokat megvalósítani.

Polimorfizmus (polymorphism)

A polimorfizmus lehetővé teszi, hogy az öröklés során bizonyos (elavult) viselkedési formákat (metódusokat) a származtatott osztályban új tartalommal valósítsunk meg, és az új, lecserélt metódusokat a szülő osztály tagjaiként kezeljük.

Példaként tegyük fel, hogy a Teherautó és a Kerekpár osztályok öröklik a Jármű osztály Gyorsít() metódusát. A Teherautó esetén a Gyorsít() parancs a GáztAd() műveletet jelenti, míg Kerekpár esetén a Pedáloz() metódus hívását. Ahhoz, hogy a gyorsítás helyesen működjön, a származtatott osztályok Gyorsít() metódusával felül kell bírálnunk (override) a Jármű osztálytól örökölt Gyorsít() metódust. Ez a felülbíráló polimorfizmus.

A legtöbb OOP nyelv a parametrikus polimorfizmust is támogatja, ahol a metódusokat típusoktól független módon, mintegy mintaként készítjük el a fordító számára. A C# nyelven általánosított típusok (generics) készítésével alkalmazhatjuk ezt a lehetőséget.

F1.3 Objektum-orientált C# programpélda

Végezetül nézzük meg az elmondottak alapján elkészített C# osztály kódját! Most legfontosabb az első benyomás, hiszen a részletekkel a jegyzet további fejezeteiben foglalkozunk. Az osztály kódját önálló Járművek.cs fájlban elhelyezve, és egy Windows Forms alkalmazás projekthez adva, elérhetjük annak elemeit.

using System.Windows.Forms; public interface BiztonságosJármű { void ABS(); void ESP(); } public class Teherautó { private int sebességFokozat = 0; // 1, 2, 3, 4, 5 előre, -1 hátra protected string gyártó; protected string motor; protected string fékrendszer; protected double maximálisTerhelés; public Teherautó(string gy, string m, string fek, double teher) { gyártó = gy; motor = m; fékrendszer = fek; maximálisTerhelés = teher; } public

void Indít() { ESebesség = 1; } public void GáztAd() { } public virtual void Fékez() { MessageBox.Show("A hagyományosan fékez."); } public void BalraFordul() { } public void JobbraFordul() { } public int ESebesség { get { return sebességFokozat; } set { if ((value > -1 && value < 6) || (value == -1 && sebességFokozat == 0)) sebességFokozat = value; } } } public class Kisteherautó : Teherautó { public Kisteherautó(string gy, string m, string fek) : base(gy, m, fek, 20) { } } public class Kamion : Teherautó, BiztonságosJármű { public Kamion(string gy, string m, string fek, double teher) : base(gy, m, fek, teher) { } public override void Fékez() { ABS(); MessageBox.Show("A EBS-sel fékez."); } public void Navigál() { ESP(); } public void ABS() { }

public void ESP() { } }

A fenti osztályok elemei grafikusan is megjelenthetők, ha a Visual Studio Professional változatában, a Solution Explorer ablakban, a Form.cs elemen használjuk a (jobb egérgomb lenyomásakor) felbukkanó menüt. A menüből a View Class Diagram menüpontot kell kiválasztanunk.

F2. A C# nyelvű programokban gyakran használt osztályok

A gyors alkalmazásfejlesztés alapja, hogy a programunkat előre elkészített osztályok/komponensek felhasználásával állíthatjuk elő. A C# nyelv nem rendelkezik saját osztálykönyvtárral, minden kész osztályt, amit a C# programokban használunk (Convert, Random, Form, Console stb.) a .NET Framework biztosítja számunkra.

F2.1 A .NET keretrendszer és a C# nyelv

A .NET Framework egy Windows-ba integrált keretrendszer, amely biztosít egy virtuális futtatórendszert (CLR – Common Language Runtime) és egy egységes osztálykönyvtárat (Class Libraries), több ezer kész osztállyal. A CLR gondoskodik a C# nyelven fejlesztett, és egy köztes kódra (IL - intermediate language) lefordított programok futtatásáról, ahogy ez jól látható az alább ábrán:

Ahhoz, hogy a :NET számára lefordítható legyen a C# programunk, a C# nyelvnek meg kell felelnie egy általános nyelvleírásnak, a Common Language Infrastructure-nak (CLI). A CLI-nek fontos részét képezi az általános típusleírás (Common Type Specification,CTS). Ez a típusleírás teljesen objektum-orientált, amelyben minden típusnak egy-egy osztály felel meg a System névtérben. A C# nyelv minden típusa mögött valójában egy CTS osztály áll, és a C# programunkban bármelyiket használhatjuk:

F2.2 Az egyszerű típusok

típusosztályainak alkalmazása

A C# nyelvű programokban az automatikus becsomagolás (boxing) és kicsomagolás (unboxing) mechanizmusa révén minden adat objektumként is viselkedhet. Ezért különösen hasznosak a felsorolt típusosztályok statikus és nem statikus tagjainak ismerete. Példaként tekintsük a System.Int32, System.Double és a System.Char osztályokat, hiszen a többi típusosztály hasonló tagokkal rendelkezik.

Először vegyük sorra a közös metódusokat! A nem statikus metódusokat a megfelelő típusú változókkal vagy konstans értékekkel használhatjuk:

int x = 12;

Int32 y = 23;

C# típusnév

CTS típusosztály

byte

System.Byte

sbyte

System.SByte

char

System.Char

bool

System.Boolean

short

System.Int16

int

System.Int32

long

System.Int64

ushort

System.UInt16

uint

System.UInt32

ulong

System.UInt64

float

System.Single

double

System.Double

decimal

System.Decimal

object

System.Object

string

System.String

75

```
String s = x.ToString();
s += y.ToString();
s += 729.ToString();
```

Metódus

Leírás

x.CompareTo(y)

Megadja x és y viszonyát. Ha x<y -1, ha x==y 0, ha x>y 1 a függvény értéke.

x.Equals(y)

true értékkel jelzi, a x egyenlő y-nal. Ellenkező esetben false értékkel tér vissza.

x.GetType()

System.Type típusú objektumban típusinformációkat ad vissza.

x.GetTypeCode()

A System.TypeCode felsorolás elemeivel azonosítható típuskóddal tér vissza.

x.ToString()

x.ToString(formátum))

x értékét sztringgé alakítva adja vissza. Az átalakítás formátum megadásával szabályozható.

A statikus metódusokat és adatmezőket a típusnév felhasználásával érjük el:

int érték;

bool sikeres;

sikeres = Int32.TryParse("1223", out érték);

sikeres = int.TryParse("729", out érték);

Adatmező

Leírás

típus.MaxValue

A típus értékkészletének legnagyobb eleme (nem használható bool típus esetén, helyette Boolean.TrueString).

típus.MinValue

A típus értékkészletének legkisebb eleme (nem használható bool típus esetén, helyette Boolean.FalseString).

Metódus

Leírás

típus.Parse(sztring)

A sztringben tárolt értéket megkísérli adott típusúvá alakítani. Sikeres esetben a függvény értéke tartalmazza az eredményt. Az átalakítás sikertelenségéről kivételek tudósítanak, mint például FormatException, ArgumentNullException stb.

típus.TryParse (sztring, out x)

A sztringben tárolt értéket megkísérli adott típusúvá alakítani. Sikeres esetben az eredmény az osztály típusával egyező típusú x változóba kerül, és a függvény értéke true lesz. Sikertelen esetben hamis értékkel tér vissza.

F2.2.1 Numerikus típusosztályok használata

A numerikus típusok esetén, a szövegből való konvertálás során akár különböző előírásokkal is vezérelhetjük az átalakítást. Ekkor a két elemző (parse) metódus átdefiniált változatait kell használnunk:

típus.Parse(sztring, NumberStyles_kifejezés)

típus.TryParse (sztring, NumberStyles_kifejezés, null, out x)

A második argumentumban a System.Globalization.NumberStyles felsorolás elemeiből bináris logikai műveletekkel képzett kifejezést kell átadnunk, mint például:

NumberStyles.AllowDecimalPoint | NumberStyles.AllowExponent

Megjegyezzük, hogy az átalakítások során túlcsordulás (OverflowException) is felléphet.

A valós típusok esetén a statikus mezők sora kibővül a végtelen és a nem értelmezhető szám (NAN) konstansokkal:

Mező

Leírás

típus.Epsilon

Az adott típusú legkisebb pozitív érték.

típus.NaN

Nem szám jelzése.

típus.NegativeInfinity

A negatív végtelen értéke.

típus.PositiveInfinity

A pozitív végtelen értéke

A változók különleges értékeit bool típusú, statikus metódusokkal ellenőrizhetjük: IsInfinity(), IsNegativeInfinity(), IsPositiveInfinity() és IsNaN().

F2.1.2 A Char osztály

76

A Char osztály bool típusú, statikus metódusaival különböző vizsgálatokat végezhetünk. A metódusok egy Char típusú karaktert, vagy egy sztringet és egy pozíciót várnak argumentumként:

Metódus

A vizsgálat tárgya

IsControl()

vezérlő karakter-e,

IsDigit()

decimális számjegy-e,

IsLetter()

betű-e,

IsLetterOrDigit()

betű vagy számjegy-e,

IsLower()

kisbetű-e,

IsNumber()

számjegy-e,

IsPunctuation()

írásjel-e,

IsSeparator()

elválasztó karakter-e,

IsSymbol()

szimbólum-e,

IsUpper()

nagybetű-e,

IsWhiteSpace()

tagoló karakter-e.

A karaktereket kisbetűvé és nagybetűvé alakíthatjuk a ToLower() és a ToUpper() statikus metódusokkal. A konverzió során területi (kultúrafüggő) beállításokat is figyelembe vehetünk a System.Globalization.CultureInfo osztály felhasz-nálásával:

```
Char c1 = char.ToLower('O');
char c2 = Char.ToUpper('ő', System.Globalization.CultureInfo.CurrentCulture);
```

Ha egy karakter számjegyet tartalmaz, azt double típusú számmá alakíthatjuk a GetNumericValue() metódus hívásával:

```
double x = char.GetNumericValue('7');
```


F2.3 Szövegek kezelése

C# nyelvben a szövegek tárolására és kezelésére string (System.String) és System.Text.StringBuilder típusú objektumokat használjuk. A String típusú változókban tárolt szöveg nem változtatható meg, ellentétben a StringBuilder típusú objektumokkal, melyekben lévő szöveg módosítható. További szövegkezelő osztályokat találunk a System.Text és a System.Text.RegularExpressions névterületeken.

F2.3.1 A String osztály és a string típus

A String osztály (illetve a neki megfelelő string típus) unicode kódolású szöveg tárolására használható, azonban a tárolt szöveg nem módosítható.

```
string s = "Programozzunk";
s = "A C# programozási nyelv";
```

A fenti példában első látásra úgy tűnik, hogy megváltoztattuk az s sztring tartalmát. Valójában azonban eldobtuk az s referencia által hivatkozott objektumot, és a második sor után egy új String típusú objektumra hivatkozik az s.

A tárolt szöveg karakterei sorszámozottak – az első karakter sorszáma 0, ha a karakterek száma N, akkor az utolsó karakter sorszáma N-1.

A String (string) típusú objektumot többféle módon is létrehozhatunk:

 Stringliterállal (szöveg konstanssal) inicializálva:

```
string fejléc1 = "A feladat megoldása";
string fejléc2 = @"A feladat megoldása";
```

 Karaktertömbből a teljes tömböt felhasználva, vagy a tömb adott indexű elemétől, adott darabszámú karakterének segítségével:

char[] ct = { 'C', 'S', 'h', 'a', 'r', 'p' };

String s1 = new String(ct); // CSharp

string s2 = new String(ct, 1, 3); // Sha

 Adott karakter adott számú ismétlésével:

string s3 = new String('C', 7); // CCCCCCC

A karaktersorozatok kezelését a C# operátorokkal is támogatja:

 Az értékadás (=) során új objektum jön létre a megadott, új tartalommal. A += művelet végrehajtását követően az új karaktersorozat objektum tartalma a régi szöveg összefűzve a jobb oldali operandussal:

String s;

s = "Programozzunk ";

s += "C#-ban!"; // Programozzunk C#-ban!

 A == és a != relációs műveletek nem a referenciákat, hanem a szövegtartalmakat hasonlítják össze string típusú változók esetén (más reláció nem használható):

String s1 = "alma", s2 = "a";

s2 += "lma";

bool v1 = s1 == s2; // true

bool v2 = s1 != s2; // false

bool v3 = (object)s1 == (object)s2; // false

 Az indexelés operátorával a sztring karaktereit egyenként is kiolvashatjuk: az első karakter sorszáma 0, míg az utolsóé Length-1: A Length tulajdonság megadja a karaktersorozatban tárolt karakterek számát.

String s = "BME";

string sr = string.Empty;

for (int i = 0; i < s.Length; i++)

sr += s[i].ToString() + ','; // B,M,E

F2.3.1.1 A String osztály statikus metódusai

A String osztály statikus metódusai egy vagy több sztring objektum kezelését segítik. Ezek közül is külön kell foglalkoznunk a karaktersorozatok összehasonlításának lehetőségeivel.

 Két sztringet a Compare() metódus változataival hasonlíthatunk össze – a visszatérési érték a két karaktersorozat viszonyát tükrözi:

-1, ha strA < strB,

0, ha strA == strB,

1, ha strA > strB.

int Compare(string strA, string strB)

int Compare(string strA, int indexA, string strB, int indexB, int length)

int Compare(string strA, string strB, bool ignoreCase)

int Compare(string strA, int indexA, string strB, int indexB, int length, bool ignoreCase)

int Compare(string strA, string strB, StringComparison comprisonType)

int Compare(string strA, int indexA, string strB, int indexB, int length, StringComparison comparisonType)

int Compare(string strA, string strB, bool ignoreCase, CultureInfo culture)

int Compare(string strA, int indexA, string strB, int indexB, int length, bool ignoreCase, CultureInfo culture)

Alaphelyzetben teljes sztringeket hasonlítanak össze a metódusok, azonban a kezdőpozíciók (indexA, indexB) és a hossz (length) megadásával részsztringek is összehasonlíthatók. Az alapértelmezés szerinti kis- és nagybetűk megkülönböztetését tilthatjuk le az ignoreCase paraméter true értéke való állításával. Az összehasonlítás menetét a System.StringComparison felsorolás elemeivel, illetve a System.Globalization.CultureInfo osztály objektumával is vezérelhetjük.

String.Compare("C#", "Java") // -1

String.Compare("C++", 0, "C#", 0, 1) // 0

String.Compare("AlmaFa", "almafa", true) // 0

String.Compare("Albert", "Ádám", StringComparison.CurrentCulture) // 1

String.Compare("Albert", "Ádám", false, new System.Globalization.CultureInfo("hu-HU")) // 1

 Két sztringet a CompareOrdinal() metódussal összevetve, az összehasonlítás a megfelelő helyen álló karakterek kódjainak kivonásával megy végbe. A visszatérési érték ekkor az utolsó különbség, amely:

negatív, ha strA < strB,

nulla, ha strA == strB,

pozítiv, ha strA > strB.

int CompareOrdinal(string strA, string strB)

int CompareOrdinal(string strA, int indexA, string strB, int indexB, int length)

String.Compare("ABC", "AM") // -1

String.CompareOrdinal("ABC", "AM") // -11

 Két sztringet az Equals() metódussal összehasonlítva megtudhatjuk, hogy tartalmuk egyezik-e (true), vagy sem (false).

bool Equals(string a, string b)

bool Equals(string a, string b, StringComparison comparisonType)

String.Equals("Almafa", "almafa") // false

String.Equals("Almafa", "almafa", StringComparison.OrdinalIgnoreCase) // true

 A Concat() metódussal két, három vagy több sztringet egyetlen sztringgé fűzhetünk össze.

string Concat(string str0, string str1)

string Concat(string str0, string str1, string str2)

string Concat(string str0, string str1, string str2, string str3)

string Concat(params string[] values)

string.Concat("A", "BC", "DEF") // ABCDEF

 A Join() metódus a value paraméterben megadott sztringtömb elemeit, az első paraméterben szereplő elválasztó szöveggel egyetlen sztringgé kapcsolja össze. (Szükség esetén a tömb egy része is kijelölhető startIndex, count.)

string Join(string separator, string[] value)

string Join(string separator, string[] value, int startIndex, int count)

string[] sv = { "12", "23", "7", "29" };

string s = String.Join(".,.", sv); // 12.,.23.,.7.,.29

 String típusú változót különböző típusú adatokból formátum (format) figyelembevételével is előállíthatunk a Format() metódus segítségével.

string Format(string format, Object arg0)

string Format(string format, params Object[] args)

string Format(string format, Object arg0, Object arg1)

string Format(string format, Object arg0, Object arg1, Object arg2)

string Format(IFormatProvider provider, string format, params Object[] args)

string s = String.Format("{0:X} {1:d}", 20041002, DateTime.Now); // 131CD2A 2014.10.23.

Az alábbi függvény a valós számok fixpontos formázását segíti:

public string ValósFormat(double szám, int mezőhossz, int tizedesek)

{

string formátum = "{0," + mezőhossz + ":F" + tizedesek + "}";

return String.Format(formátum, szám);

}

 Végezetül álljon itt két metódus, melyek közül a Copy() másolatot készít a megadott sztringről, míg a másikkal IsNullOrEmpty() megtudhatjuk, hogy a megadott String objektum létezik-e illetve üres-e.

string Copy (string str)

bool IsNullOrEmpty(string str)

F2.3.1.2 A String osztály nem statikus metódusai

A String osztály nem statikus metódusai az objektumban tárolt szöveggel végeznek műveleteket, azonban soha sem változtatják meg azt.

A String objektumok tartalmának tesztelése

A tesztelő (összehasonlító) metódusok int vagy bool visszatérési értékkel jelzik a művelet eredményét:

 A CompareTo() metódus az objektumban tárolt szöveget összeveti a paraméterben megadott sztringgel. A visszatérési érték – a Compare() metódushoz hasonlóan – -1, 0 vagy 1 értékkel jelzi a karaktersorozatok viszonyát. Szövegek azonosságának vizsgálatára az Equals() metódust ajánlott használni.

int CompareTo(string strB)

string név = "Anna";

int n = név.CompareTo("Adrienn"); // 1

 A Equals () metódus true értékkel jelzi, ha a két objektum ugyanazt a szöveget tárolja.

bool Equals(string strB)

bool Equals(string strB, StringComparison comparisonType)

string név = "Anna";

bool b = név.Equals("Adrienn"); // false

 A Contains() metódus true értékkel tér vissza, ha az objektumban tárolt szöveg tartalmazza az strB szövegét.

bool Contains(string strB)

string ma = DateTime.Now.ToLongDateString(); // 2014. október 23.

bool b = ma.Contains("október"); // true

 A StartsWith() és az EndsWith() metódusokkal egyszerűen megvizsgálhatjuk, hogy a tárolt szöveg a megadott szöveggel kezdődik-, illetve végződik-e. A két metódust azonos módon kell hívni, ezért csak az első paraméterezését adjuk meg. (A további paraméterek értelmezése megegyezik a Compare() metódus paramétereinél elmondottakkal.)

bool StartsWith(string strB)

bool StartsWith(string strB, StringComparison comparisonType)

bool StartsWith(string strB, bool ignoreCase, CultureInfo culture)

string html = "<H1>C# programozás</H1>";

bool kezd = html.StartsWith("<H1>"); // true

bool veg = html.EndsWith("<H2>"); // false

 Az IndexOf() metódus a megadott karakter vagy sztring (value) első előfordulásának pozíciójával, míg a LastIndexOf() metódus az utolsó előfordulás helyével tér vissza. A két metódus paraméterezése megegyezik, ezért csak az IndexOf() metódusét közöljük. Alaphelyzetben a metódusok a teljes szövegben keresnek, azonban szükség esetén kijelölhető a kezdőpozíció (startIndex), illetve a keresés tartománya (count). A LastIndexOf() metódus a kezdőpozíciótól visszafelé keres. Sikertelen esetben a függvények értéke -1.

int IndexOf(char value)

int IndexOf(string value)

int IndexOf(string value, StringComparison comparisonType)

int IndexOf(char value, int startIndex)

int IndexOf(string value, int startIndex)

int IndexOf(string value, int startIndex, StringComparison comparisonType)

int IndexOf(char value, int startIndex, int count)

int IndexOf(string value, int startIndex, int count)

int IndexOf(string value, int startIndex, int count, StringComparison comparisonType)

```
string ns = "www.bme.hu";
int p1 = ns.IndexOf('b'); // 4
int p2 = ns.IndexOf(".hu"); // 7
int p3 = ns.LastIndexOf('w', 5, 4); // 2
int p4 = ns.IndexOf('w', 5, 4); // -1
```

- A következő két metódus (IndexOfAny(), LastIndexOfAny()) szintén karakterpozíciót keres az objektumban tárolt szövegben, azonban egyetlen karakter helyett az anyOf tömbben megadott karakterhalmaz elemeinek bármelyikét megtalálhatják. A LastIndexOfAny() metódus a kiindulási pozíciótól visszafelé haladva keres. (A fenti, hasonló nevű metódusokhoz hasonlóan, szintén csak az egyik metódus paraméterezését adjuk meg.)

```
int IndexOfAny(char[] anyOf)
int IndexOfAny(char[] anyOf, int startIndex)
int IndexOfAny(char[] anyOf, int startIndex, int count)
string mondat = "Why do Java Programmers wear glasses? Because, they don't C#.";
char[] irasjel = { ',', '.', '!', '?' };
int p1 = mondat.IndexOfAny(irasjel); // 36
int p2 = mondat.LastIndexOfAny(irasjel); // 60
int p3 = mondat.IndexOfAny(irasjel, 0, 12); // -1
```

A string objektumok tartalmának feldolgozása

A feldolgozó metódusok működésének eredményeként egy új string, karakter- vagy sztringtömb keletkezik.

- Az Insert() metódus az objektumban tárolt szövegbe adott pozíciótól (startIndex) beszúrja a megadott szöveget (value), és az eredményt egy új string objektumban adja vissza.

```
string Insert(int startIndex, string value)
string s2, s1 = "Cogito sum.";
s2 = s1.Insert(6, ", ergo"); // Cogito, ergo sum.
```


- A Remove() metódus az objektumban tárolt szöveg karaktereit törli az adott pozíciótól (startIndex) a szöveg végéig, illetve az adott pozíciótól count hosszon.

```
string Remove(int startIndex)
string Remove(int startIndex, int count)
String s2, s1 = "Veni, vidi, vici!";
s2 = s1.Remove(5, 6).Remove(10); // Veni, vici
```

- A Replace() metódus a tárolt szöveg minden oldChar karakterét, illetve OldValue szövegét a megadott karakterre, illetve sztringre cseréli.

```
string Replace(char oldChar, char newChar)
string Replace(string oldValue, string newValue)
String s2, s1 = "abrakadabra!";
```

s2 = s1.Replace('a', 'e').Replace("re", "ber"); // ebberkedebber!

 A Substring() metódus a tárolt szöveg startindex pozíción kezdődő részével tér vissza. Alaphelyzetben a szöveg végéig, a length paraméter megadása esetén pedig a megadott hosszig hozza létre az új sztringet.

string Substring(int startIndex)

string Substring(int startIndex, int length)

String s2, s1 = "Párizs megér egy misét!";

s2 = s1.Substring(13); // egy misét!

s2 = s1.Substring(0, 12); // Párizs megér

 A ToLower() és a ToUpper() metódusok a tárolt szöveget kis-, illetve nagybetűssé alakítva hozzák létre az új sztringet.

string ToLower()

string ToLower(CultureInfo culture)

string ToUpper()

string ToUpper(CultureInfo culture)

String s2, s1 = "Eppur si move!";

s2 = s1.ToLower(); // eppur si move!

s2 = s1.ToUpper(); // EPPUR SI MOVE!

A fenti metódusok kultúra független változatai a ToLowerInvariant() és a ToUpperInvariant() függvények:

string ToLowerInvariant()

string ToUpperInvariant()

 A PadLeft() és a PadRight() metódusokkal adott hosszú (totalWidth) sztringet hozunk létre, balról, illetve jobbról kitöltve szóközökkel vagy a megadott paddingChar karakterrel. Amennyiben a totalWidth kisebb vagy egyenlő, mint a sztring hossza, másolatot kapunk a karaktersorozatról.

string PadLeft(int totalWidth)

string PadLeft(int totalWidth, char paddingChar)

string PadRight(int totalWidth)

string PadRight(int totalWidth, char paddingChar)

String s2, s1 = "2.789";

s2 = s1.PadLeft(7); // 2.789

s2 = s1.PadRight(7, '0'); // 2.78900

 A Trim(), TrimStart() és TrimEnd() metódusok segítségével a szöveg minkét végéről, az elejéről, illetve a végéről eltávolíthatjuk a paramétertömbben megadott karaktereket. A paraméter nélküli Trim() hívás, illetve a null argumentumként való átadása esetén a metódusok a tagoló karaktereket (white-spaces) törlik a megfelelő helyekről.

string Trim()

string Trim(params char[] trimChars)

```
string TrimEnd(params char[] trimChars)
string TrimStart(params char[] trimChars)
String s2, s1 = "<<>>1, 2, 3, 4<<>>";
char[] vc = { '_', '<', '>' };
s2 = s1.Trim('<', '>'); // 1, 2, 3, 4
s2 = s1.TrimStart(vc); // 1, 2, 3, 4<<>>
s2 = s1.TrimEnd('>', '<'); // <<>>1, 2, 3, 4
```

 A ToCharArray() metódus az objektumban tárolt szöveg startIndex pozícióján kezdődő length karakterét egy új karaktertömbbe másolja, és visszatér a tömb hivatkozásával. A paraméter nélküli változat a teljes sztringet másolja.

```
char[] ToCharArray()
char[] ToCharArray(int startIndex, int length)
String s2, s1 = "<<>>1, 2, 3, 4<<>>";
string vc = "_<>";
s2 = s1.Trim(vc.ToCharArray()); // 1, 2, 3, 4
```

 A CopyTo() metódus a tárolt szöveg sourceIndex pozícióján kezdődő count karakterét a megadott destination karaktertömbbe másolja a destinationIndex indextől kezdve.

```
void CopyTo(int sourceIndex, char[] destination, int destinationIndex, int count)
```

 A Split() metódus az objektumban tárolt szöveget a megadott tagoló (karakterek vagy sztringek) mentén szétdarabolja, és a részsztringeket tömbben adja vissza. A count paraméterrel a visszaadott szövegek számát korlátozhatjuk, míg az options paraméter segítségével a keletkező üres sztringeket eldobhatjuk (StringSplit-Options.RemoveEmptyEntries).

```
string[] Split(params char[] separator)
string[] Split(char[] separator, int count)
string[] Split(char[] separator, StringSplitOptions options)
string[] Split(string[] separator, StringSplitOptions options)
string[] Split(char[] separator, int count, StringSplitOptions options)
string[] Split(string[] separator, int count, StringSplitOptions options)
String s1 = "2014.10.23";
string[] st = s1.Split('.');
int év, hó, nap;
if (st.Length == 3)
{
év = Convert.ToInt32(st[0]);
hó = Convert.ToInt32(st[1]);
nap = Convert.ToInt32(st[2]);
}
```

F2.3.2 A StringBuilder osztály

Mint már korábban említettük, a System.String osztály objektumaiban tárolt szöveg nem változtatható meg, ezért minden módosítás során új objektum jön létre a memóriában. Ezzel szemben a System.Text.StringBuilder osztály objektumainak szövege módosítható: kitörölhetünk belőle, beszúrhatunk szövegrészletet, és a szöveg hossza is változhat.

Ezt az teszi lehetővé, hogy a StringBuilder objektumaihoz rendelt szövegpuffer dinamikus. A puffer aktuális méretét a Capacity tulajdonság jelzi, míg az éppen tárolt szöveg hosszát (karaktereinek számát) a Length tulajdonság tartalmazza. Amíg a műveletek során keletkező eredményszöveg elfér a pufferben (Lenght≤Capacity), nem történik memóriafoglalás. Ha azonban a szöveg hosszabb a puffernél, új puffer jön létre, és a Capacity felveszi a Length tulajdonság értékét. Mindkét tulajdonság értéke egymástól függetlenül beállítható, azonban a Capacity<Lenght esetben ArgumentOutOfRangeException kivétel jön létre. A Length tulajdonság állításával a tárolt karaktersorozatot csonkolhatjuk, illetve 0 értékű bájtokkal kibővíthetjük.

A fentieken kívül használható még a MaxCapacity tulajdonság, amellyel a sztringpuffer maximális hosszát állíthatjuk be. A maximális méret túllépése esetén szintén a fenti kivétel jön létre.(Alaphelyzetben a MaxCapacity értéke 2G – 2147483647.)

A StringBuilder típusú objektum indexelőjének (obj[index])segítségével a sztring karaktereit egyenként lekérdezhet-jük, illetve módosíthatjuk.

A StringBuilder típusú objektumokat különböző konstruktorhívásokkal hozhatunk létre:

 StringBuilder() – üres, 16 karakter hosszú (Capacity) sztringpuffert keletkezik,

StringBuilder ures1 = new StringBuilder(); // Capacity 16, Lenght 0

 StringBuilder(int méret) – adott hosszúságú puffert hoz létre:

StringBuilder ures2 = new StringBuilder(30); // Capacity 30, Lenght 0

 StringBuilder(String str) – az argumentumban megadott sztringet tartalmazó puffer jön létre:

StringBuilder szoveg = new StringBuilder("C# programozás"); // Capacity 16, Lenght 14

 StringBuilder(String str, int méret) – az argumentumban megadott sztringet tartalmazó, adott méretű puffer keletkezik:

StringBuilder szoveg = new StringBuilder("C# programozás", 28); // Capacity 28, Lenght 14

 StringBuilder(String str, int, poz int db, int méret) – az str sztring kijelölt részét tartalmazó, adott méretű puffer keletkezik:

StringBuilder szoveg = new StringBuilder("C# programozás", 0, 2, 28); // Capacity 28, Lenght

// a tárolt szöveg: C#

A következőkben összefoglaljuk a StringBuilder objektumokban tárolt sztringek módosítására használható metódusokat: A metódusok, a void típusú CopyTo() kivételével, az aktuális objektumreferenciájával térnek vissza.

 Append(típus adat) – az átdefiniált változatokkal bármilyen előre definiált típusú adatot (a karaktertömböt is beleértve) szövegként a sztringhez fűzhetünk.

StringBuilder sb = new StringBuilder();

sb.Append("Iván ").Append(10).Append(true); // Iván 10True

```
sb.Length = 0;
sb.Append(new char[] { 'C', '#', '!' }); // C#!
```

Lehetőség van adott karakter ismétlésével előállított karaktersorozat, illetve részsztring hozzáadására is:

```
sb.Length = 0;
sb.Append('O', 3); // OOO
sb.Length = 0;
sb.Append(new char[] { 'A', 'B', 'C', 'D', 'E' }, 1, 3); // BCD
sb.Length = 0;
sb.Append("Java & C# prog", 7, 2); // C#
```


 AppendLine(string szöveg) – az opcionálisan megadott szöveg után sorvége jelet is fűz az objektumhoz:

```
StringBuilder sb = new StringBuilder();
sb.AppendLine("Első sor");
sb.AppendLine("Második sor");
MessageBox.Show(sb.ToString());
```

 AppendFormat(string formátum, adatlista) – a felsorolt adato(k) a formátum szerint sztringgé alakítva kapcsolódnak az objektumhoz:

```
StringBuilder sb = new StringBuilder();
sb.AppendFormat("{1} {0} {2}", 10, "Iván", true); // Iván 10 True
```

 CopyTo(int poz, char[] céltömb, int célindex, int darab) – a tárolt sztring darab karakterét a megadott karaktertömbbe másolja, adott indextől kezdve.

 Insert(int poz, típus adat) – az Append() metódus paraméterezésének megfelelő módon megadott adat a megadott pozíciótól beszúródik a tárolt sztringbe:

```
StringBuilder sb = new StringBuilder();
sb.Append("ABCD"); // ABCD
sb.Insert(3, 729).Insert(5, "X", 3); // ABC72XXX9D
```

 Remove(int poz, int hossz) – adott pozíciótól, adott számú karaktert töröl a karaktersorozatból:

```
StringBuilder sb = new StringBuilder(".NET FrameWork");
sb.Remove(0, 1).Remove(3, sb.Length - 7); // NETWork
```

 Replace(mit, mire) – a mit minden előfordulását mire cseréli a sztringben:

```
StringBuilder sb1 = new StringBuilder("A1A2A3A4A5");
sb1.Replace('A', 'a'); // a1a2a3a4a5
StringBuilder sb2 = new StringBuilder("az málna, az barack");
sb2.Replace("az", "a"); // a málna, a barack
```

Minkét esetben kijelölhetjük a csere kezdőpozícióját és hosszát:

StringBuilder sb3 = new StringBuilder("A1A2A3A4A5");

sb3.Replace('A', 'a', 3, 4); // A1A2a3a4A5

StringBuilder sb4 = new StringBuilder("az málna, az barack");

sb4.Replace("az", "a", 10, 3); // az málna, a barack

F3. A Windows Forms alkalmazások alapelemei

Napjainkban a C# programokból már több, a .NET keretrendszer által biztosított, grafikus felületet megvalósító megoldás közül is választhatunk:

 A Windows Forms (WinForms) az elsőként kifejlesztett felület (.NET 1.0), amely raszteres grafikára (GDI+) épül, és teljes mértékben a számítógép processzora által vezérelt.

 A Windows Presentation Foundation (WPF) a később kifejlesztett felület (.NET 3.0), amely vektoros grafikára épül, és célja a 3D grafikus kártyák lehetőségeinek minél jobb kihasználása.

 A Modern UI hordozható eszközökre szánt, egyszerűsített, grafikus felület, amely a WPF architektúrán alapult.

A fenti megoldások közül ebben a részben a Windows Forms alapismeretek gyűjtöttük csokorba. A Microsoft Visual Studio egy felülettervező eszközt biztosít számunkra, amellyel grafikusan készíthetjük el a programunk felhasználói felületét (GUI). A felülethez tartozó kód automatikusan létrejön, így nem szükséges, hogy mi írjuk meg a teljes programot.

85

using System; using System.Collections.Generic; using System.ComponentModel; using System.Data; using System.Drawing; using System.Linq; using System.Text; using System.Windows.Forms; namespace WindowsFormsApplication { public partial class Form1 : Form { public Form1() { InitializeComponent(); } } }

A felülettervező eszköz jellemzői:

 Egy eszköztárból (ToolBox) választhatunk vezérlőket, melyeket aztán grafikusan (a „húzd és dobd” módszerrel) helyezhetjük a formra.

 A vezérlők tulajdonságait (Properties) és eseményeit (Events) külön ablakban állíthatjuk be.

 Bármikor válthatunk a kód és a tervező nézet között, így felváltva szerkeszthetjük a felületet és a program kódját.

 A grafikus felület kialakításához szükséges osztályokat a System.Windows.Forms névtér tartalmazza.

F3.1 A Form osztály gyakran használt elemei

A saját ablakunk ősosztálya a Form, amely példányosítással vagy specializációval (származtatással) használható. A megjelenő ablak pozícióját és méreteit a képernyő koordináta-rendszerében kell értelmeznünk:

xyHeightTopWidthLeft

86

A Form fontosabb tulajdonságait táblázatban foglaltuk össze:

Tulajdonságnév Leírás

AcceptButton; CancelButton

A formra elhelyezett gombok lehetséges alapértelmezett jelentése.

BackColor

Az ablak hátterének a színe. A beállításhoz a System.Drawing.Color struktúra tulajdon-ságait használjuk:

```
private void button1_Click(object sender, EventArgs e)
{
this.BackColor = Color.Azure;
this.ForeColor = Color.Red;
}
```

BackgroundImage

Beállított háttérkép.

BackgroundImageLayout

A beállított háttérkép elrendezése.

Bounds

Az ablak pozícióját és méreteit tartalmazó téglalap (Rectangle) lekérdezése, beállítása.

A tulajdonság használata programból:

```
private void button1_Click(object sender, EventArgs e)
{
this.Bounds = new Rectangle(100, 100, 300, 200);
this.Bounds = new Rectangle(new Point(100, 100), new Size(300, 200));
int xp = Bounds.Left;
int yp = Top;
int w = Width;
int h = Bounds.Height;
}
```

ClientRectangle

Az ablak (fejléc és keretek nélküli) munkaterületének adatai.

ClientSize

Az ablak (fejléc és keretek nélküli) munkaterületének méretei.

ControlBox, MinimizeBox,

MaximizeBox

A vezérlőgombok megjelennek-e az ablak fejlécében?

Cursor

Az egérmutató formája az ablak felett.

DialogResult

A párbeszédablak működésének eredménye.

Enabled

Engedélyezett-e?

Font

A form és a formon lévő vezérlők alapértelmezett betűtípusa

ForeColor

A formon lévő vezérlők alapértelmezett szövegszíne.

FormBorderStyle

Az ablakkeretének stílusa: méretezhető, rögzített stb.

KeyPreview

Elfogjuk-e a billentyű-eseményeket, mielőtt azok a fókuszban levő vezérlőhöz kerülnek?

Left, Top

Az ablak bal felső sarkának távolságai a képernyő bal felső sarkától.

Location

Az ablak bal felső sarkának távolsága a képernyő bal felső sarkától x;y formában. A tulajdonság használata C# programból:

```
private void button1_Click(object sender, EventArgs e)
{
this.Location = new Point(120, 230);
int xPoz = Location.X;
int yPoz = Location.Y;
}
```

MainMenuStrip

Hivatkozás az ablak menüjére (MenuStrip).

Opacity

Az ablak átlátszóságának mértéke (100% nem átlátszó).

Right, Bottom

Az ablak jobb alsó sarkának koordinátái pixelben (csak olvasható tulajdonságok).

Size

Az ablak méretei pixelben szélesség, magasság formában. A tulajdonság használata programból:

```
private void button1_Click(object sender, EventArgs e) { this.Size = new Size(300, 200); int w = this.Size.Width; int h = this.Size.Height; }
```

StartPosition

Megadja, hogy hol helyezkedjen el az ablak megjelenéskor (pl. a képernyő közepén).

Text

Az ablak fejlécében megjelenő szöveg.

Visible

Látható-e az ablak?.

WindowState

Az ablak állapota (normál, maximális méretű stb.)

Width, Height

Az ablak szélessége és magassága pixelben.

(A szürkével szedett tulajdonságokat csak programból állíthatjuk.)

A példákban szereplő Point, Size, és Rectangle struktúrák deklarációját a System.Drawing névtérben találjuk. Ezek segítségével az ablak pozícióját és méreteit egyszerűbben is megadhatjuk, mint a Left, Top, Width és Height tulajdonságokkal.

Metódusok segítségével egyszerűen vezérelhetjük a form működését a C# programunkból. Nézzünk néhány jól használható metódust!

Metódus Leírás

CenterToScreen()

Az ablak pozicionálása a képernyő közepére.

Close()

Az ablak bezárása.

Hide()

Az ablak eltüntetése, mint a Visible = false.

Invalidate()

Érvényteleníti az ablak tartalmát, így az újra fog rajzolódni (a Paint esemény hívásával).

Refresh()

Az ablak tartalmának frissítése.

SetBounds(left, top, width, height)

Az ablak pozíciójának és méreteinek beállítása.

Show()

Az ablak megjelenítése, mint a Visible = true.

ShowDialog()

Az ablak megjelenítése (modális) párbeszédablakként.

Update()

A ablak tartalmának frissítése.

A form eseményei közül csak néhányat emelünk ki.

Esemény Leírás

Activated és Deactivated

Az ablak aktívvá, illetve inaktívvá válását jelző események.

Click

Egérkattintás valahol a form szabad területén.

FormClosing

Az ablak bezárásnak kérésekor hívódik meg. Az e paraméter Cancel tulajdonságának true értéket adva, megakadályozhatjuk az ablak bezárását:

private void Form1_FormClosing(object sender, FormClosingEventArgs e) { if (MessageBox.Show("Valóban ki akar lépni?", "Figyelem", MessageBoxButtons.YesNo) == DialogResult.No) { e.Cancel = true; // megakadályozzuk az ablak bezárását } }

KeyPress

Aktív form esetén a billentyűk lenyomásakor keletkező (karakter) esemény.

A KeyPreview = true; beállítást követően a form összes szöveges vezérlője csak decimális jegyeket fogad el:

private void Form1_KeyPress(object sender, KeyPressEventArgs e) { if (e.KeyChar > '9' || e.KeyChar < '0') { e.KeyChar = (char)0; } }

Load

A form első betöltődésekor hívódik meg, azonban még annak megjelenése előtt. Gyakran használjuk a formon elhelyezett vezérlők inicializálására. Ez a form alapértelmezés szerint eseménye, melynek kezelőjét a formon való kétszeri kattintással is elkészíthetjük.

MouseClick

Az egérkattintás eseménye, lekérdezhető egérpozícióval.

MouseDown, MouseUp

Az egérgomb lenyomásának illetve felengedésének eseménye. Az események egymás utáni sorrendje egyetlen egérkattintáskor: MouseDown □ Click □ MouseClick □ MouseUp.

MouseMove

Az egérmutató mozgatásának eseménye.

Az alábbi eseménykezelőben a jobb egérgombot lenyomva tartva, és az egeret mozogatva, kijelezzük az egérpozíciót a form fejlécében.

private void Form1_MouseMove(object sender, MouseEventArgs e) { if (e.Button == MouseButtons.Right) { this.Text = "Egérpozíció: (" + e.X+","+e.Y+")"; } }

Paint

A form újrarajzolásakor meghívódó esemény.

Resize

Az ablak méretének változtatását jelző esemény.

Az alábbi példában biztosítjuk, hogy az ablak négyzet alakú maradjon az átméretezés során:

private void Form1_Resize(object sender, EventArgs e) { if (this.Size.Height != this.Size.Width) { this.Size = new Size(this.Size.Width, this.Size.Width); } }

Shown

Az ablak első megjelenítésekor fellépő esemény.

F3.2 Gyakran használt vizuális vezérlőelemek

A Visual Studio fejlesztői környezet kényelmessé és hatékonnyá teszi a Windows Forms alkalmazások felhasználói felületének kialakítását. Az eszköztárból (ToolBox) több

mint száz vezérlőelem közül választhatunk. Ezek közül a következőkben csak néhány bemutatására vállalkozunk.

F.3.2.1 A vezérlők közös tulajdonságai, metódusai és eseményei

A konkrét vezérlőelemek bemutatása előtt áttekintjük azokat a tulajdonságokat, metódusokat és eseményeket, amelyek a tárgyalt vezérlők többségénél azonosak.

A vezérlők neve

A vezérlők nevét a (string típusú) Name tulajdonság tárolja. Mivel a C# programban a vezérlő objektumra hivatkozó változó neve egyúttal a vezérlő neve is, ezért a változónevekre érvényes megkötések vonatkoznak a Name tulajdonság értékéres. A Name tulajdonságot célszerű a tervezés első lépéseként beállítani, mivel a beállított név az esemény-kezelő metódusok nevében is megjelenik.

Amikor egy vezérlőt a formra helyezünk, akkor a Visual Studio rendszer automatikusan elnevezi a létrejövő objektumot. A névben általában a komponens kisbetűvel kezdődő neve szerepel egy sorszámmal kiegészítve (label1, button1, textBox1, checkBox1 stb.). Ha ismét a formra teszünk egy, az előzővel azonos típusú vezérlőt, akkor azonosítóban eggyel nagyobb sorszám jelenik meg (button2, textBox2 stb).

A vezérlők elhelyezkedése

A vizuális vezérlőelemek elhelyezkedését és méretét a befoglaló téglalapjuk bal felső sarkának koordinátái (Left, Top, Left), szélessége (Width) és magassága (Height) határozzák meg. A formra helyezett vezérlők pozíciója, a form munkaterületének bal felső sarkához képest relatív. A pixelben kifejezett koordináták értelmezését az alábbi ábra segíti:

xyLeftTopWidthHeight

A formhoz hasonlóan a Point típusú Location tulajdonsággal egy lépésben áthelyezhetjük a vezérlőt, a Size típusú Size tulajdonsággal átméretezhetjük azt, míg a Rectangle típusú Bounds tulajdonsággal mindkét műveletet egyszerre is elvégezhetjük. Ez utóbbi műveletet a SetBound() metódus is támogatja. A csak olvasható Right és Bottom tulajdonságok a vezérlőt befoglaló téglalap jobb alsó sarkának koordinátáit tartalmazzák.

private void button1_Click(object sender, EventArgs e) { button1.Location = new Point(50, 50); button1.Size = new Size(100, 40); button1.Bounds = new Rectangle(50, 50, 100, 40); MessageBox.Show("Bal alsó sarok koordinátái:" + button1.Right + "," + button1.Bottom); button1.SetBounds(10, 10, 50, 30); }

Az AutoSize logikai tulajdonság true értékre állítása után a vezérlő figyelmen kívül hagyja a megadott méretadatokat, és a mérete akkora lesz, hogy elférjen benne a tárolt szöveg és/vagy kép. (Képet általában az Image tulajdonság segítségével tölthetünk a vezérlőkbe.)

Az (AnchorStyles típusú) Anchor tulajdonság segítségével a vezérlő pozícióját és méreteit az ablak széleihez köthetjük, így az ablak átméretezésével a vezérlő elhelyezkedése és mérete is megváltozhat. A tulajdonság használata a Visual Studio rendszerből és programból:

button1.Anchor = (AnchorStyles.Top | AnchorStyles.Left);

A vezérlők megjelenése

A vizuális vezérlőelemek a program futása során lehetnek láthatók – ez az alapértelmezés –, és lehetnek rejtettek. Ezen két állapot egyike bármikor beállítható a bool típusú Visible tulajdonság segítségével: true esetén a vezérlőelem látható, false esetén pedig nem. Ugyanezt a működést a vezérlő Show(), illetve Hide() metódusának hívásával is elérhetjük.

Az ugyancsak bool típusú Enabled tulajdonság true értéke engedélyezi, a false értéke pedig tiltja a vezérlő működtetését. A letiltott vezérlő látható ugyan, azonban színe szürke, és nem fogadja a felhasználói beavatkozásokat.

A vezérlők többségének háttérszíne megadható a Color típusú BackColor tulajdonsággal. Ha a vezérlőn szöveg is megjelenik, annak színét a ForeColor tulajdonsággal állíthatjuk be. A színek megadásához a System.Drawing névtér Color és SystemColors osztályainak statikus tulajdonságait használhatjuk:

button1.BackColor = Color.Yellow; button1.ForeColor = SystemColors.ControlDark;

A formhoz hasonlóan a vezérlőknek is van az egérmutató megjelenítését szabályozó Cursor (Cursor típusú) tulajdonsága. Ez akkor érvényesül, amikor az egérmutatót a vezérlő felett mozgatjuk. Az egérmutató programból történő beállítását a Cursors osztály Cursor típusú statikus tulajdonságai segítik:

button1.Cursor = Cursors.IBeam;

A szöveget tartalmazó vezérlők esetén a megjelenést a tárolt szöveg és annak betűtípusa is befolyásolja. Az esetek nagy részében a szöveget a string típusú Text tulajdonságban érjük el, míg a betűtípusról a Font tulajdonság beállításával gondoskodhatunk.

A vezérlők használata billentyűzetről

A vezérlők egy részének működtetéséhez a billentyűzetet is használhatjuk, például szöveg bevitele, választógomb bejelölése. Kérdés azonban, hogy a formon elhelyezett több vezérlő közül melyik fogadja a billentyűzet-eseményeket. Minden ablakban legfeljebb egy kijelölt vezérlőelem lehet, amely megkapja a billentyűzet-eseményeket. Ilyenkor azt mondjuk, hogy ez a vezérlő birtokolja az inputfókuszt. (Az inputfókuszt nem fogadják a Label, a Panel, a GroupBox, a PictureBox stb.)

A vezérlők többsége képes az inputfókusz fogadására, amit egérrel vagy a <Tab> billentyű megnyomásával helyezhetünk rá. Az inputfókusz vezérlőre való helyezéséhez a (bool típusú) TabStop tulajdonságnak true értéket kell adnunk. A tabulátor gombbal való körbejárás sorrendjét a (int típusú) TabIndex tulajdonság határozza meg – a 0 indexű elem kerül először a fókuszba. A logikai típusú Focused tulajdonság igaz értékkel jelzi, ha a vezérlő a fókuszban van. Az inputfókuszt programból a Focus() metódus hívásával helyezhetjük valamelyik vezérlőre.

Az alábbi példában először töröljük a szövegmező tartalmát, majd pedig ráhelyezzük az inputfókuszt:

private void button1_Click_1(object sender, EventArgs e) { textBox1.Clear(); textBox1.Focus(); }

Az inputfókusz megkapásáról az Enter vagy a GotFocus események értesítenek, míg elvesztését a Leave vagy a LostFocus események jelzik. Az inputfókusszal rendelkező vezérlők további, alacsonyabb szintű billentyűzet-események (KeyPress, KeyDown, KeyUp) fogadásra is képesek.

A vezérlők és az egér

Napjainkban a Windows alatt futó programok vezérlésének legfőbb eszköze az egér, így a vezérlőket felkészítették az egértől érkező események minél pontosabb fogadására. A form esetén ismertetett egéreseményeken (MouseDown, MouseUp, MouseMove) túlmenően a MouseEnter esemény jelzi, ha az egérmutató belép a vezérlő területére, és a MouseLeave, ha távozik onnan.

A vezérlők többsége érzékeli, ha az egérrel kattintunk a területe felett. Ezt két esemény közül választva is feldolgozhatjuk. A Click esemény akkor is jelez, ha az egér mellett egy billentyűvel kattintunk (<Szóköz>, <Enter>). Ezzel szemben a MouseClick esemény bekövetkeztekor biztosak lehetünk abban, hogy az egérrel kattintott a felhasználó.

F.3.2.2 A vezérlőcsoportok kialakítása

Egyszerűbb felhasználó felületek (UI) elkészítésekor a vezérlőket a formon szoktuk elhelyezni. Nagyobb mennyiségű elem esetén többféle tároló vezérlő közül is választhatunk, amelyek alkalmazásával vezérlőcsoportokat alakíthatunk ki. A csoportosító elemek közül leggyakrabbam a Panel és a GroupBox tárolókat használjuk. A tárolók használatának nagy előnye, hogy a rájuk helyezett vezérlőket a tárolóval együtt tudjuk mozgatni a formon.

A felhasználói felület kialakításakor mindig az aktív tároló vagy a form lesz a vezérlő szülő objektuma (Parent). Egy adott vezérlő pozíciója mindig a szülő tároló területéhez (bal felső sarkához) képest relatív.

Felhívjuk a figyelmet arra, hogy a csoportosító vezérlők használatának ellenére minden vezérlő eseménykezelő metódusa a form osztályán belül jön létre, így nem helyezhetünk el azonos nevű vezérlőket a különböző csoportokban. A vezérlők tulajdonságaira és metódusaira szintén a megszokott módon hivatkozhatunk.

F.3.2.3 Vezérlőgombok

A vezérlőgombok lehetőséget nyújtanak a program futásának szabályozására. A gombok működésük alapján három csoportra oszthatók. A nyomógomb (Button) olyan, mint a kapucsengő, megnyomáskor jelet ad, és elengedéskor alapállapotba kerül. A jelölőnégyzetek (CheckBox) - a gombokhoz hasonlóan - jelzést adnak, azonban képesek megtartani az állapotukat, mint a villanykapcsoló. Választógombokat (RadioButton) is használhatunk, amelyek olyanok, mint a rádió csatornaváltó gombjai. A választógombok jelet adnak, megtartják állapotukat, azonban egyszerre csak egy lehet bekapcsolt állapotban a csoportba sorolt választógombok közül.

A Button (nyomógomb)

A Button típusú vezérlők a nyomógombok működését szimulálják. A nyomógomb megnyomásakor végrehajtandó tevékenységet a kattintás eseményt (Click vagy MouseClick) feldolgozó eseménykezelő metódusban kell elhelyeznünk. A gombnyomás eseményt a PerformClick() metódus hívásával is kiválthatjuk.

A Text tulajdonság a nyomógomb feliratát tartalmazza, amely első megjelenéskor megegyezik a nyomógomb nevével (Name). A feliratban az & (ampersand) mögötti karakter aláhúzva jelenik meg. A felhasználó a billentyűzetről is kiválthatja a gombnyomás eseményt, ha az aláhúzott karaktert tartalmazó billentyűt az <Alt> billentyűvel együtt nyomja meg. (Ha az azonosítóban && írunk, akkor az & jel jelenik meg.)

A nyomógombokat leggyakrabban párbeszédablakokban használjuk, ezért alapvető szerepe van a (DialogResult felsorolás típusú) DialogResult tulajdonságnak. Gombnyomáskor a gomb DialogResult jellemzőjének értéke (Abort, Cancel, OK stb.) bemásolódik a

ShowDialog() hívással megjelenített szülőablak DialogResult tulajdonságába, és bezárul a szülőablak.

A CheckBox (jelölőnégyzet)

A jelölőnégyzet olyan gomb, mely megtartja beállított állapotát. A jelölőnégyzet két- vagy háromállapotú is lehet. Ha a ThreeState logikai típusú tulajdonság igaz értékű, akkor három, egyébként pedig kétállapotú. A jelölőnégyzet alakja kikapcsolt állapotban egy üres négyzet, míg a bekapcsolt állapotát a (pipa) jelzi. A háromállapotú jelölőnégyzet harmadik állapotában egy kitöltött négyzet jelenik meg.

A jelölőnégyzet állapotának tárolására a (CheckState felsorolás típusú) CheckState tulajdonság szolgál:

CheckState

ThreeState

Unchecked

Checked

Indeterminate

true

false

A Checked tulajdonság igaz értékkel jelzi, ha a jelölőnégyzet bejelölt állapotban van (Checked vagy Indeterminate), ellenkező esetben értéke false. Mindkét állapotjelző tulajdonság csak akkor változik meg a gombon való kattintáskor, ha az AutoCheck tulajdonság értéke true (ami alapérték).

A jelölőnégyzetek további tulajdonsága a (System.Drawing.ContentAlignment felsorolás típusú) CheckAlign, amely a vezérlőben a négyzet és a szöveg (Text) elhelyezkedését szabályozza. A szöveg állhat például a négyzet bal oldalán, a jobb oldalán, alatta vagy felette.

Az alapértelmezett CheckedChanged esemény akkor lép fel, ha a Checked tulajdonság értéke megváltozik. A CheckState tulajdonság értékének változását jelzi a CheckStateChanged esemény.

A RadioButton (választógomb)

A választógombok két állapottal rendelkeznek:

 Bekapcsolt állapotban a Checked tulajdonság true értékű, és a kör közepén egy pont

jelenik meg.

 Kikapcsolt állapotban a Checked értéke false, a kör pedig üres:

Az AutoCheck és a CheckAlign tulajdonságok valamint az alapértelmezett CheckedChanged esemény a jelölőnégy-zetek azonos nevű tagjaihoz hasonlóan alkalmazhatók.

A választógombokat általában csoportokba szervezve használjuk. Egy csoportba tartozó választógombok közül csak egy lehet bekapcsolt állapotú. Ha egy nem bekapcsolt gombon kattintunk, akkor az bekapcsolódik, és a csoport többi választógombja pedig kikapcsolt lesz. A formon és a különböző csoportosító vezérlőkben (Panel, GroupBox) elhelyezett választógombok egymástól függetlenül működnek.

Jól strukturált megoldást alakíthatunk ki, ha felhasználjuk a vezérlőelemek object típusú Tag tulajdonságát, melynek alapértéke null.

A példában a Tag tulajdonságban különböző szövegeket tárolunk, melyeket a választógombok bejelölésekor szövegmezőbe írunk. A megoldás alapötlete, hogy csupán egyetlen CheckedChanged eseménykezelőt készítünk (a radioButton1 gombhoz), amit aztán a másik két gomb eseménykezelőjeként is beállítunk a tulajdonságok ablakban.

A közös eseménykezelő metódus sender paraméterét RadioButton típusúvá átalakítjuk, majd megnézzük, hogy bejelölt állapotba került-e gomb. Amennyiben igen, a Tag tulajdonságban tárolt objektumot szöveggé alakítva a szövegmezőbe másoljuk. A megoldás akkor működik hiba nélkül, ha csak választógombok szerepelnek a csoportban – ellenkező esetben futás közbeni hibajelzést kapunk.

namespace WindowsFormsApplication { public partial class Form1 : Form { public Form1() { InitializeComponent(); } private void radioButton1_CheckedChanged(object sender, EventArgs e) { RadioButton rb = (RadioButton)sender; if (rb.Checked) textBox1.Text = rb.Tag.ToString(); } private void Form1_Load(object sender, EventArgs e) { radioButton1.Tag = "Választógomb1"; radioButton2.Tag = "Választógomb2"; radioButton3.Tag = "Választógomb3"; } } }

F.3.2.4 Szövegmegjelenítők

A Label (címke) és a TextBox (szövegmező) szövegmegjelenítő vezérlőket szöveges információk megjelenítésére és kezelésére használjuk. A vezérlők egy-, illetve többsoros szöveget is tárolhatnak. A megjelenítésen kívül a szövegmezőt szöveg bevitelére is használhatjuk, sőt a bevitt szöveget el is rejthetjük egy jelszókarakter beállításával.

Alaphelyzetben a szövegmező körül egy keret látható, ami azonban hiányzik a címkéknél. Mindkét vezérlőnél a BorderStyle tulajdonság beállításával intézkedhetünk a keret megjelenéséről. (A BorderStyle felsorolás elemei: None, FixedSingle és Fixed3D.) A tárolt, illetve beírt szöveget mindkét esetben a Text tulajdonságon keresztül érhetjük el.

A Label (címke)

A címke vagy felirat szöveg megjelenítésére szolgál. Ennek megfelelően alapvető tulajdonságai a feliratok elhelyezkedésével és a szöveg megjelenítési módjának beállításával kapcsolatosak.

A címkéket általában szövegmezők vagy más vezérlők azonosítására, megjelölésére használjuk, a vezérlő fölé vagy mellé helyezve. A címkére nem lehet az inputfókuszt ráhelyezni, azonban egéreseményekkel rendelkezik.

A címkék AutoSize tulajdonsága alaphelyzetben true értékű, így a címke mérete mindig akkora, hogy elférjen benne a beírt szöveg. Rögzített méretű címkében a (ContentAlignment felsorolás típusú) TextAlign tulajdonság segítségével kilenc módon igazíthatjuk el a szöveget. (Ha a szöveg nem fér el címkében, akkor a szöveg elejétől kezdve annyi karakter jelenik meg, amennyi belefér.)

A címkék a megadott szöveget több sorban jelenítik meg, ha a szövegben '\n' (soremelés) karaktereket helyezünk el:

label1.Text = "Első sor\nMásodik sor\nHamadik sor";

A címkék semmilyen billentyűzet-eseményt nem fogadnak, azonban az egéreseményekhez eseménykezelőket készít-hetünk, mint például az alapértelmezett Click eseményhez.

A TextBox (szövegszerkesztő, szövegmező)

Az TextBox vezérlő alaphelyzetben egyszerű, egysoros szövegszerkesztési feladatok ellátására, szöveg beolvasására, információk kijelzésére használható. Ebben a módjában a PasswordChar tulajdonság megadásával elfedhetjük a begépelt karaktereket.

Amennyiben a MultiLine tulajdonságot true értékűre állítjuk, a szövegmező lehetővé teszi többsoros szöveg megjelenítését, bevitelét és szerkesztését. A bevitelhez még engedélyeznünk kell az <Enter> billentyű feldolgozását az AcceptsReturn logikai tulajdonság igaz értékűre állításával.

Programból többféleképpen is feltölthetjük a többsoros szövegezőt. Windows alatt a sorvégek jelzésére a "\r\n" Escape-szekvenciákat kell használnunk:

textBox2.Text = "Első sor\r\nMásodik sor\r\nHarmadik sor"; // vagy textBox2.Text = "Első sor"+Environment.NewLine+"Második sor"+Environment.NewLine+"Harmadik sor";

A többsoros szövegmezőkhöz (vízszintes, függőleges vagy mindkét irányú) görgetősávokat is illeszthetünk. A vízszintes görgetősáv csak akkor jelenik meg, ha a WordWrap (szövegtördelés) tulajdonság értékét az alapértelmezett true helyett a false–ra állítjuk.

Az egy- és többsoros szövegmezőn belül a szöveget balra, jobbra vagy középre igazíthatjuk a (HorizontalAlignment felsorolás típusú) TextAlign tulajdonság segítségével. A MaxLenght tulajdonsággal pedig korlátozhatjuk a tárolt (bevitt) szöveg maximális hosszát. A szövegmezőkben tárolt egy vagy több sor tartalmát a sztringtömb típusú Lines tulajdonság felhasználásával is elérhetjük – a Lines.Length pedig megadja a szövegsorok számát.

Amennyiben a szövegmező ReadOnly tulajdonságát igaz értékűre állítjuk, valamint hamisra módosítjuk a TabStop tulajdonságot, a szövegmező címkeként használható.

Szövegszerkesztési műveletek

A szövegmezők kezelik a bennük tárolt szöveg valamely részletének (<Shift+nyíl> billentyűkkel vagy egérrel történő) kiválasztását. Az alábbi tulajdonságokkal azonosíthatjuk a kiválasztott szövegrészt, illetve programból magunk is kiválaszthatunk egy részszöveget:

SelectionStart

Az első kiválasztott karakter pozícióját tárolja (0 az első karakter indexe). Ha nincs kiválasztott szöveg, a SelectionStart a szövegkurzor (caret) pozícióját tartalmazza.

SelectionLength

A kiválasztott karakterek számát tárolja. (Többsoros esetben a kiválasztott karakterek száma sorvégenként 2-vel növekszik.)

SelectedText

A felhasználó által kiválasztott szövegrészt tartalmazza.

Select()

Programból a kiválasztás elvégzése a pozíció és a hossz beállítása után.

SelectAll()

A teljes tárolt szöveg kiválasztása.

A kiválasztott szöveget metódusokkal kivághatjuk - Cut(), a vágólapra (clipboard) másolhatjuk – Copy(), a vágólapról beilleszthetjük() – Paste(). A Paste() string paramétert fogadó változatával a kiválasztott szöveget a megadott sztringre cserélhetjük. A Clear() metódus hívásával törölhetjük a szövegmező tartalmát.

A CanUndo logikai tulajdonság valamint az Undo() és ClearUndo() metódusok felhasználásával az elvégzett műveletek érvénytelenítését is kezelhetjük.

A szövegmezők alapértelmezés szerint eseménye a TextChanged, ami jelzi, hogy a vezérlő tartalma megváltozott, akár a programból, vagy akár felhasználói beavatkozások eredményeként.

Az alábbi példában a formon két szövegmező helyezkedik el. Ha a textBox1 szövege változik, akkor a textBox2-be átmásoljuk az első szövegmező tartalmát, nagybetűssé alakítva. A megoldáshoz a textBox1 TextChanged eseményét kezelő metódusban dolgozunk:

private void textBox1_TextChanged(object sender, EventArgs e) { textBox2.Text = textBox1.Text.ToUpper(); }

Az alábbi példában egy sztringtömb elemeivel inicializáljuk a többsoros szövegmezőt, gombnyomásra pedig kiolvassuk a sorait és üzenetablakban meg is jelenítjük azokat:

namespace WindowsFormsApplication { public partial class Form1 : Form { string[] szöveg = { "Alfa", "Béta", "Gamma", "Delta" }; public Form1() { InitializeComponent(); } private void Form1_Load(object sender, EventArgs e) { textBox1.Multiline = true; textBox1.Height = 120;

textBox1.ScrollBars = ScrollBars.Vertical; textBox1.Lines = szöveg; } private void button1_Click(object sender, EventArgs e) { string se = string.Empty; szöveg = textBox1.Lines; foreach (string s in szöveg) se += s + "\n"; MessageBox.Show(se); } } }

F.3.2.5 Listavezérlők

Gyakran előfordul, hogy sok adat közül kell egyet vagy többet kiválasztanunk a programunk futásához. Ezen adatok kompakt és áttekinthető elrendezéséhez a program ablakában lista vezérlőket használhatunk. A listavezérlők megjelenítik a lista elemeit (items), amelyek közül a felhasználó választhat. A választásról esemény értesíti a programot, míg a kiválasztott elem(ek)et tulajdonságok azonosítják.

A ListBox típusú vezérlő a közönséges lista (listaablak), míg a CheckedListBox típusú komponens a listaelemek előtt jelölőnégyzeteket tartalmaz. Egy szövegmező és egy lista egybeépítéseként keletkezett a ComboBox típusú kombinált lista, melynek speciális, helytakarékos változata a lenyíló lista.

A listavezérlők mindegyikének lelke az elemeket (objektumokat) tároló, ObjectCollection típusú Items tulajdonság. Az Items.Count jellemző az listában tárolt elemek számát tartalmazza. Az elemeket pedig az Items indexelésével érjük el. Az Items objektum metódusaival listaműveletek végezhetünk:

Add(objektum)

Az objektum hozzáadása a lista végéhez.

AddRange(listBox2.Items)

A listBox2 elemeinek hozzáadása a lista végéhez.

AddRange(objektumTömb)

Az objektumTömb elemeit hozzáadja a lista végéhez.

Clear()

A lista elemeinek törlése.

Contains(objektum)

Igaz értékkel tér vissza, ha az objektum benne van a listában.

CopyTo(objektumTömb, index)

A lista elemeit kimásolja egy létező objektumtömbbe, adott indextől kezdve.

IndexOf(objektum)

Megadja, hogy az objektum milyen index alatt szerepel a listában. Ha az objektum nincs a listában, -1 értékkel tér vissza.

Insert(index, objektum)

Az objektum beszúrása a listába, az index pozícióba.

Remove(objektum)

Az objektum törlése a listából.

RemoveAt(index)

Az adott indexű pozíción található elem törlése a listából.

Felhívjuk a figyelmet arra, hogy az Items ugyan objektumokat (pontosabban objektumreferenciákat) tárol, a listavezérlők Windows megfelelői szövegekkel dolgoznak. Példaként tekintsük az összes listafajta közös logikai típusú Sorted tulajdonságát, melynek true értékre állításával a listák elemei szövegként rendezve jelennek meg a

97

vezérlőkben. Ahhoz, hogy számok sorrendje megfelelő legyen a rendezett listában, a számokat formázott szövegként kell a listához adnunk, mint ahogy ezt az alábbi példában tesszük:

public partial class Form1 : Form { Random rnd = new Random(); public Form1() { InitializeComponent(); } private void Form1_Load(object sender, EventArgs e) { listBox1.Sorted = listBox2.Sorted = true; listBox1.ScrollAlwaysVisible = listBox2.ScrollAlwaysVisible = true; int szám; for (int i = 0; i < 10; i++) { szám = rnd.Next(5, 15 + 1); listBox1.Items.Add(szám); listBox2.Items.Add(szám.ToString("00")); } } }

A ListBox (lista)

A lista lényege, hogy több szöveges elem közül kiválaszthatunk egyet vagy többet. Az elemek megadásához a program fejlesztése során a Properties ablak az Items tulajdonság sorában található gombon kell kattintanunk. A megjelenő „String Collection Editor” ablakban pedig megadhatjuk az elemeket.

Ha a lista elemei nem férnek el a lista ablakában, automatikusan egy függőleges görgetősáv jelenik meg a lista jobb oldalán. (A görgetősáv mindig látható, ha a ScrollAlwaysVisible tulajdonságot igaz értékűre állítjuk.) A vízszintes görgetősáv elhelyezéséről a lista alján a HorizontalScrollbar logikai tulajdonság true értékre való állításával gondoskodhatunk.

Ha a MultiColumn logikai tulajdonság true értékű, a listában annyi oszlop jelenik meg, hogy szükségtelenné váljon a függőleges görgetősáv. Az oszlopok szélességét a ColumnWidth tulajdonság szabályozza, melynek 0 alapértéke esetén az oszlopok alapértelmezett szélességűek lesznek.

98

A listákat felhasználásuk alapján két csoportba sorolhatjuk. Az egyik csoportba azok a listák tartoznak, amelyekből csak egyetlen elemet választhatunk. A másik csoportba tartozó

listák elemei közül pedig egyszerre többet is kiválaszt-hatunk. A (SelectionMode felsorolás típusú) SelectionMode tulajdonság értéke dönti el, hogy lehet-e több elemet kiválasztani.

A SelectionMode felsorolás értékei:

None

Egyetlen elem sem választható.

One

Pontosan egy elem választható.

MultiSimple

Több elem is kiválasztható az egér segítségével.

MultiExtended

Több elem is kiválasztható az egér, illetve a billentyűzet (<Ctrl>, <Shift>, nyilak) segítségével.

Az elemek kiválasztását a lista maga kezeli, és a választásnak megfelelően a SelectedIndex tulajdonságba írja az éppen kiválasztott elem sorszámát, míg többszörös választásnál frissíti az indexeket tartalmazó (ListBox. SelectedIndexCollection típusú) SelectedIndices kollekció tartalmát. Ha nincs elem kiválasztva, a SelectedIndex értéke -1, a SelectedIndices kollekció pedig üres. Minkét tulajdonság megváltozását a SelectedIndexChanged alapértelmezett esemény jelzi a programunknak.

A sorszámokon túlmenően a kiválasztott elem(ek) is elérhető(k) a SelectedItem, illetve többszörös kiválasztású lista esetén a (ListBox.SelectedObjectCollection típusú) SelectedItems tulajdonságban.

Metódusok is segítik a választások kezelését. A GetSelected(index) metódus true értékkel jelzi, ha a megadott indexű elemet kiválasztották. A SetSelected(index, true/false) metódus hívásával az adott elemet kiválasztottá tehetjük (true második argumentum), illetve megszüntethetjük a kiválasztottságát (false második argumentum).

Az elemek kiválasztott állapotát a ClearSelected() metódus hívásával, vagy a SelectedIndex tulajdonság -1 értékre való átállításával szüntethetjük meg.

Az egyszeres választás kezelését az alábbi Form1 osztály mutatja:

public partial class Form1 : Form { public Form1() { InitializeComponent(); } private void Form1_Load(object sender, EventArgs e) { string[] munkanapok = {"hétfő", "kedd", "szerda", "csütörtök", "péntek"}; listBox1.Items.Clear(); listBox1.SelectionMode = SelectionMode.One; listBox1.Items.AddRange(munkanapok); listBox1.SetSelected(1, true); } private void listBox1_SelectedIndexChanged(object sender, EventArgs e) {

MessageBox.Show(listBox1.SelectedItem.ToString()); // vagy int index = listBox1.SelectedIndex; MessageBox.Show(listBox1.Items[index].ToString()); } }

Többszörös választás esetén kissé bonyolultabb a program:

public partial class Form1 : Form { public Form1() { InitializeComponent(); } private void Form1_Load(object sender, EventArgs e) { string[] munkanapok = {"hétfő", "kedd", "szerda", "csütörtök", "péntek"}; listBox1.SelectionMode = SelectionMode.MultiSimple; listBox1.Items.Clear(); listBox1.Items.AddRange(munkanapok); listBox1.SetSelected(1, true); listBox1.SetSelected(3, true); } private void listBox1_SelectedIndexChanged(object

sender, EventArgs e) { string sm = string.Empty; foreach (string s in listBox1.SelectedItems) sm += s + ", "; sm = sm.TrimEnd(new char[] { ',', ' ' }); MessageBox.Show(sm); // vagy sm = "";

100

int index = 0; for (int i = 0; i <listBox1.SelectedIndices.Count; i++) { index = listBox1.SelectedIndices[i]; sm += listBox1.Items[index] + ", "; } sm = sm.TrimEnd(new char[] { ',', ' ' }); MessageBox.Show(sm); } }

A listában tárolt szövegek feldolgozását segítik a különböző szövegkereső metódusok, amelyek sikertelen esetben ListBox.NoMatches értékkel térnek vissza, míg sikeres esetben a megtalált elem indexével:

FindString(sztring)

Az első olyan elem keresése, amely a megadott szöveggel kezdődik.

FindString(sztring, index)

Az első olyan elem keresése az adott sorszámú elemtől, amely a megadott szöveggel kezdődik.

FindStringExact(sztring)

Az első olyan elem keresése, amely teljesen megegyezik a megadott szöveggel.

FindStringExact(sztring, index)

Az első olyan elem keresése az adott sorszámú elemtől, amely teljesen megegyezik a megadott szöveggel.

A CheckedListBox (bejelölt lista)

A bejelölt lista az egyszeres kijelölésű listához hasonlóan használható azzal a kiegészítéssel, hogy az elemek előtt jelölőnégyzet áll. Az alapértelmezett SelectedIndexChanged esemény továbbra is valamely elem kiválasztását jelzi, míg az új ItemCheck esemény a bejelölt állapot megváltozását közvetíti a programnak.

A bejelölt elemek indexét valamint hivatkozását ebben az esetben is kollekciók tárolják: CheckedIndices és CheckedItems. A CheckOnClick logikai tulajdonság true-ra állításával a jelölőnégyzet gyorsabban vált bejelölt állapotba az egérkattintás során.

A SetItemChecked(index, true/false) és SetItemCheckState(index, állapot) metódusok az elemek jelöltségének programból való kezelését (a jelölt állapot beállítását, törlését) segítik. A bool típusú GetItemChecked(index) és a CheckState típusú GetItemCheckState(index) függvényekkel lekérdezhetjük a lista adott indexű elemének bejelölt állapotát.

public partial class Form1 : Form { public Form1() { InitializeComponent(); } private void Form1_Load(object sender, EventArgs e) { string[] munkanapok = { "hétfő", "kedd", "szerda", "csütörtök", "péntek" };

101

checkedListBox1.Items.Clear(); checkedListBox1.Items.AddRange(munkanapok); checkedListBox1.SetSelected(1, true); checkedListBox1.SetItemChecked(4, true); checkedListBox1.SetItemCheckState(3, CheckState.Checked); } private void checkedListBox1_SelectedIndexChanged(object sender, EventArgs e) { MessageBox.Show(checkedListBox1.SelectedItem.ToString(), "Kiválasztott"); // vagy int index = checkedListBox1.SelectedIndex; MessageBox.Show(checkedListBox1.Items[index].ToString(), "Kiválasztott"); } private

void checkedListBox1_ItemCheck(object sender, ItemCheckEventArgs e) { string sm = string.Empty; foreach (string s in checkedListBox1.CheckedItems) sm += s + ", "; MessageBox.Show(sm, "Bejelölt"); // vagy sm = ""; int index = 0; for (int i = 0; i < checkedListBox1.CheckedIndices.Count; i++) { index = checkedListBox1.CheckedIndices[i]; sm += checkedListBox1.Items[index] + ", "; } MessageBox.Show(sm, "Bejelölt"); } }

A ComboBox (kombinált lista)

Mint már korábban említettük a ComboBox kombinált lista egy egysoros szövegmező és egy egyszeres kiválasztású lista összeépítéséből keletkezett. Ennek következtében a kombinált lista tulajdonságai, metódusai és eseményei megfelelnek a TextBox és a TListBox osztályok tagjainak.

A kombinált lista többféle típusát használhatjuk a (ComboBoxStyle felsorolás típusú) DropDownStyle tulajdonság értékétől függően:

- Az egyszerű (Simple) kombinált lista esetén mindig megjelenik a listaablak, és a szövegmező tartalma módosítható. A listaablak mérete rögzített, és nem lehet becsukni. A szövegmezőbe tetszőleges szöveget írhatunk, vagy a listából is választhatunk egy elemet.
- A lenyíló (DropDown) kombinált lista (ez az alapértelmezés) működése annyiban tér el az egyszerű kombinált listától, hogy listaablak nem jelenik meg automatikusan. A listaablak megjeleníthető a szövegmező oldalán található nyomógombon való kattintással, vagy programból a DroppedDown logikai tulajdonság állításával.
- A lenyíló (DropDownList) lista annyiban különbözik a lenyíló kombinált listától, hogy a szövegmező tartalma csak listaelem választásával módosul. Általában akkor használjuk, ha a felhasználónak adott tételek közül kell választania. Alapértelmezés szerint a lenyíló listaablakok 8 elemet jelenítenek meg.

Megjegyezzük, hogy a MaxDropDownItems egész típusú tulajdonság értékének változtatásával növelhetjük vagy csökkenthetjük a megjelenített tételek számát. Programból a SelectedIndex tulajdonság segítségével állíthatjuk be, hogy melyik listaelem jelenjen meg a szövegmezőben.

A kombinált lista listájából történő választást jelző SelectedIndexChanged esemény mellet további események is rendelkezésünkre állnak. A TextUpdate esemény jelzi, hogy a szövegmező tartalmát a felhasználó módosította. (Ez az esemény nem jelzi, ha a szövegmező tartalmát a programból változtatjuk meg.) Természetesen a TextChanged esemény is elérhető. Lenyíló listák esetén a DropDown esemény tudatja, hogy a lista lenyílt, illetve a DropDownClosed esemény pedig azt, hogy bezáródott.

Az alábbi példában elektronikus szorzótáblát készítünk kombinált listák felhasználásával:

public partial class Form1 : Form { public Form1() { InitializeComponent(); } private void Form1_Load(object sender, EventArgs e) { for (int i=1; i<=10; i++) { comboBox1.Items.Add(i); comboBox2.Items.Add(i); } comboBox1.SelectedIndex = 0; comboBox2.SelectedIndex = 0;

textBox1.Text = "1"; } private void button1_Click(object sender, EventArgs e) { int a = Convert.ToInt32("0"+comboBox1.Text); int b = Convert.ToInt32("0"+comboBox2.Text); int c = a * b; textBox1.Text = c.ToString(); } private void comboBox1_TextChanged(object sender, EventArgs e) {

button1.PerformClick(); } private void comboBox2_TextChanged(object sender, EventArgs e) { button1.PerformClick(); } }

104

F3.2.6 Görgethető vezérlők

Ebbe a csoportba tartoznak a vízszintes és a függőleges görgetősávok (VScrollbar, HScrollbar), valamint a numerikus fel-le léptető vezérlő (Numeric-UpDown).

A görgethető vezérlők közös jellemzője, hogy csúszka és/vagy nyilak segítségével egy értéktartományban lépegethetünk. Ehhez a működéshez be kell állítanunk a tartomány legkisebb (Minimum) és legnagyobb (Maximum) elemét, valamint a lépések hosszát.

Görgetősávok esetén a hosszabb és a rövidebb csúszkaelmozdulásokhoz különböző lépéshosszt is megadhatunk a LargeChange és SmallChange tulajdonságokkal. A numerikus léptető esetén csupán egyetlen lépéshosszt állíthatunk be az Increment tulajdonságban. A vezérlők aktuális állapotának megfelelő értéket a Value tulajdonság tartalmazza, amely csuszkák esetén int típusú, míg a numerikus léptető estén nagy pontosságú valós (decimal) típusú.

A numerikus léptetőben látható szám megjelenését további tulajdonságokkal szabályozhatjuk: DecimalPlaces, ThousandsSeparator és Hexadecimal.

Mindhárom vezérlő esetén a Value tulajdonság értékének bármilyen megváltozását a ValueChanged esemény jelzi. A görgetősávoknál a görgetés műveletéről az alapértelmezett Scroll esemény tudósít.

namespace WindowsFormsApplication { public partial class Form1 : Form { public Form1() { InitializeComponent(); } private void Form1_Load(object sender, EventArgs e) { hScrollBar1.Minimum = 0; hScrollBar1.Maximum = 255; hScrollBar1.LargeChange = 10; hScrollBar1.SmallChange = 5; hScrollBar1.Value = 23; numericUpDown1.Minimum = (decimal)-Math.PI; numericUpDown1.Maximum = (decimal)Math.PI; numericUpDown1.Increment = 0.1M; numericUpDown1.Value = 0;

105

} private void hScrollBar1_Scroll(object sender, ScrollEventArgs e) { this.Text = hScrollBar1.Value.ToString(); } private void hScrollBar1_ValueChanged(object sender, EventArgs e) { this.Text = hScrollBar1.Value.ToString(); } private void numericUpDown1_ValueChanged(object sender, EventArgs e) { this.Text = numericUpDown1.Value.ToString(); } } }

106

F3.2.7 Időzítő (Timer) vezérlő

Az alkalmazások gyakran periódikusan ismétlődve hajtanak végre egyes feladatokat. Az időzítő objektumok bizonyos idő leteltével újra és újra meghívnak egy eseménykezelő metódust, így ismétlődve végrehajtanak egy adott feladatot.

Mivel az időzítő nem vizuális komponens, a formon nem jelenik meg - a Visual Studio a form alatt helyezi el az ikonját egy sávban.

Az időzítőt többször el lehet indítani (timer1.Enabled = true; vagy timer1.Start();), és meg lehet állítani (timer1.Enabled = false; vagy timer1.Stop();), továbbá meg lehet változtatni azt az intervallumot (Interval), aminek elteltével végrehajtja az (Tick) eseménykezelő metódust.

Az int típusú Interval tulajdonság szabályozza az időzítő működését. Az Interval tulajdonságban megadott, ezredmásodpercben értendő érték meghatározza, hogy milyen időközönként aktivizálódjon az időzítő Tick alapeseménye.

Az alábbi példában gombnyomásra indítunk, illetve leállítunk egy időzítőt, ami másodpercenként növel egy számlálót, melynek értékét megjelenti az ablak fejlécében:

public partial class Form1 : Form { int számláló = 0; private void timer1_Tick(object sender, EventArgs e) { számláló++; this.Text = számláló.ToString(); } private void Form1_Load(object sender, EventArgs e) { timer1.Interval = 1000; // 1 másodperc timer1.Enabled = false; button1.Text = "Start"; } private void button1_Click(object sender, EventArgs e) { timer1.Enabled = !timer1.Enabled; if (timer1.Enabled)

107

button1.Text = "Stop"; else button1.Text = "Start"; } }

108

F3.3 Ablakmenü készítése

A Windows alkalmazások többsége menürendszert használ a különféle részfeladatok végrehajtásának kezdeményezéséhez. A Visual Studio menütervezője lehetővé teszi – új menüpontok hozzáadásával, törlésével valamint átnevezésével – a menürendszerek könnyű és hatékony kialakítását.

Az ablakmenü a program ablakának címsora alatt helyezkedik el, és egy menüsorból (főmenüből) és abból lenyíló menükből áll. Természetesen a lenyíló menüknek is lehetnek újabb lenyíló menüi (almenüi).

A lenyíló menüt aktiváló menüpontok mellett egy jobbra mutató háromszög jelzi a további menüelemeket. A menüket legtöbbször nyomógomb vezérlők helyett alkalmazzuk. Különösen akkor van szükség menüre, amikor a form teljes területét szeretnénk használni, például grafika megjelenítése céljából.

F3.3.1 A főmenü létrehozása

A főmenü létrehozásához egy MenuStrip típusú komponenst kell a formra helyeznünk. Ez önmagában nem vizuális elem, csak a működésének eredménye jelenik meg a formon.

Tegyük aktívvá a menuStrip1 ikonját a form alatti sávban, majd pedig kattintsunk a form menüsorában a „Type Here ” feliratú mezőre, és írjuk be a „&Fájl” szöveget! (A & karakter hatására a Fájl menüpont kiválasztható lesz az <Alt+F> billentyűkombinációval, és az F betű aláhúzva jelenik meg.)

Miután létrejött a Fájl menüpont a Visual Studio létrehozott hozzá egy lenyíló menüt, és annak az első menüpontját:

Kattintsunk ismét a bekeretezett „Type Here” mezőre, és adjuk meg a „&Megnyitás…”, majd pedig megismételve a műveletet a „M&entés…” menüpontokat. (A három pont azt jelöli, hogy a menüválasztás hatására valamilyen párbeszédbak fog megnyílni.) Ha az egérmutatót a „Type Here” mező fölé visszük, annak bal oldalán megjelenik egy nyíl, amellyel egy menü nyitható meg. A menüpontok segítségével menüpontot, kombinált listát, elválasztó vonalat

109

vagy szövegmezőt helyezhetünk el a menünkben. Menüpont létrehozásakor egy ToolStripMenuItem típusú objektum jön létre, melynek toolStripmenuItemN neve lesz a menüpontban, amit természetesen módosíthatunk.

Példaként adjunk egy elválasztó vonalat és egy „&Kilépés” menüpontot a menünkhöz! Folytassuk a menüépítést egy újabb lenyíló menüvel! Ehhez kattintsunk a Fájl menüpont melletti „Type Here” mezőre, és adjuk meg a „&Szerkesztés” nevet, majd pedig a fentiekben bemutatott módon adjuk hozzá a Szerkesztés menühöz a Kivágás, Másolás és Beillesztés menüpontokat!

Utolsó lépésként adjunk a menünkhöz egy bejelölt „Automatikus mentés”menüpontot. Ehhez a menüpont szokásos megadása után az egér jobb gombjával kattintva az újonnan létrehozott menüponton, felbukkan egy menü, amelyből a Checked menüpontot kell kiválasztanunk. Ahhoz, hogy a jelölés működjön, a menüpont tulajdonságai közül a CheckOnClick-et állítsuk true értékűre!

A kijelölt menüponthoz gyorsító billentyűt (például <Ctrl+Shift+Alt+F1>, <Ctrl+K>) is rendelhetünk, a ShortcutKeys tulajdonság gombján való kattintás után megjelenő ablakból választva. A kiválasztott gyorsító billentyű automatikusan megjelenik a menüponttól jobbra.

Az elkészült menüt a program futtatásával ellenőrizhetjük. A programból kilépve a menüt módosíthatjuk vagy továbbfejleszthetjük.

Tervezési módban egérkattintással tudjuk kiválasztani a szerkesztendő menüt, melyen belül az egér mozgatásával lépkedhetünk a menüpontok között. Az elvégezhető műveleteket a menüpont (jobb egérkattintás hatására) felbukkanó menüjéből érjük el:

Set Image..

A menüpont elé egy kis képet szúrhatunk be..

Enabled

A menüpont engedélyezhető, tiltható.

Checked

A menüpont elé egy jelölőnégyzet helyezhető.

ShowShortcutKeys

A gyorsító billentyűk megjelenítésének engedélyezése, illetve tiltása.

Convert To

A menüpont átalakítása a négy alap menüpontok (kombinált listává, elválasztó vonallá vagy szövegmezővé) egyikévé.

Insert

A négy alap menüpontból választott, új menüpont beszúrása az aktuális menüpont elé

Edit DropDownItems…

A lenyíló menü elemeinek szerkesztése.

Select

A form, a menüsor vagy a menüpontot tartalmazó menü kijelölése.

Cut, Copy, Paste, Delete

Szerkesztési műveletek a menüponttal.

Document Outline

A menürendszer vázlatos megjelenítése.

Properties

A menüpont tulajdonságainak megjelenítése a Properties ablakban.

110

F3.3.2.A menüválasztás kezelése C# programból

A normál menüpontokhoz az alapértelmezés szerinti Click esemény kezelőjében rendelhetünk műveleteket. Az eseménykezelő kódját egyszerűen elérhetjük, ha tervezési módban a kérdéses menüponton kétszer kattintunk az egér bal gombjával. Példaként nézzük a Kilépés menüponthoz rendelt eseménykezelő metódust:

private void kilépésToolStripMenuItem_Click(object sender, EventArgs e) { Close(); }

Az eseménykezelő nevéből láthatjuk, hogy a Visual Studio menütervezője az általunk megadott felirat (Text) felhasználásával alakította ki a menüpont objektum nevét (Name). A nevet az első eseménykezelő elkészítése előtt érdemes megváltoztatni, mivel a névváltoztatás előtt létrehozott eseménykezelő metódusok neve nem változik meg.

A következő példában a bejelölt menüpont jelölésének kezelését magunk végezzük a menüpont Click esemény-kezelőjében:

private void automatikusMentésToolStripMenuItem_Click(object sender, EventArgs e) { automatikusMentésToolStripMenuItem.Checked = !automatikusMentésToolStripMenuItem.Checked; }

111

F3.4 Párbeszédablakok

A Windows alkalmazások általában több ablakot (formot) tartalmaznak. Az ablakok közül egy adja az alkalmazás menüvel ellátott főablakát, míg a továbbiak párbeszédablakok, amelyek menüpontok kiválasztásakor vagy nyomógombok lenyomásával jelennek meg.

Saját tervezésű párbeszédablak készítéséhez a Windows Forms Application projekthez további formokat kell adnunk a Project/Add Windows Form… menüpont kiválasztásával. Párbeszédablakok elkészítésekor ugyanazokat a szabályokat kell figyelembe vennünk, mint egyablakos alkalmazások esetén.

F3.4.1 Modális és nem modális párbeszédablakok

Az ablakok és a párbeszédablakok a megjelenítési módjuktól függően lehetnek modálisak vagy nem modálisak. A modális ablakot be kell zárni ahhoz, hogy dolgozhassunk a program más részeivel (ablakaival). Azokat a párbeszédablakokat, amelyek fontos információkat jelenítenek meg, javasolt modálisként létrehozni. (Ilyen modális ablakok a MessageBox típusú párbeszédablakok.) A modális megjelenítéshez a formok ShowDialog() metódusát használjuk, amely csak a párbeszédablak bezárása után tér vissza a hívóhoz. A párbeszédablak automatikusan bezáródik, ha a rajta elhelyezett gombok DialogResult tulajdonságának a None értéktől eltérő értéket (Ok, Cancel stb.) állítunk be.

Az alábbi példában modális párbeszédablakban kérjük be a felhasználó nevét és e-mail címét. A Form2 párbeszédablak osztálya:

public partial class Form2 : Form { private void Form2_Load(object sender, EventArgs e) { button1.DialogResult = System.Windows.Forms.DialogResult.OK; button2.DialogResult = System.Windows.Forms.DialogResult.Cancel; this.AcceptButton = button1; this.CancelButton = button2; } public void Kiolvas(out string név, out string email) { név = textBox1.Text; email = textBox2.Text; } }

A program főablaka csupán egy nyomógombot tartalmaz, melynek megnyomásakor megjelenik a párbeszédablak:

public partial class Form1 : Form {

112

private void button1_Click(object sender, EventArgs e) { Form2 adatbevitel = new Form2(); adatbevitel.StartPosition = FormStartPosition.CenterScreen; DialogResult dr = adatbevitel.ShowDialog(); string név = "", email =""; adatbevitel.Kiolvas(out név, out email); if (dr == System.Windows.Forms.DialogResult.OK) { MessageBox.Show(név + "\t" + email); } } }

A nem modális ablakok lehetővé teszik, hogy az ablakok között váltsunk azok bezárása nélkül. Az ilyen ablakok programból való kezelése sokkal nehezebb, mint a modálisoké, mivel az ablakokat a felhasználó bármilyen sorrendben elérheti. A nem modális megjelenítést az ablakok Show() metódusa biztosítja, amely azonnal visszatér a hívás helyére, mihelyt megjelenik a párbeszédablak. Ebben az esetben a párbeszédablak bezárásáról, és az ablak információinak átadásáról a szülő ablaknak magunknak kell gondoskodni (Close()).

Az alábbi példában tetszőleges számú névjegyablakot megjelenthetünk. A névjegyet tartalmazó Form2 osztály:

public partial class Form2 : Form { private void Form2_Load(object sender, EventArgs e) { this.AcceptButton = button1; } private void button1_Click(object sender, EventArgs e) { this.Close(); } }

A főablak gombnyomás eseménykezelőjében megjelenítjük a nem modális párbeszédablakot:

private void button1_Click(object sender, EventArgs e) { Form2 névjegy = new Form2(); névjegy.Show();

113

}

F3.4.2 A MessageBox modális üzenetablak

A MessageBox osztály statikus Show() metódusának hívásával különböző elemeket tartalmazó üzenetablakokat jeleníthetünk meg. A Show() metódus 21 túlterhelt változata közül csak néhány bemutatására szorítkozunk, azonban mindegyik DialogResult típusú értékkel tér vissza.

Legegyszerűbb esetben csupán egy üzenetszöveg paraméterrel rendelkezik a metódus:

DialogResult Show(string text).

A hívásokban mindig egy-egy argumentummal többet adhatunk meg, egészen az alábbi formáig:

DialogResult Show(string text, // az üzenet szövege

string caption, // az ablak címsora

MessageBoxButtons buttons, // nyomógombok

MessageBoxIcon icon) // ikon

Nézzünk néhány konkrét üzenetablak-kialakítást!

MessageBox.Show("Üzenet", "Fejléc");

MessageBox.Show("Üzenet", "Fejléc",

MessageBoxButtons.YesNo,

MessageBoxIcon. Question);

A metódus által visszaadott értékből megtudhatjuk, hogy melyik gomb megnyomásával zárult le az üzenetablak:

DialogResult eredmény = MessageBox.Show("Üzenet", "Fejléc", MessageBoxButtons.YesNo, MessageBoxIcon.Question); if (eredmény == System.Windows.Forms.DialogResult.Yes) MessageBox.Show("Igen :)"); else MessageBox.Show("Nem :(");

114

F3.4.3 Általános párbeszédablakok

A C# támogatja a Windows rendszer általános (common) párbeszédablakainak alkalmazását. Az eszköztár Dialogs csoportjában találjuk meg az egyes ablaktípusokat. Az általános párbeszédablakok használatában közös vonás, hogy tulajdonságok beállításával paraméterezzük az ablak megjelenését, illetve tulajdonságok tartalmazzák a felhasználó beállításait.

A párbeszédablakokat a form tervezése során is paraméterezhetjük, ha nem vizuális komponensként a formra helyezzük az ikonjukat. A megjelenítéshez futás közben kell hívnunk az objektumuk ShowDialog() metódusát.

Az alábbiakban azonban az ablakok típusosztályát használva teljes mértékben programból végezzük el a megjelenítést. A C# kódból jól nyomon követhető a hívások paraméterezése. Ha a felhasználói felület tervezésekor hozzuk létre az ablakobjektumot, akkor a példákból el kell hagyni az objektum new művelettel történő létrehozását.

Színkiválasztó (ColorDialog) párbeszédablak private void button1_Click(object sender, EventArgs e) { ColorDialog ablak = new ColorDialog(); // engedélyezzük az ablak színkeverő részének megjelenését ablak.AllowFullOpen = true; // minden lehetséges alapszín megjelenítését kérjük ablak.AnyColor = true; // nemcsak a tiszta színeket kérjük ablak.SolidColorOnly = false; // a megjelenéskor kiválasztott szín ablak.Color = Color.Red; if (ablak.ShowDialog() == DialogResult.OK) { this.BackColor = ablak.Color; } }

Könyvtárböngésző (FolderBrowserDialog) párbeszédablak private void button1_Click(object sender, EventArgs e) { FolderBrowserDialog ablak = new FolderBrowserDialog(); // a gyökérkönyvtár beállítása a System könyvtárra ablak.RootFolder = Environment.SpecialFolder.System; // az aktuális könyvtár lesz a kiválasztott

115

ablak.SelectedPath = Environment.CurrentDirectory; // engedélyezzük új könyvtár létrehozását ablak.ShowNewFolderButton = true; // leírás ablak.Description = "Könyvtárválasztás példa"; if (ablak.ShowDialog() == DialogResult.OK) MessageBox.Show(ablak.SelectedPath); }

Betűtípus (FontDialog) párbeszédablak private void button1_Click(object sender, EventArgs e) { FontDialog ablak = new FontDialog(); // a színlista megjelenítése ablak.ShowColor = true; // az Apply gomb megjelenítése ablak.ShowApply = true; // az Effect csoport megjelenítése ablak.ShowEffects = true; // a Help gomb megjelenítése ablak.ShowHelp = true; // a választható fontméretek ablak.MinSize = 12; ablak.MaxSize = 23; if (ablak.ShowDialog() != DialogResult.Cancel) {

116

this.Font = ablak.Font; } }

Fájlnyitás (OpenFileDialog) párbeszédablak

Egyetlen állomány kiválasztása:

private void button1_Click(object sender, EventArgs e) { string fájlnév = ""; OpenFileDialog ablak = new OpenFileDialog(); // az alapértelmnezett kiterjesztés ablak.DefaultExt = "txt"; // a lehetséges kiterjesztések ablak.Filter = "Szövegfájlok (*.txt)|*.txt|Minden fájl (*.*)|*.*"; // a szűrők közül az aktuális beállítása ablak.FilterIndex = 2; // kiindulási könyvtár ablak.InitialDirectory = "C:\\Munka"; // az aktuális könyvtár visszaállítása az ablak lezárását követően ablak.RestoreDirectory = true; // az ablak felirata ablak.Title = "Válasszon ki egy szövegfájlt"; // figyelmeztetést kérünk, ha a beírt fájl vagy útvonal nem létezik ablak.CheckFileExists = true; ablak.CheckPathExists = true;

117

// a bejelölt "Open as read-only" jelölőnégyzet megjelenítése ablak.ReadOnlyChecked = true; ablak.ShowReadOnly = true; if (ablak.ShowDialog() == DialogResult.OK) fájlnév = ablak.FileName; }

Több állomány kiválasztása:

private void button1_Click(object sender, EventArgs e) { string fájlnevek = ""; OpenFileDialog ablak = new OpenFileDialog(); // az alapértelmnezett kiterjesztés ablak.DefaultExt = "txt"; // a lehetséges kiterjesztések ablak.Filter = "Adatfájlok (*.txt;*dat)|*.txt;*dat|Minden fájl (*.*)|*.*"; // kiindulási könyvtár ablak.InitialDirectory = "C:\\Munka"; // az ablak felirata ablak.Title = "Válasszon ki adatfájlokat!"; // több fál is választható ablak.Multiselect = true;

118

if (ablak.ShowDialog() == DialogResult.OK) { foreach (string fájlnév in ablak.FileNames) { fájlnevek += fájlnév + "\n"; } MessageBox.Show(fájlnevek); } }

Fájlmentés (SaveFileDialog) párbeszédablak private void button1_Click(object sender, EventArgs e) { string fájlnév = ""; SaveFileDialog ablak = new SaveFileDialog(); // az alapértelmnezett kiterjesztés ablak.DefaultExt = "txt"; // a lehetséges kiterjesztések ablak.Filter = "Szövegfájlok (*.txt)|*.txt|Minden fájl (*.*)|*.*"; // a szűrők közül az aktuális beállítása ablak.FilterIndex = 2; // kiindulási könyvtár ablak.InitialDirectory = @"C:\Munka";

119

// az aktuális könyvtár visszaállítása az ablak lezárását követően ablak.RestoreDirectory = true; // az ablak felirata ablak.Title = "Válasszon ki egy szövegfájlt"; // figyelmeztetést kérünk, ha a beírt fájl vagy útvonal nem létezik ablak.CheckFileExists = true; ablak.CheckPathExists = true; if (ablak.ShowDialog() == DialogResult.OK) fájlnév = ablak.FileName; }

F4. Állományok kezelése

A legtöbb C# alkalmazás készítésekor fontos szempont a program bemenő adatainak kényelmes megadása. Ugyancsak jogos elvárás az alkalmazás futtatásakor keletkező adatok tárolása későbbi feldolgozáshoz. Mindkét esetben több szinten is találunk megoldásokat a .NET rendszerben, melyek közül ebben a részben a System.IO névtér osztályait hívjuk segítségül.

F4.1 A System.IO névtér osztályainak áttekintése

A .NET keretrendszerben a fájlalapú be- és kiviteli (Input/Output, I/O) műveleteket biztosító alapvető osztályok helye a System.IO névtér. A névtér több típusa a fájlrendszer mappáinak és állományainak programozott kezelésére összpontosít. Ezek alkalmazására a fejezet végén térünk vissza, adott feladatot elvégző C# kódrészletek összegyűjtésével.

Directory,

DirectoryInfo

Ezekkel az osztályokkal a számítógép könyvtárstruktúráját kezelhetjük. Míg a Directory típus funkcionalitását statikus tagok segítségével érhetjük el, addig a DirectoryInfo osztály hasonló működéséhez objektumokat kell létrehoznunk.

DriveInfo

A számítógép meghajtóiról szolgáltat részletes információkat.

120

File,

FileInfo

Metódusokkal segítik az állományok létrehozását, másolását, törlését, mozgatását és megnyitását, továbbá segítik FileStream típusú objektumok létrehozását. A műveletek a File osztály statikus metódusain, illetve a FileInfo osztály objektumain keresztül érthetők el.

Path

Statikus metódusokkal segíti a fájl- és könyvtárelérési utakat tartalmazó sztringek feldolgozását.

Figyelmünk középpontjában azonban az állományok különböző módon történő feldolgozását segítő osztályok állnak.

BinaryReader,

BinaryWriter

Lehetővé teszik egyszerű (egész, logikai, valós) és string típusú adatok bináris formában történő tárolását és visszaolvasását.

BufferedStream

Más adatfolyamok olvasási és írási műveleteihez ideiglenes tárolókat (puffereket) rendel.

FileStream

Támogatja fájlok bájtfolyamként történő elérését, feldolgozását.

MemoryStream

Adatfolyamot hoz létre a memóriában, és lehetővé teszi annak bájtfolyamként történő elérését.

Stream

Bájtfolyamok univerzális kezelésének absztrakt alaposztálya.

StreamWriter,

StreamReader

Ezekkel az osztályokkal soros elérésű fájlokban tárolhatunk, illetve onnan visszaolvashatunk szöveges információkat.

StringWriter,

StringReader

Sztringekben tárolt szöveges információk kezelését (írását/olvasását) segítő osztályok.

TextReader,

TextWriter

Bájtfolyamok karakteres feldolgozását segítő absztrakt alaposztályok.

F4.2 Adatfolyamok a .NET rendszerben

A .NET rendszerben a Stream osztály minden olyan osztály őse, amely bájtok tárolásával, továbbításával foglalkozik. A bájtok lehetnek fájlban (FileStream), memóriában (MemoryStream) tárolt adatok, hálózati kommunikáció elemei (NetworkStream), vagy akár egy titkosítási eljárás alanyai (CryptoStream).

0. bájt1. bájtn-1. bájt...Stream - bájtfolyam

Az adatok milyenségétől függetlenül a Stream alaposztály bájtonkénti írást/olvasást biztosít, illetve lehetővé teszi az aktuális bájtpozíció lekérdezését, módosítását (seeking). Amennyiben a bájtos adatelérés nem megfelelő számunkra, olvasó és író osztályok segítségével a bájtok sorozatát szövegsorként, illetve bináris adatként egyaránt értelmezhetjük.

Példaként tekintsük az alábbi bájtfolyamot:

65

32

67

35

32

78

89

69

76

86

Amennyiben a fenti bájtsort szövegként olvassuk be, a “A C# NYELV” szöveghez jutunk:

StreamReader sr = new StreamReader(@"D:\Adat.dat"); string s = sr.ReadLine(); sr.Close();

Ugyanezt a bájtsort egy-egy char, byte, int és float típusú változóba beolvasva az alábbi adatokat kapjuk:

121

BinaryReader br = new BinaryReader(File.Open(@"D:\Adat.dat", FileMode.Open)); char c = br.ReadChar(); byte b = br.ReadByte(); int v = br.ReadInt32(); float f = br.ReadSingle(); br.Close();

'A'

32

1310729027

5.614955E+13

(A programrészletek csak illusztrációként szerepelnek, a szükséges lépésekkel később ismerkedünk meg.)

Az adatfolyamok működésének megértését nagyban segítik, ha áttekintjük a Stream osztály nyilvános tulajdonságait és metódusait, melyeket a belőle származtatott osztályok mindegyike örökli.

A Stream osztály metódusai és tulajdonságai

Rövid leírás

CanRead, CanWrite, CanSeek

Megadja, hogy az adatfolyam támogatja-e az olvasás, írás és pozicionálás műveletét.

Close()

Bezárja az adatfolyamot, és felszabadítja a hozzá kapcsolódó erőforrásokat.

Flush()

Kiüríti az adatfolyamhoz tartozó ideiglenes tárolókat, amennyiben ilyenek tartoznak az adatfolyamhoz.

Length

Megadja az adatfolyam bájtban kifejezett hosszát.

Position

Megadja az aktuális adatfolyam-pozíciót (bájtsorszámot).

Read(), ReadByte()

Beolvas egy bájtsort (vagy egyetlen bájtot) az adatfolyamból, és a beolvasott bájtok számával módosítja az aktuális pozíciót az adatfolyamban.

Seek()

Beállítja az aktuális adatfolyam-pozíciót az adatfolyam elejéhez (SeekOrigin.Begin), aktuális pozíciójához (SeekOrigin.Current), illetve a végéhez (SeekOrigin.End) képest.

SetLength()

Beállítja az adatfolyam bájtban kifejezett hosszát.

Write(), WriteByte()

Egy bájtsort (vagy egyetlen bájtot) ír az adatfolyamba, és a kiírt bájtok számával módosítja az aktuális pozíciót az adatfolyamban.

122

Az adatfolyamok kezelésének általános lépéseit az alábbiak szerint csoportosíthatjuk.

1. Az adatfolyam megnyitása, melynek sikeres elvégzését követően a programunkban egy objektumon keresztül érhetjük el az adatokat.

2. Az adatok feldolgozása, amihez három művelet közül választhatunk: írás, olvasás, pozicionálás. (Amennyiben a pozíciót kiválaszthatjuk, tetszőleges (random) elérésű adatfolyamról beszélünk, míg ellenkező esetben az elérés soros, szekvenciális.)

3. A feldolgozás végeztével az adatfolyamot le kell zárni. Bizonyos esetekben a lezárást megelőzi az ideiglenes tárolók tartalmának mentése (flush).

F4.3 Fájlok bájtonkénti elérése - a FileStream osztály

Állományok bájtos adatfolyamként való kezeléséhez használhatjuk a FileStream osztály objektumait. Az objektumok létrehozásakor különböző konstruktorok közül választhatunk. A fájl elérési útvonalának megadása mellett intézkedhetünk a megnyitás módjáról, az adatok eléréséről, illetve arról, hogy a fájlhoz más program hozzáférhet-e.

FileStream(string útvonal, FileMode nyitási_mód)

FileStream(string útvonal, FileMode nyitási_mód, FileAccess elérés)

FileStream(string útvonal, FileMode nyitási_mód, FileAccess elérés, FileShare megosztás)

A konstruktorok hívásakor a FileMode, FileAccess és FileShare típusú paraméterek helyén felsorolások elemeit kell megadnunk.

Felsorolás

A felsorolás elemei

FileMode

Append (írható, létező fájl megnyitásakor annak végére pozícionál, vagy egy új fájlt hoz létre), Create (új állományt készít, már meglévő fájl esetén is), CreateNew (csak akkor hozza létre az új fájlt, ha az még nem létezik), Open (megnyit egy létező állományt), OpenOrCreate (megnyit egy létező fájlt, vagy újat készít, ha az nem létezik), Truncate (létező állományt nyit meg, a tartalmának eldobásával).

FileAccess

Read (olvasás), Write (írás), ReadWrite (olvasás és írás).

FileShare

None (más program nem éri el) , Read (csak olvashatja), Write (csak írhatja), ReadWrite (írhatja és olvashatja).

A Stream osztálytól örökölt tulajdonságokon és metódusokon túlmenően lehetőségünk nyílik egy adott fájlrész más programok előli zárolására: Lock(), illetve a zárolás feloldására: Unlock(). Ezen ritkán használt műveletek mellett ismerkedjünk meg a bájtok írására és olvasására használható metódusok paraméterezésével!

void WriteByte(byte érték)

int ReadByte()

void Write(byte[] tömb, int kezdőpozíció, int darab)

int Read(byte[] tömb, int kezdőpozíció, int darab)

Az elmondottak bemutatására készítsünk programot, amely egy kiválasztott állomány első 12 bájtját hexadecimális és karakteres formában is megjeleníti! A fájl kijelöléséhez használjuk a jegyzet F3.4.3. fejezetében bemutatott OpenFileDialog párbeszédablakot! Ügyeljünk arra, hogy a vezérlőkarakterek (kódjuk<32) nem jeleníthetők meg!

Egy PDF állomány első 12 bájtja:

using System.IO; // . . . private void button1_Click(object sender, EventArgs e) { string fájlnév = ""; OpenFileDialog ablak = new OpenFileDialog(); ablak.Filter = "Minden fájl (*.*)|*.*"; ablak.InitialDirectory = "C:\\Munka";

ablak.ReadOnlyChecked = true; if (ablak.ShowDialog() == DialogResult.OK) { fájlnév = ablak.FileName; try { byte[] adatok = new byte[12]; FileStream fs = new FileStream(fájlnév, FileMode.Open, FileAccess.Read); // nyitás fs.Read(adatok, 0, 12); // feldolgozás fs.Close(); // lezárás string s = ""; for (int i = 0; i < adatok.Length; i++) { s += " " + adatok[i].ToString("X2") + " "; // hexadecimális szám s += adatok[i] < 32 ? '.' : (char)adatok[i]; // karakter } MessageBox.Show(s); } catch (Exception ex) { MessageBox.Show(ex.Message); } } }

F4.4 Szövegfájlok írása és olvasása - a StreamWriter, StreamReader osztályok

A StreamWriter és a StreamReader osztályok minden olyan esetben kényelmesen alkalmazhatók, amikor szöveges adatokat, adatsorokat kívánunk fájlba írni, illetve onnan beolvasni. Mindkét osztály alaphelyzetben Unicode kódolású karaktereket dolgoz fel, azonban a konstruktorukban másféle kódolást is megadhatunk a

System.Text.Encoding osztály statikus tulajdonságait (ASCII, Unicode, UTF8 stb.) használva.

Amennyiben a A StreamWriter és a StreamReader osztályok konstruktoraiban egy állomány elérési útját adjuk meg, az adatfolyam objektum is létre jön a konstruálás során. Stream típusú objektumot átadva a konstruktoroknak ez a lépés elmarad.

StreamWriter(Stream adatfolyam)

StreamWriter(Stream adatfolyam, Encoding kódolás)

StreamWriter(string útvonal)

StreamWriter(string útvonal, bool hozzáírásra)

StreamWriter(string útvonal, bool hozzáírásra, Encoding kódolás)

A hozzáírásra paraméter segítségével megadhatjuk, hogy hozzáírás (true), vagy felülírás (false) történjen létező fájl esetén.

StreamReader(Stream adatfolyam)

StreamReader(Stream adatfolyam, Encoding kódolás)

StreamReader(string útvonal)

StreamReader(string útvonal, Encoding kódolás)

A sikeres konstruálást követően, a létrejövő objektumok csak olvasható BaseStream hivatkozásán keresztül elérhetjük a háttérben használt bájtfolyamot, bár erre a szöveges adatok feldolgozásakor nincs szükségünk. Az író objektumok NewLine tulajdonságával pedig lekérdezhetjük, illetve beállíthatjuk a szövegsorvége karaktereket, melyek alaphelyzetben ”\r\n”. Az olvasó objektumok EndOfStream logikai tulajdonsága true értékkel jelzi, ha az olvasás során az aktuális pozíció elérte az adatfolyam végét.

A műveletek végeztével a Close() metódussal le kell zárni az objektumokat, és a hozzájuk tatozó adatfolyamokat. Írás objektum esetén a lezárást ajánlott a Flush() metódus hívása után elvégezni.

Az írási és az olvasási műveletek elvégzésre egy sor metódus áll a rendelkezésünkre, melyek közül csak a legfontosabbak ismertetésére térünk ki.

Szövegek kiírását a StreamWriter típusú objektum metódusainak segítségével végezhetjük el. A WriteLine() metódusok a Write() metódusokkal ellentétben a szöveg kiírása után a sorvége karaktereket is kiírják – vagyis új sort kezdenek. Mindkét metódust hasonlóan használhatjuk tetszőleges alaptípusú adat, karaktertömb, egy vagy több adat formázott kiírására.

void Write(alaptípus érték)

void Write(char[] tömb)

void Write(char[] tömb, int kezdőindex, int darab)

void Write(string formátum, object arg)

void Write(string formátum, object arg1, object arg2)

void Write(string formátum, object arg1, object arg2, object arg3)

void Write(string formátum, params object[] arg)

void WriteLine()

void WriteLine (alaptípus érték)

void WriteLine (char[] tömb)

void WriteLine (char[] tömb, int kezdőindex, int darab)

void WriteLine (string formátum, object arg)

void WriteLine (string formátum, object arg1, object arg2)

void WriteLine (string formátum, object arg1, object arg2, object arg3)

void WriteLine (string formátum, params object[] arg)

Az olvasási műveletekhez a StreamReader típusú objektumok metódusait hívjuk:

int Peek()

A soron következő karakter lekérdezése a fájlpozíció léptetése nélkül. A visszaadott-1 érték jelzi a művelet sikertelenségét.

int Read()

A soron következő karakter beolvasása az aktuális pozíciótól. A visszaadott-1 érték jelzi a művelet sikertelenségét.

int Read(char[] tömb, int kezdőindex, int darab)

Legfeljebb darab számú karaktert olvas az aktuális pozíciótól, és a tömbbe másolja a megadott indexű elemtől. A visszatérési érték a sikeresen beolvasott karakterek száma.

string ReadLine()

Szövegsor beolvasása az aktuális pozíciótól. Az adatfolyam vége utáni olvasáskor null értékkel tér vissza.

string ReadToEnd()

Szöveg beolvasása az aktuális pozíciótól az adatfolyam végéig. Az adatfolyam vége utáni olvasáskor üres sztringgel ("") tér vissza.

Nézzünk néhány alapvető programrészletet a szövegfájlok kezelésére!

```
// Új szöveges fájl létrehozása és írása StreamWriter sw = new
    StreamWriter("C:\\Munka\\Szoveg.txt"); try { sw.Write("C# ");
    sw.WriteLine("progarmozás."); sw.WriteLine(2015); } finally { sw.Flush(); sw.Close();
    }
```



```
// Szöveges állomány olvasása a fájl végéig StreamReader sr = new
    StreamReader("C:\\Munka\\Szoveg.txt"); string s = ""; try { while (!sr.EndOfStream) //
    amíg el nem érjük a fájl végét { s = sr.ReadLine(); listBox1.Items.Add(s); } } finally {
    sr.Close(); }
```

Készítsünk programot, amely gombnyomásra véletlen darabszámú, véletlen síkbeli koordináta-párokkal tölt fel az Utvonal.txt szövegfájlt. Például, az ábra bal oldalán látható útvonalnak a jobb oldali állomány felel meg.

(110, 36)ADBCE(12, 23)(198, 109)(52, 101)(77, 146)

512,23110,36198,10952,10177,146

Utvonal.txt

Egy másik gomb megnyomásakor a program a szövegfájl feldolgozásával határozza meg az útvonal hosszát!

using System.IO; // . . . private void btnFeltolt_Click(object sender, EventArgs e) { Random rnd = new Random(); StreamWriter sw = new StreamWriter(@"C:\Munka\Utvonal.txt"); int n = rnd.Next(23); // az adatok száma sw.WriteLine(n); int x, y; for (int i = 0; i < n; i++) { x = rnd.Next(2015); y = rnd.Next(2015); sw.WriteLine("{0},{1}", x, y); } sw.Flush(); sw.Close(); } private void btnFeldolgoz_Click(object sender, EventArgs e) { double uthossz = 0; StreamReader sr = new StreamReader("C:\\Munka\\Utvonal.txt"); int n = Convert.ToInt32(sr.ReadLine()); // az adatok száma string sor; string[] sxy; double xkezd = 0, ykezd = 0, xveg = 0, yveg = 0;

for (int i = 0; i < n; i++) { sor = sr.ReadLine(); sxy = sor.Split(','); // a szövegsor feldarabolása a vesszőnél xveg = Convert.ToDouble(sxy[0]); yveg = Convert.ToDouble(sxy[1]); if (i != 0) // a 2. ponttól számolunk { uthossz += Math.Sqrt(Math.Pow(xveg - xkezd, 2) + Math.Pow(yveg - ykezd, 2)); } xkezd = xveg; ykezd = yveg; } sr.Close(); MessageBox.Show("Úthossz: " + uthossz.ToString("####.00")); }

F4.5 Bináris fájlok írása és olvasása - a BinaryWriter, BinaryReader osztályok

A bináris fájlok legfontosabb jellemzője, hogy az adatokat úgy tárolja, mint ahogy a változók a memóriában. Bináris fájl írásakor a memóriában tárolt adatokhoz tartozó bájtsorozat változatlan formában íródik ki, és olvasáskor ez töltődik be. Mivel a típusonként a lefoglalt bájtok száma és értelmezése igencsak eltérő lehet, csak akkor tudjuk feldolgozni egy bináris állomány tartalmát, ha pontosan ismerjük az adatok típusát és kiírásának sorrendjét.

Ha a konstruktorokra tekintünk, azonnal szembeötlik, hogy nem adhatjuk meg közvetlenül egy fájl elérési útvonalát, hanem egy megfelelő eléréssel megnyitott adatfolyam-objektumot kell használnunk.

BinaryWriter(Stream outputAdatfolyam)

BinaryWriter(Stream outputAdatfolyam, Encoding kódolás)

BinaryReader(Stream inputAdatfolyam)

BinaryReader(Stream inputAdatfolyam, Encoding kódolás)

Erre a célra a FileStream típusú objektumok (lásd F.4.3.) tökéletesen megfelelnek, azonban sokkal egyszerűbb megoldásokhoz jutunk a File osztály Create(), Open(), OpenRead(), OpenWrite(), statikus metódusaival.

File.Create(string útvonal)

File.Open(string útvonal, FileMode nyitási_mód)

File.Open(string útvonal,FileMode nyitási_mód,FileAccess elérés,FileShare megosztás)

File.OpenRead(string útvonal)

File.OpenWrite(string útvonal)

Az adatátviteli műveleteken túlmenően a bináris írás és olvasás folyamatát egyaránt a Close() metódussal kell lezárnunk. A Close() metódus hívásakor a háttérben álló adatfolyam is automatikusan bezáródik. A bináris író objektum lehetővé teszi a fájlpozíció kiválasztását Seek(), valamint az ideiglenes tárolók tartalmának fájlba mentését Flush().

A bináris írás elvégzését a BinaryWriter osztály több metódussal is támogatja. Az alaptípusú adatok mellett bájtokat vagy karaktereket tároló tömböket, illetve azok egy részét is a fájlba írhatjuk:

void Write(alaptípus érték)

void Write(byte [] bájtTömb)

void Write(char [] karakterTömb)

void Write(byte [] bájtTömb, int kezdőindex, int darab)

void Write(char [] karakterTömb, int kezdőindex, int darab)

127

A BinaryReader osztály olvasó metódusainak többsége visszatérési értékként adja meg a kiolvasott adato(ka)t:

sbyte ReadSByte()

byte ReadByte()

byte [] ReadBytes(int darab)

short ReadInt16()

int ReadInt32()

long ReadInt64()

ushort ReadUInt16()

uint ReadUInt32()

ulong ReadUInt64()

bool ReadBoolean()

char ReadChar()

char [] ReadChars(int darab)

float ReadSingle()

double ReadDouble()

decimal ReadDecimal()

string ReadString()

A fentiektől eltérő módon működnek az alábbi metódusok:

int PeekChar()

A soron következő karakter lekérdezése a fájlpozíció léptetése nélkül. A visszaadott-1 érték jelzi a művelet sikertelenségét.

int Read()

A soron következő karakter beolvasása az aktuális pozíciótól. A visszaadott-1 érték jelzi a művelet sikertelenségét.

int Read(char[] karakterTömb, int kezdőindex, int darab)

Legfeljebb darab számú karaktert olvas az aktuális pozíciótól, és a tömbbe másolja a megadott indexű elemtől. A visszatérési érték a sikeresen beolvasott karakterek száma.

int Read(byte[] bájtTömb, int kezdőindex, int darab)

Legfeljebb darab számú bájtot olvas az aktuális pozíciótól, és a tömbbe másolja a megadott indexű elemtől. A visszatérési érték a sikeresen beolvasott bájtok száma.

Példaként oldjuk meg az előző rész útvonalszámító feladatát bináris adatok tárolásával (Utvonal.dat)! Az állomány logikai szerkezete megegyezik a szövegfájlnál használttal, azonban az egészeket tároló 4 bájtok folytonosan következnek egymás után.

(110, 36)

A

D

B

C

E

(12, 23)

(198, 109)

(52, 101)

(77, 146)

050000000c000000170000006e00000024000000c60000006d0000034000000650000004d00000092000000

Utvonal.dat

(Az ábrán a bájtok hexadecimális megjelenítése és 4 bájtos csoportosítása csak a jobb megértést segítik.)

using System.IO; // . . . private void btnBinFeltolt_Click(object sender, EventArgs e) { Random rnd = new Random(); FileStream fs = new FileStream(@"C:\Munka\Utvonal.dat", FileMode.Create); BinaryWriter bw = new BinaryWriter(fs); int n = rnd.Next(23); // az adatok száma bw.Write(n); int x, y;

for (int i = 0; i < n; i++) { x = rnd.Next(2015); y = rnd.Next(2015); bw.Write(x); bw.Write(y); } bw.Flush(); bw.Close(); } private void btnBinfeldolgoz_Click(object sender, EventArgs e) { double uthossz = 0; FileStream fs = File.OpenRead("C:\\Munka\\Utvonal.dat"); BinaryReader br = new BinaryReader(fs); int n = br.ReadInt32(); // az adatok száma double xkezd = 0, ykezd = 0, xveg = 0, yveg = 0; for (int i = 0; i < n; i++) { xveg = br.ReadInt32(); yveg = br.ReadInt32(); if (i != 0) // a 2. ponttól számolunk { uthossz += Math.Sqrt(Math.Pow(xveg - xkezd, 2) + Math.Pow(yveg - ykezd, 2)); } xkezd = xveg; ykezd = yveg; } br.Close(); MessageBox.Show("Úthossz: " + uthossz.ToString("####.00")); }

A fenti példákból az egyszerűség kedvéért kimaradt a kivételek feldolgozása, bár a metódusok többsége valamilyen kivétellel jelzi működésének sikertelenségét. Álljon itt a btnBinfeldolgoz_Click() eseménykezelő kivételkezeléssel kiegészített változata, amely gyakorlatilag minden fájlkezelő metódusban alkalmazható:

private void btnBinfeldolgozEx_Click(object sender, EventArgs e) { double uthossz = 0; BinaryReader br; try { FileStream fs = File.OpenRead("C:\\Munka\\Utvonal.dat"); br = new BinaryReader(fs); } catch (IOException ex) { MessageBox.Show("Fájlnyitási hiba: " + ex.Message); return; } try { int n = br.ReadInt32(); // az adatok száma double xkezd = 0, ykezd = 0, xveg = 0, yveg = 0; for (int i = 0; i < n; i++) { xveg = br.ReadInt32(); yveg = br.ReadInt32(); if (i != 0) // a 2. ponttól számolunk { uthossz += Math.Sqrt(Math.Pow(xveg - xkezd, 2) + Math.Pow(yveg - ykezd, 2)); } xkezd = xveg; ykezd = yveg; } }

```
catch (IOException ex) { MessageBox.Show("Fájl I/O hiba: " + ex.Message); return; } finally
    { br.Close(); } MessageBox.Show("Úthossz: " + uthossz.ToString("####.00")); }
```

Külön try-catch blokkba kerültek a megnyitási műveletek, míg a fájl feldolgozását egy try-catch-finally blokk tartalmazza. A finally részbe helyezett Close() metódus fájlhiba esetén is elvégzi a lezárást.

Végezetül készítsünk programot, amely megkeresi az Utvonal.dat bináris állományban tárolt legnagyobb értéket. A megoldásban a feldolgozást a fájl végének eléréséig végezzük!

```
private void btnLegnagyobb_Click(object sender, EventArgs e) { int legnagyobb =
    int.MinValue, adat = 0; BinaryReader br = new
    BinaryReader(File.OpenRead("C:\\Munka\\Utvonal.dat")); while
    (br.BaseStream.Position < br.BaseStream.Length) { adat = br.ReadInt32(); if (adat >
    legnagyobb) legnagyobb = adat; } br.Close(); MessageBox.Show("A legnagyobb adat: "
    + legnagyobb); }
```

F4.6 A fájlrendszer kezelésének legfontosabb lépései – a Directory, File és Path osztályok

A fejezet végén- a teljesség igénye nélkül - néhány olyan megoldást szedtünk csokorba, amelyek jól használhatók fájlkezelő alkalmazások készítésekor. Mint említettük, a Directory, File és Path osztályok statikus metódusai objektumok létrehozása nélkül hívhatók – természetesen a using System.IO; programsor megadását követően.

Valamely könyvtár vagy fájl létezésének vizsgálata:

```
if (Directory.Exists("C:\\Munka")) MessageBox.Show("A mappa létezik"); else
    MessageBox.Show("A mappa nem létezik");
```

```
if (File.Exists("C:\\Munka\\Utvonal.txt")) MessageBox.Show("A fájl létezik"); else
    MessageBox.Show("A fájl nem létezik");
```

Adott mappában található állományok neveinek lekérdezése egy sztring tömbbe:

```
string[] mindenFile = Directory.GetFiles(@"C:\Munka"); // vagy
```



```
string[] mindenFile = Directory.GetFiles(@"C:\Munka", "*.*");
```

Adott mappában található alkönyvtárak neveinek lekérdezése egy sztring tömbbe:

```
string[] alkonyvtarak = Directory.GetDirectories(@"C:\Munka");
```

Üres mappa törlése, teljes könyvtárág és állomány törlése:

```
Directory.Delete("C:\\Munka\\Munka1"); // üres mappa
    Directory.Delete("C:\\Munka\\Munka2", true); // teljes könyvárág
    File.Delete("C:\\Munka\\Utvonal.dat"); // fájl törlése
```

Könyvtárstruktúra létrehozása

Létrehozza a megadott útvonal minden mappáját, amennyiben azok nem léteznek.

```
Directory.CreateDirectory("C:\\Munka\\Aldir1\\Aldir11");
```

Könyvtárak és állományok áthelyezése (átnevezése)

```
// azonos meghajtón levő könyvtár áthelyezése a tartalmával együtt string forrasDir =
    @"C:\Munka"; string celDir = @"C:\Temp\Mentes"; Directory.Move(forrasDir,
    celDir); // állomány áthelyezése string forrasFajl = @"C:\Munka\Adatok.txt"; string
    celFajl = @"D:\Mentes\MentettAdatok.txt"; File.Move(forrasFajl, celFajl);
```

Adott fájl (eredeti) tartalmának áthelyezése egy másik állományba (hova)

A művelet előtt a másik állományról mentést készít, és a művelet végeztével az eredeti fájlt törli. A fájloknak azonos meghajtón kell elhelyezkedniük.

string fajlEredeti = @"C:\Munka\Eredeti.txt"; string fajlHova = @"C:\Munka\Hova.txt"; string fajlHovalMentes = @"C:\Munka\Hova.bak"; File.Replace(fajlEredeti, fajlHova, fajlHovalMentes); File.Replace(fajlEredeti, fajlHova, null); // nem készül mentésfájl

Állományok másolása

string forrasFajl = @"C:\Munka\Adatok.txt";

string celFajl1 = @"D:\Mentes\MentettAdatok.txt"; string celFajl2 = @"D:\Mentes\Adatok.txt"; File.Copy(forrasFajl, celFajl1, true); // felülírással File.Copy(forrasFajl, celFajl2); // nem írja felül

Teljes elérési úttal megadott fájlnév összetevőkre bontása:

string utvonal = Application.ExecutablePath; // a futó .EXE program elérési útvonala // C:\Munka\Program.exe string mappa = Path.GetDirectoryName(utvonal); // C:\Munka string kiterjesztes = Path.GetExtension(utvonal); // .exe string fajlnevKiterjesztessel = Path.GetFileName(utvonal); // Program.exe string fajlnevKiterjesztesNelkul = Path.GetFileNameWithoutExtension(utvonal); // Program string teljesUtvonal = Path.GetFullPath("."); // C:\Munka

Egyszerűsített fájlkezelés a File osztály statikus metódusaival

Ha a feldolgozni kívánt állomány kevés adatot tartalmaz, akkor a szükséges műveleteket gyorsabban elvégezhetjük, ha előtte a fájl teljes tartalmát betöltjük a memóriába, majd a munka végeztével visszaírjuk azt a lemezre. A műveletek többségének van olyan párja, ahol az alapértelmezett Unicode karakterkódolás helyett más kódolást is előírhatunk.

byte[] ReadAllBytes(string útvonal)

A bináris állomány teljes tartalmát egy byte tömbben adja vissza. A tömböt a metódus hozza létre.

string[] ReadAllLines(string útvonal)

string[] ReadAllLines(string útvonal, Encoding kódolás)

A szöveges állomány sorait egy string tömbben adja vissza. A tömböt a metódusok készítik el.

string ReadAllText(string útvonal)

string ReadAllText(string útvonal, Encoding kódolás)

A szöveges állomány teljes tartalmát egy sztringben adja vissza.

void WriteAllBytes(string útvonal, byte[] bájtTömb)

A byte tömb tartalmát a megadott bináris állományba írja.

void WriteAllLines(string útvonal, string[] karakterTömb)

void WriteAllLines(string útvonal, string[] karakterTömb, Encoding kódolás)

A karakter tömb tartalmát a megadott szövegfájlba írja.

void WriteAllText(string útvonal, string sztring)

void WriteAllText(string útvonal, string sztring, Encoding kódolás)

A sztringet a megadott szöveges állományba írja.

void AppendAllText(string útvonal, string sztring)

void AppendAllText(string útvonal, string sztring, Encoding kódolás)

A megadott sztring tartalmát szöveges állományhoz fűzi.

Készítsünk programot, amely gombnyomásra szöveges állományokból feltölt egy listaablakot és egy többsoros szövegmezőt! Egy másik gomb megnyomásakor pedig szövegfájlokba menti a vezérlők tartalmát.

private void btnBeolvas_Click(object sender, EventArgs e) { string[] elemek = File.ReadAllLines(@"C:\Munka\Elemek.txt"); listBox1.Items.AddRange(elemek); textBox1.Text = File.ReadAllText(@"C:\Munka\Szoveg.txt"); } private void btnKiir_Click(object sender, EventArgs e) { string[] elemek = new string[listBox1.Items.Count]; listBox1.Items.CopyTo(elemek, 0); File.WriteAllLines(@"C:\Munka\Elemek.txt", elemek); File.WriteAllText(@"C:\Munka\Szoveg.txt", textBox1.Text); }

F5. A grafika programozásának alapjai

A .NET keretrendszer vizuális komponenseit használva, különböző, grafikus felületű alkalmazásokat készíthetünk. A programunk megjelenését azonban képek betöltésével, illetve grafikák programból történő megjelenítésével egyedivé tehetjük, amely így jobban megfelelhet a felhasználói igényeknek.

Napjainkban több lehetőség közül is választhatunk (Windows Forms Application, WPF Application). Ebben a fejezetben az ún. hagyományos Windows (Forms) grafika lehetőségeivel ismerkedünk meg, amely legelső .NET rendszer megjelenésével (2002) vált elérhetővé a Windows rendszerben.

F5.1 Bevezetés

A GDI (Graphics Device Interface) a Windows hagyományos 2D-s grafika-programozási felülete, amelyet GDI+-ban alapvetően átdolgoztak. A GDI+ legfontosabb jellemzője, hogy objektum-orientált lett. Osztályok segítik a kétdimenziós alakzatok, a betűkészletek, a színek és a képek kezelését. A szokásos megoldásokon túlmenően egy sor újdonság támogatja a látványosabb, haladó szintű grafikák késztését, amely megoldásokat külön fejezetben részletezzük.

A GDI+ grafika használatához szükséges osztályok (és struktúrák) névterekbe csoportosítva állnak a rendelkezésünkre. A System.Drawing névtér automatikusan beépül a Windows Forms Application projektbe, azonban a szükséges alnévterek megadásáról magunknak kell gondoskodni.

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Windows.Forms;

System.Drawing

Ez a névtér tartalmazza a legfontosabb GDI+ osztályokat (Graphics, Pen, Brush, Font, Bitmap, Image stb.) és struktúrákat (Color, Point, Rectangle, Size).

133

System.Drawing.Drawing2D

A haladó szintű kétdimenziós grafika osztályait tárolja. Itt találjuk a Brush osztályból származtatott speciális kitöltések osztályait (HatchBrush, LinearGradientBrush, stb.), egyenesek és görbék sorozatát leíró GraphicsPath osztályt, valamint geometriai transzformációk Matrix osztályát. Ugyancsak itt találunk a vonalakkal és a kitöltésekkel kapcsolatos felsorolásokat (PenType, LineCap, HatchStyle).

System.Drawing.Design

A névtér különböző szerkesztőkkel (BitmapEditor, ColorEditor, FontEditor stb.) bővíti a felhasználói felület logikai és grafikai lehetőségeit a fejlesztési fázisban.

System.Drawing.Printing

A névtér nyomtatási szolgáltatásokat biztosít a Windows Forms alkalmazások számára.

System.Drawing.Imaging

A GDI+ haladó szintű képkezelési megoldásaihoz ad osztályokat (ImageFormat, BitmapData, Metafile stb.) a névtér.

A GDI+ leggyakrabban használt elemeit a System.Drawing és a System.Drawing.Drawing2D névterek tartalmazzák. Az alábbiakban felsoroltunk néhány alapvető System.Drawing osztályt és struktúrát.

Osztály

Rövid ismertető

Bitmap, Icon, Image, SystemIcons

bitképek, ikonok osztályai és a grafikus képek absztrakt alaposztálya,

Brush, Brushes, SystemBrushes

absztrakt ecset alaposztály más ecsetek származtatásához, valamint kész ecsetek a különböző alakzatok kitöltésekhez,

Font, FontFamily, SystemFonts

a szövegtulajdonságok beállítását és lekérdezését (méret, stílus, betűtípus stb.) segítő osztályok,

Graphics

a GDI+ grafikus felületének objektum-modellje,

Pen, Pens, SystemPens

a toll valamint kész tollak osztályai szakaszok (lines) és görbék (curves) rajzolá-sához,

Region

a teljes rajzterület adott téglalap alakú részét leíró osztály,

SolidBrush

egyszínű kitöltési mintával rendelkező ecset,

TextureBrush

adott képpel való kitöltéshez használható ecset,

Color, SystemColors

(ARGB) színek elérésének és kezelésének támogatása,

Point, PointF

síkbeli pont megadása (X, Y) int vagy float (F) típusú koordinátákkal,

Rectangle, RectangleF

téglalap definiálása a bal felső sarok koordinátáinak, valamint a szélesség és magasság adatainak megadásával (X, Y és Width, Height),

Size, SizeF

méret (Width, Height) megadása téglalap alakú területek esetén.

F5.2 A Graphics osztály

A C# alkalmazások ablaka (form), illetve az ablakban elhelyezett vezérlők mindegyike saját grafikus felülettel rendelkezik, melyet a Graphics típusú objektumok segítségével érhetjük el. Mindegyik grafikus felülethez saját koordináta-rendszer és koordináta-egységek (PageUnit) tartoznak. Alaphelyzetben a mértékegység képpont (GraphicsUnit.Pixel).

134

A koordináta-rendszer origója a vezérlők bal felső sarkában helyezkedik el, az x-tengely jobbra, az y-tengely pedig lefelé mutat. Form esetén a koordináta-rendszer a keret nélküli munkaterülethez illeszkedik, melynek méreteit a ClientSize tulajdonság tartalmazza.

x-tengelyy-tengely(x,y)

A vezérlők és form grafikus felületének objektumát többféleképpen is elérhetjük. Amennyiben olyan rajzot szeretnénk készíteni, ami minden frissítéskor automatikusan megjelenik, akkor a Paint eseményt kezelő metódusban kell a szükséges utasításokat elhelyezni. (A fenti ablak vezérlői közül a Form, a Panel, a PictureBox és a Button rendelkezik Paint eseménnyel.) A Paint eseménykezelő metódusa paraméterben kapja a grafikus felület objektumának hivatkozását:

private void Form1_Paint(object sender, PaintEventArgs e) { Graphics g = e.Graphics; g.DrawLine(Pens.Green, 0, 0, 120, 230); // zöld vonalat húz }

135

Minden esetben használható megoldást ad a vezérlők és a form CreateGraphics() metódusának hívása:

private void button1_Click(object sender, EventArgs e) { Graphics gForm = this.CreateGraphics(); // a form grafikus felülete gForm.DrawLine(Pens.Green, 0, 0, 120, 230); Graphics gPBox = pictureBox1.CreateGraphics(); // a képmező grafikus felülete gPBox.DrawLine(Pens.Green, 0, 0, 120, 230); }

Az ilyen módon elkészített grafika a vezérlő/form újrarajzolásakor eltűnik, és – esetünkben – csak a gomb ismételt megnyomásakor jelenik meg újra.

A grafikus felületek Graphics típusú objektumai képezik a grafika programozásának alapját. A Graphics osztály tulajdonságai és metódusai a vonalas és festett (kitöltött) alakzatok rajzolásán túlmenően, szövegek és képek programozott megjelenítését is segítik.

Az azonos műveletet végző túlterhelt metódusok lehetővé teszik, hogy mindig a legmegfelelőbb módon hívjuk azokat. Az egyszerű int vagy float típusú koordináták, illetve méretadatok mellett, Point, Rectangle és Size típusú adatok is szerepelhetnek az argumentumlistákon. A hívások további argumentumai a fenti névterekben tárolt ún. grafikus objektumok közül kerülnek ki. Az alábbiakban megismerkedünk a legfontosabb grafikus objektumokkal:

Color

Más grafikus objektumok színének beállításához használjuk. GDI+-ban a hagyományos RGB színek áttetszősége (alfa blend) is megadható (ARGB színek).

Pen

Vonalas alakzatok: egyenes szakasz, téglalap, ellipszis, ellipszisív stb.) rajzolásához használt toll.

Brush

Zárt felületek adott mintával, színekkel vagy bitképpel való kitöltését segítő ecset

Font

Szövegek megjelenítéséhez alkalmazott beállítások összessége, betűtípus.

Image

A képek kezeléséhez metódusokat és tulajdonságokat biztosító alaposztály.

F5.3 A grafikus objektumok

Mielőtt C# programból grafikát hoznánk létre, szükséges megismerkednünk a grafikus objektumok készítésének és használatának fogásaival.

F5.3.1 Színek – a Color struktúra

A színekkel kapcsolatos információkat és műveleteket a Color struktúra tartalmazza. Ugyanezt a struktúrát használtuk korábban a vezérlők (háttér- és előtér-) színének beállításakor. A Color struktúra statikus tulajdonságaival 141, névvel ellátott, előre definiált szín közül választhatunk, melyek teljes listája az F5.5. részben található.

A Color típus a színértékeket 32-biten tárolja, melyek bájtjai az alfa (Alpha), a piros (Red), a zöld (Green) és a kék (Blue) összetevők értékét (0-tól 255-ig) jelölik. A 0 azt jelenti, hogy hiányzik az adott színösszetevő, míg 255-nél maximálisan jelen van a színben. A 0-ás alfa érték esetén a szín teljesen áttetsző, míg 255 esetén teljesen átlátszatlan. (Az előre definiált színek mindegyike átlátszatlan.) Az egyes összetevők bájtértékét az A, R, G és B tulajdonságok szolgáltatják. A színek átlátszóságát az alábbi ábra szemlélteti:

Tetszőleges RGB és ARGB színt kikeverhetünk a FromArgb() statikus metódus hívásával. A metódus túlterhelt változatai különböző bemenő adatokkal rendelkeznek, azonban mindegyik egy Color struktúrával tér vissza.

Color FromArgb(int argb)

Szín előállítása egy 32-bites egész értékből.

Color FromArgb(int alfa, Color alapszín)

Az alapszín áttetszőségének (alfa) módosítása.

Color FromArgb(int piros, int zöld, int kék)

RGB szín készítése. Ekkor az alfa értéke 255, vagyis a szín nem átlátszó.

Color FromArgb(int alfa, int piros, int zöld, int kék)

ARGB szín előállítása.

Az alábbi példában az egyik gomb megnyomásakor egy véletlen RGB színt adunk meg a form háttérszínének, míg egy másik gomb minden megnyomásakor a háttérszín összetevőit négyötöd részére csökkentjük!

private void button1_Click(object sender, EventArgs e) { Random rnd = new Random(); Color szin = Color.FromArgb(rnd.Next(256), rnd.Next(256), rnd.Next(256)); this.BackColor = szin; } private void button2_Click(object sender, EventArgs e) { byte r = BackColor.R; byte g = BackColor.G; byte b = BackColor.B; this.BackColor = Color.FromArgb(4 * r / 5, 4 * g / 5, 4 * b / 5); }

Felhívjuk a figyelmet arra, hogy Color típusú objektumot interaktívan is létrehozhatunk, az F3.4.3. fejezetben bemutatott ColorDialog párbeszédablak felhasználásával.

F5.3.2 Tollak – a Pen osztály

A tollat adott színű, stílusú és vastagságú vonalak rajzolására használjuk. A tollakat a Pen osztály konstruktorainak hívásával hozzuk létre, azonban a jellemzőket tulajdonságok segítségével módosíthatjuk is.

Pen(Color szín)

Pen(Color szín, float vastagság)

Pen(Brush ecset)

Pen(Brush ecset, float vastagság)

Létrehozáskor megadhatjuk a toll színét és vastagságát, melynek alapértéke 1. A konstruktorhívásban szereplő ecset a vonalat megjelenítő elemi téglalapok kifestését definiálja. Az elkészített toll jellemzőit (Brush, Color, Width) tulajdonságok segítségével később is módosíthatjuk.

Az előredefiniált színekhez a Pens osztály statikus tulajdonságai 1 vastagságú tollakat definiálnak.

Húzzunk különböző (1, 2, .., 10) vastagságú, lila színű, függőleges vonalakat az ablakba. (A megoldásban a grafikus felületet DrawLine() metódusát ciklusban hívjuk.)

private void button1_Click(object sender, EventArgs e) { Graphics g = this.CreateGraphics(); int ablakX = ClientSize.Width; int ablakY = ClientSize.Height; float dx = ablakX / 11F, x = 0; Pen toll = new Pen(Color.Magenta); for (int v = 1; v <= 10; v++) { toll.Width = v; x += dx; g.DrawLine(toll, x, 10, x, ablakY - 30); } }

F5.3.3 Ecsetek – a Brush osztály

A Brush egy absztrakt alaposztály, ezért csak a leszármazott osztályaival (SolidBrush, TextureBrush, RectangleGradientBrush és LinearGradientBrush) készíthetünk objektumokat. Ebben a részben csak az elsővel foglalkozunk részletesebben. Az elkésztett ecseteket grafikus alakzatok (téglalapok, ellipszisek, poligonok stb.) kitöltéséhez használjuk.

A SolidBrush típussal egyszínű ecseteket készíthetünk, melyek színét a Color tulajdonságon át később módosíthatjuk:

SolidBrush(Color szín)

Az előredefiniált színekhez a Brushes osztály 141, névvel ellátott ecsetet biztost számunkra.

A formon való egérkattintás hatására rajzoljunk magyar zászlót a programunk ablakába! (A kifestett téglalap megjelenítéséhez használjuk a grafikus felület FillRectangle() metódusát!)

```
private void Form1_Click(object sender, EventArgs e) { Graphics g = this.CreateGraphics(); int ablakX = ClientSize.Width; int ablakY = ClientSize.Height; int m = ablakY / 3; g.FillRectangle(Brushes.Red, 0, 0, ablakX, m); g.FillRectangle(Brushes.White, 0, m, ablakX, 2 * m); g.FillRectangle(Brushes.Green, 0, 2 * m, ablakX, ablakY); }
```


F5.3.4 Betűtípusok – a Font és FontFamily osztályok

Szövegek megjelenítéséhez betűtípusokat kell létrehoznunk a Font osztály felhasználásával. A nagyszámú konstruktorban a legkülönbözőbb paraméterek szerepelnek:

- általában első helyen kell megadni egy már létező betűtípus objektumot vagy egy betűtípuscsalád (FontFamily) objektumát vagy nevét,
- az új betűtípus stílusa, amely a FontStyle felsorolás elemeinek bitenkénti logikai műveletekkel történő összekapcsolásával alakítható ki,
- a betűtípus mérete, melynek mértékegysége a pont (1/72”), vagy ha szerepel a paraméterlistán GraphicsUnit típusú elem, akkor annak értéke határozza meg a mértékegységet.

A betűtípuscsalád (font family) alapvető hatással van a betűtípus (font) megjelenésére. Általánosan elterjedt családok az Arial, a Verdana, a Courier New, a Times New Roman és a Sans Serif. Egy betűcsaládot csak akkor használhatunk betűtípus létrehozásához, ha előtte telepítettük a számítógépünkre. (Megjegyezzük, hogy a fent említett betűtípuscsaládokat minden Windows rendszer tartalmazza.)

A teljesség igénye nélkül nézzünk néhány betűtípus-létrehozást! Leggyakrabban azt a formát használjuk, ahol megadjuk a betűtípuscsalád nevét és a betűk pontokban kifejezett méretét:

```
Font betűtípus = new Font("Times New Roman", 23);
```

Amennyiben a FontFamily objektumot is elkészítjük, hozzáférhetünk a betűtípus-metrikákhoz (GetCellAscent(), GetLineSpacing() stb.)

```
FontFamily család = new FontFamily("Arial");

Font betűtípus = new Font(család, 23);

int sortáv = család.GetLineSpacing(FontStyle.Regular);
```

A következő konstruktorhívással vastagított és dőlt stílusú betűtípust hozunk létre. A FontStyle felsorolás lehetséges értékei: Regular, Bold, Italic, Underline, Strikeout.

```
Font betűtípus = new Font("Consolas", 32, FontStyle.Bold | FontStyle.Italic);
```

Hasznos megoldás, amikor egy meglévő betűtípusból kiindulva hozzuk létre az újat, a stílus módosításával:

```
Font betűtípus = new Font(this.Font, FontStyle.Strikeout);
```

Utolsó példaként készítsünk 1 centiméter méretű betűket tartalmazó betűtípust!

```
Font betűtípus = new Font("Calibri", 10, GraphicsUnit.Millimeter);
```

vagy

```
Font betűtípus = new Font("Calibri", 10, FontStyle.Bold, GraphicsUnit.Millimeter);
```

A konstruktorban megadott jellemzők a Font tulajdonságainak segítségével bármikor lekérdezhetők, azonban nem módosíthatók:

Bold

Igaz értékkel jelzi, ha a betűtípus stílusa vastagított.

FontFamily

A betűtípus betűtípuscsaládját szolgáltatja.

Height

Képpontokban megadja a betűtípushoz tartozó szövegsor-magasságot

Italic

Igaz értékkel jelzi, ha a betűtípus stílusa dőlt.

Name

A betűtípus arculatának nevét szolgáltatja

Size

Megadja a betűtípus méretét a unit tulajdonságban tárolt mértékegységekben.

SizeInPoints

Megadja a betűtípus méretét pont mértékegységekben.

Strikeout

Igaz értékkel jelzi, ha a betűtípus stílusa áthúzott.

Style

Megadja a betűtípus stílusát.

Underline

Igaz értékkel jelzi, ha a betűtípus stílusa aláhúzott.

Unit

Megadja a betűtípus mértékegységét a GraphicsUnit felsorolás elemeként.

Az elkészített betűtípust valamely vezérlő Font tulajdonságához rendelhetjük, illetve felhasználhatjuk grafikus metódus segítségével történő szöveg-megjelenítéshez. Betűtípus objektum interaktív létrehozását segíti az F3.4.3. fejezetben tárgyalt FontDialog általános párbeszédablak.

F5.3.5 Képek – az Image és a Bitmap osztályok

A grafikus képformátumokat két csoportba sorolhatjuk. Raszteres kép esetén a kép pontjainak színét tároljuk (pixel). Ezzel szemben a vektoros képek a rajzolás lépéseiből – vonalhúzás, kifestés stb. – épülnek fel. .NET rendszerben a raszteres képeket a Bitmap osztály valósítja meg, míg a vektoros képeket a MetaFile osztály írja le. Mindkét osztály őse az absztrakt Image osztály, melynek tulajdonságai és metódusai nagyban egyszerűsítik a képfeldolgozás lépéseit.

A Windows által támogatott formátumú raszteres kép (Bmp, Gif, Jpeg, Png, Tiff) betöltésekor mindig Bitmap típusú objektum jön létre, míg a vektoros adatokat (Emf, Wmf) MetaFile típusú objektum tárolja. Mivel mindkét osztály közös őse az Image, így gyakran Image típusú referenciával hivatkozunk az objektumokra. Mivel ebben a

részben csak a raszteres képekkel foglalkozunk, a képkezelést segítő megoldásokat a Bitmap osztály segítségével fogjuk csokorba.

Külön kategóriát képez az ikonok Icon osztálya. Az ikon olyan kisméretű bitkép, amely átlátszó háttérrel rendelkezik, és amelynek méretét a Windows rendszer határozza meg. Az ikonokat .ICO kiterjesztésű fájlok tárolják.

Állományban tárolt képek betöltése bitkép objektumba

A Bitmap osztály konstruktorai és az Image osztály néhány statikus metódusa egyaránt lehetővé teszik, hogy egy képfájl tartalmát bitkép objektumba töltsük. A példákban a képek megjelenítéséhez a PictureBox Image tulajdonságát, és a Form BackgroundImage tulajdonságát használjuk. (Megjegyezzük, hogy a grafikus vezérlők többsége rendelkezik e tulajdonságok valamelyikével.)

private void button1_Click(object sender, EventArgs e) { Bitmap kép1 = new Bitmap(@"C:\Munka\Cica.jpg"); pictureBox1.Image = kép1; pictureBox1.Width = pictureBox1.Image.Width; pictureBox1.Height = pictureBox1.Image.Height; Bitmap háttérKép = (Bitmap)Image.FromFile(@"C:\Munka\Tél.jpg"); this.BackgroundImage = háttérKép; this.BackgroundImageLayout = ImageLayout.Stretch; System.IO.FileStream fs = System.IO.File.Open(@"C:\Munka\CSharp.png", System.IO.FileMode.Open); Bitmap kép2 = (Bitmap)Image.FromStream(fs); fs.Close(); pictureBox2.Image = kép2; pictureBox2.SetBounds(0, ClientSize.Height - kép2.Height, kép2.Width, kép2.Height); }

A Bitmap osztály további konstruktoraival már meglévő képből készthetünk új, esetleg átméretezett bitképet:

private void button1_Click(object sender, EventArgs e) { Bitmap kép1 = new Bitmap(@"C:\Munka\Cica.jpg"); Bitmap kép2 = new Bitmap(kép1); Bitmap kép3 = new Bitmap(kép1, 230, 320); // 230x320 pixel méretű lesz }

Új bitkép létrehozása

A Bitmap objektumok konstruálásával adott a lehetőség, hogy futás közben hozzunk létre bitképet. Alaphelyzetben 32-bites ARGB képpontokat tároló bitkép jön létre. Amennyiben megadjuk a képpontformátumot a System.Drawing.Imaging.PixelFormat felsorolás elemével, másféle bitképet is készíthetünk. Az alábbi példában egy alapértelmezett formátumú és egy 24-bites TrueColor (RGB) bitképet késztünk.

A példában a Bitmap osztály SetPixel() és GetPixel() metódusainak használatát is bemutatjuk, melyek segítségével a bitképet képpontonként (lassan) érjük el:

void SetPixel(int x, int y, Color color)

Color GetPixel(int x, int y)

private void button1_Click(object sender, EventArgs e) { Bitmap kép1 = new Bitmap(200, 150); // Format32bppArgb for (int x = 0; x < kép1.Width; x++) for (int y = 0; y < kép1.Height; y++) { if (x / 100 == y / 100) kép1.SetPixel(x, y, Color.Blue); else kép1.SetPixel(x, y, Color.Yellow); } pictureBox1.Image = kép1; Bitmap kép2 = new Bitmap(300, 200, System.Drawing.Imaging.PixelFormat.Format24bppRgb); Color pixel; for (int x = 0; x < kép2.Width; x++) for (int y = 0; y < kép2.Height; y++) { if (x < kép1.Width && y < kép1.Height) { pixel = kép1.GetPixel(x, y); kép2.SetPixel(x, y, Color.FromArgb(pixel.G, pixel.B, pixel.R)); } else kép2.SetPixel(x, y, Color.Cyan); } pictureBox2.Image = kép2; }

Műveletek a bitképekkel

A bitkép objektumok RotateFlip() metódusának hívásával, a RotateFlipType felsorolás elemeinek megfelelően a képet elforgathatjuk a kép középpontja körül, a megadott szöggel, az óramutató járásával megegyezően (rotate), és tükrözhetjük a kép megadott (vízszintesY/függőlegesX) tengelyére (flip). Nézzük néhány példát a műveletekre!

RotateNoneFlipNone

Rotate180FlipNone

RotateNoneFlipX

Rotate90FlipNone

Rotate90FlipY

A legtöbb raszteres kép esetén (pl. JPG, PNG) azonos, például 72 vagy 96 képpont/coll (dpi) felbontást alkalmaznak vízszintes (HorizontalResolution) és függőleges (VerticalResolution) irányban egyaránt. Némely képformátum esetén a felbontást mindkét irányban beállíthatjuk a SetResolution() metódus hívásával.

A memóriában tárolt raszteres képet állományba (vagy adatfolyamba) menthetjük a bitkép objektum Save() metódusainak felhasználásával. Amennyiben argumentumként csak a fájlnevet adjuk meg, a kimentett kép formátuma megegyezik a beolvasott kép formátumával. Ha a bitképet programból hozzuk létre, a kiírt kép formátuma PNG lesz.

A Save() metódus további változataiban a mentés formátumát is meghatározhatjuk a System.Drawing.Imaging névtérben definiált ImageFormat felsorolás elemeivel (Bmp, Gif, Icon, Jpeg, Png és Tiff).

private void button1_Click(object sender, EventArgs e) { Bitmap kép = new Bitmap(@"C:\Munka\Golyók.Tif"); kép.Save(@"C:\Munka\Bitkép.Tif"); kép.Save(@"C:\Munka\Bitkép.bmp", System.Drawing.Imaging.ImageFormat.Bmp); System.IO.FileStream fs = System.IO.File.Open(@"C:\Munka\Bitkép.png", System.IO.FileMode.Create); kép.Save(fs, System.Drawing.Imaging.ImageFormat.Png); fs.Close(); }

F5.3.6 A Point, a Size és a Rectangle struktúrák

A Graphics osztály úgynevezett grafikus metódusainak hívását hivatottak segíteni a Point/PointF, a Size/SizeF és Rectangle/RectangleF struktúrák. (Az F betűvel végződő nevű típusok float típusú, míg a többiek int típusú adatokat tárolnak.) Windows rendszerben az esetek többségében téglalap alakú alakzatokkal dolgozunk, melyek síkbeli elhelyezkedését a bal felső sarok koordinátái, méretét pedig a szélessége és a magassága határozza meg.

WidthHeight(X, Y)

Kézenfekvő tehát a pont, a méret és a téglalap struktúrák kialakítása. Nézzünk néhány példát a különböző konstruálási megoldásokra!

private void button1_Click(object sender, EventArgs e) { Point p0 = new Point(); Point p1 = new Point(12, 23); Size m0 = new Size(); Size m1 = new Size(p1); Point p2 = new Point(m1); Size m2 = new Size(400, 300); Rectangle r0 = new Rectangle(); Rectangle r1 = new Rectangle(p1, m2); Rectangle r2 = new Rectangle(12, 23, 400, 300); }

Az adatok tárolásán túlmenően statikus metódusok és tulajdonságok sora teszik hatékonyabbá a programozást. Mindhárom típus esetén definiáltak az== és a != műveletek, míg a pontok és a méretek a méretekkel módosíthatók (+, -). Különböző kerekítési módszerek

(Ceiling(), Round(), Truncate()) választásával térhetünk át a valós adatokat tartalmazó struktúrákról az égeszeket tárolókra.

A nem statikus metódusok közül kiemelést érdemel az IsEmpty(), mellyel könnyen ellenőrizhetjük a tárolt adatok 0 voltát. A Rectangle típus statikus és nem statikus metódusaival két téglalap metszetét (Intersect()) és egyesítését (Union()) is meghatározhatjuk.

Ugyancsak hasznos műveletek a téglalap eltolása adott távolságokkal (Offset()), valamint a téglalap méreteinek megnövelése a megadott méretekkel (Inflate()) méghozzá úgy, hogy a téglalap középpontja helyben marad. (Ez utóbbi művelet során természetesen a bal felső sarok koordinátái is módosulnak). Zárjuk a sort annak vizsgálatával, hogy az adott téglalap magában foglal-e egy pontot, pozíciót vagy egy másik téglalapot (Contains()). Megjegyezzük, hogy a téglalapok bal felső sarkának koordinátái a hagyományos (Left, Top), illetve a jobb alsó sarkának adatai a (Right, Bottom) formában is elérhetők.

142

Az alábbi példában két téglalapot definiálunk, meghatározzuk a metszet és az egyesítés téglalapokat, majd pedig megvizsgáljuk, hogy egy pont benne van-e ezekben.

private void button4_Click(object sender, EventArgs e) { string eredmeny = ""; Rectangle r1 = new Rectangle(new Point(10, 10), new Size(50, 60)); Rectangle r2 = new Rectangle(40, 30, 80, 100); Rectangle rMetszet = Rectangle.Intersect(r1, r2); Rectangle rUnio = Rectangle.Union(r1, r2); Point p = new Point(100, 100); if (rMetszet.Contains(p)) eredmeny += "a metszet tartalmazza a pontot\n"; else eredmeny += "a metszet nem tartalmazza a pontot\n"; if (rUnio.Contains(p)) eredmeny += "az egyestés tartalmazza a pontot\n"; else eredmeny += "az egyestés nem tartalmazza a pontot\n"; eredmeny += String.Format("{0}, {1}, {2}, {3}\n", r1.X, r1.Y, r1.Width, r1.Height); r1.Offset(new Point(10, 20)); eredmeny += String.Format("{0}, {1}, {2}, {3}\n", r1.X, r1.Y, r1.Width, r1.Height); r1.Inflate(new Size(50, 100)); eredmeny += String.Format("{0}, {1}, {2}, {3}\n", r1.X, r1.Y, r1.Width, r1.Height); MessageBox.Show(eredmeny); }

143

F5.4 Grafikus metódusok

A Graphics osztály egy sor metódust biztosít számunkra, amelyekkel különböző vonalakat, alakzatok körvonalát, kifestett alakzatokat, szöveget és képeket jeleníthetünk meg a grafikus felületen. A metódusok paraméterei között megtaláljuk az előző rész grafikus objektumait és struktúráit.

A metódusok túlterhelt változatai a pozíciók és a méretek megadásánál int és float típusú adatokat egyaránt hívhatók. Ebben a részben csak az int típusú adatokat fogadó metódusokkal foglalkozunk.

A metódusok neve általában utal a paraméterezésükre és a működésükre. A Draw szóval kezdődő nevű metódusok tollat (pen) használnak a vonalak és körvonalak megjelenítéséhez. (Kivételt képeznek a DrawString() és a DrawImage() függvények.) Ezzel szemben a Fill kezdetű metódusok ecsettel (brush) festik ki a létrehozott zárt alakzatokat.

F5.4.1 A grafikus felület törlése

A Clear() metódus az argumentumában megadott színnel törli a teljes grafikus felületet:

void Clear(Color hattérszín)

Amennyiben a grafikát nem a Paint esemény kezelőjében állítjuk elő, a grafikus felület alapértelmezés szerinti színnel való törlését másképp is elvégezhetjük. A vezérlők/form Invalidate() és Refresh() metódusai az adott vezérlő (form) grafikus felületének újrarajzolását kezdeményezik. Az újrarajzolás következménye üres Paint eseménykezelő esetén a felület törlése.

private void btnRajzol_Click(object sender, EventArgs e) { Graphics g = this.CreateGraphics(); g.DrawLine(Pens.Blue, 0, 0, ClientSize.Width, ClientSize.Width); // átló rajzolása a formra } private void btnTöröl2_Click(object sender, EventArgs e) { Graphics g = this.CreateGraphics(); g.Clear(Color.Coral); // törlés korál színnel } private void btnTöröl1_Click(object sender, EventArgs e) { this.Refresh(); // törlés a form háttérszínével }

F5.4.2 Rajzolás/festés pontok megadásával

A rajzrutinokat sokféleképpen lehet csoportosítani. Jelen csoportokba sorolás alapja, hogy milyen alapelemek segítségével definiáljuk a megjelenítendő alakzatot. Az első csoportban egyszerű, síkbeli pontokkal határozzuk meg a rajzolás eredményét. A pontokat általában két int vagy float típusú koordinátával, illetve Point/PointF típusú struktúrákkal egyaránt megadhatjuk. (Emlékeztetőül, a float típust csak ott szerepeltetjük, ahol feltétlenül szükséges.)

A DrawLine() metódussal két pontot köthetünk össze a megadott tollal rajzolt egyenessel:

void DrawLine(Pen toll, Point pt1, Point pt2)

void DrawLine(Pen toll, int x1, int y1, int x2, int y2)

Egyenes szakaszok láncolatát a DrawLines() metódus hívásával tudjuk kényelmesen megjeleníteni. A metódus egy Point típusú tömb argumentumot vár, melynek az első két eleme definiálja az első szakasz végpontjait, és minden következő elem az előző szakaszhoz csatlakozó szakasz végpontja.

void DrawLines(Pen toll, Point[] pontok)

Az alábbi példában egy gomb megnyomásakor vonalakat rajzolunk a DrawLine() különböző paraméterezésével, egy másik gomb megnyomásával pedig ugyanazt az ábrát vonallánckénti jelenítjük meg.

private void button1_Click(object sender, EventArgs e) { Graphics g = this.CreateGraphics(); g.DrawLine(Pens.Black, 10, 40, 60, 10); g.DrawLine(new Pen(Color.DarkOrange,5), new Point(10, 100), new Point(60, 120)); Point vp = new Point(120, 70); g.DrawLine(Pens.Green, 60, 10, vp.X, vp.Y); g.DrawLine(Pens.Red, new Point(60, 120), vp); } private void button2_Click(object sender, EventArgs e) { Point[] pontok = {new Point(10, 40), new Point(60, 10), new Point(120, 70), new Point(60, 120), new Point(10, 100) }; Graphics g = this.CreateGraphics(); g.DrawLines(Pens.DarkGreen, pontok); }

A DrawLines() argumentumaként megadott pontok egy sokszög pontjainak is tekinthetők, melyeket a DrawPolygon() hívással köthetünk össze. Az első és az utolsó pont közötti szakaszt automatikusan behúzza a metódus.

void DrawPolygon(Pen toll, Point[] pontok)

private void button1_Click(object sender, EventArgs e) { Point[] pontok = {new Point(10, 40), new Point(60, 10), new Point(120, 70), new Point(60, 120), new Point(10, 100) }; Graphics g = this.CreateGraphics(); g.DrawPolygon(Pens.Blue, pontok); }

Mivel a sokszög egy zárt alakzat, festéssel is megjeleníthetjük. A FillPolygon() metódus első argumentumában egy ecsetet vár a toll helyett.

void FillPolygon(Brush ecset, Point[] pontok)

A Fill hívások a teljes alakzatot az ecsettel festve jelentik meg. Amennyiben a körvonalat is látni szeretnénk, akkor a vonalas rajzoló metódust is meg kell hívnunk, mint ahogy azt az alábbi példában tesszük:

private void button1_Click(object sender, EventArgs e) { Point[] pontok = {new Point(10, 40), new Point(60, 10), new Point(120, 70), new Point(60, 120), new Point(10, 100) }; Graphics g = this.CreateGraphics(); g.FillPolygon(Brushes.Yellow, pontok); g.DrawPolygon(new Pen(Color.Red, 3), pontok); }

Bézier-görbék

Az utolsókét csoport rutinjai a megadott pontokat felhasználva közelítő görbét jelenítenek meg. A DrawBezier() metódussal olyan görbét készíthetünk, amely a megadott pontokat közelíti az előre megadott sorrendben, azonban nem halad át rajtuk, vagy legalábbis nem mindegyeiken. A metódus paramétereként 4 pontot kell megadni, egy kezdőpontot, két vezérlő pontot és egy végpontot.

void DrawBezier(Pen toll, Point kezdőp, Point vezérlőp1, Point vezérlőp2, Point végp)

void DrawBezier(Pen toll, float x1, float y1, float x2, float y2,

float x3, float y3, float x4, float y4)

Az alábbi példa ábráján szürke színnel a görbét meghatározó négy pontot összekötő egyenes szakaszokat is megjelenítettük:

private void button1_Click(object sender, EventArgs e) { Graphics g = this.CreateGraphics(); Point[] pontok = { new Point(10, 40), new Point(10, 100), new Point(60, 120), new Point(120, 70) }; g.DrawBezier(new Pen(Color.Red, 3), pontok[0], pontok[1], pontok[2], pontok[3]); g.DrawLines(Pens.Gray, pontok); }

Bézier-görbék láncolatát is elkészíthetjük, ha a kezdő 4 pont után ismételve, 3-3 (2 vezérlő és egy vég-) pontot tömbben tárolunk.

void DrawBeziers(Pen toll, Point[] pontok)

private void button2_Click(object sender, EventArgs e) { Point[] pontok = { new Point(10, 40), new Point(10, 100), new Point(60, 120), new Point(120, 70), new Point(180, 10), new Point(230, 40), new Point(230, 100), new Point(180, 120), new Point(60, 10), new Point(10, 40) }; Graphics g = this.CreateGraphics(); g.DrawBeziers(new Pen(Color.Red, 3), pontok); g.DrawLines(Pens.Gray, pontok); }

Kardinális spline görbék

A kardinális spline a vezérlő pontokat interpoláló, azaz az előre megadott pontokon adott sorrendben áthaladó görbe, tulajdonképpen első rendben, folytonosan csatlakozó, harmadfokú Hermit-görbék sorozata.

A DrawCurve() metódusok a tömbben megadott pontokon keresztül kardinális spline görbét rajzolnak. Bizonyos függvényváltozatokban megadhatjuk, hogy melyik elemtől, hány szegmens jelenjen meg, valamint a görbe érintőinek nagyságát befolyásoló tenzió is beállítható. (Ha a tenzió 0, poligont kapunk; szokásos értéke 0.5F, és értéke általában kisebb, mint 1.0F.)

void DrawCurve(Pen toll, Point[] pontok)

void DrawCurve(Pen toll, Point[] pontok, float tenzió)

void DrawCurve(Pen toll, Point[] pontok, int kezdőIndex, int szegmensSzám, float tenzió)

void DrawCurve(Pen toll, PointF[] pontok, int kezdőIndex, int szegmensSzám)

A Bézier-görbék példáiban alkalmazott pontokra kardinális spline görbéket is fektethetünk. (A második esetben a kezdő- és a végpont ugyan egybeesik, azonban ebben a pontban a kiinduló és a beérkező görbék érintői különböznek.):

private void button1_Click(object sender, EventArgs e) { Graphics g = this.CreateGraphics(); Point[] pontok = { new Point(10, 40), new Point(10, 100), new Point(60, 120), new Point(120, 70) }; g.DrawCurve(new Pen(Color.Red, 3), pontok); g.DrawLines(Pens.Gray, pontok); }

private void button2_Click(object sender, EventArgs e) { Point[] pontok = {new Point(10, 40), new Point(10, 100), new Point(60, 120), new Point(120, 70), new Point(180, 10), new Point(230, 40), new Point(230, 100), new Point(180, 120), new Point(60, 10), new Point(10, 40) }; Graphics g = this.CreateGraphics(); g.DrawCurve(new Pen(Color.Red, 3), pontok); g.DrawLines(Pens.Gray, pontok); }

A DrawClosedCurve() metódussal zárt spline görbét rajzolhatunk, a kezdőpont ismételt megadása nélkül.

void DrawClosedCurve(Pen toll, Point[] pontok)

Ha a fenti példákban lecseréljük a metódusokat, és a második esetben az utolsó pontot eldobjuk, az alábbi görbékhez jutunk:

A zárt kardinális spline görbéket kifestve is megjeleníthetjük a FillClosedCurve() metódus segítségével:

void FillClosedCurve(Brush ecset, Point[] pontok)

A fenti példáknál maradva, a festéssel megjelenített zárt spline görbék:

F5.4.3 Befoglaló téglalappal megadott alakzatok rajzolása/festése

A grafikus metódusok másik nagy csoportjának elemei a megjelenítendő alakzat helyét és méreteit a befoglaló téglalapjukkal határozzák meg. Ennek megfelelően a legkülönfélébb paraméterezési lehetőségekkel találkozunk:

 négy számadat, melyek közül az első kettő a bal felső sarok koordinátái, a harmadik a téglalap szélessége, míg a negyedik a magassága,

 egy Rectangle típusú téglalap struktúra.

Téglalap rajzolása, festése

Egy vagy több téglalapot rajzolhatunk tollal, vagy festhetünk ecsettel az alábbi metódusok felhasználásával:

void DrawRectangle(Pen toll, Rectangle téglalap)

void DrawRectangle(Pen toll, int x, int y, int szélesség, int magasság)

void DrawRectangles(Pen toll, Rectangle[] téglalapok)

void FillRectangle(Brush ecset, Rectangle téglalap)

void FillRectangle(Brush ecset, int x, int y, int szélesség, int magasság)

void FillRectangles(Brush ecset, Rectangle[] téglalapok)

Példaként, töltsük fel a formot egyre csökkenő méretű, de azonos középpontú, véletlen színezésű téglalapokkal!

private void button1_Click(object sender, EventArgs e) { Random rnd = new Random(); Color szin; Pen toll; // A kiindulási téglalap Rectangle r = new Rectangle(0, 0, ClientSize.Width, button1.Top-5); Graphics g = this.CreateGraphics(); while (r.Width > 20 && r.Height > 20) { szin = Color.FromArgb(rnd.Next(256), rnd.Next(256), rnd.Next(256)); toll = new Pen(szin, 3); g.DrawRectangle(toll, r); r.Inflate(-20, -20); } }

Módosítsuk a programot úgy, hogy kifestett téglalapok jelenjenek meg!

private void button1_Click(object sender, EventArgs e) { Random rnd = new Random(); Color szin; Brush ecset; // A kiindulási téglalap Rectangle r = new Rectangle(0, 0, ClientSize.Width, button1.Top-5); Graphics g = this.CreateGraphics(); while (r.Width > 20 && r.Height > 20) { szin = Color.FromArgb(rnd.Next(256), rnd.Next(256), rnd.Next(256)); ecset = new SolidBrush(szin); g.FillRectangle(ecset, r); r.Inflate(-20, -20); } }

148

Ellipszisek, körök rajzolása, festése

Minden ellipszist egyértelműen megadhatunk a befoglaló téglalapjával, illetve kört a befoglaló négyzetével.

HeightWidth(X, Y)

Ezért nem is csodálkozunk azon, hogy a DrawEllipse() és a FillEllipse() metódusok paraméterezése teljesen megegyezik a téglalapot megjelenítő metódusokéval.

void DrawEllipse(Pen toll, Rectangle téglalap)

void DrawEllipse(Pen toll, int x, int y, int szélesség, int magasság)

void FillEllipse(Brush ecset, Rectangle téglalap)

void FillEllipse(Brush ecset, int x, int y, int szélesség, int magasság)

Az alábbi példában az x-koordináta és a szélesség változtatásával 10 ellipszist rajzolunk, majd a form közepén kifestünk egy 60 képpont sugarú kört:

private void button1_Click(object sender, EventArgs e) { Pen toll = new Pen(Color.Blue, 2); Graphics g = this.CreateGraphics(); int magasság = ClientSize.Height - 20; int szélesség = ClientSize.Width - 20; int sugár = 60; for (int i = 0; i < 10; i++) g.DrawEllipse(toll, 10 + 10 * i, 10, szélesség - 2 * 10 * i, magasság); g.FillEllipse(Brushes.Red, ClientSize.Width / 2 - sugár, ClientSize.Height / 2 - sugár, 2 * sugár, 2 * sugár); }

149

A grafikus felületen nem tudunk képponkokat megjeleníteni. Amennyiben szükségünk van kirajzolt pontokra, azokat 1 képpont sugarú, festett ellipszisként állíthatjuk elő. Az alábbi példában a bal egér gomb minden lenyomáskor kiteszünk az egérmutató pozíciójába egy pontot, míg a jobb egér gomb lenyomásakor töröljük a formot.

private void Form1_MouseDown(object sender, MouseEventArgs e) { switch (e.Button) { case MouseButtons.Left: Graphics g = this.CreateGraphics(); g.FillEllipse(Brushes.Red, e.X - 1, e.Y - 1, 2, 3); break; case MouseButtons.Right: this.Refresh(); break; } }

Ellipszisívek, ellipsziscikkek rajzolása, festése

Ellipszisív (arc) rajzolásakor szintén az ellipszist magában foglaló téglalapból indulunk ki. Ezen túlmenően az x-tengelytől, az óramutató járásával megegyező irányban meg kell

adnunk annak a sugárnak a szögét fokokban, ami kijelöli az ív kezdetét, majd pedig az ív szögét kell definiálnunk szintén fokokban.

Teljesen azonos módon rajzolhatunk, illetve festhetünk ellipsziscikket (pie), ha behúzzuk az ellipszisív végeihez tartozó sugarakat. Az elmondottak alapján a felhasználható metódusok:

void DrawArc(Pen toll, Rectangle téglalap, float kezdőSzög, float ívSzög)

void DrawArc(Pen toll, int x, int y, int szélesség, int magasság,

int kezdőSzög, int ívSzög)

void DrawPie(Pen toll, Rectangle téglalap, float kezdőSzög, float ívSzög)

void DrawPie(Pen toll, int x, int y, int szélesség, int magasság,

int kezdőSzög, int ívSzög)

void FillPie(Brush ecset, Rectangle téglalap, float kezdőSzög, float ívSzög)

void FillPie(Brush ecset, int x, int y, int szélesség, int magasság,

int kezdőSzög, int ívSzög)

Készítsünk programot, amely megjeleníti a sugárveszély jelet!

private void button1_Click(object sender, EventArgs e) { int kx = ClientSize.Width / 2; int ky = ClientSize.Height / 2; int r = Math.Min(kx / 2, ky / 2); int r2 = r / 4; Graphics g = this.CreateGraphics(); g.FillRectangle(Brushes.Yellow, ClientRectangle); g.FillPie(Brushes.Black, kx - r, ky - r, 2 * r, 2 * r, 0, -60); g.FillPie(Brushes.Black, kx - r, ky - r, 2 * r, 2 * r, 180, 60); g.FillPie(Brushes.Black, kx - r, ky - r, 2 * r, 2 * r, 60, 60); g.FillEllipse(Brushes.Black, kx - r2, ky - r2, 2 * r2, 2 * r2); g.DrawEllipse(new Pen(Color.Yellow, r2 / 4), kx - r2, ky - r2, 2 * r2, 2 * r2); }

F5.4.4 Szöveg megjelenítése

Szöveg grafikus felületen való megjelenítésére a DrawString() metódust használjuk:

void DrawString(string s, Font betűtípus, Brush ecset, float x, float y)

void DrawString(string s, Font betűtípus, Brush ecset, PointF pont)

void DrawString(string s, Font betűtípus, Brush ecset, RectangleF téglalap)

A metódus hívásakor mindig meg kell adnunk a megjeleníteni kívánt szöveget, a betűtípust valamint a betűk kifestéséhez használt ecsetet. A szöveg kezdőpozícióját többféleképpen is meghatározhatjuk. Téglalap átadása esetén, a téglalapból kilógó szövegrész nem jelenik meg. A szövegkiírás iránya alaphelyzetben vízszintes, azonban transzformációk alkalmazásával tetszőleges irány használható. Mindhárom metódus rendelkezhet egy további StringFormat típusú formátum paraméterrel, amellyel finombeállításokat adhatunk meg, mint például a szöveg igazítása, iránya stb.

Az alábbi példában különböző paraméterezéssel hívjuk a DrawString() metódust. Az ábrán látható szürke pontok a kiírás kezdőpozícióját jelölik.

private void button1_Click(object sender, EventArgs e) { Graphics g = this.CreateGraphics(); Brush brFekete = Brushes.Black; Brush brPiros = Brushes.Red; Font fntC = new Font("Courier", 18); Font fntTNR = new Font("Times New Roman", 16, FontStyle.Bold); Font fntCSMS = new Font("Comic Sams MS", 14, FontStyle.Italic | FontStyle.Underline); StringFormat sf = new StringFormat(); sf.Alignment = StringAlignment.Center; sf.FormatFlags = StringFormatFlags.DirectionVertical;

g.DrawString("Courier betűtípus", fntC, brFekete, 10, 10); g.DrawString("Times New Roman betűtípus példa", fntTNR, brPiros, new Rectangle(10, 50, 120, 65)); g.DrawString("Comic Sams MS betűtípus", fntCSMS, brFekete, new PointF(10, 140)); g.DrawString("Függőleges szöveg", fntTNR, brPiros, 310, 120, sf); }

Szövegek illesztett elhelyezéséhez, például a koordináta-tengelyek osztásainak feliratozásakor, pontosan kell elhelyezni az értékeket. Ehhez a kiírandó szöveg befoglaló téglalapjának méreteit a MeasureString() grafikus metódussal kell lekérdeznünk.

SizeF MeasureString(string s, Font betűtípus)

SizeF MeasureString(string s, Font betűtípus, PointF pont, StringFormat formátum)

private void button1_Click(object sender, EventArgs e) { Graphics g = this.CreateGraphics(); Font f = new Font("Calibri", 16, FontStyle.Bold); float ablakSz = ClientSize.Width; float ablakM = ClientSize.Height; Brush ecset = Brushes.Green; string szoveg = "C# programozás"; float szovegSz = g.MeasureString(szoveg, f). Width; float szovegM = g.MeasureString(szoveg, f). Height; g.Clear(Color.White); g.DrawString(szoveg, f, ecset, 0, 0); g.DrawRectangle(Pens.Red, 0, 0, szovegSz, szovegM); g.DrawString(szoveg, f, ecset, (ablakSz-szovegSz)/2, (ablakM-szovegM)/2); g.DrawEllipse(Pens.Red, (ablakSz-szovegSz)/2, (ablakM-szovegM)/2, szovegSz, szovegM); g.DrawString(szoveg, f, ecset, ablakSz - szovegSz, ablakM - szovegM); }

F5.4.5 Képek a grafikus felületen

Raszteres képek megjelenítésére a DrawImage() és DrawImageUnscaled() metódusok túlterhelt változatait használjuk. Mivel a DrawImage() metódusnak több mint 30 változata áll a rendelkezésünkre, csak az alapvetőkkel foglalkozunk.

Az alábbi metódusok a képet eredeti méretben jelenítik meg a megadott pozícióban. Az utolsó metódus a megadott téglalapra korlátozza a képet.

void DrawImageUnscaled(Image kép, Point pont)

void DrawImageUnscaled(Image kép, int x, int y)

void DrawImageUnscaledAndClipped(Image kép, Rectangle téglalap)

A következő metódusok a teljes képet a bal felső sarkát definiáló pontban jelenítik meg:

void DrawImage(Image kép, int x, int y)

void DrawImage(Image kép, Point pont)

A következő metódusok hívásakor az eredeti kép a megadott téglalapba összenyomva lesz látható:

void DrawImage(Image kép, int x, int y, int szélesség, int magasság)

void DrawImage(Image kép, Rectangle téglalap)

A kép paralelogrammában is elhelyezhető, amennyiben a paralelogrammát 3 ponttal definiáljuk a megadott tömbben:

void DrawImage(Image kép, Point[] pontok3)

A fenti DrawImage() hívásokat szemlélteti az alábbi programrészlet:

private void button1_Click(object sender, EventArgs e) { Bitmap kép = new Bitmap(@"C:\Munka\Cica.jpg"); Graphics g = this.CreateGraphics();

g.Clear(Color.LightBlue); g.DrawImage(kép, 10, 10); g.DrawImage(kép, new Rectangle(200, 10, 200, 80)); g.DrawImage(kép, 10, 150, 50, 100); Point balfelsoSarok = new Point(210, 150); Point jobbfelsoSarok = new Point(320, 150); Point balalsoSarok = new Point(250, 250); Point[] para = { balfelsoSarok, jobbfelsoSarok, balalsoSarok }; g.DrawImage(kép, para); }

Arra is van lehetőség, hogy a forrásképnek csak egy részét jelenítsük meg a fenti metódusokban is alkalmazott célterületen. Új elem a forráskép mértékegységének megadása.

void DrawImage(Image kép, int x, int y, Rectangle képTéglalap, GraphicsUnit képEgység)

void DrawImage(Image kép, Rectangle célTéglalap, int képX, int képY,

int képSzélesség, int képMagasság, GraphicsUnit képEgység)

void DrawImage(Image kép, Rectangle célTéglalap, Rectangle képTéglalap,

GraphicsUnit képEgység)

void DrawImage(Image kép, Point[] pontok3, Rectangle képTéglalap,

GraphicsUnit képEgység)

private void button1_Click(object sender, EventArgs e) { Bitmap kép = new Bitmap(@"C:\Munka\Cica.jpg"); Graphics g = this.CreateGraphics(); g.Clear(Color.LightBlue); g.DrawImage(kép, 10, 10, new Rectangle(0,0,50,50),GraphicsUnit.Pixel); g.DrawImage(kép, new Rectangle(200, 10, 200, 80), 0, 0, 50, 50, GraphicsUnit.Pixel); g.DrawImage(kép, new Rectangle(10, 150, 50, 100), new Rectangle(0, 0, 50, 50), GraphicsUnit.Pixel); Point balfelsoSarok = new Point(210, 150); Point jobbfelsoSarok = new Point(320, 150); Point balalsoSarok = new Point(250, 250); Point[] para = { balfelsoSarok, jobbfelsoSarok, balalsoSarok }; g.DrawImage(kép, para, new Rectangle(0, 0, 50, 50), GraphicsUnit.Pixel); }

153

Külön metódusok segítik az ikonok megjelentését a grafikus felületen:

void DrawIcon(Icon ikon, Rectangle célTéglalap)

void DrawIcon(Icon ikon, int x, int y)

void DrawIconUnstretched(Icon ikon, Rectangle célTéglalap)

154

private void button1_Click(object sender, EventArgs e) { Icon ikon = new Icon(@"C:\Munka\Naptar7.ico"); Graphics g = this.CreateGraphics(); g.Clear(Color.LightBlue); g.DrawIcon(ikon, 10, 10); g.DrawIcon(ikon, new Rectangle(100, 10, 100, 100)); }

A betöltött, illetve az újonnan létrehozott bitképek grafikus felületére is rajzolhatunk a grafikus metódusokkal. Ehhez elegendő megszereznünk a grafikus felület objektumának hivatkozását a Graphics.FromImage(kép) hívással. Ezzel a megoldással akár a háttérben is készthetünk rajzokat.

private void button1_Click(object sender, EventArgs e) { Bitmap bmp = new Bitmap(200, 150); Graphics gBmp = Graphics.FromImage(bmp); // a bitkép grafikus felülete int n = 6; int sz = bmp.Width / n, m = bmp.Height / n; Font f = new Font("Calibri", 52, FontStyle.Bold); gBmp.Clear(Color.Yellow); for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) gBmp.FillEllipse(Brushes.SkyBlue, new Rectangle(i * sz, j * m, sz, m)); gBmp.DrawString("C#", f, Brushes.Red, bmp.Width / 4, bmp.Height / 4); Graphics g =

this.CreateGraphics();// a form grafikus felülete g.DrawImage(bmp, new Rectangle(10, 10, 100, 75)); Point[] para = { new Point(270, 10), new Point(270, 210), new Point(120, 10) }; g.DrawImage(bmp, para); }

F5.4.6 Egyszerű síkbeli transzformációk alkalmazása

A grafikus felület metódusai az alakzatok megjelenítése során egy síkbeli transzformációkat leíró, Matrix típusú mátrixot használnak. (Ehhez a mátrixhoz közvetlenül is hozzáférhetünk a Graphics objektumok Transform tulajdonságán keresztül.) Mélyebb geometriai fejtegetés helyett, csak a legegyszerűbb transzformációk (az eltolás, az elforgatás és a skálázás) bemutatására szorítkozunk.

void TranslateTransform(float dx, float dy)

void RotateTransform(float szög)

void ScaleTransform(float sx, float sy)

void ResetTransform()

Mindegyik metódus hívása a transzformációs mátrixon fejti ki hatását, az utolsó metódus törli a transzformációkat. Mivel a beállított transzformációs műveletek a grafikus felület koordináta-rendszerére vonatkoznak, így ugyanazok a rajzolási műveletek különböző eredményt adnak a transzformációk beállítását követően.

155

private void Rajzol(Graphics g, Color szin) { g.DrawEllipse(new Pen(szin, 3), 10, 10, 200, 80); g.DrawLine(new Pen(szin, 3), 10, 10, 210, 90); g.DrawString("C#", new Font("Courier", 30), new SolidBrush(szin), 140, 20); } private void button1_Click(object sender, EventArgs e) { Graphics g = this.CreateGraphics(); Rajzol(g, Color.Red); // az eredeti rajz g.TranslateTransform(250.0F, 0.0F); // eltolás 250 egységgel a pozitív x-irányba Rajzol(g, Color.Green); // eltolt rajz g.RotateTransform(45.0F); // elforgatás 45 fokkal Rajzol(g, Color.Blue); // eltolt és elforgatott rajz g.ScaleTransform(1.4F, 0.8F); // skálázás x-irányban 1.4x, y-irányban 0.8x Rajzol(g, Color.Black); // eltolt, elforgatott és skálázott rajz g.ResetTransform(); // a trasznformációk törlése g.DrawEllipse(new Pen(Color.Gold, 3), 10, 150, 200, 80); }

Készítsünk programot, amely felrajzolja a Yin Yang ábrát, majd pedig 10 fokonként megforgatja az időzítő (Timer) segítségével! További követelmény, hogy az ablak méretének változásakor ne maradjon „szemét – korábbi grafika” a formon.

Ez utóbbi követelmény egyszerűen biztosítható, ha a form Resize eseményének kezelőjéből az ablak újrarajzolását kérjük (Refresh()), hiszen a Paint esemény kezelőjét nem definiáltuk.

private float szog = 0; private void YinYang() { Graphics g = this.CreateGraphics(); g.ResetTransform(); g.TranslateTransform(ClientSize.Width / 2, ClientSize.Height / 2); g.ScaleTransform(1.5F, 1.5F); g.RotateTransform(szog); g.FillEllipse(Brushes.White, -100, -100, 200, 200); g.FillPie(Brushes.Black, -100, -100, 200, 200, 90, -180); g.FillPie(Brushes.White, -50, -100, 100, 100, 90, -180); g.FillPie(Brushes.Black, -50, 0, 100, 100, 90, 180); g.DrawEllipse(new Pen (Color.Black,3), -100, -100, 200, 200); g.FillEllipse(Brushes.Black, -15, -65, 30, 30); g.FillEllipse(Brushes.White, -15, 35, 30, 30); }

156

private void button1_Click(object sender, EventArgs e) { timer1.Interval = 100; timer1.Enabled = !timer1.Enabled; } private void timer1_Tick(object sender, EventArgs

e) { szog += 10; YinYang(); } private void Form1_Resize(object sender, EventArgs e) { this.Refresh(); }

F5.5 A Color struktúrában definiált színnevek

A Color struktúra csak olvasható, statikus tulajdonságai 141 színt tárolnak név szerint. A nevek közül 140 a Unix X-11 rendszerből származik. Az egyetlen nem Unix X-11 név a Transparent, amely egy, a rendszer által meghatározott, átlátszó színt jelöl.

AliceBlue 240,248,255

LightSalmon 255,160,122

AntiqueWhite 250,235,215

LightSeaGreen 32,178,170

Aqua 0,255,255

LightSkyBlue 135,206,250

Aquamarine 127,255,212

LightSlateGray 119,136,153

Azure 240,255,255

LightSteelBlue 176,196,222

Beige 245,245,220

LightYellow 255,255,224

Bisque 255,228,196

Lime 0,255,0

Black 0,0,0

LimeGreen 50,205,50

BlanchedAlmond 255,255,205

Linen 250,240,230

Blue 0,0,255

Magenta 255,0,255

BlueViolet 138,43,226

Maroon 128,0,0

Brown 165,42,42

MediumAquamarine 102,205,170

BurlyWood 222,184,135

MediumBlue 0,0,205

CadetBlue 95,158,160

MediumOrchid 186,85,211

Chartreuse 127,255,0

MediumPurple 147,112,219

Chocolate 210,105,30

MediumSeaGreen 60,179,113
Coral 255,127,80
MediumSlateBlue 123,104,238
CornflowerBlue 100,149,237
MediumSpringGreen 0,250,154
Cornsilk 255,248,220
MediumTurquoise 72,209,204
Crimson 220,20,60
MediumVioletRed 199,21,112
Cyan 0,255,255
MidnightBlue 25,25,112
DarkBlue 0,0,139
MintCream 245,255,250
DarkCyan 0,139,139
MistyRose 255,228,225
DarkGoldenrod 184,134,11
Moccasin 255,228,181
DarkGray 169,169,169
NavajoWhite 255,222,173
DarkGreen 0,100,0
Navy 0,0,128
DarkKhaki 189,183,107
OldLace 253,245,230
DarkMagena 139,0,139
Olive 128,128,0
DarkOliveGreen 85,107,47
OliveDrab 107,142,45
DarkOrange 255,140,0
Orange 255,165,0
DarkOrchid 153,50,204
OrangeRed 255,69,0
DarkRed 139,0,0
Orchid 218,112,214
DarkSalmon 233,150,122
PaleGoldenrod 238,232,170
DarkSeaGreen 143,188,143
PaleGreen 152,251,152

DarkSlateBlue 72,61,139
PaleTurquoise 175,238,238
DarkSlateGray 40,79,79
PaleVioletRed 219,112,147
DarkTurquoise 0,206,209
PapayaWhip 255,239,213
DarkViolet 148,0,211
PeachPuff 255,218,155
DeepPink 255,20,147
Peru 205,133,63
DeepSkyBlue 0,191,255
Pink 255,192,203
DimGray 105,105,105
Plum 221,160,221
DodgerBlue 30,144,255
PowderBlue 176,224,230
Firebrick 178,34,34
Purple 128,0,128
FloralWhite 255,250,240
Red 255,0,0
ForestGreen 34,139,34
RosyBrown 188,143,143
Fuschia 255,0,255
RoyalBlue 65,105,225
Gainsboro 220,220,220
SaddleBrown 139,69,19
GhostWhite 248,248,255
Salmon 250,128,114
Gold 255,215,0
SandyBrown 244,164,96
Goldenrod 218,165,32
SeaGreen 46,139,87
Gray 128,128,128
Seashell 255,245,238
Green 0,128,0
Sienna 160,82,45
GreenYellow 173,255,47

Silver 192,192,192
Honeydew 240,255,240
SkyBlue 135,206,235
HotPink 255,105,180
SlateBlue 106,90,205
IndianRed 205,92,92
SlateGray 112,128,144
Indigo 75,0,130
Snow 255,250,250
Ivory 255,240,240
SpringGreen 0,255,127
Khaki 240,230,140
SteelBlue 70,130,180
Lavender 230,230,250
Tan 210,180,140
LavenderBlush 255,240,245
Teal 0,128,128
LawnGreen 124,252,0
Thistle 216,191,216
LemonChiffon 255,250,205
Tomato 253,99,71
LightBlue 173,216,230
Turquoise 64,224,208
LightCoral 240,128,128
Violet 238,130,238
LightCyan 224,255,255
Wheat 245,222,179
LightGoldenrodYellow 250,250,210
White 255,255,255
LightGreen 144,238,144
WhiteSmoke 245,245,245
LightGray 211,211,211
Yellow 255,255,0
LightPink 255,182,193
YellowGreen 154,205,50
F5.6 Mintafeladatok

Az alábbiakban három tipikusnak mondható feladat megoldásával mutatjuk be a programozott grafika használatát. A lépéseket nem magyarázzuk teljes részletességgel, hiszen a fejezet előző részeiben megtettük ezt.

F5.6.1 Rajzolás szövegfájlból vett adatokkal

Jelenítsük meg az ablakban az F4.4 fejezet példaprogramja által előállított Utvonal.txt állomány tartalmá! Az egyenesekkel összekötött csomópontokat kifestett kör jelölje, és a koordináták is jelenjenek meg a kör alatt!

using System.IO; private void button1_Click(object sender, EventArgs e) { Graphics g = this.CreateGraphics(); Pen toll = new Pen(Color.FromArgb(100,0,0,255), 5); Font f = new Font("Calibri", 10); StringFormat sf = new StringFormat(); sf.Alignment = StringAlignment.Center; // eltoljuk, hogy minden elférjen az ablakban g.TranslateTransform(20, -20); StreamReader sr = new StreamReader("C:\\Munka\\Utvonal.txt"); int n = Convert.ToInt32(sr.ReadLine()); // az adatok száma string sor; string[] sxy; int xkezd = 0, ykezd = 0, xveg = 0, yveg = 0; for (int i = 0; i < n; i++) { sor = sr.ReadLine(); sxy = sor.Split(','); // a szövegsor feldarabolása a vesszőnél xveg = Convert.ToInt32(sxy[0]); yveg = Convert.ToInt32(sxy[1]); // a csomópont rajzolása g.FillEllipse(Brushes.Red, xveg - 10, ClientSize.Height - yveg - 10, 20, 20); // a koordináták megjelenítése g.DrawString("(" + sor + ")", f, Brushes.Red, xveg, ClientSize.Height - yveg + 10,sf); if (i != 0) // a 2. ponttól húzunk egyenest { // 2. ponttól kezdve a csomópontok összekötése g.DrawLine(toll, xkezd, ClientSize.Height - ykezd, xveg, ClientSize.Height - yveg); } xkezd = xveg; ykezd = yveg; } sr.Close(); }

F5.6.2 Pattogó billiárdgolyó animációja

Készítsünk alkalmazást, amely az ablak közepén álló billiárdgolyónak megfelelő alakzatot gombnyomás hatására, véletlenszerű irányban és lépésekkel elindítja, majd az ablakot határoló kerethez teljesen rugalmasan ütközteti! A mozgatást időzítő vezérelje!

Teljesen rugalmatlan ütközés esetén a golyó sebességének fallal párhuzamos összetevője változatlan marad, míg a falra merőleges összetevő előjelet vált.

using System; using System.Drawing; using System.Windows.Forms; namespace PattogoBilliardgolyo { public partial class Form1 : Form { private const int r = 20; private int x = r, y = r; private int dx = 1, dy = 1; public Form1() { InitializeComponent(); } private void Form1_Paint(object sender, PaintEventArgs e) { Graphics g = e.Graphics; g.FillEllipse(Brushes.White, x - r, y - r, 2*r, 2*r); } private void button1_Click(object sender, EventArgs e) { Random rnd = new Random(); dx = rnd.Next(r) * (2 * rnd.Next(2) - 1); dy = rnd.Next(r) * (2 * rnd.Next(2) - 1); x = ClientSize.Width / 2; y = ClientSize.Height / 2; Refresh(); timer1.Interval = 60; // 600ms timer1.Enabled = !timer1.Enabled; if (timer1.Enabled) button1.Text = "Stop"; else button1.Text = "Start"; }

private void timer1_Tick(object sender, EventArgs e) { x += dx; y += dy; if (x <= r || x >= (ClientSize.Width - r)) dx = -dx; if (y <= r || y >= (ClientSize.Height - r)) dy = -dy; Refresh(); // kirajzoljuk! } } }

Ha r sugarú golyóval dolgozunk, az ábrán látható módon figyelembe kell vennünk a sugár méretét.

A program ablaka futás közben:

F5.6.3 A ferde hajítás szimulációja

Egy testet vízszintes terepen, a felszínnel α szöget bezáróan, v0 sebességgel lövünk ki. Készítsünk programot, amely szimulálja a kilőtt test mozgását! Az adatokat úgy vetítsük az ablakba, hogy a test a bal alsó sarokból indul, és a jobb alsó sarokba érkezik.

A légellenállás nélküli ferde hajítás matematikai modelljéhez használt koordináta-rendszer és jelölések:

A mozgáspálya megjelenítéséhez szükségünk van az s távolságra:

A fizikai összefüggések felhasználásával megkaphatjuk a vízszintes és függőleges irányú sebességeket az idő függvényében. Ezzel már meghatározhatjuk a mozgáspálya pontjait:

A szimulációban egy kicsiny dt időtartamban egyes vonalú, egyenletes mozgással számolunk mindkét irányban. Minden dt időtartamban, függőleges irányban az előző és a következő sebesség-összetevők átlagát használjuk a számításokhoz.

private void button1_Click(object sender, EventArgs e) { const float g = 9.81F; float v0, alfa; alfa = (float)(Convert.ToSingle(textBox1.Text) * Math.PI / 180); v0 = Convert.ToSingle(textBox2.Text); float s, arany; s = (float)(v0 * v0 * Math.Sin(2 * alfa) / g); arany = ClientSize.Width / s; Graphics grafika = this.CreateGraphics(); grafika.Clear(Color.SkyBlue); float vy1, vy2, x, y, t, m, dt = 0.05F, fx, fy; x = y = t = m = 0; vy1 = (float)(v0 * Math.Sin(alfa)); while (y >= 0) { vy2 = vy1 - g * dt; x += (float)(v0 * Math.Cos(alfa) * dt); y += (vy1 + vy2) * dt / 2; if (y > m) m = y; fx = x * arany; fy = ClientSize.Height - y * arany; grafika.FillEllipse(Brushes.Red, fx - 5, fy - 5, 10, 10); vy1 = vy2; t += dt; } MessageBox.Show(String.Format("Idő: {0} mp\nTávolság: {1} m\nMagasság: {2} m", t, x, m)); }

A megoldásban piros színnel kiemeltük azokat a sorokat, amelyek a megjelenítésért felelősek. g2sinvs20□□□tgvvvvyx□□□□□□□□sincos00