

A C# programozási nyelv a felsőoktatásban

Programozás tankönyv

**Dr. Kovács Emőd
Hernyák Zoltán
Radványi Tibor
Király Roland**

Tartalom

| | |
|--|----|
| Tartalom | 2 |
| Történeti áttekintő | 7 |
| Első generációs programozási nyelvek: GÉPI KÓD..... | 8 |
| Második generációs programozási nyelvek: ASSEMBLY | 9 |
| Változók az assembly nyelvben | 10 |
| Vezérlési szerkezetek az assembly nyelvben | 13 |
| Eljáráshívás az assembly nyelvben | 14 |
| Több modulból álló programok az assembly nyelvben | 14 |
| Harmadik generációs programozási nyelvek: PROCEDURÁLIS NYELVEK | 15 |
| Az első nagyon fontos változás – az eljárás fogalmának bevezetése..... | 15 |
| A második fontos változás a változó fogalmának finomodása: | 15 |
| A harmadik fontos változás – a típusrendszer bővíthetősége..... | 16 |
| A negyedik fontos változás – a vezérlési szerkezetek bevezetése | 17 |
| Az ötödik fontos változás – a hardware függetlenség..... | 17 |
| Három-és-fél generációs programozási nyelvek: OBJEKTUM ORIENTÁLT NYELVEK | 18 |
| Negyedik generációs programozási nyelvek: SPECIALIZÁLT NYELVEK | 18 |
| Ötödik generációs programozási nyelvek: MESTERSÉGES INTELLIGENCIA NYELVEK | 18 |
| A programozási nyelvek csoportosítása | 18 |
| A programozási nyelveket más szempontból vizsgálva egy másik csoportosítás fedezhető fel:..... | 18 |
| Imperatív (procedurális) nyelvek: | 18 |
| Applikatív (funkcionális) nyelvek:..... | 19 |
| Logikai nyelvek:..... | 19 |
| Objektum-orientált nyelvek: | 19 |
| Futtató rendszerek | 19 |
| Bevezetés A Microsoft®.NET | 22 |
| „Helló Világ!” | 28 |
| Feladatok: | 36 |
| „Alap I/O” | 37 |
| Az alapvető Input/Output | 38 |
| Programozási feladatok | 53 |
| „Szelekció alapszint” | 54 |
| A logikai típusú változó | 55 |
| A feltételes utasítás..... | 56 |
| Az elágazás..... | 57 |
| Néhány egyszerű példát a szelekció alkalmazására. | 58 |
| Döntsük el egy számról, hogy páros-e! | 58 |
| Oldjuk meg az együtthatóival adott másodfokú egyenletet! | 59 |
| Megoldásra ajánlott feladatok | 60 |
| Előírt lépésszámú ciklusok | 61 |
| Feladatok: | 69 |
| „Vektorok!” | 71 |
| Vektorok kezelése | 72 |
| Tömb deklarálása | 72 |

| | |
|---|-----|
| A tömb elemeinek elérése | 73 |
| A tömb elemeinek rendezése, keresés a tömbben | 74 |
| Vektor feltöltése billentyűzetről..... | 78 |
| Vektor feltöltése véletlenszám-generátorral..... | 80 |
| N elemű vektorok kezelése..... | 81 |
| Összegzés | 81 |
| Maximum és minimum kiválasztása | 82 |
| Eldöntés | 83 |
| Kiválogatás..... | 84 |
| Dinamikus méretű vektorok | 86 |
| Az ArrayList főbb jellemzői: | 86 |
| Az ArrayList főbb metódusai:..... | 86 |
| Feladatok dinamikus tömbre | 87 |
| Többdimenziós tömbök..... | 92 |
| További megoldásra ajánlott feladatok | 96 |
| „Logikai ciklusok” | 99 |
| A ciklusok | 100 |
| A while ciklus..... | 100 |
| A break | 104 |
| A continue | 105 |
| A do while ciklus..... | 105 |
| A foreach | 106 |
| Programozási feladatok | 108 |
| „Szelekció emelt szint!” | 109 |
| Szelekció haladóknak | 110 |
| További megoldásra ajánlott feladatok | 116 |
| Stringek kezelése..... | 117 |
| A string típusú változó | 118 |
| A string-ek mint vektorok | 123 |
| Programozási feladatok | 126 |
| Eljárások alapfokon..... | 127 |
| Feladatok: | 133 |
| Függvények írása..... | 136 |
| Feladatok: | 141 |
| Eljárások és függvények középfokon..... | 142 |
| Feladatok: | 147 |
| Eljárások és függvények felsőfokon | 148 |
| „WinForm” | 156 |
| A Windows Formok | 157 |
| Hibakezelés | 157 |
| A try és a catch | 157 |
| A finally blokk | 160 |
| Kivételek feldobása | 161 |
| Checked és unchecked | 162 |
| Programozási feladatok | 163 |
| Új Projekt készítése..... | 164 |
| Feladatok | 170 |
| A látható komponensek, a kontrollok | 171 |
| Button (Gomb) | 171 |
| Label, Linklabel | 172 |

| | |
|--|-----|
| Textbox..... | 173 |
| CheckBox..... | 174 |
| GroupBox..... | 175 |
| MainMenu..... | 176 |
| RadioButton..... | 178 |
| ComboBox..... | 179 |
| ListView..... | 183 |
| TreeView..... | 186 |
| TabControl..... | 189 |
| DateTimePicker komponens..... | 191 |
| MonthCalendar komponens..... | 194 |
| HorizontalScrollBar és VerticalScrollBar komponensek..... | 197 |
| Listbox..... | 199 |
| Panel..... | 200 |
| PictureBox..... | 201 |
| Timer komponens..... | 202 |
| NumericUpDown..... | 203 |
| ProgressBar..... | 205 |
| TrackBar ellenőrzés..... | 206 |
| HelpProvider..... | 207 |
| ImageList..... | 211 |
| RichTextBox..... | 215 |
| ToolTip..... | 218 |
| ContextMenu..... | 220 |
| NotifyIcon..... | 222 |
| StatusBar..... | 223 |
| ToolBar..... | 225 |
| Formok használata, formok típusai..... | 228 |
| A rendszer által biztosított üzenetablakok használata..... | 228 |
| Modális és nem modális formok..... | 231 |
| Dialógusok..... | 234 |
| A fontDialog..... | 234 |
| ColorDialog..... | 236 |
| PrintPreviewDialog..... | 237 |
| Fájl megnyitása és mentése..... | 238 |
| MessageBox..... | 238 |
| Feladatok..... | 239 |
| Többszálú programozás..... | 240 |
| Feladatok..... | 243 |
| „Grafikai alapok!”..... | 244 |
| A grafikus szoftver..... | 245 |
| GDI+..... | 247 |
| GDI+ osztály és interfész a .NET-ben..... | 247 |
| Névterek, namespaces..... | 247 |
| A System.Drawing névtér osztályai és struktúrái..... | 248 |
| A Graphics osztály..... | 248 |
| A GDI+ újdonságai..... | 250 |
| Gazdagabb színkezelés és színátmenetek lehetősége..... | 250 |
| Antialiasing támogatás..... | 250 |
| Cardinal Spline-ok..... | 251 |

| | |
|---|-----|
| Mátrix transzformációk | 252 |
| Skálázható régiók (Scalable Regions) | 252 |
| Alpha Blending | 252 |
| Sokkféle grafikus fájl formátum támogatása (Support for Multiple-Image Formats): .. | 253 |
| Néhány változás a GDI+ programozásban | 254 |
| Vonal rajzolás GDI+ használatával | 254 |
| Metódusok felülbíráltása (Method Overloading) | 255 |
| Többé nincs a grafikus kurzornak aktuális pozíciója (Current Position) | 255 |
| Szétválasztott metódus a rajzolásra (Draw) és a kitöltésre (Fill) | 256 |
| Regiók létrehozása | 257 |
| Interpoláció és approximáció | 261 |
| Hermit-görbe | 261 |
| Bézier-görbe | 262 |
| de Casteljau-algoritmus | 262 |
| A Bézier-görbe előállítása Bernstein-polinommal | 263 |
| Bézier-görbe néhány tulajdonságai | 263 |
| Harmadfajú Bézier-görbék | 264 |
| Kapcsolódó Bézier-görbék | 265 |
| Cardinal spline | 267 |
| Pontranszformációk | 271 |
| Homogén koordináták | 271 |
| Áttérés hagyományos Descartes koordinátákról homogén koordinátákra: | 271 |
| Visszatérés homogén koordinátákról Descartes koordinátákra: | 271 |
| Pontranszformációk | 271 |
| Első példa: | 272 |
| Második példa: | 273 |
| GDI+ transzformációs metódusai | 274 |
| Eltolás: | 275 |
| Elforgatás az origó körül alfa szöggel pozitív irányba: | 275 |
| Tükrözés: | 275 |
| Skálázás: | 276 |
| Nyírás: | 276 |
| A koordináta rendszer transzformálása és a Path használata | 278 |
| Grafikus konténerek | 279 |
| „Adatok kezelése!” | 281 |
| Az ADO.NET Adatkezelés C#-ban | 282 |
| SQL Server Enterprise Manager Az SQL Server és az MMC | 282 |
| Új elemek létrehozása | 283 |
| MS SQL szerver elérése C#-ból | 288 |
| Bevezetés az ADO.NET kapcsolódási eszközeihez | 288 |
| Connection String | 288 |
| Kapcsolat létrehozása és megszüntetése | 288 |
| Összetett kapcsolatok (Pooling Connections) | 289 |
| Tranzakciók | 289 |
| Beállítható kapcsolat tulajdonságok | 289 |
| Kapcsolatok tervezéskor a Server Explorer-ben | 290 |
| Kapcsolat tervezési eszközök Visual Studio-ban | 290 |
| Kapcsolat létrehozása SQL Server-hez ADO.NET használatával | 290 |
| Kapcsolat bontása | 291 |
| ADO.NET kapcsolat objektumok létrehozása | 291 |

| | |
|--|-----|
| Kapcsolat létrehozása | 291 |
| Kapcsolat létrehozása SQL Server-hez | 292 |
| SQL Server kapcsolat Server Explorer-ben | 292 |
| Kapcsolódás SQL Server-hez az alkalmazásunkból | 292 |
| Kapcsolódás létrehozása vizuális eszközökkel | 292 |
| Server Explorer-ből | 292 |
| A DataTable osztály | 293 |
| Szűrés és rendezés az ADO.NET-ben | 293 |
| Szűrés és rendezés a DataView objektum segítségével | 295 |
| Az alapértelmezett nézet | 296 |
| A RowFilter tulajdonság | 296 |
| Rendezés a DataViewban | 296 |
| Tárolt eljárások | 302 |
| Mi is az a Transact-SQL? | 302 |
| Alapvető programozási szerkezetek: | 302 |
| Változók | 302 |
| Feltételes szerkezetek használata | 303 |
| CASE utasítások | 304 |
| While ciklusok | 304 |
| A CONTINUE utasítás | 305 |
| A BREAK utasítás | 305 |
| RETURN utasítások használata | 305 |
| WAITFOR utasítások használata | 305 |
| RAISERROR utasítások használata | 305 |
| KURZOROK használata | 306 |
| Mint a példaprogramban is látható, a változók típusa meg kell, hogy egyezzen a kinyert sorok oszlopainak típusával. | 307 |
| Függvények használata | 307 |
| Felhasználói függvények létrehozása | 308 |
| Helyben kifejtett táblaértékű függvények | 309 |
| Többutasításos táblaértékű függvények | 309 |
| Tárolt eljárások létrehozása: | 310 |
| Tárolt eljárások végrehajtása: | 311 |
| Kioldók | 311 |

Programozás tankönyv

I. Fejezet

Történeti áttekintő

Hernyák Zoltán

A számítástechnika történetének egyik fontos fejezete a programozási nyelvek kialakulása, története, fejlődése. A fejlődés során a programozási nyelvek szintaktikája változott meg, elősegítve a programozási hibák minél korábban (lehetőleg fordítási időben) történő felfedezését. Egy igazán jó programozási nyelven nagyon sok hibafajta eleve el sem követhető, mások könnyen elkerülhetőek.

Egy jó programozási nyelv sokféle jellemzővel rendelkezik. Emeljünk ki néhányat ezek közül:

- könnyen elsajátítható alapelvekkel rendelkezik,
- könnyen áttekinthető forráskód,
- könnyen módosítható, bővíthető a forráskód,
- nehéz hibát elkövetni kódolás közben,
- könnyen dokumentálható a kód.

A programozási nyelveket generációkba lehet sorolni a fejlődés bizonyos szakaszait figyelembe véve:

Első generációs programozási nyelvek: GÉPI KÓD

A számítógépek a NEUMANN elveknek megfelelően a végrehajtandó programutasításokat a memóriában tárolják. A memória ma már alapvetően byte szervezésű, egyetlen nagy méretű byte-sorozatnak tekinthető. Minden byte egy egész számot jelölhet, 0..255 értéktartományból. Ebből az következik, hogy a mikroprocesszor alapvetően az utasításokat is számoknak tekinti.

A gépi kódú programozási nyelvben az utasításokat számkódok jelölik. Amennyiben az utasításnak vannak paraméterei, úgy azokat is számként kell megadni. A gépi kódban létező fogalom a *regiszter*, amely a mikroprocesszoron belüli tárlórekeszt jelöl. Egy ilyen rekesz tartalma egy egész szám lehet. A regisztereknek kötött nevük van, pl. AX, BX, CX, DX. A 32 bites processzorokon a regiszterek nevei felvették az 'E' előtagot (Extended AX regiszter – EAX). Aránylag kevés regiszter áll rendelkezésre (kevesebb mint 20 darab), és többnek speciális feladat volt, ezért nem lehetett akármilyen célra felhasználni. Két szám összeadását az alábbi módon kell végrehajtani:

1. Olvassuk be az első számot az EAX regiszterbe a memóriából.
2. Az EAX regiszterhez adjuk hozzá a második számot.
3. Az eredményt (az EAX regiszter új értékét) tároljuk a memória egy másik pontján.

| | | |
|----------|--------|---------------------|
| 0044F02B | 8B45F4 | mov eax, [ebp-\$0c] |
| 0044F02E | 0345F0 | add eax, [ebp-\$10] |
| 0044F031 | 8945EC | mov [ebp-\$14], eax |

1. ábra

Az utasításokat számkódok jelölik. Ezek a számok 0..255 közötti egész számok. A számkódokat leggyakrabban hexadecimális formában adják meg. A bal oldali oszlopban a memóriacímeket adjuk meg (0044F02B,...) ahol az adott gépi kódú utasítást tároljuk. A gépi kódú utasítások a második oszlopban vannak (a 8B45F4

számsorozat egyetlen gépi kódú utasítást (8B), valamint a paramétereit jelöli: honnan kell beolvasni az értéket az EAX regiszterbe (45,F4)).

A fentieken is látszik, hogy a gépi kódú programozási nyelv nehézkes, nehezen tanulható. A kész program nehezen megérthető, nem áttekinthető.

Sok más hátránya mellett külön kiemelendő, hogy a gépi kódú programokat alkotó utasítások csak az adott mikroprocesszor számára érthetőek. Vagyis más processzor esetén az utasításkódok is mások. Nemcsak számkódjukban különböznek, hanem esetleg kevesebb vagy több utasítás van, illetve más-más a paraméterezése a hasonló feladatú utasításoknak. Ha egy gépi kódban programozó számára egy másik processzorra kellett programot írni, először még el kellett sajátítania a különbségeket. Nyilván az alapelvek maradtak, de az utasítások különbözősége sok nehézséget okozott.

A programozó szemszögéből a gépi kódban történő programozás nagyon lassú folyamat. Aprólékosan lehet csak a programot felépíteni. Az utasítások nagyon alacsony szintűek voltak, egy egyszerű összeadás művelet is - mint láttuk a fenti példán – három utasításból állt. Egy nagyobb rendszer elkészítése olyan időigényes feladat lenne, hogy inkább csak rövidebb, egyszerű programokat készítettek benne a programozók.

Előnyei persze akadnak ennek a nyelvnek is: a gépi kódú utasítások segítségével maximalizálhatjuk a programunk futási sebességét, vagy memória-kihasználtságát (vagy mindkettőt egyszerre), hiszen megkötések nélkül felhasználhatjuk a mikroprocesszor minden lehetőségét, és szabadon használhatjuk a memóriát is.

Második generációs programozási nyelvek: ASSEMBLY

A gépi kódú programozási nyelv hátrányai miatt új nyelvet kellett kifejleszteni.

Az ASSEMBLY nyelv első közelítésben a 'megérthető gépi kód' nyelve. Az utasítások számkódjait néhány betűs (2-3-4 betűs) ún. mnemonikkal helyettesítették. Egy ilyen mnemonik (emlékeztető szócska) a gépi kódú utasítás jelentéstartalmára utalt. Például a „memória-tartalom beolvasása egy regiszterbe” (bemozgatás) az angol MOVE=mozgatni szó alapján a MOV mnemonikot kapta. A „két szám összeadása” az angol ADD=összeadni mnemonikot kapta. Az 1. ábrán a harmadik oszlopban szerepelnek a gépi kódú utasítások assembly nyelvű megfelelői.

A MOV utasítás önmagában nem lefordítható gépi kódra, hiszen azt is meg kell adni, hogy melyik memória-cím tartalmát kell betölteni melyik regiszterbe. Az utasítás egyik lehetséges formája „MOV EAX,<memcím>”. Ennek már egyértelműen megfelel egy gépi kódú utasításkód (mov eax = 8B), a memóriacímet pedig a további számkódok írják le.

Ennek megfelelően az assembly nyelv egy adott mikroprocesszor adott gépi kódjához készült el. Ezért az assembly nyelv is processzor-függő, de ezen a szinten újabb fogalmak jelentek meg:

Forráskód: az assembly nyelvű programot a CPU nem képes közvetlenül megérteni és végrehajtani. Az assembly nyelvű programot egy szöveges file-ban kell megírni (forráskód), majd le kell fordítani gépi kódú, ún. tárgyprogramra (object code).

Fordítás: az a folyamat, amely közben a forráskódot egy **fordítóprogram** (compiler) gépi kódú utasítások sorozatára transzformálja.

Az assembly nyelv esetén ezt a fordítóprogramot ASSEMBLER-nek nevezték. Az assembler végigolvasta a forráskódot sorról-sorra, e közben generálta a gépi kódú utasítássorozatot - a mnemonikok alapján előállította a megfelelő gépi kódú utasítás számkódját. Az assembler nem bonyolult program, hiszen a fordítás egyszerű szabályok szerint működik.

Az ASSEMBLER a forráskód feldolgozása közben egyszerű ellenőrzéseket is elvégez. Amennyiben valamely mnemonikot nem ismeri fel (pl. a programozó elgépelte), akkor a fordítási folyamat leáll, az assembler hibaüzenettel jelzi a hiba okát és helyét a forráskódban. A forráskódban az olyan jellegű hibákat, melyek súlyossága folytán a fordítás nem fejezhető be – fordítási időben történő hibának nevezzük. Ennek oka általában a nyelv szabályai (szintaktikája) elleni vétség, ezért ezt **szintaktikai hibának** is nevezzük.

Az assembly nyelvű programok forráskódja olvashatóbb, mint a gépi kód, illetve könnyebben módosítható. Az assembler a program szintaktikai helyességét ellenőrzi le, emiatt az első eszköznek is tekinthetjük, amelyet a programozók munkájának segítésére (is) alkottak.

A fordítás ellenkező művelete a **de-compiling**. Ennek során a gépi kódú program-változatból a programozó megpróbálta visszaállítani annak assembly nyelvű eredetijét. Erre sokszor azért volt szükség, mert az eredeti forráskód elveszett, de a programon mégis módosítani kellett. A gépi kódú programok módosíthatóságának nehézségét jelzi, hogy az assembly-programozók már nem voltak hajlandók közvetlenül a gépi kódot módosítani, inkább helyreállították az assembly forráskódot, abban elvégezték a módosításokat, majd újra lefordították (generálták) a gépi kódú programot.

Ez a művelet persze némi veszteséggel járt – hiszen a tényleges eredeti assembly forráskódban többek között megjegyzések is lehettek. Ezek a megjegyzések a gépi kódú változatba nem kerültek bele, ezért visszaállítani sem lehet őket.

Ami viszont nagyon fontos lépés – a **fordítóprogramok** megjelenésével megjelent az igény ezek **intelligenciájának fejlődésére!**

Változók az assembly nyelvben

Az assembly programokban a memória-címekre azok sorszámaival lehetett hivatkozni. Egy 128Mb memóriával szerelt számítógépben $128 \cdot 1024 \cdot 1024$ db

sorszámot lehet használni, mivel a memória ennyi db byte-ot tartalmaz. A példában ezek a sorszámok a [0 .. 134.217.727] intervallumból kerülnek ki¹.

Az assembly programok sokszor tényleges memóriacímeket (sorszámokat) tartalmaztak számkonstansok formájában. Ez sok szempontból nem volt elég rugalmas:

- A számok nem hordozzák a jelentésüket, nehezen volt kiolvasható a programból, hogy a 1034-es memóriacímen milyen adat van, mi az ott lévő érték jelentése.
- A programban e memóriacímek sokszor előfordultak. Ezért a memóriacímek nem voltak könnyen módosíthatóak.
- A memóriacím alapján nem dönthető el, hogy az ott tárolt adat hány byte-ot foglal el, ugyanis a memóriacím csak a memóriabeli kezdőcímet jelöli.

A helyzet javítása érdekében a programozók elkezdtek konstansokat használni a programjaikban. Ezeket a programok elejére összefoglaló jelleggel, táblázat-szerű módon írták le:

| | |
|-----------|--------|
| FIZETES | = 1034 |
| ELETKOR | = 1038 |
| NYUGDIJAS | = 1040 |

Ezek után az assembly programokban az ADD EAX,[1034] helyett (ami azt jelentette, hogy add hozzá az [1034] memóriacímen található értéket az EAX regiszterben éppen benne levő értékhez) azt írhatták, hogy ADD EAX,[FIZETES]. Ez sokkal olvashatóbb forma, másrészt amennyiben a fizetés értékét mégsem az 1034-es memóriacímen kellett tárolni (változás), akkor egyetlen helyen kellett csak átírni – a program eleji táblázatban.

Ez az egyszerű és logikus megfontolás indította el a 'változó' fogalmának fejlődését. A fenti esetben a FIZETES volt az azonosító, melyhez a programozó konstans formájában rendelt hozzá memóriacímet – helyet a memóriában.

Még fejlettebb assembler fordítók esetén a fenti táblázat az alábbi formában is felírható volt:

| | |
|-----------|-------------|
| FIZETES | = 1034 |
| ELETKOR | = FIZETES+4 |
| NYUGDIJAS | = ELETKOR+2 |

Ebben az esetben már kiolvasható volt a táblázatból, hogy a FIZETES névvel jelölt memóriaterület 4 byte-os tároló hely volt, hiszen ekkora memória-szakasz fenntartása után a következő memóriaterület ('ELETKOR' elnevezéssel) 4 byte-tal távolabbi ponton kezdődik.

Ez a változat azért is jobb, mert segíti, hogy elkerüljük két tárolóterület átfedését a memóriában (átlapolás). Megakadályozni nem tudja, hiszen amennyiben a fizetés tárolási igénye 8 byte lenne, akkor a fenti esetben az életkor még átlógna a fizetés

¹ 128*1024*1024-1

tárlóhelyének utolsó 4 byte-jára. Ennek persze a programozó az oka, aki rosszul írta fel a táblázatot. Ha azonban a fizetés tárolására 2 byte is elég lenne, akkor 2 byte-nyi felhasználatlan terület keletkezne a memóriában (ami szintén nem szerencsés).

A fenti problémákon túl az assembler azt sem tudta ellenőrizni, hogy a tárlórekeszt megfelelően kezeli-e a programozó a későbbiekben:

| | |
|--------------------|-----------------------------------|
| MOV EAX, [FIZETES] | // EAX 32 bites regiszter, 4 byte |
| MOV AX, [FIZETES] | // AX 16 bites regiszter, 2 byte |
| MOV AL, [FIZETES] | // AL 8 bites regiszter, 1 byte |

A fenti utasítások mindegyike elfogadható az assembly nyelv szintaktikai szabályai szerint. Az első esetben a megadott memóriacímről 4 byte-nyi adat kerül be az EAX nevű regiszterbe. Második esetben csak 2 byte, harmadik esetben csak 1 byte. A memória-beolvasást ugyanis a fogadó regiszter mérete befolyásolja. Amennyiben a fizetés 4 byte-on kerül tárolásra, úgy a második és harmadik utasítás nagy valószínűséggel hibás, hiszen miért olvasnánk be a fizetést tartalmazó számsorozatnak csak az egyik felét?

Az ilyen jellegű hibákra azonban az assembler nem tudott figyelmeztetni, mivel számára a FIZETES csak egy memóriacím volt. A helyes kezelésre a programozónak kellett ügyelnie. Az assembler nem is tudta volna ellenőrizni a kódot ebből a szempontból, hiszen nem volt információja arról, hogy mit is ért a programozó *fizetés* alatt.

Ezt a plusz információt hívják a programozási nyelvben **típusnak**.

A típus sok mindent meghatároz. Többek között meghatározza az adott érték tárolási méretét a memóriában. A típus ismeretében a fenti táblázat felírható az alábbi formában:

| |
|----------------|
| DWORD FIZETES |
| WORD ELET KOR |
| BYTE NYUGDIJAS |

A fenti táblázat szerint a FIZETES egy dupla-szó (DWORD), aminek helyigénye 4 byte. Az ELET KOR egy szimpla szó (WORD), 2 byte helyigényű. A NYUGDIJAS egy byte, aminek helyigénye (mint a neve is mutatja) 1 byte.

Amikor az ASSEMBLER-ek már a fenti táblázatot is kezelték, akkor már képesek voltak a tárterület-címeket automatikusan kiosztani. Az első azonosító címéhez képest a primitív típusnevek (dword, word, byte, ...) tárolási igényét ismervén automatikusan növelték a memóriacímeket, és adtak értéket az azonosítóknak. Ezek az értékek továbbra is memóriacímek voltak, de az automatikus kiosztás miatt a memóriaterületek átlapolásának esélye még kisebbre zsugorodott.

A fenti primitív típusnevek még nem jelölték az azonosítók tényleges típusát. A fordító mindössze a tárigény-szükséglet kiszámítására használta fel. A hibás kezelés lehetőségét még mindig megengedte, vagyis egy 4 byte-os helyigényű értéknek még mindig szabad volt az egyik felét beolvasni, és dolgozni vele.

Nem voltak 'finom' típusnevek. Nem volt 'char', 'bool', 'short int', 'unsigned short int' típusok. Ezek mindegyike 1 byte tárolási igényű, csak ezen 1 byte-on hordozott információ jelentésében térnek el. Mivel a jelentést a fordító még nem kezelte, csak egy olyan típusnév volt, amelynek 1 byte volt a tárolási igénye.

Ennek megfelelően ezt még nem tekinthetjük tényleges típusnévnek, mindössze tárolási-igény névnek.

A változó más komponenseinek (élettartam, hatáskör) kezelése is hiányzott az assembly nyelvből, és az assembler programokból.

Vezérlési szerkezetek az assembly nyelvben

Az assembly nyelv másik szegényes tulajdonsága a vezérlési szerkezetek hiánya. Lényegében csak az alábbi vezérlési szerkezetek találhatók meg:

- **Szekvencia:** a program utasításainak végrehajtása a memóriabeli sorrend alapján történik.
- **Feltétlen vezérlésátadás** (ugró utasítás): a program folytatása egy másik memóriabeli pontra tevődik át, majd attól a ponttól kezdve a végrehajtás újra szekvenciális.
- **Feltételes vezérlésátadás** (feltételes ugró utasítás): mint az egyszerű ugró utasítás, de az ugrást csak akkor kell végrehajtani, ha az előírt feltétel teljesül.
- **Visszatérés** az ugró utasítást követő utasításra (azon helyre, ahonnan az ugrás történt).

Ezekből kellett összerakni a programot. Egy egyszerű elágazást ennek megfelelően az alábbi módon kellett kódolni:

| | | |
|--|-----|----------|
| HA feltétel AKKOR | ... | feltétel |
| kiértékelése ... | | |
| Ut1 | | |
| UGRÁS_HA <i>feltétel</i> HAMIS CIMKE1-re | | |
| Ut2 | | |
| UT1 | | |
| KÜLÖNBEN | | UT2 |
| Ut3 | | |
| UGRÁS CIMKE2-re | | |
| Ut4 | | |
| @CIMKE1: | | |
| HVÉGE | | UT3 |
| <i>folytatás</i> | | UT4 |
| @CIMKE2: | | |
| <i>folytatás</i> | | |

A fentiből talán sejthető, hogy az assembly nyelvű programból kibogozni, hogy itt valójában feltételes elágazás történt – nem egyszerű. Hasonló problémákkal jár a

ciklusok megtervezése és kódolása is – különösen az egymásba ágyazott ciklusok esete.

Eljáráshívás az assembly nyelvben

Az assembly nyelv elvileg ad lehetőséget eljárashívásra is az alábbi formában:

```
ELJARASHIVAS    kiiras  
  
    @kiiras:  
    ...  
    VISSZATÉRÉS_A_HÍVÁST_KÖ    VETŐ_UTASÍTÁSRA
```

A 'kiiras' itt valójában címke (programsort jelölő név), de megfelel az eljárásnév primitív fogalmának. A nevesített címkék egy jó névválasztással, nagyon sokat könnyítene a programkód olvashatóságán.

A probléma nem is itt rejtőzik, hanem hogy az eljárásnak hogyan adunk át paramétereket? Illetve, ha ez nem eljárás, hanem függvény, akkor hol kapjuk meg a visszatérési értéket? Illetve honnan tudjuk, milyen típusú adattal, értékkel tér vissza az adott függvény?

A fenti kérdésekre a válaszokat maga az assembly nyelv nem tartalmazza. Paraméterek átadására például több mód is van – csakúgy mint a függvények visszatérési értékének visszaadására. Ezeket a lehetőségeket maga a gépi kód tartalmazza, és az assembly nyelv értelemszerűen átvette.

A programozók szívesen fejlesztettek általános, újra felhasználható eljárásokat és függvényeket, melyek segítségével a programok fejlesztési ideje, és a tesztelési ideje is alaposan lerövidült. De ezek egymással való megosztása a fenti akadályok miatt nehézkes volt. Egy másik programozó által fejlesztett, nem megfelelően dokumentált eljárás felhasználása a kódban néha több energia felemésztésével járt, mint újra megírni a szóban forgó eljárást.

Több modulból álló programok az assembly nyelvben

A több modulra tagolás azért volt fontos, mert az assembly programok általában nagyon hosszúak voltak, másrészt az akkori számítógépek még nagyon lassúak voltak. Egy hosszú forráskód fordítása nagyon sok időbe került. Ugyanakkor a program jelentős része a fejlesztés közben már elkészült, azt a programozó nem módosította – mivel a program már másik pontján járt a fejlesztés. Ezt a szakaszt újra és újra lefordítani az assemblerrel felesleges idő és energiapocsékolás volt.

Ezért bevezették a több modulból történő fordítást. Ekkor az assembler a fordításhoz egy listát kapott, hogy mely forráskód-fileokból tevődik össze a project. Az assembler a forráskód-modulokat egyenként fordította le egy-egy **tárgykód** (object) állományba. Amennyiben a forráskódon nem történt módosítás, úgy azt az assembler egy gyors ellenőrzéssel észrevette, és nem generálta újra a hozzá tartozó object kódot. Így

csak azon forráskódok fordítása történt meg, melyek változtak az utolsó fordítás óta. Miután az assembler végzett a forráskódok fordításával, egy másik, speciális feladatot végző program következett, a **szerkesztő** program (linker). A linker a sok apró kis tárgykód alapján készítette el a működőképes programot.

Ez jelentősen meggyorsította a fordítást egyéb hátrányok nélkül. Sőt, a programozók így az újra felhasználható, általános eljárásaikat külön kis kód-gyűjteményben tárolták, és újabb project kezdése esetén eleve hozzácsatolták a project forráskód-listájához.

Ez tovább erősítette a vágyat az általános célú eljárások írására, és egymás közötti megosztásra. De az assembly nyelv ez irányú képességeinek hiánya ebben továbbra is komoly gátat jelentett.

A fentiek bizonyítják, hogy az assembly nyelv sok olyan lehetőséget rejtett magában, amely miatt megérdemli a külön generációs sorszámot. Ugyanakkor a nyelvi korlátok gátolták a programozási stílus fejlődését.

Harmadik generációs programozási nyelvek: PROCEDURÁLIS NYELVEK

Az assembly nyelv hiányosságainak kiküszöbölésére születtek a harmadik generációs nyelvek.

Az eljárásorientált (procedurális) nyelvek sok szempontból elvi, szemléletbeli váltást követeltek meg az assembly programozóktól. A frissen felnövekvő programozó nemzedék, akik nem hordoztak magukban rossz szokásokat és hibás beidegződéseket – azonnal és gyorsan átvették ezeket a szemléletbeli előírásokat.

Az első nagyon fontos változás – az eljárás fogalmának bevezetése.

Az eljárás (és függvény) nyelvi elemmé vált. Az eljárásoknak neve volt, és rögzített paraméterezése (**formális paraméterlista**). Ez leírta, hogy az eljárás meghívása során milyen adatokat, értékeket kell az eljárás számára átadni. Ezen túl a nyelv rögzítette az átadás módját is. Ezzel elhárult az általános célú eljárások írásának legjelentősebb akadálya. Ugyanakkor, hasonló jelentős lépésként a fordítóprogram az eljárás hívásakor ellenőrizte, hogy a megadott összes adatot átadjuk-e az eljárásnak (**aktuális paraméterlista**). Ez újabb fontos mérföldkő a fordítóprogram intelligenciájának fejlődésében.

A másik fontos változás a változó fogalmának finomodása:

A változónak van:

- Neve (azonosítója), ezzel lehet a kódban hivatkozni rá.

- Típusa (mely meghatározza a memóriabeli helyigényét, és tárolási (kódolási) módját).
- A típus ezen túl meghatározza az adott nevű változóval elvégezhető műveletek körét is (numerikus típusú változóval végezhető az osztás, szorzás, kivonás, összeadás, ...), míg logikai típusúval a logikai műveletek (és, vagy, xor, ...).
- A kifejezésekben szereplő adatok és változók típusait a fordítóprogram elemzi, összeveti, és ellenőrzi a kifejezés típushelyességét.
- A programban lehetetlenné vált a változó tárhelyének részleges kezelése (a változó értékét reprezentáló byte-ok csak egy részének kiolvasása, módosítása). Ezzel is nagyon sok tipikus programozó hiba kiszűrhetővé vált.
- A változókat általában kötelezően **deklarálni** kellett. Ennek során a programozó *bejelentette* a fordítóprogram számára érthető formában, hogy az adott azonosító (változónév) alatt mit ért (milyen típust). A deklaráció helye további információkat jelent a fordítóprogram számára – meghatározza a változó élettartamát és hatáskörét is.

A változó élettartama:

- **statikus**: a változó a program indulásának pillanatától a futás végéig a változó folyamatosan létezik, és változatlan helyen lesz a memóriában.
- **dinamikus**: a változó a program futása közben jön létre és szűnik meg (akár többször is).

A statikus változók fontosak az adatok megőrzése szempontjából. A fontos, sokáig szükséges adatokat statikus változókban tároljuk. A dinamikus változók a memóriaterület gazdaságos felhasználása szempontjából fontosak – a változó csak addig legyen a memóriában, amíg fontos. Amint feleslegessé vált – megszűnik, és a helyére később más változó kerülhet.

A változó hatásköre:

- **globális**: a program szövegében több helyen (több eljárásban is) elérhető, felhasználható.
- **lokális**: a program szövegében a változó felhasználása helyhez kötött, csak egy meghatározott programrészben (körülhatárolt szegmensben) használható fel.

A globális változók minden esetben statikusak is. A dinamikus változók pedig általában lokálisak. A dinamikus változó létrehozása és megszűnése ezen lokális területhez kötődik – amikor a program végrehajtása eléri ezt a pontot, belép erre a területre, akkor a változó automatikusan létrejön. Amikor a program végrehajtása elhagyja ezt a területet, akkor a változó automatikusan megszűnik, helye felszabadul a memóriában.

A harmadik fontos változás – a típusrendszer bővíthetősége

A magas szintű programozási nyelvek eleve adott típusokkal készültek. A nyelvi alaptípusokból további (felhasználó által definiált) típusokat lehet készíteni. Ezen típusok a meglévő típusok szűkítései (felsorolás típus, résztartomány-típus), vagy összetett algebrai adatszerkezetek is lehetnek (pl. struktúrák, vektorok, listák, ...).

A negyedik fontos változás – a vezérlési szerkezetek bevezetése

Az assembly nyelv ugróutasításából megszervezhető vezérlési szerkezetek körét csökkentették, és rögzítették azokat:

- **szekvencia:** az utasításokat a forráskódban rögzített sorrendben kell végrehajtani.
- **szelekció:** feltételes elágazás (pl. a *ha ... akkor ... különben ...* szerkezetek).
- **iteráció:** adott programrész ismétlése (előírt lépésszámú ciklus, logikai feltételhez kötött ciklusok, halmaz alapú ciklusok, ...).

A vezérlési szerkezetek e formája áttekinthető, egyszerű, könnyen olvasható kódot eredményez. Mills bizonyította, hogy minden algoritmus kódolható a fenti három vezérlési szerkezet használatával, így az ugró utasítások szükségtelenné váltak.

Természetesen, amikor a fordítóprogram a gépi kódú változatot generálja, akkor a fenti szerkezeteket ugró utasítások formájában valósítja meg – hiszen a gépi kódban csak ezek szerepelnek.

Az ötödik fontos változás – a hardware függetlenség.

A procedurális nyelvek már nem processzor függők. A fordítóprogram ismeri az adott számítógép processzorának gépi kódját – és a procedurális nyelven megírt magas szintű kódot az adott gépi kódra fordítja. Amennyiben a programot más platformon is szeretnénk futtatni, úgy a magas szintű forráskódot az adott számítógépre írt fordítóprogrammal újra kell fordítani – a forráskód bármilyen változtatása nélkül.

A memóriaterület kiosztását a fordítóprogram végzi a változó-deklarációk alapján. A program egy adott pontján mindig egyértelműen megadható a változó hatáskörök figyelembevételével, hogy mely változók érhetők el, és melyek nem. A dinamikus változók létrehozását és megszüntetését a fordítóprogram által generált kód automatikusan végzi. A típusokhoz tartozó tárhely-igényt a fordítóprogram kezeli, kizárt a memória-átlapolás és nem keletkeznek fel nem használt memóriaterületek. Emiatt nagyon sok lehetséges programozási hiba egyszerűen megszűnt létezni.

Fennmaradt azonban egy nagyon fontos probléma: a felhasználó által definiált típusokhoz nem lehet operátorokat definiálni, emiatt kifejezésekben nem lehet az új típusokat felhasználni. Vagyis a felhasználói adattípus *kevesebbet* ér, mint a *'gyári'*, eleve létező elemi típus. Ezen a szinten a nyelv fejlődése nem haladhatja meg ezt a pontot.

Három-és-fél generációs programozási nyelvek: OBJEKTUM ORIENTÁLT NYELVEK

Az objektum orientált programozási nyelvek (OOP nyelv) ezen a ponton jelentenek fejlődést. A felhasználó sokkal egyszerűbben és szabadabban készítheti el a saját típusait. Meglévő típusok továbbfejlesztésével (öröklődés) kevés munkával készíthet új típusokat. A saját típusaihoz (általában) készíthet operátorokat is (melyeknek jelentését természetesen le kell programozni). Ezek után a saját típus szinte minden szempontból egyenragúvá válik a nyelvi alaptípusokkal. A saját típusokhoz nem csak operátorokat rendelhet, hanem megadhat függvényeket és eljárásokat is, amelyek az adott típusú adatokkal végeznek valamilyen műveletet. Mivel ezen függvények és operátorok az adott típushoz tartoznak, a típus részeinek tekintendők. Az egy típusba sorolt adattároló változók (mezők), a hozzájuk tartozó műveletek és operátorok csoportját (halmazát) osztálynak nevezzük.

Egy OOP nyelvben tehát szintén megtalálhatóak az eljárások és függvények, illetve a paraméterek, változók. A vezérlési szerkezetek is a megszokott három formára épülnek (szekvencia, szelekció, iteráció). Ezért az OOP nyelvek inkább csak szemléletmódban mások (melyik eljárást és függvényt hova írjuk meg), mint kódolási technikákban. Ezért az OOP nyelveket nem tekintik külön generációnak.

Negyedik generációs programozási nyelvek: SPECIALIZÁLT NYELVEK

A negyedik generációs nyelvek speciális feladatkörre készült nyelvek. Ezen nyelvek jellemzője, hogy nagyon kevés nyelvi elemmel dolgoznak, és nagyon egyszerű, szinte mondatszerűen olvasható utasítások fogalmazhatók meg. Erre jó példa az SQL nyelv, amely elsősorban adatbázis-kezelésre van felkészítve.

Ötödik generációs programozási nyelvek: MESTERSÉGES INTELLIGENCIA NYELVEK

A mesterséges intelligencia programozási nyelvekkel elvileg az emberi gondolkodás leírása történne meg, gyakorlatilag e nyelvek kutatása, fejlesztése még folyamatban van.

A programozási nyelvek csoportosítása

A programozási nyelveket más szempontból vizsgálva egy másik csoportosítás fedezhető fel:

Imperatív (procedurális) nyelvek:

Ezen nyelvek közös jellemzője, hogy a program fejlesztése értékadó utasítások megfelelő sorrendben történő kiadására koncentrálódik. Az értékadó utasítás baloldalán egy változó áll, a jobb oldalán pedig egy megfelelő típusú kifejezés. A szelekció (elágazás) csak azt a célt szolgálja, hogy bizonyos értékadó utasításokat csak adott esetben kell végrehajtani. A ciklusok pedig azért vannak, hogy az értékadó utasításokat többször is végrehajthassunk. Az értékadó utasítások során részeredményeket számolunk ki – végül megkapjuk a keresett végeredményt.

Applikatív (funkcionális) nyelvek:

A funkcionális nyelveken a kiszámolandó kifejezést adjuk meg, megfelelő mennyiségű bemenő adattal. A programozó munkája a kifejezés kiszámításának leírására szolgál. A program futása közben egyszerűen kiszámítja a szóban forgó kifejezést.

Egy funkcionális nyelvben nincs változó, általában nincs ciklus (helyette rekurzió van). Értékadó utasítás sincs, csak függvény visszatérési értékének megadása létezik. A funkcionális nyelvek tipikus felhasználási területének a természettudományos alkalmazások tekinthetők.

Logikai nyelvek:

Az ilyen jellegű nyelveken tényeket fogalmazunk meg, és logikai állításokat írunk le. A program ezen kívül egyetlen logikai kifejezést is tartalmaz, melynek értékét a programozási nyelv a beépített kiértékelő algoritmusával segítségével, a tények és szabályok figyelembevételével meghatározza.

A logikai nyelvek tipikus felhasználási területe a szakértői rendszerek létrehozásához kapcsolódik.

Objektum-orientált nyelvek:

Az OOP nyelveken a program működése egymással kölcsönhatásban álló objektumok működését jelenti. Az objektumok egymás műveleteit aktiválják, melyeket interface-ek írnak le. Ha egy művelet nem végrehajtható, akkor az adott objektum a hívó félnek szabványos módon (kivételkezelés) jelzi a probléma pontos okát.

Futtató rendszerek

A fordító programok által generált tárgykódokat a szerkesztő program önti végleges formába. Az elkészült futtatható programot ezen időpont után az operációs rendszer kezeli, és futtatja.

A futtatás háromféleképpen történhet:

Direkt futtatás: a generált kód az adott mikroprocesszor gépi kódú utasításait tartalmazza. Ennek megfelelően az utasítássorozatot az operációs rendszer egyszerűen átadja a mikroprocesszornak és megadja a program kezdő pontját. A processzor e pillanattól kezdve önállóan végrehajtja a következő utasítást, követi az ugrási pontokat, írja és olvassa a memória hivatkozott területeit anélkül, hogy tudná valójában mit csinál a program az adott ponton. A processzor feltétlenül megbízik a programkódban, vita nélkül engedelmeskedik és végrehajtja a soron következő utasítást.

Ezen módszer előnye a maximális végrehajtási sebesség. Ugyanakkor jegyezzük meg, hogy a gépi kód szintjén nincs típusfogalom, és szinte lehetetlen eldönteni, hogy az adott utasítás a program feladata szempontjából helyes-e, szükséges-e, hibás-e.

Mivel a memóriában (a nagy byte-halmazban) az adatok területén is byte-ok találhatóak, ezért elvileg elképzelhető, hogy a program vezérlése egy hibás ugró utasításnak köszönhetően áttér egy ilyen területre, és az adatokat gépi kódú utasításokként próbálja értelmezni. Ez persze valószínűleg nem fog menni. Vagy valamely utasítás paraméterezése lesz értelmezhetetlen, vagy egy olyan kódot fog találni a processzor, amely nem értelmezhető utasításkódnak. Ekkor a processzor hibás működés állapotára tér át, amely a számítógép leállításához (lefagyás) is vezethet. A mai processzorok már védekeznek ez ellen, és ilyen esemény detektálásakor speciális hibakezelő rutinok futtatására térnek át.

A hiba bekövetkezése ellen védelmet jelent, hogy a mai programok elszeparált területeken tárolják a programkódot, és az adatokat. A hibás ugróutasítás emiatt felfedezhető, hiszen egy adatterület belsejébe irányul. A processzor már ekkor leállítja a program futását, és áttér a hibakezelő állapotra.

Másik védelmi mechanizmus szerint a kód-terület nem írható, csak olvasható. Így a processzor képes felfedezni a hibás értékadó utasításokat, melyek egy művelet eredményeképpen kapott értéket egy olyan memóriaterületre írná be, ahol utasításkódok vannak. Ez meggátolja a programkód futás közbeni módosítását (önmódosító kód) amely egy időben igen elterjedt volt. Ma már e módszert inkább csak a vírusok és egyéb kártékony programok használják.

Interpreterrel futtatás: a fordítóprogram ekkor nem generál közvetlenül végrehajtható gépi kódú utasításokat, hanem egy köztes kódot, ahol az eredeti programozási nyelv utasításai vannak számkódokká átfordítva, a paraméterei is már feldolgozott, egyszerűsített formában kódoltak. Ezt a 'tárgykódot' egy futtató rendszer (az interpreter) futás közben utasításonként elemzi, és hajtja végre az előírt szabályok szerint.

Az interpreteres rendszerekben a futtató rendszer a lefordított kódot még végrehajtás előtt elemzi, szükség esetén az utolsó pillanatban korrigálja, pontosítja azt (pl. változóhivatkozások). Ilyen rendszerekben az is megoldható, hogy a változókat nem kell deklarálni explicit módon a program szövegében. A futtató rendszer 'kitalálja' az

adott programsorhoz érve, hogy milyen változókra van szüksége, és pontosan e pillanatban hozza csak létre. Ezen kívül az is megoldható, hogy a változó típusa futás közben derüljön csak ki, vagy akár menet közben többször meg is változzon. A futtató rendszer az adott utasításhoz érve ellenőrzi hogy ott éppen milyen változók szükségesek, ellenőrzi hogy azok pillanatnyi típusa megfelelő-e. Ha nem megfelelő, akkor vagy leáll hibaüzenettel, vagy törli a nem megfelelő típusú változót és létrehoz egy megfelelő típusút helyette.

Az interpreteres rendszerek sokkal rugalmasabbak. Egy adott utasítás végrehajtása előtt ellenőrizheti a szükséges feltételek teljesülését, sőt, akár korrigálhatja is a feltételeket, hogy megfeleljenek az utasításnak. Ezen rendszerek futtatása sokkal biztonságosabb. A felügyelő interpreter időben leállíthatja a futó programot, amennyiben azt nem találja megfelelőnek.

Az interpreteres rendszerek vitathatatlan hátránya, hogy egyrészt lassú a futtatás, másrészt a generált 'tárgykód' az adott programozási nyelv szintaktikai lenyomatát hordozza, nem univerzális. Egy ilyen 'tárgykód'-ból az eredeti forráskód általában szinte veszteség nélkül helyreállítható, még a változónevek és az eljárásnevek is visszaállíthatóak. Egy adott nyelv interpretere (pl. a BASIC interpreter) nem képes futtatni más interpreteres nyelv fordítóprogramja által generált 'tárgykódot'.

Virtuális futtatás: ez az interpreteres elv módosítását jelenti. A fordító nem direktben futtatható gépi kódú programot generál, hanem egy nem létező (virtuális) processzor virtuális gépi kódjára generál programot. Ezen 'gépi kód' eltérhet a jelenlegi gépi kódtól, sokkal magasabb szintű utasításokat tartalmaz, és sokkal több típust ismer.

Ezen nyelv interpreteres futtató rendszere (processzor-szimulátor, virtuális gép) sokkal egyszerűbb, hiszen a generált kód alacsonyabb szintű utasításokat tartalmaz, mint általában az interpreteres rendszerekben. A generált kód már nem feltétlenül hasonlít, nem feltétlenül hordozza az eredeti programozási nyelv szintaktikai lenyomatát, belőle az eredeti forráskód csak veszteségesen állítható helyre (bár sokkal kevesebb veszteség árán, mint a gépi kódból).

A virtuális futtatás előnye, hogy amennyiben egy programozási nyelv fordítóprogramja képes e virtuális gépi kódú program generálására, úgy a futtatáshoz már felhasználható a kész futtató rendszer. Valamint ezen a virtuális nyelven generált kód más-más processzoron is futtatható lesz, amennyiben az adott processzora is létezik a futtató rendszer.

A legismertebb ilyen nyelv a JAVA, ahol a futtató rendszert Java Virtual Machine-nak (JVM-nek) hívják.

Bevezetés A Microsoft®.NET

Hernyák Zoltán

A Microsoft.NET-et (továbbiakban *dotNet*) sokféleképpen lehet definiálni. A Microsoft honlapján például az alábbi meghatározás szerepel: „ez egy software technológiák halmaza, amely információkat, embereket, rendszereket és eszközöket kapcsol össze. Ez az új generációs technológia a Web szolgáltatásokon alapul – kis alkalmazásokon, amelyek képesek kapcsolatba lépni egymással csakúgy, mint nagyobb méretű alkalmazásokkal az Internet-en keresztül.”²

Másik lehetséges meghatározás szerint a dotNet egy programfejlesztési környezet, mely számtalan hasznos szolgáltatással segíti a programozók mindennapi munkáját.

Amikor egy programozó egy alkalmazás fejlesztésébe kezd, sok más dolog mellett ismernie kell, és figyelembe kell vennie azt a környezetet, amelyben az alkalmazása futni fog. A környezet egyik legfontosabb jellemzője az operációs rendszer. A dotNet egyféle szemszögből nézve az operációs rendszert helyettesíti elrejtve, eltakarván a tényleges operációs rendszert a fejlesztő elől.

Az operációs rendszert egy programozó teljesen más szempontból nézi, mint egy felhasználó. A felhasználó számára az operációs rendszer az, amibe be kell jelentkezni, alkönyvtárakat és file-okat kezel, parancsokat lehet rajta keresztül kiadni akár karakteres felületen, akár egér segítségével. Ezzel szemben a programozó az operációs rendszert elsősorban az API szempontjából nézi: melyek azok a funkciók, feladatok, amelyeket az operációs rendszer elvégez a program feladatai közül, és melyek azok, amelyeket nem.

Az API – Application Programming Interface – írja le egy adott operációs rendszer esetén, hogy melyek azok a szolgáltatások, amelyeket az operációs rendszer eleve tartalmaz, és amelyeket a programozó a fejlesztői munka során felhasználhat. Ezen szolgáltatásokat pl. a Windows a rendszer-könyvtáraiban megtalálható *DLL* (Dynamic Link Library) file-okban tárolja. Minden DLL több függvényt és eljárást tartalmaz, melyekre hivatkozhatunk a programjainkban is. E függvények és eljárások összességét nevezzük API-nak.

Egy operációs rendszer API leírásában szerepelnek a következő információk:

- a függvény melyik DLL-ben van benne,
- mi a függvény neve,
- mik a paraméterei,
- a függvénynek mi a feladata (mit csinál),
- mik a lehetséges visszatérési értékek,
- milyen módon jelzi a függvény a hibákat,
- stb...

A Microsoft-os világban az első ilyen API-t az első, általuk előállított operációs rendszer tartalmazta. Ez volt a DOS. Ez nem tartalmazott túl sok elérhető szolgáltatást – lévén a DOS egy egyfelhasználós, egyfeladatos operációs rendszer.

² <http://www.microsoft.com/net/basics/whatis.asp>

DOS-API \Rightarrow Win16-API \Rightarrow Win32-API \Rightarrow dotNet-API

A Win16-API sokkal több szolgáltatást tartalmazott, lévén hogy az már grafikus operációs rendszerrel, a Windows 3.1-el került be a köztudatba. Ez már nem csak a grafikus felület miatt tartalmazott több felhasználható szolgáltatást, hanem mivel ez egy többfeladatos operációs rendszer – teljesen új területeket is megcélzott.

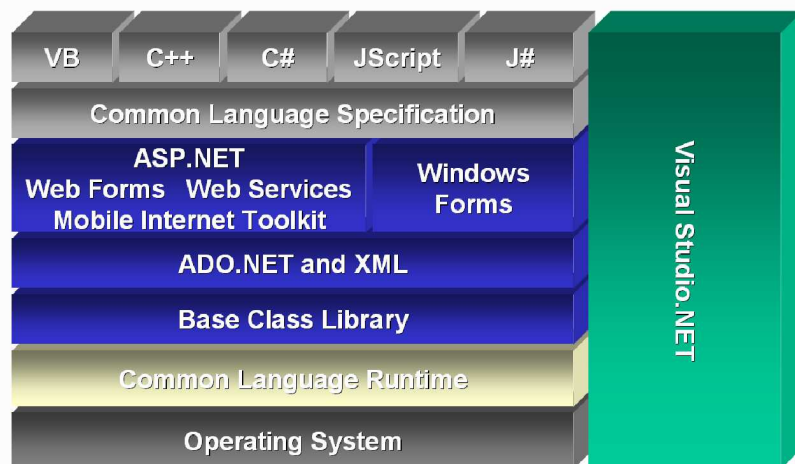
A Win32-API a fejlettebb Windows operációs rendszerekben jelent meg, mint pl. a Windows '95. Ebben javítottak a többszálú programok kezelésén, és bevezettek bizonyos jogosultsági módszerek kezelését is (és még számtalan mást is).

A dotNet ezen szempontból egy új API-nak tekinthető. Ez olyannyira igaz, hogy egy dotNet környezetben a programozónak semmilyen más API-t (elvileg) nem kell ismernie. A *dotNet* (elvileg) ma már több különböző operációs rendszeren is képes működni – de a programozónak ezt nem kell feltétlenül tudnia – hiszen ő már nem használja az adott operációs rendszer API-ját, csak a *dotNet*-et által definiált függvényeket és eljárásokat.

Tekintsük át a Microsoft.NET felépítését:

Microsoft .NET rendszertechnika

Univerzális, „managed” kód fejlesztési platform



Legalsó szinten az operációs rendszer található. Mivel egy jól megírt, biztonságos operációs rendszer nem engedi meg, hogy a felügyelete alatt futó programok önállóan kezeljék a számítógép hardware elemeit, ezért a programok csakis az operációs rendszeren keresztül kommunikálhatnak egymással, használhatják fel az erőforrásokat (hálózat, file-rendszer, memória, ...) – vagyis az operációs rendszer API-n keresztül. Ez a réteg egy jól megírt operációs rendszer esetén nem kerülhető meg.

A következő szinten van a Common Language Runtime – a közös nyelvi futtató rendszer. Ez az egyik legérdekesebb réteg. A CLR lényegében egy processzor-emulátor, ugyanis a dotNet-es programok egy virtuális mikroprocesszor virtuális gépi kódú utasításkészletére van fordítva. Ennek futtatását végzi a CLR. A futtatás maga interpreter módban történik, ugyanakkor a dotNet fejlett futtató rendszere a virtuális gépi kódú utasításokat futás közben az aktuális számítógép aktuális mikroprocesszorának utasításkészletére fordítja le, és hajtja végre.

Az első ilyen előny, hogy a szóban forgó virtuális gépi kódú nyelv típusos, ezért a programkód futtatása közben a memória-hozzáféréseket ellenőrizni lehet – így meggátolható a helytelen, hibás viselkedés. Másrészt az utasítások végrehajtása előtt ellenőrizni lehet a jogosultságot is – vagyis hogy az adott felhasználó és adott program esetén szabad-e végrehajtani az adott utasítást – pl. hálózati kapcsolatot létesíteni, vagy file-ba írni. Ugyanakkor meg kell jegyezni, hogy egy JIT Compiler (JIT=Just In Time) gondoskodik arról, hogy a végrehajtás megfelelő hatékonysággal történjen. Ezért egy *dotNet*-es program hatékonysága, futási sebessége elhanyagolhatóan kisebb, mint a natív kódú programoké.

A második réteg tartalmazza a *dotNet* API nagyobb részét. A Base Class Library tartalmazza azokat a szolgáltatásokat, amelyeket egy *dotNet*-es programozó felhasználhat a fejlesztés közben. Lényeges különbség a megelőző API-kal szemben, hogy ez már nem csak eljárások és függvények halmaza, hanem struktúrált, névterekbe és osztályokba szervezett a sok ezer hívható szolgáltatás. Ez nem csak áttekinthetőbb, de hatékonyabb felhasználhatóságot jelent egy, az objektum orientált programozásban jártas programozó számára.

A következő réteg, az ADO.NET és XML a 21. századi adatelérési technológiákat tartalmazza. Ezen technikák ma már egyre hangsúlyosabbak, mivel a mai alkalmazások egyre gyakrabban használják fel e professzionális és szabványos technikákat adatok tárolására, elérésére, módosítására. A rétegek segítségével a programok a háttértárolókon képesek adatokat tárolni, onnan induláskor azokat visszaolvasni.

A következő réteg kétfelé válik – aszerint hogy az alkalmazás felhasználói felületét web-es, vagy hagyományos interaktív grafikus felületen valósítjuk meg. A Windows Forms tartalmazza azon API készletet, melyek segítségével grafikus felületű ablakos, interaktív alkalmazásokat készíthetünk. A másik lehetséges választás a WEB-es felületű, valamilyen browser-ban futó nem kifejezetten interaktív program írása. Ezek futtatásához szükséges valamilyen web szerver, a kliens oldalon pedig valamilyen internetes tallózó program, pl. Internet Explorer vagy Mozilla.

A következő réteg – a Common Language Specification – definiálja azokat a jellemzőket, melyeket a különböző programozási nyelvek a fejlődésük során történelmi okokból különböző módon értelmeztek. Ez a réteg írja le az alaptípusok méretét, tárolási módját - beleértve a string-eket - a tömböket. Fontos eltérés például, hogy a C alapú nyelvekben a vektorok indexelése mindig 0-val kezdődik, míg más

nyelvekben ez nem ennyire kötött. A réteg nem kevés vita után elsimította ezeket, a különbségeket.

A *CLS* réteg fölött helyezkednek el a különböző programozási nyelvek és a fordítóprogramjaik. A *dotNet* igazából nem kötött egyetlen programozási nyelvhez sem. A *dotNet* nem épül egyetlen nyelvre sem rá, így nyelvfüggetlen. Elvileg bármilyen programozási nyelven lehet *dotNet*-es programokat fejleszteni, amelyekhez létezik olyan fordítóprogram, amely ismeri a *CLS* követelményeit, és képes a *CLR* virtuális gépi kódjára fordítani a forráskódot.

A fenti két követelmény betartásának van egy nagyon érdekes következménye: elvileg lehetőség van arra, hogy egy nagyobb projekt esetén a projekt egyik felét egyik programozási nyelven írjuk meg, a másik felét pedig egy másikon. Mivel mindkét nyelv fordítóprogramja a közös rendszerre fordítja le a saját forráskódját – így a fordítás során az eredeti forráskód nyelvi különbségei eltűnnek, és a különböző nyelven megírt részek zökkenőmentesen tudnak egymással kommunikálni.

A Microsoft a C++, illetve a Basic nyelvekhez készítette el a *dotNet*-es fordítóprogramot, valamint tervezett egy új nyelvet is, melyet C#-nak (ejtsd szí-sharp) nevezett el. Az új nyelv sok más nyelv jó tulajdonságait ötvözi, és nem hordozza magával a *kompatibilitás megőrzésének* terhét. Tiszta szintaktikájával nagyon jól használható eszköz azok számára, akik most ismerkedek a *dotNet* világával.

Ugyanakkor nem csak a Microsoft készíti fordítóprogramokat erre a környezetre. Egyik legismertebb, nemrégiben csatlakozott nyelv a Delphi.

A fentiekén túl van még egy olyan szolgáltatása a *dotNet* rendszernek, melynek jelentőségét nem lehet eléggé hangsúlyozni: automatikus szemétyűjtés. Ez a szolgáltatás a memória-kezeléssel kapcsolatos, és a program által lefoglalt, de már nem használt memória felszabadítását végzi. Ezt egy Garbage Collector nevű programrész végzi, amely folyamatosan felügyeli a futó programokat. Ennek tudatában a programozónak csak arra kell ügyelni, hogy memóriát igényeljen, ha arra szüksége van. A memória felszabadításáról nem kell intézkednie, az automatikusan bekövetkezik.

Ha megpróbálnánk összegezni, hogy miért jó *dotNet*-ben programozni, az alábbi főbb szempontokat hozhatjuk fel:

- az alkalmazás operációs rendszertől független lesz
- független lesz az adott számítógép hardware-től is, gondolván itt elsősorban a mikroprocesszorra
- nem kell új programozási nyelvet megtanulnunk ha már ismerünk valamilyen nyelvet (valószínűleg olyan nyelven is lehet .NET-ben programozni)
- kihasználhatjuk az automatikus memória-menedzselés szolgáltatásait (Garbage Collector)

- felhasználhatjuk a programfejlesztéshez az eleve adott Base Class Library rendkívül széles szolgáltatás-rendszerét, ami radikálisan csökkenti a fejlesztési időt

A dotNet keretrendszer (Microsoft.NET **Framework**) jelenleg ingyenesen letölthető a Microsoft honlapjáról. A keretrendszer részét képezi a BCL, és a CLR réteg, valamint egy parancssori C# fordító. Ennek megfelelően a dotNet programozási környezet ingyenesen hozzáférhető minden programozást tanulni vágyó számára. Ami nem ingyenes, az a programozási felület (IDE = Integrated Development Environment – Integrált Fejlesztői Környezet). A dotNet-ben a Microsoft által fejlesztett ilyen környezetet Microsoft Visual Studio.NET-nek nevezik. Ez nem csak egy színes szövegszerkesztő. Részét képezi egy 3 CD-t megtöltő súgó, mely tartalmazza a BCL leírását, példákkal illusztrálva. Az IDE nem csak a forráskód gyors áttekintésében segít a színekkel történő kiemeléssel (syntax highlight), hanem a program írás közben már folyamatosan elemzi a forráskódot, és azonnal megjelöli a hibás sorokat benne. Környezetérzékeny módon reagál a billentyűk leütésére, és kiegészíti ill. javítja az éppen begépelés alatt álló kulcsszót. Ezen túl a felületből kilépés nélkül lehet a forráskódot lefordítani, és a generált programot elindítani. A Studio kényelmes módon kezeli a több file-ból álló programokat (project), illetve a több project-ből álló programokat is (solution).

Programozás tankönyv

III. Fejezet

„Helló Világ!”

Hernyák Zoltán

Majd minden programozó ezzel a kedves kis programmal kezdi a programozástanulást: írunk olyan számítógépes programot, amely kiírja a képernyőre, hogy „helló világ” (angolul: Hello World). Ezen program elhíresült példaprogram – ezzel kezdjük hát mi is ismerkedésünket ezzel a különös, néha kissé misztikus világgal.

```
class Saját
{
    static void Main()
    {
        System.Console.WriteLine("Hello Világ");
    }
}
```

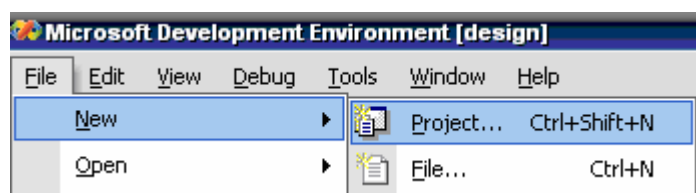
A fenti példaprogramot készítsük el egy tetszőleges editor programmal (akár a jegyzetomb is megfelel) HelloVilag.CS néven, majd *command* prompt-ból fordítsuk le a kis programunkat: **csc HelloVilag.cs** parancs kiadása segítségével.

A csc.exe – a C-Sharp compiler - a merevlemezen a Windows rendszerkönyvtárban, a C:\WINDOWS\Microsoft.NET\Framework\<version> alkönyvtárban található, ahol a <version> helyében a dotNET Framework verziószámát kell behelyettesíteni (például: v1.1.4322). Amennyiben a számítógép nem ismeri fel a csc programnevet, úgy vagy írjuk ki a teljes nevet, vagy vegyük fel a PATH-ba a fenti alkönyvtárat.

A fenti parancs kiadása után ugyanazon alkönyvtárban, ahova a program szövegét is lementettük, elkészül egy HelloVilag.exe futtatható program is. Elindítva e kis programot, az kiírja a képernyőre a kívánt üdvözlő szöveget.

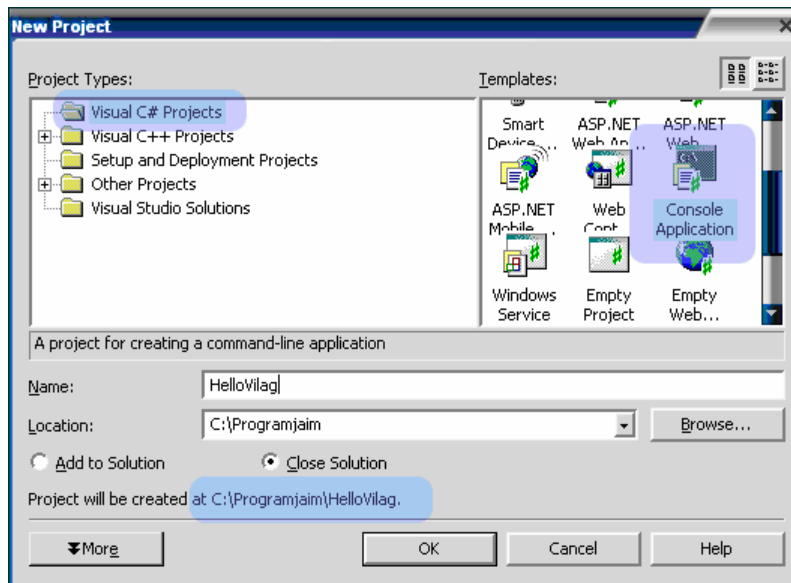
Nos, ezen módja a programkészítésnek nagyon 'kőkorszaki'. Nézzük, milyen módon kell ezt csinálni a 21. században...

Első lépésként indítsuk el a Microsoft Visual Studio.NET fejlesztői környezetet, majd a **File/New/Project** menüpontot válasszuk ki:



1. ábra

A felbukkanó párbeszédablak arra kíváncsi, milyen típusú program írását kívánjuk elkezdni:



2. ábra

A fenti párbeszédablak minden pontja roppant fontos! Első lépésként a bal felső részen határozzuk meg, milyen nyelven kívánunk programozni (ez C#).

Második lépésként adjuk meg, milyen típusú programot akarunk fejleszteni ezen a nyelven. Itt a sok lehetőség közül egyelőre a *Console Application*-t, a legegyszerűbb működési rendszerű programtípust választjuk ki.

Harmadik lépésként a *Location* részben adjuk meg egy (már létező) alkönyvtár nevét. Negyedik lépésben válasszuk ki a készülő program nevét a *Name* részben. Mivel a program több forrásszövegből fog majd állni, ezért a *Studio* külön alkönyvtárat fog neki készíteni. Ezen alkönyvtár neve a *Location* és a *Name* részből tevődik össze, és a „*Project will be created...*” részben ki is van írva, jelen példában ez a `C:\Programjaim\HelloVilag` alkönyvtár lesz!

Az *OK* nyomógombra kattintva rövid időn belül nagyon sok minden történik a háttérben. Ezt egy *Windows Intéző* indításával azonnal tapasztalhatjuk, ha megtekintjük a fenti alkönyvtár tartalmát. Több file is generálódott ebben az alkönyvtárban. De térjünk vissza a *Studio*-hoz!

```

using System;

namespace HelloVilag
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        /// <summary>
        /// The main entry point for the
        /// application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            //
            // TODO: Add code to start
            // application here
            //
        }
    }
}

```

A generált program nagyon sok sorát kitörölhetnénk, és akit zavarnak a programban lévő felesleges programsorok azok tegyék is meg bátran (csak ne végezzenek fél munkát!).

Mire figyeljünk? A programunk utasításokból áll. Egyelőre jegyezzük meg a következő fontos szabályt: ***minden programutasítást a Main függvény belsejébe kell írni! Sehova máshova!*** A Main függvény belsejét a Main utáni kezdő kapcsos zárójel és a záró kapcsos zárójel jelzi!

```

using System;
namespace HelloVilag
{
    class Class1
    {
        static void Main(string[] args)
        {
            // ...
            // ide kell írunk a program sorait
            // ...
        }
    }
}

```

Koncentráljunk most a valódi problémára: a programnak ki kell írnia a képernyőre a Helló Világ szöveget! Vigyünk a kurzort a *Main* függvény belsejébe, és a hiányzó sort (`System.Console.WriteLine("Hello Világ");`) gépeljük be.

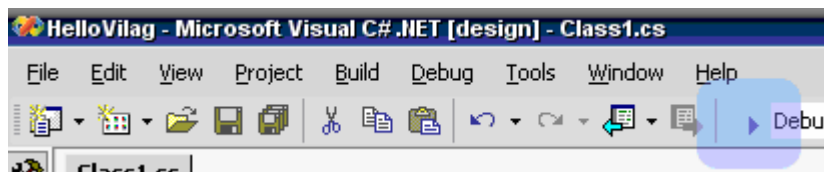
```

using System;
namespace HelloVilag
{
    class Class1
    {
        static void Main(string[] args)
        {
            System.Console.WriteLine("Hello Világ");
        }
    }
}

```

A kiíró utasítás neve WriteLine, mely az angol Write=írni, Line=sor szavakból tevődik össze. A szöveget a konzolra kell írni (Console=karakteres képernyő). Ez egy, a rendszer (angol: System=rendszer) által eleve tartalmazott utasítás, ezért a parancs teljes neve *System.Console.WriteLine*. A kiírandó szöveget a parancs után gömbölyű zárójelbe kell írni, és dupla idézőjelek közé kell tenni. Az utasítás végét pontosvesszővel kell lezárni (mint a 'pont' a mondat végén).

Az elkészült programot a **File/Save All** menüponttal mentjük le, majd indítsuk el. Ennek több módja is van. Az első: **Debug/Start** menüpont. Második: üssük le az **F5** billentyűt. A harmadik: kattintsunk az indítás gombra, amelyet egy kis háromszög szimbolizál:



3. ábra

Mivel a programindítás igen fontos, és sűrűn használt funkció, ezért javasoljuk, hogy jegyezzük meg az F5 billentyűt!

Máris van egy problémánk: a program elindul, rövid időre felvillan egy fekete színű kis ablak, esetleg elolvashatjuk a kiírást, de csak ha elég gyorsak vagyunk, mert az ablak azonnal be is csukódik! *Miért?* A válasz: a program egyetlen utasítást tartalmaz: a szöveg kiírását. Mivel a program ezt már megtette, más dolga nincsen, ezért így a futása be is fejeződik. És ha a program már nem fut, akkor bezáródik az ablak!

A megoldás is erre fog épülni: adjunk még dolgot a programnak, hogy ne fejeződjön be azonnal! **Első módszer:** utasítsuk arra, hogy várjon valamennyi idő elteltére. A szóban forgó utasítás:

```

System.Threading.Thread.Sleep(1000);

```


Ezen utasításnak zárójelben (paraméterként) ezredmásodpercben kell megadni a várakozás idejét. A megadott érték (1000) éppen 1 másodpercnyi várakozást jelent. A Sleep=aludni jelentése konkrétan annyi, hogy a program jelen esetben 1 másodpercre 'elalszik', e közben nem történik semmi (még csak nem is álmodik), de a következő utasítást majd csak egy másodperc letelte után fogja végrehajtani a számítógép!

Lényeges, hogy hova írjuk ezt a sort! A szabály értelmében a Main függvény belsejébe kell írni. De milyen sorrendben?

A programozás tanulás legjobb módszere a próbálkozás! Próbáljuk meg a kiírás *elő* beírni!

```
static void Main(string[] args)
{
    System.Threading.Thread.Sleep(1000);
    System.Console.WriteLine("Hello Világ");
}
```

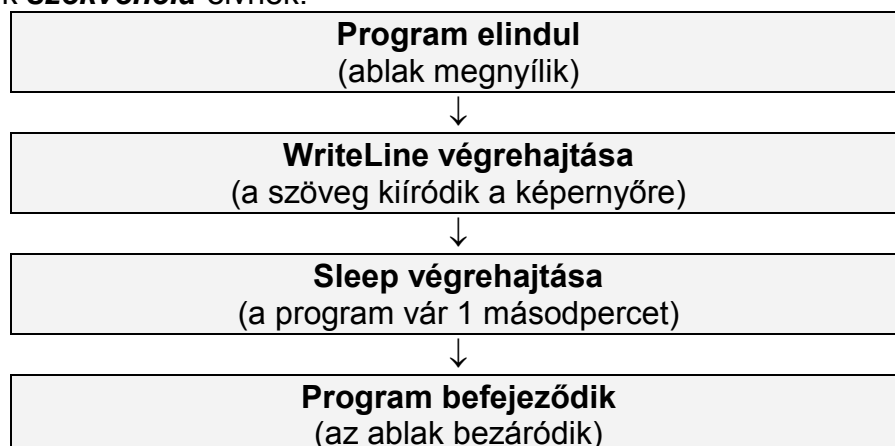
Indítsuk el a programot, és gondolkozzunk el a látottakon: a kis ablak előbukkan, kis ideig nem történik semmi, majd felvillan a kiírás, és azonnal bezáródik az ablak.

Próbáljuk meg fordított sorrendben:

```
static void Main(string[] args)
{
    System.Console.WriteLine("Hello Világ");
    System.Threading.Thread.Sleep(1000);
}
```

A kiírt szöveg megjelenik, eltelik egy kis idő (1 másodperc), majd bezáródik az ablak. *Miért?*

A számítógép a program sorait nem akármilyen sorrendben hajtja végre. Van egy nagyon fontos szabály: az utasításokat ugyanabban a sorrendben kell végrehajtani, amilyen sorrendben a programozó leírta azokat. Ezt azt elvet később pontosítjuk, és elnevezzük **szekvencia**-elvnek.



Persze felmerül a kérdés, hogy 1 mp elég-e? Növelhetjük az időt persze az érték átállításával. De mennyire? Ez a megoldás nem elég rugalmas. Néha több időt kellene várni, néha kevesebbet. Keressünk **más megoldást**:

```
System.Console.ReadLine();
```

A `Sleep`-es sort cseréljük le a fenti sorra. Próbáljuk elindítani a programot. Mit tapasztalunk? Hogy az ablak nem akar bezáródni! Üssük le az *Enter* billentyűt – és ekkor az ablak bezáródik.

Mi történt? Az ablak akkor záródik be, amikor a program futása befejeződik. Ezek szerint a program még futott, azért nem záródott be! Mivel most összesen két utasításunk van (a `WriteLine`, és a `ReadLine`), melyik utasítás miatt nem fejeződött be a program? Az nem lehet a `WriteLine`, mert a szöveg már kiíródott a képernyőre, ez az utasítás végrehajtása tehát már befejeződött! Ekkor csak a `ReadLine` utasítás maradt. Nos, a `ReadLine` utasítást alapvetően akkor használjuk majd, ha adatot kérünk be a billentyűzetről. Az adatbevitelt az *Enter* leütésével jelezzük, ezért a `ReadLine` mindaddig nem fejeződik be, amíg le nem ütjük az *Enter*-t.

*Megjegyzés: a megoldás azért nem tökéletes, mert bár a `ReadLine` ellátja a feladatát, de sajnos az *Enter* leütése előtt lehetőségünk van bármilyen szöveget begépelni, s ez elrontja a dolog szépségét. De egyelőre ez a módszer lesz az, amit használni fogunk.*

Kerüljünk közelebbi kapcsolatba a programmal, és a programozással! Figyeljük meg lépésről lépésre a működést! Ehhez ne az F5 billentyűvel indítsuk el a programot, hanem az F11 billentyűvel (vagy a *Debug/Step into* menüponttal):

```
static void Main(string[] args)
{
    System.Console.WriteLine("Hello Világ");
    System.Threading.Thread.Sleep(1000);
}
```

4. ábra

A sárga (szürke) kiemelés a következő végrehajtandó utasítást jelöli. Közben láthatjuk, hogy a program ablaka létrejött, de még egyelőre üres. Üssük le újra az F11 billentyűt! A sárga kiemelő csík átlépett a `Sleep` sorára. Ezt azt jelenti, hogy a `WriteLine` végrehajtásra került? Ellenőrizzük! A programablakra kattintva láthatjuk, hogy kiíródott a szöveg a képernyőre. Üssük le újra az F11 billentyűt. Egy másodpercig nem történik semmi (a program 'elalszik'), majd újra felbukkan a sárga kiemelés a `Main` függvény blokkjának záró kapcsos zárójelén. Ez az utolsó pillanat, amikor még ellenőrizhetjük a program kiírásait a fekete háttérű ablakban. Újabb F11 leütésére a program végképp befejeződik, az ablak bezáródik.

Nagyon ügyeljünk, hogy ha már elkezdtük a programot futtatni lépésenként, akkor ne módosítsuk a program szövegét (ez főként akkor fordul elő, ha véletlenül leütünk valamilyen billentyűt). Ha ez mégis előfordulna, a Studio nem fogja tudni, mit is akarunk tőle:



5. ábra

A szöveg fordítása: *„A forráskód megváltozása nem fog jelentkezni a program futásában, amíg azt újra nem indítja. Ha folytatja a futtatást, a forráskód és a futó program nem fog megegyezni.”* Ez azt jelenti, hogy ha pl. átírjuk a szöveget „Helló Világ!”-ről „Helló Mindenki”-re, és folytatjuk a program futtatását, akkor is még az eredeti „Helló Világ” fog kiíródni. Ennek oka az, hogy amikor elkezdtük a programot futtatni (az első F11 leütésével), a Studio rögzítette az állapotot (a forráskód akkori tartamát), és a közben bekövetkezett változásokat nem fogja figyelembe venni. Ez azért nagy gond, mert mi a forráskódban a már megváltozott szöveget látjuk, és esetleg nem értjük, hogy miért nem az történik, amit látunk, ha végrehajtjuk a következő lépést!

A *Restart* lenyomásával leállíthatjuk a program futtatását, vagy a *Continue* lenyomásával dönthetünk úgy is, hogy a változás ellenére folytatjuk az eredeti program futtatását.

A program futtatását egyébként bármikor megszakíthatjuk a **Debug/Stop debugging** menüponttal, vagy a Shift-F5 leütésével.

A fenti módszert főként akkor használjuk, ha a program nem úgy működik, ahogyan azt szeretnénk, vagy csak nem értjük annak működését teljes mértékben. E módszert nyomkövetésnek hívjuk (angolul *debugging*), jelen esetben lépésenkénti programvégrehajtást végzünk. Később újabb módszerekkel és technikákkal bővíthetjük ez irányú tudásunkat.

Feladatok:

1. **Programozási feladat:** Írassuk ki a képernyőre a családfánk egy részét, pl. az alábbi formában:

Kis Pál (felesége: Nagy Borbála)

Kis József

Kis Aladár (felesége: Piros Éva)

Kis István

Kis Pál

2. **Programozási feladat:** Írassuk ki a képernyőre az eddig megismert C# parancsokat, és rövid leírásukat, pl. az alábbi formában:

****** WriteLine ******

Kiír egy szöveget a képernyőre.

****** ReadLine ******

Vár egy ENTER leütésére.

****** Sleep ******

Várakozik a megadott időig.

Programozás tankönyv

IV. Fejezet

„Alap I/O”

Király Roland

Az alapvető Input/Output

Az alapvető input- output, vagyis a Konzol alkalmazások ki- és bemenetének tárgyaláshoz elsőként meg kell ismernünk néhány C#-ban használatos változó típust. A változók, a változó típusok ismerete nagyon fontos, bármely programozási nyelvet szeretnénk elsajátítani, mivel a programban használt adatokat változókban és konstansokban tudjuk tárolni. A program ezekkel számol, és segítségükkel kommunikál a felhasználóval. A kommunikáció a programozási nyelvek esetében körülbelül azt jelenti, hogy adatokat olvasunk be a billentyűzetről, s a munka végeztével a kapott eredményt kiírjuk a képernyőre.

Minden változónak van neve, vagy más néven azonosítója, típusa, és tartalma, vagyis aktuális értéke. Ezeken kívül rendelkezik élettartammal és hatáskörrel. Az élettartam azt jelenti, hogy az adott változó a program futása során mikor és meddig, a hatókör azt adja, hogy a program mely részeiben használható. A változók névének kiválasztása során ügyelnünk kell a nyelv szintaktikai szabályainak betartására.

Az alábbiakban megvizsgálunk néhány példát a helyes és helytelen névadásokra:

```
Valtozo  
valtozo  
Valtozónév  
szemely_2_neve  
int  
alma#fa  
10szemely
```

Az első négy névadás helyes. Vegyük észre, hogy a `valtozo` és a `Valtozo` azonosítók két külön változót jelölnek, mivel a C# nyelv érzékeny a kis-nagybetűk különbségére. Az angol terminológia ezt *Case-sensitive* nyelvnek nevezi.

A negyedik elnevezés helytelen, mivel az `int` foglalt kulcsszó. Az ötödik névben a `#` karakter szerepel, ami nem használható. Az utolsó elnevezés számmal kezdődik, mely szintén hibás.

A változókat a programban bevezetjük, vagyis közöljük a fordító rendszerrel, hogy milyen nével milyen típusú változót kívánunk használni a programban. Ezt a folyamatot deklarációnak nevezzük.

```
valtozo_nev  tipus;
```

A típus határozza meg a változó lehetséges értékeit, értéktartományait, illetve azt, hogy, milyen műveleteket lehet értelmezni rajta, és milyen más típusokkal kompatibilis, a névvel pedig a változóra hivatkozhatunk. A kompatibilitás akkor fontos, mikor az egyik változót értékül akarjuk adni egy másiknak. Inkompatibilitás esetén a .NET fordító hibát jelez.

Más nyelvektől eltérően a C#-ban az ékezetes betűket is használhatjuk névadásra.

```
char ékezetes_betű;  
int egész;
```

A változók a memóriában tárolódnak, vagyis minden azonosítóhoz hozzárendeljük a memória egy szeletét, melyet a rendszer lefoglalva tart a változó teljes élettartama alatt. (Néhány változó típus esetén, mint a pointerok, valamivel bonyolultabb a helyzet, de a .NET rendszerben nem kell törődnünk a memória kezelésével, mivel a .NET felügyeli, lefoglalja és felszabadítja a memóriát.) Vizsgáljunk meg néhány, a C# nyelvben használatos, egyszerű típust!

| típus | Méret | Értéktartomány | A típusban tárolható adatok |
|---------|---------|--|-----------------------------|
| byte | 1 byte | 0 tól 255 ig | Előjel nélküli egész számok |
| int | 4 byte | -2,147,483,648 tól 2,147,483,647 ig | előjeles egész számok |
| float | 4 byte | $\pm 1.5 \times 10^{-45}$ tól $\pm 3.4 \times 10^{38}$ ig | Valós(lebegőpontos) számok |
| double | 8 byte | $\pm 5.0 \times 10^{-324}$ tól $\pm 1.7 \times 10^{308}$ ig | Valós(lebegőpontos) számok |
| decimal | 16 byte | $\pm 1.0 \times 10^{-28}$ tól $\pm 7.9 \times 10^{28}$ ig | Valós(lebegőpontos) számok |
| bool | 1 byte | true/false | True, false értékek |
| char | 2 byte | U+0000 tól U+ffff ig | Unicode karakterek |
| string | - | | Karakterláncok |

A táblázatban felsorolt típusokkal deklarálhatunk változókat. Az alábbi példa bemutatja a deklaráció pontos szintaktikáját.

```
int i;  
char c;  
string s;
```

A változóinkat kezdőértékkel is elláthatjuk. A kezdőérték adása azért is fontos, mert az érték nélkül használt változók kifejezésekben való szerepeltetése esetén a fordító hibát jelez.

```
int k=0;
char c='a';
string z="alma";
```

A következő program bemutatja, hogyan lehet a változókat deklarálni, és kezdőértékkel ellátni. A képernyőn nem jelenik meg semmi, mivel kiíró és beolvasó utasításokat nem használunk. Ezeket a fejezet későbbi részeiben tárgyaljuk.

```
namespace deklaracio
{
    class valtozok_
    {
        [STAThread]
        static void Main(string[] args)
        {
            int r=0;
            float h,l;
            int a=0,b=1,c=0;
            int d=a+b;
            int k=a+10;

            float f;

            char ch;
            char cr='a';

            bool bo=true;
            bool ba;

            string s1;
            string s2="Hello!";
        }
    }
}
```

Gyakran előforduló hiba, hogy a deklaráció során nem adunk nevet vagy típust a változónak, vagy egyáltalán nem deklaráljuk, de a programban próbálunk hivatkozni rá. Ekkor a .NET fordító a futtatáskor hibát jelez. Előfordul, hogy nem megfelelő típusú kezdő értékkel látjuk el a változókat. Ebben az esetben a következő hibaüzenetek jelenhetnek meg a képernyőn:

- *Cannot implicitly convert type 'string' to 'int'*
- *Cannot implicitly convert type 'int' to 'string'*

Azonos változónevek esetén is hibaüzenetet kapunk. Gyakori hiba az is, hogy az osztály, vagyis a class neve megegyezik valamely változó nevével, esetleg lefoglalt kulcsszót akarunk alkalmazni a névadásnál.

Bizonyos esetekben, amikor nem *Error*, hanem *Warning* típusú hibaüzenetet kapunk, a fordító olyan hibát talál a programunkban, amittől az még működőképes, de hatékonyságát csökkenti. Ilyen hiba lehet, ha egy változót deklarálunk, de nem használunk fel.

A következő példában láthatunk néhány rossz deklarációt. (- *hogya* a programozás során ne kövessünk el hasonló hibákat.)

```
int a="alma"; az int típus nem kompatibilis a string konstanssal
string f=2; az s változóba számot akarunk elhelyezni
int class=10; a class foglalt szó
int void=10; a void az eljárásoknál használatos címke
```

A következő példa megmutatja, hogy a programjaink mely részeiben deklarálhatunk.

```
using System;

namespace AlapIO
{
    class IO_
    {
        int a,b;
        int d=10;
        char c;
        bool t,f=false;
        string s;
    int y = (int)3.0;

        [STAThread]
        static void Main(string[] args)
        {
            int l;
            string s="ez egy string konstans";
            int a = 12;
            int b = a + 10;
            bool t = true;
            bool f = !t;
            char c = 'a';

        }
    }
}
```

A fenti változók (a, b, c, d, t, f) a programunk futása közben használhatóak, az értékadások után tartalommal, értékkel rendelkeznek, melyet a program futása alatt, vagy a változóra vonatkozó következő értékadásig meg is tartanak. A felhasználó még mindig nem láthatja őket, és az értéküket nem tudja módosítani. Ahhoz, hogy lehetővé tegyük a program használójának a változók értékének manipulálását, nekünk kell a megfelelő utasításokat beépíteni a forráskódba. Amennyiben ezt tesszük, ügyelnünk kell a programkód helyességére.

Az értékadásnak jól meghatározott szintaktikája van. Az értékadó utasítás bal oldalán a változó azonosítója áll, középen egyenlőség jel, a jobb oldalon pedig az érték, vagy kifejezés, melynek az aktuális értékét a változóban tárolni szeretnénk.

```
int d=2;
c=a+40;
k=(10+4)/2;
int y = (int)3.0;
```

A helytelenül felírt értékadást a fordító hibaüzenettel jelzi.

A példában az y változónak egy valós típust adunk értékül, de ebben az esetben az (int)3.0 típus kényszerítéssel nem okozunk hibát. Ez egy ún.: explicit konverzió. A változók egy érdekes típusa a literál. Akkor használjuk, mikor a programban egy konkrét értéket szeretnénk szerepeltetni, pl.:

```
hibauzenet_1 = "Helytelen Értékadás";
max_Db = 20;
```

A literálok alapértelmezés szerint int típusúak, ha egészek, és double, ha valósak. Amennyiben float típust szeretnénk készíteni, az érték után kell írni az f karaktert, long típus esetén az l, illetve ulong esetén az ul karakterpárt, stb.

```
Float_literal = 4.5f;
Long_literal = 4l;
```

A C# programokban állandókat, vagy más néven konstansokat is definiálhatunk. A konstansok a program futása alatt megőrzik értéküket, s nem lehet felüldefiniálni őket, illetve értékadó utasítással megváltoztatni értéküket. Más nyelvektől eltérően, itt a konstansnak is van típusa.

```
const int a=10;
const string s="string típusú konstans";
```

A programjainknak fontos része a felhasználóval való kommunikáció. Adatokat kell kérni tőle, vagy közölnünk kell, mi volt a program futásának eredménye. Ahhoz, hogy az adatokat, vagyis a változók tartalmát be tudjuk olvasni vagy meg tudjuk jeleníteni a képernyőn, a .NET rendszerben igénybe vehetjük a C# alapvető I/O szolgáltatásait, a System névtérben található Console osztály ide tartozó metódusait (függvények és eljárások).

```
System.Console.Read();
System.Console.Write();
System.Console.ReadLine();
System.Console.WriteLine();
```

A Console.Write() és a Console.WriteLine() a kiírásra, míg a Console.Read() és a Console.ReadLine() a beolvasásra használható. A beolvasás azt jelenti, hogy az ún.: standard input stream –ről

várunk adatokat. Amennyiben a Read() beolvasó utasítást használjuk, int típusú adatot kapunk, a ReadLine() metódus esetében viszont stringet. Ez kiderül, ha megnézzük a két metódus prototípusát.

```
public static string ReadLine();  
public static int Read();
```

Jól látszik, hogy a Read() int típusú, a ReadLine() viszont string. Adat beolvasásakor természetesen nem csak erre a két típusra van szükségünk, ezért az input adatokat konvertálnunk kell a megfelelő konverziós eljárásokkal, melyekre később bővebben kitérünk. A System hivatkozás elhagyható a metódusok hívásakor, amennyiben azt a program elején, a using bejegyzés után felvesszük a következő módon:

```
using System;
```

Ezt a műveletet névtér importálásnak nevezzük és a könyv későbbi fejezeteiben bővebben olvashatunk a témáról.

Mikor eltekintünk a névtér importálástól, akkor az adott metódus teljes, minősített, vagy q nevééről beszélünk, ami a függvény névtérben elfoglalt helyével kezdődik. Ez a minősítő előtag (pl.: System.Console...), így a program bármely részéből meghívhatjuk az adott függvényt, vagy eljárást. Erre szükség is lehet, mivel a System névtér is több fájlból (DLL) áll, amelyeket a fordító nem biztos, hogy megtalál a hivatkozás nélkül. Ez nem csak a System -re, hanem valamennyi névtérre igaz.

```
Pl.:System.Thread.Threadpool.QueueUserWorkItem();
```

(a többszálú programok készítésénél van jelentősége, később még visszatérünk rá)

Ahhoz, hogy használni tudjuk a Console metódusait, meg kell vizsgálnunk néhány példát. A következő program bemutatja a beolvasás és a kiírás mechanizmusát.

```
using System;  
  
namespace ConsoleApplication7  
{  
    class Class1  
    {  
        [STAThread]  
        static void Main(string[] args)  
        {  
            int a=0,b=0;  
            Console.Write("a erteke : ");  
            a=Convert.ToInt32(Console.ReadLine());  
            Console.Write("b erteke : ");  
            b=Convert.ToInt32(Console.ReadLine());  
            Console.WriteLine("\n a={0}\n b={1}\n a+b={2}",a,b,a+b);  
        }  
    }  
}
```

```
        Console.ReadLine();  
    }  
}
```

Amennyiben futtatjuk a programot a képernyőn a következő *output* (kimenet) jelenik meg:

```
a erteke : 10  
b erteke : 110  
  
a=10  
b=110  
a+b=120
```

A `System.Console.WriteLine()` metódussal a standard output - ra, vagyis a képernyőre tudunk írni, pontosabban a képernyőn megjelenő *Consol* alkalmazás ablakába. A metódusban ún.: formátum string - et, vagy más néven maszkot alkalmaztunk, hogy a változók értékeit formázottan, vagyis a számunkra megfelelő alakban tudjuk megjeleníteni.

A formátum string tartalmaz konstans részeket (`a=`, `b=`, `a+b=`) ami változatlan formában kerül a képernyőre. A `{0}`, `{1}`, `{2}` bejegyzéseket arra használjuk, hogy a formátum string megfelelő pontjaira behelyettesítsük a paraméterlistában felsorolt változók értékeit: a `{0}` jelenti a nulladik, vagyis a sorban az első változó helyét, a `{1}` a második változó helyét, és így tovább. Amennyiben a `{}` zárójelek között olyan értéket adunk meg, mely nem létező változóra hivatkozik, a program leáll.

```
Console.WriteLine("{1} {3}", i);
```

A példában az első és a harmadik változóra hivatkoztunk (sorrendben a második és a negyedik), de ilyenek nem léteznek, mivel az egyetlen változó a kiíró utasításban az `i`, mely a nulladik helyen áll. A helyes hivatkozás tehát:

```
Console.WriteLine("{0}", i);
```

Ezek a bejegyzések nem hagyhatóak el, mivel csak így tudjuk a változók tartalmát a képernyőre írni. A sorrendjük sem mindegy. Vegyük észre, hogy a példaprogramban a második `Console.WriteLine()` paraméter listájában a változók fel vannak cserélve, a kiírási sorrend mégis megfelelő, mivel a formátum definiálja, a változók helyét a kiírt szövegben.

(Érdekes kérdés lehet, hogy a `{`, `}`, `{0}` karaktereket hogyan írjuk a képernyőre. A `Console.WriteLine("{} {0} = {0}", a);` nem megfelelő. A kedves olvasó kipróbálhatja a `Console.WriteLine("{{0}} = {0}", a);` utasítást...)

A formátum maszkjába a következő vezérlő karaktereket helyezhetjük el:

| | |
|----|----------------------|
| \b | Backspace |
| * | Újsor |
| \t | vízszintes tabulátor |
| \\ | Fordított perjel |
| \' | Aposztróf |
| \" | Idézőjel |
| \n | Sortörés |

Vegyük észre, hogy a programunk megjeleníti a tárolt értékeket, s azok összegét is, de nem ad lehetőséget felhasználónak a változók értékének a megváltoztatására.

Ahhoz, hogy tovább tudjunk lépni, el kell sajátítanunk, hogyan kezelhetjük le a felhasználótól érkező inputot, vagyis a billentyűzetről bevitt adatokat.

A felhasználói input kezelésére lehetőséget biztosítanak a `Console.Read()` ,és a `Console.ReadLine()` metódusok. Mindkettő esetében adatokat olvashatunk be a standard inputról, vagyis a billentyűzetről, s ha megtámogatjuk a két metódust a `Convert` osztály konverziós függvényeivel, nemcsak `int` és `string` típusú adatokat, hanem szám típusú, vagy logikai értékeket is beolvashatunk a felhasználótól. A példaprogramban a `Convert.ToInt32()` metódust használjuk az input 32-bites `int` típusú konvertálására. Ha a beolvasás során az input stream –ről hibás adatot kapunk, mely nem konvertálható, a programunk leáll.

Az ilyen jellegű hibákra megoldás lehet a kivételek kezelése, melyről a későbbi fejezetekben szót ejtünk majd.

Nézzünk meg a példát a beolvasásra és a konverzióra!

```
using System;

namespace ConsoleApplication7
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            int a=0,b=0;
            Console.Write("a erteke : ");
            a=Convert.ToInt32(Console.ReadLine());
            Console.Write("b erteke : ");
            b=Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("\n a={0}\n b={1}\n a+b={2}",a,b,a+b);
            Console.ReadLine();
        }
    }
}
```

Ebben az alkalmazásban a változók a felhasználói inputról nyernek értéket, így a beolvasott adatok változtatásával minden lefutásnál más-más eredményt kapunk. A program végén a

Console.ReadLine() utasítást arra használjuk, hogy a Consol ablak ne tűnjön el a képernyőről, miután a program lefutott. Ez fontos a Consol alkalmazások esetén, mivel ezek nem esemény vezérelt működésűek. A programok elindulnak, végrehajtják a programozó által definiált utasításokat, majd leállnak, és a kimeneti képernyőt is bezárák. Így a felhasználó nem lát szinte semmit az egészből.

A fentiek ismeretében vegyük sorra, hogy a változók milyen módon kaphatnak értéket!

Egyszerű értékadással:

```
A=2;  
B="alma";  
C=A+B;  
E=2*A;
```

A standard inputról:

```
A=Convert.ToInt32(Console.ReadLine());  
B=Console.ReadLine();
```

Kezdőérték adásával:

```
int D=10;  
char c="a"
```

A változókba beolvasott értékeket fel is tudjuk használni, és legtöbbször ez is a célunk. Miért is olvasnánk be értékeket, ha nem kezdünk velük semmit? A következő program két változó értékéről eldönti, hogy melyik a nagyobb. Ez a program viszonylag egyszerű, és jól bemutatja a billentyűzetről nyert adatok egész számmá konvertálását. A Convert osztály ToInt32() metódusával alakítjuk a beolvasott, itt még String típusként szereplő számot, majd azt konvertáljuk egész számmá (int), s adjuk értékül az a változónak. A b változóval ugyanígy járunk el. A beolvasást követően megvizsgáljuk, hogy a számok egyenlők-e, ha ez a feltétel nem teljesül, megnézzük, hogy melyik változó értéke a nagyobb, majd az eredményt kiírjuk a képernyőre. Nézzük meg a forráskódot!

```

using System;

namespace Convert
{
    class KN
    {
        [STAThread]
        static void Main(string[] args)
        {
            int a=0,b=0;
            Console.Write("a értéke? = ");
            a=Convert.ToInt32(Console.ReadLine());
            Console.Write("b értéke? = ");
            b=Convert.ToInt32(Console.ReadLine());
            if (a==b)
            {
                Console.WriteLine("{0} és {1} egyenlőek",a,b);
            }
            else
            {
                if (a>b)
                {
                    Console.WriteLine("{0} a nagyobb mint {1}",a,b);
                }
                else
                {
                    Console.WriteLine("{0} a nagyobb mint {1}",b,a);
                }
            }
            Console.ReadLine();
        }
    }
}

```

Az alkalmazás kimenetére kerek mondat formájában írjuk ki az eredményt úgy, hogy a kiírt string konstanst konkatenáljuk a változók értékével, így foglalva mondatba a program futásának eredményét. Ez gyakori megoldás a felhasználóval történő kommunikáció megvalósítására. Mindenképpen jobb, ha összefüggő szöveg formájában beszélgetünk. A hétköznapi kommunikáció során is jobb, ha kerek mondatokat használunk, hogy jobban megértsenek minket.

A program használhatóságát növeli, vagyis felhasználóbarát programokat készíthetünk.

Biztosan sokan emlékeznek a következő párbeszédre:

- Mennyi?
- Harminc.
- Mi harminc?
- Mi mennyi?

Az ilyen és hasonló párbeszédok érthetatlenné teszik a kommunikációt mind a valós életben, mind a programok világában. Az elsőben már egész jól megtanultuk a társas érintkezés formáit, tanuljuk meg a másodikban is!

A kiíratás és beolvasás mellett a műveletvégzés is fontos része a programoknak. A műveleteket csoportosíthatjuk a következő módon:

- Értékadás: $a = 2$,
- matematikai műveletek: $z = a + b$,
- összehasonlítások: $a = b$, $a < c$,
- feltételes műveletek.

A műveleti jelek alapján három kategóriát hozhatunk létre. Unáris, bináris és ternáris műveletek. Az elnevezések arra utalnak, hogy a műveletben hány operandus szerepel.

Az aritmetikai értékadásokat rövid formában is írhatjuk. A következő táblázat bemutatja a rövidítések használatát, és azt, hogy a rövidített formulák mivel egyenértékűek.

| rövid forma | Használat | Jelentés |
|-----------------|----------------------|--|
| <code>+=</code> | <code>x += 2;</code> | <code>x = x + 2;</code> |
| <code>-=</code> | <code>x -= 2;</code> | <code>x = x - 2;</code> |
| <code>*=</code> | <code>x *= 2;</code> | <code>x = x * 2;</code> |
| <code>/=</code> | <code>x /= 2;</code> | <code>x = x / 2;</code> (egész osztás) |
| <code>%=</code> | <code>x %= 2;</code> | <code>x = x % 2;</code> (maradékos osztás) |

Most, hogy elegendő tudással rendelkezünk a Consol I/O parancsairól és a műveletek használatáról, készítsünk programot, mely a felhasználótól bekéri egy tetszőleges kör sugarát, majd kiszámítja annak kerületét és területét.

A programhoz szükséges ismeretek: a kör kerülete: $2r\pi$, és a területe: $r^2\pi$, vagy $r^2\pi$.

A feladat az, hogy egy változóba beolvassuk az adott kör sugarát, majd a képernyőre írjuk a körhöz tartozó kerületet és a területet. A kifejezések kiszámítását elvégezhetjük a kiíró utasításban is, csak a megfelelő formátum string-et kell elkészítenünk, és a π értékét behelyettesítenünk a képletekbe.

Honnan vegyük a π -t? Írhatnánk egyszerűen azt, hogy 3.14, de ebben az esetben nem lenne elég pontos a számításunk eredménye. Használjuk inkább a C# math osztályban definiált PI konstanst!

Ekkor a program a következő módon írható le:


```

using System;

namespace ConsoleApplication7
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            int r=0;
            Console.WriteLine("kör sugara : ");
            r=Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("A kör kerülete : {0},
                területe : {1} ",2*r*Math.PI,r*r*Math.PI);
            Console.ReadLine();
        }
    }
}

```

Vizsgáljuk meg, hogyan működik a program! Az *r* változóban tároljuk a billentyűzetről beolvasott értéket, a kör sugarát, majd a `Console.WriteLine()` – metódusban kiszámoljuk a megadott kifejezés alapján a kör kerületét és területét. Az eredmény kiíródik a képernyőre, s a program az *enter* leütése után befejezi futását.

Az egész típusú számokkal már jól boldogulunk, de sajnos az élet nem ilyen egyszerű. A legtöbb alkalmazásban szükség van a típusok közti konverzióra.

A fejezetben már szóltunk a típusok konverziójáról, és a `Convert` osztályról, de a teljesség kedvéért vegyük sorra a fontosabb metódusait!

Az alábbi felsorolás tartalmazza azokat a konvertáló metódusokat, melyeket az alapvető I/O alkalmazása során használni fogunk:

```

ToBoolean()
ToByte()
ToChar()
ToString()
ToInt32()
ToDateTime()

```

A `ToBoolean()` metódus logikai értékkel tér vissza.

```

Console.WriteLine("{0}", Convert.ToBoolean(1));
Bool b=Convert.ToBoolean(Console.ReadLine());

```

Logikai *true* értéket kapunk eredményül. Nullánál *false* visszatérési értéket kapnánk.

A `ToByte()` metódus `byte` típust ad vissza, de ez a fenti kiíró utasításban nem követhető nyomon, mivel a kiírt érték `byte` típusban is 1, viszont az értéktartomány megváltozik. Egy bájt nál nagyobb érték nem fér el a `byte` típusban. Nagyobb szám konvertálása hibához vezet.

```
Console.WriteLine("{0}", Convert.ToByte(1));
```

A `ToChar()` metódus a karakter ASCII kódjával tér vissza. Ez a metódus jól használható a számrendszerek közti átváltások programozására, mivel a kilences számrendszertől fölfelé történő konvertáláskor a 9-nél nagyobb számok esetén az ABC nagy betűit használjuk.

10=A, 11=B, ..., 15=F.

Az átváltáskor, ha az adott számjegy nagyobb, mint 9, átalakíthatjuk a következő módon:

```
Convert.ToChar(szamjegy+55);
```

A példában a számjegy ASCII kódját töltük el annyival, hogy az megegyezzen a megfelelő tizenhatos számrendszerbeli számjegy ASCII kódjával. A kódot ezután karakterré konvertáltuk. A következő programrészlet a fent említett eljárás használatával a 10 szám helyett egy A betűt ír a képernyőre.

```
Console.WriteLine("{0}", Convert.ToChar(10+55));
```

Amennyiben a `ToChar()` bemenő paramétere a 11+55 lenne, a B betű jelene meg a képernyőn.

A `ToString()` string-et konvertál a bemenő paraméterből.

```
string s=Convert.ToString(1.23);  
Console.WriteLine("{0}", s);
```

Eredménye az 1.23 szám string-ként. Konvertálás után 1.23-al nem végezhetünk aritmetikai műveletet, mivel `string` típust készítettünk belőle, de alkalmas `string`-ekhez való hozzáfűzésre.

A `ToInt32()` metódust már használtuk az előző programokban, de álljon itt is egy példa a használatára.

```
int a=Convert.ToInt32(Console.ReadLine());
```

A kódrészletben az int típusú változóba a billentyűzetről olvasunk be értéket. A Console.ReadLine() metódust "paraméterként" átadjuk a Convert.ToInt32() függvénynek, így a beolvasott érték egész típusú számként (int) kerül a memóriába.

Másik lehetőség a konverzióra az i=Int32.Parse(Console.ReadLine()); forma használata. A két megoldás azonos eredményt szolgáltat.

A felsorolás végére hagytuk a ToDateTime() metódust, mivel a dátum típussal még nem foglalkoztunk.

Előfordulnak programok, ahol a felhasználótól a születési dátumát, vagy éppen az aktuális dátumot szeretnénk megkérdezni. Amennyiben ki akarjuk írni a képernyőre amit beolvastunk, nincs szükség konverzióra, de ha dátum típusként tárolt adattal szeretnénk összehasonlítani azt, vagy adatbázisban tárolni, akkor használnunk kell a ToDateTime() függvényt.

```
Console.WriteLine("dátum :{0}",Convert.ToDateTime("2004/12/21"));
```

A fenti programrészlet futásának az eredménye a következő:

```
datum : 2004.12.21. 0:00:00
```

Látható, hogy a dátum mellett az aktuális idő is megjelenik, ami alapértelmezésként 0:00:00. Ebből következik, hogy ToDateTime() paraméterében megadott string-ben az aktuális időt is fel tudjuk venni a következő módon:

```
Console.WriteLine("datum + idő : {0}  
",Convert.ToDateTime("2004/12/21 1:10:10"));
```

Ennek a kódrészletnek az eredményeként a dátum mellett az idő is kiíródik a képernyőre. Természetesen a C# -ban sokkal több konverziós metódus létezik, de a programjaink megírásához a felsoroltak elegendőek. (Amennyiben a többire is kíváncsiak vagyunk, használjuk a .NET dinamikus HELP rendszerét!)

A példákban a konvertálást a kiíró utasítással kombináltuk, hogy az eredmény megjelenjen a képernyőn, de a metódusokat változók értékének a beállításakor, vagy típus konverzió esetén is használhatjuk.

```
string s=Convert.ToString(1.23);  
int k=Convert.ToInt32('1');  
k=2+Convert.ToInt32(c);  
char c=Convert.ToChar(1);  
bool b=Convert.ToBoolean(1);
```

Logikai érték esetén a beolvasás a következő módon oldható meg:

```
bool b;  
b=Convert.ToBoolean(Console.ReadLine());
```

Ennél a programrészletnél a true szót kell begépelni a billentyűzeten. Ez kicsit eltér a fent bemutatott `bool b=Convert.ToBoolean(1);` értékadástól. A beolvasásnál az 1 érték nem megfelelő. Nem szerencsés a logikai értékek beolvasásakor a `Read()` metódust használni, mivel az `int` típust olvas be.

A standard Input/Output kezelésének alapjait elsajátítottuk. A következő fejezetben a szelekcióval ismerkedünk meg. A szelekció, vagyis a feltételes elágazások témakörét ez a fejezet is érintette, de nem merítette ki. A teljes körű megismeréshez mindenképpen fontos a következő fejezet tanulmányozása.

A szintaktika mellett fontos megemlíteni, hogy a programozás folyamata nem a fejlesztői eszköz kód editorában kezdődik. Elsőként fel kell vázolni a születendő programok működését, tervezni kell, meg kell keresni a lehetséges matematikai megoldásokat (persze, csak ha erre szükség van). A tervek leírása sok esetben már a program vázlatát adja. *(Egyszerűbb programok esetén elég, ha a felhasználói igényeket papírra vetjük ☺.)* A tervezés, előkészítés lépésekeit soha nem szabad kihagyni. A megfelelően átgondolt, megtervezett programok elkészítése a későbbiekben felgyorsítja a programozási folyamatot, egyszerűbbé teszi a hibák keresését, javítását.

Programozási feladatok

1. Írjon programot, mely megkérdezi a felhasználó nevét, majd köszön neki, de úgy, hogy a nevén szólítja! (Pl.: Hello Kiss Attila!)
2. Próbálja meg átalakítani a fejezetben tárgyalt, a kör kerületét és területét kiszámító programunkat úgy, hogy az ne egy kör, hanem egy négyzet kerületét és területét számítsa ki!
3. Az előző programot módosítsa úgy, hogy téglalapok kerületét, területét is ki tudja számítani! (Szükség lesz újabb változó bevezetésére.)
4. Írjon programot, mely egy háromszög oldalainak hosszát olvassa be a billentyűzetről, majd megmondja, hogy a háromszög szerkeszthető-e! (A háromszög szerkeszthető, ha az $(a+b>c)$ és $(a+c>b)$ és $(b+c>a)$ feltétel teljesül.)
5. Olvasson be a billentyűzetről egy számot és mondjuk meg, hogy a szám negatív, vagy pozitív!
6. Kérjen be a billentyűzetről két számot, majd írja ki azok összegét, különbségét, szorzatát és hányadosát a képernyőre!
7. Készítsen programot, mely logikai true/false értékeket olvas be a billentyűzetről! True esetén a képernyőre az IGAZ szót írja ki a program!
8. Írjon programot, mely beolvas egy számpárt a billentyűzetről, majd kiírja a két szám számtani közepét!

Programozás tankönyv

V. Fejezet

„Szelekció alapszint”

Radványi Tibor

A logikai típusú változó

A logikai típusú változónak két lehetséges értéke van, a **TRUE** és a **FALSE**, vagyis igaz, vagy hamis. Ezek az értékek konstansként is megadhatók:

```
bool logikai = true;
```

Két kompatibilis érték összehasonlításának eredménye vagy igaz (true) vagy hamis (false) lesz. Például ha **Sz** változó **int** típusú, akkor az $Sz < 4$ kifejezés értéke lehet igaz is és hamis is, attól függően, hogy mi a változó pillanatnyi értéke. Az összehasonlítás eredményét tudjuk tárolni egy logikai típusú változóban.

```
bool logikai = true;
int sz = 2;
logikai = sz < 4;
Console.WriteLine("{0}", logikai);
```

Logikai kifejezéseket logikai operátorokkal köthetünk össze. Ezek a tagadás, a logikai és, illetve a logikai vagy operátorok. A C#-ban a műveleteket a következőképpen jelöljük:

| | |
|--------------------|----|
| Tagadás (negáció) | ! |
| ÉS (konjunkció) | && |
| VAGY (diszjunkció) | |

Nézzük az operátorok igazságtáblázatát a bemenő paraméterek különböző értékei mellett, az operációk milyen eredményt szolgáltatnak?

Tagadás:

| A | !A |
|-------|-------|
| TRUE | FALSE |
| FALSE | TRUE |

Tehát minden értéket az ellentettjére fordít.

ÉS

| A | B | A && B |
|-------|-------|--------|
| TRUE | TRUE | TRUE |
| FALSE | TRUE | FALSE |
| TRUE | FALSE | FALSE |
| FALSE | FALSE | FALSE |

Azaz kizárólag akkor lesz a művelet eredménye igaz, ha mindkét paraméter értéke igaz.

VAGY

| A | B | A B |
|------|------|--------|
| TRUE | TRUE | TRUE |

| | | |
|-------|-------|-------|
| FALSE | TRUE | TRUE |
| TRUE | FALSE | TRUE |
| FALSE | FALSE | FALSE |

Látható, hogy akkor hamis a művelet eredménye, ha mindkét paraméter értéke hamis.

Egy kifejezés kiértékelésében a zárójelek határozzák meg a kiértékelés sorrendjét, ha ez nem dönt, akkor a sorrend: TAGADÁS, ÉS, VAGY. Egyenrangú műveletek esetén a balról-jobbra szabály lép életbe.

```
bool logikai = true;
int x = 2, y = 5 ;
logikai = x > 4; // FALSE
logikai = logikai && !(y < 3); // FALSE
logikai = (x < 3) || (x > 4); // TRUE
logikai = (x >= 3) && (x <= 4); // FALSE
logikai = x > 3 && !(y < 6) || x < y; // TRUE
```

A feltételes utasítás

```
if (logikai kifejezés)
{
    igaz érték esetén végrehajtandó utasítások
}
```

```
if (y > 20)
    Console.WriteLine("Igaz kifejezés");
```

Látható, hogy ha mindössze egy utasítást szeretnénk végrehajtani, akkor a kapcsos zárójelek elhagyhatóak.

Előírhatunk összetett logikai feltételeket az **if** többszörözésével, vagy operátorok segítségével.

```
if (x > 10)
    if (y > 20)
        Console.WriteLine("Kifejezés igaz, igaz esetén");
```

A fenti kódrészlet csak akkor írja ki a szöveget, ha egyszerre teljesül, hogy $x > 10$ és ugyanakkor $y > 20$ is. Ezt leírhatjuk az és operátor alkalmazásával is:

```
if (x > 10 && y > 20)
```



```
Console.Write("Kifejezés igaz, igaz esetén ");
```

Tekintsük az alábbi programrészletet:

```
Console.Write("Kérem a karaktert: ");  
char c = (char) Console.Read();  
if (Char.IsLetter(c))  
    if (Char.IsLower(c))  
        Console.WriteLine("A karakter kisbetű.");
```

A billentyűzetről bekérünk egy karaktert. Két vizsgálatot csinálunk, az első **if** feltételében megvizsgáljuk hogy a kapott karakter betű-e, ha ez teljesül, akkor a második **if** feltételében vizsgáljuk, hogy a karakter kisbetű-e. Amennyiben mindkettő feltétel teljesül, akkor jelenik meg a szöveg a képernyőn. Bármely feltétel hamis állapota esetén a szöveg kiírása elmarad.

Az elágazás

if (logikai kifejezés)
{
 igaz érték esetén végrehajtandó utasítások
}
else
{
 Hamis érték esetén végrehajtandó utasítások
}

```
if (y > 20)  
    Console.Write("Igaz kifejezés");  
else  
    Console.Write("Hamis kifejezés");
```

Amennyiben a logikai kifejezés értéke hamis, akkor az **else** utáni utasítások kerülnek végrehajtásra. Itt is érvényes, hogy több utasítás megadása esetén használni kell a kapcsos zárójeleket.

```
public static void Main()  
{  
    Console.Write("Kérem a karaktert: ");  
    char c = (char) Console.Read();  
    if (Char.IsLetter(c))  
        if (Char.IsLower(c))  
            Console.WriteLine("a karakter kisbetű.");  
}
```

```

        else
            Console.WriteLine("A karakter nagybetű.");
    else
        Console.WriteLine("A karakter nem betű");
}

```

Ebben az esetben nem használhatunk a 2 db **if** utasítás helyett logikai operátorral összekapcsolt kifejezést, mert a hamis esetet külön kezeljük le.

Ha a belső **if** utasítás logikai kifejezése hamis, akkor a "A karakter nagybetűs." szöveg jelenik meg. Míg ha a külső **if** logikai kifejezése hamis, akkor a beírt karakter már nem is betű, hanem valami más karakter jelenik meg, pl.: számjegy. Ekkor a "A karakter nem betű" szöveg fog megjelenni.

Néhány egyszerű példát a szelekció alkalmazására.

Döntsük el egy számról, hogy páros-e!

```

static void Main(string[] args)
{
    int szam;
    szam = Int32.Parse(Console.ReadLine());
    if (szam % 2 == 0)
        Console.WriteLine("A szám páros");
    else
        Console.WriteLine(" A szám páratlan");
}

```

A deklarálás után bekérünk a egy számot. Mivel ez karaktersorozat, így egész számmá kell konvertálni. Az **if** utasítás egyszerű logikai feltételt tartalmaz. A % jel a maradékképzés operátora, a `szam % 2` megadja a `szam` nevű változó értékének kettes maradékát. Ez páros szám esetén 0, így ezt vizsgáljuk. A logikai egyenlőség reláció operátora a `==` karakterekkel jelöljük. Ne próbáljuk meg az értékekadás `=` jelével helyettesíteni.

Oldjuk meg az együtthatóival adott másodfokú egyenletet!

```
static void Main(string[] args)
{
    double a, b, c, d;
    double x1, x2;
    a = double.Parse(Console.ReadLine());
    if (a == 0)
    {
        Console.WriteLine("Így nem másodfokú az egyenlet!");
    }
    else
    {
        {
            b = double.Parse(Console.ReadLine());
            c = double.Parse(Console.ReadLine());
            d = b * b - 4 * a * c;
            if (d < 0)
                Console.WriteLine("Nincs valós megoldás");
            else
            {
                x1 = (-b + Math.Sqrt(d)) / (2 * a);
                x2 = (-b + Math.Sqrt(d)) / (2 * a);
                Console.WriteLine("X1 = {0}    X2 = {1}    ", x1, x2);
            }
        }
        Console.ReadLine();
    }
}
```

Megoldásra ajánlott feladatok

1.

Egy beolvasott számról döntse el a program hogy -30 és 40 között van-e!

2.

Két beolvasott szám közül írassuk ki a nagyobbikat!

3.

Vizsgáljuk meg hogy osztható-e egy A egész szám egy B egész számmal!

4.

Tetszőleges 0 és egymillió közötti egész számról mondja meg a program hogy hány jegyű!

5.

Adott egy tetszőleges pont koordinátaival. Határozzuk meg melyik síknegyedben van!

6.

Rendeztessünk sorba 3 egész számot!

7.

Három tetszőleges, számról döntse el a program hogy számtani sorozatot alkotnak e!

8.

Három adott számról döntse el a program, hogy lehetnek-e egy háromszög oldalainak mérőszámai.

9.

Olvassunk be négy számot és határozzuk meg a páronkénti minimumok maximumát!

10.

Írjunk programot, mely kibarkochbázza, hogy milyen négyszögre gondoltam (négyzet, téglalap, rombusz, stb)!

VI. Fejezet

Előírt lépésszámú ciklusok

”Ismétlés a tudás anyja”.

Hernyák Zoltán

Az eddig megírt programok szekvenciális működésűek voltak. A program végrehajtása elkezdődött a Main függvény első utasításán, majd haladtunk a másodikra, harmadikra, stb... amíg el nem értük az utolsó utasítást, amikor is a program működése leállt.

Nem minden probléma oldható meg ilyen egyszerűen. Sőt, leggyakrabban nem írható fel a megoldás ilyen egyszerű lépésekből. Vegyük az alábbi programozási feladatot: *írjuk ki az első 5 négyzetszámot a képernyőre* (négyzetszámnak nevezzük azokat a számokat, amelyek valamely más szám négyzeteként előállíthatóak - ilyen pl. a 25).

```
public void Main()
{
    Console.WriteLine("1. négyzetszám = 1");
    Console.WriteLine("2. négyzetszám = 4");
    Console.WriteLine("3. négyzetszám = 9");
    Console.WriteLine("4. négyzetszám = 16");
    Console.WriteLine("5. négyzetszám = 25");
}
```

A fenti program még a hagyományos szekvenciális működés betartásával íródott. De képzeljük el ugyanezt a programot, ha nem az első 5, de első 50 négyzetszámot kell kiírni a képernyőre!

Vegyük észre, hogy a program sorai nagyjából egyformák, két ponton különböznek egymástól: hogy hányadik négyzetszámról van szó, és annak mennyi az értéke. Az hogy hányadik négyzetszámról van szó – az mindig 1-gyel nagyobb az előző értéktől.

Nézzük az alábbi pszeudó-utasítássorozatot:

1. $I := 1$
 // az I változó értéke legyen '1'
2. Írd ki: $I, ". négyzetszám = ", I*I$
3. $I := I+1$
 // az I változó értéke növelve 1-el
4. ugorj újra a 2. sorra

Ennek során kiíródik a képernyőre az „1. négyzetszám= 1”, majd a „2. négyzetszám = 4”, „3. négyzetszám = 9”, stb...

A fenti kódot nevezhetjük kezdetleges ciklusnak is. **A ciklusok olyan programvezérlési szerkezetek, ahol a program bizonyos szakaszát (sorait) többször is végrehajthatjuk. Ezt a szakaszt ciklus magnak nevezzük.**

A fenti példaprogramban ez a szakasz (a ciklusmag) a 2. és a 3. sorból áll. A 4. sor lényege, hogy visszaugrik a ciklusmag első utasítására, ezzel kényszerítve a számítógépet a ciklusmag újbóli végrehajtására.

Vegyük észre, hogy ez a ciklus végtelen ciklus lesz, hiszen minden egyes alkalommal a 4. sor elérésekor visszaugrunk a 2. sorra, így a program a végtelenségig fut. Ez jellemzően helytelen viselkedés, az algoritmusok, és a programok egyik fontos viselkedési jellemzője, hogy véges sok lépés végrehajtása után leállnak. Módosítsunk a kódon:

```
1. I := 1 // az I változó legyen '1'
2. Ha az I értéke nagyobb, mint 5, akkor ugorj a 6. sorra
3. Írd ki: I, ". négyzetszám = ", I*I
4. I := I+1 // az I változó értéke növelve 1-el
5. ugorj újra a 2. sorra
6. ....
```

A 2. sorba egy feltétel került, mely ha teljesül, akkor befejezzük a ciklusmag végrehajtását, mivel 'kilépünk' a ciklusból, a ciklus utáni első utasításra ugorva.

Hogy ezen feltétel legelső alkalommal is kiértékelhető legyen, az 1. sorban beállítjuk az I változó értékét (értékadás). Valamint szintén figyeljük meg a 4. sort. Ezen sor is nagyon fontos, hiszen az I értéke e nélkül nem érhetné el a kilépéshez szükséges 6 értéket.

A ciklusoknak tehát az alábbi fontos jellemzőjük van:

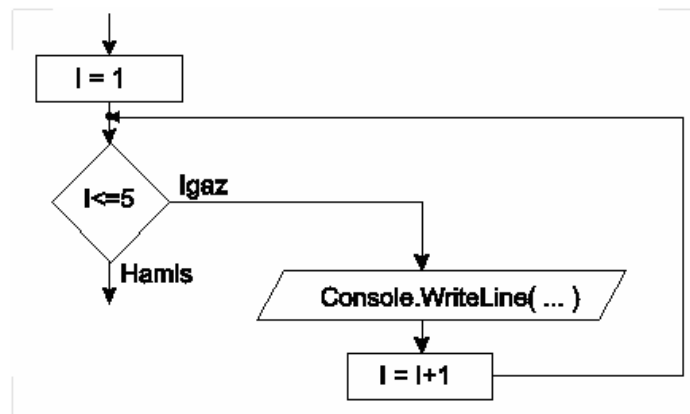
- Tartalmaznak egy utasításblokkot, amelyet akár többször is végrehajtanak (ciklusmag).
- Tartalmaznak egy feltételt (vezérlő feltétel), amely meghatározza, hogy kell-e még ismételni a ciklusmagot .
- A ciklusmag tartalmaz olyan utasítást, amelynek segítségével előbb-utóbb elérjük azt az állapotot, hogy kiléphessünk a ciklusból.
- Tartalmaznak egy kezdőértékadást (a ciklus előtt), amely biztosítja, hogy a ciklus feltétele már legelső alkalommal is egyértelműen kiértékelhető legyen.

A C#-ban a legnépszerűbb ciklus a FOR ciklus:

```
public void Main()
{
    int i;
    for (i=1;i<=5;i=i+1)
    {
        Console.WriteLine("{0}. négyzetszám = {1}",i,i*i);
    }
}
```

A fenti példában a for kulcsszóval jelöljük, hogy ciklus szeretnénk írni. A for kulcsszó után zárójelben három dolgot kell megadni:

- kezdőérték beállítása (i=1),
- ismétlési feltétel (i<=5),
- léptető utasítás (i=i+1).



Formailag a három részt pontosvesszővel kell elválasztani. A ciklusmagot pedig kapcsos zárójelek közé kell tenni (utasításblokk). A C# az alábbi módon értelmezi a for ciklust:

1. végrehajtja a kezdőértékadást,
2. kiértékeli a vezérlő feltételt, és ha HAMIS, akkor kilép a ciklusból,
3. végrehajtja a ciklusmagot,
4. végrehajtja a léptető utasítást,
5. visszaugrik a 2. lépésre.

Ha a vezérlő feltétel igaz, akkor végrehajtja a ciklusmag utasításait, ezért ebben az esetben a vezérlő feltételt a *ciklusban maradás* feltételének is nevezhetjük.

Másik dolog, amit megfigyelhetünk: a vezérlő feltétel kiértékelése már legelső alkalommal is a ciklusmag előtt hajtódik végre. Ennek megfelelően egy hibás kezdőértékadás eredményezheti azt is, hogy a ciklusmag egyetlen egyszer sem hajtódik végre. Például:

```

for (i=10;i<=5;i=i+1)
{
    Console.WriteLine(i);
}
  
```

A 'for' ciklusra jellemző, hogy tartozik hozzá egy ciklusváltozó, melyet az esetek többségében 'i'-vel jelölünk, elsősorban hagyománytiszteletből. Ezen i változó felveszi a kezdőértékét, majd szintén az esetek többségében ezt az értéket minden ciklusmag lefutása után 1-el növeli, amíg túl nem lépjük a végértéket. A for ciklus egy nagyon gyakori alakja tehát felírható az alábbi módon ('cv' jelöli a ciklusváltozót):

```

for (cv = kezdőérték, cv <= végérték, cv = cv+1) ...
  
```


Ennek megfelelően előre kiszámítható, hogy a ciklusmag hányszor hajtódik végre:

```
for (i=1;i<=10;i++)
{
    Console.WriteLine(i);
}
```

Ezen ciklus magja legelső alkalommal az $i=1$, majd $i=2$, ..., $i=10$ értékekre hajtódik végre, összesen 10 alkalommal.

Vegyük az alábbi programozási feladatot: *kérjünk be 3 számot, és írjuk ki a képernyőre a 3 szám összegét.* Az első megoldás még nem tartalmaz ciklust:

```
int a,b,c,ossz;
a = Int32.Parse( Console.ReadLine() );
b = Int32.Parse( Console.ReadLine() );
c = Int32.Parse( Console.ReadLine() );
ossz = a+b+c;
Console.WriteLine("A számok összege ={0}",ossz);
```

Ugyanakkor, ha nem 3, hanem 30 számról lenne szó, akkor már célszerűtlen lenne a program ezen formája. Kicsit alakítsuk át a fenti programot:

```
int a,ossz;
ossz = 0;
a = Int32.Parse( Console.ReadLine() );
ossz = ossz+a;
a = Int32.Parse( Console.ReadLine() );
ossz = ossz+a;
a = Int32.Parse( Console.ReadLine() );
ossz = ossz+a;
Console.WriteLine("A számok összege ={0}",ossz);
```

A fenti program ugyanazt teszi, mint az előző, de már látszik, melyik az utasításcsoport, amelyet 3-szor meg kell ismételni. Ebből már könnyű ciklust írni:

```
int i,a,ossz=0;
for(i=1;i<=3;i=i+1)
{
    a = Int32.Parse( Console.ReadLine() );
    ossz = ossz+a;
}
Console.WriteLine("A számok összege ={0}",ossz);
```

Sokszor ez a legnehezebb lépés, hogy egy már meglévő szekvenciális szerkezetű programban találjuk meg az ismételhető lépéseket, és alakítsuk át ciklusos szerkezetűvé.

Programozási feladat: *Készítsük el az első 100 egész szám összegét.*

```
int i,ossz=0;
for (i=1; i<=100 ; i=i+1)
{
    ossz = ossz + i;
}
Console.WriteLine("Az első 100 egész szám összege
={0}",ossz);
```

Jegyezzük meg, hogy a ciklusokban az $i=i+1$ utasítás nagyon gyakori, mint léptető utasítás. Ezt gyakran $i++$ alakban írják a programozók. A $++$ egy operátor, jelentése 'növelj meg 1-gyel az értékét'. A párja a $--$ operátor, amelynek jelentése: 'csökkentsd 1-gyel az értékét'.

Másik észrevételünk, hogy aránylag gyakori, hogy a ciklusmag egyetlen utasításból áll csak. Ekkor a C# megengedi, hogy ne tegyük ki a blokk kezdet és blokk vége jeleket. Ezért a fenti kis program az alábbi módon is felírható:

```
int i,ossz=0;
for (i=1; i<=100 ; i++ )
    ossz = ossz + i;
Console.WriteLine("Az első 100 egész szám összege
={0}",ossz);
```

Magyarázat: *a program minden lépésben hozzáad egy számot az 'ossz' változóhoz, amelynek induláskor 0 a kezdőértéke. Első lépésben 1-et, majd 2-t, majd 3-t, ..., majd 100-t ad az ossz változó aktuális értékéhez, így állítván elő az első 100 szám összegét.*

Megjegyzés: *a fenti feladat ezen megoldása, és az alkalmazott módszer érdekes tanulságokat tartalmaz. Ugyanakkor a probléma megoldása az első N négyzetszám összegképletének ismeretében egyetlen értékadással is megoldható:*

```
ossz= 100*101 / 2;
```

Programozási feladat: *Határozzuk meg egy szám faktoriálisának értékét. A számot kérjük be billentyűzetről.*

```
int szam,i,fakt=1;
szam = Int32.Parse( Console.ReadLine() );
for (i=1; i<=szam ; i++ )
    fakt = fakt * i;
Console.WriteLine("A(z) {0} szám faktoriálisa = {1}",
szam, fakt );
```

Magyarázat: *pl. a 6 faktoriális úgy számolható ki, hogy $1*2*3*4*5*6$. A program a 'fakt' változó értékét megszorozza először 1-el, majd 2-vel, majd 3-al, ..., végül magával a számmal, így számolva ki lépésről lépésre a végeredményt.*

Megjegyzés: a 'szam' növelésével a faktoriális értéke gyorsan nő, ezért csak kis számokra (kb. 15-ig) tesztelhetjük csak a fenti programot.

Programozási feladat: *Határozzuk meg egy szám pozitív egész kitevőjű hatványát! A számot, és a kitevőt kérjük be billentyűzetről!*

```
int szam,kitevo,i,ertek=1;
szam = Int32.Parse( Console.ReadLine() );
kitevo = Int32.Parse( Console.ReadLine() );
for (i=1; i<=kitevo ; i++ )
    ertek = ertek * szam;
Console.WriteLine("A(z) {0} szám {1}. hatványa = {2}",
    szam,kitevo,ertek);
```

Magyarázat: *pl. 4 harmadik hatványa kiszámolható úgy, hogy $4*4*4$. A program a 'ertek' változó értékét megszorozza annyiszor a 'szam'-al, ahányszor azt a kitevő előírja.*

Programozási feladat: *Határozzuk meg egy szám osztóinak számát! A számot kérjük be billentyűzetről!*

```
int szam,i,db=0;
szam = Int32.Parse( Console.ReadLine() );
for (i=1; i<=szam ; i++ )
    if (szam%i==0) db++;
Console.WriteLine("A(z) {0} szám osztóinak száma:
{1}.", szam, db);
```

Magyarázat: *a 10 lehetséges osztói az 1..10 tartományba esnek. A módszer lényege, hogy sorba próbálkozunk a szóba jöhető számokkal, mindegyiket ellenőrizve, hogy az osztója-e a számnak, vagy sem. Az oszthatóságot a '%' operátorral ellenőrizzük. A % operátor meghatározza a két operandusának osztási maradékát (pl. a $14\%4$ értéke 2 lesz, mert 14 osztva 4-gyel 2 osztási maradékot ad). Amennyiben az osztási maradék 0, úgy az 'i' maradék nélkül osztja a 'szam' változó értékét, ezért az 'i' változóban lévő érték egy osztó. Minden osztó-találat esetén növeljük 1-gyel a 'db' változó értékét.*

Programozási feladat: *Írassuk ki a képernyőre egy szám összes többszörösét, amelyek nem nagyobbak, mint 100, csökkenő sorrendben! A számot kérjük be billentyűzetről!*

```
int szam,i;
szam = Int32.Parse( Console.ReadLine() );
for (i=100; i>=szam ; i-- )
    if (i % szam == 0) Console.WriteLine("{0} ",i);
```

Magyarázat: *elindulunk 100-tól, visszafele haladunk, minden lépésben csökkentve a ciklusváltozónk értékét (i--). Minden lépésben megvizsgáljuk, hogy a szóban forgó 'i' többszöröse-e az eredeti számnak. Ez akkor teljesül, ha a szám maradék nélkül*

osztja az 'i'-t. Valahányszor találunk ilyen 'i' értéket, mindannyiszor kiírjuk azt a képernyőre.

Ugyanezen feladat megoldható hatékonyabban, ha figyelembe vesszük, hogy ezen számok egymástól 'szam' távolságra vannak:

```
int szam,i,max;
szam = Int32.Parse( Console.ReadLine() );
max = (100/szam)*szam;
for (i=max; i>=szam ; i=i-szam )
    if (i % szam == 0) Console.WriteLine("{0} ",i);
```

Magyarázat: a '100/szam' osztási művelet, mivel két egész szám típusú érték között kerül végrehajtásra automatikusan egész osztásnak minősül. Ha a 'szam' értéke pl. 7, akkor a '100/szam' eredménye 14. Ha ezt visszaszorozzuk a 'szam'-al, akkor megkapjuk a legnagyobb olyan értéket, amely biztosan többszöröse a 'szam'-nak, és nem nagyobb mint 100, és a legnagyobb ilyen többszöröse a 'szam'-nak, amely nem nagyobb mint 100 ('max'). Ez lesz a kiinduló értéke a ciklusváltozónak. A további többszörösöket úgy kapjuk, hogy a kiinduló értéket minden cikluslépésben csökkentjük 'szam'-al.

Feladatok:

1. **Programozási feladat:** Állapítsuk meg egy billentyűzetről bekért számról, hogy prímszám-e! A prímszámoknak nincs 1 és önmagán kívül más osztója.
2. **Programozási feladat:** Állapítsuk meg két billentyűzetről bekért számról, hogy mi a legnagyobb közös osztójuk! A legnagyobb olyan szám, amely mindkét számot osztja. Ezen értéket meghatározhatjuk kereséssel (ciklus), vagy az Euklideszi algoritmussal is.
3. **Programozási feladat:** Állapítsuk meg két billentyűzetről bekért számról, hogy relatív prímek-e! Akkor relatív prímek, ha a legnagyobb közös osztójuk az 1.
4. **Programozási feladat:** Állítsuk elő egy szám prímtényezős felbontását! Pl: $360=2*2*2*3*3*5!$
5. **Programozási feladat:** Állapítsuk meg, hogy egy adott intervallumba eső számok közül melyik a legnagyobb prímszám! Az intervallum alsó és felső határának értékét kérjük be billentyűzetről! Próbáljunk keresni idő-hatékony megoldásokat!
6. **Programozási feladat:** Írjunk olyan programot, amely egy összegző ciklussal kiszámolja és kiírja az alábbi számtani sorozat első 20 elemének összegét: 3,5,7,9,11,stb.! Ellenőrizzük le az eredményt a számtani sorozat összegképlete segítségével!
7. **Programozási feladat:** Írjunk olyan programot, amely kiszámolja és kiírja az alábbi változó növekményű számtani sorozat első 20 elemének összegét: 3,5,8,12,17,23,30,stb.!
8. **Programozási feladat:** Írjunk olyan programot, amely bekéri egy tetszőleges számtani sorozat első elemét, és a differenciát! Ezek után kiírja a képernyőre a számtani sorozat első 20 elemét, az elemeket egymástól vesszővel elválasztva, egy sorban!
9. **Programozási feladat:** Írjunk olyan programot, amely bekéri egy tetszőleges mértani sorozat első elemét, és a kvóciensét! Ezek után kiírja a képernyőre a mértani sorozat első 20 elemét, és az elemek összegét!
10. **Programozási feladat:** Számoljuk ki és írjuk ki a képernyőre a 2^n értékeit $n=1,2,...,10$ -re!
11. **Programozási feladat:** Számoljuk ki és írjuk ki a képernyőre az $a_n=a_{n-1}+2^n$ sorozat első 10 elemét, ha $a_1=1!$

12. **Programozási feladat:** Írjunk olyan programot, amely addig írja ki a képernyőre a $a_n = 2^n - 2^{n-1}$ sorozat elemeit a képernyőre, amíg a sorozat következő elemének értéke meg nem haladja az 1000-t! A sorozat első eleme: $a_1 = 1$!
13. **Programozási feladat:** Határozzuk meg az első n négyzetszám összegét! N értékét kérjük be billentyűzetről!
14. **Programozási feladat:** Határozzuk meg egy $[a, b]$ intervallum belsejébe eső négyzetszámokat (írjuk ki a képernyőre), és azok összegét! Az a és b értékét kérjük be billentyűzetről!
15. **Programozási feladat:** Számoljuk ki és írjuk ki a képernyőre a Fibonacci sorozat első 10 elemét! A sorozat az alábbi módon számítható ki:
- $$\begin{aligned} a_1 &= 1 \\ a_2 &= 1 \\ a_n &= a_{n-1} + a_{n-2} \quad \text{ha } n > 2 \end{aligned}$$

Programozás tankönyv

VII. Fejezet

„Vektorok!”

Radványi Tibor

Vektorok kezelése

A tömb egy összetett homogén adatstruktúra, melynek tetszőleges, de előre meghatározott számú eleme van. Az elemek típusa azonos. A tömb lehet egy vagy többdimenziós, a dimenzió száma nem korlátozott. Fejezetünkben az egydimenziós tömbök kezelésével foglalkozunk, melynek alapján a többdimenziós tömbök kezelése is elsajátítható.

Tömb deklarálása

Hasonlóan történik, mint egy hagyományos változóé, csak a [] jellel kell jelezni, hogy tömb következik:

```
int[] tm;
```

Ilyenkor a tömbünk még nem használható, hiszen a memóriában még nem történt meg a helyfoglalás. Ezt a new operátor használatával tehetjük meg.

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

vagy egyszerűen:

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

Ezzel létrehoztunk egy 5 elemű, **int** típusú tömböt. Használhatjuk, feltölthetjük, műveleteket végezhetünk vele. A fentiekhez hasonlóan tudunk string, valós, stb tömböt deklarálni:

```
string[] stm = new string[5]; // 5 elemű string tömb  
double[] dtm = new double[10]; // 10 elemű valós tömb  
bool[] btm = new bool[7]; // 7 elemű logikai tömb
```

A tömb elemeire indexeik segítségével hivatkozhatunk. Ez mindig 0-tól indul és egyesével növekszik.

```
tm[0] = 1; tm[1] = 2; tm[2] = 17; tm[3] = 90; tm[4] = 4;  
stm[0] = "Első";  
btm[0] = true;
```

Megtehetjük, hogy a tömb deklarációjakor meg adjuk az elemeit, ha van rá lehetőségünk és ismeretünk:


```
int[] tm2 = new int[3] {3,4,5};  
string[] stm2 = new string[3] {"első","második","harmadik"};
```

Ha az előbbi módon használjuk a tömb deklarációt, azaz rögtön megadjuk a kezdőértékeket is, akkor a new operátor elhagyható:

```
int[] tm3 = {1,2,3};  
string[] stm3 = {"aaa","bbb","ccc"};
```

A tömb elemeinek elérése

A tömb elemein végiglépkedhetünk egy egyszerű for ciklus alkalmazásával, vagy használhatjuk a foreach ciklust, ha általános függvényt szeretnénk írni.

```
int i;  
for (i = 0; i < 5; i++)  
{  
    tm[i] = 1; // alapértéket ad a tm tömb elemeinek  
}  
  
for (i = 0; i < tm.Length; i++)  
{  
    tm[i] = 2 * tm[i]; // duplázza a tm tömb elemeit  
}  
  
foreach (int s in tm)  
{  
    System.Console.WriteLine(s.ToString()); // kiírja a tm tömb  
    elemeit  
}
```

A **length** tulajdonság segítségével megkapjuk a tömb méretét, melyet akár ciklusban is használhatunk paraméterként.

A tömb elemeinek rendezése, keresés a tömbben

A tömb kiíratásához használhatunk egy egyszerű metódust:

```
private static void ShowArray(int[] k)
{
    foreach (int j in k)
        System.Console.Write(j.ToString()+" ");
    System.Console.WriteLine();}
```

Ennek a segítségével vizsgáljuk meg az Array osztály lehetőségeit.

A Sort metódus, melynek egy paramétere van, helyben rendezi a paraméterként kapott tömböt.

Deklarációja:

```
public static void Sort(Array array);
```

Használata:

```
tm[0] = 1; tm[1] = 2; tm[2] = 17; tm[3] = 90; tm[4] = 4;
ShowArray(tm);
Array.Sort(tm);
ShowArray(tm);
```

Eredmény:

Az első kiíratáskor:

1 2 17 90 4

A második kiíratáskor:

1 2 4 17 90

Ebben az esetben a teljes tömb rendezésre kerül. Ha a Sort metódus három paraméteres változatát használjuk, akkor a tömb egy résztömbjét rendezhetjük.

Deklarációja:

```
public static void Sort(Array array, int index, int length);
```

Ekkor az **index** paramétertől kezdve a **length** paraméterben megadott hosszban képződik egy részhalmaza a tömb elemeinek, és ez kerül rendezésre.

Használata:

```
tm[0] = 1; tm[1] = 2; tm[2] = 17; tm[3] = 90; tm[4] = 4;  
    ShowArray(tm);  
    Array.Sort(tm, 3, 2);  
    ShowArray(tm);
```

Eredmény:

Az első kiíratáskor:

1 2 17 90 4

A második kiíratáskor:

1 2 17 4 90

Tehát csak a 90 és a 4 cserélt helyet, mert csak az utolsó két elem rendezését írják elő a paraméterek (3,2).

A rendezés következő esete, amikor két tömb paramétert használunk.

Deklarációja:

```
public static void Sort( Array keys, Array items);
```

Keys : egy egydimenziós tömb, mely tartalmazza a kulcsokat a rendezéshez

Items: elemek, melyek a Keys tömb elemeinek felelnek meg.

A metódus rendezi a keys tömb elemeit, és ennek megfelelően az items tömb elemeinek sorrendje is változik.

Használata:

A string tömb elemeinek kiíratásához használt metódus:

```
private static void ShowArrayST(string[] k)
{
    foreach (string j in k)
        System.Console.Write(j+" ");
    System.Console.WriteLine();
}

tm[0] = 1; tm[1] = 34; tm[2] = 17; tm[3] = 90; tm[4] = 4;
string[] stm2 = new string[5] {"első","második","harmadik", "negyedik",
"ötödik"};

ShowArray(tm);
ShowArrayST(stm2);
Array.Sort(tm, stm2);
ShowArray(tm);
ShowArrayST(stm2);
```

Eredmény az első kiíratás után:

1 34 17 90 4

első második harmadik negyedik ötödik

Eredmény a második kiíratás után:

1 4 17 34 90

első ötödik harmadik második negyedik

Az előző kulcsos rendezéshez hasonlóan, de csak részalmaz rendezésre használható az alábbi metódus:

Deklaráció:

```
public static void Sort(Array keys, Array items, int index, int
length);
```

keys: egy egydimenziós kulcsokat tartalmazó tömb

items: egy egydimenziós, a kulcsokhoz tartozó elemeket tartalmazó tömb

index: a rendezendő terület kezdőpozíciója

length: a rendezendő terület hossza

Használata:

```
tm[0] = 1; tm[1] = 34; tm[2] = 17; tm[3] = 90; tm[4] = 4;
string[] stm2 = new string[5] {"első","második","harmadik", "negyedik",
"ötödik"};

ShowArray(tm);
ShowArrayST(stm2);

Array.Sort(tm,stm2,3,2);

ShowArray(tm);
ShowArrayST(stm2);
```

Eredmény az első kiíratás után:

1 34 17 90 4

első második harmadik negyedik ötödik

Eredmény a második kiíratás után:

1 34 17 4 90

első második harmadik ötödik negyedik

Következzen néhány alapeladat, melyenek megoldásához tömböket használunk.

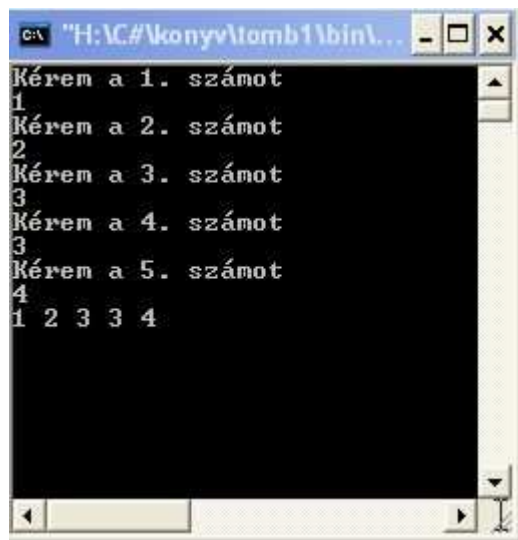
Vektor feltöltése billentyűzetről

Ebben az egyszerű példában bekérünk 5 db egész számot a billentyűzetről. Mivel a `Console.ReadLine()` függvény visszatérési értéke `string`, ezért ezt át kell alakítani a megfelelő formátumra a `Convert` osztály `ToInt32` metódusának a segítségével.

Semmiféle ellenőrzést nem tartalmaz a kód, így csak egész számok esetén működik jól. Ha szám helyett szöveget, például „almafa” írunk az adatbekérés helyére, akkor a program hibaüzenettel leáll.

```
int[] tm = new int[5];
int i;
for (i=0; i<5; i++)
{
    Console.WriteLine("Kérem a {0}. számot",i+1);
    tm[i] = Convert.ToInt32(Console.ReadLine());
}
ShowArray(tm);
```

A futás eredménye:



Módosítsuk úgy a programot, hogy csak olyan értékeket fogadjon el, melyek még eddig nem szerepeltek. Azaz kérjünk be a billentyűzetről 5 db különböző egész számot.

```

int[] tm = new int[5];
int i, j;
bool nem_volt = true;
for (i=0; i<5; i++)
{
    Console.WriteLine("Kérem a {0}. számot",i+1);
    tm[i] = Convert.ToInt32(Console.ReadLine());
    nem_volt = true;
    j = 0;
    for (j = 0; j < i; j++)
        if (tm[i] == tm[j])
        {
            i--;
            break;
        }
}
ShowArray(tm);

```

A futás eredménye:

```

C:\> "H:\C#\konyvtomb1\bin\Debug\to...
Kérem a 1. számot
1
Kérem a 2. számot
2
Kérem a 3. számot
2
Kérem a 3. számot
2
Kérem a 3. számot
2
Kérem a 3. számot
3
Kérem a 4. számot
4
Kérem a 5. számot
5
1 2 3 4 5

```

Vektor feltöltése véletlenszám-generátorral

Nézzük meg az előző feladatot, de most használjuk a tömb feltöltéséhez a véletlenszám-generátort. Ebben az esetben biztosítjuk, hogy az értékek típushelyesek legyenek.

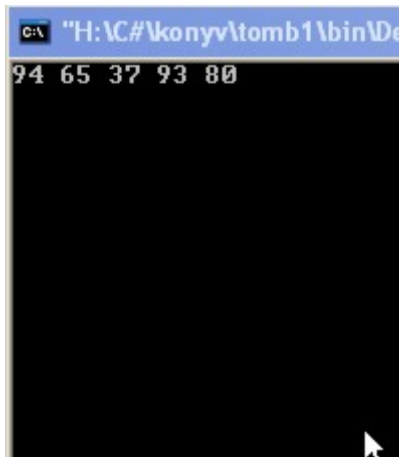
A véletlenszám-generátort a **Random** osztályból érjük el. Készítsünk egy **rnd** nevű példányt belőle. Használatkor az **rnd** példány paramétereként adjuk meg a felső határát a generálandó számnak.

```
int[] tm = new int[5];
int i, j;
bool nem_volt;
Random rnd = new Random(); //Egy példány a Random osztályból
for (i=0; i<5; i++)
{
    tm[i] = rnd.Next(100);
}
ShowArray(tm);
```

Oldjuk meg ezt a feladatot is úgy, hogy ne engedjük meg az ismétlődést a generált számok között.

```
int[] tm = new int[5];
int i, j;
bool nem_volt;
Random rnd = new Random(); //Egy példány a Random osztályból
for (i=0; i<5; i++)
{
    tm[i] = rnd.Next(100);
    nem_volt = true;
    j = 0;
    for (j = 0; j < i; j++)
        if (tm[i] == tm[j])
        {
            break;
        }
}
ShowArray(tm);
```


A futás eredménye:



N elemű vektorok kezelése

Egyszerű programozási tételek alkalmazása vektorokra

Összegzés

Feladat: Adott egy 10 elemű, egész típusú tömb. Töltsük fel véletlen számokkal, majd határozzuk meg a számok összegét.

Megoldás:

```
int[] tm = new int[10];
int i, sum = 0;
Random rnd = new Random();
for (i=0 ; i<10; i++)
{
    tm[i] = rnd.Next(100,200);
    Console.Write("{0}  ",tm[i]);
}
Console.WriteLine();
for (i=0; i<10; i++)
    sum += tm[i];
Console.WriteLine("A számok összege: {0}", sum);
```

Magyarázat:

Az első sorban deklarálunk egy **int** típusú elemeket tartalmazó tömböt, melynek a neve **tm**. Rögtön le is foglaljuk a memóriában 10 elem helyét.

Az **i** ciklusváltozó, a **sum** a gyűjtőváltozó lesz, melyben a tömb elemeinek az összegét generáljuk. Ezért ennek 0 kezdőértéket kell adni.

Létrehozunk a **Random** osztályból egy **rnd** nevű példányt. Ennek a segítségével fogjuk előállítani a véletlen számokat. Az első **for** ciklusban feltöltjük a tömböt a véletlen számokkal.

Az összegzést a második **for** ciklus végzi, végiglépkedve az elemeken, és azok értékével növeli a **sum** változó pillanatnyi értékét. Majd kiíratjuk a végeredményt.

Maximum és minimum kiválasztása

Feladat: Adott egy 10 elemű, egész számokat tartalmazó tömb. Töltsük fel véletlen számokkal, majd határozzuk meg a legnagyobb illetve legkisebb elem értékét.

Megoldás:

```
int[] tm = new int[10];
int i, max, min;
Random rnd = new Random();
for (i=0 ; i<10; i++)
{
    tm[i] = rnd.Next(100,200);
    Console.Write("{0}  ",tm[i]);
}
Console.WriteLine();
max = tm[0]; min = tm[0];
for (i = 1; i < 10; i++)
{
    if (tm[i] > max) max = tm[i];
    if (tm[i] < min) min = tm[i];
}
Console.WriteLine("A számok minimuma: {0},  maximuma: {1}", min, max);
```

Magyarázat:

A program eleje hasonló az összegzésnél látottakkal. Egy max és egy min változót is deklarálunk. Itt fogjuk megjegyezni az aktuális legnagyobb és legkisebb elemet. A példában az elem értékét jegyezzük meg, de van rá lehetőség, hogy a tömbindexet tároljuk el, attól függően, hogy a feladat mit követel meg tőlünk.

Az értékek feltöltése után a következő **for** ciklussal végignézzük az elemeket, és ha találunk az aktuális szélsőértéknél nagyobb illetve kisebb tömbértéket, akkor onnan kezdve az lesz a max, vagy a min értéke.

Eldöntés

Feladat: Feladat: Adott egy 10 elemű, egész számokat tartalmazó tömb. Töltsük fel véletlen számokkal, majd kérjünk be egy egész számot a billentyűzetről. Döntsük el hogy a bekért szám szerepel e a 10 elemű tömbben.

Megoldás:

```
int[] tm = new int[10];
int i;
Random rnd = new Random();
for (i=0 ; i<10; i++)
{
    tm[i] = rnd.Next(100,200);
    Console.Write("{0}  ",tm[i]);
}
Console.WriteLine();
int szam = Int32.Parse(Console.ReadLine());
i = 0;
while (i < 10 && szam != tm[i]) i++;
if (i < 10) Console.WriteLine("A szám szerepel a tömbben");
    else Console.WriteLine("A szám nem szerepel a tömbben");
Console.ReadLine();
```

Magyarázat:

A tömb feltöltése után kérjünk be egy számot a billentyűzetről. Ügyelve a konverzió szükségességére. Ehhez itt a **Int32.Parse** függvényt használtuk.

Az eldöntéshez egy **while**, előtesztelő ciklust használunk összetett logikai feltétellel. Az első feltétel azt ellenőrzi, hogy benne vagyunk e még a tömbben, a második, hogy még nem találtuk meg a keresett értéket. Mivel logikai **ÉS** kapcsolat van a két feltétel között, így a ciklusmag akkor hajtódik végre, ha mindkét feltétel igaz. A ciklusmagban az *i* ciklusváltozó értékét növeljük. Ha bármelyik feltétel hamissá válik, akkor kilépünk a ciklusból. Így utána az **if** feltétellel döntjük el a kilépés okát, azaz hogy megtaláltuk a szám első előfordulását, vagy elfogyott a tömb. Ennek megfelelően írjuk ki üzenetünket. Figyeljünk oda, hogy az **if** feltételben nem alkalmazhatjuk a **while** feltétel második részét, mert ha nincs ilyen tömbem, akkor a vizsgálatkor már *i* = 11 lenne, így nem létező tömbemre hivatkoznánk.

Kiválogatás

Feladat: Adott egy 10 elemű, egész számokat tartalmazó tömb. Töltsük fel véletlen számokkal, majd írassuk ki a páros számokat közülük.

Megoldás:

```
int[] tm = new int[10];
int i;
Random rnd = new Random();
for (i=0 ; i<10; i++)
{
    tm[i] = rnd.Next(100,200);
    Console.Write("{0} ",tm[i]);
}
Console.WriteLine();
for (i = 0; i < 10; i++)
    if (tm[i] % 2 == 0) Console.WriteLine("{0} ",tm[i]);
Console.ReadLine();
```

Magyarázat:

A tömb feltöltése után a második cikluson belül írjuk a megfelelő feltételt az **if** szerkezetbe. Ennek teljesülése esetén kiírjuk a tömbemet. A feltétel, a párosság

vizsgálata. Ezt az maradékképzés % operátorával vizsgáljuk. Ha a tömbelem 2-vel osztva 0 maradékot ad, akkor a szám páros. Ebben az esetben kerül kiíratásra.

Dinamikus méretű vektorok

Dinamikus elemszámú tömbök kezelését valósítja meg az **ArrayList** osztály.

Deklarálás:

```
ArrayList arl = new ArrayList();
```

Az ArrayList főbb jellemzői:

Capacity: az elemek számát olvashatjuk illetve írhatjuk elő.

Count: az elemek számát olvashatjuk ki. Csak olvasható tulajdonság.

Item: A megadott indexű elemet lehet írni vagy olvasni.

Az ArrayList főbb metódusai:

Add: Hozzáad egy objektumot a lista végéhez.

BinarySearch: Rendezett listában a bináris keresés algoritmusával megkeresi az elemet

Clear: Minden elemet töröl a listából.

CopyTo: Átmásolja az ArrayList-át egy egydimenziós tömbbe.

IndexOf: A megadott elem első előfordulásának indexét adja vissza. A lista első elemének az indexe a 0.

Insert: Beszúr egy elemet az ArrayListbe a megadott indexhez.

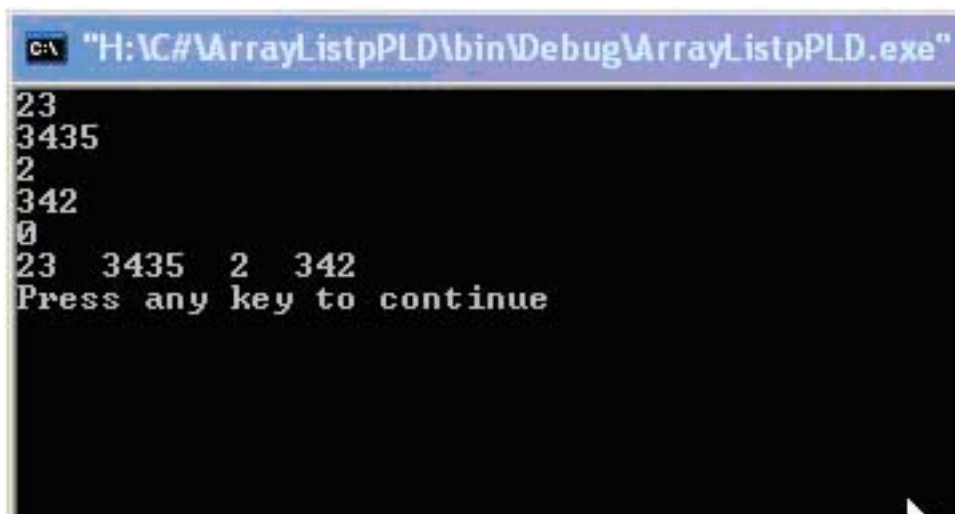
Sort: Rendezi az elemeket az ArrayListben.

Feladatok dinamikus tömbre

Kérjünk be számokat a nulla végjelig

```
ArrayList dtm = new ArrayList();  
int i = 0, elem;  
do  
{  
    elem = Int32.Parse(Console.ReadLine());  
    if (elem != 0) dtm.Add(elem);  
}  
while (elem != 0);
```

Deklarálunk egy `dtm` nevű **ArrayList** objektumot, és példányosítjuk is. Az adatbekéréskor a **ReadLine()** függvény által visszaadott karakterláncot konvertáljuk `Int32` típusúvá, és a konverzió eredményét tároljuk az `elem` nevű változóba. Amennyiben ez nem nulla értékű, hozzáadjuk az `ArrayList`ánkhoz az `add` metódus meghívásával. Az adatbekérést mindaddig folytatjuk, míg nullát nem írunk be.



A 0 beírása után az ellenőrzés miatt vissza írtuk az `ArrayList` tartalmát az alábbi függvény meghívásával:

```
static void Kiir(ArrayList al)  
{  
    foreach (int a in al)  
    {  
        Console.Write("{0} ", a);  
    }  
}
```

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

Feladat:

Adott a síkon koordinátaival néhány pont. Határozzuk meg azt a minimális sugarú kört, mely magába foglalja az összes pontot. adjuk meg a középpontját és a sugarát!

A feladat első felét nézzük most. Hogyan kezeljük a pontokat, és adatait.

```
class TPont  
{  
    public double x, y;  
}
```

Ezzel létrehoztunk egy TPont osztályt, mely a síkbeli pont két koordinátáját, mint adatmezőt tartalmazza. Ezen az osztályon fog alapulni a feladat megoldása.

Hozzunk létre egy kört leíró osztályt is. Ehhez a kör középpontjának koordinátáira és a sugarának hosszára lesz szükségünk. Használjuk fel a már meglévő TPont osztályt.

```
class TKor  
{  
    public TPont kozepont = new TPont();  
    public double sugar;  
}
```

A feladatunk megkeresni azt a két pontot, melyek távolsága a legnagyobb, és az általuk meghatározott szakasz, mint átmérő fölé rajzolt kört adatait meghatározni. Ehhez használjuk a következő osztályt, ahol egy végpontjainak koordinátaival adott szakasz fölé rajzolt kör adatait tudjuk meghatározni.


```

class TSzakasz_kore : TKor
{
public TPont Kezd = new TPont();
public TPont Veg = new TPont();

public void szakasz_fole_kor()
{
    kozeppont.x = (Veg.x + Kezd.x)/2;
    kozeppont.y = (Veg.y + Kezd.y)/2;
    System.Console.WriteLine("{0}    {1}",kozeppont.x,kozeppont.y);
    sugar = System.Math.Sqrt((Veg.x-Kezd.x)*(Veg.x-Kezd.x)+
                               (Veg.y-Kezd.y)*(Veg.y-Kezd.y))/2;
}
}

```

A kozeppont változó a TKor osztályan lett deklarálva, az öröklés miatt ez az osztály is látja, és a public láthatósága miatt használhatja is.

Ezek után minden eszközünk megvan ahhoz, hogy megoldjuk a feladatot. Tekintsük az alábbi deklarációt, mely lehetővé teszi tetszőleges számú pont felvételét a síkon. Itt használjuk ki a dinamikus ArrayList adta lehetőségeket.

```

public long pontok_szama = 0;
public System.Collections.ArrayList pontok =
    new System.Collections.ArrayList();

```

A pontok nevű ArrayListbe fogjuk tárolni a pontok adatait. Ezeket feltölthetjük billentyűzetről:

```

public void pontok_beo()
{
    int i = 1;
    while (i<=pontok_szama)
    {
        TPont pont = new TPont();
        System.Console.WriteLine("Kérem az {0}. pont X koordinátáját ... ", i);
        pont.x = System.Convert.ToDouble(System.Console.ReadLine());
        System.Console.WriteLine("Kérem az {0}. pont Y koordinátáját ... ", i);
        pont.y = System.Convert.ToDouble(System.Console.ReadLine());
        pontok.Add(pont);
        i++;
    }
    System.Console.WriteLine("Koordináták beolvasva...");
}

```

Keressük meg a maximális távolságra lévő pontokat a ponthalmazból. Ehhez lehet ezt az egyszerű metódust használni:

```

private void max_tav_pontok()
{
    double maxtav = 0, tav;
    foreach (TPont pont1 in pontok)
    {
        foreach (TPont pont2 in pontok)
        {
            tav = System.Math.Sqrt((pont2.x-pont1.x)*(pont2.x-pont1.x)+
                                   (pont2.y-pont1.y)*(pont2.y-pont1.y));
            if (maxtav < tav)
            {
                maxtav = tav;
                Kezd.x = pont1.x;
                Kezd.y = pont1.y;
                Veg.x = pont2.x;
                Veg.y = pont2.y;
            }
        }
    }
}

```

Ha a fenti metódusokat egy olyan osztályban hozzuk létre, melyet a TSzakasz_kore osztályból származtattunk, akkor a megoldást mindössze két metódus meghívása szolgáltatja:

```
public void kor_megadasa()
{
    max_tav_pontok();
    szakasz_fole_kor();
}
```

Ekkor a Main függvény tartalma a következő lehet:

```
System.Console.WriteLine("\n\n Sok pont esete: ");
TSik_pontokat_foglalo_kor sk = new TSik_pontokat_foglalo_kor();
System.Console.WriteLine(" --- Kérem a pontok számát: --- ");
int p_count = System.Convert.ToInt32(System.Console.ReadLine());
sk.pontok_szama = p_count;
sk.pontok_beo();
sk.kor_megadasa();

System.Console.WriteLine("A kör középpontja O( {0} , {1} )és
sugara: R = {2}",
                sk.kozeppont.x, sk.kozeppont.y, sk.sugar);
```

Ezzel a feladatot megoldottuk.

Többdimenziós tömbök

Sok feladat megoldásához az egydimenziós tömb struktúra már nem elegendő, vagy túl bonyolulttá tenné a kezelést. Ha például gyűjteni szeretnénk havonként és azon belül naponként a kiadásainkat. A kiadások számok, de a napokhoz rendelt indexek nehézkesek lennének, mert február 1-hez a 32-t, június 1-hez már a 152-es index tartozna. Egyszerűbb, ha a kiadásainkat két indexel azonosítjuk. Az egyik jelentheti a hónapot, a másik a napot. Ha már az évet is szeretnénk tárolni, akkor egy újabb index bevezetésére van szükségünk.

Nézzük meg egy egyszerű példán keresztül a deklarációt. Töltsünk fel egy 3x3 mátrixot véletlen számokkal és írassuk ki őket mátrix formában a képernyőre.

```
static void Main(string[] args)
{
    int[,] tm = new int[3,3];
    int i, j ;
    Random rnd = new Random();
    for (i = 0; i < 3; i++)
    {
        for (j = 0; j < 3; j++)
        {
            tm[i,j] = rnd.Next(10,20);
            Console.Write("{0}  ",tm[i,j]);
        }
        Console.WriteLine();
    }
    Console.ReadLine();
}
```

Látható, hogy a deklarációkor a két indexet úgy jelezzük, hogy egy vesszőt teszünk a zárójelbe, majd a példányosításkor megadjuk az egyes indexekhez tartozó elemszámot.

A kétdimenziós tömböt feltölteni elemekkel, illetve kiíratni azokat elég két ciklus, ahol az egyik ciklusváltozó az egyik indextartományt futja végig, míg a másik ciklusváltozó

a másik tartományt. Ha nem két, hanem többdimenziós tömböt használunk, akkor a ciklusok száma ennek megfelelően fog nőni.

A példában látható, hogy a mátrix forma megtartás érdekében egyszerűen a sorok elemeit egymás mellé **Write** utasítás alkalmazásával írtuk ki. A belső ciklus minden sor kiírásáért felelős, így a sorok végén a soremelésről gondoskodtunk a **WriteLine** utasítással, melynek nem volt paramétere.

A kétdimenziós tömböt feltölthetjük kezdőértékkel, hasonlóan az egydimenziós esethez:

```
int[,] tm2 = new int[2,2] {{1,2},{3,4}};
```

Tekintsük a következő feladatot:

Töltsünk fel egy 3x3 mátrixot a billentyűzetről egész számokkal. Írassuk ki a szokásos formátumban, majd generáljuk a transzponáltját, és ezt is írassuk ki.

A szükséges tömb és ciklusváltozók deklarálása:

```
int[,] tm = new int[4,4];  
int i, j ;
```

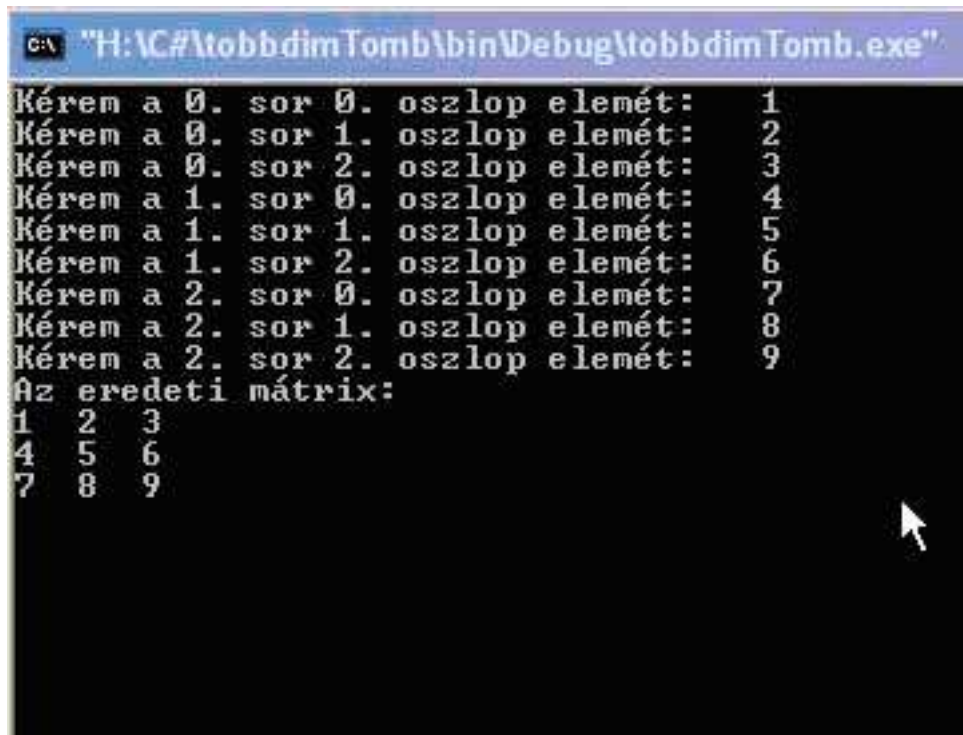
Töltsük fel a mátrixot billentyűzetről egész számokkal:

```
for (i = 0; i < 3; i++)  
    for (j = 0; j < 3; j++)  
    {  
        Console.Write("Kérem a {0}. sor {1}. oszlop elemét: ", i, j);  
        tm[i,j] = int.Parse(Console.ReadLine());  
    }
```

Írassuk ki a tömb elemeit mátrix formában:

```
Console.WriteLine("Az eredeti mátrix: ");  
for (i = 0; i < 3; i++)  
{  
    for (j = 0; j < 3; j++)  
        Console.Write("{0} ", tm[i,j]);  
    Console.WriteLine();  
}
```

Ez futás közben így néz ki:



```
C:\H:\C#\tobbdimTomb\bin\Debug\tobbdimTomb.exe
Kérem a 0. sor 0. oszlop elemét: 1
Kérem a 0. sor 1. oszlop elemét: 2
Kérem a 0. sor 2. oszlop elemét: 3
Kérem a 1. sor 0. oszlop elemét: 4
Kérem a 1. sor 1. oszlop elemét: 5
Kérem a 1. sor 2. oszlop elemét: 6
Kérem a 2. sor 0. oszlop elemét: 7
Kérem a 2. sor 1. oszlop elemét: 8
Kérem a 2. sor 2. oszlop elemét: 9
Az eredeti mátrix:
1 2 3
4 5 6
7 8 9
```

Deklaráljunk egy tr tömböt a transzponált mátrix elemeinek. Majd generáljuk le az eredeti **tm** tömbből. A transzponált mátrixot az eredeti mátrix sorainak és oszlopainak felcserélésével kapjuk.

```
int[,] tr = new int[3,3];
for (i = 0; i < 3; i++)
    for (j = 0; j < 3; j++)
        tr[i,j] = tm[j,i];
```

Újra ki kell íratni egy tömböt, most a transzponált mátrixot.

```
for (i = 0; i < 3; i++)
{
    for (j = 0; j < 3; j++)
        Console.Write("{0} ",tr[i,j]);
    Console.WriteLine();
}
```

A program futtatását bemutató képernyő:

```
C:\ "H:\C#\tobbdimTomb\bin\Debug\tobbdimTomb.exe"
Kérem a 0. sor 0. oszlop elemét: 1
Kérem a 0. sor 1. oszlop elemét: 2
Kérem a 0. sor 2. oszlop elemét: 3
Kérem a 1. sor 0. oszlop elemét: 4
Kérem a 1. sor 1. oszlop elemét: 5
Kérem a 1. sor 2. oszlop elemét: 6
Kérem a 2. sor 0. oszlop elemét: 7
Kérem a 2. sor 1. oszlop elemét: 8
Kérem a 2. sor 2. oszlop elemét: 9
Az eredeti mátrix:
1 2 3
4 5 6
7 8 9
A transzponált mátrix:
1 4 7
2 5 8
3 6 9
```

Vegyük észre, hogy a két kiíratásban csak a tömb neve a különböző. Feleslegesen írjuk le kétszer ugyanazt a kódot. Erre jobb megoldást ad az eljárásokról és függvényekről szóló fejezet.

További megoldásra ajánlott feladatok

1.

Határozzuk meg N db pozitív egész szám harmonikus közepét!

2.

Határozzuk meg két N dimenziós vektor skaláris szorzatát, ha adottak a két vektor koordinátái.

3.

5 km-enként megmértük a felszín tengerszint feletti magasságát. (összesen N mérést végeztünk) A méréseket szárazföld felett kezdtük és fejeztük be. Ott van tenger, ahol a mérés eredménye 0, másutt >0 . Határozzuk meg, van-e ezen a tengerszakaszon sziget!

4.

Az előző feladatból mondjuk meg

- a. Hány szigeten jártunk?
- b. Adjuk meg azt a szigetet, ahol a legmagasabb csúcs van!
- c. Határozzuk meg a tengerszakaszok hosszát!
- d. Állapítsuk meg hogy van-e két egyforma szélességű sziget!

5.

Egy nyilvántartásban a személyekről négy adatot tartanak nyilván: magasság, szemszín, hajszín, életkor. Döntsük el, hogy van-e két olyan személy, akiket ez a nyilvántartás nem különböztet meg.

6.

Határozzuk meg N ($N > 1$) természetes számnál nem nagyobb, legnagyobb négyzetszámot!

7.

Állítsunk elő 50 véletlen számot 1 és 500 között.

- a. Válogassuk ki a 100-nál nagyobb páratlan számokat!
- b. Válogassuk ki a hárommal nem osztható páros számokat!
- c. Válogassuk ki a 200-nál nem nagyobb és 300-nál nem kisebb számok közül azokat, melyek a számjegyeinek összege öttel osztható!
- d. Válogassuk ki a 100 és 300 közé eső, 3-as számjegyet tartalmazó számokat!
- e. Keressük meg a legnagyobb 3-mal nem osztható, de 7-el osztható számot!
- f. Keressük azt a legkisebb számot, mely tartalmaz 3-as számjegyet!

8.

Készítsünk egy totótipp-sort, vegyük figyelembe, hogy az 1-es 50%, x 30%, 2-es 20% valószínűséggel következik be.

9.

Az X N elemű vektor elemein végezzük el a következő transzformációt: mindegyik elemet helyettesítsük saját magának és szomszédainak súlyozott átlagával. A súly legyen az elem indexe a vektorban.

10.

Egy mennyiség pontos mérését 9 azonos célműszeren egyenként 100 alkalommal végezték el. Mindegyik műszer az i . adatát azonos pillanatban mérte. Az eredményeket egy sorozatban helyezték el úgy, hogy az első 100 szám az első műszeren mért adatot jelenti, ezt a második műszeren mért adatok követik, és így tovább. Határozzuk meg az összes mérés átlagát, és számítsuk ki minden egyes mérésnél a 9 műszeren mért értékek átlagát!

11.

Határozzuk meg $A(N,N)$ mátrix felső háromszöge elemeinek összegét!

12.

Adott $A(N,N)$ mátrix. A mátrix fődiagonálisába írjunk olyan értékeket, hogy a mátrix minden sorában az elemek összege 0 legyen!

13.

Egy osztály tesztlapokat töltött ki. Minden kérdésre 4 lehetséges válasz volt, és csak 1 jó. A válaszokat egy mátrixban tároljuk. Készítsünk programot, amely összehasonlítja a válaszokat a helyessel, majd megmondja, melyik tanuló hány jó választ adott. A helyes válaszokat tároljuk egy vektorban.

14.

Egy színház pénztárának nyilvántartása tartalmazza, hogy egy adott előadásra mely helyekre keltek már el jegyek. $B(i,j)$ tömbölem az i . sor j . helyét jelenti a baloldalon, $J(i,j)$ ugyanezt a jobboldalon. A tömbölem értéke 1, ha az adott helyre szóló jegy már elkelt, 0 ha még szabad. Írjunk programot mely keres két szomszédos szabad helyet az adott előadásra.

15.

Az $A(N,2)$ mátrixban egy gráfot ábrázoltunk: a mátrix egy sora a gráf éleit írja le úgy, hogy megadja azoknak a csúcsoknak a sorszámát, amelyekre az él illeszkedik. Határozzuk meg a csúcsok fokszámát!

16.

Írj programot, mely kiírja két számokat tartalmazó tömb közös elemeit!

Programozás tankönyv

VIII. Fejezet

„Logikai ciklusok”

Király Roland

A ciklusok

A programozás során sokszor kerülünk szembe olyan problémával, amikor valamely utasítást, vagy utasítások sorozatát többször kell végrehajtanunk, és sokszor még az sem ismert, hogy az utasításokat hányszor kell ismételni.

Kevés számú ismétlésnél megoldhatjuk a problémát olyan módon is (és ez a *rossz megoldás*), hogy az utasításokat többször egymás után leírjuk, de gondoljunk arra az esetre, mikor egy lépést ezerszer kell végrehajtani. Ilyen és ehhez hasonló helyzetekben célszerű az iteráció használata.

Az ismétlések megvalósításának az eszköze a ciklus-utasítás, melynek több formája létezik. Ebben a fejezetben a logikai ciklusokat nézzük át.

A logikai ciklusoknak két alapvető formája létezik a C# nyelvben.

- Elöl tesztelő while ciklus
- Hátral tesztelő do while ciklus

A while ciklus

A while ciklus általános formája a következőképpen írható le:

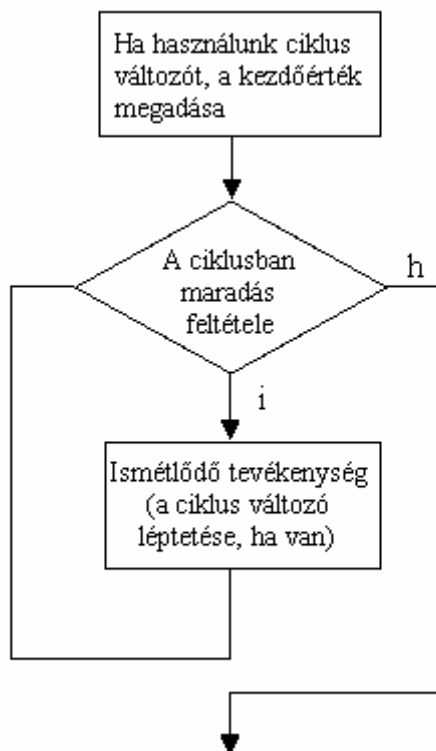
```
while (feltétel)
{
    utasítás1;
    utasítás2;
    ...
    utasításn;
}
```

A fenti ciklus a while kulcsszóval kezdődik, majd a zárójelek közti részben a ciklusban maradás feltételét kell megadni. A ciklus magjában lévő utasításokat, melyek tetszőleges számban fordulhatnak elő, pontosvesszővel választjuk el egymástól. Innen tudja a fordító, hogy hol végződik az aktuális, és hol kezdődik a következő utasítás.

A ciklus magját kapcsos zárójelek { } közé kell tenni, jelezve ezzel a ciklus utasításainak a kezdetét és végét (*egy utasítás esetén a zárójelek elhagyhatóak, de javasolt a használatuk a*

programok jobb áttekinthetősége érdekében). A kapcsos zárójelek közti utasítás blokkot ismétli a rendszer mindaddig, amíg a feltétel igaz, vagyis TRUE értékű.

Folyamatábrával szemlélítve a while ciklus a következő formában írható le:



A folyamatábra a while ciklus logikai modellje. Az első feldolgozást szimbolizáló téglalap csak akkor szükséges, ha a ciklusban használunk ciklus változót, esetleg azt a ciklus bennmaradási feltételeként alkalmazzuk. A ciklus feltétel része tartalmazza a ciklusban maradás feltételét. A második feldolgozás blokk tartalmazza a ciklus utasításait, valamint a ciklus változójának léptetését, természetesen csak abban az esetben, ha van ciklusváltozó. Ha megértjük a folyamatábra és az általános forma működési elvét, akkor azok alapján bármilyen while ciklust el tudunk készíteni.

Nézzünk meg egy konkrét példát a while ciklus használatára! A következő ciklus kiírja a 0-10 közötti egész számokat a konzol képernyőre.

```
int i=0;
while (i<=10)
{
    Console.WriteLine("{0}", i);
    i++;
}
```

(Természetesen a fenti ismétlést megvalósíthattuk volna egyszerűen for ciklussal is.)

A while-t használó megoldásnál ügyeljünk arra, hogy a ciklus változóját léptetni kell, vagyis az értékét minden lefutáskor növeljük meg eggyel, mert a for ciklussal ellentétben itt a rendszer erről nem gondoskodik. *(Emlékezzünk vissza a for ciklusra! Ott a ciklus változó növelése automatikus és nem is célszerű azt külön utasítással növelni).*

A változó léptetésére a következő formulát használtuk:

```
i++;
```

A C típusú nyelvekben, így a C# -ban is, ez a rövidített forma megengedett, ekvivalens az `i=i+1;` és az `i+=1;` utasítással. *(A IV. fejezetben az aritmetikai műveletekről és azok rövidítéséről bővebben olvashattunk.)*

Ügyeljünk arra is, hogy a ciklus változójának mindig adjunk kezdő értéket, ellenkező esetben az ismétlések száma nem lesz megfelelő.

Természetesen a while ciklust leginkább a logikai értékek vizsgálatakor használjuk, s nem számláló ciklusként *(arra ott van a for ☺).*

Az előző fejezetekben már láthattunk néhány példát a logikai ciklusokra, de ott nem részleteztük a működésüket. A következő példa bemutatja, hogyan lehet tízes számrendszerbeli számokat átváltani kettes számrendszerbe.

```
using System;

namespace Atvaltas
{
    class atvalt
    {
        [STAThread]
        static void Main(string[] args)
        {
            int[] t=new int[8];
            int k=0;
            int szam=8;
            int j;
            while (szam!=0)
            {
                t[k]=szam % 2;
                szam=szam / 2;
                k++;
            }

            j=k-1;

            while (j>=0)
            {
                Console.Write("{0}",t[j]);
                j--;
            }
            Console.ReadLine();
        }
    }
}
```

Az első while ciklus feltétele a $(szam!=0)$, a ciklus szempontjából azt jelenti, hogy az ismétléseket addig kell végezni, míg a *szam* változó értéke nullától nagyobb. Nulla érték esetén a ciklus befejezi futását.

Megfigyelhetjük, hogy ebben a ciklusban nem tudjuk előre megmondani a lefutások számát, mivel az mindig a futás közben végrehajtott műveletek eredményétől függ (*annyi lefutás van, ahány osztással elérjük a nulla értéket*). A második ciklusban, amelyik a kettes számrendszerbeli számjegyeket írja a képernyőre, a ciklus változó kezdőértéke a $k-1$. Az értékét nem növeljük, hanem csökkentjük, így a tömbben lévő számjegyeket visszafelé írjuk ki a képernyőre, de csak azokat, melyek az osztás során jöttek létre. A tömb többi értékét nem vizsgáljuk, mivel azok a feladat szempontjából nem értékes számjegyek.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

Az átváltás után a *k* változó értéke 5, mert az átváltandó számjegy a 8, így az első ciklusunkban négy osztás történt, és az utolsó lefutáskor még növeltük a *k* értékét eggyel. Ebből az következik, hogy az utolsó értékes számjegy a negyedik indexen, vagyis a $k-1$ helyen van. (*Próbáljuk meg átírni a fenti programot úgy, hogy ne csak a 8-as számra működjön, hanem a felhasználótól beolvasott tetszőleges értékekre!*)

Vegyük sorra, hogy mikor célszerű logikai, vagy más néven feltételes ciklusokat alkalmazni.

- Amennyiben nem tudjuk meghatározni a ciklus ismétlésinek a számát (*futás közben dől el*).
- Ha a leállás valamilyen logikai értéket szolgáltató kifejezés eredményétől függ.
- Több leállási feltétel együttes használatakor.
- Az előző három pont együttes fennállása esetén.

Több feltétel használatakor az egyes kifejezéseket, logikai értékeket célszerű ÉS, illetve VAGY kapcsolatba hozni egymással. A while ciklus ekkor addig fut, míg a feltétel minden része igaz. Ha bármelyik rész-feltétel, vagy a feltételek mindegyike hamissá válik, akkor az egész kifejezés értéke hamis lesz, és az ismétlés befejeződik.

Mindig gondoljuk végig, hogy a feltételek összekapcsolására && vagy || operátort használunk. Vannak olyan problémák, ahol mindkettőt használnunk kell egy kifejezésen belül. Ügyeljünk a megfelelő zárójelezésre.

Vizsgáljuk meg az alábbi két ciklust. Az első jó eredményt fog szolgáltatni, a futása akkor áll meg, ha az *i* értéke eléri a tömb elemszámát, vagy ha megtaláljuk a nulla elemet. A második ciklus nem biztos, hogy megáll.

```
i=0;
while ( (i<10) && (t[i] != 0) )
{
    Console.WriteLine(" {0} ",t[i]);
    i += 1;
}

i=0;
while ( (i<10) || (t[i] != 0) )
{
    Console.WriteLine(" {0} ",t[i]);
    i += 1;
}
```

Az ÉS kapcsolat, vagyis az && operátor használatakor ügyeljünk a megfelelő zárójelezésre és a precedencia szabályra. A zárójelek elhagyása, vagy rossz helyen történő alkalmazása nehezen észrevehető hibákat eredményez.

Az alábbi két formula közül az első helyes eredményt produkál, a második viszont hibásat, mivel abban szándékosan elrontottuk a zárójelezést.

```
while ( (i<10) && (a!=2))
while ( i<(10 && a)!=2)
```

A rosszul felírt zárójelek miatt a fordító először a (10 && a) részt értékeli ki, s ez hibához vezet.

A break

A C# nyelvben a while ciklusból ki lehet ugrani a break utasítással, amit általában egy feltétel teljesüléséhez kötünk. Az alábbi ciklus feltétel részébe a true értéket írtuk, ami azt jelenti, hogy a feltétel mindig igaz, vagyis a ciklus soha nem áll meg. A feltételes elágazásban a d=5 feltétel teljesülése esetén a break utasítás állítja meg a while ciklust.

```
d = 0;
While (true)
{
    if (d=5)
    {
        break;
    }
    d += 1;
}
```


A continue

A continue utasítással visszaadjuk a vezérlést a while ciklus feltételére. Ekkor a continue utáni rész teljesen kimarad az aktuális lefutáskor. Ritkán használjuk, de mindenképpen meg kell ismerkednünk vele.

```
while (i < tomb.Count)
{
    int a = Int32.Parse( Console.ReadLine() );
    if (a <= 0) continue;
    i++;
    tomb[i] = a;
}
```

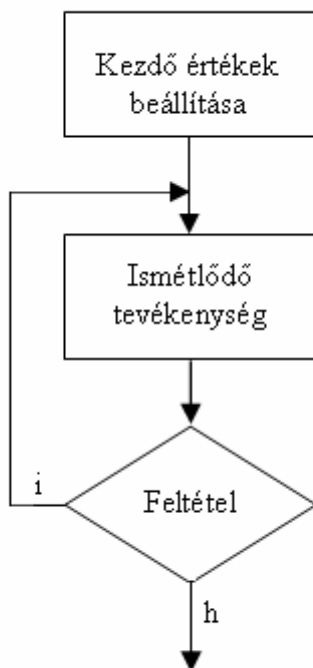
A programrészlet egy tömb elemeinek billentyűzetről való feltöltését valósítja meg úgy, hogy negatív számokat nem fogad el. Amennyiben negatív szám érkezik az inputról, a `continue` visszadobja a vezérlést a ciklus elejére.

A do while ciklus

A ciklus utasítás következő típusa a hátul tesztelő változat. Ennek érdekessége az, hogy a ciklus magja egyszer mindenképpen lefut, mivel a feltétel vizsgálata a ciklus végén van.

```
do
{
    Utasítások;
} while (feltétel)
```

A do while ciklust is érdemes felírni folyamatábrával, hogy jobban megértsük a működését.



A folyamatábrán jól látszik, hogy a ciklusban maradás feltétele a ciklus utasításai után szerepel, így egyszer mindenképpen végrehajtódnak a ciklus magjában definiált utasítások. Amíg a feltétel igaz, a ciklus fut, ha hamissá válik, megáll.

A következő példa egy lehetséges megvalósítását mutatja meg a hátul tesztelő do while ciklusnak.

```
char c;  
Console.WriteLine("Kilépés : v");  
do  
{  
    c=Console.Read();  
    Console.Write("a beolvasott változó : {0}\n",c);  
} while (c!='v');
```

Az ismétlés akkor áll meg, ha a c karakter típusú változóba a "v" értéket adja meg a felhasználó, de egyszer mindenképpen lefut.

Ahogy a fenti példa is mutatja a do while ciklust jól használható ellenőrzött, vagy végjelig történő beolvasásra.

A foreach

Van egy speciális ismétlés, a foreach, mely képes a gyűjtemények, vagy a tömbök végigjárására. Ez azt jelenti, hogy nem szükséges ciklusutasítást szerveznünk, és nincs szükség ciklusváltozóra sem, mellyel a tömböt, vagy gyűjteményt indexelnénk. Igaz, hogy ez

nem logikai ciklus, és az összetett adat szerkezetekkel még nem foglalkoztunk, de a teljesség kedvéért vizsgáljuk meg ezt a típust is! A foreach általános formája a következő:

```
foreach (adattípus változó in tömbnév)
{
    Utasítások;
}
```

Az alábbi program részlet bemutatja, hogyan kell használni a foreach utasítást:

```
Char[] t=new char[] { 'f','o','r','e','a','c','h' };

Foreach (char c in t)
{
    Console.WriteLine("{0}",c);
}
```

A példában a foreach sorra veszi a t tömb elemeit. Az aktuális tömb elem a c változóból olvasható ki. Az ismételés az utolsó elem kiírása után áll le, vagyis mikor végigjárta a tömböt. Természetesen a fenti program részletet for, vagy while ciklussal is megvalósíthattuk volna, de a foreach-t pontosan arra találták ki, hogy ne kelljen figyelni a tömb elemszámát és a tömb elemeit se kelljen a programozónak indexelnie. Ráadásul a foreach használatakor nem fordulhat elő, hogy túl, vagy alul indexeljük a gyűjteményt.

A következő példában a while ciklussal felírt változatot láthatjuk. Ez a program-részlet több utasításból áll, szükség van egy ciklusváltozóra, melynek az értékét növelnünk kell, és a kezdőértékről is gondoskodnunk kell.

```
char[] t=new char[] { 'f','o','r','e','a','c','h' };
i = 0;
while (i< t.length())
{
    Console.WriteLine("{0}",t[i]);
    i += 1;
}
```

A feltétel vezérelt ciklusokat a programozás számos területén használjuk. Fontosak a szöveges információk kezelésénél, a fájlkezelésnél, s minden olyan probléma megoldásánál, amikor az iterációt for ciklussal nem lehet, vagy nem célszerű megvalósítani.

Programozási feladatok

1. Írjon programot, mely kiírja a képernyőre az első tíz egész számot visszafelé!
2. Alakítsa át az előző programot úgy, hogy az csak a páros számokat írja a képernyőre!
3. Írassa ki a képernyőre az első n egész szám összegét!
4. Írassa ki a képernyőre az első n egész négyzetszámot!
5. Írjon programot, mely beolvassa a billentyűzetről egy intervallum kezdő és végétértékét, majd kiírja a képernyőre az intervallumba eső egész számok közül azokat, melyek 3-mal és 5-tel is oszthatóak!
6. Állapítsa meg a felhasználótól beolvasott számról, hogy az prímszám-e!
(*Prímszámok azok a számok, melyek csak önmagukkal és eggyel oszthatóak.*)
7. Keresse meg az első n darab prímszámot!
8. Készítsen programot, mely beolvassa a billentyűzetről az egész számokat egy meghatározott végjelig. Az éppen beolvasott számnak írja ki a négyzetét.
9. A felhasználótól beolvasott számot a program váltsa át kettes számrendszerbe!
10. Programja olvasson be a billentyűzetről egész számokat egy meghatározott végjelig (legyen a végjel -999999), majd a végjel beolvasása után írja ki a legnagyobbat és a legkisebbet a kapott értékek közül.
11. Írjon programot, mely kiírja az első n db páros szám összegét a képernyőre!
12. Készítsen programot, mely beolvas n db számot a billentyűzetről, majd meghatározza a számok átlagát.

Programozás tankönyv

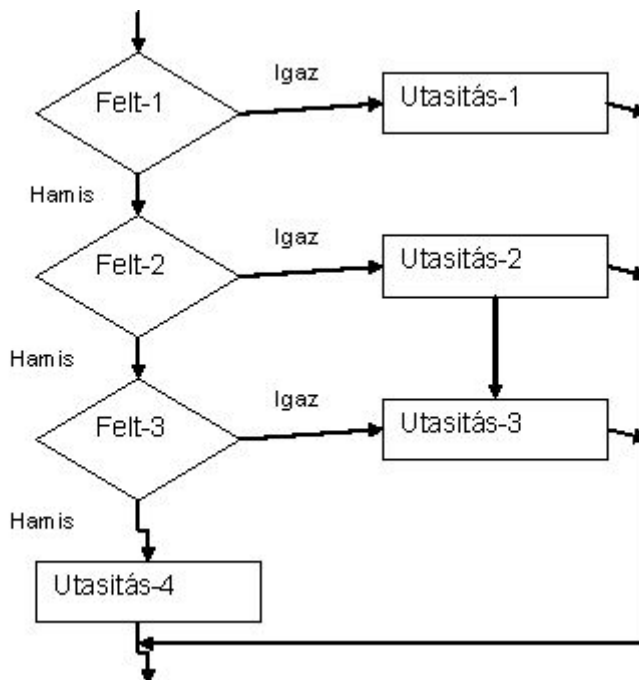
IX. Fejezet

„Szelekció emelt szint!”

Radványi Tibor

Szelekció haladóknak

A szelekciónak több ága is lehet. Ilyenkor a feltételek közül legfeljebb egy teljesülhet, vagyis ha az egyik feltétel teljesül, akkor a többit már nem kell vizsgálni sem. A többágú szelekció folyamatábrája:



Egymásba ágyazott **if** utasítások sorozatával tudjuk megvalósítani a fenti szerkezetet. Ha az első feltétel nem teljesül, akkor a második feltétel kiértékelése következik. És így tovább. A végrehajtás legfeljebb egy igaz ágon fog futni. Az ábra szerint az Utasítás-4 akkor kerül végrehajtásra, ha egyik feltétel sem teljesült. Az utolsó **else** ág elhagyható. Hogyan valósítjuk ezt meg C#-ban? Ehhez nézzünk egy egyszerű, klasszikus példát.

Feladat:

Olvassunk be egy nemnegatív egész számot, valakinek az életkorát, és kortól függően írjuk ki a megfelelő szöveget:

0 - 13 évig : gyerek
14 – 17 évig : fiatalkorú
18 – 23 évig : ifjú
24 – 59 évig : felnőtt
60 - : idős

Megoldás:

```
static void Main(string[] args)
{
    int kor;
    kor = Int32.Parse(Console.ReadLine());
    if (kor < 14) Console.WriteLine("Gyerek");
    else
        if (kor < 18) Console.WriteLine("Fiataalkorú");
    else
        if (kor < 24) Console.WriteLine("Ifjú");
    else
        if (kor < 60) Console.WriteLine("Felnőtt");
    else Console.WriteLine("Idős");
    Console.ReadLine();
}
```

A többágú szelekció másik megoldása a **switch** utasítás.

```
switch (kifejezés)
{
    case konstans1 kifejezés:
        utasítás1
        ugrás
    case konstans2 kifejezés:
        utasítás2
        ugrás
    case konstansn kifejezés:
        utasításn
        ugrás
    [default:
        utasítás
        ugrás]
}
```

Kifejezés: egész típusú, vagy string kifejezés.

Ugrás: Kilép a vezérlés a **case** szerkezetből. Ha elhagyjuk, akkor a belépési ponttól számítva a további **case**-eket is feldolgozza.

Default: Ha egyetlen case feltétel sem teljesül, akkor a default ág kerül végrehajtásra.

Egy egyszerű felhasználás:

```
static void Main(string[] args)
{
    int k;
    Console.Write("Kérem válasszon (1/2/3)");
    k = Int32.Parse(Console.ReadLine());
    switch (k)
    {
        case 1:
            Console.WriteLine("1-et választotta");
            break;
        case 2:
            Console.WriteLine("2-őt választotta");
            break;
        case 3:
            Console.WriteLine("3-at választotta");
            break;
        default:
            Console.WriteLine("nem választott megfelelő számot.");
            break;
    }
    Console.ReadLine();
}
```


Ugyanez feladat megoldható konvertálás nélkül, stringek felhasználásával is:

```
static void Main(string[] args)
{
    string k;
    Console.Write("Kérem válasszon (1/2/3)");
    k = Console.ReadLine();
    switch (k)
    {
        case "1":
            Console.WriteLine("1-et választotta");
            break;
        case "2":
            Console.WriteLine("2-őt választotta");
            break;
        case "3":
            Console.WriteLine("3-at választotta");
            break;
        default:
            Console.WriteLine("nem választott megfelelő számot.");
            break;
    }
    Console.ReadLine();
}
```

Feladat:

Olvassunk be egy hónap és egy nap sorszámát! Írjuk ki ha a beolvasott számok nem jó intervallumba esnek. A február legyen 28 napos.

Megoldás:

```
static void Main(string[] args)
{
    int ho, nap;
    Console.Write("Kérem a hónapot:");
    ho = int.Parse(Console.ReadLine());
    Console.Write("Kérem a napot:");
    nap = int.Parse(Console.ReadLine());
    if ((ho == 1 || ho == 3 || ho == 5 || ho == 7 || ho == 8 || ho == 10
|| ho == 12)
        && (nap > 31 || nap < 1))
        Console.WriteLine("A nap legfeljebb 31 lehet");
    else
        if ((ho == 4 || ho == 6 || ho == 9 || ho == 11) && (nap > 30 || nap <
1))
            Console.WriteLine("A nap legfeljebb 30 lehet");
        else
            if ((ho == 2) && (nap > 28 || nap < 1))
                Console.WriteLine("A nap legfeljebb 28 lehet");
            else
                if (ho < 1 || ho > 12) Console.WriteLine("A hónap csak 1 és 12 között
lehet.");
            else
                Console.WriteLine("Rendben");
    Console.ReadLine();
}
```

Nézzük a lehetséges futási eredményeket. Első, amikor minden rendben, második, amikor a hónap megfelelő, de a nap nem helyes, és a harmadik lehetőség amikor a hónap száma nem jó.

```
"H:\C#\szelhal\bin\Debug\szelhal.exe"  
Kérem a hónapot:4  
Kérem a napot:23  
Rendben  
  
I
```

Első eset, amikor minden rendben

```
"H:\C#\szelhal\bin\Debug\szelhal.exe"  
Kérem a hónapot:7  
Kérem a napot:45  
A nap legfeljebb 31 lehet
```

Második eset, amikor a nap 45, ez nem lehet!

```
"H:\C#\szelhal\bin\Debug\szelhal.exe"  
Kérem a hónapot:45  
Kérem a napot:12  
A hónap csak 1 és 12 között lehet.
```

Harmadik eset, amikor a hónap nem lehet 45.

További megoldásra ajánlott feladatok

1.

Olvassa be Zsófi, Kati, Juli születési évét. Írja ki a neveket udvariassági sorrendben. Először az idősebbeket.

2.

Olvasson be egy egész óra értéket. Köszönjön a program a napszaknak megfelelően. 4 – 9 Jó reggelt. 10 – 17 Jó napot. 18 – 21 Jó estét. 22 – 3 Jó éjszakát.

3.

Adjon meghárom diszjunkt intervallumot. Eztán olvasson be egy számot, és írja ki hogy melyik intervallumba esik.

4.

Olvassunk be egy számsorozatot, majd számoljuk meg hogy ezekből mennyi a negatív, nulla, pozitív. A negatívokat és a pozitívokat rakjuk át egy másik sorozatba.

5.

Határozzuk meg az $A(N)$ vektor elemeinek összegét úgy hogy a páratlan értékű elemek a (-1) szorosúkkal szerepeljenek az összegben.

6.

Egy dobókockával 100-szor dobunk. Adjuk meg hogy ezek közül hány darab volt páros, illetve a párosak közül mennyit követett közvetlenül páratlan szám.

7.

Adott egy $A(N)$ vektor. Számoljuk meg, hány pozitív érték van a vektor azon elemei között, amelyek indexe prímszám.

8.

Adott N ember neve, személyi száma. Válogassuk ki a halak csillagképben született férfiak neveit (február 21 – március 20).

9.

Adott egy természetes számokat tartalmazó vektor. Elemei közül válogassuk ki azokat, melyek relatív prímek a 15-höz.

Programozás tankönyv

X. Fejezet

Stringek kezelése

Király Roland

A string típusú változó

A C#-ban, mint minden magas szintű nyelvben létezik a string típus. Az OOP nyelvekben a string-ekre osztályként tekinthetünk, mivel vannak metódusaik, tulajdonságaik, és minden olyan paraméterük, mely a többi osztálynak, de dolgozni úgy tudunk velük, mint az összes többi programozási nyelv string típusával. Hasonlóan a többi változóhoz, a string-eket is deklarálni kell.

```
string s;  
string w;
```

Adhatunk nekik értéket:

```
string a = "hello";  
string b = "h";  
b += "ello";  
s="A2347";  
s="1234567890";
```

Kiolvashatjuk, vagy éppen a képernyőre írhatjuk a bennük tárolt adatokat:

```
string w=s;  
Console.WriteLine(s);
```

Figyeljük meg, hogy a fent szereplő értékadásban (s="1234567890") számok sorozatát adtuk értékül az s string típusnak. Az s változóban szereplő értékkel ebben az esetben nem számolhatunk, mert az nem numerikus értékként tárolódik, hanem ún.: karakterlánc-literálként, így nem értelmezhetőek rá a numerikus típusoknál használt operátorok.

Igaz, hogy a + művelet értelmezhető a string-ekre is, de működése eltér a számoknál megszokottól. *(A fejezet későbbi témáinál kitérünk a + operátor működésére.)*

Fontos megemlíteni, hogy a C alapú nyelvekben a \ (*backslash*) karakter ún.: escape szekvencia. Az s="C:\hello" karakter sorozat hibás eredményt szolgáltat, ha elérési útként szeretnénk felhasználni, mivel a \h -nak nincs jelentése, viszont a rendszer megpróbálja értelmezni. Ilyen esetekben vagy s="C:\\hello" vagy s=@"C:\hello" formulát használjuk! Az első megoldásban az első \ jel elnyomja a második \ jel hatását. A második megoldásnál a @ jel gondoskodik arról, hogy az utána írt string a "C:\hello" formában megmaradjon az értékadásnál, vagy a kiíró utasításokban.

Ha egy string-nek nem adunk értéket, akkor induláskor 'null' értéke van.

Az üres string-et vagy `s=""`; formában, vagy `s=String.Empty` formában jelezhetjük.

A relációk használata kisebb megkötésekkel a string-eknél is megengedett. Az összehasonlításokat elvégezhetjük logikai kifejezésekben, elágazások és logikai ciklusok feltétel részében. A string-ek egyes elemeit tudjuk `<`, vagy `>` relációba hozni egymással `s[1]<s[2]`, a teljes string-re viszont csak a `!=` és a `==` relációkat alkalmazhatjuk. Ha mégis szeretnénk összehasonlítani két string-et, a következő megoldást használhatjuk:

```
Str1.CompareTo(str2)
```

Ezzel a metódussal össze tudjuk hasonlítani két string értékét. A következő példa ezt be is mutatja:

```
int cmp;
string str1="abc";
string str2="cde";
cmp="szoveg".CompareTo("szoveg");

if ( cmp == 0) {Console.WriteLine("str1 = str2");}
if ( cmp > 0) {Console.WriteLine("str1 < str2");}
if ( cmp < 0) {Console.WriteLine("str1 > str2");}
```

A két string egyenősége esetén a visszatérési érték 0. Ha a kapott érték kisebb, mint 0, akkor az str1 ABC sorrendben előbb van az str2 -nél, ha nagyobb, mint 0, akkor str2 van előbb. A példában a második if feltétele teljesül, mivel az abc karaktorsor előbb szerepel a sorrendben, mint a cde.

A string-eket is használhatjuk konstansként. Ugyanúgy definiáljuk őket, mint a többi típus állandóit.

```
const string="string konstans";
```

Ahogy azt már említettük, C# nyelvben a string-ek is rendelkeznek metódusokkal, mint bármely más osztály. Ez némiképp megkönnyíti a használatukat, mert nem a programozónak kell gondoskodnia pl.: a szóközök eltávolításáról, a kis és nagybetűs átalakításról, a keresés metódusainak definiálásáról, és nem kell külső rutinokhoz folyamodnia, mint a Pascal nyelv `pos()`, vagy `copy()` eljárásainál.

Az Objektum Orientált megközelítés előnyei jól nyomon követhetők a string típus esetében. Az egységbezárás lehetővé teszi, hogy rendelkezzen saját metódusokkal. A következő példaprogram bemutatja a string-ek használatának néhány lehetőségét.

```

using System;

namespace String_01
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            string s;
            s="alma";
            Console.WriteLine("{0} - A string elejéről levágtuk az
a karaktereket.",s.TrimStart('a'));
            Console.WriteLine("{0} - A string végéről levágtuk az
a karaktereket.",s.TrimEnd('a'));
            Console.WriteLine("{0} - Nagybetűssé alakítottuk a
sztringet.",s.ToUpper());
            Console.WriteLine("{0} - Kisbetűssé alakítottuk a
sztringet.",s.ToLower());
            Console.ReadLine();
        }
    }
}

```

A program futása a következő:

```

lma - A string elejéről levágtuk az a karaktereket.
alm - A string végéről levágtuk az a karaktereket.
ALMA - Nagybetűssé alakítottuk a sztringet.
alma - Kisbetűssé alakítottuk a sztringet.

```

Az első WriteLine- ban az s string TrimStart() metódusát használtuk, mely eltávolítja a szöveg elejéről a paraméter listájában megadott karakter összes előfordulását. (Az *aaalma* string- ből az *lma* stringet állítja elő.)

A TrimStart(), TrimEnd() és a Trim() metódusuk paraméterek nélkül a string elejéről és végéről levágják a szóközöket.

A következő metódus a TrimEnd(). Ez a string végéről vágja el a megadott karaktereket.

A ToUpper() és a ToLower() metódusokkal a kis- és nagybetűs átalakításról gondoskodhatunk.

Vegyük észre, hogy az s="alma" utasításnál a string-et macskakörmök ("...") közé tettük, míg a TrimStart() és TrimEnd() metódusok paramétereit aposztrófok ('...') zárják közre, mivel az első esetben string-et, a másodikban karaktert definiáltunk. Gyakori hiba, hogy a kettőt

felcseréljük. Ilyenkor a fordító hibát jelez, mivel a karakterek nem kezelhetők stringként és fordítva.

A string-ből kimásolhatunk részeket a Substring() metódussal, melynek paraméter listájában meg kell adni a kezdő indexet, ahonnan kezdjük a másolást, és a másolandó karakterek számát. A következő példa megmutatja a Substring() használatát.

```
Console.WriteLine("Az eredeti szó= alma, a kiemelt részlet ={0}  
",s.Substring(1,2));
```

Ha a Substring() csak egy paramétert kap és a paraméter értéke belül van a string index tartományán, akkor az adott indextől a string végéig az összes karaktert kimásoljuk. A paraméterezéskor ügyeljünk arra, hogy mindig csak akkora indexet adjunk meg, ahány eleme van a string-nek. Arra is figyelni kell, hogy a string első eleme a 0. indexű elem, az utolsó pedig a karakeszám-1.

A stringek-re a + műveletet is értelmezhető, de ilyenkor összefűzésről beszélünk. Amennyiben a fenti példában szereplő s stringek közé a + operátort írjuk, akkor:

```
s="alma"  
s=s+" a fa alatt";
```

az s tartalma megváltozik, kiegészül a + jel után írt string- konstanssal. Az eredmény: "alma a fa alatt". Amennyiben a fenti értékadásban szereplő változót és konstanst felcseréljük, s="a fa alatt"+s, eredményül az "a fa alattalma" karaktersort kapjuk. Ez azt jelenti, hogy az adott string-hez jobbról és balról is hozzáfűzhetünk más string-eket.

A keresés művelete is szerepel a metódusok között. Az IndexOf() függvény -1 értéket ad vissza, ha a paraméter listájában megadott karakter, vagy karaktersorozat nem szerepel a string- ben, 0, vagy pozitív értéket, ha szerepel. Mivel a stringek 0- tól vannak indexelve, találat esetén 0-tól a string hosszáig kaphatunk visszatérési értéket, ami a találat helyét, vagyis a kezdő indexét jelenti a keresett karakternek, vagy karakterláncnak.

A következő példa bemutatja, hogyan használhatjuk az IndexOf() metódust. A program beolvas egy tetszőleges mondatot és az abban keresendő szót. Amennyiben a szó szerepel a mondatban, a képernyőre írja a mondatot, a szót és azt, hogy a szó szerepel-e vagy sem a mondatban. A C# nyelv kis-nagybetű érzékeny (case-sensitive), ezért a vizsgálat előtt a beolvasott adatokat nagybetűssé alakítjuk, hogy a tényleges tartalom alapján döntsünk.

```

namespace string_02
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            string s;
            string w;
            Console.WriteLine("Gépeljen be egy tetszőleges mondatot
: ");

            s=Console.ReadLine();
            Console.WriteLine("Adja meg a keresendő szót : ");
            w=Console.ReadLine();
            s=s.ToUpper();
            w=w.ToUpper();
            if (s.IndexOf(w)>-1)
            {
                Console.WriteLine("A(z) {0} szó szerepel a(z) {1}
mondatban",w,s);
            }
            else
            {
                Console.WriteLine("A(z) {0} szó nem a(z) {1}
mondatban");
            }
            Console.ReadLine();
        }
    }
}

```

A program futásának eredménye:

```

Gépeljen be egy tetszőleges mondatot :
Alma a fa alatt.
Adja meg a keresendő szót :
alma
A(z) ALMA szó szerepel a(z) ALMA A FA ALATT. mondatban
_

```

A programunk megfelelően működik, de az Output-ja nem tökéletes, mivel az eredeti mondatot nagybetűsre alakítja, majd ezt a formáját írja vissza a képernyőre, a felhasználó által begépelte kis- vagy nagy betűket vegyesen tartalmazó forma helyett.

A problémára több megoldás is létezik. Az egyik, hogy az eredeti változókat megőrizzük, azok tartalmát segédváltozókból tároljuk, amiket átalakíthatunk nagybetűssé, műveleteket végezhetünk velük, majd a program végén az eredeti változók tartalmát írjuk ki a képernyőre. *(Ezen megoldás elkészítése a kedves Olvasó feladata. Nincs más dolga, csak két újabb változót bevezetnie, és ezekben átmásolni az eredeti változók tartalmát...)*

A string-ek mint vektorok

A stringek kezelhetők vektorként is. Hivatkozhatunk rájuk elemenként, mint a karaktereket tartalmazó tömbökre.

```
string s="123456789";
for (i=0;i<s.Length;i++)
{
    Console.WriteLine("{0}",s[i]);
}
```

A fenti példában a string Length tulajdonságával hivatkoztunk a hosszára, majd a ciklus magjában mindig az aktuális elemet írtuk ki a képernyőre. A hossz tulajdonság a string-ben szereplő karakterek számát adja vissza. A 0. az első, a length-1. az utolsó elem indexe. Amennyiben a for ciklusban egyenlőség is szerepelne ($i \leq s.Length$) a ciklus végén hibaüzenetet kapnánk, mivel az s-nek elfogytak az elemei. Ezt túlindexelésnek nevezzük. *(Gyakori hiba kezdő programozóknál – ne kövessük el!)*

Az s string i. eleme, vagyis az aktuális indexelt elem karakter típus, mivel karakterek sorozatából épül fel, vagyis az s[i], a string egy eleme karakter. Ezt a következő példán keresztül be is láthatjuk.

Az s=s[1]; értékadó utasítás eredménye a következő hibaüzenet jelenik meg:

```
Cannot implicitly convert type 'char' to 'string',
```

Vagyis a két típus nem kompatibilis. Az értékadás akkor válik helyessé, ha az s[1] elemet konvertáljuk string-gé.

```
s=s[1].ToString();
```

Ennél az értékadásnál már nem kapunk hibaüzenetet. Az értékadás eredménye az s 1-es indexű eleme.

Annak ellenére, hogy a string-ek elemenként is olvashatóak, az ilyen típusú műveletek kellő körültekintést igényelnek. Az értékadásoknál inkább használjuk a típus megfelelő beszűrő rutinját.

A string-ekbe ún.: rész string-eket szűrhetünk be az Insert() metódus segítségével. A paraméter listának tartalmaznia kell az indexet, ahova szeretnénk beszűrni, és a karaktert, vagy a karakterláncot. A következő példa bemutatja az Insert() metódus használatát.

```
s="123456"  
s=s.Insert(2,"ABCD");
```

A fenti programrészlet eredményeként az s változó tartalma az alábbi érték lesz:

```
"12ABCD3456"
```

A következő példa megmutatja, hogyan tudunk tetszőleges szöveget a felhasználó által megadott kulcs segítségével titkosítani. A titkosításhoz a XOR (*kizáró-vagy*) operátort használjuk fel. *(Amennyiben a titkosított szöveget egy változóban tároljuk, a XOR újbóli alkalmazásával a string eredeti állapotát kapjuk vissza.)* A C# nyelvben a XOR műveletet a ^ karakter jelenti, működése megegyezik a matematikai logikából ismert kizáró-vagy operátorral. *(A titkosításhoz és a visszafejtéshez érdemes függvényt készíteni.)*

```
using System;  
  
namespace ConsoleApplication4  
{  
    class Class1  
    {  
        [STAThread]  
        static void Main(string[] args)  
        {  
            string s;  
            int key;  
            int i;  
  
            Console.WriteLine("Gépelje ide a  
titkosítandó szöveget : ");  
            s=Console.ReadLine();  
  
            Console.WriteLine("Adja meg a titkosítás kulcsát (numerikus adat): ");  
            key=Convert.ToInt32(Console.ReadLine());  
  
            for (i=0;i<s.Length;i++)  
            {  
                Console.Write((char) (s[i]^key));  
            }  
            Console.ReadLine();  
        }  
    }  
}
```

A program kimenete a következő képen látható:

```
Gépelje ide a titkosítandó szöveget :  
alma a fa alatt  
Adja meg a titkosítás kulcsát (numerikus adat):  
1  
'm1'!'!g'!'m'uu
```

A for ciklus kiíró utasításában a beolvasott szöveget alakítjuk át úgy, hogy minden karakterét kizáró-vagy kapcsolatba hozzuk a felhasználótól kapott kulccsal. Ez által az eredeti szövegből egy matematikai úton titkosított szöveget kapunk, melyet az eredeti kulcs ismerete nélkül nem lehet visszaállítani. *(Természetesen a titkosított szöveg visszafejtése lehetséges, számos módszer létezik rá, pl.: karakter statisztikák, kódfejtő algoritmusok, matematikus ismerősök ☺.)*

A titkosítás eredményét az egyszerűség kedvéért nem tároltuk, csak kiírtuk a képernyőre. A Kiíró utasításban ún.: típus-kényszerítést alkalmaztunk, mivel a XOR eredményeként kapott érték nem karakter, hanem numerikus adat.

```
Console.Write((char)(s[i]^key));
```

Ezzel a megoldással értük el, hogy a képernyőn nem az adott karakter kódja, hanem maga a karakter látható.

A string típus szinte minden komolyabb programban előfordul, különösképpen igaz ez a fájlkezelő programokra. Természetesen a string-ek használatának még rengeteg finomsága létezik. Használhatjuk őket számrendszerek közti átváltások eredményének tárolására, olyan nagy számok összeadására, szorzására, melyekhez nem találunk megfelelő "méretű" változókat.

Sajnos nem minden programozási nyelv tartalmazza a string típust, és nagyon kevés olyan nyelv van, melyben osztályként használhatjuk, de ahol megtaláljuk *(ilyen a C# nyelv)*, éljünk vele, mert megkönnyíti és meggyorsítja a programozást.

Programozási feladatok

1. Írja a képernyőre a következő string konstansokat: "c:\alma", "c:\\alma"!
2. Írjon programot, mely megszámolja, hogy az inputként érkező mondatban hány darab "a" betű van!
3. Az előző programot alakítsa át úgy, hogy a számlálандó betűt is a felhasználó adja meg!
4. Készítsen karakter statisztikát a felhasználótól beolvasott szöveg karaktereiről! A statisztika tartalmazza az adott karaktert, és azt, hogy hány darab van belőle.
5. Az input szövegből távolítsa el a szóközöket!
6. Olvasson be egy mondatot és egy szót! Mondja meg, hogy a szó szerepel-e a mondatban!
7. A beolvasott mondatról döntse el, hogy az visszafelé is ugyanazt jelenti-e! (Az "Indul a görög aludni", vagy a "Géza kék az ég" visszafelé olvasva is ugyanazt jelenti.) Ügyeljen a mondatvégi írásjelekre, mivel azok a mondat elején nem szerepelnek.
8. Olvasson be egy számot egy string típusú változóba, majd alakítsa szám típusúvá!
9. Olvasson be egy egyszerű, a négy alap-műveletet tartalmazó kifejezést (1+2,1*2,4-3,6/2, stb.) a billentyűzetről egy string típusú változóba, majd a kifejezés értékét írja ki a képernyőre!
10. A beolvasott mondat kisbetűit alakítsa nagybetűsre, a nagybetűs karaktereket pedig kisbetűsre!
11. Két mondatról döntse el, hogy azonosak-e! Ha a kis és nagybetűk különböznek, a programnak akkor is megoldást kell adnia!
12. Az inputként beolvasott szövegben cserélje ki az összes szóközt a # karakterre, majd az így kapott szöveget írja ki a képernyőre!
13. Állapítsa meg, hogy az input szövegben szerepelnek-e számok!

Eljárások alapfokon

Oszd meg és szedd szét...

Hernyák Zoltán

Hosszabb programok írása esetén amennyiben a teljes kódot a Main tartalmazza, a program áttekinthetetlen lesz.

Mindenképpen javasolt a kód részekre tördelése. Ennek során olyan részeket különítünk el, amelyek önmagában értelmes részfeladatokat látnak el. E részfeladatoknak jól hangzó nevet találunk ki (pl. bekérés, feltöltés, kiírás, maximális elem megkeresése, listázás, kigyűjtés, stb.), ezeket a program külön részén, a Main fv-től elkülönítve írjuk meg.

Az ilyen, önálló feladattal és névvel ellátott, elkülönített programrészletet **eljárásnak** nevezzük.

Az eljárásnak

- a visszatérési érték típusa kötelezően void,
- van neve (azonosító),
- lehetnek paraméterei,
- van törzse.

Pl:

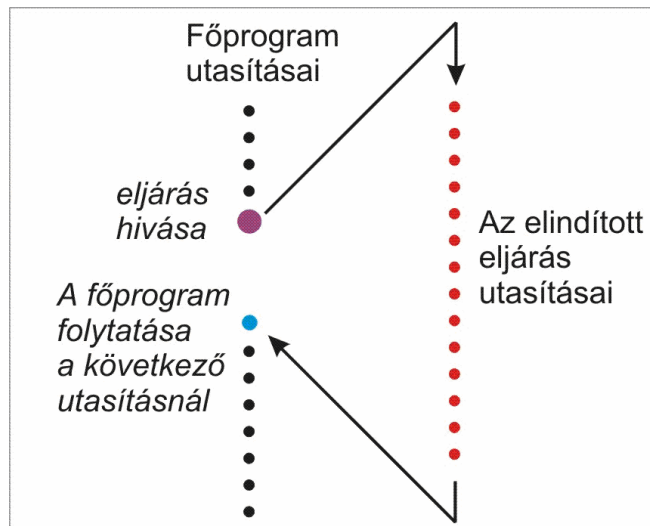
```
static void Információ()  
{  
    Console.WriteLine("*****");  
    Console.WriteLine("** Példa program **");  
    Console.WriteLine("Programozó: Kiss Aladár Béla");  
    Console.WriteLine("Készült: 2004.11.23");  
}
```

A fenti példában az eljárás neve 'Információ', és saját utasításblokkja van (a '{' és '}' közötti rész). Az eljárás (egyelőre) 'static void' legyen, és a zárójelpár megadása kötelező abban az esetben is, amennyiben nincsenek paraméterek.

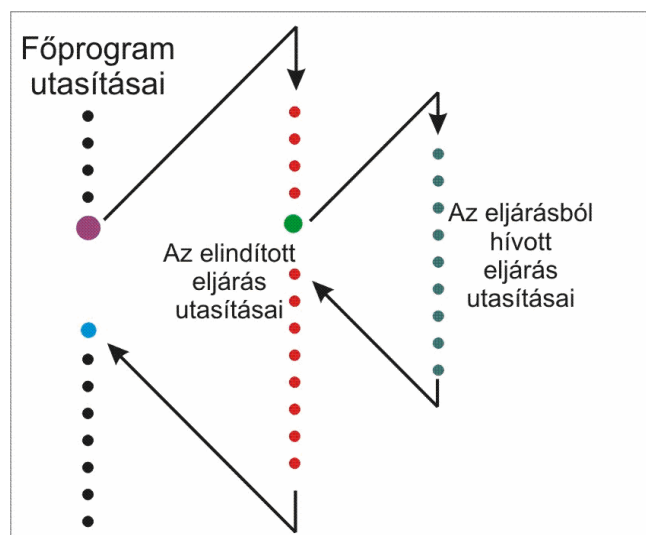
Amennyiben a program adott pontján szeretnénk végre hajtani az eljárásban foglalt utasítássorozatot, úgy egyszerűen hivatkoznunk kell rá a nevével (eljárás indítása, eljárás meghívása):

```
static void Main(string[] args)  
{  
    Információ();  
}
```

Ekkor az adott ponton az eljárás végrehajtásra kerül, majd a végrehajtás visszatér a hívást követő utasításra, és folytatódik tovább a program futása:



Természetesen van rá lehetőség, hogy ne csak a Main()-ből hívjunk eljárásokat, hanem az eljárásokból is van lehetőség további eljárásokat hívni:



C#-ban az eljárások neve azonosító, ezért érvényes az azonosító névképzési szabálya: betűvel vagy aláhúzással kezdődik, betűvel, számjeggyel, aláhúzás karakterekkel folytatódhat (de nem tartalmazhat szóközt)! A C#-ban szabad ékezetes karaktereket is használni az eljárás-nevekben, de nem javasolt – karakterkészlet és kódlap függő forráskódot eredményez, e miatt másik gépen másik programozó által nehezen olvashatóvá válik a forráskód.

Hová írjuk az eljárásainkat?

Az Objektum Orientált programozás elvei szerint a programok építőkövei az osztályok. Az osztály (class) mintegy csoportba foglalja az egy témakörbe, feladatkörbe tartozó eljárásokat, függvényeket, változókat. E miatt a program felépítése a következő lehet:

```

using System;

namespace Tetszoleges_Nev
{
    class PeldaProgram
    {
        Más eljárások ...
        static void Main(string[] args)
        {
            ...
        }
        További eljárások ...
    }
}

```

Az egyéb eljárásainkat ugyanabba a csoportba (osztályba) kell megírni, mint ahol a Main() fv található. A sorrendjük azonban lényegtelen, mindegy, hogy a saját eljárásainkat a Main() fv előtt, vagy után írjuk le, illetve írhatjuk őket vegyes sorrendben is. Az eljárások egymás közötti sorrendje sem lényeges. Ami fontos, hogy közös osztályba (class) kerüljenek.

A programok eljárásokra tagolása persze újabb problémákat szül. Az eljárások *'nem látják'* egymás változóit. Ennek oka a változók hatáskörének problémakörében keresendő.

Minden változónévhez tartozik egy hatáskör. A hatáskör megadja, hogy az adott változót a program **szövegében** mely részekben lehet 'látni' (lehet rá hivatkozni). A szabály az, hogy a változók csak az őket tartalmazó blokk-on belül 'láthatók', vagyis **a változók hatásköre az őket tartalmazó blokkra terjed ki.**

```

static void Main(string[] args)
{
    int a=0;
    Feltoltes();
    Console.WriteLine("{0}",a);
}

static void Feltoltes()
{
    a=10;
}

```

A fenti kód hibás, mivel az 'a' változó hatásköre a 'Main' függvény belseje, az 'a' változóra a 'Feltoltes()' eljárásban nem hivatkozhatunk. Ha mégis megpróbáljuk, akkor a C# fordító a *The name 'a' does not exist ...* kezdetű fordítási hibaüzenettel jelzi ezt nekünk.

Ezt (hibásan) az alábbi módon reagálhatjuk le:

```
static void Main(string[] args)
{
    int a=0;
    Feltoltes();
    Console.WriteLine("{0}",a);
}

static void Feltoltes()
{
    int a=10;
}
```

Ekkor a 'Feltoltes()' eljárásban is létrehozunk egy másik, szintén 'a' nevű változót. Ez a változó sajátja lesz a 'Feltoltes()' -nek, de semmilyen kapcsolatban nem áll a 'Main()' függvény-beli 'a' változóval. A nevek azonossága ellenére ez két különböző változó, két különböző adatterületen helyezkedik el. Ezért hiába tesszük bele az 'a' változóba a '10' értéket, ezen érték a 'Feltoltes()' eljárásbeli 'a' változóba kerül bele, nem a Main()-beli 'a' változóba.

Hogyan kell megosztani változókat eljárások között?

Mivel a hatáskör-szabály szigorú, és nincs kivétel, ezért mindaddig, amíg az 'a' változót a Main() függvény blokkjában deklaráljuk, addig e változót más eljárásokban nem érhetjük el. A megoldás: ne itt deklaráljuk!

```
class PeldaProgram
{
    static int a=0;

    static void Main(string[] args)
    {
        Feltoltes();
        Console.WriteLine("{0}",a);
    }

    static void Feltoltes()
    {
        a=10;
    }
}
```

Azokat a változókat, amelyeket szeretnénk megosztani az eljárásaink között, azokat az eljárásokat összefogó osztály (class) belsejében, de az eljárásokon kívül kell deklarálni. Mivel az eljárásaink static jelzővel vannak ellátva, ezért a közöttük megosztott változókat is ugyanúgy static jelzővel kell ellátni. Ha ezt elmulasztanánk, akkor a static eljárások nem 'látnák' a nem static változókat.

Nézzünk egy összetett feladatot: kérjünk be egy tíz elemű vektor elemeit billentyűzetről, majd írjuk ki a vektor elemeinek értékét a képernyőre egy sorban egymás mellé vesszővel elválasztva, majd számoljuk ki és írjuk ki a képernyőre a vektor elemeinek összegét.

Figyeljük meg, hogy a Main() függvény nem tartalmaz lényegében semmilyen kódot, a tényleges műveleteket eljárásokba szerveztük, és jól csengő nevet adtunk nekik:

```
class PeldaProgram
{
    static int[] tomb=new int[10];
    static int osszeg=0;
    //.....
    static void Main(string[] args)
    {
        Feltoltes();
        Kiiras();
        OsszegSzamitas();
        Console.WriteLine("A tomb elemek osszege={0}"
            ,osszeg);
    }
    //.....
    static void Feltoltes()
    {
        for(int i=0;i<tomb.Length;i++)
        {
            Console.Write("A tomb {0}. eleme=",i);
            string strErtek=Console.ReadLine();
            tomb[i] = Int32.Parse( strErtek );
        }
    }
    //.....
    static void Kiiras()
    {
        Console.Write("A tomb elemei: ");
        for(int i=0;i<tomb.Length;i++)
            Console.Write("{0}, ",tomb[i]);
        Console.WriteLine();
    }
    //.....
    static void OsszegSzamitas()
    {
        osszeg=0;
        for(int i=0;i<tomb.Length;i++)
            osszeg = osszeg + tomb[i];
    }
    //.....
}
```

A változók között meg kellett osztani magát a tömböt, mert azt minden eljárás használta. Ezen túl meg kellett osztani egy 'osszeg' nevű változót is, mert ezen változó értékét az 'OsszegSzamitas()' eljárás számolja ki, és a Main() függvény írja ki a képernyőre.

Feladatok:

16. **Programozási feladat:** Készítsünk olyan eljárást, amely egy megosztott tömb elemeit kéri be billentyűzetről, de csak pozitív számokat fogad el!
17. **Programozási feladat:** Készítsünk olyan eljárást, amely egy megosztott tömb elemeit kéri be billentyűzetről, de nem enged meg ismétlődést (két egyenlő értékű elemet nem fogad el)!
18. **Programozási feladat:** Készítsünk olyan eljárást, amely egy megosztott tömb elemeit generálja 100-900 közötti véletlen számokkal (a véletlen szám generálásához a System.Random osztály példányát kell használni)!
19. **Programozási feladat:** Készítsünk olyan eljárást, amely egy megosztott tömb elemei közül meghatározza a legnagyobb elem értékét, és ezen értéket berakja egy megosztott 'max' nevű változóba!
20. **Programozási feladat:** Készítsünk olyan eljárást, amely egy megosztott tömb elemei közül meghatározza a legnagyobb elem sorszámát, és ezen értéket berakja egy megosztott 'maxI' nevű változóba!
21. **Programozási feladat:** Készítsünk olyan eljárást, amely egy feltöltött, megosztott tömb alapján meghatározza, hogy hány különböző érték szerepel a tömbben! Az eredményt egy 'kulonbozo_db' nevű megosztott változóba helyezi el!
22. **Programozási feladat:** Készítsünk olyan eljárást, amely két megosztott, feltöltött tömb elemeit mint halmazokat kezeli, és elkészíti a két tömb unióját egy 'unio' nevű szintén megosztott tömbbe! Mekkora legyen ezen 'unio' tömb mérete?
23. **Programozási feladat:** Készítsünk olyan eljárást, amely két megosztott, feltöltött tömb elemeit mint halmazokat kezeli, és elkészíti a két tömb metszetét egy 'metszet' nevű szintén megosztott tömbbe! Mekkora legyen ezen 'metszet' tömb mérete?
24. **Programozási feladat:** Készítsünk olyan eljárást, amely két megosztott, feltöltött tömb elemeit mint halmazokat kezeli, és elkészíti a két tömb különbségét egy 'kulonbseg' nevű szintén megosztott tömbbe! Mekkora legyen ezen 'kulonbseg' tömb mérete?
25. **Programozási feladat:** Egy ösvényen tócsák és száraz foltok váltogatják egymást. Egy arra járó sétáló maximum 4 egymás melletti tócsát képes átugrani egyszerre. Az ösvényt egy tömbben írjuk le, ahol a 0 jelenti a tócsát, az 1 jelenti a száraz foltot. Készítsünk olyan eljárást, amely egy ilyen tömb

alapján meghatározza, hogy a sétáló képes-e átjutni az ösvény egyik végéből a másikba anélkül, hogy tócsába kellene lépnie!

26. **Programozási feladat:** Készítsünk olyan eljárást, amely egy megosztott tömb elemeit végigolvasva megadja, hogy a tömb elemei
- a: növekvő sorrendbe vannak-e rendezve,
 - b: csökkenő sorrendbe vannak-e rendezve,
 - c: rendezetlenek.
27. **Programozási feladat:** Készítsünk olyan eljárást, amely egy feltöltött tömb elemeinek ismeretében megadja a leghosszabb olyan szakasz elemszámát, ahol a tömb elemek növekvő sorrendben vannak!
28. **Programozási feladat:** Készítsünk olyan eljárást, amely egy feltöltött tömb elemeinek ismeretében megadja azt a legszűkebb intervallumot, amelyből a tömb elemek származnak!
29. **Programozási feladat:** Készítsünk olyan eljárást, amely egy 'a' tömb elemeit fordított sorrendbe bemásolja egy 'b' tömbbe!
30. **Programozási feladat:** Készítsünk olyan eljárást, amely egy, egész számokkal feltöltött 'a' tömb elemei közül azokat, amelyek párosak, bemásolja egy 'b' tömbbe. A maradék helyeket töltsük fel 0-kal!
31. **Programozási feladat:** Készítsünk olyan eljárást, amely egy 'a' tömb páros elemeit átmásolja egy 'b' tömb elejére, a maradék elemeket pedig a 'b' tömb végére!
32. **Programozási feladat:** Készítsünk olyan eljárást, amely egy 'a' tömb elemeit 1-gyel lejjebb csúsztatja! Az 'a' tömb régi legelső eleme legyen most a legutolsó (ciklikus csúsztatás)!
33. **Programozási feladat:** Készítsünk olyan eljárást, amely egy 'a' tömb elemeit n-nel lejjebb csúsztatják ciklikusan! Az 'n' értékét szintén egy megosztott változóból vegye ki az eljárás (n=0 esetén az elemek helyben maradnak, n=1 esetén előző feladat)! Ha az n értéke nagyobb, mint a tömb elemszáma, akkor is helyesen működjön a program!
34. **Programozási feladat:** Készítsünk olyan eljárást, amely két egyenlő hosszú 'a' és 'b' vektor szorzatát számítja ki
($\text{szorzat} = a[0]*b[0] + a[1]*b[1] + \dots + a[n]*b[n]$)!
35. **Programozási feladat:** Készítsünk olyan eljárást, amely egy polinom helyettesítési értékét számolja ki. A polinom együtthatóit egy 'a' tömb tárolja! A polinom általános alakja $a[0]*x^n + a[1]*x^{n-1} + a[2]*x^{n-2} + \dots + a[n]$. Az x értékét

az eljárás az 'x' változóban találja meg. A számítás gyorsítására használjuk fel a Horner elrendezés adta előnyöket!

XII. Fejezet

Függvények írása

Számold ki és ide vele...

Hernyák Zoltán

A függvény rokon fogalom az eljárással – egy apró különbséggel. A függvény egy olyan eljárás, amely olyan részfeladatot old meg, melynek pontosan egy végeredménye is van – egy érték.

Amennyiben az a részfeladat, hogy egy értékekkel már feltöltött tömb eleminek összegét kell kiszámítani, a végeredmény egyetlen számérték. Amennyiben ezt eljárás segítségével oldjuk meg, úgy a végeredményt egy megosztott változóba kell elhelyezni. Ez sokszor nem túl szerencsés megoldás, mert a függvény kötődik ezen megosztott változó nevéhez. Amennyiben egy másik programba szeretnénk ezt függvényt áthelyezni, úgy vele együtt kell mozgatni a megosztott változót is.

Az ilyen jellegű feladatokat megoldó alprogramokat függvények formájában írjuk meg.

Amennyiben függvényt akarunk írni, két fontos dolgot kell szem előtt tartanunk:

- A függvényeknél rögzíteni kell, hogy milyen típusú értéket adnak majd vissza. Ezt a függvény neve előtt kell feltüntetni (a 'void' helyett).
- A függvények ezek után kötelesek minden esetben egy ilyen típusú értéket vissza is adni! A függvény visszatérési értékét a 'return' kulcsszó után írt kifejezésben kell feltüntetni.

Pl:

```
static int OsszegSzamitas()  
{  
    int sum=0;  
    for(int i=0;i<tomb.Length;i++)  
        sum = sum + tomb[i];  
    return sum;  
}
```

A fenti részben e függvény egy egész számot fog visszaadni, ezt jelöljük a típusnévvel: `int`. A függvény saját változót használ fel segédváltozóként (`sum`) a számítás során, majd a függvény végén a `return sum`-al jelöli, hogy a `sum` változóban szereplő értéket (ami `int` típusú) adja vissza, mint visszatérési értéket.

A függvények meghívása hasonlóan történik, mint az eljárások hívása. Csak mivel a függvény vissza is ad egy értéket, ezért gondoskodni kell róla, hogy ezen értéket a függvényt meghívó programrész fogadja, feldolgozza.

```
static void Main(string[] args)  
{  
    Feltoltes();  
    Kiiras();  
    int osszesen=OsszegSzamitas();  
    Console.WriteLine("A tomb elemek osszege={0}",osszesen);  
}
```

Itt a meghívó ponton az `OsszegSzamitas()` függvény által visszaadott értéket a meghívó rögtön eltárolja egy 'osszesen' nevű változóba, melyből később ki tudja írni a képernyőre ezen értéket.

Ebben a formában a program már nem kell, hogy megosszon 'összeg' változót, hiszen az 'OsszegSzamitas()' már nem ilyen módon közli az eredményt a Main() függvénnyel, hanem egyszerűen visszaadja azt.

```
class PeldaProgram
{
    static int[] tomb=new int[10];

    //.....
    static void Main(string[] args)
    {
        Feltoltes();
        Kiiras();
        int osszesen=OsszegSzamitas();
        Console.WriteLine("A tomb elemek osszege={0}",osszesen);
    }

    //.....
    static int OsszegSzamitas()
    {
        int sum=0;
        for(int i=0;i<tomb.Length;i++)
            sum = sum + tomb[i];
        return sum;
    }

    //.....
}
```

A 11. fejezetben felsorolt eljárások legnagyobb részét át lehet írni függvényekre. Vegyük például a maximális elem értékének meghatározását.

```
static int MaximalisElemErteke()
{
    int max=tomb[0];
    for(int i=1;i<tomb.Length;i++)
        if (tomb[i]>max) max=tomb[i];
    return max;
}
```

E függvény a megosztott 'tomb' elemei közül keresi ki a legnagyobb elem értékét. Meghívása az alábbi módon történhet meg:

```
int maxElemErteke = MaximalisElemErteke();
Console.WriteLine("A tomb legnagyobb elemének
értéke={0}",
    maxElemErteke);
```

Amennyiben azt szeretnénk ellenőrizni, hogy egy adott érték szerepel-e egy – értékekkel már feltöltött – tömb elemei között, akkor az alábbi függvényt írhatnánk meg:

```
static bool Szerepel_E()
{
    for(int i=0;i<tomb.Length;i++)
        if (tomb[i]==keresett_elem) return true;
    return false;
}
```

A fenti függvényben kihasználjuk azt, hogy a 'return' kulcsszó hatására a függvény azonnal megszakítja a futását, és visszatér a hívás helyére. Amennyiben az egyenlőség teljesül, úgy a keresett érték szerepel a tömbben, ezért a ciklus további futtatása már felesleges – megvan a keresett válasz. A 'return false' utasításra már csak akkor jut el a függvény, ha az egyenlőség egyszer sem teljesült. Ekkor a kérdésre a válasz a false érték.

A fenti függvény meghívása a következő módon történhet:

```
bool valasz = Szerepel_E();
if (valasz) Console.WriteLine("Nem szerepel a tömbben");
else Console.WriteLine("Szerepel a tömbben");
```

Más módja, hogy a függvény visszatérési értékét rögtön az 'if' feltételében használjuk fel:

```
if (Szerepel_E()) Console.WriteLine("Nem szerepel a
tömbben");
else Console.WriteLine("Szerepel
a tömbben");
```

A függvények ennél bonyolultabb értékeket is visszaadhatnak:

```
static int[] Feltoltes()
{
    int[] t = new int[10];
    for(int i=0;i<t.Length;i++)
    {
        Console.Write("A tomb {0}. eleme=",i);
        string strErtek=Console.ReadLine();
        t[i] = Int32.Parse( strErtek );
    }
    return t;
}
```

A fenti függvény létrehoz egy *int*-ekből álló, 10 elemű tömböt a memóriában, feltölti elemekkel a billentyűzetről, majd visszaadja a feltöltött tömböt az őt meghívónak:

```
int tomb[] = Feltoltes();
```

Amennyiben a feltöltött tömböt egy megosztott változóba szeretnénk tárolni, akkor azt az alábbi módon kell megoldani:

```
class PeldaProgram
{
    static int[] tomb;

    // .....
    static void Main(string[] args)
    {
        tomb = Feltoltes();
        Kiiras();
        ...
    }
    ...
}
```

Ekkor a 'tomb' változót először csak deklaráltuk, megadtuk a típusát. Majd a Main() függvény belsejében, a megfelelő időpontban meghívjuk a Feltoltes() függvényt, és az általa elkészített és feltöltött tömböt betesszük a megosztott változóba.

Feladatok:

36. **Programozási feladat:** Készítsünk olyan függvényt, amely egy téglalap két oldalának ismeretében kiszámítja a téglalap területét!
37. **Programozási feladat:** Készítsünk olyan függvényt, amely meghatározza két szám legnagyobb közös osztóját!
38. **Programozási feladat:** Készítsünk olyan függvényt, amely eldönti egy számról, hogy hány osztója van!
39. **Programozási feladat:** Készítsünk olyan függvényt, amely eldönti egy számról, hogy prímszám-e!
40. **Programozási feladat:** Készítsünk olyan függvényt, amely megszámolja, hogy egy adott érték hányszor szerepel egy tömbben!
41. **Programozási feladat:** Készítsünk olyan függvényt, amely meghatározza két megosztott tömb elemeinek ismeretében a metszetük elemszámát!
42. **Programozási feladat:** Készítsünk olyan függvényt, amely meghatározza két megosztott tömb elemeinek ismeretében a különbségük elemszámát (hány olyan elem van a két tömbben összesen, amelyik csak az egyik tömbben van benne)!
43. **Programozási feladat:** Készítsünk olyan függvényt, amely meghatározza egy tömb elemeinek átlagát!
44. **Programozási feladat:** Készítsünk olyan függvényt, amely meghatározza egy tömbben hány 3-al osztható, de 5-el nem osztható szám van!

Eljárások és függvények középfokon

Hernyák Zoltán

Amikor eljárást (vagy függvényt) írunk, az alprogram törzsében sokszor hivatkozunk ilyen megosztott (közös) változókra. E változókban az eljárás a program előző részei által előállított adatokat kap meg, vagyis **bemenő adatokat** fogad. Az eljárás ezen adatok segítségével újabb értékeket állíthat elő, melyeket elhelyezhet megosztott változókban, ahol a többi eljárás megtalálhatja őket. Így állít elő az eljárás **kimenő adatokat**.

Bemenő adatokat az eljárás azonban nem csak a megosztott változókban vehet át, hanem paraméterekben is.

A paramétereket az eljárás fejrészában kell feltüntetni. Fel kell sorolni vesszővel elválasztva a bemenő adatok típusát, és egy azonosítót (nevet) kell adni ezen adatoknak. Ezt a listát **formális paraméterlistának** hívjuk.

Pl:

```
static void Kiiras(int a,int b)
{
    Console.WriteLine("A {0}+{1}={2}",a,b,a+b);
}
```

A fenti példában ezen eljárás két bemenő értéket vár. Mindkettő 'int' típusú, vagyis egész számok. Az eljárás törzsében a paraméterekre a formális paraméterlistában feltüntetett azonosítókkal (név) hivatkozhatunk. A fenti eljárás kiírja a két számot a képernyőre, majd a két szám összegét is.

Amikor ezen eljárást meg akarjuk hívni, akkor ezen eljárásnak a fenti bemenő adatokat át kell adni. A hívás helyén feltüntetett paraméterlistát (mely az aktuális bemenő adatok értékét tartalmazza) **aktuális paraméterlistának** hívjuk. Aktuális paraméterlistában már sosem írunk típusokat, hanem konkrét értékeket!

```
Kiiras(12,15);
```

A fenti kódrészlet elindítja a Kiiras eljárást, átadván neki a két bemenő adatot. Az eljárás elindulása előtt feltölti a paraméterváltozókat az aktuális értékekkel (a=12, b=15), majd utána kezdődik az eljárástörzs utasításainak végrehajtása.

Aktuális paraméterlista

```
Kiiras(12,15);
```

```
static void Kiiras(int a,int b)
```

Formális paraméterlista

Bizonyos szempontból nézve a formális paraméterlista egyúttal változó-deklarációnak is minősül, hiszen a formális paraméterlistában lényegében típusok és nevek vannak megadva. Az eljárás törzsében ezekre a paraméterekre mint adott

típusú változókra hivatkozhatunk. Ezen változók kezdőértékkel rendelkeznek, a **kezdőértékeket az aktuális paraméterlistában adjuk meg.**

Ennek megfelelően az aktuális és a formális paraméterlistára szigorú szabályok vonatkoznak:

- az aktuális paraméterlistában pontosan annyi értéket kell felsorolni, amennyit a formális paraméterlista alapján az eljárás vár tőlünk,
- az aktuális paraméterlistában pontosan olyan típusú értékeket kell rendre megadni, mint amelyet a formális paraméterlista szerint az adott helyen fel kell tüntetni.

Hibásak az alábbiak:

```
Kiiras(12.5,15);           // 12.5 nem 'int' !  
Kiiras(12);                // túl kevés paraméter  
Kiiras(12,15,20);         // túl sok paraméter  
Kiiras("Hello",20);       // a "Hello" nem 'int' típusú
```

Helyesek az alábbiak:

```
Kiiras(12,15);  
Kiiras(3*4,15*3-12);  
int x=8; Kiiras(x,x+2);
```

Megállapíthatjuk, hogy az aktuális paraméterlistában olyan értéket kell írunk, amelynek a végeredménye jelen esetben 'int' típusú. A hívás helyére írhatunk számkonstanst (literál), kifejezést – melynek kiszámítása int-et eredményez, illetve változót is (ekkor a változó aktuális értékét adjuk át).

A konstans (literál) és a változó is kifejezésnek minősül, csak egyszerű kifejezés. Ezért általánosan azt mondhatjuk, hogy **az aktuális paraméterlistában olyan kifejezést kell írunk, melynek típusa megfelel a formális paraméterlistában leírt követelményeknek.** Ezt a típus-megfelelőséget a kompatibilis típusok szabályai írják le. Az OO nyelvekben a típuskompatibilitást is az OO szabályai írják le. Egyelőre annyit jegyezzünk meg, hogy az 'int' kompatibilis a 'double'-val, az 'int'-ek (sbyte, uint, int, ...) kompatibilisek egymással, csakúgy mint a double típusok is (double, float), persze ha az aktuális érték a kívánt típusban az adott pillanatban elfér.

Pl:

```
static void Kiiras(double a)  
{  
    Console.WriteLine("A szám fele={0}",a/2);  
}
```

Ezen eljárás egy tetszőleges racionális számot vár, majd kiírja az adott szám felét.

```
Kiiras(12);
```


Aktuális paraméterlista

```
Kiiras(12);
```

int

átalakítás

double

```
static void Kiiras(double a)
```

Formális paraméterlista

Ebben az esetben az aktuális paraméterlistában nem egy tört, hanem egy egész számot adtunk át. Ez ugyan nem pontosan ugyanolyan típusú, mint amit a formális paraméterlistában leírtunk, de az aktuális érték (12) konvertálható (kompatibilis) a kért típusra.

```
int x=12; Kiiras(x);
```

Ezen példa szerint is az aktuális érték egy 'int' típusú érték, de ez elfogadható, ha a fogadó oldalon akár 'double'-t is képesek vagyunk fogadni.

Fordítva nem igaz:

```
static void Kiiras_Egesz(int a)
{
    Console.WriteLine("A szám kétszeres={0}", a*2);
}
```

```
double z=12.5; Kiiras_Egesz (z);
```

```
double z=12.5;
```

```
Kiiras_Egesz(z);
```

double

átalakítás?!
(nem lehet)

int

```
static void Kiiras_Egesz(int a)
```

Értelemszerűen a küldő oldal (aktuális paraméterlista) hiába próbálná átadni a 12.5 értéket, a fogadó oldal (formális paraméterlista) csak egész típusú értéket tud átvenni. Ezért ezt a paraméterátadást a C# fordító nem engedi, még akkor sem, ha ...

```
double z=12; Kiiras_Egesz (z);
```

Ekkor hiába van a 'z' változóban olyan érték, amelyet a fogadó oldal akár át is tudna venni, ez általános esetben nem biztonságos. A hívás helyére olyan típusú kifejezést

kell írunk, amely ennél több garanciát ad. Ezért nem engedi meg a C# fordító, hogy a típusok ennyire eltérjenek egymástól.

Ennél persze bonyolultabb típusok is szerepelhetnek a formális paraméterlistában:

```
static void Kiiras(int[] tomb)
{
    Console.Write("A tomb elemei: ");
    for(int i=0;i<tomb.Length;i++)
        Console.Write("{0}, ",tomb[i]);
    Console.WriteLine();
}
```

A fenti példában a 'Kiiras' eljárás egy komplett tömböt vár paraméterként. A tömb elemei 'int' típusú értékek kell hogy legyenek. A hívás helyére ekkor természetesen egy ennek megfelelő értéket kell írni:

```
int[] x = new int[10];
Kiiras( x );
```

Itt az 'x' változó típusa 'int[]', ami megfelel a fogadó oldali elvárásoknak, ezért a fenti eljáráshívás típusában megfelelő, így helyes is.

Feladatok:

- 45.**Programozási feladatok:** készítsünk olyan függvényt, amely kap paraméterként két számot, és visszaadja a két szám közül a nagyobbik értékét! Amennyiben a két szám egyenlő, úgy az első szám értékét kell visszaadni!
- 46.**Programozási feladatok:** készítsünk olyan függvényt, amely kap két *int* tömböt paraméterként, mindkét tömbben 3-3 szám van! A tömbök egy-egy háromszög oldalainak hosszát írják le! Adjuk vissza a nagyobb kerületű háromszög kerületének értékét!
- 47.**Programozási feladatok:** készítsünk olyan függvényt, amely meghatározza egy paraméterben megadott, *int* típusú tömbben a leghosszabb egyenlő elemekből álló szakasz hosszát!
- 48.**Programozási feladatok:** készítsünk olyan eljárást, amely egy megosztott tömböt feltölt véletlen elemekkel egy megadott intervallum elemei közül úgy, hogy két egyenlő érték ne forduljon elő a tömbben! Az intervallum kezdő és végértékeit paraméterként adjuk át!

Programozás tankönyv

XIV. Fejezet

Eljárások és függvények felsőfokon

Hernyák Zoltán

Kérdés: tudunk-e a paraméterváltozókon keresztül értéket visszaadni?

Erre sajnos a C#-ban nem egyszerű válaszolni. A helyzet igen bonyolult mindaddig, amíg a referencia típusú változókkal alaposan meg nem ismerkedünk.

Addig is használjuk az alábbi szabályokat:

- amennyiben a paraméterváltozók típusa a nyelv alaptípusai (int, double, char, string, ...) közül valamelyik, úgy nem tudunk értéket visszaadni a hívás helyére,
- ha a paraméter típusa tömb, akkor a tömbelemeken keresztül azonban igen.

Pl.:

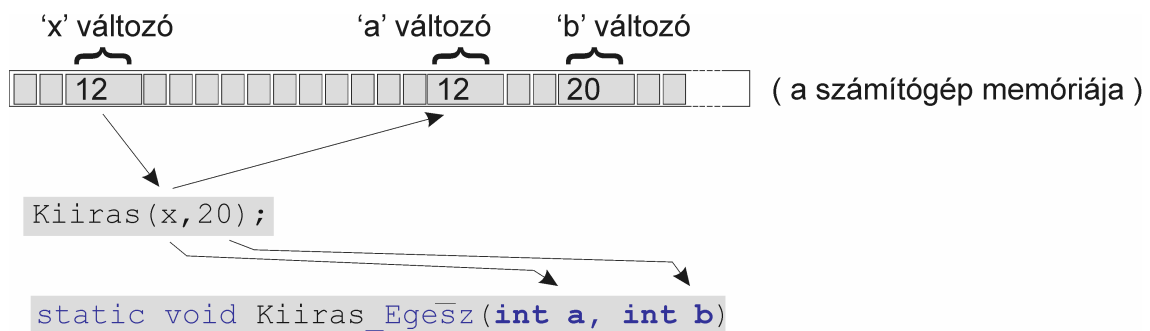
```
static void Kiiras(int a, int b)
{
    Console.WriteLine("A {0}+{1}={2}", a, b, a+b);
    a = a+b;
}
```

Ha a fenti eljárást az alábbi módon hívjuk meg ...

```
int x=12;
Kiiras(x,20);
Console.WriteLine("A hivas utan az x erteke x={0}",x);
```

... ekkor kiíraskor az 'x' változó értéke még mindig 12 lesz. Mivel a paraméterátadás során a híváskori érték (12) átadódik az eljárás 'a' változójába mint kezdőérték (a=12), de utána a hívás helyén szereplő 'x' változó, és a fogadó oldalon szereplő 'a' változó között minden további kapcsolat megszűnik. Ezért hiába teszünk az 'a' változóba az eljáráson belül más értéket (a+b), az nem fogja befolyásolni az 'x' értékét, marad benne a 12.

A fenti technikát **érték szerinti paraméterátadásnak** nevezzük. Ekkor az aktuális paraméterlistában feltüntetett értéket a fogadó oldal átveszi – a nélkül, hogy a kapcsolatot fenntartaná. Az érték egyszerűen átmásolódik a fogadó oldal változóiba.



Mint az ábrán is látszik, az 'x' aktuális értéke (12) átmásolódik a memória külön területén helyet foglaló 'a' változóba. E miatt az `a=a+b` értékadás eredménye is ezen a külön területen kerül tárolásra, és nem zavarja, nem változtatja meg az 'x' értékét.

Amennyiben azonban a paraméterátadás-átvétel során tömböket adunk át, úgy a helyzet megváltozik:

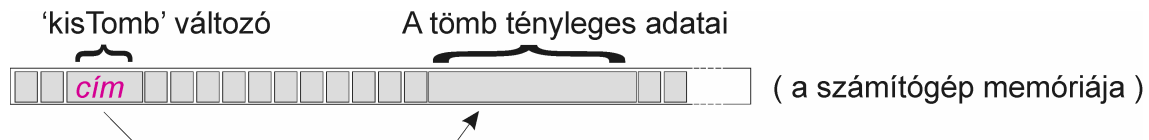
```
static void Feltoltes(int[] tomb)
{
    tomb[0] = 10;
}
```

```
static int[] kisTomb=new int[10];
Feltoltes( kisTomb );
Console.WriteLine("A 0. tombelem={0}", kisTomb[0] );
```

A hívás helyén szereplő kis tömb még feltöltetlen, amikor átadjuk a 'Feltoltes' eljárásnak. Az eljárás fogadja a tömböt a 'tomb' változóban, majd megváltoztatja a 'tomb' elemeinek értékét. Ezen változás azonban beleíródik a 'kisTomb' által reprezentált tömb-be is, ezért az eljárás lefutása után a kisTomb[0] értéke már 10 lesz.

Az ilyen jellegű paraméterátadást **referencia-szerinti átadásnak** nevezzük!

A referencia típus nem jelentkezik külön a változó deklarációja során mint külön kulcsszó, vagy egyéb jelzés, ezért nehéz felismerni. Egyelőre fogadjuk el szabályként, hogy azok a változók lesznek referencia típusúak, amelyek esetén a 'new' kulcsszót kell használni az érték megadásakor (példányosítás).



A referencia-típusú változóknál dupla memóriafelhasználás van. Az elsődleges területen ilyenkor mindig egy memóriacímet tárol a rendszer. Ezen memóriacím a másodlagos memóriaterület helyének kezdőcíme lesz. Ezen másodlagos memóriaterületet a 'new' hozza létre.

Az nyelvi alaptípusok (int, double, char, string, ...) értékének képzésekor nem kell a 'new' kulcsszót használni ...

```
int a = 10;
```

... de a tömböknél igen:

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

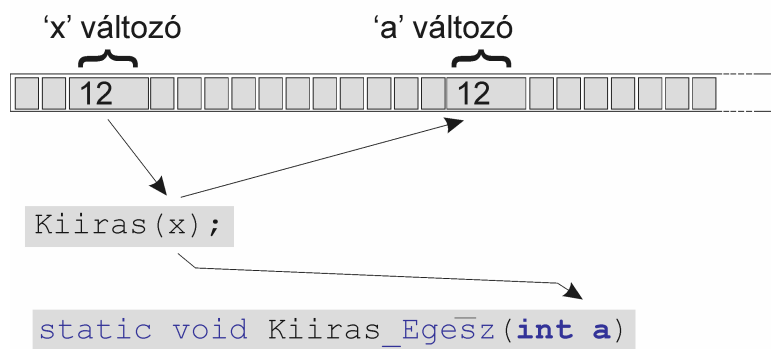
Az ilyen jellegű változókat általában az jellemzi, hogy viszonylag nagy memória-igényük van. Egy 'int' típusú érték tárolása csak 4 byte, míg a fenti tömb tárolása 40 byte.

Az érték szerinti paraméterátadás során az érték az átadás pillanatában két példányban létezik a memóriában – lemásolódik:

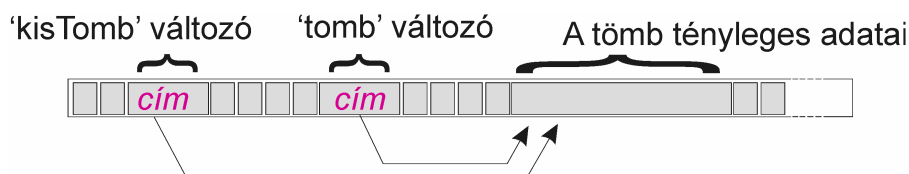
```
int x=12;  
Kiiras(x);
```

```
static void Kiiras(int a)  
{  
...  
}
```

A számítógép memóriája:



A referencia típusú változók esetén azonban nem másolódik le az érték még egyszer a memóriában (ami a tömb esetén a tömbelemek lemásolását jelentené), hanem csak a szóban forgó tárterület címe (memóriacíme) adódik át. Ekkor a fogadó oldal megkapja a memóriaterület címét, és ha beleír ebbe a memóriaterületbe, akkor azt a küldő oldal is észlelni fogja majd:



```
static void Feltoltes(int[] tomb)  
{  
    tomb[0] = 10;  
}
```

```
static int[] kisTomb=new int[10];  
Feltoltes( kisTomb );
```

A fenti ábrán látszik, hogy a 'kisTomb' változónk aktuális értéke átmásolódik a 'tomb' aktuális értékébe. Csakhogy a 'kisTomb' értéke egy memóriacím, ez az, ami valójában átmásolódik a 'tomb' változóba, és nem ténylegesen a tömb elemei.

Mivel a 'tomb' változó is ismerni fogja a tömbelemek tényleges helyét a memóriában, ezért amennyiben megváltoztatja azokat, úgy a változtatásokat a 'kisTomb'-ön keresztül is el tudjuk majd érni, hiszen mindkét esetben ugyanazokra a tömbelemekre hivatkozunk a memóriában.

Ez az oka annak, hogy a tömbelemek értékének módosítása a fogadó oldalon maradandó nyomot hagy a memóriában – amikor az eljárás már befejezte a futását, az értékek akkor is hozzáférhetőek.

A referencia-szerinti paraméterátadás is érték szerinti paraméterátadás, a változó értéke ilyenkor egy memóriacím, mely mint érték másolódik át a paraméterváltozókba. Az már más kérdés, hogyha a fogadó oldal megkapja a terület kezdőcímét, akkor azt meg is változtathatja. Mivel az eredeti változó is ugyanezen területre hivatkozik, ezért a változásokat a küldő oldalon később észlelni, (pl: kiolvasni) lehet.

Amennyiben a fogadó oldalon referencia típusú paramétert fogadunk, úgy a küldő oldalon csak olyan kifejezés szerepelhet, melynek végeredménye egy referencia:

```
Feltoltes( new int[20] );
```

Az előzőekben bemutatott kifejezés létrehoz egy 20 elemű int tömböt. A new foglalja le számára a memóriát, majd visszaadja a terület kezdőcímét. Ezt a kezdőcímet általában egy megfelelő típusú változóban tároljuk el, de amennyiben erre nincs szükség, úgy a memóriacímet tárolás nélkül átadhatjuk egy eljárásnak paraméterként.

```
static int[] Feltoltes(int[] tomb)
{
    for(int i=0;i<tomb.Length;)
    {
        Console.WriteLine("A tomb {0}. eleme=",i);
        string strErtek=Console.ReadLine();
        int x = Int32.Parse( strErtek );
        if (x<0) continue;
        tomb[i] = x;
        i++;
    }
    return tomb;
}
```

Az előzőekben bemutatott függvény paraméterként megkapja egy int típusú tömb kezdőcímét. A tömb elemeit billentyűzetről tölti fel oly módon, hogy csak pozitív számot fogad el. A feltöltött tömb címét (referenciáját) visszaadja.

Mivel a függvény ugyanazt a memóriacímet adja vissza, mint amit megkapott, ennek nem sok értelme látszik. Figyeljük meg azonban a hívás helyét:

```
int[] ertekek = Feltoltes( new int[40] );
```


A hívás helyén a frissen elkészített tömb memóriacímét nem tároljuk el, hanem rögtön átadjuk a függvénynek. A tömb a hívás helyén készül el, üresen (csupa 0-val inicializálva), a függvény feltölti ezt a tömböt elemekkel, majd visszaadja az immár feltöltött tömb címét.

A fenti megoldás azért szép, mert egy sorban oldja meg a tömb létrehozását, feltöltését. Amennyiben a Feltoltes() csak egy eljárás lenne, úgy az alábbi módon kellene meghívni:

```
int[] ertekek = new int[40];  
Feltoltes( ertekek );
```

Amennyiben egy alaptípusú paraméter-változón keresztül szeretnénk értéket visszaadni, úgy azt jelölni kell:

```
static void ParosElemek(int[] tomb, ref int db, ref int osszeg)  
{  
    db=0;  
    osszeg=0;  
    for(int i=0;i<tomb.Length;i++)  
        if (tomb[i] % 2 == 0)  
        {  
            db++;  
            osszeg = osszeg + tomb[i];  
        }  
}
```

A fenti eljárás két értéket is előállít – a páros tömbelemek számát, és összegét. Mivel ez már kettő darab érték, ezért nem írhatjuk meg függvényként. Egy függvény csak egy értéket adhat vissza.

Az ilyen jellegű paraméterátadást **cím szerinti paraméterátadásnak** nevezzük. Ekkor a hívás helyén is jelölni kell, hogy az átadott változóban kimenő adatokat várunk:

```
int x=0,y=0;  
ParosElemek(tomb, ref x, ref y);  
Console.WriteLine("A paros elemek db={0},  
osszeguk={1}", x, y);
```

A **'ref'** kulcsszóval kell jelölni, hogy az adott paraméterben értéket is vissza szeretnénk adni. Ugyanakkor a **'ref'** kulcsszó ennél többet jelent – a **'ref'** paraméterben értéket is adhatunk át az eljárás számára. Ez a paraméter klasszikus **átmenő** paraméter.

Az átmenő paraméterek egyidőben bemenő és kimenő paraméterek. Az eljárásban ezen paraméter értékét felhasználhatjuk, és meg is változtathatjuk.

```
int x,y;
ParosElemek(tomb, ref x, ref y);
Console.WriteLine("A paros elemek db={0},
osszeguk={1}",x,y);
```

Amennyiben a paraméterként átadott változók nem rendelkeznének értékkel a hívás pillanatában (definiálatlan változó), úgy a C# fordító szintaktikai hibát jelez.

Ez a fenti függvény esetén egyébként értelmetlen, hiszen a függvény *db* és *osszeg* paraméterváltozóinak kezdőértéke érdektelen. A probléma forrása az, hogy e paraméterek csak kimenő értékeket hordoznak, bemenő értékük érdektelen. Az ilyen paramétereket nem a 'ref', hanem az 'out' kulcsszóval kell megjelölni!

```
static void ParosElemek(int[] tomb, out int db, out int osszeg)
{
    db=0;
    osszeg=0;
    for(int i=0;i<tomb.Length;i++)
        if (tomb[i] % 2 == 0)
        {
            db++;
            osszeg = osszeg + tomb[i];
        }
}
```

A hívás helyén is:

```
int dbszam,summa;
ParosElemek(tomb, out dbszam, out summa);
```

A 'ref' és az 'out' módosítók között az egyetlen különbség, hogy a hívás helyén (aktuális paraméterlistában) szereplő változóknak 'out' esetén nem kötelező kezdőértéküknek lenniük. Valamint egy 'out'-os paraméterváltozót a fordító kezdőérték nélkülinek tekint, és a függvényben kötelező értéket adni ezeknek a paraméterváltozóknak a függvény visszatérési pontja (return) előtt.

A 'ref' esetén a függvényhívás előtt a változóknak kötelező értéket felvenni, és a függvényben nem kötelező azt megváltoztatni.

Aki esetleg keresné, 'in' módosító nincs a C#-ban, ugyanis a paraméterek alapértelmezésben bemenő adatok, vagyis az 'in' módosítót ki sem kell írni.

```
static void Csere(ref int a, ref int b)
{
    int c;
    c=a;
    a=b;
    b=c;
}
```

A fenti eljárás a paramétereként megkapott két változó tartalmát cseréli fel. Mivel ilyenkor érdekes a paraméterek aktuális értéke is, ezért a 'ref' kulcsszót kell használni.

A fenti eljárás kitűnően felhasználható egy rendező eljárásban:

```
static void Rendezes(int[] tomb)
{
    for(int i=0;i<tomb.Length-1;i++)
        for(int j=i+1;j<tomb.Length;j++)
            if (tomb[i]>tomb[j]) Csere(ref tomb[i],ref
tomb[j]);
}
```

XV. Fejezet

„WinForm”

**Radványi Tibor
Király Roland**

A Windows Formok

Hibakezelés

A C# nyelv a futásidejű hibák kiszűrésére alkalmas eszközt is tartalmaz, mely eszköz segítségével a programjainkba hibakezelő (*kivétel kezelő*) kódrészeket építhetünk.

A kivétel kezelő eljárások lényege, hogy elkerüljük a futás közbeni hibák esetén felbukkanó hibaüzeneteket, és megvédjük programjainkat a váratlan leállástól.

A try és a catch

A try parancs segítségével a programok egyes részeihez hibakezelő rutinokat rendelhetünk, melyek hiba esetén az általunk megírt hibakezelő eljárásokat hajtják végre, s ami nagyon fontos, megakadályozzák a program leállítását.

A catch segítségével azokat a kivételeket kaphatjuk el, melyek a try blokkban keletkeznek. Itt határozhatjuk meg, hogy milyen rutinok fussanak le és hogyan kezeljék a felmerülő hibákat. A catch segítségével különböző hibák egyidejű kezelését is megvalósíthatjuk, de erről majd bővebben szólnunk a fejezet későbbi részeiben. Most nézzünk meg egy példát a try használatára!

```
using System;

namespace ConsoleApplication6
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            string s;
            int i;
            Console.WriteLine("Kérem gépeljen be egy tetszőleges mondatot!");
            s=Console.ReadLine();
            try
            {
                for (i=0;i<20;i++)
                {
                    Console.WriteLine("Az s string {0}. eleme = {1}",i,s[i]);
                }
            }
            catch
```

```

        {
            Console.WriteLine("Hiba a program futása során...");
        }

        Console.ReadLine();
    }
}

```

Amennyiben futtatjuk a fenti programot, és a beolvasásnál 20 karakternél rövidebb mondatot adunk meg, a hibakezelő eljárás elindul, mivel a try a catch blokkhoz irányítja a vezérlést. A catch blokkban definiált hibakezelő rész lefut, vagyis a képernyőn megjelenik a hibaüzenet. Abban az esetben, ha nem használjuk a hibakezelő eljárásokat, a programunk leáll, és a hiba kezelését a .NET vagy az operációs rendszer veszi át. Ebből a példából is jól látható milyen nagy szükség lehet a kivételkezelőre, ha jól működő programokat akarunk írni. Fokozottan érvényes ez azokra a programokra, ahol a felhasználó adatokat visz be. A következő programban erre láthatunk példát. Amennyiben az inputról nem megfelelő adatok érkeznek, működésbe lép a kivétel-kezelés, a catch blokkban elkapjuk a hibát és kiírjuk az okát a képernyőre.

```

using System;

namespace ConsoleApplication4
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            int i;

            try
            {
                Console.WriteLine("Az i értéke? : ");
                i=Convert.ToInt32(Console.ReadLine());
            }

            catch(Exception e){Console.WriteLine(e.Message);}
            Console.ReadLine();
        }
    }
}

```

A programban látható, hogy a try után kapcsos zárójelek közt adjuk meg a hibát okozható program részletet. A catch hibakezelő rutinait szintén kapcsos zárójelek közé kell írni, ezzel jelezve a fordítónak, hogy hol kezdődik és végződik a hibakezelés.

A fenti példában a túlindexelés és a típus különbségek mellett előfordulhatnak más hibák is, melyek a program írásakor rejtve maradnak. Az ilyen jellegű hibákra az előző kód nincs felkészítve, mivel nem definiáltuk a lehetséges hibákat. Ez azt jelenti, hogy a try blokkban előforduló bármely hiba esetén ugyanaz a hibaüzenet jelenik meg a képernyőn.

A catch blokkban lehetőségünk van a különböző okok miatt keletkezett hibák szétválasztására, a hiba típusának meghatározására.

A catch parancs a kivételt paraméterként is fogadhatja, ahogy a fenti példában láthattuk. A paraméterként megadott változó System.Exception típusú, melyből kiolvashatjuk a hiba okát, amit az e.Message rutinnal kapunk meg, és a megfelelő hibakezelőt indíthatjuk el.

```
catch ( System.Exception e )
{
    Console.WriteLine(e.Message);
    //hibák kezelése
}
```

A képernyőn megjelenő hibaüzenet nem nagyon beszédes, illetve gyakran túlságosan sok, nehezen érthető információt tartalmaz. A célunk sem az, hogy a programunk használóját hosszú, értelmezhetetlen üzenetekkel terheljük, mert így nem teszünk többet, mint az eredeti hiba-ablak. Ráadásul a felhasználó nem is nagyon tudja kezelni a keletkezett hibákat, még annak ellenére sem, hogy kiírjuk a képernyőre a hiba minden paraméterét.

Sokkal jobb megoldás, ha megállapítjuk a hiba okát, majd a megfelelő hibakezelés aktivizálásával meg is szüntetjük azt. *(Ekkor még mindig ráérünk kiírni a képernyőre, hogy hiba történt, és a javított hibáért a felhasználó nem is haragszik annyira... Talán még elismerően bólint is...)*

Írhatunk olyan catch blokkokat is, melyek egy bizonyos típusú hiba elfogására alkalmasak.

```
int a=0;
double c;

try
{
    c = 10 / a + 30;
}

catch (ArithmeticException ar)
{
    Console.WriteLine("Aritmetikai hiba : {0}",ar);
}
```

A catch blokkok sorrendje sem mindegy. A helyes megközelítés az, ha az általános hibák elé helyezzük azokat a hibákat, melyekre már eleve számítani lehet a programban, mint pl.: a fenti matematikai hiba.

A System névtérben rengeteg kivétel típust definiáltak. A fontosabbakat az alábbi táblázatban foglaltuk össze.

| Kivétel neve | Leírása |
|-----------------------------|---|
| MemberAccessExcepction | Tagfüggvény hozzáférési hiba |
| ArgumentException | Hibás tagfüggvény-paraméter |
| ArgumentNullException | Null értékű tagfüggvény paraméter |
| ArithmeticException | Matematikai művelet-hiba |
| ArrayTypeMismatchException | Tömbtípus hibája (érték tároláskor) |
| DivideByZeroException | Nullával való osztás |
| FormatException | Hibás paraméter-formátum |
| IndexOutOfRangeException | Tömb túl, vagy alulindexelése |
| InvalidCastException | Érvénytelen típus átalakítás |
| NotFiniteNumberException | A keletkezet érték nem véges (hibás számalak) |
| NullReferenceException | Null értékre való hivatkozás |
| NotSupportedException | Nem támogatott tagfüggvény |
| OutOfMemoryException | Elfogyott a memória |
| OverflowException | Túlcsordulás (checked esetén) |
| StackOverflowException | Verem túlcsordulás |
| TypeInitializationException | Hibás típus beállás (static konstruktornál) |

A finally blokk

Sokszor lehet szükség arra is, hogy egy program részlet hibás és hibátlan működés esetén is lefusson. Tipikus példa erre a fájlkezelés, ahol a megnyitott fájlt hiba esetén is le kell zárni. Az ilyen típusú problémákra nyújt megoldást a finally kulcsszó.

A finally blokkban elhelyezett kód mindig lefut, függetlenül az előtte keletkezett hibáktól. *(Nem igaz ez arra az esetre, ha a program végzetes hibával áll le.)*

A következő példa bemutatja, hogyan alkalmazhatjuk a nyelv finally kulcsszavát:

```
int a=0;
double c;

try
{
    c=10/a;
}
catch (ArithmeticException ar)
{
    Console.WriteLine("Aritmetikai hiba : {0}",ar);
}
```



```
finally
{
    Console.WriteLine("A program ezen része
    mindenképpen lefut");
}
```

Kivételek feldobása

A C# nyelv lehetőséget ad a saját magunk által definiált kivételek használatára is. A kivételek dobása a `throw` kulcsszóval történik.

```
throw (exception);
throw exception;
```

A program bármely szintjén, bárhol *"dobhatjuk"* a kivételt a `throw` segítségével, és egy tetszőleges `catch` blokkal el is kaphatjuk. Amennyiben nem kapjuk el sehol, az a `Main()` függvény szintjén is megjelenik, végül az operációs rendszer lekezeli a saját hibakezelő rutinjával, ami legtöbbször azt jelenti, hogy a programunk leáll.

A következő példa bemutatja, hogyan dobhatunk saját kivételt.

```
class Verem
{
    public Object Pop()
    {
        if (vm>0) { vm--; return t[vm]; }
        else throw new Exception("Üres a verem");
    }
}
```

A példában a verem tetejéről akarunk levenni egy elemet akkor, ha az nem üres. Amennyiben nincs már elem a veremben, ezt egy kivétel dobásával jelezzük. A kivétel feldobását a `throw` végzi, melynek paramétere egy `exception` osztály: `Exception("Hibaüzenet")`. A kivétel dobásakor a `new` kulcsszót használtuk, mivel a definiált kivétel is egy osztály, így példányosítani kellett. A konstruktor paramétere egy `string`, melyben a hibaüzenetet adhatjuk meg, amit a kivétel elkapásakor kiírhatunk. *(Ha nem kapjuk el, az operációs rendszer akkor is kiírja egy hibaablakban a hibaüzenetként megadott stringet.)*

Checked és unchecked

A C# nyelv tartalmaz két további kulcsszót a kivételek kezelésére és a hibák javítására. Az egyik a checked a másik pedig az unchecked. Amennyiben egy értékadáskor egy változóba olyan értéket szeretnénk elhelyezni, mely nem fér el annak értéktartományában, OverflowException hibával leáll a programunk. Jobb esetben az ilyen hibákat el tudjuk kapni a megfelelő catch{} blokkal, de az unchecked kulcsszó használatával megakadályozhatjuk a kivétel keletkezését is, mivel ilyenkor elmarad a hibaellenőrzés.

```
unchecked
{
    int a=2000000000000000;
```

Az értékadás megtörténik, a változóba bekerül a csonkított érték. *(Amekkora még elfér benne.)*

A checked alkalmazásával pontosan az előbbi folyamat ellenkezőjét érjük el. Az ellenőrzés mindenképpen megtörténik, és kivétel keletkezik a hiba miatt.

A checked és unchecked kulcsszavak nem csak blokként használhatóak: checked{}, unchecked{}, hanem kimondottan egy kifejezés vagy értékadás vizsgálatánál is. Ekkor a következő formában írhatjuk őket:

```
checked( kifejezés, művelet, vagy értékadás);
unchecked(kifejezés, művelet, vagy értékadás);;
```

Ebben a formában csak a zárójelek közé írt kifejezésekre, vagy egyéb műveletekre vonatkoznak. Alapértelmezés szerint a checked állapot érvényesül a programokban található minden értékadásra és műveletre. Csak nagyon indokolt esetekben használjuk ki az unchecked nyújtotta lehetőségeket, mert könnyen okozhatunk végzetes hibákat a programokban.

A fentiek ismeretében elmondhatjuk, hogy a hibakezelés, a kivételek kezelése nagyon fontos és hasznos lehetőség a C# nyelvben. Nélküle nehéz lenne elképzelni hibátlanul működő programokat. Igaz, hogy a kivételkezelést alkalmazó programok sem tökéletesek, de talán nem állnak le végzetes hibákkal, nem keserítik sem a programozó, sem a felhasználó életét.

Programozási feladatok

1. Írjon programot, mely egy meghatározott végjelig szám párokat olvas be a billentyűzetről és a szám párok első elemét elosztja a másodikkal, az eredményt pedig kiírja a képernyőre! Nullával való osztás esetén a program jelezze a hibát a felhasználónak!
2. Készítsen programot, mely verem kezelést valósít meg egy n elemű vektorban! A program a `Pop()` és a `Push()` műveletek hibája esetén dobjon kivételt!
3. Az előző programot módosítsa úgy, hogy a kivételek dobáskor a hibaüzenetek megjelenjenek a képernyőn!
4. Készítsen programot, mely egy n elemű tömbbe olvas be egész típusú értékeket, de csak páros számokat fogad el!
5. Módosítsa az előző programot úgy, hogy a beolvasásnál csak 3-mal és 5-tel osztható számokat fogadjon el!
6. Készítsen programot, mely a fejezetben felsorolt kivételek nevét kiírja a képernyőre. A kivételek nevét egy konstans tömbben tárolja!
7. Írjon programot, mely számokat olvas be a billentyűzetről, majd a képernyőre írja a beolvasott számok négyzetét! A program csak számokat fogadjon el inputként. Nem számtípus beolvasásakor kivételekkel kezelje le az előforduló hibákat!
8. Készítsen programot, mely számokat olvas be a billentyűzetről egy string típusú változóba. A beolvasott számot alakítsa szám típusúvá, majd tárolja egy `int` típusú változóban. A beolvasás és a konverzió során keletkező hibákat kivételekkel kezelje le! (Hiba az is, ha a felhasználó a string-ben nem számokat ad meg. Erre külön hívjuk fel a figyelmét!)

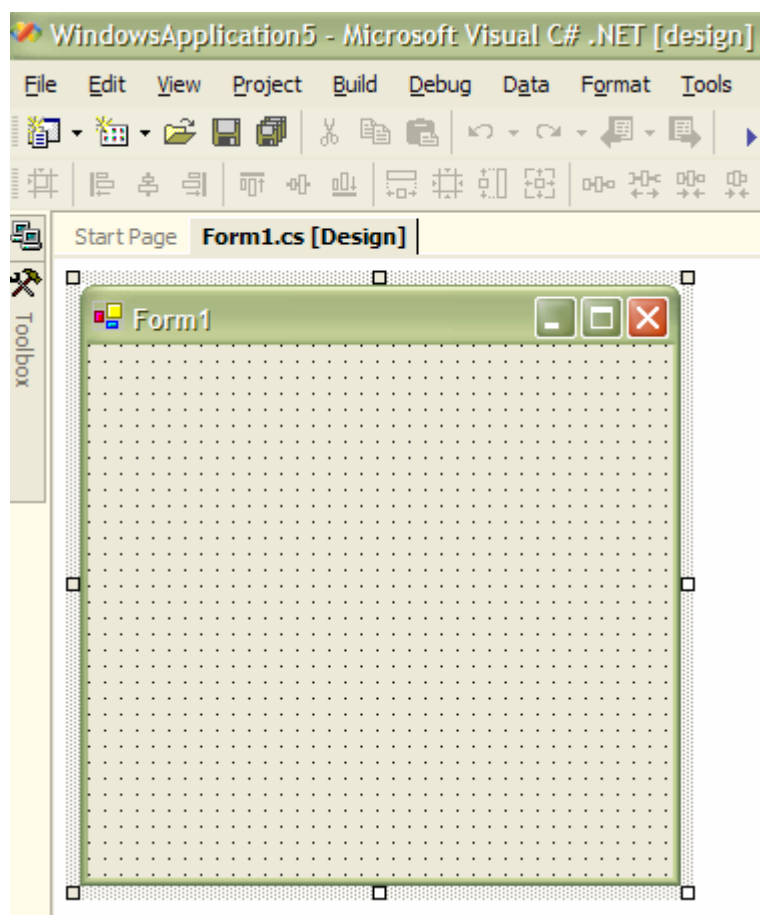
Új Projekt készítése

A Windows Application projektek készítése már a kezdeti lépésekben eltér a Console Application típusétól, mivel lényegesen több forráskód és erőforrás szükséges a készítésükhöz. Az ilyen típusú programoknak rendelkezniük kell egy ablakkal, az ablakot leíró osztállyal, s a forráskóddal, ami definiálja az előbbieket működését. Az alkalmazás fő része maga a Form, melyre a többi komponenst rakhatjuk. Amikor futtatjuk a programot, akkor is ez a Form lesz az, ami megjelenik a képernyőn, mint Windows alkalmazás.

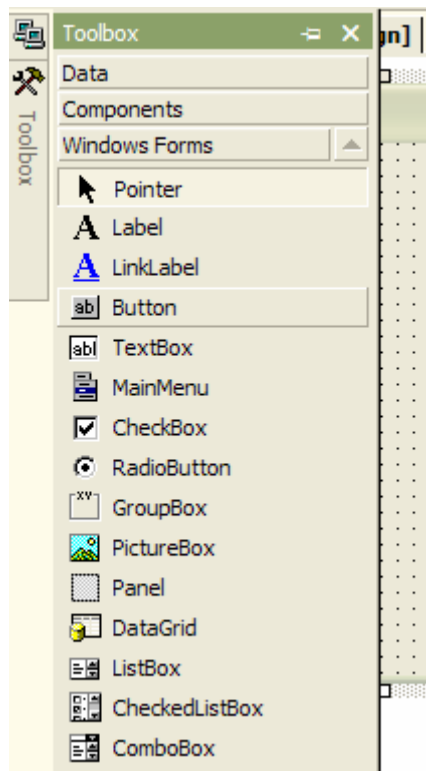
Készítsük el az első Form-mal rendelkező programunkat! Indítsuk el .NET fejlesztői eszközt! A *File* menü *new* menüpontjában található listából válasszuk a *new project*-et! Ezt megtehetjük úgy is, hogy a *Start Page* lapon, ami az indításkor megjelenik a szerkesztő részben, rákattintunk a *new project* linkre.

Ekkor elindul a New Project - varázsló, ahol ki kell választanunk azt a programozási nyelvet, mellyel dolgozni szeretnénk. Jelöljük ki a C# nyelvet! Az ablak jobb oldali részében meg kell mondanunk, hogy milyen típusú alkalmazást szeretnénk készíteni. Most ne a megszokott Console Application típust válasszuk, hanem a Windows Applicationt! Határozzuk meg a program nevét, majd azt a könyvtárat, ahova el akarjuk menteni a fájlokat! Érdekes külön könyvtárat készíteni minden programnak, hogy a fájlok ne keveredjenek össze.

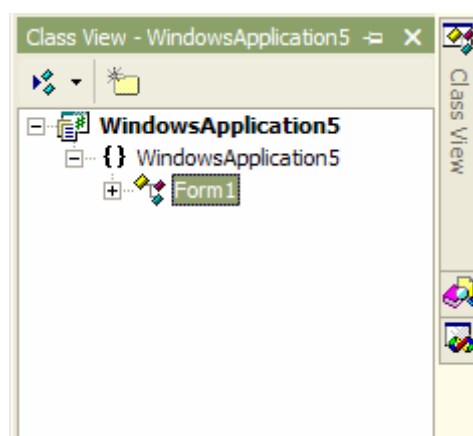
Az jóváhagyást követően a .NET dolgozni kezd. Generálja a programunkhoz szükséges fájlokat, előállítja a Form-ot, a kódszerkesztőbe betölti a forráskódot.



A forráskódot kezdetben nem is láthatjuk, csak a Form felületét a Design ablakban. Itt tudjuk szerkeszteni, és vezérlő elemeket is itt adhatunk hozzá. A Form-ra helyezhető komponenseket a bal oldali, Toolbox feliratú, előugró panelen találjuk meg. (A panel a többihez hasonlóan fixálható, de sajnos túl sokat takar a hasznos felületből. A vezérlők működését más fejezetek mutatják be.)



A .NET ablakának a jobb oldalán (nem alapértelmezés szerinti beállítások esetén máshol is lehet) találunk egy függőleges panelt, melynek a felirata Class View. A panelre kattintva tudjuk megmondani a szerkesztőnek, hogy mutassa meg a forráskódot. A panel a projekt elemeit fa struktúrába szervezve tartalmazza. Ha itt kiválasztjuk a Form1 címkét, és duplán kattintunk rá, a szerkesztőbe betöltődik a Form-hoz tartozó kód.



A program sorait tanulmányozva rögtön feltűnik az, hogy a using rész kibővült a következő hivatkozásokkal:

```
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
```

Az importált elemek közt helyet kapott a Windows form-okat leíró osztály, a komponensekhez és a grafikus felület programozásához szükséges hivatkozások. Amennyiben újabb komponenseket adunk a Form-hoz, a lista tovább bővíülhet.

A forráskód tartalmazza a saját névtérét is, melyet tetszés szerint átnevezhetünk.

```
namespace WindowsApplication1{...}
```

A névtérhez tartozó program részeket kapcsos zárójelei között helyezhetjük el. Itt foglal helyet a Form osztály definíciója, konstruktora, destruktora és az inicializálásához szükséges utasítások.

```
public class Form1 : System.Windows.Forms.Form
{
    private System.ComponentModel.Container components = null;

    public Form1()
    {
        InitializeComponent();
    }

    protected override void Dispose( bool disposing )
    {
        if( disposing )
        {
            if (components != null)
            {
                components.Dispose();
            }
        }
        base.Dispose( disposing );
    }

    #region Windows Form Designer generated code

    private void InitializeComponent()
    {
        this.components = new
System.ComponentModel.Container();
        this.Size = new System.Drawing.Size(300,300);
        this.Text = "Form1";
    }
    #endregion ...
}
```

A Main() függvényt a lista végén, de még a Class zárójelein belül definiálhatjuk, de ezt a .NET megteszi helyettünk. A Main() függvénynek jelenleg az egyetlen utasítása az Appliaction.Run();, melynek a feladata az alkalmazás elindítása.

```
static void Main()  
{  
    Application.Run(new Form1());  
}
```

A Form Designer és a Form1.cs, vagyis a forráskód ablakai közt válthatunk, ha rákattintunk egérrel a kívánt ablak fejlécére. Amennyiben a Form-on, vagy a ráhelyezett vezérlők valamelyikének a felületén duplát kattintunk, a kódszerkesztőbe ugrik a kurzor, de ebben az esetben az adott elem valamely eseményét kapjuk. A programozás során a két ablak közt folyamatosan váltogatni kell, mivel a Windows Application projektek készítése a forráskód és a felület együttes szerkesztését igényli.

A projekt futtatása az F5 billentyű leütésekor indul el. A .NET ellenőrzi a kód helyességét, ezután elindítja a Programot. (Természetesen a háttérben bonyolult műveletek hajtódnak végre, elindul az előfordítás, optimalizálás, stb.) A Form megjelenik a képernyőn a szerkesztő ablaka előtt. Ez a Form külön alkalmazásként fut, rendelkezik saját memória területtel, megjelenik a Windows Task- listájában. Ha leállítjuk, visszakérülünk a szerkesztőbe, s tovább dolgozhatunk a forráskódon.

A .NET kezelői felületén még számos panel található a felsoroltak mellett. A Toolbox felett foglal helyet a Server Explorer, mely lehetőséget nyújt a számítógép erőforrásainak (adatbázisok, DLL fájlok, különböző szerverek, service-ek) megtekintésére, tallózására.

A jobb oldalon találjuk Dinamikus Help rendszert, ami csak abban az esetben működik megfelelően, ha telepítettük a számítógépünkre az MSDN Library minden CD-jét, vagy rendelkezünk internet kapcsolattal. A Help rendszert az F1 billentyűvel is aktivizálhatjuk. Próbáljuk ki, hogyan működik! Gépeljük be a szerkesztőbe az int szót, jelöljük ki, majd nyomjuk le az F1-et. Egy új ablak nyílik, melyben az int típusról megtalálható, összes információ szerepelni fog: forráskód példák, leírások, praktikus dolgok. A Help használata elengedhetetlen a Windows programok írásakor, mivel a rendelkezésre álló osztályok és tagfüggvények száma olyan hatalmas, hogy senki nem tudja fejben tartani azokat.

Végül a képernyő alsó részén kaptak helyet a futtatási információkat és az esetleges hibák okát feltáró Output és Debug ablakok, melyeket a Consol Application projekteknel is megtalálhattunk. Ezeket be is zárhatjuk, mivel a következő futtatáskor úgyis újra megjelennek.

A Console Application projektek készítése bonyolultabb az eddig tanult alkalmazásokénál. Nagyobb programozói tudást igényelnek és feltételezik a magas szintű Objektum Orientált programozási ismereteket.

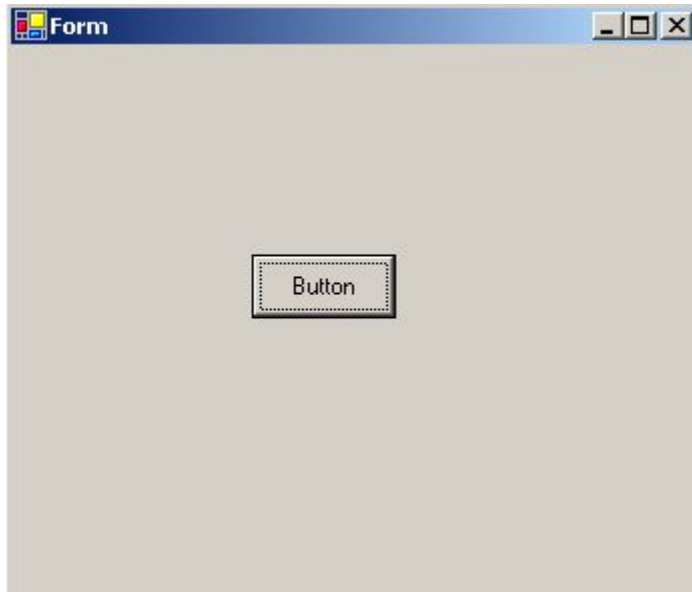
Feladatok

1. Hozzon létre egy új, Windows Application projektet!
2. Mentse el a háttértárra az aktuális alkalmazás minden adatát!
3. Keresse meg az alkalmazás mentett adatait, majd töltsse vissza azokat!
4. Futtassa az alkalmazást, és vizsgálja meg, hogy az Output és a Debug ablakban milyen üzenetek jelennek meg!
5. Próbálja meg értelmezni az üzeneteket!
6. A létrejött állományok közül keresse meg a form1.cs állományt majd nyissa meg azt a .NET fejlesztői eszközzel!
7. A Help rendszer segítségével keresse meg a form-okról rendelkezésre álló lehető legtöbb információt.

A látható komponensek, a kontrollok

Button (Gomb)

A nyomógomb komponens által a felhasználó műveletet hajthat végre. Például elindíthat, vagy megszakíthat egy folyamatot.



Properties (Tulajdonságok)

AllowDrop

Anchor

BackColor

BackgroundImage

ContextMenu

DialogResult

Dock

Enabled

Font

ForeColor

ImageList

ImageIndex

Location

Engedélyezi, vagy tiltja a drag and drop funkciót. Alapbeállításban tiltva van.

Form melyik részéhez igazítson

Gomb hátterének a színe

Gomb hátterének a képe

Jobbgombos előbukkanó menü

Beállítja a gomb visszatérési értékét.

Beállítja a gomb helyzetét a következő pozíciókba:

Top – felső részhez

Bottom – alsó részhez

Left – baloldalra

Right – jobboldalra

Fill – kitölti az egész Form-ot

Engedélyezett-e

A gombon található szöveg betűtípusa

A gombon található szöveg színe

Összerendelés a gomb és az imagelist között.

Az imagelistben megadott képnek az indexe

A gomb a bal felső saroktól való

| | |
|-------------------|---|
| Locked | távolsága x;y formában |
| Modifiers | Gomb lezárása |
| | A gomb hozzáférésének a lehetőségei: public, private, protected, protected internal, internal |
| Size | A gomb mérete |
| TabStop | Elérhető-e tabulátorral |
| TabIndex | A tabulátoros lépegetés sorrendje |
| Text | Gomb felirata |
| TextAlign | A felirat elhelyezkedése a gombon |
| Visible | Látható-e |
| Események: | |
| Click | Gombra kattintás |
| Enter | |
| GotFocus | Gomb fókuszbba kerülése |
| KeyPress, KeyDown | Billentyű lenyomása amíg a gomb fókuszban van |
| KeyUp | Billentyű elengedése amíg a gomb fókuszban van |
| Leave | Fókuszból kikerülés |
| MouseEnter | Mutató a gomb fölé helyezkedik el |
| MouseLeave | Mutató elhagyja a gombot |
| ParentChanged | Szülő megváltozik |

Label, Linklabel

A label szöveget jelenít meg, amihez a felhasználó nem fér hozzá.



Közös tulajdonságok:

Name

AllowDrop

Anchor

BackColor

BackgroundImage

Enabled

Font

ForeColor

ImageList

ImageIndex

Location

Size

Text

Visible

Név

Engedélyezi, vagy tiltja a drag and drop funkciót.

Form melyik részéhez igazítson

A címke hátterének a színe

A címke hátterének a képe

Engedélyezett-e

A címkén található szöveg betűtípusa

A szöveg színe

Összerendelés a címke és az imagelist között.

Az imagelistben megadott kép indexe

A gombnak a bal felső saroktól mért távolsága x;y formában

A címke mérete

A címke felirata

Látható-e

Linklabel tulajdonságai:

ActiveLinkColor

LinkBehavior

LinkColor

LinkVisible

VisitedLinkColor

Az aktivált link színe

A link stílusa amikor a mutató fölötte van

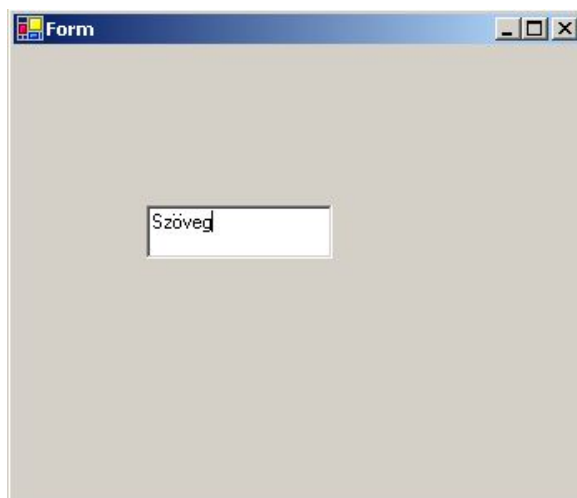
A link színe

Volt-e már megnézve?

Megnézett link színe

Textbox

A textbox egy olyan területet jelöl ki, ahol a felhasználó általában egysoros szöveget adhat meg, vagy módosíthat.



Tulajdonságai:

AutoSize
BorderStyle
CharacterCasing

Lines
MaxLength

Multiline
PasswordChar

Text

Események:

KeyPress, KeyDown

KeyUp

Leave
MouseEnter
MouseLeave

Automatikus méret

A textbox keretének a stílusa

A betű mérete

Lower – kisbetűs

Upper – nagybetűs

Normal – vegyes

Textboxban található sorok sorszámozva

A textboxba írható szöveg maximális hossza

Többsoros megjelenítés

Az textboxban megjelenő karakter (jelszavaknál)

Textboxban található szöveg

Billentyű lenyomása amíg a gomb fókuszbán van

Billentyű elengedése amíg a gomb fókuszbán van

Fókuszról kikerülés

Mutató a gomb fölé helyezkedik el

Mutató elhagyja a gombot

CheckBox

Akkor használjuk, amikor egy igaz/hamis, igen/nem választási lehetőséget szeretnénk adni a felhasználó számára.

A CheckBoxnak és a RadioButtonnak a funkciója ugyanaz. Mindkettő lehetővé teszi a felhasználó számára, hogy egy listából válasszon. De amíg a CheckBoxnál a lista elemeinek kombinációját választhatjuk ki, addig a RadioButtonnál már csak a lista egy elemét.

Az Appearance (megjelenés) tulajdonságnál azt állíthatjuk be, hogy a CheckBox egy általános kiválasztó négyzet legyen vagy egy nyomógomb.

A ThreeState tulajdonság azt határozza meg, hogy a CheckBoxnak két vagy három állapota legyen. Ha kettő akkor a CheckBox aktuális értékét lekérdezni és beállítani a Checked tulajdonságnál tudjuk. Ha három akkor mindezt a CheckState tulajdonságnál tudjuk megtenni.

Megjegyzés: A Checked tulajdonság 3 állapotú CheckBox esetén mindig igaz értéket ad vissza

A FlatStyle a CheckBox stílusát és Megjelenését határozza meg. Ha ez FlatStyle.Systemre van állítva, akkor a felhasználó operációs rendszerének beállításai határozza meg ezt.

Megjegyzés: Mikor a FlatStyle tulajdonság FlatStyle.Systemre van állítva, akkor CheckAlign tulajdonság figyelmen kívül van hagyva, és a ContentAlignment.MiddleLeft vagy ContentAlignment.MiddleRight igazítást figyelembe véve jelenik meg. . Ha a CheckAlign tulajdonság az egyik jobb igazításra van állítva akkor ez a vezérlő elem úgy jelenik meg, hogy a

ContentAlignment.MiddleRight igazítást használja; különben pedig a ContentAlignment.MiddleLeft igazítás állítja be.

Példa kód:

```
// CheckBox létrehozása és inicializálása
CheckBox checkBox1 = new CheckBox();

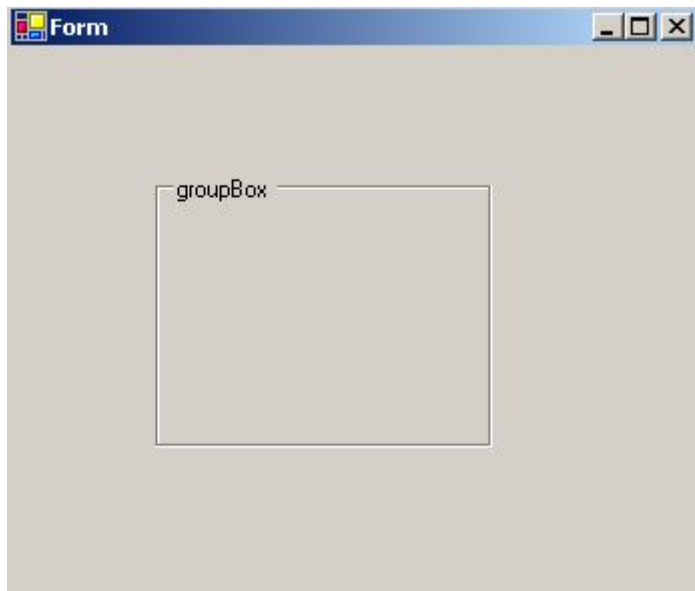
//CheckBox megjelenítése nyomógombként
checkBox1.Appearance = Appearance.Button;

// A megjelenés automatikus frissítésének kikapcsolás
checkBox1.AutoCheck = false;

// A CheckBox hozzáadása a formhoz.
Controls.Add(checkBox1);
```

GroupBox

A GroupBox egy terület keretét határozza meg egy vezérlő elem csoport körüli felirattal vagy felirat nélkül. Logikailag egységbe tartozó vezérlő elemek összefogására használjuk a formon. A GroupBox egy konténer, ami vezérlő csoportokat tud meghatározni.



A GroupBoxot jellemzően RadioButton csoportok összefogására használjuk. Ha van két GroupBoxunk, mindkettőben néhány RadioButton, akkor mindkettőben ki tudunk választani egyet-egyet.

A GroupBoxhoz a Control tulajdonság Add metódusával tudunk hozzáadni vezérlőket.

Példa kód:

```
private void InitializeMyGroupBox()
{
    //Egy GroupBox és 2 RadioButton létrehozása és inicializálása.
    GroupBox groupBox1 = new GroupBox();
    RadioButton radioButton1 = new RadioButton();
    RadioButton radioButton2 = new RadioButton();

    // A GroupBox FlatStyle tulajdonságának beállítása.
    groupBox1.FlatStyle = FlatStyle.System;

    // A RadioButtons hozzáadása a GroupBoxhoz.
    groupBox1.Controls.Add(radioButton1);
    groupBox1.Controls.Add(radioButton2);

    // A GroupBox hozzáadása a formhoz
    Controls.Add(groupBox1);
}
```

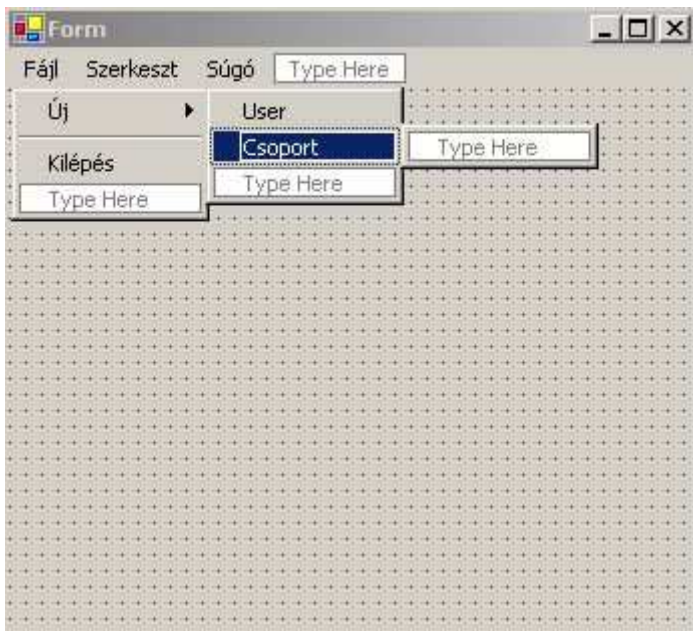
MainMenu

A MainMenu egy konténer a form menüstruktúrájának számőára. A MainMenu MenuItem objektumokból áll össze melyek egyedi menüutasítások a menü struktúrában. Minden menüitemhez tartozik egy utasítás amit végrehajt az alkalmazásunkban vagy az al, vagy az ősz menüelemeken. Ahhoz, hogy beillesszük a MainMenu-t a formba és az megjelenjen rajta össze kell egyeztetni a form Menu tulajdonságával.



Azokhoz az alkalmazásokhoz melyek több nyelvet támogatnak lehet használni a RightToLeft tulajdonságot, ami a menüelem szövegét jobbról balra jeleníti meg, mint az arab nyelv írásakor.

Egy formhoz lehet készíteni több MainMenut is ha több menüstruktúrát szeretnénk megvalósítani. Ha újra akarjuk használni a MinMenu egy speciális menüstruktúrában, akkor használjuk a CloneMenu metódust, ami egy másolatot készít. Amikor kész van a másolat, akkor el lehet végezni a megfelelő módosításokat az új menüstruktúrán.

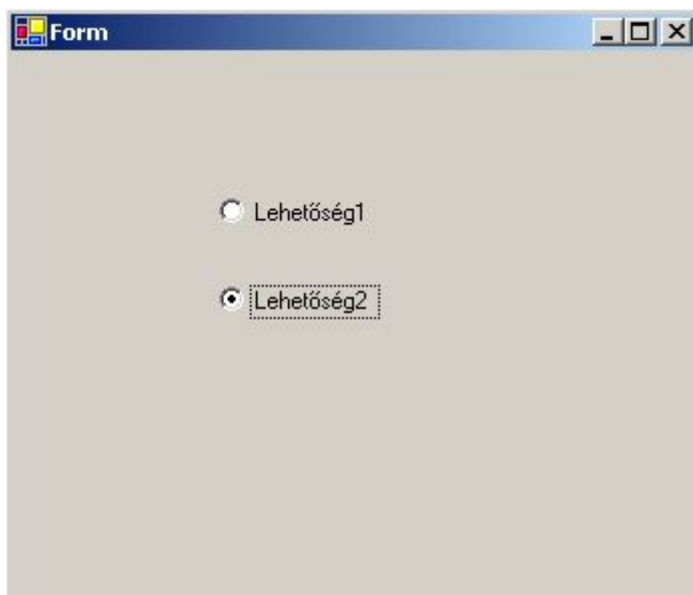


Példa kód:

```
public void CreateMyMainMenu()  
{  
    //Üres MainMenu létrehozása.  
    MainMenu mainMenu1 = new MainMenu();  
  
    MenuItem menuItem1 = new MenuItem();  
    MenuItem menuItem2 = new MenuItem();  
  
    menuItem1.Text = "File";  
    menuItem2.Text = "Edit";  
    // 2 MenuItem hozzáadása a MainMenuhoz.  
    mainMenu1.MenuItems.Add(menuItem1);  
    mainMenu1.MenuItems.Add(menuItem2);  
  
    // A MainMenu beillesztése Formlbe.  
    Menu = mainMenu1;  
}
```

RadioButton

A CheckBox és a RadioButton a funkciója ugyanaz. Mindkettő lehetővé teszi a felhasználó számára, hogy egy listából válasszon. De amíg a CheckBoxnál a lista elemeinek kombinációját választhatjuk ki, addig a RadioButtonnál már csak a lista egy elemét.



A RadioButton meg tud jeleníteni egy szöveget, vagy egy képet, vagy mindkettőt.

Amikor a felhasználó kiválaszt egy RadioButtont egy csoportból akkor a többi automatikusan üresre változik. Minden RadioButton ami egy adott konténeren belül van (mint pl. egy form) egy csoportot alkot. Ha egy formra több csoportot akarunk rakni, akkor minden egyes csoportnak helyezünk el egy konténert a formon (mint pl. GroupBox, Panel).

A Checked tulajdonsággal tudjuk lekérdezni és beállítani a RadioButton állapotát. A RadioButton úgy nézhet ki mint egy nyomógomb, vagy úgy mint egy hagyományos RadioButton. Ezt az Appearance tulajdonság határozza meg.

Példa kód:

```
private void InitializeMyRadioButton()
{
    // RadioButton létrehozása és inicializálása.
    RadioButton radioButton1 = new RadioButton();

    // RadioButton nyomógombként való megjelenítése.
    radioButton1.Appearance = Appearance.Button;

    // A Click esemény hatására történő kinézet frissítés
    // kapcsolás.
    radioButton1.AutoCheck = false;

    // A RadioButton hozzáadása a formhoz
    Controls.Add(radioButton1);
}
```

ComboBox

A listák nagy helyet foglalnak a formokon, és az általuk felkínált elemek nem bővíthetők a felhasználó által. Ezen problémák megoldására használhatjuk a **combobox** osztályt. Ez egyesíti a szerkesztő mező (edit) és a lenyíló lista tulajdonságait. Első pillantásra egy TextBox-ot láthatunk a jobb oldalán egy nyílacskával.



Feladata: Adatok legördülő ablakban történő megjelenítése

Megjelenés

| | |
|---------------|--|
| BackColor | a ComboBox háttérszíne |
| Cursor | a kurzor típusa, ami megjelenik a vezérlő fölött, amikor az egérkurzort fölé mozgatjuk |
| DropDownStyle | a megjelenést és a működést vezérli |
| Font | szöveg megjelenítésére használt betűtípus |
| ForeColor | előtérszín (pl. betűszín) |
| Text | a vezérlőben látható szöveg |

Viselkedés

| | |
|------------------|---|
| AllowDrop | meghatározza, hogy a vezérlő fogadhat-e „Fogd-és-vidd” értesítéseket |
| ContextMenu | helyi menü, ami a vezérlőn jobb egérgombbal történő kattintásra jelenik meg |
| DrawMode | A ComboBox megrajzolási módját vezérli |
| DropDownWidth | a legördülő ablak szélessége pixelben |
| Enabled | azt jelzi, hogy a ComboBox engedélyezett, vagy nem |
| IntegralHeight | true, ha a listarészlet csak teljes elemeket tartalmazhat, különben false |
| ItemHeight | a ComboBox egy elemének magassága pixelben |
| MaxDropDownItems | A legördülő listában egyszerre látható bejegyzések maximális száma |
| MaxLength | a ComboBox beviteli részébe írható karakterek maximális száma |
| Sorted | Meghatározza, hogy a vezérlő tartalma |

| | |
|----------|--|
| TabIndex | rendezett vagy sem az elem helyét adja meg a TAB sorrendben |
| TabStop | megmutatja, hogy az elem kiválasztható-e a TAB billentyű használatával |
| Visible | a vezérlő látható vagy nem |

Adat

| | |
|---------------|--|
| DataSource | a listát jelöli, ahonnan a vezérlő az elemeit veszi |
| DisplayMember | az adatforrás egy tulajdonsága, mezője, amit a comboboxban meg kívánunk jeleníteni |
| Items | Collection, a ComboBox elemeit tartalmazza |
| Tag | Tetszőleges célokra használható egész értékű mező |

Tervezés

| | |
|---------------|--|
| Name | A vezérlő neve |
| Locked | megmutatja, hogy a vezérlő átméretezhető, átmozgatható-e |
| Modifiers | A vezérlő láthatósági szintjét jelöli |
| Anchor | Horgony; a vezérlő mely szélei rögzítettek az őt tartalmazó konténer széléihez képest |
| Dock | megmutatja, hogy a vezérlő mely szélei vannak összekapcsolva az őt tartalmazó elem szélével |
| Location | Beállítja vagy lekérdezi a vezérlő bal felső sarkának az őt tartalmazó elem bal felső sarkától mért relatív távolságát |
| SelectedValue | Kiválasztott elem értéke |

Események

| | |
|----------------------|--|
| Click | kattintáshoz kötődő esemény |
| DoubleClick | Dupla kattintáshoz kapcsolódó esemény |
| DrawItem | akkor következik be, amikor egy bizonyos elemet vagy területet meg kell rajzolni |
| DropDown | azt jelzi, hogy a ComboBox menüje legördült |
| DropDownStyleChanged | jelzi, hogy a DropDownStyle tulajdonság megváltozott |
| HelpRequested | A felhasználó segítséget kér a vezérlőről |
| MeasureItem | akkor következik be, amikor egy |

| | |
|----------------------|--|
| | bizonyos elem magasságát ki kell számítani |
| SelectedIndexChanged | akkor következik be, amikor a ComboBox 'SelectedIndex' tulajdonsága megváltozik, azaz újabb elem kerül kijelölésre |
| StyleChanged | jelzi, ha megváltozott a vezérlő stílusa |
| SystemColorsChanged | bekövetkezik, amikor a rendszerszínek megváltoznak |

A következő események mindegyike egy tulajdonság megváltozását jelzi:

| | |
|----------------------|---|
| BackColorChanged | háttérszín |
| ContextMenuChanged | helyzetérzékeny menü |
| CursorChanged | kurzor |
| DataSourceChanged | adatforrás |
| DisplayMemberChanged | megjelenítendő adattag-forrás |
| DockChanged | igazítás |
| EnabledChanged | engedélyezettségi állapot |
| FontChanged | betűtípus |
| ForeColorChanged | előtérszín (betűszín) |
| LocationChanged | helyzet (Lásd Location) |
| ParentChanged | szülő |
| SelectedValueChanged | kiválasztott érték |
| SizeChanged | méret |
| TabIndexChanged | tab-sorrendbeli hely |
| TabStopChanged | TAB-bal történő kiválaszthatóság beállítása |
| TextChanged | szöveg |
| VisibleChanged | vizuális láthatóság |

A kritikus tulajdonság a **DropDownStyle** lehetséges értékei:

| | |
|--------------|---|
| Simple | Szerkeszthető mező, a lista mindig látszik |
| DropDown | Szerkeszthető mező, a lista lenyitható. (alapértelmezett) |
| DropDownList | Nem szerkeszthető a mező, a lista lenyitható |

A listához hasonlóan a comboboxban is az **Items** tulajdonság tárolja a lista elemeit. Ez a tulajdonság egy **ObjectCollection** típus. Így kezelése a szokásos metódusok használatával lehetséges.

| | |
|--------|---------------------------------------|
| Count | Indexer, az elemek számát adja vissza |
| Add | Új elem felvétele a listához |
| Insert | Új elem beszúrása a listába |
| Remove | Elem törlése |

A felhasználó által kiválasztott elemet a **SelectedItem** tulajdonságon keresztül érjük el, ami egy objektumot ad vissza. Ha az indexére van szükségünk, akkor a **SelectedIndex** tulajdonságot használjuk.

Tekintsünk néhány alpműveletet a példa segítségével:

A 'Feltöltés' gombeseménykezelője egyszerű, egész számokkal tölti fel a combobox items tulajdonságát:

```
for (int i=0;i<10;i++)
    comboBox1.Items.Add(i.ToString());
```

A 'Törlés' gomb mind a Items tárolót, mind a text mezőt törli:

```
comboBox1.Items.Clear();
comboBox1.Text="";
```

Egy egyszerű switch szerkezettel módosíthatjuk a stílusát a comboboxnak:

```
switch (comboBox1.DropDownStyle)
{
    case ComboBoxStyle.Simple: comboBox1.DropDownStyle =
        ComboBoxStyle.DropDown;
        button3.Text="DropDown";
        break;
    case ComboBoxStyle.DropDown: comboBox1.DropDownStyle =
        ComboBoxStyle.DropDownList;
        button3.Text="DropDownList";
        break;
    case ComboBoxStyle.DropDownList: comboBox1.DropDownStyle =
        ComboBoxStyle.Simple;
        button3.Text="Simple";
        break;
}
```

ListView

Ha az előzőeknél is kifinomultabb listát szeretnénk használni, akkor erre lehetőséget ad a ListView osztály.

Feladata: elemek gyűjteményének –különböző nézetekben történő- megjelenítése

Megjelenés

| | |
|-------------|--|
| BackColor | a ListView háttérszíne |
| BorderStyle | A keret stílusa |
| CheckBoxes | azt mutatja, hogy megjelennek-e CheckBoxok az elemek |

| | |
|--------------------|---|
| | mellett |
| Cursor | a kurzor típusa, ami megjelenik a vezérlő fölött, amikor az egérkurzort fölé mozgatjuk |
| Font | szöveg megjelenítésére használt betűtípus |
| ForeColor | előtérszín (pl. betűszín) |
| FullRowSelect | Megmutatja, hogy egy elemen történő kattintás kijelöli-e az elem összes al-elemét is |
| GridLines | rácsvonalak jelennek meg az elemek és a részelemek körül. |
| View | az elemek megjelenítési módja (ikonok, részletek, lista...) |
| Viselkedés | |
| Activation | meghatározza, hogy milyen típusú műveletre van szükség a felhasználó részéről egy elem aktiválásához |
| Alignment | megmutatja az elemek igazítási módját a ListView-n belül |
| AllowColumnReorder | jelzi, hogy a felhasználó megváltoztathatja-e az oszlopok sorrendjét |
| AllowDrop | jelzi, hogy a vezérlő fogadhat-e „Fogd-és-vidd” értesítéseket |
| AutoArrange | az ikonok automatikus rendezettségét jellemzi |
| ContextMenu | helyi menü, ami a vezérlőn jobb egérgombbal történő kattintásra jelenik meg |
| Columns | A vezérlőben megjelenő oszlopfejlécek, Collection |
| Enabled | azt jelzi, hogy a combobox engedélyezett, vagy nem |
| HeaderStyle | Oszlopfejlécek stílusa |
| HideSelection | azt jelöli, hogy a vezérlő kijelölt elemén megmarad-e a kijelölés, amikor a vezérlő elveszíti a fókust |
| HoverSelection | megmutatja, hogy kijelölhető-e egy elem azáltal, hogy az egérkurzort fölötte hagyjuk |
| ImeMode | |
| Items | A ListView elemei |
| LabelEdit | megengedi a felhasználónak, hogy az elemek címkéit megváltoztassák |
| LabelWrap | azt jelöli, hogy a címke szövege több sorra törhető-e |
| LargeImageList | a lista ikonjai Nagy ikonok nézetben |
| MultiSelect | engedélyezi több elem egyszerre történő kijelölését |
| Scrollable | Meghatározza, hogy a vezérlőben megjelenhetnek-e gördítősávok, amennyiben nincs elég hely az ikonok számára |
| SmallImageList | a lista ikonjai Kis ikonok nézetben |
| Sorting | elemek rendezésének módja |
| StateImageList | a lista alkalmazás által meghatározott állapotokkal kapcsolatos ImageList-je |
| TabIndex | az elem helyét adja meg a TAB sorrendben |
| TabStop | megmutatja, hogy az elem kiválasztható-e a TAB billentyű használatával |
| Visible | a vezérlő látható vagy nem |
| Adat | |
| Tag | Teszőleges célokra használható egész értékű mező |

Tervezés

| | |
|-----------|--|
| Name | A vezérlő neve |
| Locked | megmutatja, hogy a vezérlő átméretezhető, átmozgatható-e |
| Modifiers | A vezérlő láthatósági szintjét jelöli |
| Anchor | Horgony; a vezérlő mely szélei rögzítettek az őt tartalmazó konténer széleihez képest |
| Dock | megmutatja, hogy a vezérlő mely szélei vannak összekapcsolva az őt tartalmazó elem szélével |
| Location | Beállítja vagy lekérdezi a vezérlő bal felső sarkának az őt tartalmazó elem bal felső sarkától mért relatív távolságát |

Események

| | |
|----------------------|--|
| Click | kattintáshoz kötődő esemény |
| ColumnClick | oszlopfejlécre történő kattintáshoz kötődő esemény |
| DoubleClick | Dupla kattintáshoz kapcsolódó esemény |
| ItemActivate | Elem aktiválása |
| ItemDrag | akkor következik be, amikor a felhasználó elkezd „vonszolni” egy elemet |
| AfterLabelEdit | elemcímke módosítása után következik be |
| AfterLabelEdit | elemcímke módosítása előtt jelentkezik |
| HelpRequested | A felhasználó segítséget kér a vezérlőről |
| ItemCheck | egy elem „check” állapotának megváltozásához tartozik |
| SelectedIndexChanged | akkor következik be, amikor a ComboBox 'SelectedIndex' tulajdonsága megváltozik, azaz újabb elem kerül kijelölésre |
| StyleChanged | jelzi, ha megváltozott a vezérlő stílusa |
| SystemColorsChanged | bekövetkezik, amikor a rendszerszínek megváltoznak |

A következő események mindegyike egy tulajdonság megváltozását jelzik:

| | |
|--------------------|----------------------------------|
| BackColorChanged | háttérszín |
| ContextMenuChanged | helyzetérzékeny menü |
| CursorChanged | kurzor |
| DockChanged | igazítás |
| EnabledChanged | engedélyezettségi állapot |
| FontChanged | betűtípus |
| ForeColorChanged | előtérszín (betűszín) |
| LocationChanged | helyzet (Lásd Location) |
| ParentChanged | szülő |
| SizeChanged | méret |
| TabIndexChanged | tab-sorrendbeli hely |
| TabStopChanged | TAB-bal történő kiválaszthatóság |
| VisibleChanged | vizuális láthatóság |

TreeView

Ez az osztály az elemek hierarchikus rendben történő megjelenését szolgálja.

Feladata: címkézett elemek hierarchikus gyűjteményének megjelenítése

Megjelenés



| | |
|-------------|--|
| BackColor | a ComboBox háttérszíne |
| BorderStyle | A keret stílusa |
| CheckBoxes | azt mutatja, hogy megjelennek-e CheckBoxok az elemek mellett |
| Cursor | a kurzor típusa, ami megjelenik a vezérlő fölött, amikor az egérkurzort fölé moztatjuk |
| Font | szöveg megjelenítésére használt betűtípus |
| ForeColor | előtérszín (pl. betűszín) |
| ItemHeight | a TreeView egy elemének magassága pixelben |

Viselkedés

| | |
|---------------|---|
| AllowDrop | meghatározza, hogy a vezérlő fogadhat-e „Fogd-és-vidd” értesítéseket |
| ContextMenu | helyi menü, ami a vezérlőn jobb egérgombbal történő kattintásra jelenik meg |
| Enabled | azt jelzi, hogy a combobox engedélyezett, vagy nem |
| FullRowSelect | Megmutatja, hogy egy elemen történő kattintás kijelöli-e az elem összes al-elemét is |
| HideSelection | azt jelöli, hogy a vezérlő kijelölt elemén megmarad-e a kijelölés, amikor a vezérlő elveszíti a fókuszt |
| HotTracking | megmutatja, hogy az elemek hyperlink-stílusúvá válnak-e, amikor az egérmutató föléjük ér |
| ImageIndex | az alapértelmezett képindex a csomópontok számára |
| ImageList | a vezérlő ImageList-je, amelyből a csomópontokhoz tartozó képek származnak |

| | |
|--------------------|---|
| Indent | a gyermekcsomópontok behúzása pixelben |
| LabelEdit | megengedi a felhasználónak, hogy az elemek címkeit megváltoztassa |
| Nodes | Gyökércsomópontok a TreeView-n belül |
| PathSeparator | a csomópontok teljes elérési útvonalának megadásához használt elválasztó sztring |
| Scrollable | Meghatározza, hogy a vezérlőben megjelenhetnek-e gördítősávok, amennyiben nincs elég hely az elemek megjelenítése számára |
| SelectedImageIndex | alapértelmezett képindex a kiválasztott csomópontok számára |
| ShowLines | csomópontokat összekötő vonalak megjelenítése |
| ShowPlusMinus | plusz/mínusz gombok megjelenítése a szülőcsomópontok mellett |
| ShowRootLines | vonalak megjelenítése a szülőcsomópontok mellett |
| Sorted | Meghatározza, hogy a vezérlő tartalma rendezett vagy sem |
| TabIndex | az elem helyét adja meg a TAB sorrendben |
| TabStop | megmutatja, hogy az elem kiválasztható-e a TAB billentyű használatával |
| Visible | a vezérlő látható vagy nem |

Adat

Tag: Tesztölekes célokra használható egész értékű mező

Tervezés

| | |
|-----------|--|
| Name | A vezérlő neve |
| Locked | megmutatja, hogy a vezérlő átméretezhető, átmozgatható-e |
| Modifiers | A vezérlő láthatósági szintjét jelöli |
| Anchor | Horgony; a vezérlő mely szélei rögzítettek az őt tartalmazó konténer széleihez képest |
| Dock | megmutatja, hogy a vezérlő mely szélei vannak összekapcsolva az őt tartalmazó elem szélével |
| Location | Beállítja vagy lekérdezi a vezérlő bal felső sarkának az őt tartalmazó elem bal felső sarkától mért relatív távolságát |
| Size | A vezérlő mérete pixelben |

Események

| | |
|----------------|---|
| Click | kattintáshoz kötődő esemény |
| DoubleClick | Dupla kattintáshoz kapcsolódó esemény |
| ItemDrag | akkor következik be, amikor a felhasználó elkezd „vonszolni” egy elemet |
| AfterCheck | akkor következik be, amikor a TreeNode egy CheckBox-ának értéke megváltozik |
| AfterCollapse | Lista felgördítése után következik be |
| AfterExpand | Lista legördítése után következik be |
| AfterLabelEdit | elemcímke módosítása után következik be |
| AfterSelect | akkor váltódik ki, amikor a kijelölés megváltozik |

| | |
|---------------------|---|
| BeforeCheck | kiváltódik, mielőtt a TreeNode CheckBox kijelölésre kerül |
| BeforeCollapse | Lista felgördítése után következik be |
| BeforeExpand | Lista legördítése után következik be |
| BeforeLabelEdit | elemcímke módosítása előtt jelentkezik |
| BeforeSelect | a TreeNode kijelölse előtt váltódik ki |
| HelpRequested | A felhasználó segítséget kér a vezérlőről |
| StyleChanged | jelzi, ha megváltozott a vezérlő stílusa |
| SystemColorsChanged | bekövetkezik, amikor a rendszerszínek megváltoznak |

A következő események mindegyike egy tulajdonság megváltozását jelzik:

| | |
|--------------------|----------------------------------|
| BackColorChanged | háttérszín |
| ContextMenuChanged | helyzetérzékeny menü |
| CursorChanged | kurzor |
| DockChanged | igazítás |
| EnabledChanged | engedélyezettségi állapot |
| FontChanged | betűtípus |
| ForeColorChanged | előtér szín (betűszín) |
| LocationChanged | helyzet (Lásd Location) |
| ParentChanged | szülő |
| SizeChanged | méret |
| TabIndexChanged | tab-sorrendbeli hely |
| TabStopChanged | TAB-bal történő kiválaszthatóság |
| VisibleChanged | vizuális láthatóság |

TabControl

Feladata: lapok összefüggő halmazát alkotja

Megjelenés



| | |
|-------------|--|
| Cursor | a kurzor típusa, ami megjelenik a vezérlő fölött, amikor az egérkurzort fölé mozgatjuk |
| Font | szöveg megjelenítésére használt betűtípus |
| ImageList | a TabControlhoz |
| Megjelenés | |
| Alignment | megmutatja a fülek hol helyezkednek el a TabControlon belül |
| AllowDrop | jelzi, hogy a vezérlő fogadhat-e „Fogd-és-vidd” értesítéseket |
| Appearance | megjelenési beállítások |
| ContextMenu | helyi menü, ami a vezérlőn jobb egérgombbal történő kattintásra jelenik meg |
| DrawMode | A TabControl megrajzolási módját vezérli |
| Enabled | azt jelzi, hogy a TabControl engedélyezett, vagy nem |
| HotTrack | megmutatja, hogy a fejlécek vizuálisan megváltoznak-e, amikor az egérmutató föléjük ér |

| | |
|--------------|---|
| ItemSize | Meghatározza a vezérlő al-ablakainak méretét |
| MultiLine | Meghatározza, hogy csak egy, vagy vagy több fül is lehet a vezérlőn belül |
| Padding | meghatározza, hogy ,mennyi extra hely legyen a vezérlő „fülei” körül |
| ShowToolTips | Meghatározza, hogy látszódnak-e a lapokhoz tartozó ToolTip-ek. |
| SizeMode | az egyes lapok méretezési módját állítja be |
| TabIndex | az elem helyét adja meg a TAB sorrendben |
| TabStop | megmutatja, hogy az elem kiválasztható-e a TAB billentyű használatával |
| Visible | a vezérlő látható vagy nem |

Adat

Tag: Teszőleges célokra használható egész értékű mező

Tervezés

| | |
|------------|--|
| Name | A vezérlő neve |
| DrawGrid | megmutatja, hogy a pozícionáló rács kirajzolásra kerüljön-e |
| GridSize | meghatározza a pozícionáló rács méretét |
| Locked | megmutatja, hogy a vezérlő átméretezhető, átmozgatható-e |
| Modifiers | A vezérlő láthatósági szintjét jelöli |
| SnapToGrid | Meghatározza, hogy a vezérlőknek kapcsolódnia kell-e a pozícionáló rács |
| Anchor | Horgony; a vezérlő mely szélei rögzítettek az őt tartalmazó konténer széleihez képest |
| Dock | megmutatja, hogy a vezérlő mely szélei vannak összekapcsolva az őt tartalmazó elem szélével |
| Location | Beállítja vagy lekérdezi a vezérlő bal felső sarkának az őt tartalmazó elem bal felső sarkától mért relatív távolságát |
| Size | A vezérlő mérete pixelben |
| TabPage | A lapok a TabControlban |

Események

| | |
|-------------|---------------------------------------|
| Click | kattintáshoz kötődő esemény |
| DoubleClick | Dupla kattintáshoz kapcsolódó esemény |

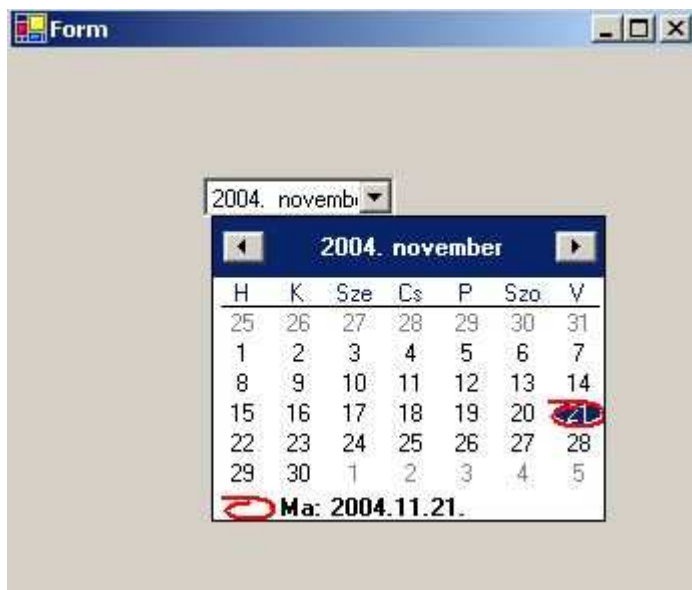
| | |
|----------------------|--|
| DrawItem | akkor következik be, amikor egy bizonyos elemet vagy területet meg kell rajzolni |
| HelpRequested | A felhasználó segítséget kér a vezérlőről |
| SelectedIndexChanged | akkor következik be, amikor a ComboBox 'SelectedIndex' tulajdonsága megváltozik, azaz újabb elem kerül kijelölésre |
| StyleChanged | jelzi, ha megváltozott a vezérlő stílusa |
| SystemColorsChanged | bekövetkezik, amikor a rendszerszínek megváltoznak |

A következő események mindegyike egy tulajdonság megváltozását jelzik:

| | |
|--------------------|----------------------------------|
| BackColorChanged | háttérszín |
| ContextMenuChanged | helyzetérzékeny menü |
| CursorChanged | kurzor |
| DockChanged | igazítás |
| EnabledChanged | engedélyezettségi állapot |
| FontChanged | betűtípus |
| LocationChanged | helyzet (Lásd Location) |
| ParentChanged | szülő |
| SizeChanged | méret |
| TabIndexChanged | tab-sorrendbeli hely |
| TabStopChanged | TAB-bal történő kiválaszthatóság |
| VisibleChanged | vizuális láthatóság |

DateTimePicker komponens

A DateTimePicker komponens segítségével egyszerűen kérhetünk be vagy írhatunk ki dátumot, időt. Két részből áll: egy szövegbeviteli doboz, mely a dátumot/időt tartalmazza szöveges formában, és egy lenyíló naptár, mely megegyezik a MonthCalendar komponens kinézetével és használatával.



A lenyíló naptár helyett használhatjuk a komponenst görgetőkkel is (a lenyitó gomb helyett fel-le gombok jelennek meg) a ShowUpDown tulajdonság igazra állításával. Beállíthatunk két dátumot (minimum és maximum), melyek határt szabhatnak a bevitelre, ezeknél kisebb vagy nagyobb dátumot nem lehet beírni.

A kívánt értékeket négyféle formátumban képes a komponens megjeleníteni:

Hosszú dátum (év, hónap neve, nap)

Rövid dátum (év, hónap száma, nap)

Idő (óra, perc, másodperc)

Egyéni (custom)

Tulajdonságok:

| | |
|---------------------------|--|
| CalendarFont | A megjelenő naptár betűtípusa |
| CalendarForeColor | ~ betűszíne |
| CalendarMonthBackground | ~ háttérszíne |
| CalendarTitleBackColor | ~ címsáv háttérszín |
| CalendarTitleForeColor | ~ címsáv betűszín |
| CalendarTrailingForeColor | ~ megelőző és követő hónap napjainak színe |
| Checked | Ha a ShowCheckBox tulajdonság értéke true, ellenőrizhető, hogy a felhasználó választott-e értéket False |
| CustomFormat | Egyéni formátum-string dátum és/vagy idő kiírásához |
| DropDownAlign | Beállítja, hogy a lenyíló hónap-naptár a komponenshez hogyan legyen igazítva Left vagy Right |
| Format | A komponensben szereplő dátum/idő formátuma. Választhatunk a beépített formátumok közül vagy lehet egyedi Long/Short/Time/Custom |
| MaxDate | A legkésőbbi beírható dátum |
| MinDate | A legkorábbi beírható dátum |
| ShowCheckBox | Meghatározza, hogy szerepeljen-e egy jelölőnégyzet a komponensen. Amíg a jelölőnégyzet nincs |

| | |
|------------|---|
| | kijelölve, addig nem volt érték kiválasztva. False |
| ShowUpDown | A lenyíló naptár helyett a kiválasztott dátumot/időt egy fel-le görgetővel változtathatjuk (spinner) False |
| Value | Az aktuális dátum/idő |

Események:

| | |
|--------------|---|
| CloseUp | Bekövetkezik, ha a felhasználó a lenyíló naptárból kiválasztott egy dátumot |
| DropDown | Bekövetkezik, ha a naptár lenyílik |
| ValueChanged | Bekövetkezik, ha a komponens értéke megváltozik |

Érték beírása és kiolvasása

A komponensben kiválasztott értéket a Value tulajdonság tartalmazza. Alapértelmezésben ez az érték az aktuális dátumot/időt tartalmazza. Az érték beállítható szerkesztéskor vagy futásidőben is, pl. a komponens tartalmazó form megjelenítésekor.

A komponens egy DateTime értéket ad vissza, vagy kaphat.

Tulajdonságok dátumban:

Year (év), Month (hónap), Day (nap) tulajdonságok egész számokat adnak vissza.

DayOfWeek tulajdonság a hét egy napjának nevét adja vissza (Monday, ... , Sunday)

Tulajdonságok időben:

Hour (óra), Minute (perc), Second (másodperc) és Millisecond (ezredmásodperc) tulajdonságok egész számot adnak vissza.

Értékadás:

```
DateTimePicker1.Value = new DateTime(2004, 9, 16);
```

Érték kiolvasása:

```
this.lblDatum.Text = DateTimePicker1.Value.ToString();
```

Egyéni formátum-string:

Állítsuk a Format tulajdonságot Custom-ra. A formátum-stringben az alábbiak használhatók:

| | |
|------|---|
| D | Egy vagy kétjegyű nap száma |
| Dd | Kétjegyű nap. Ha csak egyjegyű, kiegészül egy megelőző nullával |
| Ddd | Nap neve három betűs rövidítéssel |
| Dddd | Nap teljes neve |
| Hh | Kétszámjegyű óra, 12 órás ábrázolás |

| | |
|------|--|
| H | Egy vagy kétszámjegyű óra, 24 órás ábrázolás |
| HH | Kétszámjegyű óra, 24 órás ábrázolás |
| M | Egy vagy kétszámjegyű perc |
| Mm | Kétjegyű perc. Ha csak egyjegyű, kiegészül egy megelőző nullával |
| M | Egy vagy kétszámjegyű hónap |
| MM | Kétszámjegyű hónap, nullás kiegészítés |
| MMM | Hónap neve három betűs rövidítéssel |
| MMMM | Hónap teljes neve |
| S | Egy vagy kétszámjegyű másodperc |
| Ss | Kétszámjegyű másodperc, nullás kiegészítés |
| T | Egy betűs AM/PM kijelzés (dél előtt/délután) |
| Tt | Két betűs AM/PM kijelzés |
| Y | Egy számjegyű év (2001 megjelenítése: "1") |
| Yy | Két számjegyű év (2001 megjelenítése: "01") |
| Yyyy | Teljes év kiírása (2001 megjelenítése: "2001") |

A formátum-stringben szerepelhetnek még saját jelek, betűk, akár szöveg is. Ha nem szerepeltetünk benne a fenti karakterekből, akkor csak be kell írni. Ha a fenti betűk valamelyikét szeretnénk, akkor két aposztróf közé kell írni, pl.: hh:mm → 11:23
hh'd'mm → 11d23

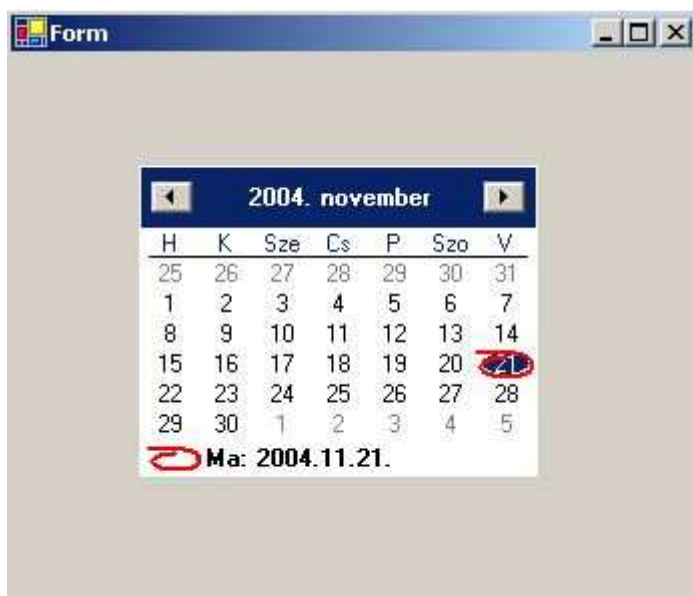
Angol stílusú dátum-formátum:

```
public void EgyeniDatumFormatum()
{
    dateTimePicker1.Format = DateTimePickerFormat.Custom;
    dateTimePicker1.CustomFormat = "MMMM dd, yyyy - dddd";
}
```

A DateTimePicker becsukott állapotban:

MonthCalendar komponens

A MonthCalendar komponens egy grafikus felületet nyújt a felhasználóknak, hogy lekérdezzék és módosítsák a dátumot. A komponens egy naptárat jelenít meg. Egy rácsháló tartalmazza a hónap számozott napjait, oszloponként rendezve a hét napjaihoz (hétfő-vasárnap). A komponensen kijelölhetünk több dátumot is (megadható számú). Lehet a dátumok között lépkedni - hónapos lépésekben - a komponens címsávjában szereplő bal vagy jobb nyílra kattintva. Ellentétben a hasonló DateTimePicker komponenssel, ebben a komponensben több dátumot is kijelölhetünk.



A komponens kinézete változtatható. Alapértelmezés szerint az aktuális nap a naptárban pirossal be van karikázva, és fel van tüntetve a rács alatt is. A rács mellett megjeleníthetjük a hetek számait is. Egy tulajdonság megváltoztatásával akár több naptár is lehet egymás mellett vagy alatt. A hét kezdő napja is megváltoztatható: Angliában a hét vasárnapkal kezdődik, nálunk hétfővel. Bizonyos dátumokat megjelölhetünk félkövér írásmóddal, kiemelve őket a többi közül (fontos dátumok). Ezek lehetnek egyedi dátumok, évenkénti vagy havi dátumok.

Tulajdonságok (properties):

| | |
|---------------------|--|
| AnnuallyBoldedDates | Tartalma az évenként előforduló fontos dátumok DateTime[] tömb |
| BoldedDates | Tartalma az egyedi fontos dátumok DateTime[] tömb |
| CalendarDimensions | A naptárak száma a komponensen belül (oszlop és sor) Alapértelmezés: Width=1; Height=1 |
| FirstDayOfWeek | A hét kezdő napja Alapértelmezés: Default, ilyenkor a rendszertől kérdezi le a területi beállításokat Értéknek adhatjuk neki a hét napjainak nevét angolul (Monday, ...) |
| MaxDate | Itt állítjuk be az elérhető legkésőbbi dátumot a naptárban Alapértelmezés: 9998.12.31. |
| MaxSelectionCount | A naptárban kijelölhető napok maximális száma Alapértelmezés: 7 |
| MinDate | Itt állítjuk be az elérhető legkorábbi dátumot a naptárban Alapértelmezés: 1753.01.01. |
| MonthlyBoldedDates | Tartalma a havonta előforduló fontos dátumok |

| | |
|-------------------|---|
| | DateTime[] tömb |
| ScrollChange | Ez a változó állítja be, hogy a komponens előre és vissza gombjaival hány hónapot lépjen a naptár Alapértelmezés: 0 (a következő/előző hónapra léptet) |
| SelectionRange | A komponensben kijelölt napok intervalluma Alapértelmezés: mindkét érték az aktuális dátumot tartalmazza |
| ShowToday | Beállítja, hogy látható-e az aktuális dátum a komponens alján Alapértelmezés: True |
| ShowTodayCircle | Beállítja, hogy a komponens bekarikázza-e az aktuális napot a naptárban Alapértelmezés: True |
| ShowWeekNumbers | Beállítja, hogy a komponensen a napok sorai előtt látszik-e a hét sorszáma (1-52) Alapértelmezés: False |
| TitleBackColor | A komponens címsávjának háttérszíne Alapértelmezés: Kék |
| TitleForeColor | A komponens címsávján megjelenő betűk színe Alapértelmezés: Fehér |
| TodayDate | Az aktuális dátum Pl.: 2004.05.06. |
| TrailingForeColor | Beállítja a naptárban megjelenő előző és következő hónapból belógó napok színét Alapértelmezés: Szürke |

Események (events)

| | |
|--------------|--|
| DateChanged | Akkor következik be, ha megváltozik a kijelölt dátumok intervalluma vagy előző/következő hónapra mozdul a naptár |
| DateSelected | Akkor következik be, ha a felhasználó kijelöl egy vagy több dátumot |

Kiemelt dátumok kezelése

Fontosabb dátumokat félkövér betűtípussal, kiemelten szerepeltethetünk a naptárban. Három tulajdonság (BoldedDates, AnnuallyBoldedDates és MonthlyBoldedDates) tárolhatja ezeket az egyedi, havonta vagy évente előforduló dátumokat. Ezek a tulajdonságok tartalmaznak egy tömböt, mely DateTime objektumokból áll. Hogyan lehet hozzáadni vagy elvenni ezek közül?

Dátum hozzáadás:

1. Hozzunk létre DateTime objektumot:

```
DateTime NyariSzunetKezdet = new DateTime(2004, 7, 1);
DateTime NyariSzunetVege = new DateTime(2004, 9, 13);
```

2. A dátum kiemelése a komponenshez (komponens példányhoz) hozzáadással:
(Három parancs: AddBoldedDate, AddAnnuallyBoldedDate, AddMonthlyBoldedDate)

```
monthCalendar1.AddBoldedDate(NyariSzunetKezdete);  
monthCalendar1.AddBoldedDate(NyariSzunetVege);
```

vagy

Eszerre több fontos dátum létrehozása és komponenshez társítása:

```
DateTime[] Vakaciok = {NyariSzunetKezdete, NyariSzunetVege};  
monthCalendar1.BoldedDates = Vakaciok;
```

Dátumok visszaállítása (félkövér szedés megszüntetése):

```
monthCalendar1.RemoveBoldedDate(NyariSzunetKezdete);
```

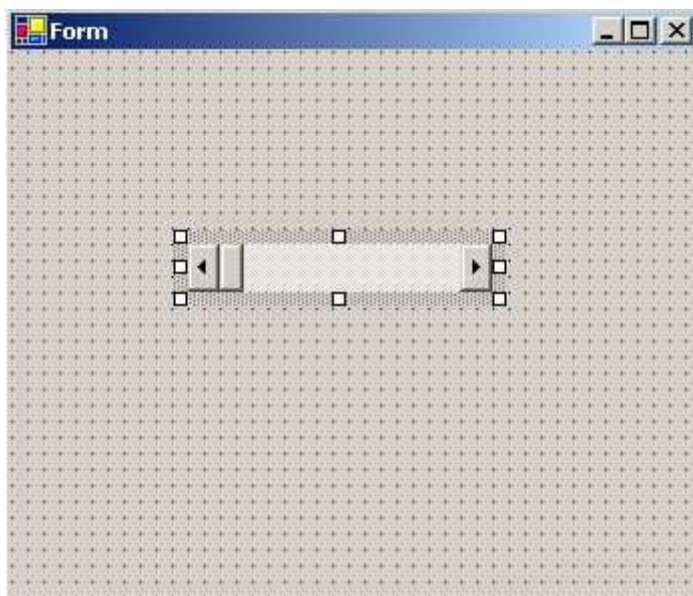
(Használható még a RemoveMonthlyBoldedDate és RemoveAnnuallyBoldedDate parancs is)

Az összes kiemelt dátum törlése:

```
monthCalendar1.RemoveAllBoldedDates();  
monthCalendar1.RemoveAllAnnuallyBoldedDates();  
monthCalendar1.RemoveAllMonthlyBoldedDates();
```

HorizontalScrollBar és VerticalScrollBar komponensek

A ScrollBar komponensek segítenek sok elemen át vagy nagy mennyiségű információban keresni vízszintes vagy függőleges görgetéssel.



A ScrollBar komponensek nem ugyanazok, mint az egyéb komponensekben (textbox, stb.) lévő, beépített görgetők.

A komponensek Value értéke tárolja, hogy hol helyezkedik el a görgetődoboz (pozíciójelző).

A ScrollBar komponensek a Scroll eseményt használják a pozíciójelző mozgásának figyelésére. Ennek a segítségével folyamatosan kérhető le a komponens Value értéke, miközben a pozíciójelzőt mozgatjuk.

Tulajdonságok:

| | |
|-------------|---|
| LargeChange | A görgető értékének változása, amikor a görgetőcsúszkán kattint a felhasználó, vagy megnyomja a PgUp/PgDn gombok valamelyikét Alap: 10 |
| Maximum | A pozíciójelző maximális helye Alap: 100 |
| Minimum | A pozíciójelző minimális helye Alap: 0 |
| SmallChange | A pozíciójelző elmozdulása, amikor a felhasználó az alsó-felső vagy bal-jobb gombok valamelyikére kattint, vagy a nyílbillentyűket megnyomja Alap: 1 |
| Value | A pozíciójelző helye Alap: 0 |

Események:

| | |
|--------|---|
| Scroll | Bekövetkezik, ha az állapotjelző elmozdul |
|--------|---|

A Value értéke egész szám, mely kezdetben 0, és meghatározza a pozíciójelző elhelyezkedését a görgetőcsúszkán. Ha minimális az érték, akkor a pozíciójelző a csúszka bal szélén (vízszintes görgető) vagy tetején (függőleges görgető) van; ha maximális, akkor jobb szélén ill. alul van. Hasonlóan, ha a Value a minimális és maximális értékek számtani közepe, akkor a csúszka közepén van. A Value értéke csak a Minimum és Maximum által megadott intervallumban lehet.

A pozíciójelző mozgatása nemcsak az irányokra kattintással lehetséges, hanem a pozíciójelző más helyre húzásával is történhet.

LargeChange

Ha a felhasználó lenyomja a PageUp vagy PageDown gombok valamelyikét, vagy a görgetősávon kattint a pozíciójelző valamelyik oldalán, úgy a Value érték a LargeChange érték szerint változik.

SmallChange

A nyílbillentyűk megnyomásakor, vagy a görgető iránygombjainak megnyomásakor a Value érték a SmallChange érték szerint változik.

Példa: vízszintes ScrollBar létrehozása és formhoz rendelése:

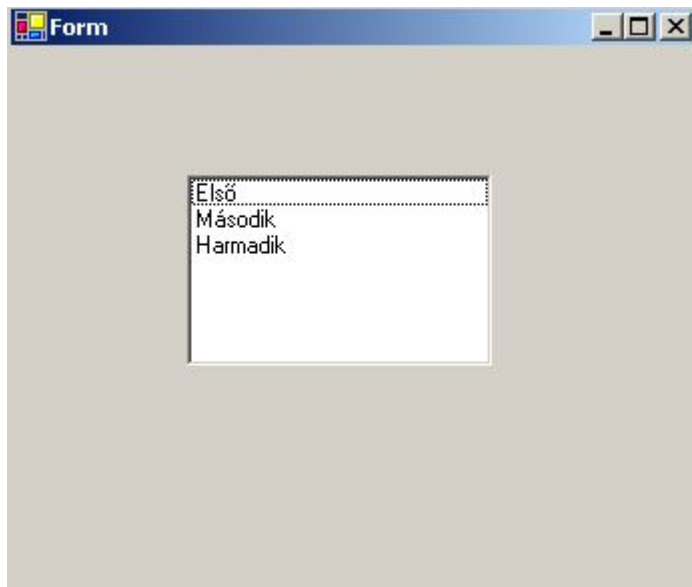
```
private void InitializeMyScrollBar()
{
    // HScrollBar létrehozása és alapállapotba állítása.
    HScrollBar hScrollBar1 = new HScrollBar();

    // A görgetőt a form aljához igazítjuk.
    hScrollBar1.Dock = DockStyle.Bottom;

    // A görgető hozzáadása a form vezérlőihez.
    this.Controls.Add(hScrollBar1);
}
```

Listbox

Egy listablak lehetővé teszi a felhasználó által megadott adatok kiválasztását a listából.



SelectionMode: a ListBox adatait jelölheti ki a felhasználó, akár egyszerre több adatot is, a megfelelő beállításokkal. Négy lehetséges beállítás létezik:

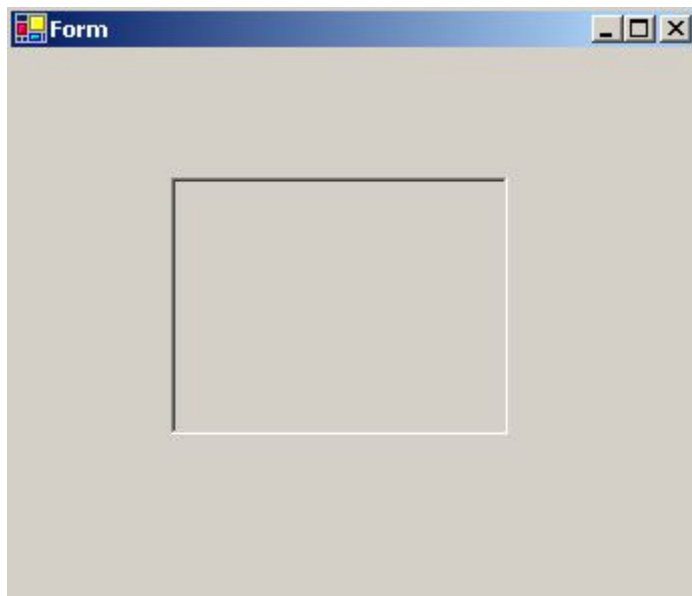
- None:
- One: egyszerre egy adatot választhatok ki a ListBoxból, ha az egérrel a ListBoxra kattintok vagy kijelölhetek a fel-, le-, balra- és jobbra-nyíl billentyűkkel. A One beállítás az alapértelmezett.
- MultiSimple: itt több adatot is kiválaszthatunk egyszerre, ha az egérrel a ListBox elemeire kattintok illetve ha a Space billentyűt nyomva tartjuk.

- MultiExtended: több adatot is kiválaszthatok ha a Shift billentyűt nyomva tartom illetve az egér gombját lenyomva tartom, és a fentebb említett billentyűkkel le-föl mozgunk.

HorizontalScrollBar: a görgetőket adhatjuk meg, ha a HorizontalScrollBar értéke true akkor látszanak a görgetők, ha false akkor nem látszanak, a false az alapértelmezett beállítás. Ha a ListBox magassága nagyobb vagy egyenlő mint a ListBox adatainak magassága, akkor nem látszik a görgető, hiába true a HorizontalScrollBar értéke.

Panel

Vannak olyan komponensek, amelyek képesek más komponensek tárolására. Ilyen tulajdonsággal rendelkező komponens a Panel. A Panel komponenshez az elemeket a tervezési idő alatt adjuk hozzá (például megtervezünk egy eszközsort). Amikor egy komponenst elhelyezünk a Panelen, akkor létrejön egy új szülő-gyermek kapcsolat a komponens és a Panel között. Ha a tervezési idő alatt valamilyen műveletet végzünk a Panellel (mozgatás, másolás, törlés stb.), akkor az érinti a tárolt komponenseket is.



A komponensek csoportba foglalásához először adjuk az alkalmazásunk Formjához a Panelt. A Panel komponens kijelölése után a szokásos módon elhelyezhetjük benne a komponenseket.

Enabled:

Ha a Panel Enabled property-je false-ra, inaktívrá van állítva akkor a Panel komponensei is inaktívak lesznek. Ilyenkor a menüelem szürke színnel kerül kijelzésre és nem választható ki.

```
Panel.Enabled=false;
```


Például ha egy gomb Enabled tulajdonságát inaktívrá állítjuk akkor nem tudunk a gombra kattintani.

```
Button.Enabled=false;
```

BorderStyle:

A Panel BorderStyle property-nél három beállítás közül választhatunk:

- None: a Panel nem különböztethető meg a Formtól, ez az alapbeállítás (a default) is.
- FixedSingle: a Panel körül jól látható keretet kapunk

```
panel1.BorderStyle = System.Windows.Forms.BorderStyle.FixedSingle;
```

- Fixed3d: a Panel jól elkülöníthető a Formtól

```
panel1.BorderStyle = System.Windows.Forms.BorderStyle.Fixed3D;
```

PictureBox

Ha képeket szeretnénk megjeleníteni alkalmazásunk Form-ján, akkor ennek legegyszerűbb módja a PictureBox komponens használata.

Miután feltettük a Form-ra a PictureBox komponenst, az Image property-jére kattintva kiválaszthatunk egy Bmp, Gif, Jpg, Ico, Emf vagy Wmf formátumú képet, melyet betölthetünk, megjeleníthetünk a PictureBox területén.

Image:

Az Image property-n keresztül rendelhetünk képet a kontrolhoz, illetve érhetjük el Image típusként. Fontos tudnunk, hogy az így betöltött kép bekerül a készítendő EXE állományba, így programunk szállításakor elegendő azt vinni, a kép állományt nem kell.

```
MyImage = new Bitmap(fileToDisplay);  
pictureBox1.Image = (Image) MyImage ;
```

SizeMode:

A PictureBoxSizeMode típusú Sizemode property-n keresztül szabályozhatjuk, hogy miként jelenjen meg a betöltött kép.

A PictureBoxSizeMode felsorolt típus a következő elemeket tartalmazza:

- AutoSize: a komponens mérete akkora lesz, mint a betöltött kép

```
pictureBox1.SizeMode = PictureBoxSizeMode.AutoSize;
```

- StretchImage: a kép mérete akkora lesz, mint amekkora a komponens

```
pictureBox1.SizeMode = PictureBoxSizeMode.StretchImage;
```

- CenterImage: a betöltött kép a komponens területének közepére igazodva jelenik meg

```
pictureBox1.SizeMode = PictureBoxSizeMode.CenterImage;
```

- Normal: alaphelyzet, ilyenkor a kép a komponens bal felső sarkához lesz igazítva

```
pictureBox1.SizeMode = PictureBoxSizeMode.Normal;
```

SizeModeChanged:

Amikor a SizeMode property értéke megváltozik, akkor kerül aktivizálásra a SizeModeChanged esemény.

ClientSize: A betöltött kép méreteit állíthatjuk be, a Size.Width és Size.Height property-kel adhatjuk meg a kép szélességét és magasságát.

```
pictureBox1.ClientSize = New Size(xSize, ySize);
```

Timer komponens

A timer komponens egy eseményt idéz elő rendszeres időközönként.



Az időtartam az Interval tulajdonságban van megadva ezredmásodpercben. Ha a timer engedélyezve van, a Tick esemény minden egyes alkalommal bekövetkezik, ha eltelt a megadott intervallum. A timer-t elindíthatjuk vagy leállíthatjuk. Ha megszakítjuk, visszaáll alaphelyzetbe. Nincs lehetőség a timer szüneteltetésére.

Tulajdonságok:

| | |
|----------|--|
| (Name) | A timer neve |
| Enabled | A timer engedélyezve van-e Alapértelmezés: False |
| Interval | Események időzítése ezredmásodpercben megadva Alapértelmezés: 100 |

Események:

| | |
|------|--|
| Tick | A megadott időintervallum elteltével következik be |
|------|--|

Timer kezelése

Lehetőség van a timer leállítására és elindítására az alábbi két paranccsal:

```
timer1.Start();  
timer1.Stop();
```

vagy más módon, az Enabled tulajdonság igazra/hamisra állításával.

Az időnként kívánt eseményeket a Tick eseménybe kell beleírni. Itt állhat többsornyi programkód vagy egy egyszerű metódushívás is. Például egy egyszerű óra Label komponensből:

```
private void timer1_Tick(object sender, System.EventArgs e)  
{  
    this.lblOra.Text = "Idő:" +  
        Convert.ToString(System.DateTime.Now.Hour) +  
        ":" + Convert.ToString(System.DateTime.Now.Minute) +  
        ":" + Convert.ToString(System.DateTime.Now.Second);  
    //egyszerűbben, dátummal:  
    //this.lblOra.Text = Convert.ToString( System.DateTime.Now );  
}
```

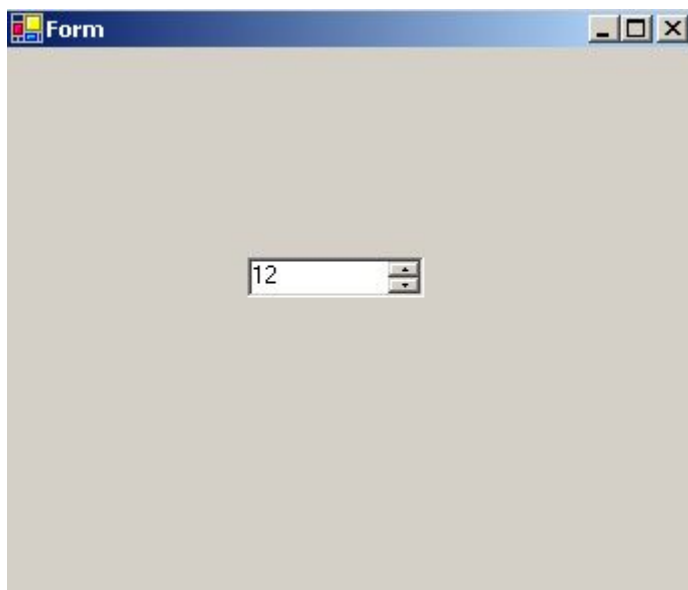
Az Interval határai:

A változó értéke 1 és 64767 közé eshet, tehát a leghosszabb intervallum is alig hosszabb egy percnél. Nincs garantálva, hogy a megadott intervallum pontosan telik el. Hogy a pontosságot biztosítsuk, a timer-nek időnként ellenőriznie kell a rendszerórát. A rendszer 18 óraütést generál másodpercenként. Így - még akkor is, ha az Interval tulajdonság ezredmásodpercben számolandó - a leggyorsabb ütemezés csak a másodperc 18-ad része lehet.

NumericUpDown

Megjegyzés

A NumericUpDown ellenőrzés egyetlen numerikus értéket tartalmaz, amely növelhető vagy csökkenthető az ellenőrzés fel vagy le billentyűjére való kattintásával. A használó be is léphet egy értékbe, de csak akkor, ha a ReadOnly tulajdonság igazra van állítva.



A numerikus kijelző alakítható a `DecimalPlaces`, `Hexadecimal` vagy a `ThousandsSeperator` tulajdonságok beállításával. Az ellenőrzésben történő hexadecimális értékek beállításakor, állítsa a `Hexadecimal` tulajdonságot `true`-ra. `ThousandsSeperator` bemutatásakor decimális számokban, amikor lehetséges, állítsa be a `ThousandsSeperator` tulajdonságot `true`-ra. Pontosabban meghatározva a decimal symbol után bemutatott számok mennyiségét állítsa be a `DecimalPlaces` tulajdonságot a decimális helyek számához.

Részletezve a megengedett értékek sorát az ellenőrzés miatt, állítsa be a `Minimum` és `Maximum` tulajdonságokat. Állítsa be az `Increment` értéket, hogy meghatározza a `Value` tulajdonság csökkenő vagy növekvő értékét, amikor a felhasználó a fel vagy le nyílra kattint.

Amikor meghívja az `UpButton` vagy `DownButton` metódusokat kóddal vagy a fel vagy le gombra kattintással, az új értéket jóváhagyja, és az ellenőrzés felülíródik az új érték megfelelő formátumával. Különösen, ha `UserEdit` `true`-ra van állítva, `ParseEditText`-et előzőleg meghívja az érték felülírása és jóváhagyása érdekében. Az értéket leellenőrzi, hogy `Minimum` és `Maximum` értékek között legyen, majd meghívja az `UpdateEditText` metódust.

A `Value` tulajdonság növelése vagy csökkentése akkor történik, amikor rákattintunk a fel vagy le nyílakra a fel-le ellenőrzésnél. A hiányérték 1.

Kivétel

| Kivételes típus | Feltétel |
|--------------------------------|-------------------------------|
| <code>ArgumentException</code> | A kijelölt érték nem pozitív. |

`True`, a használó megváltoztatta a `Text` tulajdonságot, másfelől `false`.

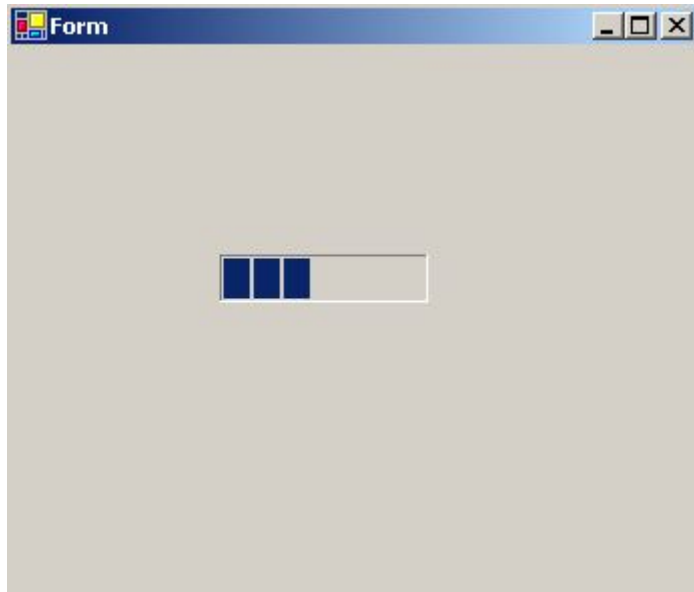
Ha a `Text` tulajdonság be van állítva, míg a `UserEdit` tulajdonság `true`-ra van állítva, akkor meghívja az `UpdateEditText` metódust. Ha a `Text` tulajdonság be van állítva,

míg a UserEdit tulajdonság false-ra van állítva, akkor a ValidateEditText metódust hívja meg.

ProgressBar

A Value Displayed beállítása a Windows Forms ProgressBar Control által

A NET Framework számos különböző módot nyújt bemutatni egy adott értéket a ProgressBar Control-on belül. Közelebb visz a döntéshez, mely a kéznél levő feladattól, illetve a megoldandó problémától függ.



- Közvetlenül állítsa be a ProgressBar Control értékét. Ez a megközelítés hasznos azokhoz a feladatokhoz, ahol ismeri a kimért adatok egészét, mintha lemezt olvasna egy adatforrásról. Ráadásul, ha az értéket csak egyszer vagy kétszer kell beállítani, ez egy könnyű módja a megoldásnak. Végül, használja ezt az eljárást, ha szükséges csökkenteni a ProgressBar által megadott értéket.
- Növelje a ProgressBar mutatót egy meghatározott értékkel. Ez a megközelítés hasznos, amikor egy egyszerű végösszeget mutat a Minimum és a Maximum között, úgy, mint az eltelt időt, vagy a fájlok számát, amelyet már feldolgozott egy ismert végösszegeből.
- Növelje a ProgressBar mutatót egy értékkel, amely változik. Ez a megközelítés akkor hasznos, amikor a megadott értéket különböző összegekben ki kell cserélni.

A ProgressBar értékének közvetlen beállítása

A legközvetlenebb út a bemutatott érték beállítására a ProgressBar által, az érték tulajdonságának a beállítása. Ezt megteheti tervezett vagy szokásos időben is.

A ProgressBar értékének közvetlen beállítása

1. Állítsa be a ProgressBar ellenőrzését Minimum és Maximum értékre.
2. A kódban, állítsa be az ellenőrzés kódjának tulajdonságát egy integerre a Minimum és a Maximum értékek között, amelyet már megadott.

Ha beállította az érték tulajdonságát a Minimum és a Maximum által meghatározott határokon belül, az ellenőrzés kiad egy ArgumentException kivételt.

A ProgressBar értékének növelése egy kijelölt intervallum által

Ha haladást mutat, amely egy meghatározott intervallumból ered, beállíthatja az értéket, aztán meghív egy metódust, amely növeli a ProgressBar Control értékét az intervallum által. Ez hasznos az időzítők és más szövegek számára, ahol a haladást nem az egész százalékaként mérik.

1. Állítsa be ProgressBar ellenőrzését Minimum és Maximum értékre.
2. Állítsa be az ellenőrzés Step tulajdonságát egy egészre, ábrázolva ezzel a ProgressBar bemutatott értékének növekedését.
3. Hívja meg a PerformStep metódust, hogy megváltoztassa a bemutatott értéket a Step tulajdonságban beállított mennyiségnek megfelelően.

A ProgressBar értékének növelése változó mennyiséggel

Végül növelheti a ProgressBar által bemutatott értéket úgy, hogy minden egyes növelés páratlan értékű legyen. Ez hasznos, amikor betartja a páratlan műveletek sorozatának útvonaltát úgy, mint különböző méretű fájlok írásakor a merevlemezre vagy a haladás az egész százalékaként való mérésekor.

A ProgressBar növelése egy dinamikai érték segítségével

1. Állítsa be a ProgressBar ellenőrzését Minimum és Maximum értékre.
2. Hívja meg az Increment metódust, annak érdekében, hogy megváltoztassa az egész által meghatározott értéket.

TrackBar ellenőrzés

A TrackBar egy ablak, amely egy slider-t és tetszőleges mértékű jeleket tartalmaz. Amikor a felhasználó mozgatja a slider-t, akár egeret, akár billentyűzetet használva, a TrackBar üzenetekben közli a változást.

A TrackBar akkor hasznos, amikor azt szeretnénk, hogy a felhasználó rangsorolja döntését egyedi érték vagy egymást követő értékek választása között. Például, használhatja a TrackBar-t megengedve a felhasználónak a billentyűzet ismétlésének fokának beállítását azáltal, hogy a slider-t egy megadott jelhez mozgatja.

A slider egy TrackBar-ban akkor nő a meghatározott mértékben, amikor a felhasználó állítja be azt. Az értékek ebben a sorrendben logical units-ként fordulnak elő. Például, ha pontosítja, hogy a logical units-nak kell, hogy legyen 0-5 terjedő sorrendje, akkor a slider csak hat pozíciót foglalhat el: egy pozíció a TrackBar bal oldalán és egy pozíció minden egyes növekvésre sorrendben. Tipikusan minden egyes pozíciót ezek közül egy jellel azonosít.

Létrehozhatunk egy `TrackBar`-t a `CreateWindowEx` funkció használatával, pontosítva a `TrackBar` Class ablak osztályt. Miután létrehoztuk a `TrackBar`-t, használhatjuk `TrackBar` üzeneteket beállítani és helyrehozni néhány tulajdonságot. A változások, amelyeket beállítottunk okozott tartalmazzák a `Minimum` és `Maximum` pozíciók beállítását a slider számára, jeleket rajzolva, beállítva a választások sorrendjét és újrapozícionálja a slider-t.

A `TrackBar` egy kacszkaringó ellenőrzés a `ScrollBar` ellenőrzéshez hasonlóan. Alakíthatjuk a sorrendet a `Minimum` tulajdonság beállításával a sorrend alacsonyabb végének pontosításával és a `Maximum` tulajdonságot a sorrend magasabb végének pontosításával.

A `LargeChange` tulajdonság definiálja a növekedést hozzáadva vagy kivonva a `Value` tulajdonságból, amikor következőre kattint a slider egyik oldalán. A `TrackBar` bemutatható vízszintesen és függőlegesen is.

Használhatjuk ezt az ellenőrzést numerikus adat betöltésekor a `Value` tulajdonságon keresztül. Bejátszhatjuk ezt az adatot egy ellenőrzésben vagy kód használatával.

`LargeChange` tulajdonság

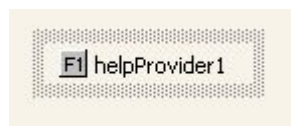
Adjon meg vagy állítson be egy értéket hozzáadva vagy kivonva a `Value` tulajdonságból, amikor a scroll boksz nagyobb távolságban mozdul el.

| Kivételes típus | Feltétel |
|-----------------|--|
| Exception | A megjelölt érték kevesebb, mint nulla |

Amikor a felhasználó a `PAGE UP` vagy `PAGE DOWN` gombokat használja, vagy a `TrackBar`-ra kattint a slider egyik oldalán, a `Value` tulajdonság a `LargeChange` tulajdonságban beállított érték szerint változik. Figyelembe kell venni a `LargeChange` tulajdonság beállítását a magasság vagy a szélesség értékek százalékaként. Ez megtartja a távolságot, hogy a `TrackBar` megfelelően változzon méretéhez képest.

HelpProvider

`HelpProvider` minden kéréلمe karbantartja a hozzákapcsolódó komponensek referenciáit. Ahhoz, hogy összekapcsoljuk a `Help` fájlt a `HelpProvider` objektummal, be kell állítani a `HelpNamespace` tulajdonságot. Ekkor meghatározunk egy típust, amit `SetHelpNavigator`-nak neveznek és hozzáadjuk a `HelpNavigator` értéket a specifikált komponenshez. Hozzáadhatunk egy kulcsszót a segítséghez a `SetHelpKeyword`-el.



Ahhoz, hogy hozzáadjunk egy jellegzetes `Help` stringet a komponenshez, használjuk a `SetHelpString` metódust. Ez a string ettől kezdődően egy pop-up menüben fog

felvillanni, amikor a felhasználó rákattint az F1 billentyűre, persze csak akkor, ha a fókuszt az a komponens birtokolja amelyikről segítséget akar kapni a felhasználó. Ha a HelpNamespace nincs még beállítva, akkor a SetHelpString-et kell használnod, hogy Help szöveget adhassunk. Ha már beállítottad a HelpNamespace-t és a Help string-et, akkor a Help HelpNamespace-ben van és elsőbbséggel rebdelkezik.

Minta példa

A bemutatandó példa demonstrálja, hogy a HelpProvider osztály, hogyan jeleníti meg a context-sensitive segítséget a formon, ami négy cím mezőt tartalmaz. A példa a SetHelpString-et használja, hogy beállítsa a segítségnyújtó ToolTip szöveget. Amikor a context-sensitive Help gombot (vagyis a kis kérdőjelet a jobb felső sarokban) használjuk és ráklikkeljük a Help kurzort a cím mezőre, akkor a ToolTip szöveg megjeleníti a hozzá fűzött szöveget. Amikor a fókuszt valamelyik cím mezőn van és megnyomjuk az F1 billentyűt, akkor az mspaint.chm Help fájl kiíródik, mert a HelpNamespace tulajdonság az mspaint.chm-re van beállítva. A HelpProvider.SetShowHelp metódusa minden cím komponenshez hozzá van rendelve, hogy azonosítsa, hogy a Help tartalom elérhető.

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

public class Form1 : System.Windows.Forms.Form
{
    private System.Windows.Forms.TextBox addressTextBox;
    private System.Windows.Forms.Label label2;
    private System.Windows.Forms.TextBox cityTextBox;
    private System.Windows.Forms.Label label3;
    private System.Windows.Forms.TextBox stateTextBox;
    private System.Windows.Forms.TextBox zipTextBox;
    private System.Windows.Forms.HelpProvider helpProvider1;
    private System.Windows.Forms.Label helpLabel;

    [STAThread]
    static void Main()
    {
        Application.Run(new Form1());
    }

    public Form1()
    {
        this.addressTextBox = new System.Windows.Forms.TextBox();
        this.helpLabel = new System.Windows.Forms.Label();
        this.label2 = new System.Windows.Forms.Label();
```



```

this.cityTextBox = new System.Windows.Forms.TextBox();
this.label3 = new System.Windows.Forms.Label();
this.stateTextBox = new System.Windows.Forms.TextBox();
this.zipTextBox = new System.Windows.Forms.TextBox();

// Help Label
this.helpLabel.BorderStyle =
System.Windows.Forms.BorderStyle.Fixed3D;
this.helpLabel.Location = new System.Drawing.Point(8, 80);
this.helpLabel.Size = new System.Drawing.Size(272, 72);
this.helpLabel.Text = "Click the Help button in the title bar, then
click a control " +
    "to see a Help tooltip for the control. Click on a control and
press F1 to invoke " +
    "the Help system with a sample Help file.";

// Address Label
this.label2.Location = new System.Drawing.Point(16, 8);
this.label2.Size = new System.Drawing.Size(100, 16);
this.label2.Text = "Address:";

// Comma Label
this.label3.Location = new System.Drawing.Point(136, 56);
this.label3.Size = new System.Drawing.Size(16, 16);
this.label3.Text = ",";

// Create the HelpProvider.
this.helpProvider1 = new System.Windows.Forms.HelpProvider();

// Tell the HelpProvider what controls to provide help for, and
// what the help string is.
this.helpProvider1.SetShowHelp(this.addressTextBox, true);
this.helpProvider1.SetHelpString(this.addressTextBox, "Enter the
street address in this text box.");

this.helpProvider1.SetShowHelp(this.cityTextBox, true);
this.helpProvider1.SetHelpString(this.cityTextBox, "Enter the city
here.");

this.helpProvider1.SetShowHelp(this.stateTextBox, true);

```

```

        this.helpProvider1.SetHelpString(this.stateTextBox, "Enter the
state in this text box.");

        this.helpProvider1.SetShowHelp(this.zipTextBox, true);
        this.helpProvider1.SetHelpString(this.zipTextBox, "Enter the zip
code here.");

        // Set what the Help file will be for the HelpProvider.
        this.helpProvider1.HelpNamespace = "mspaint.chm";

        // Sets properties for the different address fields.

        // Address TextBox
        this.addressTextBox.Location = new System.Drawing.Point(16, 24);
        this.addressTextBox.Size = new System.Drawing.Size(264, 20);
        this.addressTextBox.TabIndex = 0;
        this.addressTextBox.Text = "";

        // City TextBox
        this.cityTextBox.Location = new System.Drawing.Point(16, 48);
        this.cityTextBox.Size = new System.Drawing.Size(120, 20);
        this.cityTextBox.TabIndex = 3;
        this.cityTextBox.Text = "";

        // State TextBox
        this.stateTextBox.Location = new System.Drawing.Point(152, 48);
        this.stateTextBox.MaxLength = 2;
        this.stateTextBox.Size = new System.Drawing.Size(32, 20);
        this.stateTextBox.TabIndex = 5;
        this.stateTextBox.Text = "";

        // Zip TextBox
        this.zipTextBox.Location = new System.Drawing.Point(192, 48);
        this.zipTextBox.Size = new System.Drawing.Size(88, 20);
        this.zipTextBox.TabIndex = 6;
        this.zipTextBox.Text = "";

        // Add the controls to the form.
        this.Controls.AddRange(new System.Windows.Forms.Control[] {
            this.zipTextBox, this.stateTextBox,
            this.label3, this.cityTextBox,

```

```

        this.label2, this.helpLabel,
        this.addressTextBox));

    // Set the form to look like a dialog, and show the HelpButton.
    this.FormBorderStyle =
System.Windows.Forms.FormBorderStyle.FixedDialog;
    this.HelpButton = true;
    this.MaximizeBox = false;
    this.MinimizeBox = false;
    this.ClientSize = new System.Drawing.Size(292, 160);
    this.Text = "Help Provider Demonstration";

}
}

```

ImageList

Függvényeket biztosít az Image objektumok gyűjteményének irányításához. Ez az osztály nem örökölheto.



Észrevételek

ImageList-et tipikusan például ListView, TreeView, vagy ToolBar-nál szokás használni. ImageListhez adhatunk bitmaps, ikonokat, vagy meta fájlakat, és a különböző komponensek használhatják azokat, ha igényük van rá.

ImageList használ egy fogantyút a képek listájának irányítására. Ez a fogantyú „Handle” mindaddig nem jön létre, amíg bizonyos műveletek, nem hajtódnak végre, mint például az ImageList feltöltése képekkel.

Példa

A következő példa megmutatja a kijelölt képeket, megváltoztatja, és azután megjeleníti.

```

namespace myImageRotator
{
    using System;
    using System.Drawing;
    using System.ComponentModel;
    using System.Windows.Forms;

```

```

public class Form1 : System.Windows.Forms.Form
{
    protected Container components;
    protected ListBox listBox1;
    protected Label label2;
    protected Label label3;
    protected Label label5;
    protected PictureBox pictureBox1;
    protected Button button1;
    protected Button button2;
    protected Button button3;
    protected Button button4;
    protected Panel panel1;
    protected ImageList imageList1;
    protected Graphics myGraphics;
    protected OpenFileDialog openFileDialog1;

    private int currentImage = 0;

    public Form1()
    {
        InitializeComponent();
        imageList1 = new ImageList () ;
        // The default image size is 16 x 16, which sets up a larger
        // image size.
        imageList1.ImageSize = new Size(255,255);
        imageList1.TransparentColor = Color.White;
        // Assigns the graphics object to use in the draw options.
        myGraphics = Graphics.FromHwnd(panel1.Handle);
    }

    protected override void Dispose( bool disposing )
    {
        if( disposing )
        {
            if (components != null)
            {
                components.Dispose();
            }
        }
    }
}

```

```

        base.Dispose( disposing );
    }

    private void InitializeComponent()
    {
        // kezőérték beállítás a listBox1, label2, pictureBox1,
        // button2, button3, panel1, openFileDialog1, button4, label1,
        // button1, és imageList1-nek.
    }

    protected void button1_Click (object sender, System.EventArgs e)
    {
        displayNextImage();
    }

    protected void button2_Click (object sender, System.EventArgs e)
    {
        imageList1.Images.RemoveAt( listBox1.SelectedIndex );
        listBox1.Items.Remove( listBox1.SelectedIndex );
    }

    protected void button3_Click (object sender, System.EventArgs e)
    {
        imageList1.Images.Clear();
    }

    protected void button4_Click (object sender, System.EventArgs e)
    {
        openFileDialog1.Multiselect = true ;
        if( openFileDialog1.ShowDialog() == DialogResult.OK )
        {
            if ( openFileDialog1.FileNames != null )
            {
                for( int i = 0 ; i < openFileDialog1.FileNames.Length ;
i++ )
                {
                    addImage( openFileDialog1.FileNames[i] );
                }
            }
            else
                addImage( openFileDialog1.FileName );
        }
    }

```

```

    }

}

private void addImage(string imageToLoad)
{
    if (imageToLoad != "")
    {
        imageList1.Images.Add(Image.FromFile(imageToLoad));
        listBox1.BeginUpdate();
        listBox1.Items.Add(imageToLoad);
        listBox1.EndUpdate();
    }
}

void displayNextImage()
{
    if(imageList1.Images.Empty != true)
    {
        if(imageList1.Images.Count-1 > currentImage)
        {
            currentImage++;
        }
        else
            currentImage=0;
        panell1.Refresh();
        imageList1.Draw(myGraphics,10,10,currentImage);
        pictureBox1.Image = imageList1.Images[currentImage];
        label3.Text = "Current image is " + currentImage ;
        listBox1.SelectedIndex = currentImage;
        label5.Text = "Image is " + listBox1.Text ;
    }
}

public static void Main(string[] args)
{
    Application.Run(new Form1());
}
}

```

Tulajdonságok

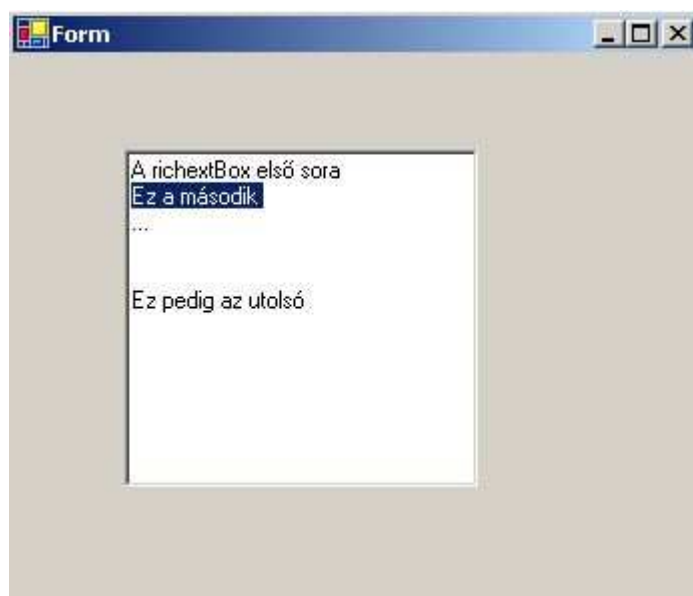
| | |
|------------------|---|
| ColorDepth | Beállítja a kép lista képeinek szín telítettségét |
| Contanier | Beállítja az IContaniert amit a komponens tartalmaz |
| Handle | Beállítja a kép lista objektum bizonyos fogantyúját |
| HandleCreated | Beállítja az értéket ha Win32-es fogantyút készítettünk. |
| Images | Itt lehet a képeket az ImageListhez fűzni |
| ImagesSize | Képméret beállítása |
| ImageStream | Beállíthatjuk a fogantyút az ImageListStreamerhez összekapcsolva az aktuális kép listával |
| Site | Beállítja az Isitetot a komponenshez |
| TransparentColor | Színáttetszőség beállítása |

RichTextBox

Reprezentál egy Windows rich text box komponenst.

Észerevételek:

A RichTextBox komponens megengedi a felhasználónak, hogy belépjen és szerkessze a szöveget, ami mellett még több, egymástól különböző formátumot is engedélyez, mint egy mezei TextBox komponens. A szöveget kijelölhetjük közvetlenül a komponensben, vagy be tölthetjük egy Rich Text Format (RTF) fájlból, vagy egy egyszerű text fájlból. A komponensen belüli szöveg lehet egyszerű karakteres vagy paragraph formátumú.



A RichTextBox komponens ad egy számot a tulajdonságairól, amit fel tudsz használni a komponensen belül, annak bármelyik részén, a szövegen, hogy alkalmazhassunk egy formátumot. Ahhoz, hogy megváltoztassuk a szöveg formátumát, először ki kell jelölni. Csak a kijelölt szövegben tudjuk megváltoztatni a karakter és paragraph formátumot. A SelectionFont tulajdonság lehetőséget biztosít számunkra, hogy a szöveg bold vagy italic legyen. Még arra is használhatjuk, hogy meghatározzuk a méretét vagy a betűképet. SelectionColor tulajdonsággal a szöveg színe állítható be. Gyorslista készítéséhez, használjuk a SelectionBullet tulajdonságot, beállíthatjuk a paragraph formátumot a SelectionIndent, SelectionRightIndent, és SelectionHangingIndent tulajdonságokkal.

A RichTextBox komponens olyan metódusokat is ad amelyek funkcióival tudsz megnyithatunk vagy menthetünk fájlokat. A LoadFile metódus megengedi, hogy megfelelő RTF vagy ASCII szöveget, tartalmazó fájlokat nyissunk meg a komponensbe. Ezen felül még olyan fájlokat is meg tudsz nyitni, amiket már más megnyitott velünk egy időben. A SaveFile metódus megengedi, hogy megfelelő RTF vagy ASCII fájl formátumban mentünk el szövegeket. Hasonló módon, mint a LoadFile metódus, a SaveFile metódust is tudjuk arra használni, hogy egy megnyitott fájlba menthetünk. A RichTextBox komponens olyan lehetőséget is ad, hogy stringeket keressünk a szövegben. A Find metódus is arra is lehetőséget biztosít, hogy hasonló stringeket keress a szövegben.

Az is megvalósítható, hogy adatot tároljon a memóriában.

Ha a komponensen belüli szöveg tartalmaz egy linket, mondjuk egy web sitera, akkor használhassuk a DetectUrls tulajdonságot, hogy megfelelően jelenjen meg a link a komponens szövegében. Ezután a LinkClicked eventtel elérhetjük, hogy ez működjön is. A SelectionProtected tulajdonsággal engedélyezhetjük azt, hogy a szöveg a komponensen belül védve legyen a felhasználók manipulációival szemben. Ezzel a levédett szöveggel a komponensen belül, hivatkozhatunk a Protected eventre, amivel közölhetjük a mezei felhasználóval, hogy ez a szövegrész számára nem módosítható, ha módosítani akarja.

Alkalmazásokat, amelyeket már a TextBox komponens is használ, könnyen bele lehet építeni a RichTextBox komponensbe, azonban a RichTextBox komponens nem tartalmazza a TextBox 64k karakter kapacitás limitjét.

Példa:

A bemutatandó példában készítünk egy RichTextBox komponenst, ami beolvas egy RTF fájlt, majd megkeresi az első helyet, ahol a "Text" kifejezés jelen van. Ezután a program megváltoztatja a kijelölt szövegrész betű típusát, méretét, színét, majd ezekkel a változtatásokkal visszamenti az eredeti fájlba. Végezetül a megoldásban hozzáadja a Form felületéhez a RichTextBoxot.

```
public void CreateMyRichTextBox()
{
    RichTextBox richTextBox1 = new RichTextBox();
    richTextBox1.Dock = DockStyle.Fill;

    richTextBox1.LoadFile("C:\\MyDocument.rtf");
    richTextBox1.Find("Text", RichTextBoxFinds.MatchCase);
}
```



```

richTextBox1.SelectionFont = new Font("Verdana", 12, FontStyle.Bold);
richTextBox1.SelectionColor = Color.Red;

richTextBox1.SaveFile("C:\\MyDocument.rtf",
RichTextBoxStreamType.RichText);

this.Controls.Add(richTextBox1);
}

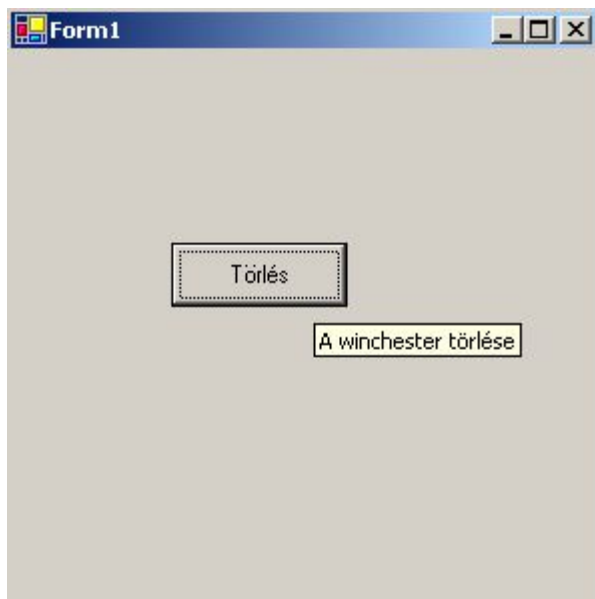
```

Néhány fontosabb Tulajdonság:

| | |
|-----------------|---|
| Dock | Hogy hol helyezkedjen el a formon belül |
| AutoSize | Automatikusan beállítja a komponens méretét font változtatás után |
| Enabled | Meghatározza, hogy lehet e módosítani a tartalmát |
| BackgroundImage | Háttérkép |
| BackColor | Háttér szín |
| Bounds | Nem kliens tartalmának a mérete |
| ClientRectangle | Annak a téglalapnak a tulajdonságait lehet beállítani, amit a kliens használ |
| ClientSize | A felhasználó által használható terület hosszúságát és szélességét lehet beállítani |
| ContextMenu | Beállítja a komponens gyorsindító menüjét |
| Controls | Tartalmazza a komponensen belül használható komponenseket |
| Cursor | Az egér mutató kinézete amíg a komponens felett van |
| HasChildren | Beállítja a komponenst ha több gyermek származik belőle |
| Height | Magasság |
| Width | Szélesség |
| Left | Balról behúzás |
| Right | Jobbról behúzás |
| Top | Felülről |
| Bottom | Alulról |
| Size | Méret |
| Stb.... | |

ToolTip

Reprezentál egy kis téglalap alakú, felbukkanó ablakot, ami tömör szövegben kijelzi annak a dolognak a tulajdonságát vagy használatát, amire az egérrel rápozícionálunk.



Észrevételek:

A ToolTip osztály lehetőséget ad, hogy segítsünk a felhasználóknak, ha rápozícionálnak az egérrel egy komponensre. A ToolTip osztályt tipikusan arra használják, hogy felhívják a felhasználók figyelmét a kontrol használatáról. Például, hozzá rendelhetünk egy TextBox komponenshez egy ToolTip szöveget, ami megadhatja a TextBox nevét és típusát, amit oda be lehet írni. Összegezve a segítségen felül a ToolTp osztály futásközbeni információadásra is nagyon jól használható. Például arra is használhatjuk, hogy kiírja a képernyőre a kapcsolat aktuális sebességét is vagy az egér fókuszát, mikor a felhasználó azt a PictureBox felett mozgatja.

A ToolTip osztályt valamennyi tárolóban használhatjuk. Ahhoz, hogy specifikáljuk egy konténert, használjuk a ToolTip komponens konstruktorát. Hogy egy ToolTip szöveg kiíródjon amikor a felhasználó egy komponens fölé pozícionál az egérrel, a ToolTip szövegnek össze kell lennie kapcsolva a komponenssel egy kérelem formájában a ToolTip osztályban. A ToolTip szöveg a komponens összekapcsolására használd a SetToolTip függvényt. A SetToolTip függvény több mint egyszer tudja utasítani a különböző komponenseknek, hogy változtassák meg a szövegüket, amivel össze vannak kötve. Ha azt akarjuk, hogy a szöveg össze legyen kötve a komponenssel,

használjuk a `GetToolTip` függvényt. Ha törölni akarjuk az összes `ToolTip` szöveget, a `RemoveAll` függvényt használjuk.

Megjegyzés: A `ToolTip` szöveg olyan komponens fölé nem íródik ki, ami nincs engedélyezve.

Példa:

A követendő példa készít egy hivatkozást és összeköti azt a Formmal egy beépített kérelemmel. A kód ezután inicializál késleltető tulajdonságokat az `AutoPopDelay`, `InitialDelay`, és `ReshowDelay`-el. A `ShowAlways` tulajdonság igazra állításával tudjuk beállítani, hogy a `ToolTip` szövegek mindaddig megjelenjenek, amíg a Form fut. Végezetül, a példabeli megoldás összeköti a `ToolTip` szöveget két komponenssel a Formon, egy `Button`-al és egy `CheckBox`-al.

```
private void Form1_Load(object sender, System.EventArgs e)
{
    // Create the ToolTip and associate with the Form container.
    ToolTip toolTip1 = new ToolTip();

    // Set up the delays for the ToolTip.
    toolTip1.AutoPopDelay = 5000;
    toolTip1.InitialDelay = 1000;
    toolTip1.ReshowDelay = 500;
    toolTip1.ShowAlways = true;

    toolTip1.SetToolTip(this.button1, "My button1");
    toolTip1.SetToolTip(this.checkBox1, "My checkBox1");
}
```

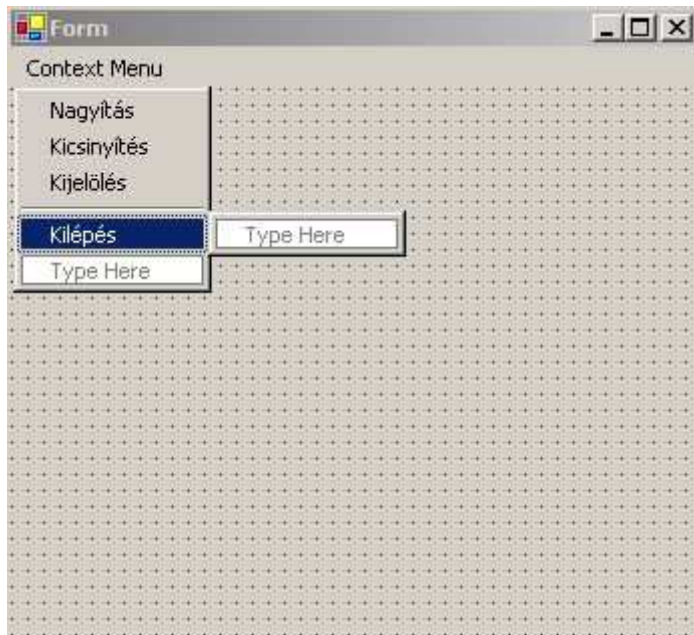
Tulajdonságok:

| | |
|----------------|---|
| Active | Megadja vagy beállítja a kijelezendő értéket, amíg a <code>ToolTip</code> aktív. |
| AutomaticDelay | Automatikus késleltetés beállítása |
| AutoPopDelay | Itt állíthatjuk be azt az időt amíg a szöveg jelen legyen, amikor rá pozícionálunk |
| InitialDelay | Az az idő amíg vár mielőtt megjelenik a szöveg |
| ReshowDelay | Annak az időnek a hosszát lehet beállítani, amíg várnia kell a szöveg felvillantásakor, mikor egyik komponensről a másikra megyünk át |
| ShowAllways | Szövegkijelzés beállítása ha a form nem aktív |
| Site | <code>ISite Component</code> |

ContextMenu

A ContextMenu-ről általában:

A ContextMenu osztály egy menüt szolgáltat, amit egy kontrol vagy a form fölött az egér jobb gombjának lenyomásával érhetünk el. Ezt a menüt általában arra használjuk, hogy a MainMenu elemeit összeválogatva, az egyes területekhez rendeljük őket, így segítve a felhasználót. Például a TextBox-hoz rendelhetünk egy contextmenüt, melynek segítségével beállítható a TextBox szövegének betűtípusa vagy a vágólapra másolhatjuk, törölhetjük stb...



A ContextMenü részei:

Publikus konstruktor

ContextMenu konstruktor

A ContextMenu osztály egy új példányát inicializálja.

Publikus propertik

Container

Visszaadja azt az Icontainer-t ami tartalmazza a komponenst.

MenuItems

Visszaadja a MenuItem objektumokat, amik a menüben vannak.

SourceControl

Visszaadja , hogy melyik kontrol jeleníti meg a menüt.

Publikus metódusok

Equals

Eldönti két objektum példányról, hogy egyenlők –e.

| | |
|----------------------|--|
| GetContextMenu | Visszaadja a ContextMenu-t, ami tartalmazza ezt a menüt. |
| GetMainMenu | Visszaadja a MainMenu-t, ami tartalmazza ezt a menüt. |
| GetType | Visszadaja az aktuális példány típusát. |
| MergeMenu | Egyesíti egy menü MenuItem objektumait az aktuális menüével. |
| Show | A menüt egy megadott pozícióban jeleníti meg. |
| ToString | Az objektumot String formára alakítja. |
| Public események | |
| Disposed | Egy eseménykezelőt ad, ami figyeli a komponens Disposed eseményét. |
| Popup | Bekövetkezik mielőtt a menü megjelenítésre kerül. |
| Protected property-k | |
| DesignMode | Egy értéket ad vissza, ami mutatja, hogy a komponens design módban van-e. |
| Events | A komponenshez tartozó eseménykezelők listáját adja vissza. |
| Protected metódusok | |
| CloneMenu | Paraméterként lemásolja a menüt az aktuális menübe. |
| Dispose | Elengedi a komponens által használt forrásokat. |
| Finalize | Felszabadítja a nem használt eszközöket forrásokat a garbage collection előtt. |

ContextMenu hozzáadása a Windows Formhoz:

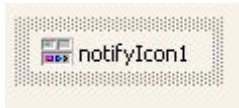
Nyissuk meg a Windows Form Designerben azt a formot amihez hozzá akarjuk adni a ContextMenu-t.

A Toolbox-ban kattintsunk kettőt a ContextMenu komponensre, ami ekkor a komponens tálcához adódik.

A ContextMenu-t a formhoz vagy kontrollokhoz rendelhetjük a Properties ablakban az objektumok ContextMenu property-jének beállításával.

NotifyIcon

NotifyIcon-ról általában:



A System tray-en látható ikonok, mutatják, milyen programok futnak a háttérben. Ilyenek például a különböző vírusirtó programok, a hangerőszabályzó stb... Ezeken az ikonokon keresztül férhetünk ezen programok felhasználói interfészéhez. A NotifyIcon osztály ezt a funkcionalitást kínálja a programok számára. Az Icon property definiálja, hogy milyen ikon jelenjen meg a System tray-en. Azt, hogy milyen felugró menü tartozzon hozzá, a ContextMenu property segítségével állíthatjuk be. A Text tulajdonság pedig akkor jelenik meg ha az egér mutatóját az ikon fölé időztetjük.

NotifyIcon részei:

Public Konstruktor

NotifyIcon Constructor

A NotifyIcon őssztály egy új példányát inicializálja.

Public Property-k

Container

Visszaadja az Icontainer-t ami tartalmazza a komponenst.

ContextMenu

ContextMenu beállítása, lekérdezése.

Icon

Icon beállítása, lekérdezése.

Text

Beállítható vagy lekérdezhető a tooltip szövege, ami akkor jelenik meg, ha az egérmutatót a system trayen lévő ikon fölé helyezzük.

Visible

NotifyIcon láthatóságának ki- és bekapcsolása.

Public Események

Click

Akkor következik be ha az ikonra kattintunk a system tray-en.

| | |
|-------------|--|
| Disposed | Egy eseménykezelőt ad, ami figyeli a komponens Disposed eseményét. |
| DoubleClick | Akkor következik be ha az ikonra duplán kattintunk a system tray-en. |
| MouseDown | Akkor következik be ha a system tray-en az ikon fölött lenyomjuk az egér gombját. |
| MouseMove | Akkor következik be ha a system tray-en az ikon fölött mozgatjuk az egér mutatóját. |
| MouseUp | Akkor következik be ha a system tray-en az ikon fölött elengedjük az egér balgombját, vagy újra megnyomjuk a jobb gombját. |

StatusBar

StatusBar-ról általában:

A StatusBar kontrol StatusBarPanel objektumokat tartalmaz. Ezeken a paneleken szövegek és/vagy ikonok találhatóak. A StatusBar kontrol tipikusan arra szolgál, hogy a formon található objektumokról információt szolgáltatson. A StatusBar-on helyet kap még a SizingGrip, amelynek segítségével a form méreteit állíthatjuk be.



StatusBar részei:

Public Konstruktor

StatusBar Constructor

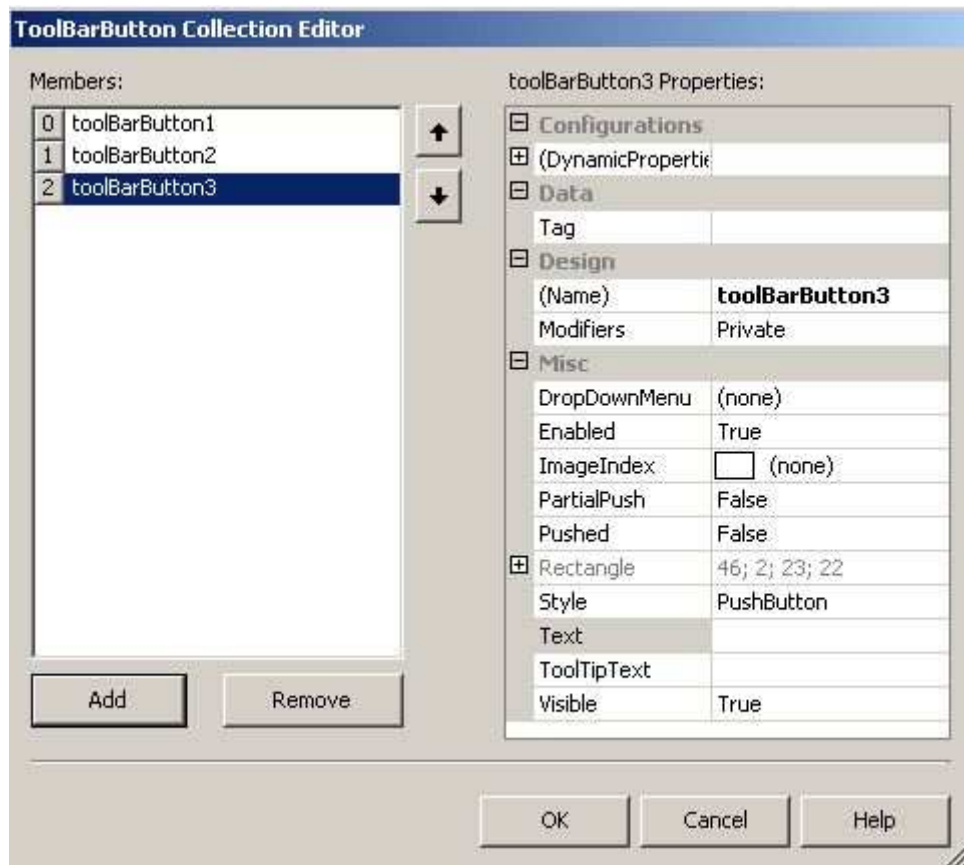
A StatusBar osztály egy új példányát inicializálja.

Public Property-k

| | |
|------------------|---|
| Name | Az objektum neve. |
| AllowDrop | Beállítható, ill. lekérdezhető az értéke annak, hogy a kontrol elfogadja –e az adatot, amit a felhasználó rádobott. |
| ContextMenu | ContextMenu beállítása, lekérdezése. |
| Cursor | Beállítható, hogy az egérmutató milyen legyen amikor a ToolBar felé ér. |
| Enabled | Ezzel lehet a StatusBar-t ki, be kapcsolni. |
| Font | Betűtípus beállítása. |
| Location | A pozíció beállítása. |
| Panels | A StatusBarPanelok collection-ja amiket a StatusBar tartalmaz. |
| ShowPanels | A panelok ki-, bekapcsolása a StatusBar-on. |
| Size | StatusBar méreteinek beállítása. |
| SizingGrip | A SizingGrip ki-, bekapcsolása (melynek segítségével a form méretei állíthatóak) |
| Text | A StatusBar-on látható szöveg |
| Visible | StatusBar láthatóságának ki-, bekapcsolása. |
| Public Események | |
| Click | Mikor a kontrolra kattintunk. |
| DragDrop | Drag-and-drop esemény bekövetkezésekor. |
| PanelClick | Amikor valamelyik panelra kattintunk. |

ToolBar

A ToolBar-ról általában:



A ToolBar kontrollokat arra használjuk, hogy ToolBarButton-okat jelenítsünk meg a képernyőn, amik standard, toggle, illetve drop-down stílusú gombok lehetnek. A gombokra képeket rakhatunk és különböző stílusúra állíthatjuk őket. A ToolBar-t úgy

használjuk, mint egy menüt. A gombok menüpontoknak felelnek meg, csak mindig szem előtt vannak, így gyorsítva a munkát.

A Toolbar részei:

Public konstruktor

ToolBar Constructor

A ToolBar osztály egy új példányát inicializálja.

Public Property-k

AllowDrop

Beállítható, ill. lekérdezhető az értéke annak, hogy a kontrol elfogadja –e az adatot, amit a felhasználó rádobott.

Appearance

A ToolBar kontrol és a gombjai megjelenítésének beállítása és lekérdezése.

AutoSize

Automatikus méretezés beállítása és lekérdezése.

BorderStyle

Szegély stílusának beállítása.

Buttons

A ToolBar-on megjelenő ToolBarButtonok gyűjteménye.

ButtonSize

A ToolBaron lévő gombok méretének beállítása, lekérdezése.

ContextMenu

ContextMenu beállítása, lekérdezése.

Cursor

Beállítható, hogy az egérmutató milyen legyen amikor a ToolBar felé ér.

Dock

Beállítható, hogy a ToolBar hol dokkoljon.

Enabled

Ezzel lehet a ToolBar-t ki, be kapcsolni.

Font

Betűtípus beállítása.

ImageList

Beállítható, hogy a gombok képei melyik ImageList-ből legyenek elérhetőek.

Location

A ToolBar pozíciójának beállítása.

Name

A Toolbar neve.

Size

A Toolbar méretének beállítása.

| | |
|------------------|--|
| Visible | Látható, nem látható. |
| Public metódusok | |
| DoDragDrop | Drag-and-drop művelet engedélyezése. |
| FindForm | Visszadja a formot, amin a kontrol rajta van. |
| Public események | |
| ButtonClick | Akkor következik be ha a ToolBaron egy ToolBarButton-ra kattintunk. |
| ButtonDropDown | Akkor következik be ha egy drop-down stílusú gombra vagy a nyílára kattintunk. |
| DragDrop | Drag-and-drop esemény bekövetkezésekor. |

Formok használata, formok típusai

Formok jellemzői:

(Name) A form azonosítója, a programon belül hivatkozhatunk a formra ezzel. Ne felejtsük módosítani a forráskódban is! Nem automatikus.

```
static void Main()
{
    Application.Run(new frmMain());
}
```

BackColor Háttérszín

BackgroundImage Háttérkép, kiválasztható (jpg, png, bmp ,....)

Font A form és a formon lévő kontrollok default fontbeállítása

FormBorderStyle A form keretének stílusa: default: sizable
None, FixedSingle; Fixed3D; FixedDialog; FixedToolWindow;
SizableToolWindow

Buttons: A formra elhelyezett gombok alapértelmezett jelentése lehet

AcceptButton;

CancelButton;

HelpButton

Menu

MainMenu obj. Neve.

Opacity

Átlátszóság, 100% az átlátszatlan, minnél kisebb a %, annál jobban átlátható a form

ShowInTaskbar Default: true; ha az értéke false, akkor nem jelenik meg a taskbaron a program.

Size (width, height) A form mérete pixelben

StartPosition A form megjelenéskor hol helyezkedjen el:

WindowsDefaultLocation

Manual

CenterScreen

CenterParent

WindowsDefaultBounds (nem ajánlott)

Text

A form fejlécében megjelenő szöveg

WindowState

Form állapota

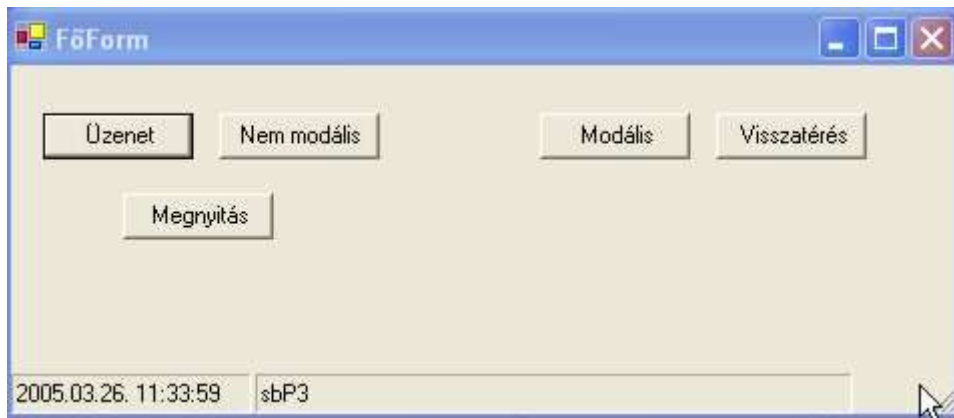
Normal, Minimized, maximized

A rendszer által biztosított üzenetablakok használata

A MessageBox osztály segítségével tudunk egy üzenetablakot megjeleníteni a képernyőn. Ez tartalmazhat szöveget, nyomógombokat, és képi szimbólumot. Ha nyomógombokat is tartalmaz, akkor a visszatérési értéke egy DialogResult osztály példánya, melynek a segítségével meghatározhatjuk, hogy a felhasználó mely nyomógombra való kattintással zárta az üzenetablakot.

Az üzenetablak megjelenítését a Show metódus végzi, melynek paraméterei határozzák meg a megjelenést.

A példában a Show metódusnál kihasználtuk ezeket a paramétereket, de lehetőségünk van elhagyni is belőle. A példaprogram Üzenet gombja bemutatja az alkalmazás módját.



A megjelenítése:

```
DialogResult dr = new DialogResult();  
dr = MessageBox.Show("ez az üzenet jelenik meg az ablakban",  
                    "Ez a form fejléce", MessageBoxButtons.YesNoCancel,  
                    MessageBoxIcon.Question);
```

Az első sorban létrehozunk a **DialogResult** osztályból egy példányt. Ennek a neve **dr**. Majd a **MessageBox.Show** metódushívással megjelenítjük az üzenetablakot. Az első paramétere a megjelenítendő szöveg, string konstans, vagy string típusú változó.

A második paramétere szintén string, ez lesz a megjelenő form fejlécének felirata. A harmadik paraméter határozza meg, hogy milyen nyomógombok jelenjenek meg a formon. A **MessageBoxButtons** egy felsorolt típus, melynek a lehetséges értékei:

AbortRetryIgnore

Ok

OKCancel

RetryCancel

YesNo

YesNoCancel

Ezek közül választhatunk a gombok kiválasztásánál.

A negyedik paraméter a képi szimbólumot határozza meg. A **MessageBoxIcon** segítségével az alábbi képecskéket választhatjuk:

Asterisk



Error



Exclamation



Hand



Information



None

Question



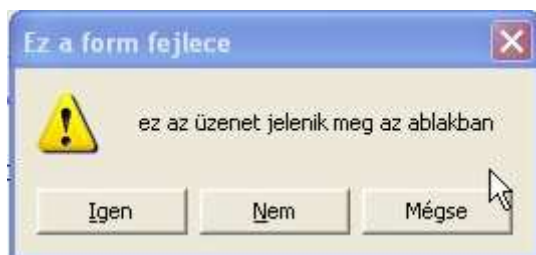
Stop



Warning



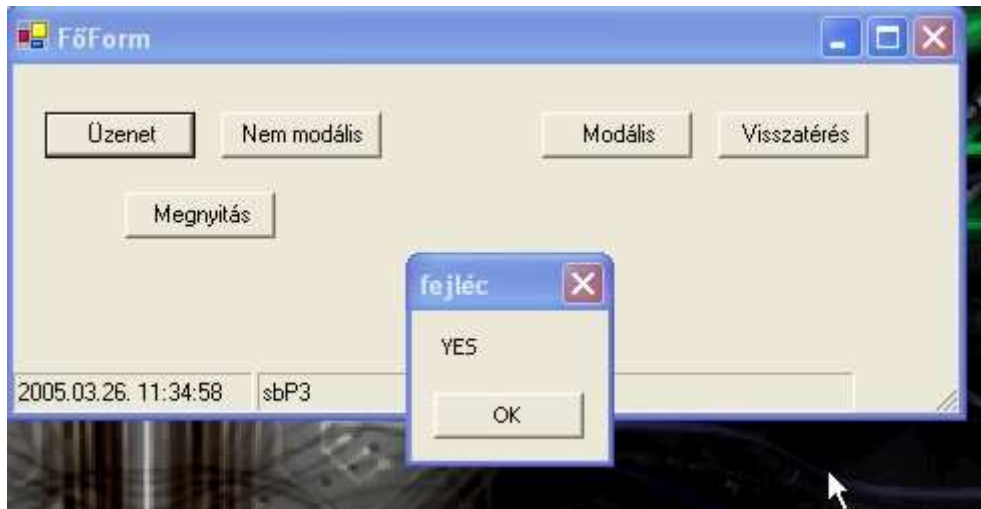
Ezek után az üzenetablak így jelenik meg:



Ezután egy egyszerű switch szerkezettel el lehet dönteni, hogy a felhasználó melyik nyomógombot választotta.

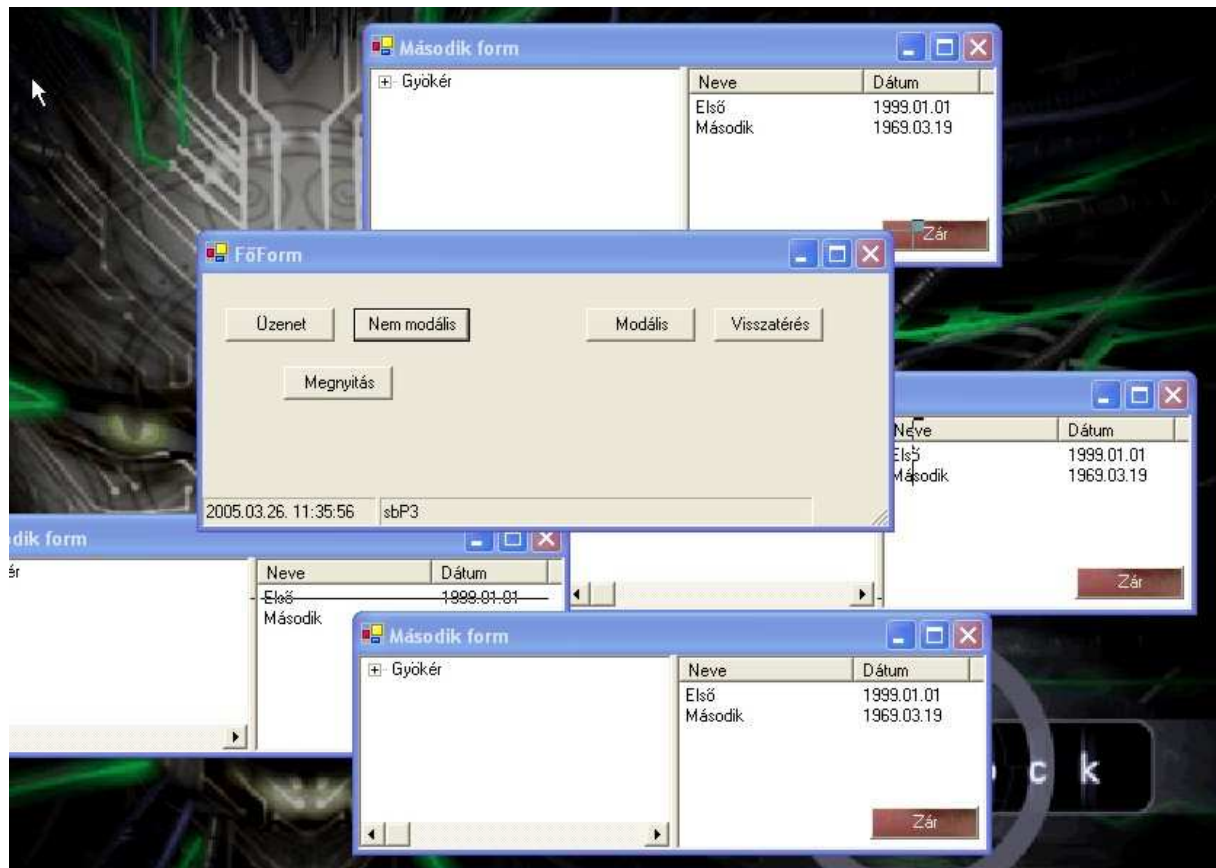
```
switch (dr)
{
    case DialogResult.Yes:
    {
        MessageBox.Show("YES", "fejléc", MessageBoxButtons.OK);
        break;
    }
    case DialogResult.No:
    {
        MessageBox.Show("NO", "fejléc", MessageBoxButtons.OK);
        break;
    }
    case DialogResult.Cancel:
    {
        MessageBox.Show("CANCEL", "fejléc", MessageBoxButtons.OK);
        break;
    }
}
```

Ebben a példában látható, hogy a MessageBox-nak lehetséges 3 paramétere is, azaz elhagyható az icon meghatározás.



Modális és nem modális formok

A saját formok megjelenítésénél két lehetőség között választhatunk. A **Show()** metódus megjeleníti a formot, a **visible** tulajdonságának true-ra állításával. A form megjelenik, de nem birtokolja a fókust kizárólagosan, hanem akár más form, akár a hívó form újra aktivizálható. Így ha nem ellenőrizzük programból, akár több ugyanolyan form is megjeleníthető. Erre látunk egy példát az következő képen.

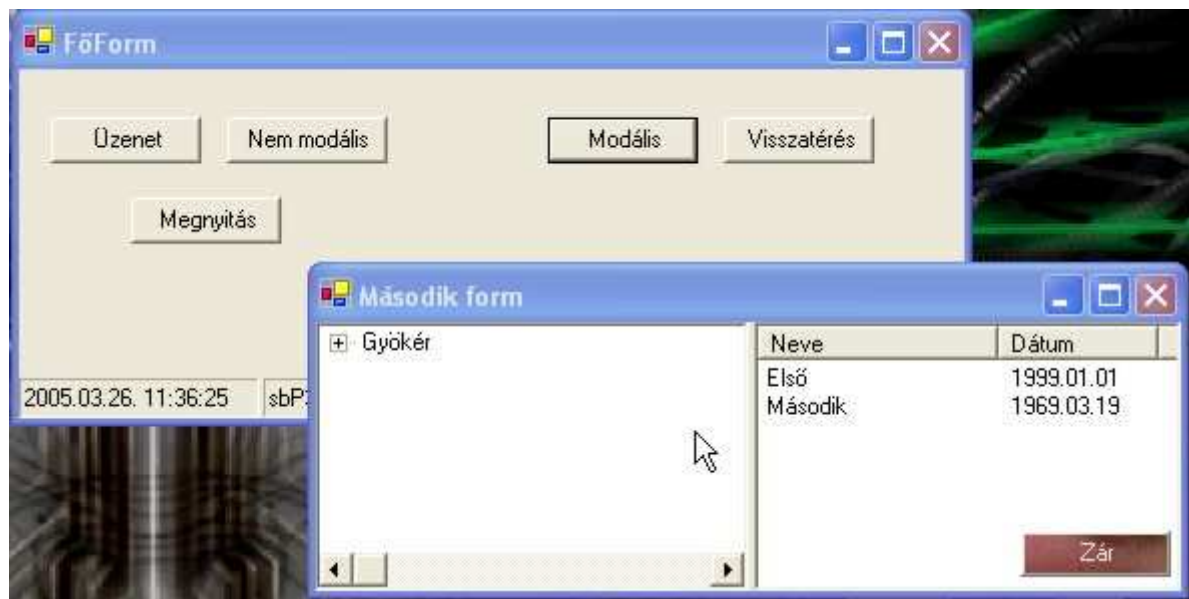


A példaprogramban a 'Nem modális' gombra kattintva érhetjük ezt el.

```
frmKetto frmK = new frmKetto();
frmK.Show();
```

Példányosítás után megjelenítjük a formunkat.

Ha a másik lehetőséget választjuk, akkor a **ShowModal()** metódust kell használnunk. Ekkor a formunk egy dialógus boxként jelenik meg, amelyik kizárólagosan birtokolja a fókuszot.



Nem aktiválhatjuk az őt hívó formot, vagy más, a programhoz tartozó, és látható formot sem. Ennek a metódusnak van visszatérési értéke, ami egy **DialogResult** példány. Ennek felhasználásával kiértékelhetjük, hogy melyik nyomógomb segítségével zárta be az ablakot a felhasználó.

A megjelenítésre példa, ha a 'Modális' gombra kattintunk.

```
frmKetto frmK = new frmKetto();
frmK.ShowDialog();
```

Ha szeretnénk használni a kiértékelés adta lehetőséget is, akkor a megjelenő formon lévő gombok tulajdonságai közül a DialogResult értéket állítsuk be a nekünk megfelelőre, majd a hívó helyen a megjelenítés köré szervezzünk egy vizsgálatot.

```
frmKetto frmK = new frmKetto();
DialogResult dr = new DialogResult();
dr = frmK.ShowDialog(this);
if (dr == DialogResult.Cancel)
    MessageBox.Show("Mégse gombbal zártad", "fejléc", MessageBoxButtons.OK);
else
    if (dr == DialogResult.OK)
        MessageBox.Show("Rendben gombbal zártad", "fejléc", MessageBoxButtons.OK);
```



Dialógusok

A különböző párbeszéd ablakoknak, dialógusoknak fontos szerepe van a Windows alatt futó programok készítésénél, mivel segítségükkel tudunk a felhasználóval kommunikálni, hibaüzeneteket generálni, vagy beállításokat, paramétereket kérni tőle.

A dialógusok megtalálhatóak a komponens palettán, de elő lehet állítani őket dinamikusan, valamely megfelelő osztályból történő származtatással is.

A fontDialog

A FontDialog elindítása és a fontkészlet kiválasztása a következő módon történik: a fontDialog.ShowDialog() eseményével tudjuk aktivizálni azt a párbeszéd panelt, melyen a felhasználói beállítások elvégezhetők, majd a megfelelő komponens fontkészletéhez hozzárendelhetjük a felhasználó választásának eredményét, vagyis a kívánt font típust.

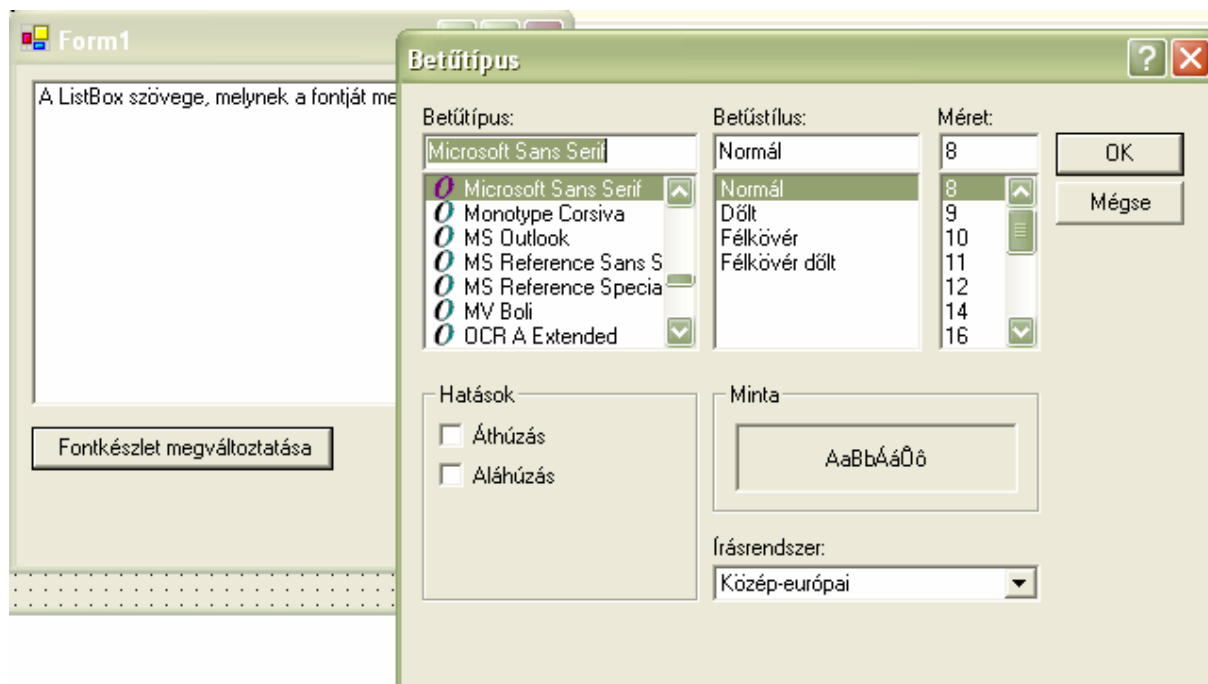
```
private void button1_Click(object sender, System.EventArgs e)
{
    fontDialog1.ShowDialog();
    listBox1.Font=fontDialog1.Font;
}
```

Az eljárás hatására a listBox-ban tárolt szöveg fontja a fontDialog-ban kiválasztott font tulajdonságait kapja meg.





Amennyiben programot szeretnénk készíteni a fontDialog komponens kipróbálására, helyezzünk a Form-ra egy listBox-ot, egy fontDialog-ot és egy nyomógombot (button)!


A listBox Items tulajdonságánál gépeljük tetszőleges szöveget a listBox-ba, a nyomógomb eseménykezelőjébe pedig a fenti programrészletet írjuk! (Csak a függvényben lévő utasításokat gépeljük be, mivel a nyomógombhoz tartozó függvényt a .NET elhelyezi a kódban!)

A futtatáskor a következő kimenet látható:



Az C# nyelv vizuális részében található többi dialógus is hasonló módon működik. Az alábbi táblázatban összefoglaltuk a fontosabbakat.

| | | |
|----------------------|----------------|---|
| Színek kiválasztása | colorDialog |  colorDialog1 |
| Font megváltoztatása | fontDialog |  fontDialog1 |
| Nyomtatási kép | printDialog |  printDialog1 |
| Fájl megnyitása | openFileDialog |  openFileDialog1 |

| | | |
|--------------|----------------|---|
| Fájl mentése | saveFileDialog |  saveFileDialog1 |
|--------------|----------------|---|

ColorDialog

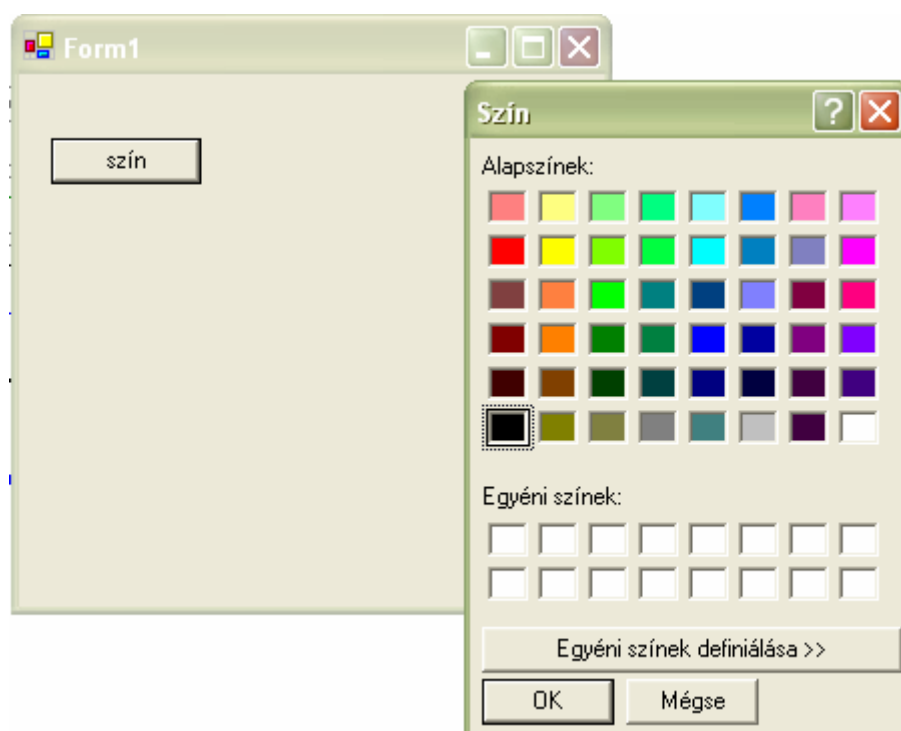
Ez a komponens alkalmas a különböző hátterek, felületek színének kiválasztására. (Természetesen a kiválasztott színt végül nekünk kell hozzárendelni az adott komponenshez.)

A következő forráskód megmutatja a colorDialog használatát:

```
private void button2_Click(object sender, System.EventArgs e)
{
    colorDialog1.ShowDialog();
}
```

Mint láthatjuk, ez a komponens is hasonlóan működik, mint az előző, a FontDialog, csak míg az a fontok, ez a színek kiválasztására alkalmas. Amennyiben ki akarjuk próbálni ezt a komponenst is, a programunkat az előző példa alapján építsük fel!

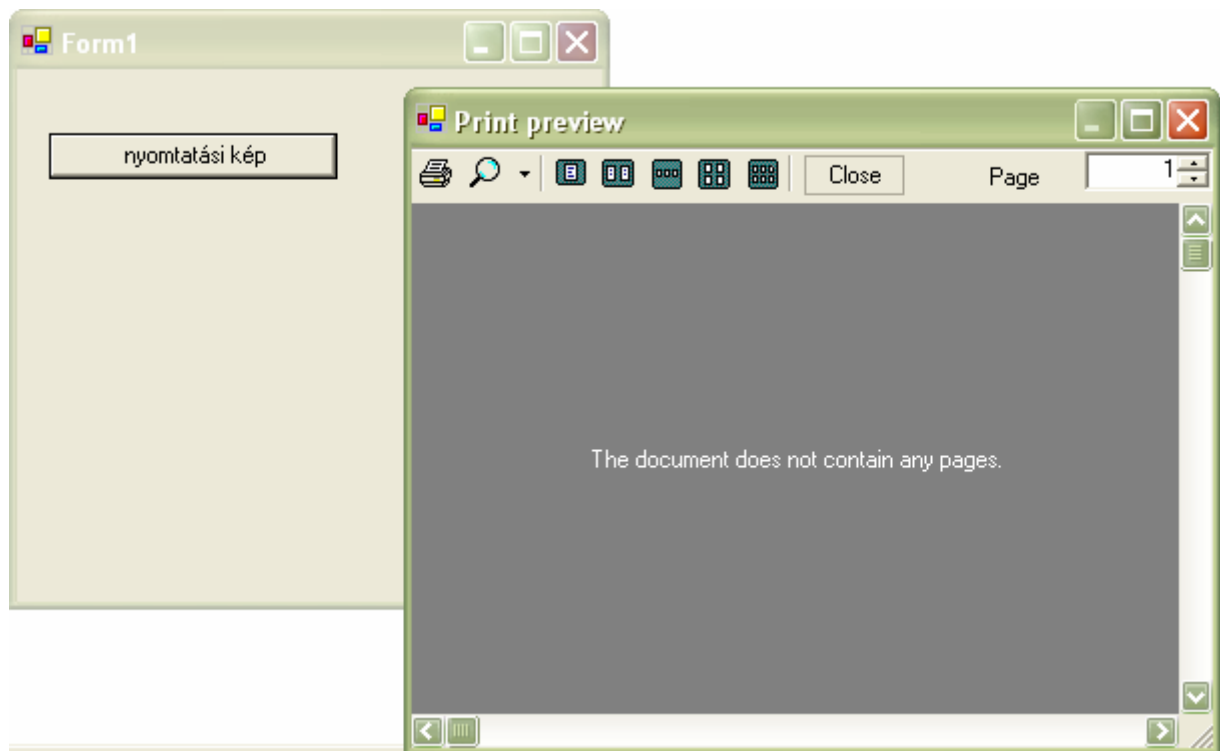
Az így megírt program kimenete a következő:



PrintPreviewDialog

Sokszor előfordul, hogy a programjainkból nyomtatóra szeretnénk küldeni bizonyos adatokat. Ilyen esetekben a nyomtatási képet nem árt megvizsgálni, s a felhasználónak megmutatni a tényleges nyomtatás előtt.

A nyomtatási kép megjelenítéséhez a printPreviewDialog komponenst használhatjuk, mely hasonló pl.: a Word ”nyomtatási kép” ablakához.



A párbeszéd ablakot úgy aktivizáltuk, ahogyan az előzőeket:

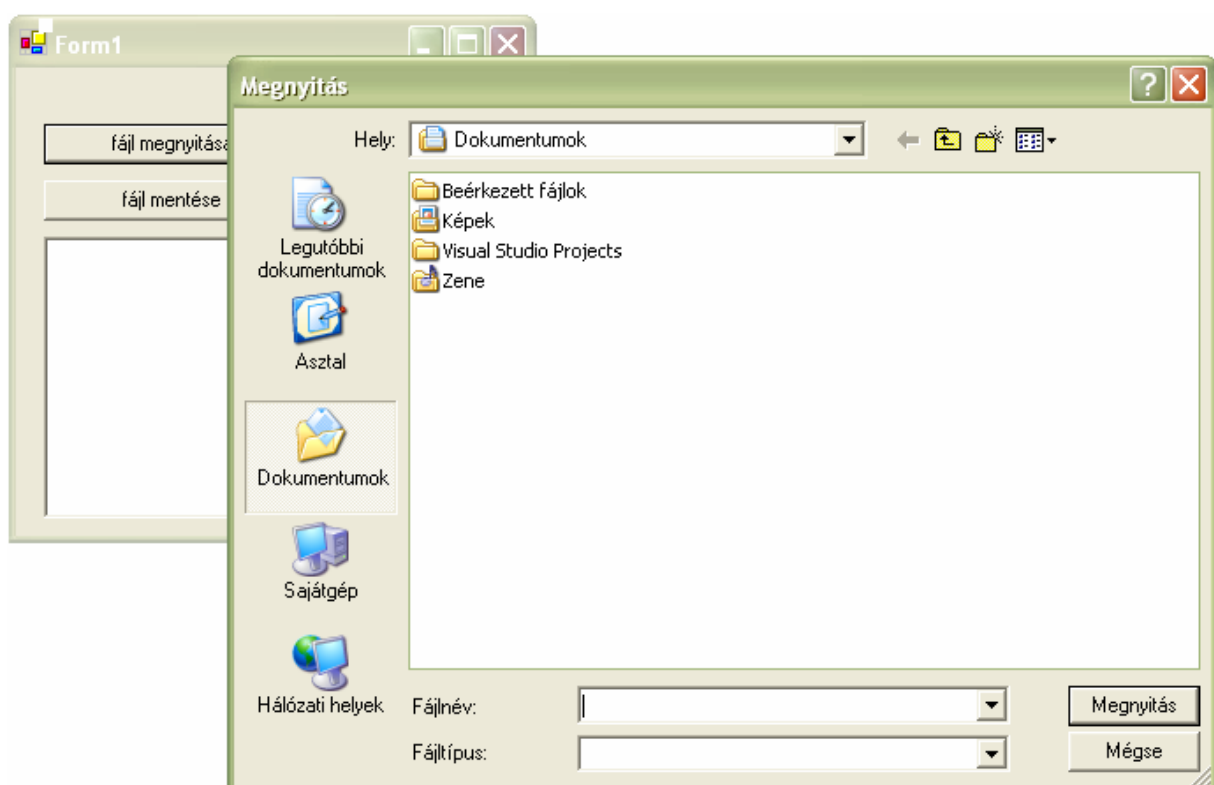
```
private void button2_Click(object sender, System.EventArgs e)
{
    printPreviewDialog1.ShowDialog();
}
```

A nyomtatás kezelése igényel még néhány sornyi forráskódot, de a megírása nem tartozik szorosan a fejezethez.

Fájl megnyitása és mentése

Az openFileDialog és a saveFileDialog a fájlok megnyitását és kimentését teszik egyszerűvé, mivel a Windows-ban megszokott fájlkezelő ablakokat aktivizálják, s támogatják a fájlkezeléshez szükséges metódusokkal.

Ez a két komponens is könnyen használható. Csak annyi a dolgunk velük, hogy meghívjuk a showDialog() függvényüket, majd a kiválasztott fájlt (fájlokat) a megfelelő metódusok segítségével elmentjük, vagy éppen betöltjük.



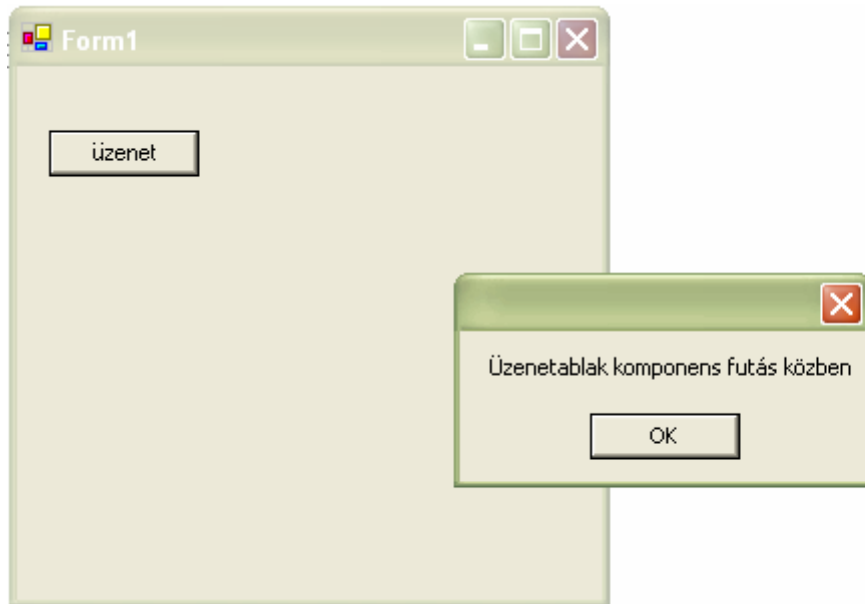
MessageBox

Fontos még megemlíteni a MessageBox osztályt is, melynek metódusaival különböző párbeszéd paneleket tudunk megjeleníteni a képernyőn, segítve ezzel a kommunikációt a felhasználóval.

A következő forráskód részlet bemutatja a MsgBox használatát:

```
MessageBox.Show("Üzenetablak komponens futás közben");
```

A MessageBox osztály Show() metódusában tudjuk elhelyezni azt az üzenetet, amelyet meg szeretnénk jeleníteni. Ha a fenti kódrészletet egy nyomógomb eseménykezelőjében helyezzük el, majd futtatjuk, az alábbi kimenethez jutunk:



Feladatok

1. Készítsünk programot, amely lehetővé teszi, hogy egy listBox-ba írt szöveg betűtípusát és a listBox háttérszínét a felhasználó meg tudja változtatni. Használjuk a fontDialog és a colorDialog komponenseket!
2. Készítsünk programot, melyben a Form háttérszínét dialógus segítségével meg lehet változtatni!
3. Készítsünk programot, mely a printPreviewDialog ablakot aktivizálja!

Többszálú programozás

A többszálú programozás viszonylag új technika, segítségével lehetőség nyílik a processzor hatékonyabb kihasználására, több processzoros rendszerek programozására. A több szálát használó alkalmazások fejlesztésének lehetősége nem a .NET újítása, de hatékony támogatást ad azok készítéséhez.

A párhuzamos, vagy többszálú programok fogalma megtévesztő lehet. Egy processzor esetén nem fejezik be futásukat hamarabb, és a szálak külön – külön sem futnak rövidebb ideig, viszont több processzor esetén a futási idő radikálisan csökkenhet.

Egy processzoros rendszerekben is előnyt jelenthet a többszálú programok írása, futtatása. Előfordulhat, hogy több program, vagy egy program több szála fut egy időben, s mialatt az egyik szál adatokra vár, vagy éppen egy lassabb perifériával, eszközzel kommunikál, a többi szál használhatja az értékes processzoridőt. Természetesen az egyes szálakat valamilyen algoritmus ütemezi, s azok szeletei egymás után futnak és nem egy időben.

A .NET rendszerben akkor is szálkezelést valósítunk meg, amikor egyszerű programokat írunk. A futó alkalmazásunk is egy programszál, ami egymagában fut, használja a különböző erőforrásokat és verseng a processzoridőért.

Amennyiben tudatosan hozunk létre többszálú programokat gondoskodni kell arról, hogy a szálkezelő elindítsa és vezérelje a szálakat. A .NET a szálkezelőnek átadott szálakat ütemezi, és megpróbálja a számukat is optimalizálni, nagyban segítve ezzel a programozó dolgát. A szálakat természetesen magunk is elindíthatjuk, még a számukat is megadhatjuk, de mindenképpen jobb ezt a rendszerre bízni.

(Az következőekben ismertetett programhoz a Microsoft által kiadott "David S Platt: Bemutatkozik a Microsoft .NET" című könyvben publikált Párhuzamos Kódokat bemutató fejezetéből vettük az alapötletet, majd ezt fejlesztettük tovább és írtuk át C# nyelvre, mert talán így tudjuk a legegyszerűbben bemutatni a több szálú

programokat. Az említett könyv olvasása javasolt annak, aki a .NET rendszer használatát és finomságait teljes mértékben meg akarja ismerni!)

A példaprogram egy listView komponens segítségével bemutatja, hogyan futnak egymás mellett a szálak, a szálakat egy ImageList-ben tárolt képsorozat elemeivel szimbolizálja. A programban egy időben több szál is fut, de a processzor egy időben csak eggyel foglalkozik, viszont cserélgeti a futó szálakat.

A Piros ikonok az aktív szakaszt mutatják, a sárgák az aktív szakasz utáni alvó periódust, a kék ikonok pedig a még feldolgozásra váró, és a már nem futó szálakat szimbolizálják. A programban szerepel egy célfüggvény, mely a szálakhoz rendelt eseményt hajtja végre. Az egyszerűség kedvéért a program célfüggvénye csak a szál ikonját és feliratát cserélgeti, ezzel jelezve, hogy egy valós, párhuzamos program ezen részében adatfeldolgozás, vagy egyéb fontos esemény történik.

Az egyszerre futó szálak számát nem a programozó határozza meg, hanem a .NET keretrendszer az erőforrások függvényében.

Induláskor közöljük a programmal a futtatni kívánt szálak számát, majd átadjuk a célfüggvényt a `System.Threading.ThreadPool.QueueUserWorkItem()` metódusnak, mely feldolgozza ezeket.

A célfüggvény mellett adhatunk a metódusnak egy tetszőleges objektumot, melyben paramétereket közölünk vele, ami átadja ezeket a célfüggvénynek.

A példában az egy mezőt tartalmazó Parameterek osztályt adunk át a paraméter listában, melyben az adott elem sorszámát közöljük a célfüggvénnyel.

```
class Parameterek
{
    public int i = 0;
    public String s = "Szál";
}
```

A program indulása után a Form- ra helyezett nyomógomb indítja el a szálkezelést. A nyomógomb eseménykezelőjében, a FOR ciklusban példányosítjuk a Parameterek osztályt (minden szálhoz egyet):

```
Parameterek P = new Parameterek();
```

Hozzáadjuk a soron következő elemet a listView komponenshez, beállítjuk az elem ikonját és feliratát:

```
listView1.Items.Add(Convert.ToString(i)+"Szál",2);
```

Értékül adjuk a ciklus változóját az osztály i változójának, ezzel átadva az adott szál sorszámát a szálkezelőnek, s közvetett úton a célfüggvénynek:

```
P.i=i;
```

Végül átadjuk a példányt a célfüggvénnyel együtt a szálkezelőnek, mely gondoskodik a szál sorsáról. Ütemezi, futtatja, majd leállítja azt.

```
System.Threading.WaitCallback CF = new  
System.Threading.WaitCallback(CelF);  
System.Threading.ThreadPool.QueueUserWorkItem(CF , P);
```

A célfüggvény a következő dolgokat végzi el, szimulálva egy valós program-szál működését:

1. Beállítja a szál ikonját a piros, „futó” ikonra, (Az ikonokat egy imageList- ből veszi), azután átírja az ikon feliratát:

```
listView1.Items[atadot.i].ImageIndex=0;  
listView1.Items[atadot.i].Text="Futó szál";
```

2. A `System.Threading.Thread.Sleep(1000)` metódussal a szálkezelő várakoztatja a szálát, majd átállítja a szálhoz tartozó ikont a sárga, „alvó” ikonra, a feliratát pedig „Alvó szál” - ra:

```
listView1.Items[atadot.i].ImageIndex=1;  
listView1.Items[atadot.i].Text="Alvó szál";  
System.Threading.Thread.Sleep(1000);
```

3. A szál ismét várakozik, majd az ikonja az eredetire vált:

```
listView1.Items[atadot.i].ImageIndex=2;  
listView1.Items[atadot.i].Text="Vége");  
System.Threading.Thread.Sleep(1000);
```

Mivel a szálak létrehozását ciklusba szerveztük, a program annyi szálát hoz létre, mint a ciklus lépésszáma, viszont azt, hogy egyszerre hány szál fut, a szálkezelő dönti el.

A listView ablakában látszik, hogy egyszerre akár három, vagy négy szál is futhat, és mellettük néhány szál éppen alszik, de lassabb gépeken ezek a számok növekedhetnek.

Sajnos a többszálú programozás közel sem ilyen egyszerű. Bonyolultabb alkalmazások esetén egy probléma mindenképpen felvetődik. A futó szálak nem használják egy időben ugyanazt az erőforrást, vagyis biztonságosan kell futniuk egymás mellett. A jól megírt többszálú programok optimálisan használják fel az osztott erőforrásokat, de csak akkor használjuk fel a lehetőségeiket, ha tisztában vagyunk a használatukban rejlő előnyökkel, és a lehetséges hibákkal is. A rosszul megírt párhuzamos működésű programokkal éppen az ellenkezőjét érjük el annak, amit szeretnénk. Növeljük a lépésszámot, a memória foglaltságát és a programok hatékonyságát.

Feladatok

1. A fejezetben szereplő forráskódok és azok magyarázata alapján állítsa elő a tárgyalt programot. A grafikus felületnek tartalmaznia kell a következő komponenseket: listView, imageList - az ikonok rajzával (ezeket a Paint programmal rajzolja meg), és egy nyomógombot a folyamat elindításához.
2. Készítsen programot, melynek grafikus felületén két progressBar komponens két futó programszál állapotát szimbolizálja! A Form-on egy nyomógomb segítségével lehessen elindítani a szimulációt. A progressBar állapota mutassa az aktuális szál feldolgozottságát!

Programozás tankönyv

XVI. Fejezet

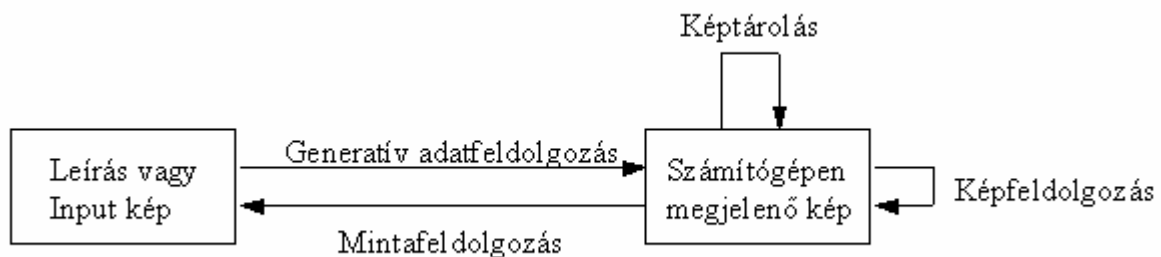
„Grafikai alapok!”

Dr. Kovács Emőd

A grafikus szoftver

A számítógépi grafika (Computer Graphics) természetesen képekkel foglalkozik. Feladata a grafikus adatok feldolgozása amely négy, nem teljesen elkülönülő főterületet ölel fel:

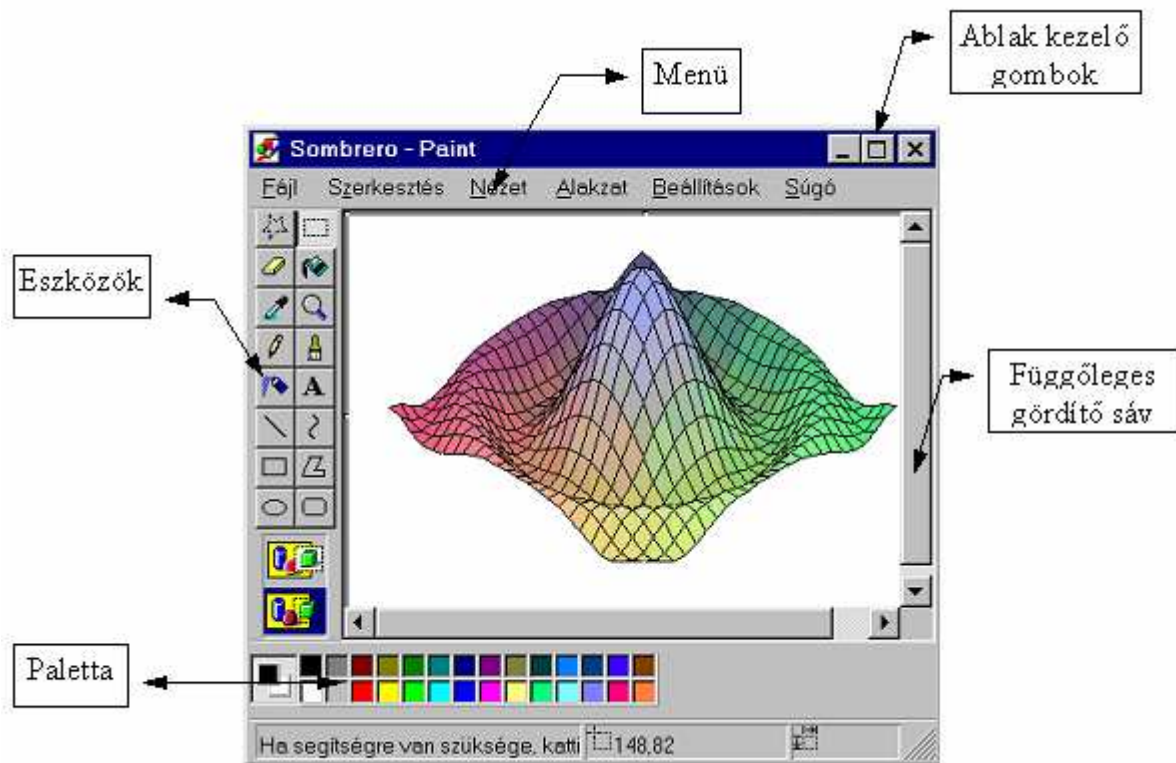
- Generatív grafikus adatfeldolgozás: képek és grafikus adatok bevitele, képek előállítása, manipulálása és rajzolása leírások alapján. Síkbeli és térbeli objektumok modellezése.
- Grafikus képek tárolása a számítógépen és adathordozókon kódolt formában.
- Képfeldolgozás: képek javítása átalakítása a későbbi feldolgozás érdekében.
- Mintafelismerés: Képekből meghatározott információk kiolvasása, leírások és adatok előállítása.



A számítógépi grafika feladatai

A számítógépi grafika önmagában nem csak a képekkel foglalkozik, hanem számtalan alkalmazási területe van a grafikus adatok feldolgozásának is. A következő felsorolásban kísérletet teszünk arra, hogy a számítógépi grafika témakörébe tartozó alkalmazásokat csoportosítsuk. A csoportosításnál az volt a szempontunk, hogy az alkalmazások milyen feladatcsoportot ölelnek fel és a használt módszerekben milyen összefüggések találhatók.

- Művészet és animáció (Art and Animation)
- Számítógéppel segített tervezés és gyártás: Computer Aided Design (CAD), Computer Aided Manufacturing (CAM)
- Bemutató és üzleti grafika (Presentation and Business Graphics)
- Tudományos és műszaki szemléltetés és szimuláció (Scientific Visualization and Simulation)
- Képelemzés és feldolgozás (Image analysis and processing)
- Grafikus kezelő felületek (Graphics User Interfaces, GUI)
- Virtuális valóság (Virtual Reality)
- Multimédia alkalmazások (Multimedia Application)
- Számítógépes játékok (Computer Games)



A Microsoft Paint programablakja

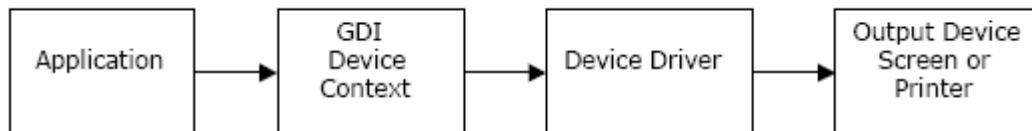


Játékra fel!

Egy komputergrafikai program a grafikus hardver szolgáltatásait grafikus könyvtárak segítségével éri el. Számos grafikus könyvtár létezik, amelyek egy része az operációs rendszerhez kötött, más része független. A továbbiakban a C#-hoz kapcsolódó Ms-Windows GDI+ lehetőségeit tárgyaljuk

GDI+

A GDI (Graphics Device Interface) biztosítja az ablakos felületet használó grafikus alkalmazásoknak az eszköz független elkészítését. Az így elkészült programok például képesek futni különböző képernyő felbontásokban, használni tudnak különböző nyomtatókat, legyen az mátrix vagy lézer nyomtató. A GDI réteg alkalmazására tekintsük meg a következő ábrát:



A GDI alkalmazása

A fenti ábrán jól látható hogy a GDI egy olyan réteg, amely biztosítja a kapcsolatot a különböző hardver eszközök és az alkalmazások között. Ez a struktúra megszabadítja a programozót a különböző hardver eszközök közvetlen programozásának fáradságos munkájától. A GDI egy program fájl, amit a számítógépen tárolunk, s az ablak alapú (Windows) alkalmazás/környezet tölti be a memóriába, amikor egy grafikus output miatt szükség van rá. A Windows betölti a szükséges eszközekezelő programot (Device Driver) is, amely elvégzi a grafikus parancs konvertálását a hardver eszköz számára érhető formába, feltéve, ha erre a GDI közvetlenül nem képes.

Device Context: alapvető eszköz, amely valójában egy olyan belső struktúra, amelyet a Windows arra használ, hogy kezelje az output eszközt.

GDI: C++ osztályok halmaza, amelyek biztosítják a grafikus adatok eljuttatását a programokból a hardverek felé a Device Driver segítségével. A .NET rendszer GDI változata a GDI+.

A GDI+ egy új fejlődési pontja a GDI változatoknak. A GDI+ –szemben a GDI-vel– egy jóval magasabb szintű programozást tesz lehetővé, a programozóknak már nem kell semmit sem tudniuk a hardver eszközökről.

A könyvünknek nem célja további részletességgel tárgyalni a GDI+ technikai leírását, csak a benne rejlő lehetőségek izgalmasak számunkra.

Amellett, hogy a GDI+ API (Application Programming Interface) rugalmasabb és teljesebb mint a GDI, nagyon sok újdonságot is megvalósítottak benne.

GDI+ osztály és interfész a .NET-ben

Névterek, namespaces

A Microsoft .NET könyvtárban minden osztályt névterekben (namespace) csoportosítottak. A namespace nem más mint az osztályhoz hasonló kategória. Például a Form-okhoz kapcsolódó osztályok a Windows.Forms névtérbe ágyazták be. Hasonlóan a GDI+ osztályok hat névtérbe vannak beágyazva, amelyeket a System.Drawing.dll tartalmaz.

GDI+ Namespaces:

GDI+ a **Drawing** névtérben és a **Drawing** névtér öt alterében van definiálva.

- System.Drawing,
- System.Drawing.Design,
- System.Drawing.Printing,
- System.Drawing.Imaging,
- System.Drawing.Drawing2D,
- System.Drawing.Text.

A System.Drawing névtér osztályai és struktúrái

A **System.Drawing Namespace** biztosítja az alapvető GDI+ funkciókat. Olyan alapvető osztályokat tartalmaz, mint a **Brush**, **Pen**, **Graphics**, **Bitmap**, **Font** stb.. A Graphics osztály alapvető fontosságú a GDI+ban, mivel tartalmazza a megjelenítő eszköz (display device) rajzoló metódusait. A következő listában felsoroltunk néhány alapvető System.Drawing osztályt és struktúrát.

Osztályok és leírásuk:

| | |
|------------------|--|
| Bitmap, Image | Grafikus képek osztályai |
| Brush, Brushes. | Ecset pl. az ellipszisek, poligonok kitöltésekhez) |
| Font, FontFamily | A szövegek formáinak a meghatározása (méret, stílus, |
| Graphics | stílus, |
| Pen | A GDI+ grafikus felületének a megadása. |
| (curves) | A toll definiálása szakaszok (lines) és görbék |
| SolidBrush | rajzolására. |
| számára. | Egyszínű kitöltési minta definiálása az ecset |

Sztruktúrák és leírásuk:

| | |
|---|---|
| Color | Szín definiálása |
| Point, PointF | Síkbeli pont megadása x, y egész és valós koordinátákkal. |
| Rectangle, RectangleF és a megadásával.(top, left and width, height). | Téglalap definiálása a bal felső sarok koordinátáinak szélesség és magasság adatainak a |
| Size | Méret megadása téglalap alakú régiók esetében |

A Graphics osztály

A **Graphics** osztály a legfontosabb a GDI+ osztályai közül. Ezen osztály metódusaival tudunk rajzolni a megjelenítő eszközünk felületére. A felület megadására két lehetőségünk adódik.

Vagy rajzoljuk a Form Paint eseményének segítségével, vagy felülírjuk a form OnPaint() metódusát. Az OnPaint metódus paramétere **PaintEventArgs** típusú lesz.

1. módszer:

```
.private void Form1_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
{
    Graphics myGraphics= e.Graphics;
    Pen myPen = new Pen(Color.Red, 3);
    myGraphics.DrawLine(myPen, 30, 20, 300, 100);
}
```

2.módszer:

```
protected override void OnPaint(System.Windows.Forms.PaintEventArgs pae)
{
    Graphics myGraphics = pae.Graphics;
    Pen myPen = new Pen(Color.Red, 3);
    myGraphics.DrawLine(myPen, 30, 20, 300, 100);
}
```

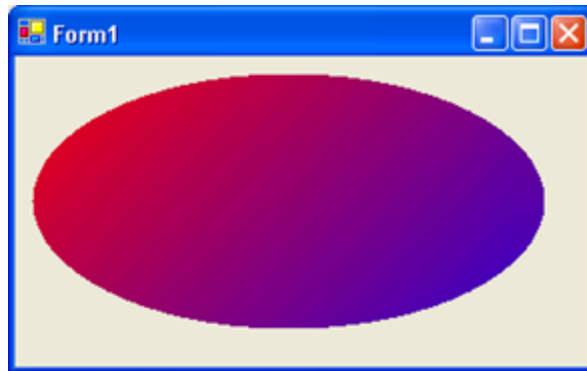
A könyvünkhöz mellékelt Videó fájlok bemutatják az alábbi rajzoló metódusok használatát.

| | |
|--|--------------------------------------|
| DrawArc | Ellipszis ív rajzolása. |
| DrawBezier, DrawBeziers | Bézier-görbék rajzolása. |
| DrawCurve, DrawClosedCurve rajzolása. | Nyílt és zárt interpoláló görbe |
| DrawEllipse | Ellipszis rajzolása. |
| DrawImage | Kép megjelenítése. |
| DrawLine | Szakaszrajzoló metódus. |
| DrawPath előállítás. | Grafikus objektum sorozat (Path) |
| DrawPie | Körcikk rajzolása. |
| DrawPolygon | Poligon rajzolása. |
| DrawRectangle | Téglalap kontúrjának a megrajzolása. |
| DrawString | Szöveg kiírása. |
| FillEllipse | Kitöltött ellipszis rajzolása. |
| FillPath | Path belső területének a kitöltése. |
| FillPie | Kitöltött körcikk rajzolása. |
| FillPolygon | Kitöltött poligon rajzolása. |
| FillRectangle | Kitöltött téglalap rajzolása. |
| FillRectangles | Téglalapok sorozatának kitöltése. |
| FillRegion | Régió belső területének a kitöltése |

A GDI+ újdonságai

Gazdagabb színkezelés és színátmenetek lehetősége

Gradient Brushes, azaz, lépcsőzetes színátmenet. Például lehetőség van egy téglalap vagy ellipszis lépcsőzetes kitöltésre:



Gradient brush hatása

Program részlet:

```
private void Form1_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
{
    Graphics g= e.Graphics;
    System.Drawing.Drawing2D.LinearGradientBrush myBrush = new
    System.Drawing.Drawing2D.LinearGradientBrush(
    ClientRectangle, Color.Red, Color.Blue,
    System.Drawing.Drawing2D.LinearGradientMode.ForwardDiagonal);
    g.FillEllipse(myBrush, 10, 10, 300, 150);
}
```

Antialiasing támogatás

A GDI+ lehetőséget biztosít a szépséghibák javítása, mint pl. a lépcsőhatás.



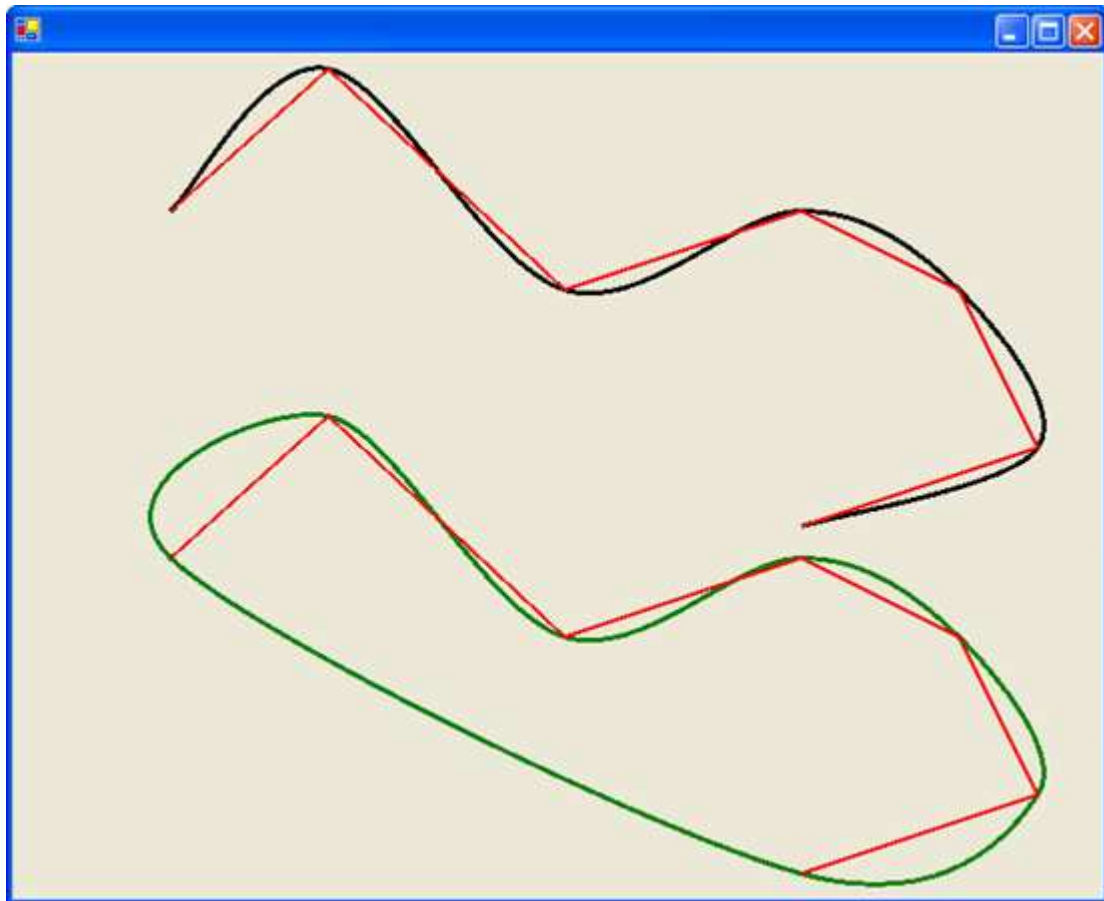
Antialiasing használatával és nélküle

Programrészlet:

```
private void Form1_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
{
    Graphics myGraphics=e.Graphics;
    myGraphics.SmoothingMode = SmoothingMode.AntiAlias;
    myGraphics.DrawEllipse(new Pen(Color.Purple,5), 10, 10, 120, 80);
    myGraphics.SmoothingMode = SmoothingMode.Default;
    myGraphics.DrawEllipse(new Pen(Color.Purple,5), 150, 10, 120, 80);
}
```

Cardinal Spline-ok

A GDI+-ban lehetőségünk van a **DrawCurve** és a **DrawClosedCurve** metódusok segítségével interpoláló Cardinal spline-ok előállítására. A spline-t egy ponttömb (`PointF[] myPoints = {...}`) segítségével tudjuk egyértelműen meghatározni. A ponttömb pontjait kontrollpontoknak, az általuk meghatározott poligont kontrollpoligonnak nevezzük. A témáról részletesen leírást találunk az Interpoláció és approximáció fejezetben.

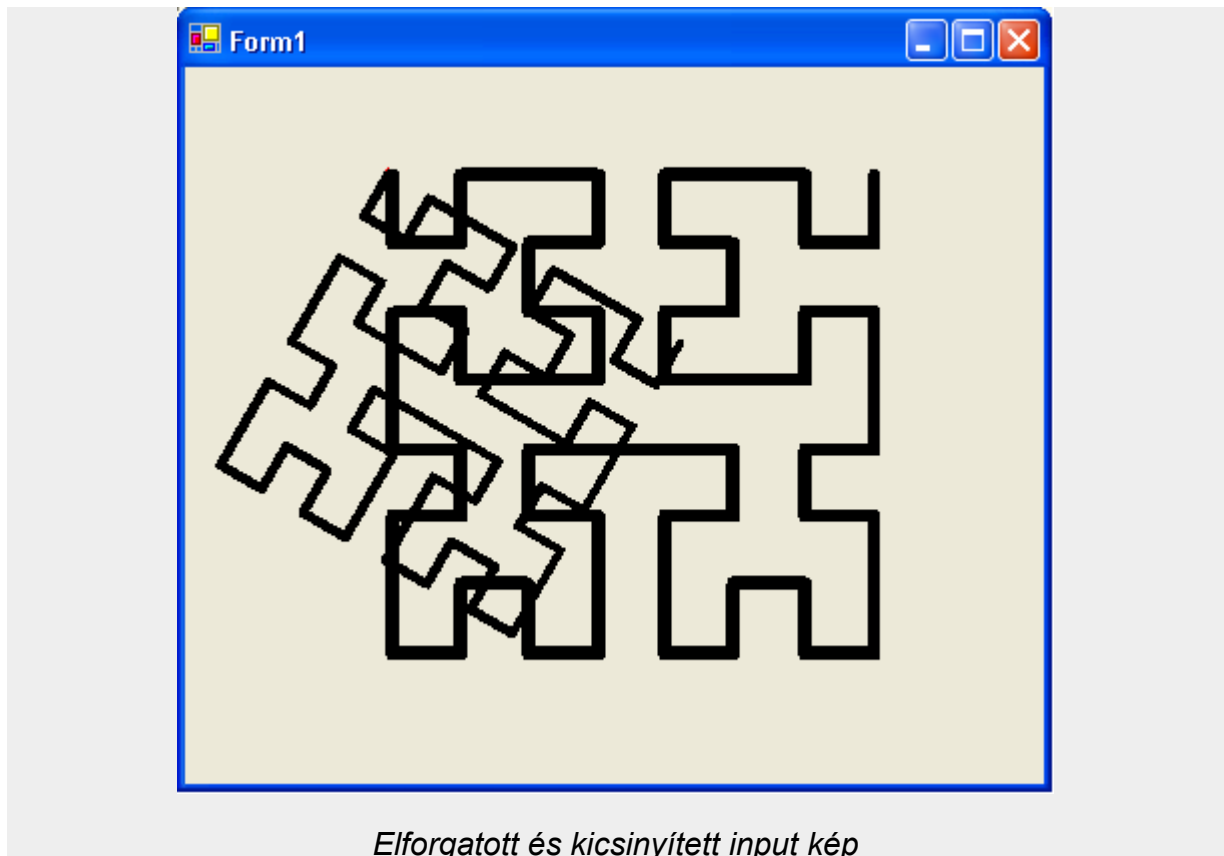


7 pontra illeszkedő nyílt és zárt interpoláló görbék

Mátrix transzformációk

A GDI+-ban széles lehetőségünk van síkbeli ponttranszformációk megadására, egymás utáni végrehajtására. A ponttranszformációkat transzformációs mátrixokkal írhatjuk le, amelyek végrehajtási sorrendjét változtathatjuk, rugalmasan kezelhetjük. Előre definiált metódusok között válogathatunk (**Translate**, **Rotate**, **Scale**, **Shear**) vagy az általános affinitás mátrixát (**Matrix** object) is megadhatjuk. A téma részletes kifejtését a Ponttranszformációk fejezetben találjuk meg.

Skálázható régiók (Scalable Regions)



```
public void Example_Scale_Rotate(object sender, PaintEventArgs e)
{
    Graphics myGraphics=e.Graphics;
    Metafile myMetafile = new Metafile("hilbert2.emf");
    myGraphics.DrawEllipse(new Pen(Color.Red,1),100,50,2,2);
    myGraphics.TranslateTransform(100, 50);
    myGraphics.DrawImage(myMetafile, 0, 0);
    e.Graphics.RotateTransform(30.0F);
    e.Graphics.ScaleTransform(0.7F, 0.7F);
    myGraphics.DrawImage(myMetafile,0, 0);
}
```

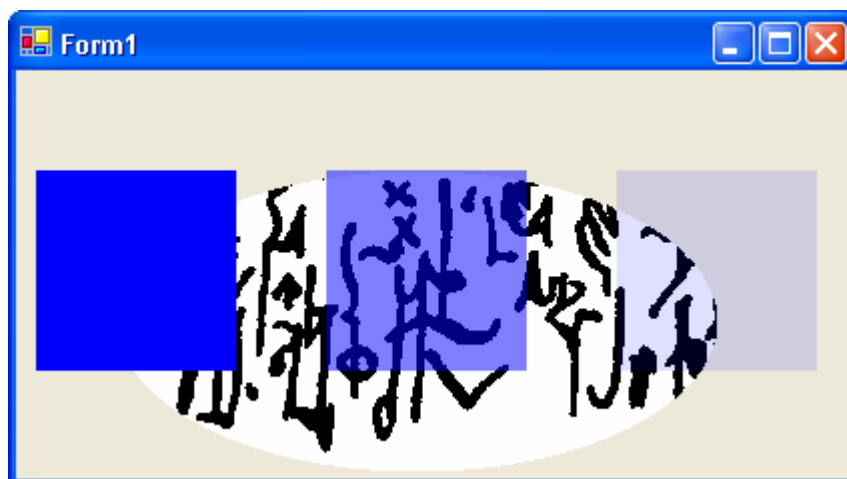
Alpha Blending

A GDI+-ban az alfa-csatorna (**Alpha Channel**) értékének változtatásával lehetővé válik, hogy különböző képek egymás előtt jelenjenek meg, így keltve olyan érzetet, mintha pl. egy objektumot ablaküvegen keresztül vagy víz alatt látnánk. Az átlátszóság mértékét a blending technikában az alfa értéke adja meg. Ez az érték

általában 0 és 1 között van, és két kép keverésének arányát határozza meg. Ha egy kép minden pixeléhez rendeltek alfa-értéket, akkor beszélünk alfa-csatornáról.

Tekintsünk egy példát:

```
private void Form1_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
{
    Graphics myGraphics= e.Graphics;
    Bitmap myBitmap = new Bitmap("navaho.jpg");
    System.Drawing.TextureBrush myBrush0 = new TextureBrush(myBitmap);
    myGraphics.FillEllipse( myBrush0,50,50,300,150);
    // nem látszik át
    System.Drawing.SolidBrush myBrush1 = new
SolidBrush(Color.FromArgb(255,0, 0, 255));
    myGraphics.FillRectangle(myBrush1,10,50,100,100);
    //félig látszik át
    System.Drawing.SolidBrush myBrush2 = new
SolidBrush(Color.FromArgb(128, 0, 0, 255));
    myGraphics.FillRectangle(myBrush2,155,50,100,100);
    //Szinte teljesen átlátszik
    System.Drawing.SolidBrush myBrush3 = new
SolidBrush(Color.FromArgb(32, 0, 0, 255));
    myGraphics.FillRectangle(myBrush3,300,50,100,100);
}
```



Nem látszik át, félig látszik át, szinte teljesen átlátszik a kék négyzet

Sokkféle grafikus fájl formátum támogatása (Support for Multiple-Image Formats):

GDI+ lehetőséget ad raszteres (Image), bittérképes (Bitmap) és metafile-ok kezelésére.

A következő formátumokat támogatja:

Raszteres formátumok:

- BMP Bitmap
- GIF (Graphics Interchange Format)
- JPEG (Joint Photographic Experts Group)
- EXIF (Exchangeable Image File)

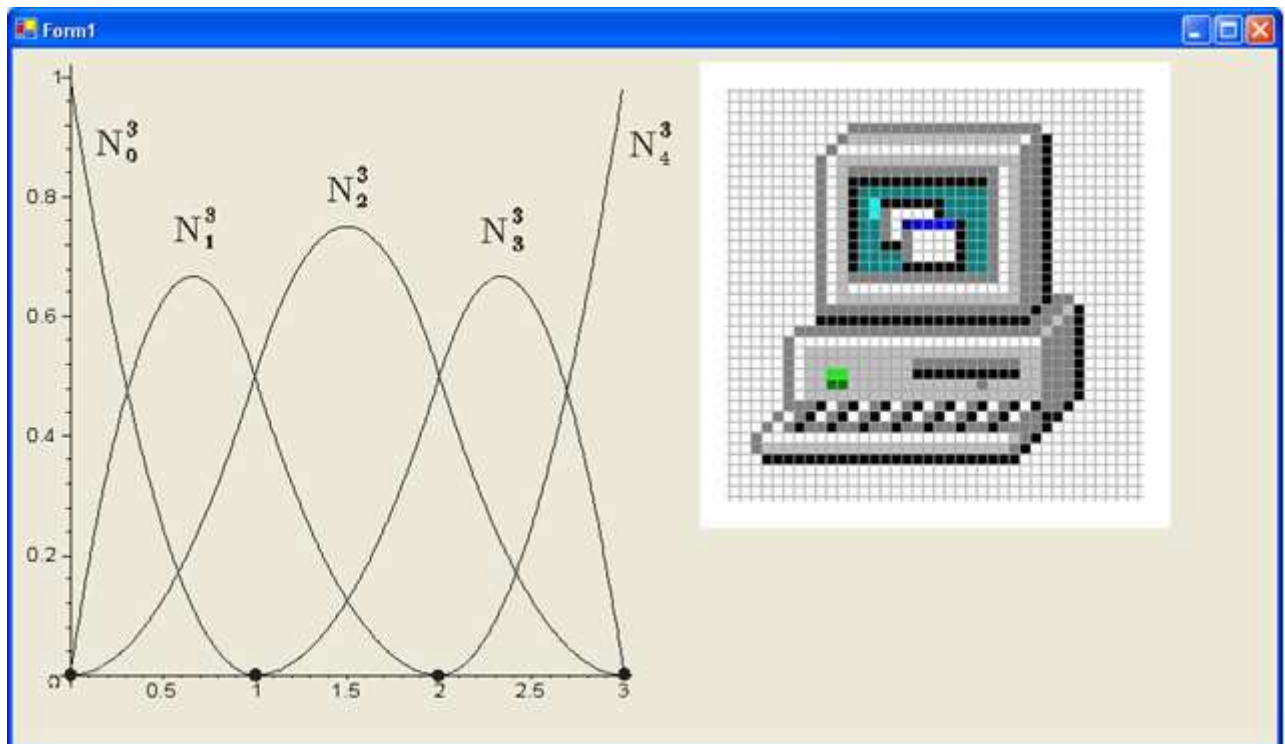
- PNG (Portable Network Graphics)
- TIFF (Tag Image File Format)

Metafile-ok, vektoros file formátumok:

- WMF (Windows Metafile), csak megjeleníthető lehet menteni nem.
- EMF (Enhanced Metafile)
- EMF+ Bővített metafile, GDI+ és/vagy GDI rekordokkal

Példa program:

```
private void Form1_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
{
    Graphics myGraphics = e.Graphics;
    Metafile myMetafile = new Metafile("meta_kep.emf");
    Bitmap myBitmap = new Bitmap("image_kep.jpg");
    myGraphics.DrawImage(myMetafile, 10, 10);
    myGraphics.DrawImage(myBitmap, 500, 10);
}
```



Metafájl és raszteres kép beillesztése

Néhány változás a GDI+ programozásban

Vonal rajzolás GDI+ használatával

Legyen feladatunk, hogy a (30,20) pozícióból a (300,100) pozícióba vonalat rajzoljunk. A GDI+-ban csak a **Graphics** és a **Pen** objektumokra lesz szükségünk. Két lehetőségünk van a rajzolásra. Vagy megrajzoljuk a vonalat a Form Paint eseményének segítségével, vagy felülírjuk a form OnPaint() metódusát. Az OnPaint metódus paramétere **PaintEventArgs** típusú lesz.

Szakasz rajzoló eljárásunk meghívja a **Graphics** osztály **DrawLine** metódusát. A **DrawLine** első paramétere a **Pen** objektum.

Most nézzük meg a két megvalósítást:

1. módszer:

```
private void Form1_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
{
    Graphics myGraphics= e.Graphics;
    Pen myPen = new Pen(Color.Red, 3);
    myGraphics.DrawLine(myPen, 30, 20, 300, 100);
}
```

2.módszer:

```
protected override void OnPaint(System.Windows.Forms.PaintEventArgs pae)
{
    Graphics myGraphics = pae.Graphics;
    Pen myPen = new Pen(Color.Red, 3);
    myGraphics.DrawLine(myPen, 30, 20, 300, 100);
}
```

Metódusok felülbírála (Method Overloading)

Nagyon sok GDI+ metódus felülbírált, azaz, számtalan metódus ugyanazon név alatt, de különböző paraméter listával. Van olyan methodus, amelynek 12 változata van (System.Drawing.Bitmap) Például a **DrawLine** metódus a következő négy formában:

```
void DrawLine(
    Pen pen,
    float x1,
    float y1,
    float x2,
    float y2) {}

void DrawLine(
    Pen pen,
    PointF pt1,
    PointF pt2) {}

void DrawLine(
    Pen pen,
    int x1,
    int y1,
    int x2,
    int y2) {}

void DrawLine(
    Pen pen,
    Point pt1,
    Point pt2) {}
```

Többé nincs a grafikus kurzornak aktuális pozíciója (Current Position)

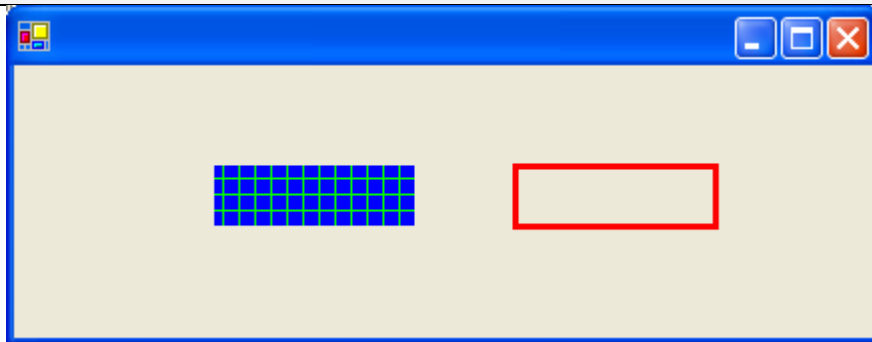
Tehát nem használhatunk moveto jellegű parancsokat, mert például az előző részben láttuk, hogy a **DrawLine** összes változatában meg kell adni a kezdő és a vég pozíciót. Hasonlóan Lineto, Linerel, Moverel jellegű utasítások sincsenek.

Szétválasztott metódus a rajzolásra (Draw) és a kitöltésre (Fill)

GDI+-nak különböző metódusai vannak pl egy téglalap körvonalának megrajzolása és belső területének a kitöltésére. A **DrawRectangle** metódus a **Pen** objektummal, a **FillRectangle** metódus pedig a **Brush** objektummal használhatjuk.

Tekintsünk egy példát:

```
private void myFillexample(PaintEventArgs e)
{
    Graphics myGraphics=e.Graphics;
    HatchBrush myHatchBrush = new HatchBrush(
        HatchStyle.Cross,
        Color.FromArgb(255, 0, 255, 0),
        Color.FromArgb(255, 0, 0, 255));
    myGraphics.FillRectangle(myHatchBrush, 100, 50, 100, 30);
    Pen myPen = new Pen(Color.FromArgb(255, 255, 0, 0), 3);
    myGraphics.DrawRectangle(myPen, 250, 50, 100, 30);
}
```



Mintával kitöltött és körvonalával megrajzolt téglalapok

A FillRectangle és DrawRectangle metódusoknak a GDI+ használatakor az első paraméter mellett még meg kell adnunk a téglalap bal felső sarkának a koordinátáit, s a téglalap szélességét, és hosszúságát (left, top, width és height). Teljesen hasonlóan járunk el az DrawEllipse és a FillEllipse metódusok használatakor is. Más programozási nyelvek grafikus könyvtáraiban szokás az ellipszist középpontjával, valamint a kis és nagy tengelyével megadni: Ellipse(x,y,a,b); ugyanez a GDI+ -ban:

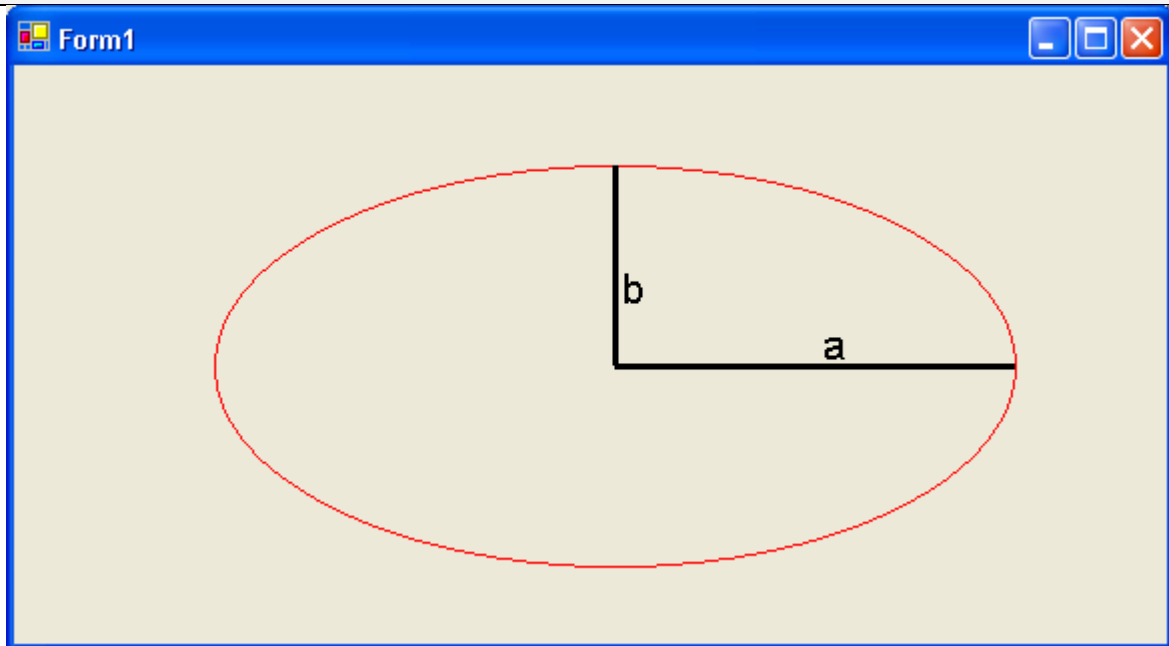
```
private void Form1_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
{
    Pen blackPen= new Pen(Color.Black, 3);
    Pen redPen= new Pen(Color.Red, 1);
    //Az ellipszis középpontja, és tengelyei
    float x = 300.0F;
    float y = 150.0F;
    float a = 200.0F;
    float b = 100.0F;
    // Átalakítás a GDI+ szerint
    float left = x-a;
    float top = y-b;
    float width = 2*a;
    float height = 2*b;
    e.Graphics.DrawEllipse(redPen,left,top, width, height);
    e.Graphics.DrawLine(blackPen,x,y,x,y-b);
}
```



```

        e.Graphics.DrawLine(blackPen, x, y, x+a, y);
//A feliratok megjelenítése
        string at = "a";
        string bt = "b";
        System.Drawing.Font drawFont = new System.Drawing.Font("Arial",
16);
        System.Drawing.SolidBrush drawBrush = new
            System.Drawing.SolidBrush(System.Drawing.Color.Black);
        // A szöveg bal felső sarkának a pozíciója.
        float ax = x+a/2;
        float ay = y-22;
e.Graphics.DrawString(at, drawFont, drawBrush, ax, ay);
        float bx = x;
        float by = y-b/2;
        e.Graphics.DrawString(bt, drawFont, drawBrush, bx, by);
    }

```



Ellipszis két tengelyével

A **Color** osztálynak négy paramétere van. Az utolsó három a szokásos RGB összetevők: piros (red), zöld (green) és a kék (blue). Az első paraméter az Alpha Blending értéke, amely a rajzoló szín és a háttér szín keverésének a mértékét határozza meg, s ezzel transzparenssé tehetjük ábráinkat

Regiók létrehozása

A GDI+-ban könnyedén formázhatunk régiókat téglalapokból (**Rectangle** object) és grafikus objektumok sorozatából (**GraphicsPath** object) Ha tehát ellipszisre vagy kerekített téglalapra, vagy egyéb grafikus alakzatra van szükségünk a régiók létrehozásánál, akkor ezeket az alakzatokat először a **GraphicsPath** objektum segítségével kell létrehoznunk, majd átadnunk a **Region** konstruktornak.

A **Region** osztály **Union** és **Intersect** metódusaival egyesíthetünk két régiót ill. meghatározhatjuk a régiók metszetét. Emellett **Xor**, **Exclude** és **Complement** metódusokat is használhatunk.

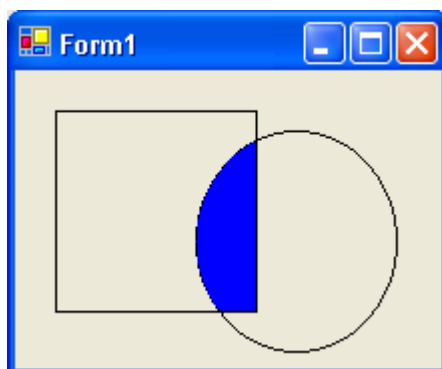
Tekintsünk egy példát:

```

public void myRegionExamples(PaintEventArgs e)
{
    // Az első téglalap létrehozása.
    Rectangle regionRect = new Rectangle(20, 20, 100, 100);
    e.Graphics.DrawRectangle(Pens.Black, regionRect);
    // Az ellipszis létrehozása GraphicsPath objektumként.
    GraphicsPath path=new GraphicsPath();
    path.AddEllipse(90, 30, 100, 110);
    // Az első régió megkonstruálása.
    Region myregion1=new Region(path);
    e.Graphics.DrawPath(new Pen(Color.Black), path);
    // Az második régió megkonstruálása.
    Region myRegion2 = new Region(regionRect);
    // Határozzuk meg a második régió metszetét az elsővel!
    myRegion2.Intersect(path);
    //myRegion2.Union(path);
    //myRegion2.Xor(path);
    //myRegion2.Exclude(path);
    //myRegion2.Complement(path);
    // Színezzük ki a metszetet!
    SolidBrush myBrush = new SolidBrush(Color.Blue);
    e.Graphics.FillRegion(myBrush, myRegion2);
}

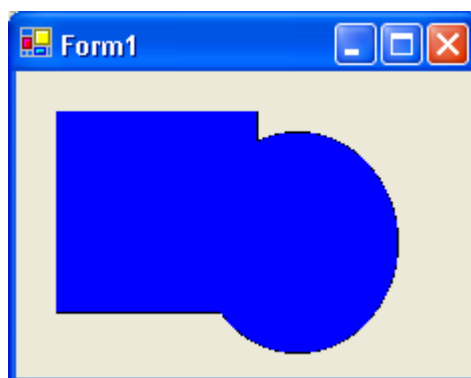
```

Futási képek:



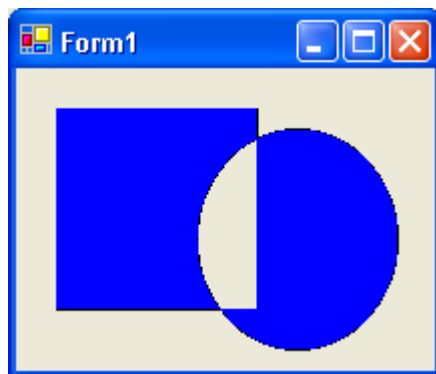
Metszet: `myRegion2.Intersect(path);`

Az **Intersect** metódus két régió esetén olyan régiót ad, amelynek pontjai mindkét régióhoz tartoznak (régiók metszete).



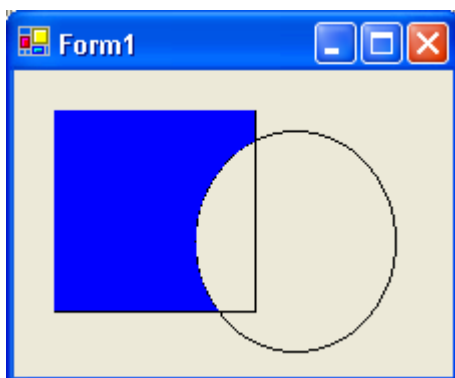
Unió: `myRegion2.Union(path);`

Az **Union** metódus két régió esetén olyan régiót ad, amelynek pontjai vagy az első vagy a második régióhoz tartoznak (régiók uniója).



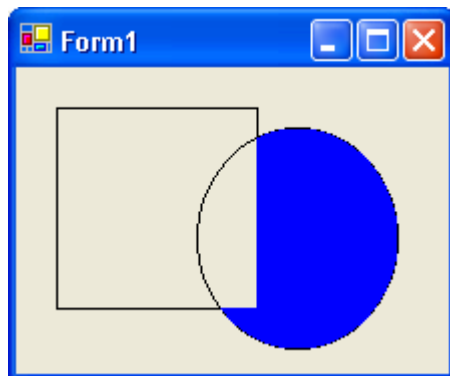
Kizáró vagy: `myRegion2.Xor(path);`

Az **Xor** metódus két régió esetén olyan régiót ad, amelynek pontjai vagy az első vagy a második régióhoz tartoznak, de nem mindkettőhöz (Kizáró vagy).



Különbség: `myRegion2.Exclude(path);`

Az **Exclude** metódus két régió esetén olyan régiót ad, amely az első régió minden olyan pontját tartalmazza, amely nincs benne a második régióban (különbség képzés).



Komplementer: `myRegion2.Complement(path);`

A **Complement** metódus két régió esetén olyan régiót ad, amely a második régió minden olyan pontját tartalmazza, amely nincs benne az első régióban (komplementer képzés).

Interpoláció és approximáció

Ebben a fejezetben a C# .NET GDI+ által nyújtott interpolációs és approximációs lehetőségek mellett a téma matematikai háttérét is megvilágítjuk, s ezzel lehetőséget adunk az általánosításra.

Hermit-görbe

A harmadfokú Hermit-görbe kezdő és végpontjával és a kezdő és végpontban megadott érintőjével adott. Adott a \mathbf{p}_0 és \mathbf{p}_1 pont, valamint a \mathbf{t}_0 és \mathbf{t}_1 érintő vektor.

Keresünk egy olyan

$$\mathbf{s}(u) = \mathbf{a}_0 u^3 + \mathbf{a}_1 u^2 + \mathbf{a}_2 u + \mathbf{a}_3, \quad u \in [0,1]$$

harmadfokú polinommal megadott görbét amelyre

$$\mathbf{s}(0) = \mathbf{p}_0, \quad \mathbf{s}(1) = \mathbf{p}_1, \quad \dot{\mathbf{s}}(0) = \mathbf{t}_0, \quad \dot{\mathbf{s}}(1) = \mathbf{t}_1.$$

teljesül, ahol a felső pont a derivált jele.

Ezek alapján az egyenletrendszer felírása és megoldása következik.

$$\mathbf{s}(0) = \mathbf{p}_0 = \mathbf{a}_3$$

$$\mathbf{s}(1) = \mathbf{p}_1 = \mathbf{a}_0 + \mathbf{a}_1 + \mathbf{a}_2 + \mathbf{a}_3$$

$$\dot{\mathbf{s}}(0) = \mathbf{t}_0 = \mathbf{a}_2$$

$$\dot{\mathbf{s}}(1) = \mathbf{t}_1 = 3 \cdot \mathbf{a}_0 + 2 \cdot \mathbf{a}_1 + \mathbf{a}_2$$

Az egyenletrendszer megoldása után

$$\mathbf{a}_0 = -2(\mathbf{p}_1 - \mathbf{p}_0) + \mathbf{t}_0 + \mathbf{t}_1$$

$$\mathbf{a}_1 = 3(\mathbf{p}_1 - \mathbf{p}_0) - 2 \cdot \mathbf{t}_0 - \mathbf{t}_1$$

$$\mathbf{a}_2 = \mathbf{t}_0$$

$$\mathbf{a}_3 = \mathbf{p}_0$$

polinom-együtthatókat kapjuk. Ezeket visszahelyettesítve és átrendezve kapjuk:

$$\mathbf{s}(u) = (2u^3 - 3u^2 + 1)\mathbf{p}_0 + (-2u^3 + 3u^2)\mathbf{p}_1 + (u^3 - 2u^2 + u)\mathbf{t}_0 + (u^3 - u^2)\mathbf{t}_1, \quad u \in [0,1]$$

Az egyenletben szereplő együttható polinomokat Hermite-polinomoknak nevezzük, és a következőképpen jelöljük:

$$H_0^3(u) = 2u^3 - 3u^2 + 1,$$

$$H_1^3(u) = -2u^3 + 3u^2,$$

$$H_2^3(u) = u^3 - 2u^2 + u,$$

$$H_3^3(u) = u^3 - u^2.$$

Ekkor a görbe felírható a Hermit-alappolinomok segítségével:

$$\mathbf{s}(u) = \mathbf{p}_0 H_0^3(u) + \mathbf{p}_1 H_1^3(u) + \mathbf{t}_0 H_2^3(u) + \mathbf{t}_1 H_3^3(u), \quad u \in [0,1]$$

Az előbbi összefüggést alapján az Hermit-görbét tekinthetjük úgy, mint a kontrolladatokat (a \mathbf{p}_0 és \mathbf{p}_1 pont, valamint a \mathbf{t}_0 és \mathbf{t}_1) súlyozott összege.

Az egységesebb szemléletmód miatt felírhatjuk a görbét mátrix alakban is:

$$\mathbf{s}(u) = \begin{pmatrix} u^3 & u^2 & u & 1 \end{pmatrix} \begin{pmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{t}_0 \\ \mathbf{t}_1 \end{pmatrix}$$

Ha a végpontbeli érintőket egyre nagyobb mértékben növeljük, akkor kialakulhat hurok a görbén, azaz átmetszheti önmagát.

Bézier-görbe

Keressünk olyan görbét, amely a megadott pontokat közelíti (approximálja) az előre megadott sorrendben és nem halad át (nem interpolálja) rajtuk, vagy legalábbis nem mindegyiken.

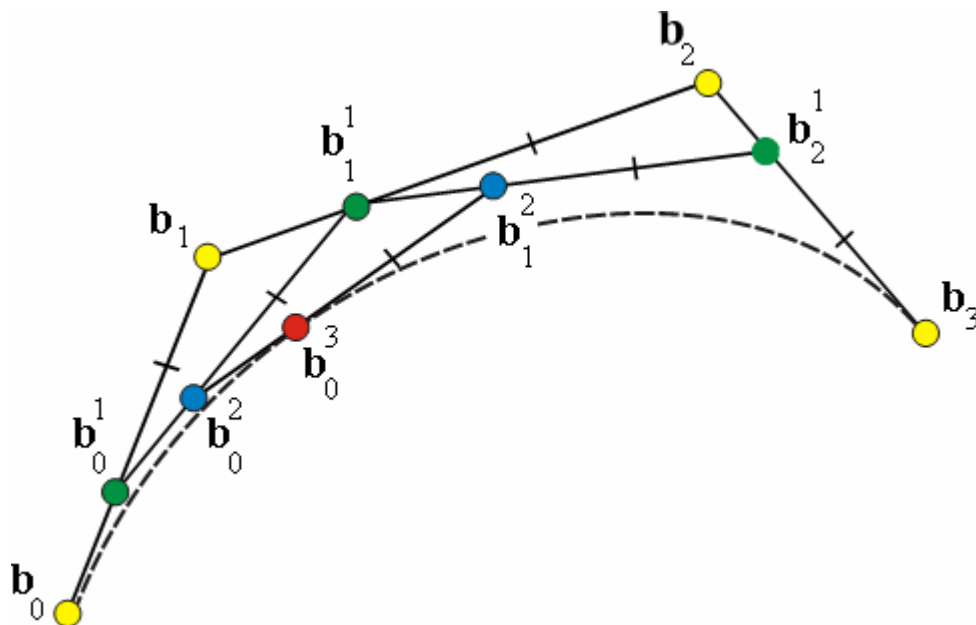
de Casteljau-algoritmus

Adottak a sík vagy a tér $\mathbf{b}_0, \dots, \mathbf{b}_n$ pontjai és $u \in [0,1]$. Ezeket később kontrollpontoknak, az általuk meghatározott poligont kontrollpoligonnak nevezzük. Legyen

$$\mathbf{b}_i^0 = \mathbf{b}_i \quad (i = 0, 1, \dots, n)$$

$$\mathbf{b}_i^r(u) = (1-u) \cdot \mathbf{b}_i^{r-1}(u) + u \cdot \mathbf{b}_{i+1}^{r-1}(u) ; \quad r = 1, \dots, n \text{ és } i = 0, 1, \dots, n-r.$$

A második egyenlet a jól ismert, hiszen a szakasz osztópont meghatározása szolgál. Most már elvégezhető a szerkesztés, ahol az így meghatározott $\mathbf{b}_0^n(u)$ pont a Bézier-görbe u paraméterhez tartozó pontja lesz.



$u = \frac{1}{3}$ paraméterhez tartozó $\mathbf{b}_0^3(u)$ Bézier-görbepont szerkesztése de Casteljau-algortmussal

A Bézier-görbe előállítás Bernsteinpolinommal

$$\mathbf{b}(u) = \sum_{i=0}^n \mathbf{b}_i \cdot B_i^n(u) \quad , u \in [0,1]$$

a Bézier-görbe u paraméterhez tartozó pontja, ahol a

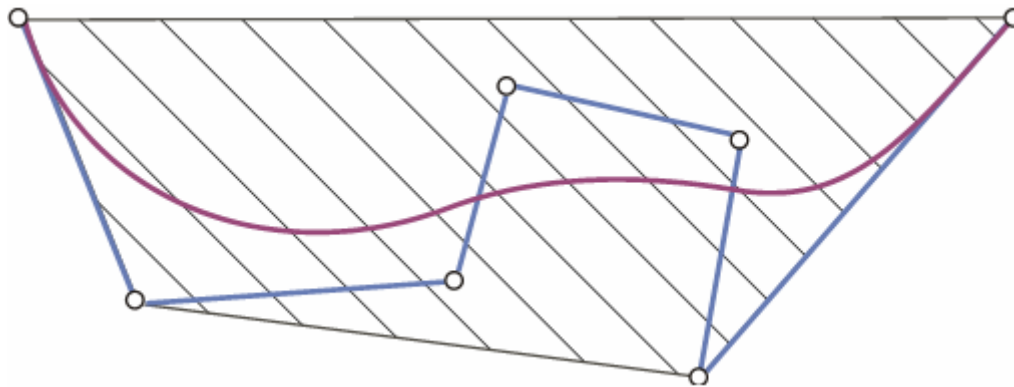
$$B_i^n(u) = \binom{n}{i} \cdot u^i \cdot (1-u)^{n-i}$$

$$\text{a Bernstein polinom. Az } \binom{n}{i} = \frac{n!}{i!(n-i)!} \text{ összefüggést a}$$

binominális tétel segítségével nyerjük. Láthatjuk, hogy a Hermit-görbéhez hasonlóan a kontrolladatokat, jelen esetben a kontrollpontokat, súlyozott összegéről van szó.

Bézier-görbe néhány tulajdonságai

- A Bézier-görbe a kontrollpontjai affin transzformációjával szemben invariáns. Ez következik a de Casteljau-féle előállításból. Ezen tulajdonságot kihasználva, a görbe affin transzformációja (Pl.:eltolás, elforgatás, tükrözés, skálázás, párhuzamos vetítés) esetén elég a kontrollpontokra végrehajtani a transzformációt, mivel a transzformált pontok által meghatározott Bézier-görbe megegyezik az eredeti görbe transzformáltjával. De centrális vetítésre nézve nem invariáns.
- Ha $u \in [0,1]$, akkor a Bézier görbe kontrollpontjainak konvex burkán belül van.

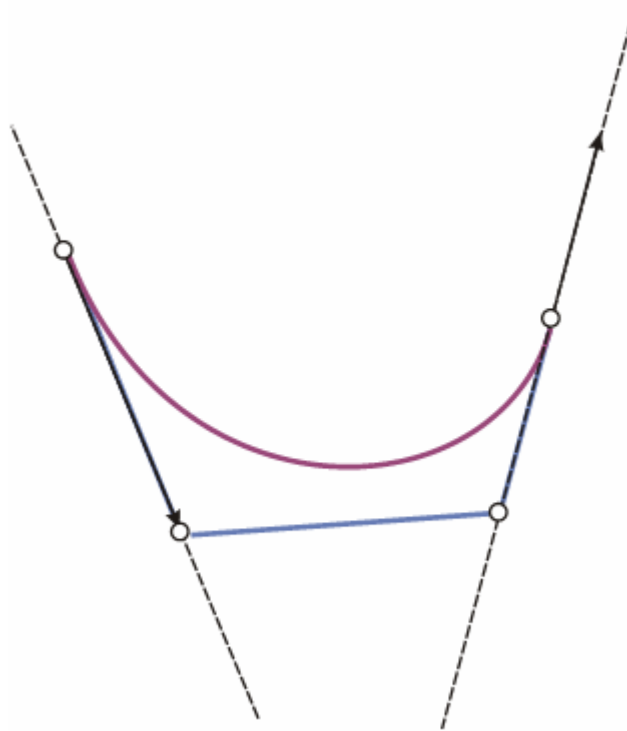


A Bézier-görbe kontrollpontjainak konvex burkán belül marad

- A Bézier-görbe az első és utolsó kontrollponton áthalad.
- A Bézier-görbe „szimmetrikus”, azaz a $\mathbf{b}_0, \dots, \mathbf{b}_n$ és a $\mathbf{b}_n, \dots, \mathbf{b}_0$ pontok ugyanazt a görbét állítják elő.
- Ha $u \in [0,1]$ akkor a Bézier-görbe kezdő- és végérintője:

$$\frac{d}{du} \mathbf{b}(0) = n(\mathbf{b}_1 - \mathbf{b}_0) \quad , \quad \frac{d}{du} \mathbf{b}(1) = n(\mathbf{b}_n - \mathbf{b}_{n-1})$$

Tehát a kezdő és a végpontban az érintők tartó egyenese a kontrollpoligon oldalai.



A Bézier-görbe és érintői $n = 3$ esetén az érintővektorok harmadát felmérve

- A görbe globálisan változtatható, azaz, ha megváltoztatjuk egy kontrollpontjának a helyzetét, akkor az egész görbe alakváltozáson megy keresztül. Bebizonyítható a Bernstein-polinomok tulajdonsága alapján, hogy a \mathbf{b}_i kontrollpontnak a $u = \frac{i}{n}$ paraméterértéknél van legnagyobb hatása a görbe alakjára. Ez utóbbi tulajdonságot nevezik úgy, hogy a görbe pszeudolokálisan változtatható. Tehát a Bézier-görbe alakváltozása jól prognosztizálható.
- A polinominális előállításból jól látható, hogy $\mathbf{b}_0, \dots, \mathbf{b}_n$ pontot, azaz $n+1$ pontot n -edfokú görbével approximál, azaz a kontrollpontok számának növekedésével nő a poligon fokszáma is.

Harmadfajú Bézier-görbék

$n = 3$ esetén a görbe Bernstein-polinom alakja:

$$\begin{aligned} \mathbf{b}(u) &= \sum_{i=0}^3 \mathbf{b}_i B_i^3(u) = \sum_{i=0}^3 \mathbf{b}_i \binom{3}{i} u^i (1-u)^{3-i} = \mathbf{b}_0 (1-u)^3 + \mathbf{b}_1 3u(1-u)^2 + \mathbf{b}_2 3u^2(1-u) + \mathbf{b}_3 u^3 = \\ &= \mathbf{b}_0 (-u^3 + 3u^2 - 3u + 1) + \mathbf{b}_1 (3u^3 - 6u^2 + 3u) + \mathbf{b}_2 (-3u^3 + 3u^2) + \mathbf{b}_3 u^3 \end{aligned}$$

A görbe érintője:

$$\dot{\mathbf{b}}(0) = 3(\mathbf{b}_1 - \mathbf{b}_0) \quad , \quad \dot{\mathbf{b}}(1) = 3(\mathbf{b}_3 - \mathbf{b}_2)$$

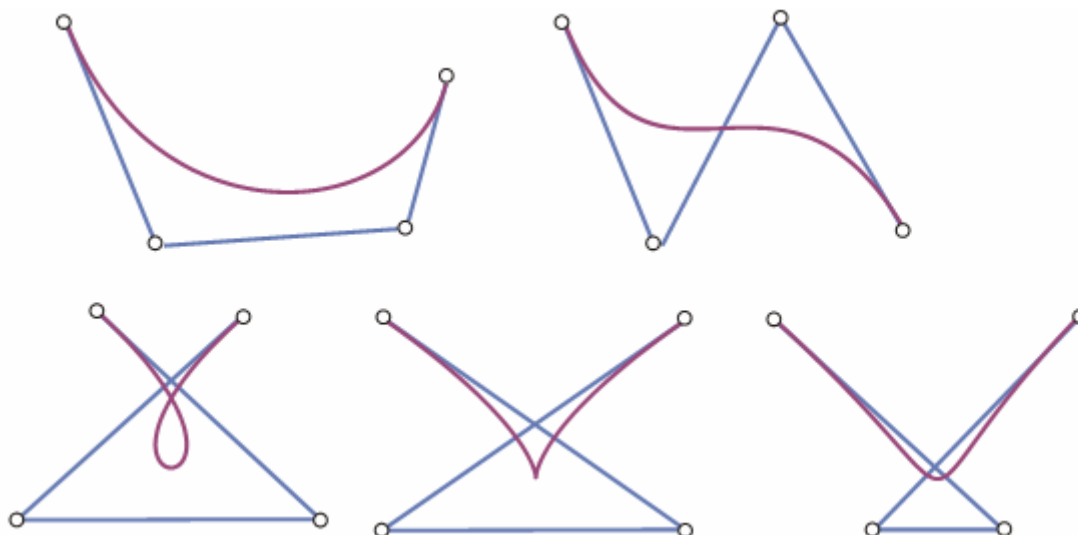
Ezután könnyedén megtalálhatjuk az Hermit-görbe és a Bézier-görbe közötti kapcsolatot. Ha az Hermit-görbe a \mathbf{p}_0 és \mathbf{p}_1 pontok, és a \mathbf{t}_0 és \mathbf{t}_1 érintőkkel van

meghatározva, akkor a kezdő és végpontbeli érintőknek meg kell egyezniük a Bézier-görbe érintőivel, azaz $\mathbf{t}_0 = \dot{\mathbf{b}}(0)$, $\mathbf{t}_1 = \dot{\mathbf{b}}(1)$, továbbá a kezdő és végpontok is egybeesnek.

Tehát az Hermit-göbével megegyező Bézier-görbe kontrollpontjai:

$$\mathbf{b}_0 = \mathbf{p}_0, \quad \mathbf{b}_1 = \mathbf{p}_0 + \frac{1}{3}\mathbf{t}_0, \quad \mathbf{b}_2 = \mathbf{p}_1 - \frac{1}{3}\mathbf{t}_1, \quad \mathbf{b}_3 = \mathbf{p}_1.$$

A harmadfokú Bézier-görbék (s természetesen az Hermit-görbék is) változatos képet mutathatnak.



Harmadfokú Bézier-görbék

Ha a harmadfokú Bézier-görbe kontrollpontjait nincsenek egy síkban, akkor a Bézier-görbe térgörbe lesz, azaz nem találunk olyan síkot amelyre a görbe minden pontja illeszkedne. A három kontrollpont esetén a másodfokú Bézier-görbe már egy jól ismert síkgörbe, parabola lesz.

Kapcsolódó Bézier-görbék

Az előző fejezetben láttuk, hogyan milyen kapcsolat van az Hermit-görbe és a Bézier-görbe között. Ha több mint két pontot szeretnénk összekötni interpoláló görbével, akkor kézenfekvő megoldás a kapcsolódó harmadfokú Bézier-görbék alkalmazása, amelyek természetesen Hermit-görbék is lehetnek. Kapcsolódó görbeívek használata igen gyakori a modellezésben. Ilyen esetekben a csatlakozásnál megadjuk a folytonosság mértékét. Ha az $\mathbf{a}(u)$, $u \in [0,1]$ és $\mathbf{b}(u)$, $u \in [0,1]$ két csatlakozó görbe amely négy négy kontrollponttal adott: $\mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3$ és $\mathbf{b}_0, \mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3$.

A kapcsolódásnál megkövetelhetünk nulladrendű C^0 , elsőrendű C^1 , illetve másodrendű C^2 folytonosságot. Általánosságban azt mondhatjuk, hogy két csatlakozógörbe kapcsolódása C^n folytonos, ha az egyik görbe deriváltjai a végpontjában megegyeznek a másik görbe deriváltjaival a kezdőpontban, az n . deriváltig bezárólag. A matematikai folytonosság vagy parametrikus folytonosság mellett létezik geometriai folytonosság is. G^0 megegyezik C^0 -lal, és G^1 folytonosan kapcsolódik két görbe, ha az érintők a kapcsolódási pontban egy irányba néznek, de

a hosszuk nem feltétlenül egyezik meg, azaz egyik érintő skalárszorosa a másiknak:

$$\frac{d}{du} \mathbf{a}(1) = \varpi \frac{d}{du} \mathbf{b}(0), \quad \varpi > 0$$

A nulladrendű folytonossághoz elegendő, ha a csatlakozáskor keletkező görbe megrajzolható anélkül, hogy a ceruzánkat felemelnénk. A mi esetünkben ez akkor teljesül, ha az első görbe végpontja megegyezik a második görbe kezdőpontjával, azaz: $\mathbf{a}(1) = \mathbf{b}(0)$ és mivel ezen görbepontok megegyeznek a megfelelő kontrollpontokkal, hiszen a Bézier-görbe a végpontokat interpolálja, ezért $\mathbf{a}_3 = \mathbf{b}_0$ kell, hogy teljesüljön.

Az elsőrendű, C^1 folytonossághoz az érintőknek kell megegyezniük, azaz

$$\frac{d}{du} \mathbf{a}(1) = \frac{d}{du} \mathbf{b}(0)$$

kell, hogy teljesüljön. Ez a kontrollpontokra az $(\mathbf{a}_3 - \mathbf{a}_2) = (\mathbf{b}_1 - \mathbf{b}_0)$ feltételt jelenti, azaz, amellet, hogy a két szegmens kezdő- és végpontja megegyezik, az $\mathbf{a}_2, \mathbf{a}_3 = \mathbf{b}_0, \mathbf{b}_1$ pontoknak egy egyenesre kell illeszkedniük és az \mathbf{a}_3 pontnak feleznie kell az $\mathbf{a}_2\mathbf{b}_1$ szakaszt.

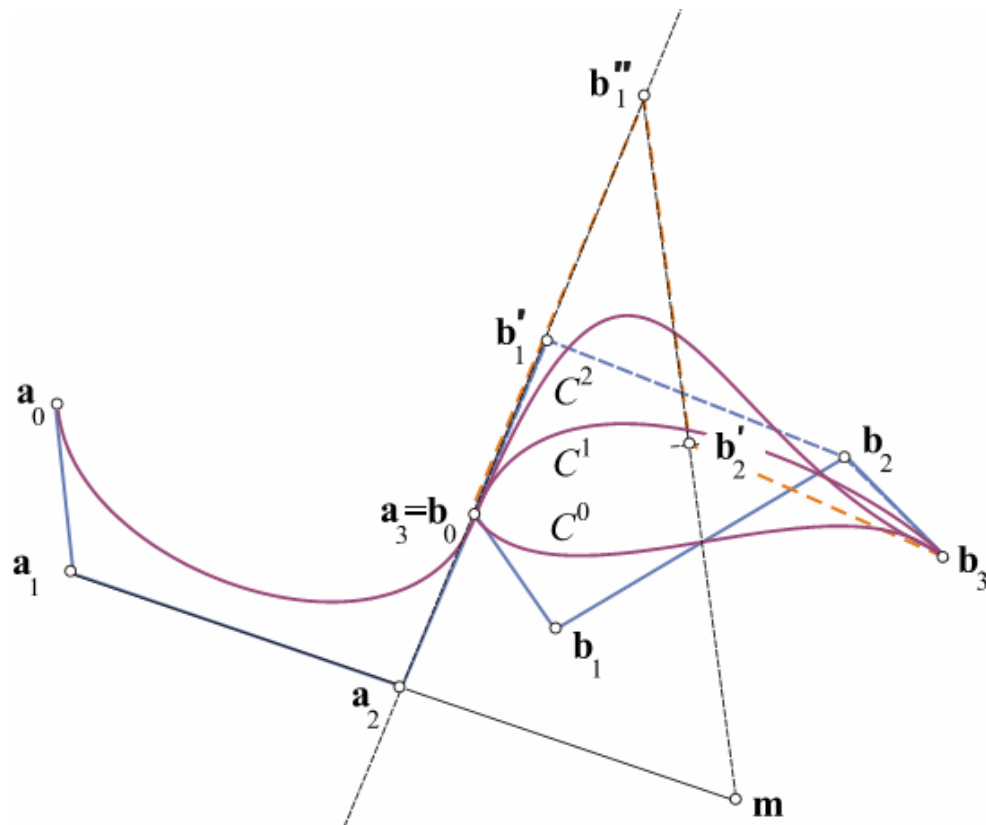
A másodrendben folytonos kapcsolódáshoz a fenti feltételeken kívül a következőnek

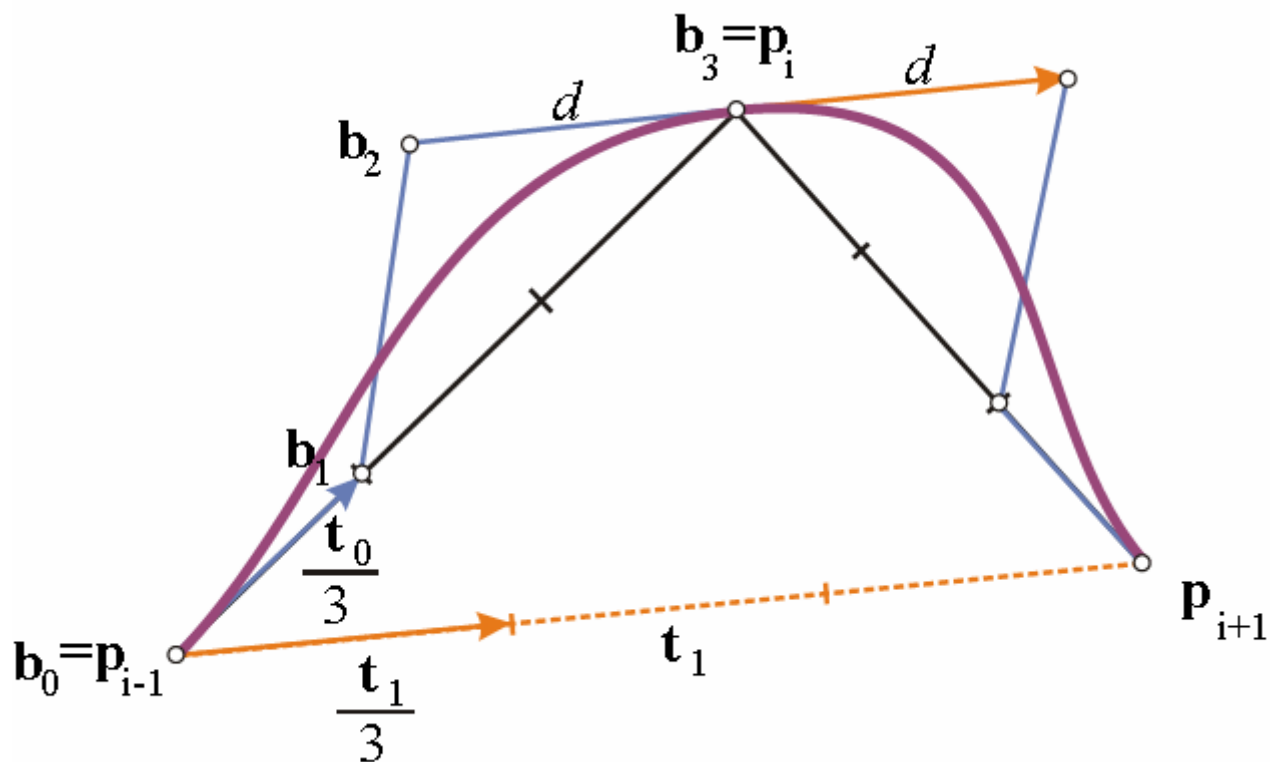
kell teljesülnie: $\frac{d^2}{du^2} \mathbf{a}(1) = \frac{d^2}{du^2} \mathbf{b}(0)$ Ez a kontrollpontokra nézve a következőt jelenti:

$$((\mathbf{a}_3 - \mathbf{a}_2) - (\mathbf{a}_2 - \mathbf{a}_1)) = ((\mathbf{b}_2 - \mathbf{b}_1) - (\mathbf{b}_1 - \mathbf{b}_0))$$

ami geometriai szempontból azt jelenti, hogy az $\mathbf{a}_1\mathbf{a}_2$ egyenes és a $\mathbf{b}_1\mathbf{b}_2$ egyenes \mathbf{m} metszéspontjára teljesül, hogy \mathbf{a}_2 felezi az $\mathbf{a}_1\mathbf{m}$ szakaszt, \mathbf{b}_1 pedig felezi az \mathbf{mb}_2 szakaszt.

A gyakorlatban C^2 folytonosságú kapcsolat elégséges, pl. animáció esetén a mozgó kamera által készített felvétel akkor lesz valóságú, ha a második derivált is megegyezik. Gondoljunk arra, hogy az út/idő függvény második deriváltja a gyorsulást adja, tehát ha a kapcsolódási pontban megváltozik a gyorsulás, akkor szaggatott felvételt kapunk.





3 pontot interpoláló C^1 folytonosan csatlakozó Bézier-görbék

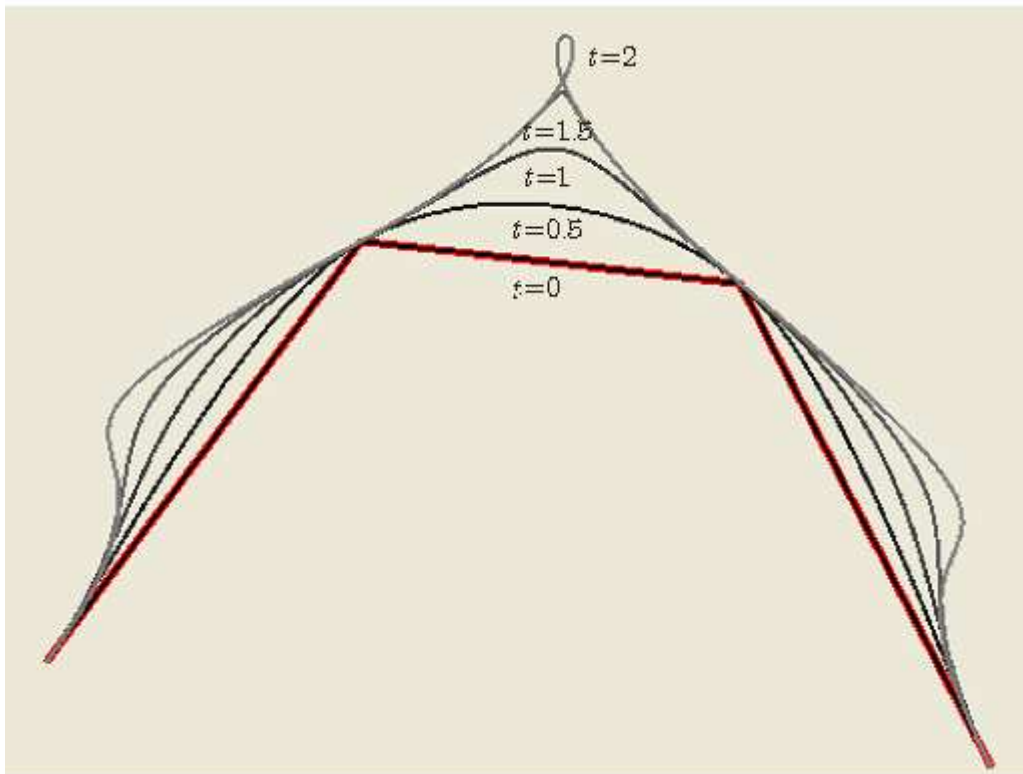
Az i . Bézier szegmens kezdő és végpontja a szomszédos \mathbf{p}_i és \mathbf{p}_{i+1} pontok. A görbe deriváltja egy közbülső \mathbf{p}_i pontban párhuzamos \mathbf{p}_{i-1} \mathbf{p}_{i+1} egyenessel, azaz

$$\dot{\mathbf{b}}_i(u) = t(\mathbf{p}_{i+1} - \mathbf{p}_{i-1}), \quad t \geq 0$$

Ne felejtjük el, hogy a harmadrendű Bézier-görbe négy pontjával (négy vektorral) adott, pl. négy kontrollpont, vagy ha Hermit-görbéként adjuk meg, akkor a kezdő és a vég pont, valamint a kezdő és a végpontban az érintők. Az i . Bézier szegmens kezdő és végpontja $\mathbf{b}_0 = \mathbf{p}_i, \mathbf{b}_3 = \mathbf{p}_{i+1}$. Mivel a görbe érintője:

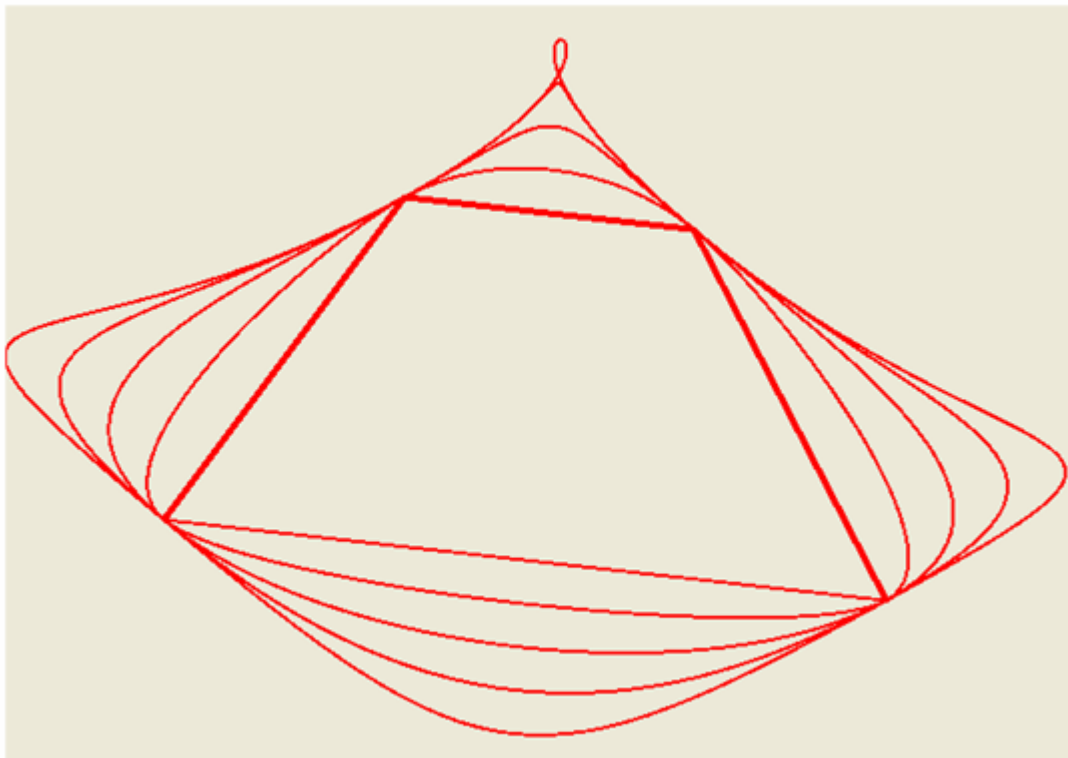
$$\dot{\mathbf{b}}(0) = 3(\mathbf{b}_1 - \mathbf{b}_0) \quad , \quad \dot{\mathbf{b}}(1) = 3(\mathbf{b}_3 - \mathbf{b}_2)$$

ezért minden kontrollpont minden szegmens esetén most már meghatározható. Az érintők meghatározásánál használhatunk egy $t \geq 0$ tenziós értéket, amely az érintők nagyságát befolyásolja. Ha $t = 0.5$ akkor a Catmul-Rom spline-t, ha $t = 0$ akkor a kontrollpontokat összekötő poligont kapjuk meg.



A t tenziós érték hatása a görbe alakjára

Zárt görbe estén az érintő a kezdő és a végpontban is az $\dot{\mathbf{b}}_i(u) = t(\mathbf{p}_{i+1} - \mathbf{p}_{i-1})$, $t \geq 0$ általános képlettel határozható meg.

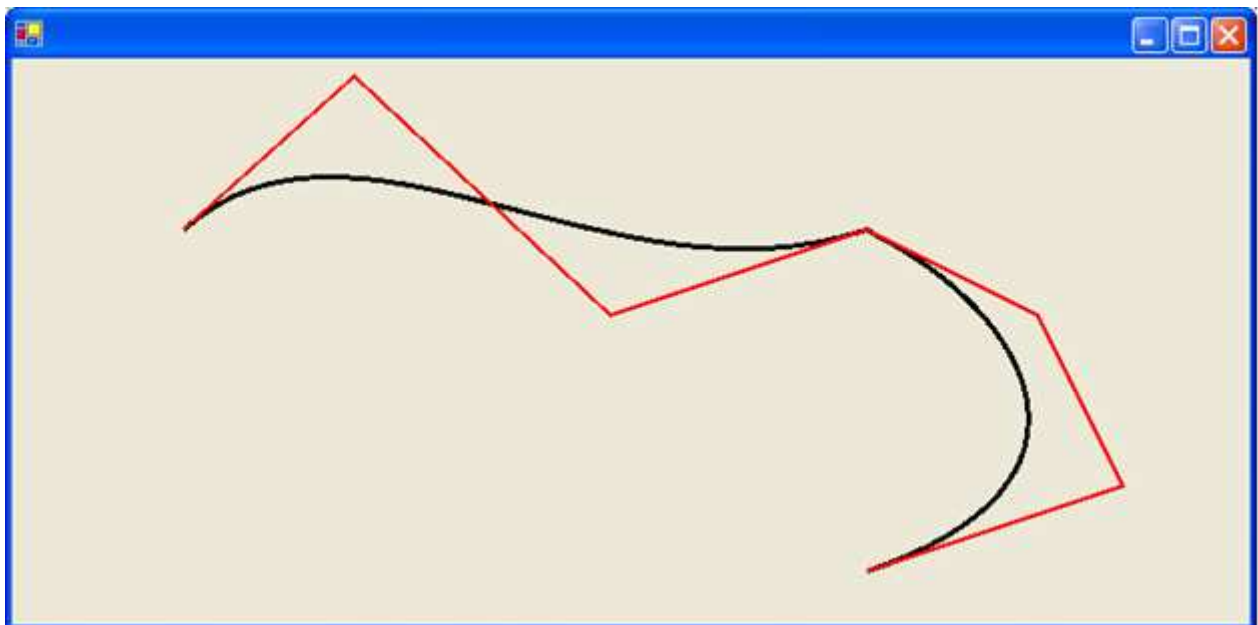


Zárt görbék különböző t értékeknél

Az előbbi képeket előállító program részlet:

```
private void Form1_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
{
    Pen redPen    = new Pen(Color.Red, 4);
    Color myColor = new Color();
    FillMode myFill=new FillMode();
// Pontok megadása
PointF point1 = new PointF(100.0F, 350.0F);
PointF point2 = new PointF(250.0F, 150.0F);
PointF point3 = new PointF(430.0F, 170.0F);
PointF point4 = new PointF(550.0F, 400.0F);
PointF[] curvePoints ={
                        point1,
                        point2,
                        point3,
                        point4,
                        };
// Kontrollpoligon megrajzolása:
e.Graphics.DrawLine(redPen, curvePoints);
// Cardinal-spline kirajzolása változó tenziós értékkel:
for (float i = 0.0F; i <= 2; i+=0.5F)
{
    float tension = i*1.0F;
    myColor=Color.FromArgb((int) (i*63.0F), (int) (i*63.0F), (int) (i*63.0F)
);
    e.Graphics.DrawCurve(new Pen(myColor,2), curvePoints,tension);
// Zárt görbe előállítás:
e.Graphics.DrawClosedCurve(new Pen(Color.Red,2),
curvePoints,tension,myFill);
}
}
```

Ahogy a fenti programrészletből láthattuk, hogy GDI+ segítségével lehetőségünk van cardinal spline előállítására **DrawCurve** metódussal. Mivel a GDI+ kihasználja, hogy a cardinal spline csatlakozó Bézier-görbékéből áll, ezért szükség van egy Bézier-görbét előállító metódusra is. A **DrawBezier** négy kontroll pontra illetve közelítő görbét, míg a **DrawBeziers** pedig C^0 folytonosan kapcsolódó Bézier-görbékét rajzol.



7 pont esetén a **DrawBeziers** metódus futási eredménye

Pontranszformációk

A GDI+ .NET-ben síkbeli, kölcsönösen egyértelmű ponttranszformációkat lehet könnyedén megvalósítani. A ponttranszformációk leírásánál **homogén koordinátákat** használunk, amit az egységes mátrix reprezentáció érdekében teszünk.

Homogén koordináták

A sík pontjait olyan rendezett számhármassal reprezentáljuk, amelyek arányosság erejéig vannak meghatározva, és mind a három koordináta egyszerre nem lehet nulla. A definíció értelmezése:

- Rendezett számhármassal: $[x_1, x_2, x_3]$
- Arányosság: az $[x_1, x_2, x_3]$ ugyanazt a pontot jelöli, mint a $[\omega x_1, \omega x_2, \omega x_3]$, ahol ω egy 0-tól különböző valós szám. Pl: $[1, -2, 2]$ ugyanazt a pontot jelöli, mint a $[2, -4, 4]$.
- $[0, 0, 0]$ homogén koordinátájú pont nem létezik.

Áttérés hagyományos Descartes koordinátákról homogén koordinátákra:

Legyen egy síkbeli pont hagyományos valós koordinátája $[x, y]$, a homogén koordináták alakja $[x, y, 1]$ lesz. Tehát $x_1=x$, $x_2=y$, $x_3=1$ megfeleltetést használtunk. Mivel a homogén koordináták csak arányosság erejéig vannak meghatározva, ezért most már szorozhatjuk a koordinátákat, egy tetszőleges nem nulla számmal.

Visszatérés homogén koordinátákról Descartes koordinátákra:

A GDI+ esetében csak olyan transzformációkat fogunk használni, amelyek esetében a harmadik koordináta egy lesz, ezért a visszatérésnél egyszerűen elhagyjuk. Általánosan ha egy pont homogén koordinátája $[x_1, x_2, x_3]$, és x_3 nem nulla, akkor az első két koordinátát eloszthatjuk a definícióban foglalt arányossági tulajdonság miatt a harmadik koordinátával:

$$[x_1/x_3, x_2/x_3, 1].$$

Ebben az esetben láthatjuk, hogy valójában az $x = x_1/x_3$ és az $y = x_2/x_3$ megfeleltetést használtuk. Ha $x_3 = 0$, akkor nincs hagyományos valós megfelelője a pontnak, ezzel az esettel nem foglalkozunk, mert az általunk használt ponttranszformációk esetében nem fordul elő.

Ponttranszformációk

A homogén koordináták felhasználásával most már egyszerűen megadhatjuk a ponttranszformációkat általános alakját

$$\mathbf{p}' = \mathbf{p} \cdot \mathbf{M}$$

ahol \mathbf{p} illetve \mathbf{p}' a transzformálandó pont illetve a transzformált helyvektora, \mathbf{M} pedig a transzformációt megadó 3×3 -as mátrix és $|\mathbf{M}| \neq 0$.

A fenti mátrixegyenlet kifejtve:

$$\begin{bmatrix} x'_1, x'_2, x'_3 \end{bmatrix} = \begin{bmatrix} x_1, x_2, x_3 \end{bmatrix} \cdot \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}$$

Mivel a GDI+ csak olyan esetekkel foglalkozik, amikor a harmadik koordináta 1 marad, ezért a következő alakra egyszerűsödik a helyzet:

$$\begin{bmatrix} x', y', 1 \end{bmatrix} = \begin{bmatrix} x, y, 1 \end{bmatrix} \cdot \begin{bmatrix} m_{11} & m_{12} & 0 \\ m_{21} & m_{22} & 0 \\ m_{31} & m_{32} & 1 \end{bmatrix}$$

Ha elvégezzük a mátrix szorzást akkor a következő egyenletrendszerhez jutunk:

$$\begin{aligned} x' &= m_{11}x + m_{21}y + m_{31} \\ y' &= m_{12}x + m_{22}y + m_{32} \end{aligned}$$

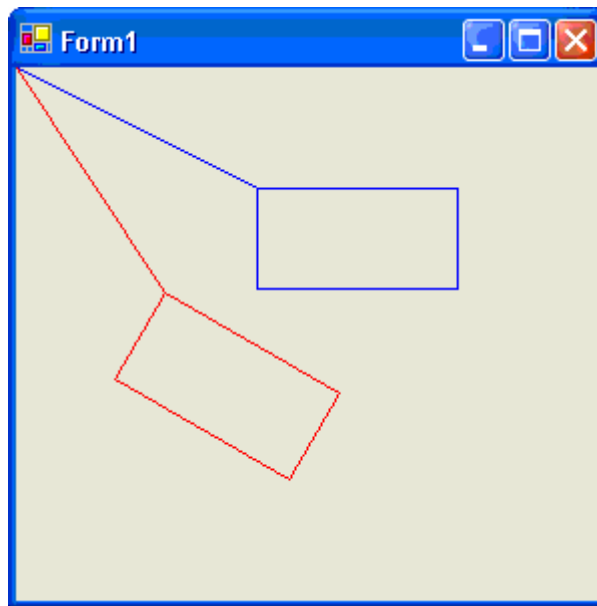
A kapott egyenletrendszer az általános síkbeli affinitást írja le. GDI+ .NET-ben az általános affinitás mátrixának megadása:

```
Matrix myMatrix = new Matrix(m11,m12,m21,m22,m31,m32);
```

Első példa:

A következő példában a **myMatrix** által definiált ponttranszformációt a `myMatrix.TransformPoints(myArray)` metódussal hajtjuk végre a **myArray** ponttömbön.

```
private void Form1_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
{
    Pen myPen = new Pen(Color.Blue, 1);
    Pen myPen2 = new Pen(Color.Red, 1);
    // téglalap pontjai
    Point[] myArray =
    {
        new Point(120, 60),
        new Point(220, 60),
        new Point(220, 110),
        new Point(120, 110),
        new Point(120, 60)
    };
    // A kék téglalap a transzformáció előtt
    e.Graphics.DrawLines(myPen, myArray);
    // a forgatás szöge radiánban
    double alpha=30*System.Math.PI/180;
    Matrix myMatrix = new Matrix((float) Math.Cos(alpha), (float)
Math.Sin(alpha), (float) -Math.Sin(alpha), (float) Math.Cos(alpha), 0, 0);
    //A transzformáció végrehajtása
    myMatrix.TransformPoints(myArray);
    //Az elforgatott téglalap megrajzolása piros tollal
    e.Graphics.DrawLines(myPen2, myArray);
}
```

Ponttömb transzformációja

Második példa:

A következő példában három **myMatrix** által definiált ponttranszformációk szorzatát állítjuk elő a **myMatrix.Multiply** metódussal, majd a **Graphics.Transform** metódussal végrehajtjuk a transzformációt a **Graphics** objektumain, egy téglalapot érintő ellipszisen. A **Multiply** metódus **MatrixOrder** paramétere határozza meg a mátrix szorzás sorrendjét, amelynek lehetséges értékei:

- Append: az új transzformációt a régi után kell végrehajtani
- Prepend: az új transzformációt a régi előtt kell végrehajtani

$$\begin{array}{c} \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ \text{Skálázás} \end{array} \cdot \begin{array}{c} \begin{bmatrix} \cos 30^\circ & \sin 30^\circ & 0 \\ -\sin 30^\circ & \cos 30^\circ & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ \text{Forgatás} \end{array} \cdot \begin{array}{c} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 150 & 50 & 1 \end{bmatrix} \\ \text{Eltolás} \end{array} = \begin{array}{c} \begin{bmatrix} 2 \cos 30^\circ & \sin 30^\circ & 0 \\ -2 \sin 30^\circ & \cos 30^\circ & 0 \\ 150 & 50 & 1 \end{bmatrix} \\ \text{Szorzat Mátrix} \end{array}$$

Formális leírása a transzformációnak

```

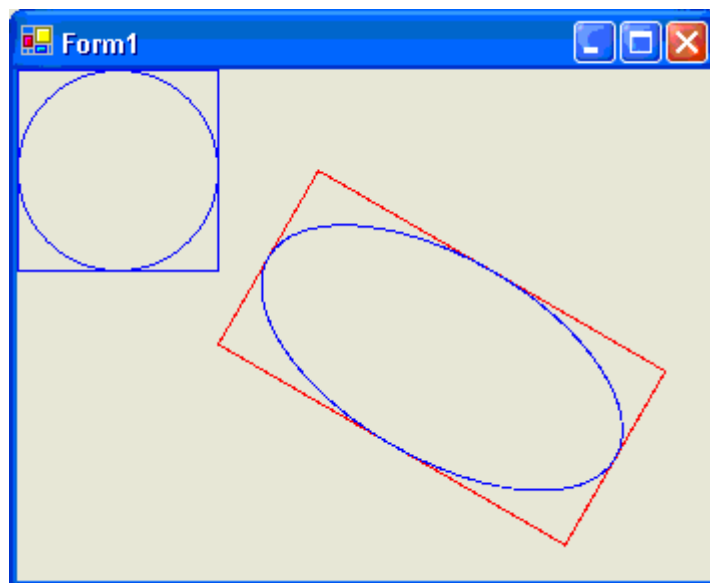
{
    Pen myPen = new Pen(Color.Blue, 1);
    Pen myPen2 = new Pen(Color.Red, 1);
    double alpha=30*Math.PI/180;
    // Mátrixok inicializálása:
    // Skálázás.
    Matrix myMatrix1 = new Matrix(2.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f);
    // Elforgatás 30 fokkal.
    Matrix myMatrix2 = new Matrix(
        (float)Math.Cos(alpha), (float)Math.Sin(alpha),
        (float)-Math.Sin(alpha), (float)Math.Cos(alpha),
        0.0f, 0.0f);
    // Eltolás.

```

```

    Matrix myMatrix3 = new Matrix(1.0f, 0.0f, 0.0f, 1.0f, 150.0f,
50.0f);
    // Matrix1 és Matrix2 összeszorozása.
    myMatrix1.Multiply(myMatrix2, MatrixOrder.Append);
    // A szorzat mátrix szorzása Matrix3-mal.
    myMatrix1.Multiply(myMatrix3, MatrixOrder.Append);
    // Érintő ellipszis rajzolása:
    e.Graphics.DrawRectangle(myPen, 0, 0, 100, 100);
    e.Graphics.DrawEllipse(myPen, 0, 0, 100, 100);
    // A szorzat transzformáció végrehajtása a Graphics objektumain:
    e.Graphics.Transform = myMatrix1;
    // Érintő ellipszis megrajzolása a transzformáció után:
    e.Graphics.DrawRectangle(myPen2, 0, 0, 100, 100);
    e.Graphics.DrawEllipse(myPen, 0, 0, 100, 100);
}

```



Skálázott, eltoló és elforgatott érintő ellipszis

GDI+ transzformációs metódusai

GDI+ két lehetőséget biztosít a ponttranszformációk végrehajtására. Az első lehetőség, hogy az általános affinitást megadó **Matrix** osztályt alkalmazzuk. A másik lehetőség, hogy használjuk a GDI+ transzformációs metódusait.

A **Graphics** osztály számtalan metódust biztosít a különböző ponttranszformációk használatára. Ezen metódusok egymás utáni végrehajtása nem más, mint a transzformációk szorzása, hiszen a transzformációs mátrixokat kell egymásután összeszorozni.

Tekintsük sorban a transzformációs mátrixokat, s az azoknak megfelelő **Graphics** vagy **Drawing2D.Matrix** osztálybeli metódusokat.

Az alkalmazott névterek (Namespace):

```

using System.Drawing; //Graphics
using System.Drawing.Drawing2D; //Matrix

```

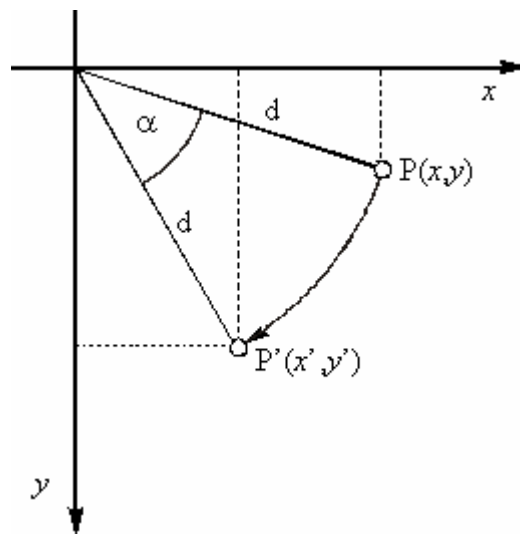
Eltolás:

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ dx & dy & 1 \end{bmatrix}, \quad \begin{aligned} x' &= x + dx, \\ y' &= y + dy. \end{aligned}$$

- `Graphics.TranslateTransform(dx, dy);`
- `Matrix myMatrix = new Matrix();`
`myMatrix.Translate(dx, dy);`

Elforgatás az origó körül alfa szöggel pozitív irányba:

Az y tengely állása miatt a szokásos pozitív iránnyal ellentétesen, az óramutató járásával megegyező irányt tekintjük pozitívnak.



Pozitív irányú forgatás a GDI+ ban.

A forgatás mátrixa:

$$M = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \begin{aligned} x' &= x \cdot \cos\alpha - y \cdot \sin\alpha, \\ y' &= x \cdot \sin\alpha + y \cdot \cos\alpha. \end{aligned}$$

- `e.Graphics.RotateTransform(alfa);`
//az alfát fokban kell megadni, s nem radiánban
- `myMatrix.Rotate(alfa);`
- `myMatrix.RotateAt(alfa, centerpont);`

Tükrözés:

Koordináta tengelyekre tükrözzünk. Erre nincs metódus a GDI+-ban, ezért nekünk kell elkészíteni a szükséges mátrixokat.

- x tengelyre való tükrözésnél minden y koordináta az ellenkezőjére változik:

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \begin{matrix} x' = x, \\ y' = -y. \end{matrix}$$

```
Matrix myMatrix = new Matrix(1,0,0,-1,0,0);
```

- y tengelyre való tükrözésnél minden x koordináta az ellenkezőjére változik:

$$M = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \begin{matrix} x' = -x, \\ y' = y. \end{matrix}$$

```
Matrix myMatrix = new Matrix(-1,0,0,1,0,0);
```

Az eltolást és az elforgatást összefoglalóan mozgásnak szokás nevezni, ugyanis ilyen esetben van olyan síkbeli mozgás, amivel az egybevágó alakzatok egymással fedésbe hozhatóak.

Skálázás:

Az x illetve az y tengely mentén $0 < s_x$, illetve $0 < s_y$ valós számokkal történő skálázás

$$M = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}, \begin{matrix} x' = s_x \cdot x, \\ y' = s_y \cdot y. \end{matrix}$$

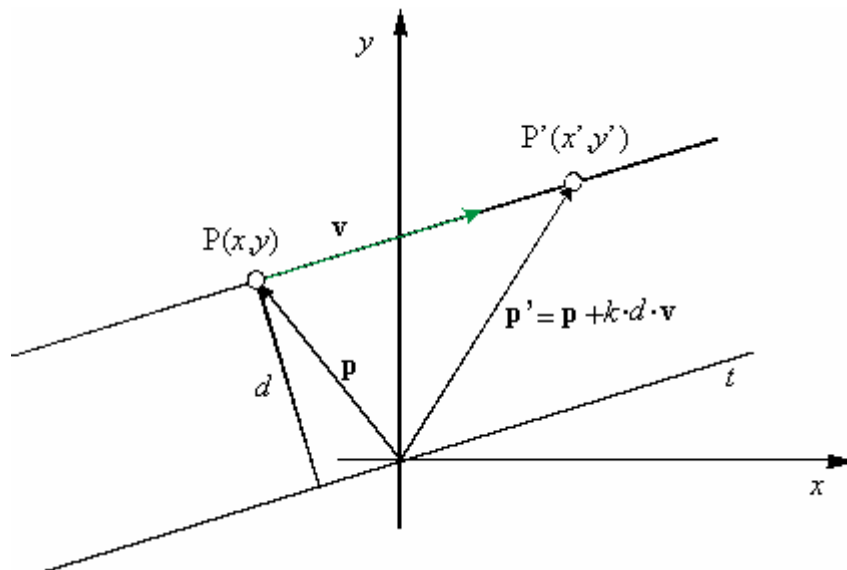
```
Graphics.ScaleTransform(sx,sy);
Matrix myMatrix = new Matrix();
myMatrix.Scale(sx,sy)
```

Ha $s_x = s_y$ akkor skálázás hasonlóság, azaz,:

- kicsinyítés ha $0 < s_x < 1$
- nagyítás, ha $1 < s_x$.

Nyírás:

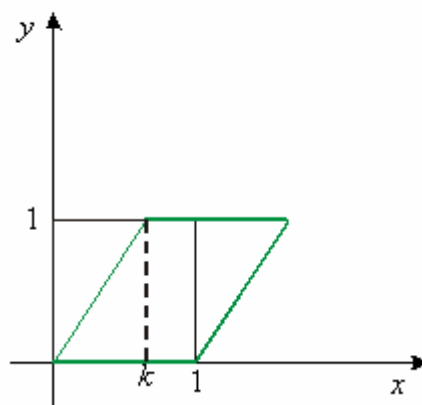
Tekintsünk egy pontonként fix t egyenest. A nyírás a sík pontjainak a t egyenessel párhuzamos elcsúsztatása, ahol a csúsztatás mértéke (k) arányos a t egyenestől való távolsággal (d).



Egy k mértékű nyírás a t pontonként fix tengellyel párhuzamosan

A GDI+-ban a koordináta tengelyek mentén vett nyírásokkal foglalkozunk. A **Shear** metódus két paramétere közül érdemes az egyiket nullának választani, attól függően, hogy x vagy y tengely irányában nyírunk, különben előre nehezen meghatározható hatást érünk el.

```
myMatrix.Shear(ShearX, ShearY);
```



Az x tengely irányú $ShearX=k$ mértékű nyírás

- Nyírás az x tengely irányában:

$$M = \begin{bmatrix} 1 & 0 & 0 \\ k & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \begin{aligned} x' &= x + k \cdot y, \\ y' &= y. \end{aligned}$$

```
myMatrix.Shear(k, 0);
```

- Nyírás az y tengely irányában:

$$M = \begin{bmatrix} 1 & k & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \begin{aligned} x' &= x, \\ y' &= y + k \cdot x. \end{aligned}$$

`myMatrix.Shear(0, k);`

A koordináta rendszer transzformálása és a Path használata

A GDI+-ban lehetőségünk van grafikus objektumok összefűzésére, azaz, path létrehozására. A **GraphicsPath** objektum lehetőséget nyújt a grafikus elemek egyetlen blokkként való kezelésére. A **Graphics** osztály **DrawPath** metódusának hívásával rajzolhatjuk meg a **GraphicsPath** szekvenciát, amelynek elemeit felsoroljuk, a felfűzéshez szükséges metódusokkal:

- Szakasz, AddLine, AddLines.
- Téglalap, AddRectangle, AddRectangles.
- Ellipszis, AddEllipse.
- Ellipszis ív, AddArc.
- Poligon, AddPolygon.
- Cardinal spline, AddCurve, AddClosedCurve.
- Bézier-görbe, AddBezier, AddBeziers.
- Körcikk, AddPie.
- Szöveg, AddString.
- Összefűzött grafikus objektumok, AddPath.

Tehát a path megrajzolásához szükségünk van a **Graphics**, a **Pen** és a **GraphicsPath** objektumokra. Lássunk egy példát:

```
{
    Graphics myGraphics= e.Graphics;
    GraphicsPath myGraphicsPath = new GraphicsPath();
    Point[] myPointArray = {
        new Point(40, 10),
        new Point(50, 50),
        new Point(180, 30)
    };

    FontFamily myFontFamily = new FontFamily("Times New Roman");
    PointF myPointF = new PointF(50, 20);
    StringFormat myStringFormat = new StringFormat();
    myGraphicsPath.AddArc(0, 0, 30, 20, -60, 230);
//Új figura kezdése
    myGraphicsPath.StartFigure();
    myGraphicsPath.AddCurve(myPointArray);
    myGraphicsPath.AddString("Hello!", myFontFamily,
3, 24, myPointF, myStringFormat);
    myGraphicsPath.AddPie(190, 0, 30, 20, -60, 230);
    myGraphics.DrawPath(new Pen(Color.Green,1), myGraphicsPath);
}
```



Nem szükséges, hogy a Path objektumai össze legyenek kötve

Grafikus konténerek

Grafikus állapotot a Graphics objektumban tároljuk. A GDI+ a konténerek használatával lehetővé teszi, hogy átmenetileg megváltoztassuk egy grafikus objektum állapotát. A **BeginContainer** és az **EndContainer** metódusok között bármilyen változtatást hajtunk végre a grafikus objektumon, az nem befolyásolja az objektum konténeren kívüli állapotát.

A következő példában különböző helyekre tesszük ki a téglalapba írt szöveget. Figyeljük meg a vonatkoztatási pont, az origó eltolását.

```
private void DrawHello(Graphics myGraphics)
{
    GraphicsContainer myContainer;
    myContainer = myGraphics.BeginContainer();
    Font myFont = new Font("Times New Roman", 26);
    StringFormat myStringFormat = new StringFormat();
    SolidBrush myBrush = new SolidBrush(Color.Gray);
    Pen myPen = new Pen(myBrush, 2);
    myGraphics.DrawRectangle(myPen, 0, 0, 100, 50);
    myGraphics.DrawString("Hello!", myFont, myBrush, 0, 0, myStringFormat);
    myGraphics.EndContainer(myContainer);
}

private void myExampleContainers(PaintEventArgs e)
{
    Graphics myGraphics = e.Graphics;
    GraphicsContainer myGraphicsContainer;
    // új origónk a (100,100) pontba kerül
    myGraphics.TranslateTransform(100, 100);
    // Hello! kiírása az új origóba.
    DrawHello(myGraphics);
    // Hello!-t újból kiírjuk, de először eltoljuk x irányba
    myGraphics.TranslateTransform(100, 0, MatrixOrder.Append);
    // Forgatás -30 fokkal a konténeren belül
    myGraphicsContainer = myGraphics.BeginContainer();
    myGraphics.RotateTransform(-30);
    DrawHello(myGraphics);
    myGraphics.EndContainer(myGraphicsContainer);
    // Hello!-t újból kiírjuk, de először tovább toljuk x irányba
    myGraphics.TranslateTransform(100, 0, MatrixOrder.Append);
    DrawHello(myGraphics);
    // Forgatás 45 fokkal és skálázás a konténeren belül
    myGraphicsContainer = myGraphics.BeginContainer();
    myGraphics.ScaleTransform(2, 1.5f);
    myGraphics.RotateTransform(45, MatrixOrder.Append);
    DrawHello(myGraphics);
    myGraphics.EndContainer(myGraphicsContainer);
}
```



Példa a konténerek használatára

Programozás tankönyv

XVII. Fejezet

„Adatok kezelése!”
Az ADO.NET

Radványi Tibor

Az ADO.NET

Adatkezelés C#-ban

SQL Server Enterprise Manager

Az SQL Server és az MMC

SQL Server Enterprise Manager a fő adminisztrációs eszköz a Microsoft® SQL Server™ 2000-hez, és biztosítja az Microsoft Management Console (MMC)-t, egy jól használható felhasználói felületet, mely a felhasználó számára lehetővé teszi, hogy:

- Meghatározhatunk szervercsoportokat a futó SQL Serveren.
- Egyedi szervereket is bejegyezhetünk egy csoportba.
- Az összes bejegyzett SQL server konfigurálása.
- Létrehozhatunk és adminisztrálhatunk minden SQL Server adatbázisokat, objektumokat, login-okat, felhasználókat és engedélyezéseket minden bejegyzett szerverre..
- Meghatározhatunk és elvégezhetünk minden SQL Server adminisztrációs feladatot minden bejegyzett szerveren.
- Szerkeszthetünk és tesztelhetünk SQL utasításokat, Batch-eket, és szkripteket interaktívan az SQL Query Analyzer segítségével.
- Segítségül hívhatjuk a sok előre elkészített varázslót az SQL Serverhez.

MMC egy olyan eszköz, mely egy általános felületet biztosít különböző szerver alkalmazások menedzselésére egy Microsoft Windows® hálózatban. Server alkalmazások létrehoznak egy komponenst, melyet az MMC használ, azáltal az MMC felhasználók egy felhasználói felület segítségével menedzselhetik a server alkalmazást. SQL Server Enterprise Manager a Microsoft SQL Server 2000 MMC komponense.

Az SQL Server Enterprise Manager elindításához válasszuk ki az Enterprise Manager ikont a Microsoft SQL Server programcsoportban. Amelyik gépen Windows 2000 fut, ott ugyanúgy indíthatjuk az SQL Server Enterprise Managert a Felügyeleti Eszközökben a Vezérlőpultban. Az MMC beillesztések is a Felügyeleti Eszközökből indíthatók. Ehhez nem kell alapbeállításként engedélyeznünk gyerekablak nyitásának lehetőségét. Ez az opció csak az SQL Server Enterprise Manager használatához kell.

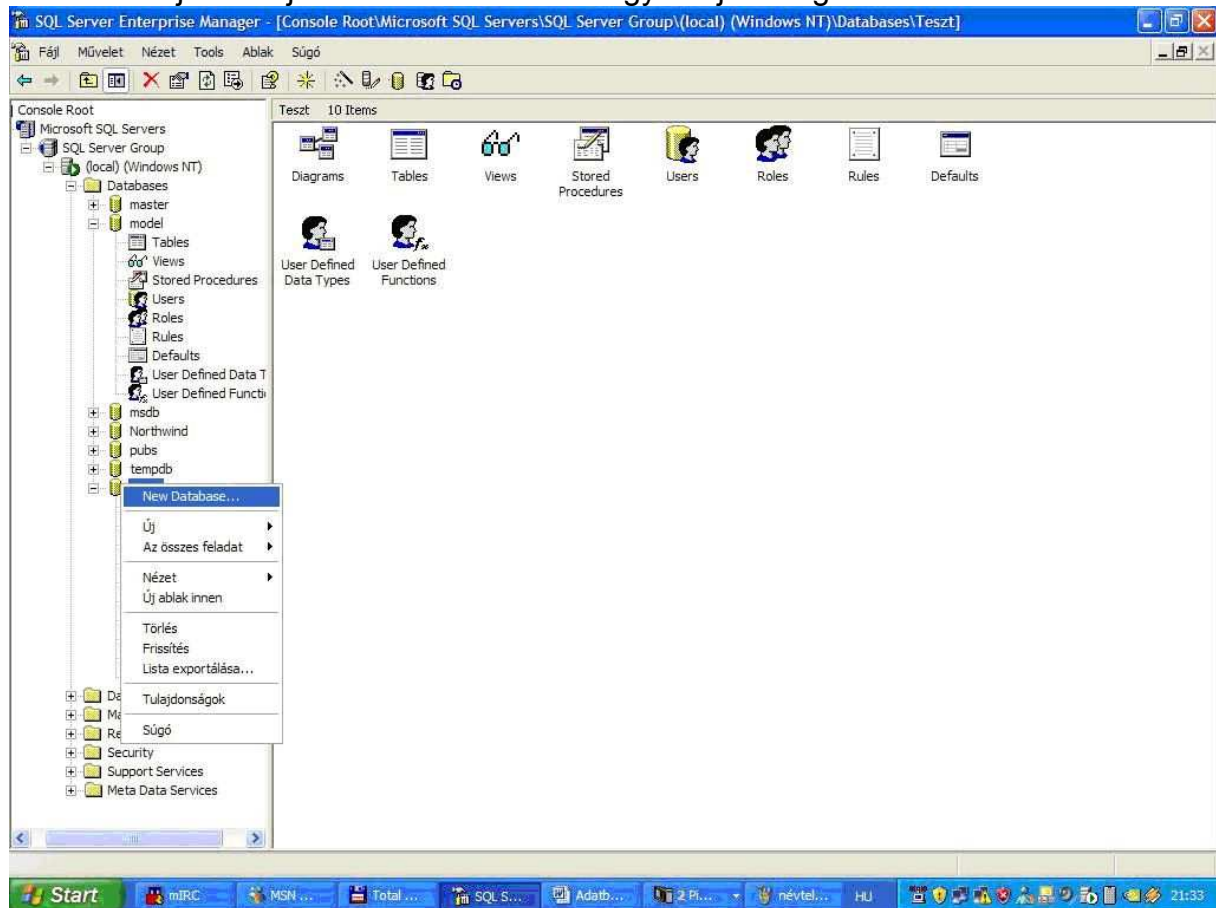
Megjegyzés Ha bejegyzünk egy új SQL szerveret a Felügyeleti Eszközökben, és utána vagy bezérjük a Felügyeleti Eszközöket, vagy egy másik számítógéphe csatlakozunk, a szerver már nem fog látszani a Felügyeleti Eszközökben. A bejegyzett szerver a SQL Server Enterprise Managerben fog látszódni

Új elemek létrehozása

Feltelepítés után, ha elindítjuk az Enterprise Managert, az elénk táruló kép hasonló más adatbázis menedzselő programokhoz. A bal oldali panel treeview-ját ha lenyitjuk, láthatjuk a szervercsoportjainkat, azon belül a hozzáadott szervereinket.

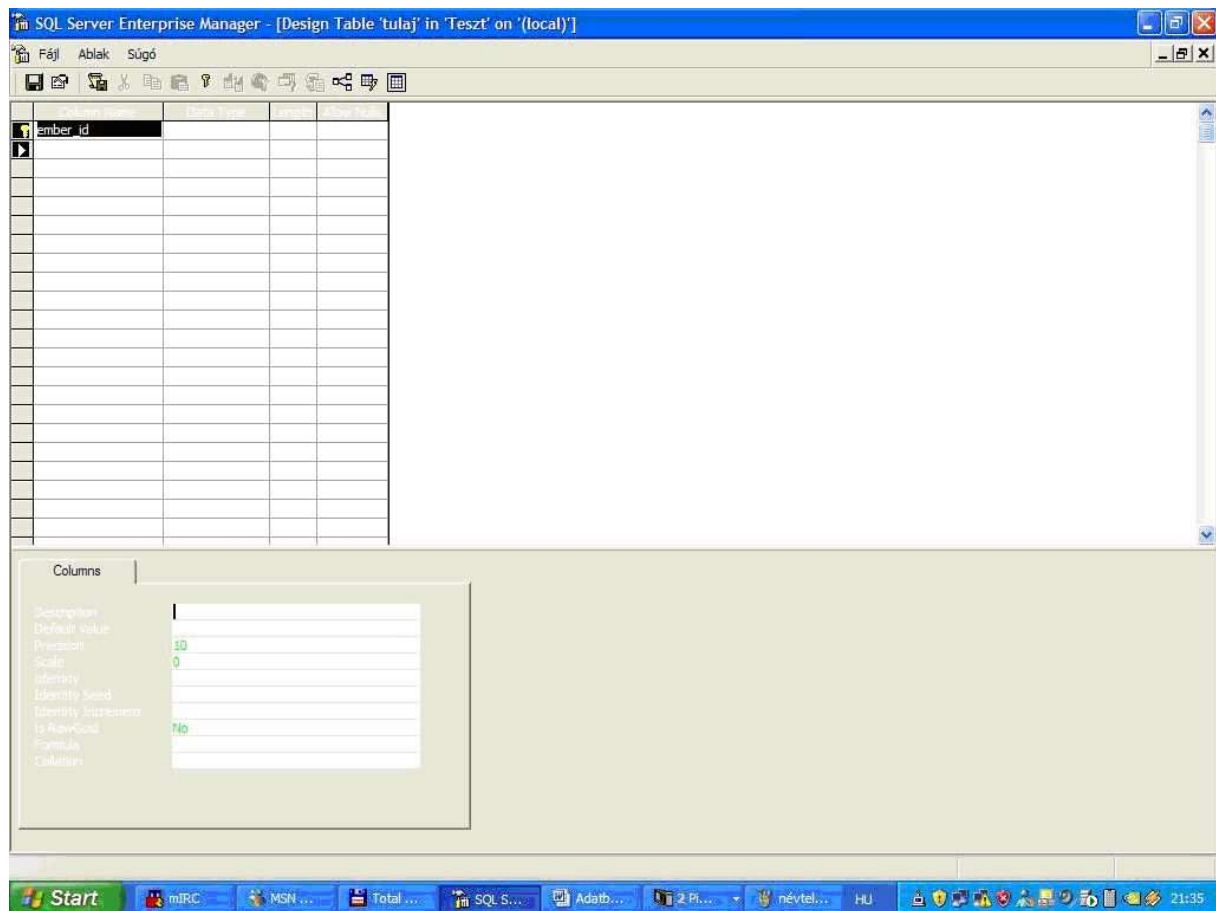
Az egyes szervereket lenyitva kapjuk meg azok beállítási, lehetőségeit, illetve jellemzőinek leírását. Itt találhatóak az adatbázisok is.

Az egyes elemekre jobb gombot nyomva kapjuk meg azt a pop-up ablakot melyből kiválaszthatjuk az új elem létrehozását vagy tulajdonságainak beállítását.



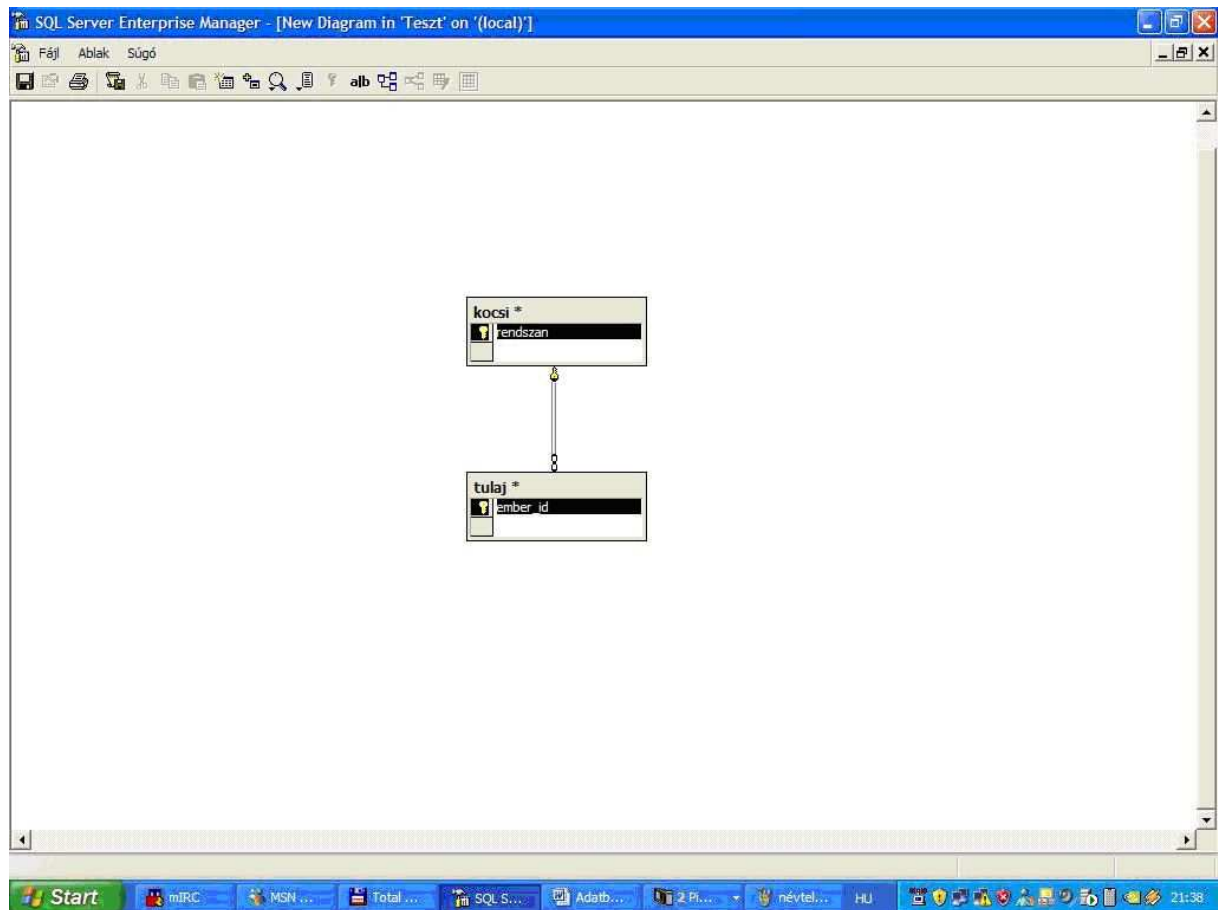
A Databases/New DataBase-re kattintva létrehozhatunk egy új adatbázist. Itt meg kell adnunk annak nevét, és kiválaszthatjuk az adattárolás könyvtárát is illetve a merevlemez szabad és adatméret kezelésének módját is. Megadhatjuk, hogy az adatbázis szerver hova mentse a napló fájlokat. Ezeknek a méretét maximálhatjuk. Mikor létrehoztuk az adatbázist, létrejön az adatbázis struktúra is, mely tartalmazza Táblákat, Nézeteket, Felhasználókat, Tárolt eljárásokat.

A tábla létrehozása úgy történik mint minden hasonló rendszerben. Megadjuk a mezőneveket, azok típusait, hosszukat és hogy engedélyezzük-e a NULL értéket. Ezeken kívül megadhatunk még az egyes mezőkről leírást, alapértelmezett értéket, Formulát és karakterkészletet. Megadhatjuk, hogy egy mező egyedi legyen-e. A kulcsok kiválasztásánál az adott mezőn jobb gombot kell nyomnunk és ott a Set Primary Key opcióval elsődleges kulcsnak választjuk az elemet. Bezáráskor mehetjük a táblát és adhatunk neki nevet. Ha a listában lévő, már létező táblákat akarjuk szerkeszteni, akkor azt az adott táblán jobb gombot nyomba a Design Table menüpontban tehetjük meg.



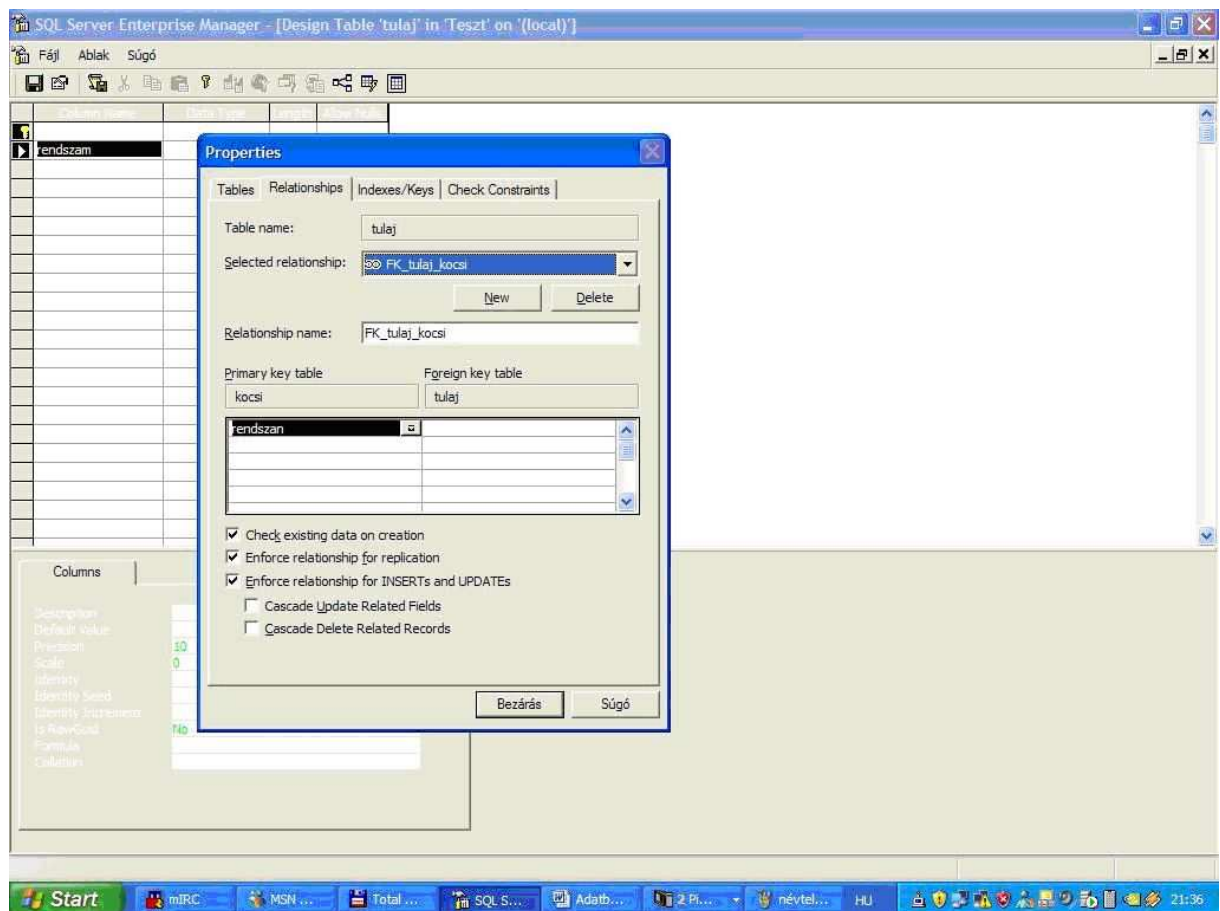
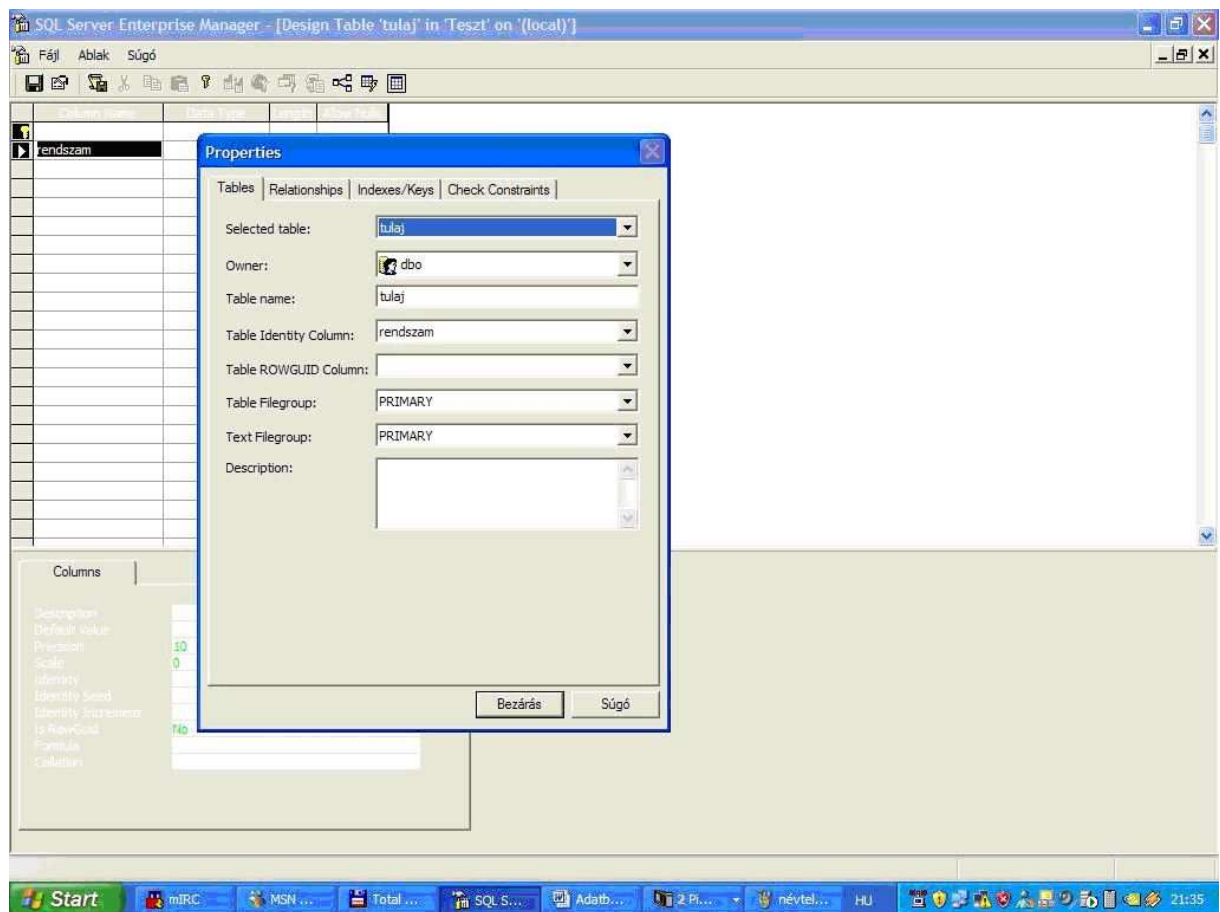
Kapcsolatokat ugyanolyan típusú mezők közt hozhatunk létre. Ezt az eszköztáron található Manage Relationshipel tehetjük meg.

Az adatbázis elemei között található a Diagram is. Ez arra szolgál, hogy látványosan szerkeszthessük a tábláinkat és a köztük lévő kapcsolatot. Amikor ebből újat hozunk létre egy varázsló segít összeállítani a megjelenítendő táblákat. Beállíthatjuk, hogy a kapcsolódó táblákatt automatikusan a listába rakja-e. A megjelenő tábláknál, ha a már Access-ben megszokott egyik elemről a másikig húzzuk lenyomva az egeret, akkor megjelenik a kapcsolatokat létrehozó form kitöltve a megfelelő adatokkal. A táblán jobb hombra kattintva sok beállítási lehetőséget kapunk annak szerkesztésére, illetve megjelenítésére. Például, hogy akár csak a kulcsokat lássuk, vagy csak a tábla neveit. Ez a nagyobb projecteknél, a sok és sokelemű tábláknál nyújthat nagy segítséget a tervezésben. A kapcsolatot jelző összekötő szakaszt is teszteszabhatjuk, feliratozhatjuk.

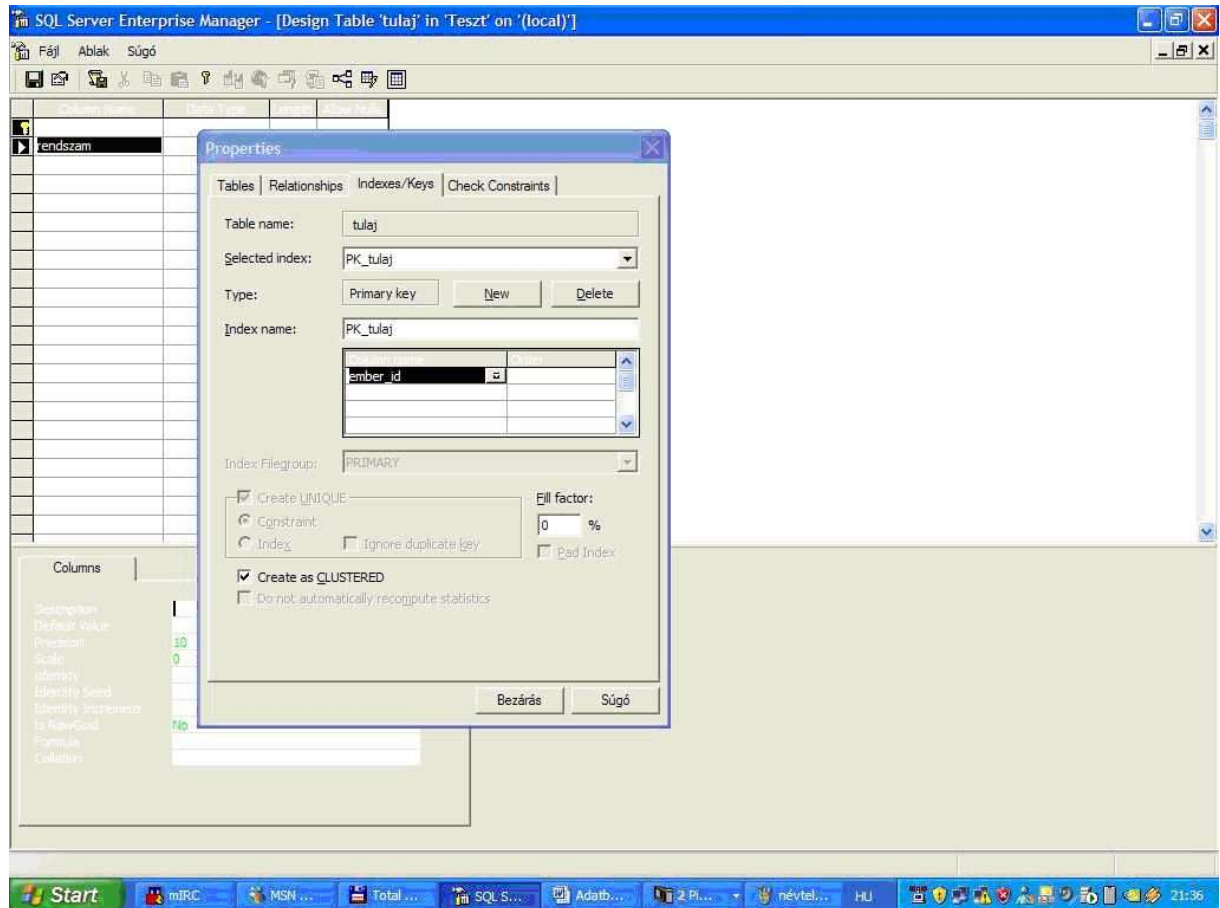


Nézzük meg jobbat az a formot, ahol a kapcsolatokat állítottuk be. Itt a táblák teljes testreszabását elvégezhetjük

- A első fül a Table: Itt az egyes táblák általános beállításai végezhetjük el. Mint például a tábla neve, tulajdonosa, egyedi mezője és a tábla leírása.
- Columns: Itt az egyes mezőket szabhatjuk testre. Majdnem minden olyan beállítást elvégezhetünk mint amit a Design table alatt. A table name mellé kattintva kapunk egy ComboBox-ot melyből kiválaszthatjuk, hogy melyik táblát akarjuk módosítani. A Column name-el pedig az adott mezőt választhatjuk ki.
- Relation: A kapcsolatokat. Az említett beállításokon kívül még be lehet állítani, hogy hogyan módosuljon a gyerek tábla törléskor és módosításkor.



- Indexes/Keys: Itt hozhatunk létre új indexeket és a létezőket testre szabhatjuk. A kiválasztott mező mellett meg kell adnunk, hogy növekvő vagy csökkenő indexet szeretnénk hozzárendelni.



MS SQL szerver elérése C#-ból

Az ADO.NET használata, SqlConnection osztály a formon dinamikusan, és kevés programozással.

Bevezetés az ADO.NET kapcsolódási eszközeihez

Egy alkalmazás és egy adatbázis közti adatcseréhez először egy kapcsolat kell az adatbázishoz. ADO.NET-ben kapcsolódásokat létrehozni és kezelni kapcsolódási objektumokkal lehet:

SqlConnection – egy objektum, mely kezeli az SQL Server-hez a kapcsolatot. Optimalizálva van SQL Server 7.0 vagy újabb verzióhoz való használatra, az OLE DB réteget megkerülve.

OleDbConnection – egy objektum, mely kapcsolatot kezeli bármely adattárolóhoz, mely OLE DB-n keresztül elérhető.

OdbcConnection – egy objektum, mely kapcsolatot kezel egy adatforráshoz, melyet vagy kapcsolat stringgel (connection string) vagy ODBC adatforrás névvel hoztak létre.

OracleConnection – egy objektum, mely Oracle adatbázisokkal való kapcsolatot kezeli.

Connection String

Minden kapcsolódási objektum nagyjából ugyanazon tagokkal rendelkezik.

Az elsődleges tulajdonság egy kapcsolat-objektumnak a ConnectionString, mely egy string-ből áll. A string-ben mező-érték párokat találunk, ez az információ szükséges egy adatbázisba bejelentkezéshez, és egy meghatározott adatbázis eléréséhez. Egy tipikus ConnectionString hasonlóképpen néz ki:

```
"Provider=SQLOLEDB.1; Data Source=MySQLServer; Initial Catalog=NORTHWIND;  
Integrated Security=SSPI"
```

Ez a connection string részlet megadja, hogy a kapcsolat a Windows beépített védelmét használja (NT hitelesítés). Egy connection string-ben szerepelhet e helyett egy felhasználónév és jelszó, de nem javasolt, mert ezek az értékek a programban lesznek eltárolva (fordításkor), ezért nem biztonságos.

Kapcsolat létrehozása és megszüntetése

Két elsődleges metódus van kapcsolatokhoz: Open és Close. Az Open metódus a ConnectionString-ben lévő információt használja az adatforrás eléréséhez és egy kapcsolat kiépítéséhez. A Close metódus lebontja a kapcsolatot. A kapcsolat

bontása lényeges, mivel a legtöbb adatforrás csak korlátozott számú kiépített kapcsolatot enged, és a kiépített kapcsolatok értékes erőforrásokat foglalnak.

Ha adatillesztőkkel vagy adatparancsokkal dolgozunk, nem kell állandóan magunknak kiépíteni és bontani a kapcsolatot. Ha a fenti objektumok egy metódusát hívjuk meg (pl. az adatillesztő Fill vagy Update metódusa), a metódus ellenőrzi, hogy a kapcsolat már ki van-e építve. Ha nincs, az illesztő kiépíti a kapcsolatot, végrehajtja a feladatát, majd bontja a kapcsolatot.

A metódusok – mint a Fill – csak akkor építik ki és bontják a kapcsolatot automatikusan, ha még nincs kiépítve. Ha van kiépített kapcsolat, a metódusok felhasználják, de nem bontják le. Ez lehetőséget nyújt adatparancsok flexibilis, saját kezű kiépítésére és bontására. Ezt használhatjuk, ha több adatillesztőnk osztozik egy kapcsolaton. Ebben az esetben nem hatékony, ha minden adatillesztő külön épít ki és bont kapcsolatot, ha meghívja a Fill metódusát. Ehelyett használhatunk egy kapcsolatot, minden illesztőhöz meghívhatjuk a Fill metódust, majd végezetül bonthatjuk a kapcsolatot.

Összetett kapcsolatok (Pooling Connections)

Az alkalmazásoknak gyakran vannak különböző felhasználói, akik azonos típusú adatbázis elérést végeznek. Például, ASP.NET Web alkalmazásokban sok felhasználó kérdezhet le ugyanattól az adatbázistól, hogy ugyanazt az adatot kapják. Ezekben az esetekben az alkalmazások teljesítménye növelhető, ha az alkalmazás megosztja, összeadja (pool) a kapcsolatot az adatforráshoz.

SqlConnection osztály használatakor a kapcsolatok megosztása automatikusan kezelve van, de rendelkezésre állnak lehetőségek, hogy a megosztást magunk kezeljük.

Tranzakciók

A kapcsolat objektumok támogatják a tranzakciókat, a BeginTransaction metódussal egy tranzakció-objektum jön létre (pl. egy SqlTransaction objektum). A tranzakció objektum pedig rendelkezik metódusokkal, melyek engedik végrehajtani vagy visszavonni a tranzakciót. A tranzakciókat kódban kezeljük.

Beállítható kapcsolat tulajdonságok

A legtöbb alkalmazásban a kapcsolat információi nem határozhatók meg tervezési időben. Például egy olyan alkalmazásban, melyet több vásárló is használni fog, nem adhatjuk meg a kapcsolatra vonatkozó adatokat tervezéskor – mint a szerver neve, stb.

A kapcsolat beállításait ezért gyakran dinamikus tulajdonságként kezeljük. Mivel a dinamikus tulajdonságok egy konfigurációs fájlban tárolódnak (és nem fordítódnak az alkalmazás bináris fájljaiba), tetszőlegesen változtathatóak.

Tipikus módszer a kapcsolat tulajdonságainak dinamikus adatokként való létrehozása. A felhasználónak ilyenkor valamilyen módot kell nyújtani (Windows

Form vagy Web Form), hogy a fontos adatokat meghatározza, majd frissítse a konfigurációs fájlt. A .NET Framework-be épített dinamikus tulajdonság szerkezet automatikusan megkapja az értékeket a konfigurációs fájlból, amikor a tulajdonság kiolvasódik, és frissíti a fájlt, ha az érték változik.

Kapcsolatok tervezésekor a Server Explorer-ben

A Server Explorer lehetőséget ad tervezéskor, hogy adatforrásokhoz kapcsolatot létrehozzunk. Tallózhatunk a meglévő adatforrások között, megjeleníthetünk információkat a táblákról, oszlopokról, és egyéb elemekről, amiket tartalmaz, létrehozhatunk és szerkeszthetünk adatbázis elemeket.

Az alkalmazásunk nem közvetlenül használja az ez úton létrehozott kapcsolatokat. Általában, a tervezés közben létrehozott kapcsolat információit arra használjuk, hogy tulajdonságokat állítsunk be egy új kapcsolat objektumhoz, amit az alkalmazáshoz adunk.

A tervezési időbe létrehozott kapcsolatokról az információkat a saját gépünkön tároljuk, függetlenül egy meghatározott projekttől vagy Solution-től. Ezért miután már létrehoztunk egy kapcsolatot tervezéskor, az meg fog jelenni a Server Explorer ablakban amikor egy alkalmazáson dolgozunk, mindig látható lesz a Visual Studio-ban (mindaddig, amíg a szerver elérhető, melyre a kapcsolat mutat).

Kapcsolat tervezési eszközök Visual Studio-ban

Általában nincs szükség közvetlenül létrehozni és kezelni kapcsolat objektumokat Visual Studio-ban. Ha olyan eszközöket használunk, mint a Data Adapter varázsló, az eszközök kérni fogják a kapcsolat adatait (azaz a connection string információkat) és automatikusan létrehoznak kapcsolat objektumokat a Form-on vagy komponensen, amin dolgozunk.

Mindamellet, ha akarjuk, magunk is hozzáadhatunk kapcsolat objektumokat a Form-hoz vagy komponenshez, és beállíthatjuk azok tulajdonságait. Ez hasznos, ha nem adatillesztőkkel dolgozunk, hanem csak adatokat olvasunk ki. Létrehozhatunk kapcsolat objektumokat, ha tranzakciókat akarunk használni.

Kapcsolat létrehozása SQL Server-hez ADO.NET használatával

A .NET Framework beépített Data Provider for SQL Server (adatszolgáltató SQL szerverhez) SqlConnection objektummal elérést nyújt Microsoft SQL Server 7.0 vagy újabb változatához.

A Data Provider for SQL Server hasonló formátumú kapcsolat stringet támogat, mint az OLE DB (ADO) kapcsolat string formátuma.

Az alábbi kód szemlélteti, hogyan építhetünk ki kapcsolatot egy SQL Server adatbázishoz:

```
SqlConnection kapcsolat = new SqlConnection("Data Source=localhost;  
Integrated Security=SSPI; Initial Catalog=adattabla");  
kapcsolat.Open();
```

Kapcsolat bontása

Javasolt mindig a kapcsolat bontása, ha befejeztük a használatát. Ehhez a kapcsolat objektumnak vagy a Close vagy a Dispose metódusát kell meghívni. Azok a kapcsolatok, melyeket nem zárunk be saját kezűleg, nem juthatnak osztott kapcsolathoz.

ADO.NET kapcsolat objektumok létrehozása

Ha adatelérés tervező eszközöket használunk Visual Studio-ban, általában nem szükséges magunknak létrehoznunk a kapcsolat objektumokat a Form-on vagy komponensen. Azonban néhány esetben alkalmasnak találhatjuk egy kapcsolat saját létrehozását.

Lehetőségeink:

Az alábbi eszközökből használhatjuk valamelyiket, mely a feladata részeként kapcsolat objektumot hoz létre:

- Data Adapter Configuration Wizard – ez a varázsló információkat kér a kapcsolatról, mely egy adatillesztőhöz lesz kapcsolva.
- Data Form Wizard – a varázsló kapcsolat objektumot hoz létre a Form részeként, melyet konfigurál.
- Húzzunk egy táblát, kijelölt oszlopokat vagy tárolt eljárást a Form-ra a Server Explorer-ből. Ezen elemek Formra áthúzásakor létrejön egy adatillesztő és egy kapcsolat is.
- Hozzunk létre egy különálló kapcsolatot. Ez a lehetőség létrehoz egy kapcsolat objektumot a Form-on vagy komponensen, melynek a tulajdonságait magunknak kell beállítanunk. Ez a megoldás akkor hasznos, ha a kapcsolat tulajdonságait futási időben szándékozzuk beállítani, vagy ha egyszerűen a tulajdonságokat magunk szeretnénk beállítani a Properties ablakban.
- Hozzunk létre kapcsolatot kóddal.

Kapcsolat létrehozása

1. A Toolbox ablak Data csoportjából húzzunk egy kapcsolat objektumot a Form-ra vagy komponensre.
2. Válasszuk ki a kapcsolatot a tervezőben és használjuk a Properties ablakot a connection string beállításához.
3. Beállíthatjuk a ConnectionString-et egy egységként.
4. Vagy
5. Külön tulajdonságait állíthatjuk (DataSource, Database, Username, stb.). Ha külön állítjuk a tulajdonságokat, a kapcsolat string létrejön automatikusan.
6. Átnevezhetjük a kapcsolat objektumot a Name tulajdonság változtatásával.
7. Ha futási időben szeretnénk a tulajdonságokat beállítani, az alkalmazás újrafordítása nélkül, meg kell határozni a kapcsolat tulajdonságait.

Kapcsolat létrehozása SQL Server-hez

Ha SQL Server-hez kapcsolódunk, használjuk a Microsoft OLE DB Provider for SQL Server eszközt. Két módon kapcsolódhatunk szerverhez:

- Vizuálisan, tervezési eszközökkel hozzunk létre kapcsolatot.
- Programkóddal hozzunk létre a kapcsolatot.

SQL Server kapcsolat Server Explorer-ben

Egyszerűen hozhatunk létre SqlConnection, SqlDataAdapter és SqlCommand objektumokat úgy, hogy a Server Explorerből a Formra húzzuk őket.

Kapcsolat létrehozása:

1. A Server Explorer-ben kattintsunk jobb egérgombbal a Data Connections-re, és válasszuk az Add Connection menüpontot. Megjelenik a Data Link Properties párbeszédablak.
2. Az alapértelmezett elérés a Microsoft OLE DB Provider for SQL Server.
3. Válasszuk ki egy szerver nevét a lenyíló listából, vagy gépeljük be a szerver helyét, ahol az elérni kívánt adatbázis található.
4. Az alkalmazásunk vagy az adatbázisunk szükségleteihez mérten válasszuk ki vagy a Windows NT Integrated Security-t, vagy adjunk meg felhasználónevet és jelszót a bejelentkezéshez.
5. Válasszuk ki az elérni kívánt adatbázist a lenyíló listából.
6. Kattintsunk az OK gombra.

Kapcsolódás SQL Server-hez az alkalmazásunkból

Kapcsolódás létrehozása vizuális eszközökkel

A kapcsolódási eszközöket vagy a Server Explorer-ből vagy a Toolbox ablak Data csoportjából használva hozzunk létre kapcsolatot:

Server Explorer-ből

Hozzunk létre egy Data Connection-t a Server Explorer-ben az SQL Server-hez a fentebb leírt módon.

Húzzuk a létrehozott kapcsolatot a Form-ra. Egy SqlConnection objektum jeleni meg a komponens-tálcán.

Toolbox-ból

Húzzunk egy SqlConnection objektumot a Form-ra. Egy SqlConnection objektum jelenik meg a komponens-tálcán, mely nincs beállítva.

A Properties ablakban válasszuk ki a ConnectionString property-t. Válasszuk ki egy meglévő kapcsolatot a lenyíló listából vagy kattintsunk a New Connection-re, és állítsuk be a kapcsolatot.

A kapcsolat kiépítése után az adatbázist kezelő eljárásokat kell létrehozni.

A DataTable osztály

Adatokat nem csak úgy kezelhetünk egy DataGrid-ben, hogy azok valóságos adatbázishoz kapcsolódnak. Lehetőségünk van arra is, hogy egy tetszőleges adatforrást használjunk, vagy akár programból generáljunk adatokat, melyeket ezek után épp úgy kezelhetünk, mintha egy tetszőleges típusú adatbázis egy-egy táblája lenne.

A megoldás kulcsa a DataTable osztályban rejlik. Ezt az osztályt használhatjuk valós adattáblák kezeléséhez épp úgy, mint a programból generált adatainkhoz.

Szűrés és rendezés az ADO.NET-ben

Vizsgáljuk meg, hogy lehet a memóriában, DataSetben tárolt adatokat tovább szűrni, illetve rendezni. Az ADO.NET két megközelítést támogat erre a műveletre:

1. A DataTable **Select** metódusának használatát. Ez a metódus szűrt és rendezett adatsorok tömbjével tér vissza.
2. A DataView objektum **filter**, **find** és **sort** metódusai. Ez az objektum hozzákapcsolható az adatmegjelenítésre képes objektumokhoz.

Nézzünk egy példát a szűrőfeltétel felépítésére:

```
„OrderDate >= '01.03.1998' AND OrderDate <= '31.03.1998'”
```

A tipikus rendezési kifejezés: a mező neve és a rendezés iránya. Ami a DESC (csökkenő) vagy az ASC (növekvő) szavakkal határozható meg.

```
„OrderDate DESC”
```

A következő kód egy példa a DataTable **Select** metódusának használatára:

```
private static void GetRowsByFilter()
{
    DataTable customerTable = new DataTable( "Customers" );
    customerTable.Columns.Add( "id", typeof(int) );
    customerTable.Columns.Add( "name", typeof(string) );

    customerTable.Columns[ "id" ].Unique = true;
    customerTable.PrimaryKey = new DataColumn[]
{CustomerTable.Columns["id"] };

    // Tíz sor hozzáadása a táblához
    for( int id=1; id<=10; id++ )
    {
        customerTable.Rows.Add(
```

```

        new object[] { id, string.Format("customer{0}", id) } );
    }
    customerTable.AcceptChanges();

    // Újabb tíz sor hozzáadása
    for( int id=11; id<=20; id++ )
    {
        customerTable.Rows.Add(
            new object[] { id, string.Format("customer{0}", id) } );
    }

    string strExpr;
    string strSort;

    strExpr = "id > 5";
    // Csökkenő sorrend a CompanyName nevű mezőben.
    strSort = "name DESC";

    DataRow[] foundRows = customerTable.Select( strExpr, strSort,
    DataRowState.Added );

    PrintRows( foundRows, "filtered rows" );

    foundRows = customerTable.Select();
    PrintRows( foundRows, "all rows" );
}

private static void PrintRows( DataRow[] rows, string label )
{
    Console.WriteLine( "\n{0}", label );
    if( rows.Length <= 0 )
    {
        Console.WriteLine( "no rows found" );
        return;
    }
    foreach( DataRow r in rows )
    {
        foreach( DataColumn c in r.Table.Columns )
        {
            Console.Write( "\t {0}", r[c] );
        }
    }
}

```

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

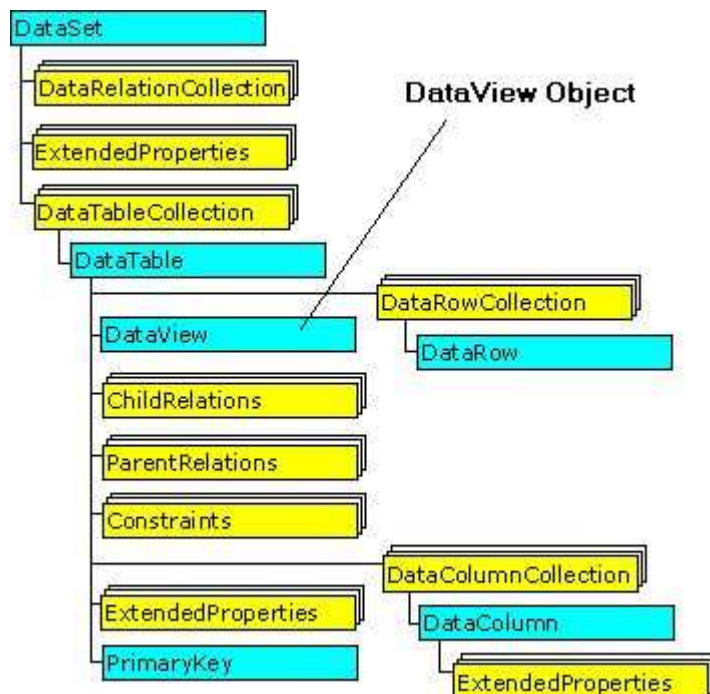
Az alapvető probléma a DataTable Select módszerével, hogy a szűrés eredményeként kapott adatsorokat egy tömbben adja vissza. Ez nem köthető sem a DataGridViewhez sem más adatmegjelenítésre alkalmas objektumhoz közvetlenül. Erre a DataView használata ad lehetőséget.

Szűrés és rendezés a DataView objektum segítségével

A DataView objektum lehetővé teszi, hogy a DataTable-ban tárolt adatokhoz különböző nézeteket hozzunk létre. Ezt a lehetőséget sokszor használjuk ki adatbázishoz kapcsolódó programoknál.

A DataView használatával a táblabeli adatokat különböző rendezési szempont szerint, illetve a sorok állapota, vagy kifejezések által meghatározott szűrők szerint mutathatjuk meg.

Ez abban különbözik a DataTable Select módszerától, hogy az eredmény sorokat nem egy tömbben adja vissza, hanem egy dinamikus táblában. Ez a képessége a DataView objektumot ideális eszközzé teszi az adatkapcsolatokat kezelő programok számára.



A DataView objektum helye az ADO.NETben

Az alapértelmezett nézet

A `DataTable.DefaultView` tulajdonság a `DataRowView` objektumot kapcsolja a `DataTable`-hoz. Így itt is használható a rendezés, szűrés és keresés a táblában.

A `RowFilter` tulajdonság

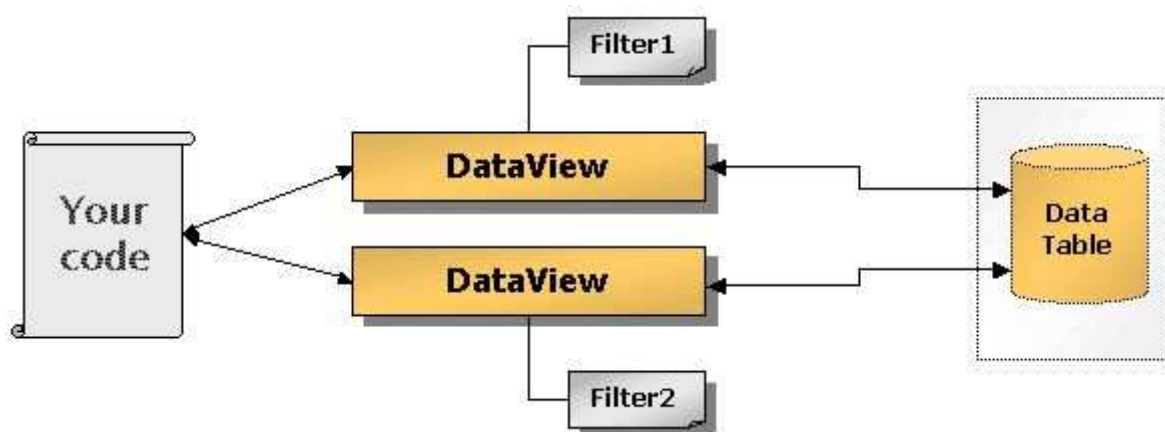
A `DataRowView.RowFilter` tulajdonsága adhatunk meg egy szűrő feltételt a sorok megjelenítéséhez a `DataRowView`-ban. A szűrőkifejezés felépítése: egy oszlopnév, operátor és egy érték a szűréshez.

```
„ LastName = 'Smith' ”
```

Rendezés a `DataRowView`-ban

A rendezéshez létre kell hoznunk egy string kifejezést, melyben megadhatjuk, hogy mely oszlopok szerint szeretnénk rendezni a sorokat, és milyen irányban.

```
„Price DESC, Title ASC”
```



Egy táblához több nézet

Nézzünk egy példát erre:

```
using System;
using System.Diagnostics;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Data.SqlClient;

namespace Akadia.DataView
{
    public class FilterOrder : System.Windows.Forms.Form
    {
        ....
        private SqlConnection cn;
        private SqlCommand cmd;
        private SqlDataAdapter da;
        private DataSet ds;

        public FilterOrder()
        {
            try
            {
                InitializeComponent();

                // Initializing
                cn = new SqlConnection("
server=xeon;database=northwind;uid=sa;pwd=manager");
                cmd = new SqlCommand("SELECT * FROM orders",cn);
                da = new SqlDataAdapter(cmd);
                ds = new DataSet();

                // Kezdő adatok betöltése
                RetrieveData();
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.ToString());
                Console.WriteLine();
            }
        }
    }
}
```

```

    }
}

// Orders Tábla adatai
private void RetrieveData()
{
    try
    {
        da.Fill(ds, "Orders");
        DataGridView.DataSource = ds.Tables[0];

        // Combobox feltöltése a mezőnevekkel
        FillSortCriteria();
    }
    catch (Exception ex)
    {
        Debug.WriteLine(ex.ToString());
        MessageBox.Show(ex.ToString());
    }
}

// Combobox feltöltése a mezőnevekkel
private void FillSortCriteria()
{
    try
    {
        if (cmbSortArg.Items.Count > 0)
        {
            return;
        }
        foreach (DataColumn dc in ds.Tables[0].Columns)
        {
            cmbSortArg.Items.Add(dc.Caption); // Sort Combobox
            cmbFields.Items.Add(dc.Caption); // Filter on Column Combobox
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.ToString());
        Console.WriteLine();
    }
}

```

```

    }

    // Setup Szűrőfeltétel beállítása
    private void SetFilter(string strFilterExpression)
    {
        try
        {
            ds.Tables[0].DefaultView.RowFilter = strFilterExpression;
            // Kiolvassuk a rekordszámot a DataViewban
            if (ds.Tables[0].DefaultView.Count > 0)
            {
                DataGrid.DataSource = ds.Tables[0].DefaultView;
            }
            else
            {
                MessageBox.Show("Filter criteria does not meet criteria");
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.ToString());
            Console.WriteLine();
        }
    }

    private void btnQuery_Click(object sender, System.EventArgs e)
    {
        try
        {
            // Clear DataSet
            ds.Clear();

            // Clear Filter
            ds.Tables[0].DefaultView.RowFilter = "";

            // Re-Retrieve Data
            RetrieveData();
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.ToString());
            Console.WriteLine();
        }
    }

```

```

    }
}

// Beállítjuk a DataViewban a rendezést
private void btnSort_Click(object sender, System.EventArgs e)
{
    try
    {
        string strSort;

        // IF Radiobox "Ascending" is checked, then
        // sort ascending ...
        if (rbAsc.Checked)
        {
            strSort = cmbSortArg.Text + " ASC";    // Note space after "
        }
        // ... else descending
        else
        {
            strSort = cmbSortArg.Text + " DESC";    // Note space after "
        }

        // Érvényesítjük a rendezést
        ds.Tables[0].DefaultView.Sort = strSort;
        DataGrid.DataSource = ds.Tables[0].DefaultView;
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.ToString());
        Console.WriteLine();
    }
}

private void btnFilterTitle_Click(object sender, System.EventArgs e)
{
    try
    {
        SetFilter("CustomerID like '" + txtFilter.Text + "'");
    }
    catch (Exception ex)

```

```

        {
            MessageBox.Show(ex.ToString());
            Console.WriteLine();
        }
    }

private void btnGeneralFilter_Click(object sender, System.EventArgs e)
{
    try
    {
        SetFilter(txtGeneralFilter.Text);
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.ToString());
        Console.WriteLine();
    }
}

private void btnFilteronColumn_Click(object sender, System.EventArgs e)
{
    try
    {
        SetFilter(cmbFields.Text + " " + txtFilterColumn.Text);
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.ToString());
        Console.WriteLine();
    }
}

...
[STAThread]
static void Main()
{
    Application.Run(new FilterOrder());
}
}

```

Tárolt eljárások

Tárolt eljárások futtatása, új tárolt eljárások készítése, rögzítése adatbázisban, használatba vételük.

Mi is az a Transact-SQL?

A Transact-SQL a Microsoft SQL megvalósítása, amely programozási szerkezetekkel egészíti ki a nyelvet. A nevét gyakran rövidítik T-SQL –re. A T-SQL segítségével, olyan SQL utasításokat tartalmazó programokat írhatunk, amelyekben a szokványos programozási szerkezetek is megtalálhatók (változók, feltételes szerkezetek, ciklusok, eljárások és függvények).

Alapvető programozási szerkezetek:

- *Változók használata*
- Feltételes szerkezetek használata
- Case utasítások használata
- While ciklusok használata
- Continue utasítás
- Break utasítás
- Return utasítások használata
- Waitfor utasítások használata
- Kurzozok használata
- Függvények használata
- Felhasználói függvények létrehozása

Változók

A következő típusokat használhatjuk:

| Típus | Leírás |
|------------|--|
| Bigint | Egész érték -2^{63} és $2^{63}-1$ közötti tartományban |
| Int | Egész érték -2^{31} és $2^{31}-1$ közötti tartományban |
| Smallint | Egész érték -2^{15} és $2^{15}-1$ közötti tartományban |
| Tinyint | 0 és 255 közötti egész érték |
| Bit | 1 vagy 0 értékű egész |
| Decimal | Rögzített pontosságú és méretű számérték $-10^{38}+1$ – től $10^{38}-1$ –ig |
| Numeric | Ugyanaz, mint a decimal |
| Money | Pénzérték a -2^{63} és $2^{63}-1$ közötti tartományban a pénzegység egy tízezrelékének pontosságával |
| Smallmoney | Pénzérték a $-214748,3648$ és $214748,3647$ közötti tartományban a pénzegység egy tízezrelékének pontosságával |
| Float | Lebegőpontos érték $-1,79E+308$ és $1,79E+308$ között |
| Real | Lebegőpontos érték $-3,4E+38$ és $3,4E+38$ között |

| | |
|------------------|---|
| Datetime | Dátum- és időérték 1753.január 1. és 9999.december 31. között, 3,33 ezredmásodperc pontossággal |
| Smalldatetime | Dátum- és időérték 1900.január 1. és 2079, június 6. között 1 perc pontossággal |
| Char | Rögzített hosszúságú nem Unicode karakterek, legfeljebb 8000 karakterig |
| Varchar | Változó hosszúságú nem Unicode karakterek, legfeljebb 8000 karakterig |
| Text | Változó hosszúságú nem Unicode karakterek, legfeljebb $2^{31}-1$ karakterig |
| Nchar | Rögzített hosszúságú Unicode karakterek, legfeljebb 4000 karakterig |
| Nvarchar | Változó hosszúságú Unicode karakterek, legfeljebb 8000 karakterig |
| Ntext | Változó hosszúságú Unicode karakterek, legfeljebb $2^{31}-1$ karakterig |
| Binary | Rögzített hosszúságú bináris adat, legfeljebb 8000 bájtig |
| Varbinary | Változó hosszúságú bináris adat, legfeljebb 8000 bájtig |
| Image | Változó hosszúságú bináris adat, legfeljebb $2^{31}-1$ bájtig |
| Cursor | Hivatkozás kurzorra (sormutatóra), vagyis sorok egy halmazára |
| Sql_variant | Bármilyen SQL SERVER adattípust tárolhat, kivéve text, ntext és timestamp típusúakat |
| Table | Sorok halmazát tárolja |
| Timestamp | Egyedi bináris szám, amely minden sormódosításnál frissül; egy táblában csak egy timestamp oszlop lehet |
| Uniqueidentifier | Globálisan egyedi azonosító (GUID, globally unique identifier) |

A változókat a DECLARE utasítással vezetjük be, amelyet a változó neve és típusa követ. A változó neve elé egy kukacjelet @ kell írunk. Egy sorban több változót is bevezethetünk.

```
declare @MyProductName nvarchar(40), @MyProductID int
```

A változók null kezdőértéket kapnak, értéküket a SET utasítással állíthatjuk be:

```
set @MyProductName = 'Szottyesz'
set @MyProductID = 5
```

Feltételes szerkezetek használata

A feltételes szerkezetek ugyanúgy működnek, mint már megszoktuk, csak a szintaktika más. Az IF utasítások bármilyen szintig egymásba ágyazhatóak. Egyszerre több utasítást is megadhatunk, csak ilyenkor BEGIN END közé kell írunk a kívánt utasításokat.

```

IF feltétel1
    BEGIN
        Utasítások1
    END
ELSE
    BEGIN
        Utasítások2
    END

```

CASE utasítások

A következő példában a SELECT utasítás eredményeként kapott értéket egy változóban tároljuk:

```

DECLARE @State nchar(2)
SET @State = 'Ma'
DECLARE @StateName nvarchar(15)
SELECT @StateName =
CASE @State
    WHEN 'CA' THEN 'California'
    WHEN 'MA' THEN 'Massachusetts'
    WHEN 'NY' THEN 'New York'
END

```

While ciklusok

Ha egy vagy több utasítás többszöri végrehajtására van szükségünk, akkor WHILE ciklusokat használhatunk. A WHILE ciklusok addig futnak, amíg a megadott feltétel igaz.

Az utasításforma a következő:

```

DECLARE @count int
SET @count = 5
WHILE (@count>0)
    BEGIN
        PRINT 'count = ' + CONVERT(nvarchar, @count)
        SET @count = @count -1
    END

```


A CONTINUE utasítás

A CONTINUE utasítással azonnal egy WHILE ciklus következő ismétlésére ugorhatunk, átugorva a ciklusból még esetleg hátralévő kódrészeket. Az utasítás hatására a végrehajtás visszaugrik a ciklus elejére.

A BREAK utasítás

Ha egy WHILE ciklusnak azonnal véget szeretnénk vetni, a BREAK utasítást használhatjuk. Az utasítás hatására a végrehajtás kikerül a ciklusból, és a program futása a ciklus utáni utasításokkal folytatódik.

RETURN utasítások használata

A RETURN utasítással egy tárolt eljárásból vagy utasítások egy csoportjából léphetünk ki, a RETURN -t követő egyetlen utasítás sem hajtódik végre. Az utasítással értéket is visszaadhatunk, de csak ha tárolt eljáráshoz használjuk.

WAITFOR utasítások használata

Előfordulhat, hogy azt szeretnénk, hogy a program megállna, mielőtt bizonyos műveleteket végrehajtanánk, például éjszaka frissítenénk a felhasználói rekordokat. A WAITFOR utasítással adhatjuk meg, hogy mennyi ideig várjon a program a többi utasítás végrehajtása előtt.

Az utasítás formája:

WAITFOR [DELAY 'időtartam' | TIME 'jelenlegi idő']

DELAY: várakozási időtartam,

TIME: pontos időpont.

Néhány példa:

```
WAITFOR DELAY '00:00:06' - 6 másodpercig vár
WAITFOR TIME '10:02:15' - 10 óra 2 perc 15 másodperckor folytatja a
végrehajtást
```

RAISERROR utasítások használata

A RAISERROR utasítással hibaüzenetet állíthatunk elő. Általában akkor van szükség erre, ha valamelyik tárolt eljárásban hiba következik be.

Az utasítás egyszerűsített formája a következő:

RAISERROR ({szám | leírás}{, súlyosság, állapot})

Itt a szám a hiba száma, amelynek 50001 és 2147483648 között kell lennie, a leírás egy 400 karakternél nem hosszabb üzenet, a súlyosság a hiba fokozata, ami 0 és 18 között lehet, az állapot pedig egy tetszőleges érték 1 és 127 között, ami a hiba hívási állapotát mutatja.

KURZOROK használata

Felmerülhet a kérdés, hogy mi is az a kurzor, s mire lehet használni. Nos amikor végrehajtunk egy SELECT utasítást, akkor egyszerre kapunk meg minden sort. Ez nem mindig megfelelő, előfordulhat például, hogy egy adott sor visszkapott oszlopértékei alapján valamilyen műveletet szeretnénk végezni. Ehhez egy kurzort (sormutatót) kell használnunk, amellyel a z adatbázisból kinyert sorokat egyenként dolgozhatjuk fel. A kurzor segítségével végiglépkedhetünk az adott SELECT utasítás által visszaadott sorokon.

Kurzor használatakor a következő lépéseket kell követnünk:

1. Változókat vezetünk be a SELECT utasítás által visszaadott oszlopértékek tárolására.
2. Bevezetjük a kurzort, megadva a megfelelő SELECT utasítást.
3. Megnyitjuk a kurzort.
4. Kiolvassuk a sorokat a kurzorból.
5. Bezárjuk a kurzort.

Egy példaprogramon keresztül bemutatjuk a kurzorok használatát, mely megmutatja, hogy hogyan jeleníthetjük meg a kurzor segítségével a Products tábla ProductID, ProductName és UnitPrice oszlopait:

```
Use Northwind

-- 1. lépés: a változók bevezetése
DECLARE @MyProductID int
DECLARE @MyProductName nvarchar(40)
DECLARE @MyUnitPrice money

-- 2. lépés: a kurzor bevezetése
DECLARE ProductCursor CURSOR FOR
SELECT ProductID, ProductName, UnitPrice
FROM Products
WHERE ProductID <= 10

-- 3. lépés: a kurzor megnyitása
OPEN ProductCursor

-- 4. lépés: a sorok kiolvasása a kurzorból
FETCH NEXT FROM ProductCursor
```

```

    INTO @MyProduct, @MyProductName, @MyUnitPrice
    PRINT '@MyProductID = ' + CONVERT(nvarchar, @MyProductID)
    PRINT '@MyProductName = ' + CONVERT(nvarchar, @MyProductName)
    PRINT '@MyUnitPrice = ' + CONVERT(nvarchar, @MyUnitPrice)
    WHILE @@FETCH_STATUS = 0
    BEGIN
        FETCH NEXT FROM ProductCursor
        INTO @MyProductID, @MyProductName, @MyUnitPrice
        PRINT '@MyProductID = ' + CONVERT(nvarchar, @MyProductID)
        PRINT '@MyProductName = ' + CONVERT(nvarchar, @MyProductName)
        PRINT '@MyUnitPrice = ' + CONVERT(nvarchar, @MyUnitPrice)
    END

    -- 5. lépés: a kurzor bezárása
    CLOSE ProductCursor
    DEALLOCATE ProductCursor

```

Mint a példaprogramban is látható, a változók típusa meg kell, hogy egyezzen a kinyert sorok oszlopainak típusával.

A kurzor bevezetése abból áll, hogy megadjuk a kurzorhoz rendelendő nevet, illetve a végrehajtani kívánt SELECT utasítást. A SELECT utasítás addig nem hajtódik végre, amíg a kurzort meg nem nyitjuk. A kurzort a DECLARE utasítás használatával vezetjük be.

Mint láthatjuk, a kurzor megnyitása az OPEN paranccsal történik.

Ahhoz hogy a sorokat ki tudjuk olvasni a kurzorból, a FETCH utasításra van szükségünk. A kurzorban számos sor lehet, így egy WHILE ciklust, s a @@FETCH_STATUS -t kell alkalmaznunk annak megállapítására, hogy a ciklusnak mikor kell véget érnie.

A @@FETCH STATUS függvény visszatérési értékei:

- 0: A FETCH utasítás sikeresen visszaadott egy sort,
- -1: A FETCH utasítás hibázott, vagy a kért sor az eredményhalmazon kívülre esett,
- -2: A lekért sor hiányzik.

A kurzort a CLOSE utasítással zárhatjuk be, s a DEALLOCATE utasítással a kurzorra való hivatkozást, mellyel felszabadíthatjuk az általa használt rendszererőforrásokat.

Függvények használata

Az SQL Server számos függvényt bocsát a rendelkezésünkre, amelyekkel értékeket nyerhetünk ki az adatbázisokból. Egy tábla sorainak számát például a COUNT () függvénnyel kaphatjuk meg.

Függvénykategóriák:

1. Összesítő függvények: egy tábla egy vagy több sora alapján adnak vissza információkat
2. Matematikai függvények: számítások végzésére használatosak
3. Karakterláncfüggvények: karakterláncokon hajtanak végre műveleteket
4. Dátum- és időfüggvények: dátum- és időkezelési műveleteket hajtanak végre
5. Rendszerfüggvények: az SQL Serverről szolgáltatnak információt
6. Beállítási függvények: a kiszolgáló beállításairól adnak információt
7. Kurzorfüggvények: a kurzorokról szolgáltatnak információt
8. Metaadatfüggvények: az adatbázisról, illetve annak elemeiről, például a táblákról adnak információt
9. Biztonsági függvények: az adatbázis felhasználóiról és szerepköreiről nyújtanak információt
10. Rendszerstatisztikai függvények: statisztikai adatokat adnak vissza az SQL Serverről
11. Szöveg- és képfüggvények: szöveg- és képkezelési műveleteket hajtanak végre

Felhasználói függvények létrehozása

Az SQL Serverben saját, úgynevezett felhasználói függvényeket is készíthetünk. Egy ilyen függvénnyel kiszámíthatunk például egy kedvezményes árat az eredeti ár és egy szorzó alapján. A felhasználói függvények létrehozására a CREATE FUNCTION utasítás szolgál, és három fajtájuk van:

Skalárfüggvények A skalárfüggvények egyetlen értéket adnak vissza, ami bármilyen típusú lehet, kivéve a text, ntext, image, cursor, table és timestamp típusokat, illetve a felhasználói adattípusokat.

Helyben kifejtett táblaértékű függvények A helyben kifejtett (inline) táblaértékű függvények table típusú objektumokat adnak vissza. A table objektumok olyanok, mint egy szabályos adatbázistábla, csak a memóriában tárolódnak. A helyben kifejtett táblaértékű függvények egyetlen SELECT utasítással kinyert adatokat adnak vissza.

Többutasításos táblaértékű függvények A többutasításos táblaértékű függvények is table típusú objektumokat adnak vissza, de a helyben kifejtett táblaértékű függvényektől eltérően több T-SQL utasítást tartalmazhatnak.

Példa egy skalárfüggvényre:

-- A DiscountPrice kiszámítja egy termék új árát az eredeti ár és a leértékelési szorzó alapján

```
CREATE FUNCTION DiscountPrice (@OriginalPrice money, @Discount float)
RETURNS money
AS
BEGIN
    RETURN @OriginalPrice * @Discount
```

END

Ha létrehoztuk a függvényt, meghívhatjuk. A skalárfüggvények meghívása a következő:

tulajdonos.függvéynév

Itt a tulajdonos a függvényt létrehozó adatbázis-felhasználó neve, a függvéynév pedig a függvény neve.

Helyben kifejtett táblaértékű függvények

A helyben kifejtett (inline) táblaértékű függvények table típusú objektumokat adnak vissza, amelyeket egyetlen SELECT utasítás tölt fel. Itt nincs BEGIN és END utasítások közé zárt utasításblokk, a függvény mindössze egy SELECT utasítást tartalmaz.

Példa egy helyben kifejtett táblaértékű függvényre:

*/*Visszaadja a Products tábla azon sorait, amelyek UnitsInStock oszlopában a paraméterként átadott újrendelési szintnél kisebb, vagy azzal egyenlő érték szerepel.*/*

```
CREATE FUNCTION ProductsToBeReordered(@ReorderLevel int)
RETURNS table
AS
(
    SELECT * FROM Products
    WHERE UnitsInStock <= @ReorderLevel
)
```

A skalárfüggvényektől eltérően meghívásukkor nem kell feltüntetnünk a tulajdonost.

```
SELECT * FROM ProductsToReordered(10)
```

Többutasításos táblaértékű függvények

Példaprogram, mely visszaadja a Products tábla azon sorait, amelyek UnitsInStock oszlopában a paraméterként átadott újrendelési szintnél kisebb, vagy azzal egyenlő érték szerepel, és beszúr egy új oszlopot Reorder néven:

```
CREATE FUNCTION ProductsToBeOrdered2(@ReorderLevel int)
RETURNS @MyProducts table
(
    ProductId int,
    ProductName nvarchar(40),
    UnitsInStock smallint,
    Reorder nvarchar(3)
```

```

)
AS
BEGIN
    -- sorok kinyerése a Products táblából és
    -- beszúrásuk a MyProducts táblába
    -- a Reorder oszlop 'NO' -ra állítása
    INSERT INTO @MyProducts
    SELECT ProductID, ProductName, UnitsInStock, 'NO'
    FROM Products;

    -- a MyProducts tábla frissítése a Reorder oszlop
    -- 'YES' -re állításával, ha a UnitsInStock
    -- kisebb, mint a @ReorderLevel, vagy azzal egyenlő
    UPDATE @MyProducts
    SET Reorder = 'YES'
    WHERE UnitsInStock <= @ReorderLevel

    RETURN
END

```

Ennek meghívásakor sem kell a tulajdonost feltüntetnünk:

```
SELECT * FROM ProductsToBeReordered2(20);
```

Az SQL Server lehetővé teszi, hogy az adatbázisokban eljárásokat tároljunk. A tárolt eljárások abban különböznek a felhasználói függvényektől, hogy jóval többféle adattípust adhatunk vissza.

Általában akkor készítünk tárolt eljárást, ha olyan feladatot kell elvégeznünk, ami erőteljesen igénybe veszi az adatbázist, vagy ha központosítani szeretnénk a kódokat, hogy az egyes felhasználóknak ne kelljen ugyanarra a feladatra saját programokat írniuk. Az intenzív adatbázis-használatára jó példa lehet egy banki alkalmazás, amelynek segítségével a számlákat frissítjük a nap végén, központosított kódra pedig akkor lehet szükség, ha a felhasználók hozzáférését az adatbázistáblákhoz korlátozni akarjuk.

Tárolt eljárások létrehozása:

Szükségünk lesz egy adatbázisra, például ProcedureTest névvel, és egy tábla néhány adattal.

```

CREATE TABLE Table01
(
    value1 int,
    value2 varchar(10)
)

```

Tárolt eljárás létrehozásához a create procedure utasítást kell használnunk. Ezt követően adhatjuk meg az eljárás nevét, majd az AS után jöhet a T-SQL kód, hogy

mit is végezzen el a létrehozott eljárás. Létrehozhatunk úgynevezett lokális és globális ideiglenes eljárásokat is. A lokális csak a saját kapcsolatban használható, azt más kívülről más nem tudja majd használni, ezzel szemben a globális eljárást más kapcsolatból is használhatják. Fontos, hogy a lokális változatok automatikusan törölődnek a kapcsolat lezárásával, míg a globálisak csak akkor, ha már minden kapcsolat lezárásra került.

Lokális ideiglenes eljárás létrehozásához az eljárás neve elé tegyünk egy # karaktert, a globálisnál pedig ## karaktert.

Természetesen paramétereket is adhatunk a tárolt eljárásnak. Ezt az eljárás neve után tehetjük meg egy vesszővel elválasztott felsorolásban. A paraméternév mindig egy @ jellel kezdődik. A név után a paraméter típusát adhatjuk meg. Ha olyan paramétert szeretnénk megadni, melyen keresztül értéket is adnánk vissza, akkor a típus után az OUTPUT jelzőt kell írunk.

Na de nézzünk egy példát:

```
create procedure Procedure01
    @a int,
    @b int
as
select convert(varchar(20), @a + @b)
```

Tárolt eljárások végrehajtása:

A tárolt eljárások végrehajtására az EXECUTE utasítást használhatjuk.

```
execute Procedure01 150, 50
```

Több eljárásnak adhatunk azonos nevet is, ezek között úgy tudunk különbséget tenni, hogy a név után pontosvesszővel megadunk egy sorszámot. Ennek ott lesz előnye, hogy ha törölni akarjuk az eljárásokat a DROP PROCEDURE utasítással, akkor elegendő megadni az eljárás nevét és annak összes változata törlésre kerül.

```
create procedure Procedure01;2
as
select min(value1) as 'Minimum value1' , max(value1) as 'Maximum
value1' from Table01
```

Futtassuk az imént létrehozott eljárást:

```
execute Procedure01;2
```

Kioldók

A kioldók (trigger) olyan különleges tárolt eljárás, amelyet az adatbázis-kezelő automatikusan futtat, amikor egy meghatározott INSERT, UPDATE vagy DELETE

utasítást egy bizonyos adatbázistáblán végrehajtunk. A kioldók igen hasznosak például akkor, ha egy tábla oszlopértékeinek változásait szeretnénk ellenőrizni. A kioldó egy INSERT, UPDATE vagy DELETE utasítás helyett is elindulhat.

A kioldókat a CREATE TRIGGER utasítással hozhatjuk létre.

A Transact-SQL –ről láthattunk egy rövid ismertetőt. A T-SQL segítségével olyan SQL utasításokat tartalmazó programokat írhatunk, amelyekben a szokványos programozási szerkezetek is megtalálhatók.

Az SQL Server számos függvényt bocsát a rendelkezésünkre, amelyekkel értékeket nyerhetünk ki az adatbázisokból.

Az SQL Serverben saját, úgynevezett felhasználói függvényeket is készíthetünk.

Az SQL Server azt is lehetővé teszi, hogy az adatbázisokban eljárásokat tároljunk. A tárolt eljárások abban különböznek a felhasználói függvényektől, hogy jóval többféle adattípust adhatnak vissza. Általában akkor használjuk, ha olyan feladatot kell elvégeznünk, ami erőteljesen igénybe veszi az adatbázist, vagy ha központosítani szeretnénk a kódokat, hogy az egyes felhasználóknak ne kelljen ugyanarra a feladatra saját programokat írniuk.