DEBRECENI EGYETEM
GAZDASÁGTUDOMÁNYI KAR
Alkalmazott Informatika és Logisztika Intézet

Alkalmazott informatikai füzetek 4.

VÁRALLYAI LÁSZLÓ

Visual C# a gyakorlatban

press

DEBRECENI EGYETEM
GAZDASÁGTUDOMÁNYI KAR
Alkalmazott Informatika és Logisztika Intézet

Alkalmazott informatikai füzetek 4.

VÁRALLYAI LÁSZLÓ

Visual C# a gyakorlatban

Debreceni Egyetemi Kiadó
Debrecen University Press
2017

Lektorok:

Dr. Csák Máté
egyetemi adjunktus

Takács Viktor László
tanársegéd

ISBN 978-963-318-651-0

Kiadta: a Debreceni Egyetemi Kiadó, az 1795-ben alapított
Magyar Könyvkiadók és Könyvterjesztők Egyesülésének a tagja
www.dupress.hu
Felelős kiadó: Karácsony Gyöngyi

1 A Visual Studio 2012 fejlesztőkörnyezet

1.1 A Visual Studio 2012 fejlesztőkörnyezet használata

Az IDE (Integrated Development Environment – integrált fejlesztőkörnyezet) egyfajta szoftver, mely segíti a programozót a programírásban és tesztelésben. A következő részekből áll:

- Forráskód szerkesztő
- Grafikus felhasználói felület tervező
- Fordító vagy értelmező program
- Debugger (Hibakereső)

1.1.1 Új projekt létrehozása

A fejlesztőkörnyezet használatát egy klasszikus példán keresztül mutatjuk be: készítünk egy programot, mely gombnyomásra megjeleníti a „Hello” feliratot egy címkén.

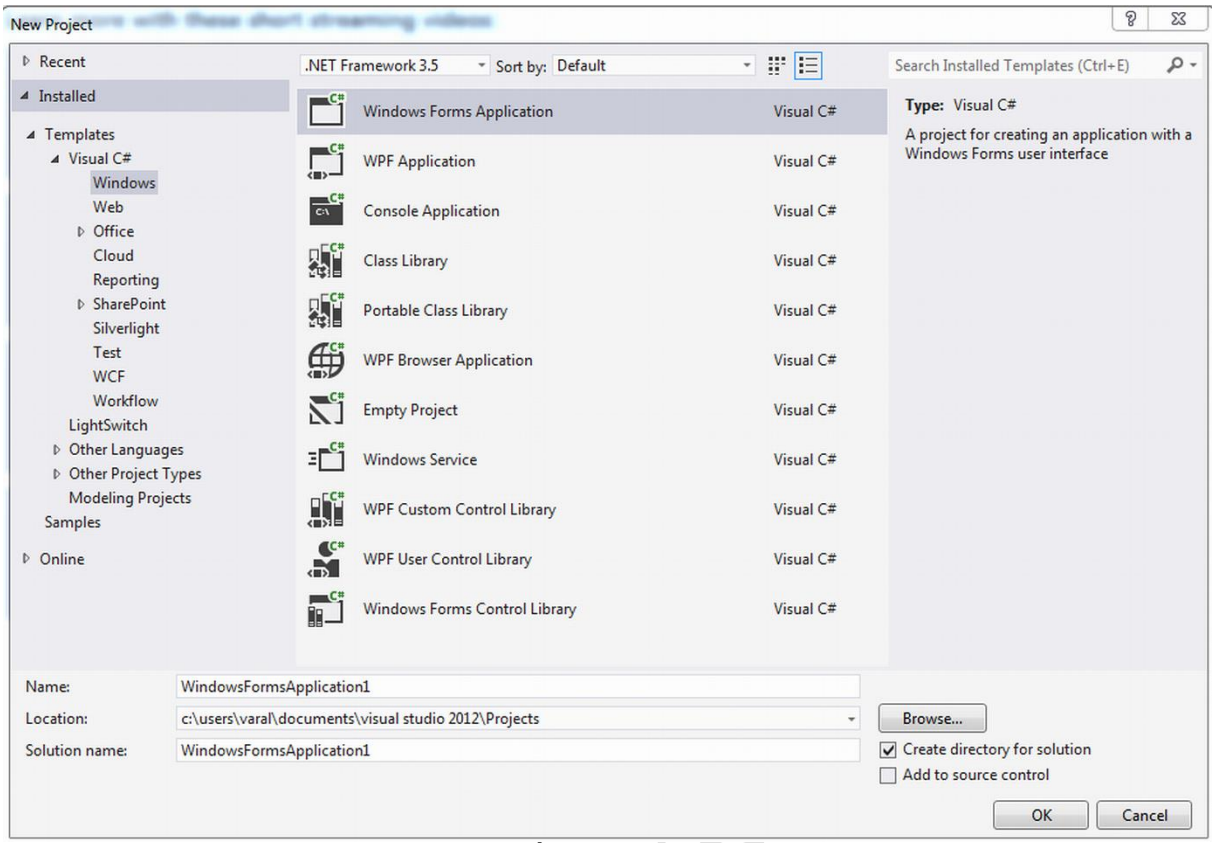
1. Indítsuk el a fejlesztőkörnyezetet!
2. A „File” menüben válasszuk a „New Project” lehetőséget (1. ábra).
3. A Visual Studio által támogatott programozási nyelvek közül válasszuk a Windows C#-ot, majd azon belül a Windowst. A felkínált sablonok közül a „Windows Forms Application”-t! Nevezzük el a projektet (pl. Hello néven) és állítsuk be a mentés helyét (ez alapértelmezetten a dokumentumok könyvtárban található „Visual Studio” könyvtár, amelyhez a verziószám kerül hozzáfűzésre, ezen belül a Projects mappában hozza létre a „Hello” nevű könyvtárat a rendszer és ide ment minden a projekthez tartozó állományt). A „Windows Application” sablon választásával olyan projektet hozunk létre, mely már tartalmaz egy üres ablakot.

1.1.2 Az űrlapszerkesztő és a Toolbox

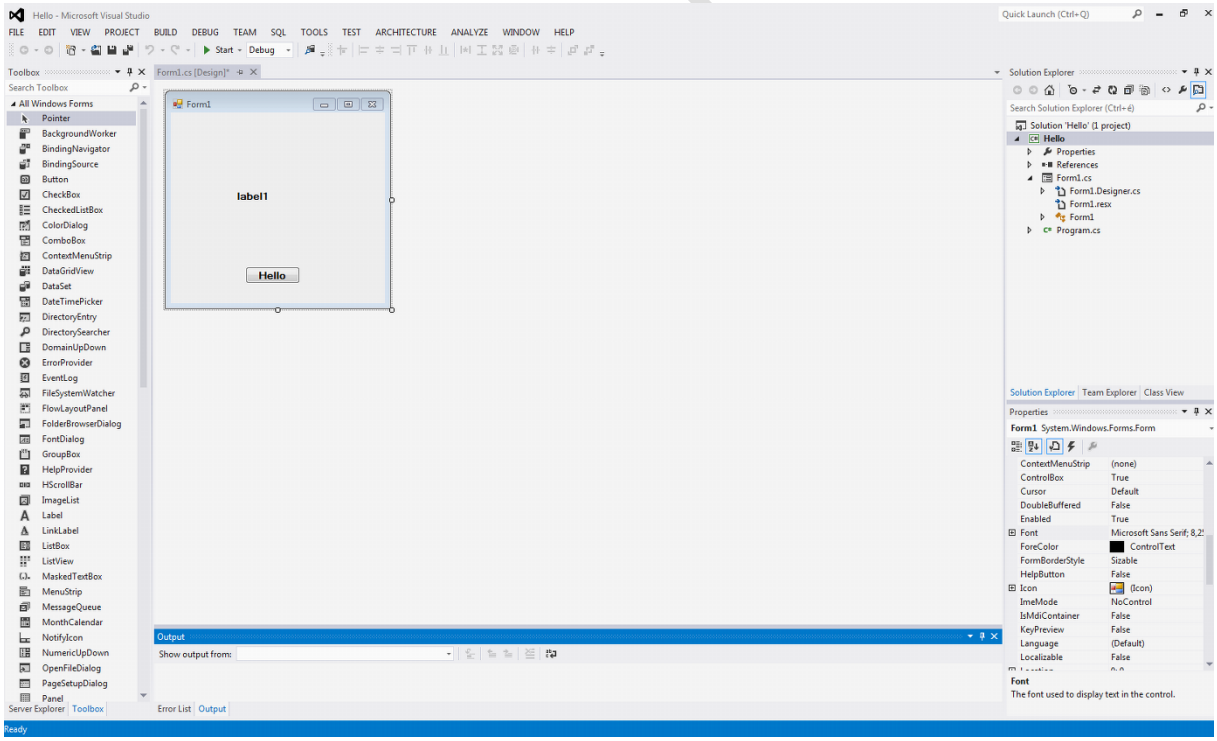
A Toolbox-ot az ablak bal oldalán található fülre kattintva nyithatjuk fel. Ha szeretnénk az ablakba dokkolni, kattintsunk az ablakfejléc gombostű ikonjára (📌)!

Az Toolbox-on látható komponenseket helyezhetjük el az űrlapon. Itt találjuk a gombot, a címkét, a szövegdobozt, a menüt, stb.

Helyezzünk el egy gombot (Button) és egy cimkét (Label) az űrlapon (Form).



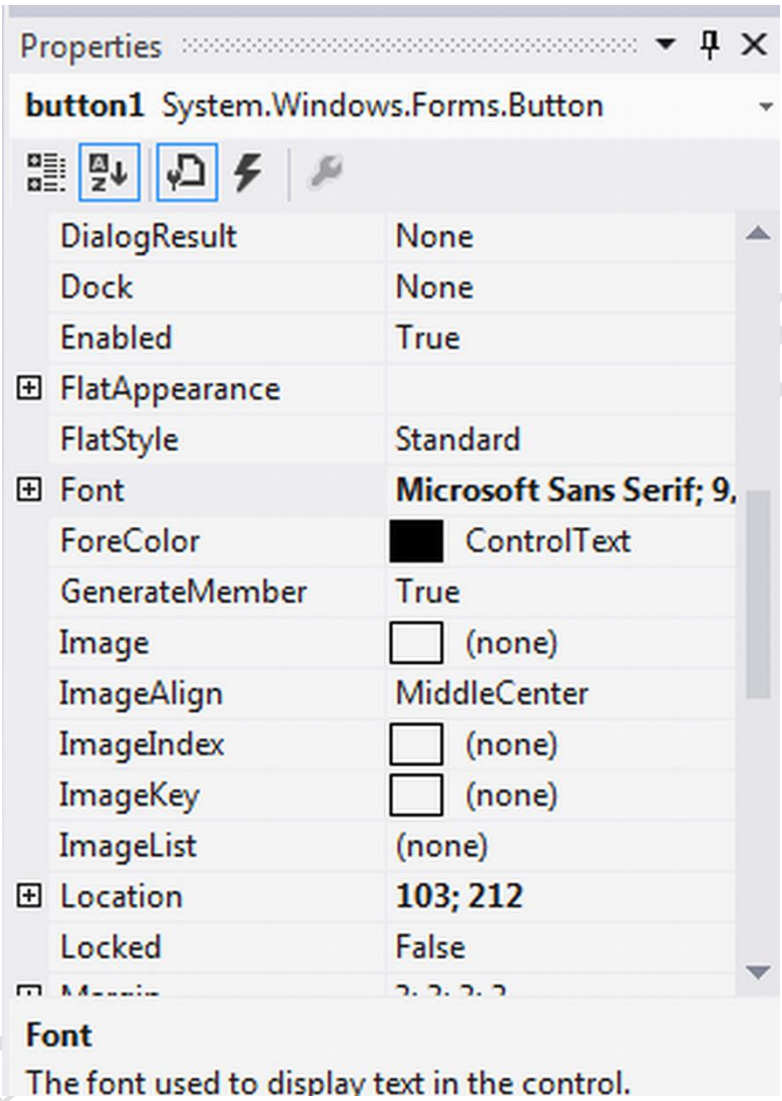
1. ábra: Új projekt létrehozása



2. ábra: Hello alkalmazás

1.1.3 A Properties ablak

A „Properties” ablakot a View / Properties Window menüponttal jeleníthetjük meg.



3. ábra: Properties (tulajdonság) ablak

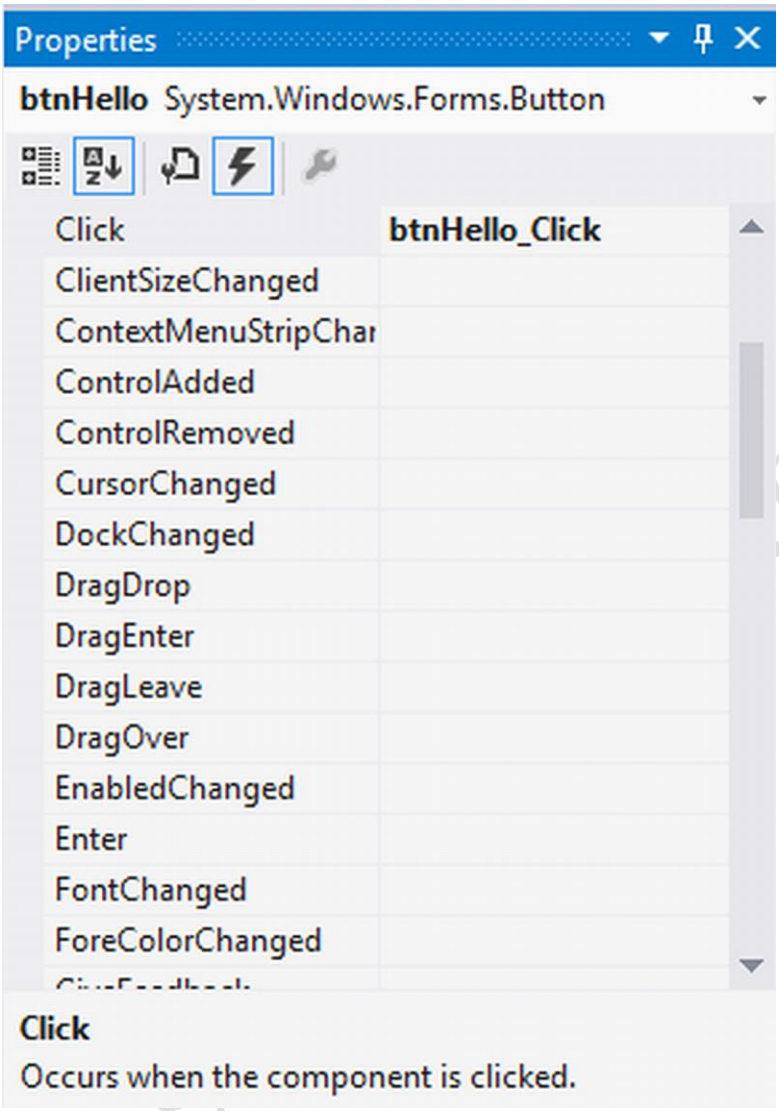
Tulajdonságok (properties) beállítása

Az űrlapokon használt komponensek tulajságait kétféleképpen állíthatjuk be:

- Design time: az űrlap tervezésekkor, a szerkesztőfelületen
- Runtime: a program futása közben, programkódból

A properties ablakban tervező nézetben állíthatjuk be a kiválasztott komponens (jelen esetben nyomógomb) tulajdonságait (3. ábra). A komponensre jellemző tulajdonságokat megtekinthetjük név szerinti sorrendben, vagy funkciójuk szerint csoportosítva.

Események (events)



4. ábra: Events (esemény) ablak

Grafikus felületek programozásánál az eseményvezéreltség elvét kell követnünk. Ez azt jelenti, hogy mindig egy adott eseményhez rendeljük a programkódot. Példánkban meghatározzuk, mi történjen akkor, ha a felhasználó a „Hello” gombra kattintott (4. ábra).

A kiválasztott komponenshez kapcsolódó eseményeket az „Events’ gombra kattintva tekinthetjük meg. Amennyiben az esemény neve mellé duplán kattintunk, a kódszerkesztőbe jutunk, ahol megírhatjuk az eseményt kiszolgáló függvényt. (btnHello_Click)

A leggyakrabban használt esemény az egérkattintás, a „Click”.

1.1.4 Kódszerkesztő

A modern IDE-k forráskód-szerkesztői több funkcióban eltérnek az egyszerű szövegszerkesztőktől:

- Syntax higlighting (szintaxis kiemelés): a szerkesztő a nyelv foglalt szavait más színnel jelöli, így azok elkülönülnek az általunk adott elnevezésektől.
- Auto Code Completion (automatikus kódkiegészítés): a forráskód szerkesztése közben a Ctrl + Space billentyűk lenyomására lista jelenik meg a képernyőn, mely tartalmazza a szintaktikailag odaillő nyelvi elemeket. Így elég egy nyelvi elem első néhány karakterét begépelni, a befejezést választhatjuk a listából. Ezzel a módszerrel nem csak a gépelési idő rövidül le, hanem csökken az elütésekből valamint a kis és nagybetűk helytelen használatából adódó hibák száma. Ahhoz, hogy a felkínált lista mindig aktuális legyen a szerkesztőnek folyamatosan elemeznie kell a teljes forráskódot, ami nagyobb projekteknél jelentős gépkapacitást igényel.
- Code collapse (Kód „összeomlasztás"): Bizonyos nyelvi struktúrák esetén, a sor elején kis négyzetben + vagy - jel jelenik meg. Ezzel elrejthető a kódnak az a része, amin épp nem dolgozunk. Az összetartozó kódrészleteket vonal köti össze, mely segít az eligazodásban.
- Auto Code Formating (Automatikus kódformázás): Az egymásba ágyazott programstruktúrákat automatikusan egymás alá igazítja. (Edit / Advanced / Format document)

1.1.5 Töréspontok és nyomkövetés

Amennyiben összetettebb algoritmust alkalmazunk előfordulhat, hogy a program szintaktikailag teljesen helyes, azonban futtatáskor mégsem a kívánt eredményt adja. Ilyen hibák felderítésére használjuk a nyomkövető funkciót. Ha a programsorok előtti szürke sávra kattintunk, akkor töréspontokat helyezhetünk el a forráskódban. Ezeket piros tömör körrel jelöli a szerkesztőprogram. Amennyiben a program futás közben törésponthoz ér, a futás megszakad, és a rendszer „debug" módba kerül. Innen programunkat soronként hajthatjuk végre az F11 funkcióbillentyűvel tudunk a következő sorra lépni, közben pedig nyomon követhetjük változóink (ha vannak) értékének alakulását.

Változók értékének követésére a legegyszerűbb módszer, ha az egeret a forráskódban egy változó fölé helyezzük, a képernyőn megjelenik a változó értéke.

Néhány fontosabb Debug funkció és funkciógomb.

- F5 Folytatja a program futtatását a következő töréspontig.
- F11 „Step into" a következő programsorra ugrik. Ha függvényhívás történik, a függvénybe is beleugrunk, és azt is soronként hajtjuk végre.

- F10 „Step over” a következő programsorra ugrik. Függvényhívásoknál nem ugrik bele a függvényekbe.
- „Step Out” befejezi a függvény futtatását, és a hívási pontnál áll meg.
  Természetesen a fenti funkciókat a Debug menün keresztül is elérhetjük.

1.2 Alapvető komponensek használata és speciális vezérlők

A Visual Studio fejlesztőkörnyezet tartalmaz egy komponenskönyvtárat, amelyet a Toolbox-on keresztül érhetünk el, ahogy erről már korábban volt szó. Ezeket a komponenseket szabadon felhasználhatjuk saját programjainkban. Természetesen ezt a komponenskönyvtárat bővíthetjük más szoftvercégektől vásárolt komponensekkel is vagy mi magunk is származtathatunk illetve írhatunk saját komponenseket, amiket a későbbiekben szabadon felhasználhatunk.

Ez a jegyzet nem referenciakönyv, így nem tartalmazhatja az összes komponens leírását, így ebben a fejezetben egy nagyon rövid összefoglalót adunk a leggyakrabban előforduló komponensekről.

Azokat a komponenseket, melyeken keresztül a felhasználó kezelni tudja az alkalmazást, vezérlőknek nevezzük. Ilyenek a gomb, a menü, a szövegdoboz, kép, stb. Léteznek olyan komponensek is, mint például az időzítő (Timer), melyek futtatás közben nem jelennek meg a képernyőn. Ezek nem tartoznak a vezérlők közé.

A következő részben tehát bemutatunk néhány gyakran használt komponenst azok fontosabb tulajdonságaival és legfontosabb eseményével illetve eseményeivel.

1.2.1 A gomb (Button)

Tulajdonságok (Properties)

Text	A gomb felirata.
Image	A gombon levő kép.
ImageAlign	A kép elhelyezkedése a gombon belül.

Események (Events)

Click	A gomb megnyomásakor következik be.

1.2.2 Szövegdoboz (TextBox)

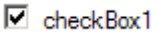
Tulajdonságok (Properties)

Text	A szövegdoboz szövege ezen a tulajdonságon keresztül állítható be és olvasható ki.
Multiline	Ha értéke igaz, a szövegdobozba több sornyi szöveget is írhatunk.
UseSystemPasswordChar	Ha értéke igaz (true), a begépelt szöveg helyén egy meghatározott karakter jelenik meg, így a jelszavakat nem lehet leolvasni a képernyőről.

Események (Events)

TextChanged	Akkor következik be, ha a szövegdoboz szövege megváltozik.

1.2.3 Jelölőnégyzet (CheckBox)



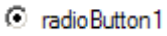
Tulajdonságok (Properties)

Checked	Jelzi, hogy a doboz bejelölt állapotban van-e.
CheckedState	A jelölőnégyzet három állapotban lehet: – bejelölt (checked) ☑ – jelöletlen (unchecked) ☐ – köztes (intermediate) ☑

Események (Events)

CheckStateChanged	Jelzi, ha a jelölőnégyzet állapota megváltozott

Rádiógomb (RadioButton)



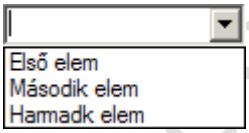
Tulajdonságok (Properties)

Checked	Jelzi, hogy a gomb benyomott állapotban van-e.

Események (Events)

CheckedChanged	Jelzi, hogy a gomb állapota megváltozott.

1.2.4 Legördülő doboz (ComboBox)



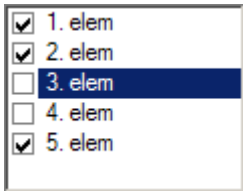
Tulajdonságok (Properties)

Items	Az items gyűjtemény tartalmazza a legördülő lista elemeit.
Sorted	Jelöli, hogy a lista elemei abc sorrendben jelenjenek-e meg.

Események (Events)

TextUpdate	Akkor következik be, ha a legördülő lista szövege megváltozik.
SelectedIndexChanged	Akkor következik be, ha a kiválasztott elem sorszáma megváltozik.

1.2.5 Jelölőnégyzet lista



Ez a komponens jelölőnégyzetekből álló listát jelenít meg. Tulajdonságai hasonlítanak a legördülő dobozéhoz (ComboBox). A fő különbség az, hogy itt egyszerre több elemet is ki tudunk választani.

Tulajdonságok (Properties)

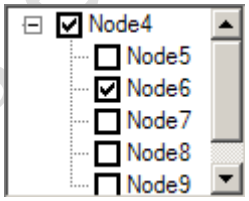
Items	Az items gyűjtemény tartalmazza a leördülő lista elemeit.
CheckedItems	Ez a gyűjtemény tartalmazza azokat az elemeket, melyeket a felhasználó kiválasztott.
Sorted	Jelöli, hogy a lista elemei abc sorrendben jelenjenek-e meg.

Események (Events)

SelectedIndexChanged	Akkor következik be, ha a kiválasztott elem sorszáma megváltozik.

1.2.6 Listadoboz (ListBox)

Egy fát jelenít meg, melyben a felhasználó több elemet is bejelölhet.



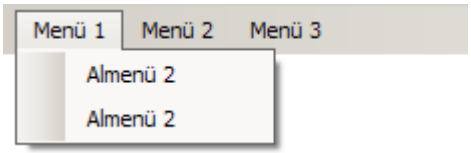
Tulajdonságok (Properties)

Nodes (collection)	A nodes gyűjtemény tartalmazza a fa elemeit.
CheckBoxes	Ha érékét igaz-ra állítjuk, a fa minden eleme előtt jelölőnégyzet jelenik meg.
FullPath	Tervezőnézetben nem elérhető tulajdonság. A felhasználó által kiválasztott elemek útvonalát adja vissza a fában.

Események (Events)

AfterCheck	Akkor következik be, ha egy jelölőnégyzet állapotát megváltoztatja a felhasználó

1.2.7 Menü építés a ToolStrip komponenssel



A menü felépítése a ToolSrtip komponens űrlapra helyezésével kezdődik. A ToolStrip DropDownItems tulajdonsága mellé kattintva egy párbeszédablak jelenik meg, melyben létrehozhatjuk a menüpontokat. Itt van lehetőségünk a menüpontok sorrendjének és tulajdonságaik beállítására is. Minden menüpont rendelkezik DropDownItems tulajdonsággal. Ez a tulajdonság egy gyűjtemény, mely tartalmazza a menüpont almenüpontjait.

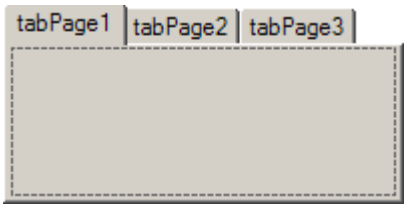
Minden menüponthoz tartozik Click esemény, mely az adott menüpont kiválasztásakor következik be.

1.2.8 Tárolók (Containers)

A tárolók olyan vezérlők, melyek más vezérlők elrendezésére szolgálnak. A tárolókba további vezérlőket helyezhetünk.

A tárolókon keresztül adatbevitel tulajdonképpen nem történik, eseményeiket is csak nagyon ritkán vagy egyáltalán nem kezeljük.

1.2.8.1 Lapozható kartoték (TabPage)



A kartotékhoz új füleket legegyszerűbben a jobb egér / Add tabs menüponttal adhatunk hozzá. A lapok sorrendjének, illetve az egyes lapok tulajdonságainak beállításához kattintsunk a TabPages tulajdonság mellé. A TabPages tulajdonság gyűjtemény, mely a kartoték lapjait tartalmazza. Itt tudjuk külön állítani az egyes lapok tulajdonságait is.

1.2.8.2 Panel

A panel vezérlőre is helyezhetünk további vezérlőket. Ha ezek túlnyúlnak a panel területén görgetősávok jelennek meg a panel szélein. Nagyméretű képek megjelenítéséhez is használhatunk panelt: Helyezzük a képet (PictureBox) a panelbe. Ha a kép nem fér be a panel területére, a megjelenő görgetősávok segítségével mozgathatjuk.

1.2.8.3 Osztott tároló (SplitContainer)

Két panelből és a közük levő elválasztó vonalból áll. Ha a BorderStyle tulajdonságát „Single”-re vagy „3D”-re állítjuk, a felhasználó egérrel állíthatja a két oldal arányát.

1.2.9 Fájlkezelő párbeszédablakok (FileOpenDialog, FileSaveDialog)

A Windows-os alkalmazásokból jól ismert fájlmegnyitás és mentés párbeszédablakok megjelenítésére szolgáló komponensek.

1.2.10 Időzítő (Timer)

Meghatározott időközönként eseményt generál.

Tulajdonságok (Properties)

Enabled	Ezzel a tulajdonsággal tudjuk engedélyezni illetve letiltani az időzítőt.
Interval	A két esemény között eltelt idő ezredmásodpercben megadva.

Események (Events)

Tick	Az esemény akkor következik be, ha a számláló lejárt.

2 Nyelvi elemek (kulcsszavak, változók, literálok, konstansok, névterek)

2.1 Kulcsszavak

Valamennyi programozási nyelvben a kulcsszavak különleges kifejezéseknek számítanak, amelyeknek egyedi jelentése van (ezért nem használhatjuk őket saját céljainkra – fenntartott szavak), hiszen ezek alkotják a definíciók és műveletek alapszavait.

2.2 Azonosítók

A programunkban használt változókat, konstansokat(literálokat), metódusokat, osztályokat bizonyos szabályok szerint el kell nevezni. Az ezeknek adott nevek az azonosítók, melyekre vonatkozóan léteznek szabályok valamennyi programozási nyelvben, így a Visual C#-ban is.

- Az azonosítók csak betűvel vagy aláhúzás jellel kezdődhetnek és nem tartalmazhatnak szóközt, speciális karaktereket (pl. #) valamint kulcsszavakat.
- Az első karakter után további betűk, számok vagy aláhúzás jelek következhetnek.
- A programban használt kulcsszavakat nem használjuk azonosítóként.

2.3 Utasítások

A kulcsszavakból, változókból, literálokból és azonosítókból kifejezéseket, utasításokat állítunk össze, amelyek összességéből áll össze a forrásprogram. A Visual C# nyelvben az utasításokat pontosvesszővel zárjuk.

A programvezérlés mikéntjét az utasítások sorrendje határozza meg. az operációs rendszer a publikus osztály konstruktorának adja át a vezérlést, majd a tovább osztályban található eljárások és függvények kerülnek meghívásra az utasítás kiadásának sorrendjében, majd végül visszatér a vezérlés az operációs rendszerhez. Az utasítások fajtái:

- összetett
- kifejezés
- elágazás
- ciklus
- ugró (jól strukturált programban ezt nem kell használni).

Ahol utasítás elhelyezhető, ott természetesen szerepelhet összetett utasítás is. Az összetett utasítás vagy blokk a {} zárójelek között felsorolt utasítások listája. Blokkon belül változók is deklarálhatók, amelyek a blokkban lokálisak lesznek. Blokkok tetszőlegesen egymásba ágyazhatók.

2.4 Változók, változótípusok

Bármilyen programot írjunk is minden esetben szükség van arra, hogy a felhasználótól a program futása során ún. input értéke(ke)t kérjünk be, illetve a számításaink során keletkező eredményeket (output) meg tudjuk jeleníteni a képernyőn. Az alapvető input-output műveletek tárgyalásához mindenképpen meg kell ismerkednünk a C# nyelvben használatos változó típusokkal Az olyan tárhelyet, amelynek tartalma változhat a program futása közben, változónak nevezzük.

Minden változónak rendelkeznie kell névvel vagy más néven azonosítóval, ezen kívül típussal, és tartalommal, vagyis aktuális értékkel. Az előzőekben említetteken kívül rendelkeznie kell élettartammal és hatáskörrel. Az élettartam azt jelenti, hogy az adott változó a program futása során mikor és meddig, a hatókör pedig azt adja meg, hogy a program mely részeiben használható.

A változókat a programban be kell vezetni, azaz közölni kell a fordítóprogrammal, hogy milyen névvel és milyen típusú változót kívánunk használni a programban. Ezt nevezzük deklarációnak.

```
típus változónév;
```

A változóinkat akár már a deklaráció során elláthatjuk kezdőértékkel is. A kezdőérték adás azért is fontos, mert az érték nélkül használt változók kifejezésekben való szerepeltetése esetén a fordító hibát fog jelezni.

```
típus változónév=kezdőérték;
```

A típus meghatározza a változó lehetséges értékeit, értéktartományait, valamint hogy, milyen műveleteket lehet végezni velük, és milyen más típusokkal kompatibilis. A névvel a változóra tudunk hivatkozni a program bármely részéből.

2.5 Literálok, konstansok

A változók egy érdekes típusa a literál, akkor használjuk őket, amikor programunkban egy konkrét értéket szeretnénk használni. A literálok tehát egyszerű, rögzített értékek, tulajdonképpen konstansok. Jelentésük mindenesetben az aktuális tartalmukkal egyezik meg.

A C# programokban állandókat, vagy más néven konstansokat is definiálhatunk. A konstansok a program futása alatt nem változtatják értéküket, s nem lehet felüldefiniálni őket, vagy értékadó utasítással megváltoztatni értéküket (ilyenkor ugyanis a fordító hibát jelez). Több más nyelvtől eltérően, a C# nyelvben a konstansnak is van típusa.

```
const byte szazalek=10;
const string=”Ez egy konstans szöveg”;
```

2.6 Műveletek (operandusok, operátorok)

A változókkal kapcsolatosan többféle típusú műveletet tudunk végezni, amelyeket az alábbi kategóriákba sorolhatjuk:

- értékadó műveletek – **a**=kifejezés;
- matematikai műveletek – +=, -=, *=, /=, %=
- relációs műveletek – >, >=, <, <=, ==, !=
- feltételes műveletek – elágazásokban használjuk
- egyéb műveletek – a fentiekbe nem sorolható műveletek

Ha a műveleti jelek oldaláról nézzük, akkor az alábbi kategóriákat állíthatjuk fel:

- Egyoperandusú műveletek (2.5.1 fejezet)
- Kétoperandusú műveletek (2.5.2 fejezet)
- Háromoperandusú műveletek (2.5.3 fejezet)

2.6.1 Egyoperandusú művelet

A fentebbi fejezetekben megismert műveletek általában kétoperandusúak voltak. Létezik azonban két olyan művelet, konkrétan a növelés és a csökkentés, amelyek eggyel növelik illetve csökkentik a változó értékét.

`++x;` ugyanaz, mint `x=x+1;` Prefixes alak

`--x;` ugyanaz, mint `x=x-1;` Prefixes alak

Az egyoperandusú műveleteknél a műveleti jelek a változó mindkét oldalán elhelyezhetők:

`x++;` ugyanaz, mint `x=x+1;` Postfixes alak

`x--;` ugyanaz, mint `x=x-1;` Postfixes alak

A növelés és csökkentés viszont nem ugyanúgy zajlik le az egyik illetve a másik esetben. A prefixes alak esetében a növelés vagy csökkentés előzetes műveletként hajtódik végre és ezt követően történnek a további műveletek (pl. értékadás). A postfixes alak esetében

a növelés vagy csökkentés utólagos műveletként hajtódik végre és ezt megelőzően történnek a további műveletek (pl. értékadás). Az eredmény szempontjából tehát abszolút nem mindegy melyik alakot használjuk.

2.6.2 Kétoperandusú műveletek

A matematikai műveletek során megismert összeadás, kivonás, szorzás, osztás maradékképzés tarozik ebbe a kategóriába, de mivel ott már ezek ismertetésre kerültek, így itt nem térünk ki külön rá.

2.6.3 Háromoperandusú művelet

A C# nyelvben egyetlen olyan művelet létezik, amely háromoperandusú, ez pedig egy feltételes művelet. Szintaxisa:

```
(feltétel) ? utasítás1 : utasítás2
```

A kifejezés három részre oszlik, mely részeket a ”?” és ”:” operátor választ el egymástól. Amennyiben a kifejezés elején álló feltétel igaz, úgy az utasítás1 kerül végrehajtásra, ha viszont a feltétel hamis, akkor az utasítás2. Ez a háromoperandusú művelet, tulajdonképpen az ”**if**” utasítás tömörebb formája.

2.7 A kifejezések

A kifejezéseket konstans értékekből, változókban tárolt adatokból, függvényekből és műveletekből állíthatjuk össze. A kifejezéseket a számítógép kiértékeli, kiszámítja és ennek eredményét felhasználhatjuk a későbbiekben. A kifejezések eredményének típusa a bennük szereplő adatok és műveletek típusától függ, kiszámításuk módját pedig a precedencia szabályok írják elő. A precedencia szabályok határozzák meg a kifejezésekben a műveleti sorrendet, vagyis azt, hogy az egyes műveleteket a számítógép milyen sorrendben végezze el. Elsőként mindig a magasabb precedenciájú művelet kerül végrehajtásra, majd utána az alacsonyabb precedenciájúak. Azonos precedenciájú műveletek esetén általában a balról jobbra szabály érvényesül.

2.8 Névterek

A névterekkel az osztályok valamilyen logika szerinti csoportosítását lehet megoldani. Ez azért is fontos, mert a .NET osztálykönyvtárai becslések szerint tízezer nevet tartalmaznak. Ilyenkor szinte elkerülhetetlen, hogy a nevek ne ismétlődjenek. A fordító sem tudná eldönteni melyik névre gondoltunk. A névterek egy adott osztály-csoport számára egy, a külvilág felé zárt világot biztosítanak, amelyben a programozó döntheti el, hogy mely osztályok láthatóak és használhatóak kívülről, és melyek nem. A névterek segítségével nagyobb újrafelhasználható kódrészeket készíthetünk, valamint könnyebben modularizálhatjuk programjainkat.

3 Feltételes utasítások – szelekció

A strukturált utasítások más egyszerű és strukturált utasításokból épülnek fel. Az utasítások által definiált tevékenységek sorban (összetett), feltételtől függően (if és switch) vagy ciklikusan ismétlődve (for, while, do-while) hajtódnak végre.

3.1 Relációs operátorok

A programjaink írása közben gyakran szükségünk lehet arra, hogy értékeket összehasonlítsunk, amelyhez szükségünk lesz a relációs operátorokra. A relációs műveletek eredménye minden esetben egy logikai érték, ami vagy true(igaz) vagy false(hamis). A relációs műveleteket a leginkább a feltételes utasításoknál és a feltételes ciklusoknál használjuk.

3.2 Logikai műveletek

Logikai műveletek alatt a szokásos NOT, AND, OR és XOR műveleteket értjük. A NOT művelet operátora a ”!” jel, a XOR műveleté a ”^” jel. A többi művelet esetében azonban megkülönböztetünk feltételes és feltétel nélküli AND vagy OR műveletet.

1. táblázat *Feltételes* műveletek

Művelet	**Műveleti jel**	**Használat**	**Magyarázat**
Feltételes AND	&&	A && B	Ha A hamis, B már nem kerül kiértékelésre
Feltételes OR	||	A || B	Ha A igaz, B már nem kerül kiértékelésre
Feltételnélküli AND	&	A & B	B mindig kiértékelésre kerül
Feltételnélküli OR	|	A | B	B mindig kiértékelésre kerül

A feltételes műveletek és kiértékelésük:

- AND (csak akkor igaz, ha mindegyik részfeltétel igaz)
- OR (csak akkor igaz, ha legalább egy részfeltétel igaz)

- NOT (a megadott logikai kifejezés értékét az ellenkezőjére változtatja)
- XOR (ha két részfeltétel ellentétes értékű – pl. TRUE és FALSE –, akkor TRUE)

3.3 Feltételes utasítások

A feltételes utasításokat akkor használjuk, ha programunkat érzékennyé kívánjuk tenni a program futása közben valamely kifejezés vagy változó értékének változására. Attól függően milyen értéket vesz fel a változó vagy kifejezés a programunk más-más programrészt hajt végre, azaz elágazik. Az elágazások lehetnek egyágúak, kétágúak vagy többágúak.

3.3.1 Egyágú szelekció

Az ”***if***” utasítás egyágú elágazást valósít meg. Szintaxisa a következő:

```
if (feltétel) utasítás;
```

természetesen az utasítás helyén utasításblokk is állhat. ***A feltételt mindig zárójelbe kell tenni!***

Amennyiben az ”if” után álló feltétel igaz, a program az utasítást vagy az utasításblokkot hajtja végre, ha ”hamis”, akkor nem tesz semmit.

3.3.2 Kétágú szelekció

A ***”if…else”*** kétágú elágazásnál a feltétel hamis esetére is adunk meg végrehajtandó kódot, amit az else kulcsszó után adunk meg. Szintaxisa a következő:

```
if (feltétel)
  utasítás1;
     else
  utasítás2;
```

ebben az esetben is az utasítás helyén utasításblokk is állhat.

Az ”if” után álló feltételtől függően a program ”igaz” esetben az ”utasítás1”-et hajtja végre, ”hamis” esetben, pedig az ”utasítás2”-őt.

Az elágazások egymásba is ágyazhatók, így gyakorlatilag tetszőleges mélységben építhetjük fel a feltételes utasítások láncolatát.

Előfordulhat, hogy egy ”if” utasítás valamelyik ágán belül egy másik feltétel alapján újabb elágazást építhetünk be.

Az egymásba ágyazott feltételes utasítások helyett több esetben összetett logikai kifejezést is alkalmazhatunk. A következő utasítás csak akkor hajtódik végre ha mindkét feltétel egyidejűleg teljesül.

```
if  (feltétel1 && feltétel2) utasítás;
&& = Logikai ÉS
```

A feltételek egymásba ágyazásánál arra érdemes odafigyelnünk, hogy ezzel ne rontsuk kódunk átláthatóságát illetve hatékonyságát. Ilyen esetben használjunk inkább többágú szelekciót.

3.3.3 Többágú szelekció

A ***"switch" utasítás*** segítségével többágú szelekciót tudunk programunkban megvalósítani.

```
switch kifejezés of
{
   case     érték1:
            utasítás1;
      break;
   case     érték2:
utasítás2;
break;
   case     érték3:
utasítás3;
break;
…
   default:
utasításN;
break;
  }
```

Az egyes "**case**" ágak az adott esetben végrehajtandó utasítást esetleg utasításokat tartalmazzák. Az egyes ágak utolsó utasítása a "**break**", amely elmaradása esetén a fordító hibát jelez. A "**default**" ág a "**case**"-ben felsorolt eseteken kívül eső egyéb lehetőségek bekövetkezése esetén hajtódik végre.

A "**switch**" utasítások "**case**" vezérlőinél az sbyte, byte, short, ushort, int, uint, long, ulong, char, string, enum típusú kifejezéseket használhatjuk.

4 Iteráció

A programunk írása közben gyakran találkozunk olyan dologgal, hogy ugyanazt a tevékenységsort többször kell végrehajtanunk, ilyenkor használunk ciklusokat. Általában minden programozási nyelv, így a C# is az utasítások végrehajtására több lehetőséget kínál. Ezek az előírt lépésszámú és feltételes ciklusok. A feltételes ciklusok kétfélék lehetnek – elöltesztelő vagy hátultesztelő – a feltétel helyétől függően.

4.1 Előírt lépésszámú ciklusok

A ***"for" ciklus***t, akkor használjuk, ha pontosan meg tudjuk mondani, hogy egy adott tevékenységet hányszor kell ismételni. Szintaxisa:

```
for (ciklusváltozó=kezdőérték; végérték vizsgálat; léptetés)
utasítás;
```

A "for" utasítás a ciklus feje, ahol megadjuk a ciklusváltozót, annak kezdőértékét, a végérték vizsgálatát illetve a léptetés mértékét.

A ciklusváltozó, a kezdőérték és a végérték csak sorszámozott típusú lehet. Itt kell megjegyeznünk, hogy az "utasítás" helyén összetett utasítás is állhat, ebben az esetben használjuk a korábban megismert "{…}" utasítás párt.

A "**foreach**" ciklus a "**for**" ciklus egy speciális esetre módosított változata. A gyakorlatban akkor használjuk, ha valamilyen összetett adattípus (pl. tömb) elemeit egytől egyig fel akarjuk dolgozni.

```
string[] nevek;
nevek = new string[3];
nevek[0] = "első";
nevek[1] = "második";
nevek[2] = "harmadik";
foreach (string nev in nevek)
   MessageBox.Show(nev.ToUpper());
```

A fenti példában egy neveket tartalmazó tömb (nevek) elemeit nagybetűs formában kívánjuk megjeleníteni, külön-külön üzenetablakokban.

4.2 Feltételes ciklusok

A feltételes ciklust akkor használjuk programunkban, ha nem ismerjük az ismétlések számát, ebben az esetben egy feltételtől tehetjük függővé az ismétlések számát. Attól függően, hogy a feltétel a ciklusba való belépéskor vagy kilépéskor vizsgáljuk, beszélhetünk elöltesztelő illetve hátultesztelő ciklusról.

Az elöltesztelő ciklus szintaxisa a következő:

```
while (feltétel)
{
   utasítás(ok);
}
```

A feltételt itt is csakúgy, mint a szelekciónál zárójelbe kell tenni.

A **"while" utasítás**ban szereplő belépési feltétel vezérli a ciklus végrehajtását. A ciklusmag (utasítás vagy utasításblokk) csak akkor kerül végrehajtásra, ha a feltétel igaz. Nagyon fontos megjegyezni, hogy a ciklusmagban kell gondoskodnunk arról, hogy a feltételt előbb-utóbb hamisra állítsuk, mert egyébként végtelen ciklusba jut programunk (soha nem tud kilépni a feltétel igaz volta miatt a ciklusból).

Ha a feltétel a ciklusba való belépés előtt hamissá válik, akkor a ciklus magja egyszer sem hajtódik végre.

A hátultesztelő ciklus szintaxisa a következő:

```
do
{
   utasítás(ok);
}while (feltétel)
```

A hátultesztelő ciklus esetén a ciklusmag egyszer mindenképpen lefut, hiszen a feltétel vizsgálata és kiértékelése csak a ciklus végén történik meg. A ciklusmag mindaddig végrehajtódik, amíg a feltétel igaz. A ciklusmagban itt is gondoskodnunk kell arról, hogy a feltételt előbb-utóbb hamisra állítsuk, mert egyébként végtelen ciklusba jut programunk.

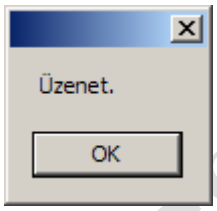
5 Üzenetablakok kezelése

5.1 Üzenetablakok

A MessageBox osztály segítségével üzenetablakot jeleníthetünk meg, mely szöveget, gombokat és egy ikont tartalmazhat. A MessageBox Osztály új példányát nem hozhatjuk létre, az üzenetet a statikus MessageBox.Show() metódus segítségével jeleníthetjük meg. A metódust több paraméterezéssel is hívhatjuk.

Ha csak egyetlen szöveges paramétert adunk át, a megadott szöveg és egy „Ok” gomb jelenik meg:

```
MessageBox.Show("Üzenet.");
```



A MessageBox esetén lehetőségünk van további paraméterek megadására is (gombok, ikonok).

A MessageBoxButtons felsorolás tagjaival adhatjuk meg, milyen gombok jelenjenek meg a felugró ablakon. A gombok felirata a Windows nyelvi beállításaitól függ. A lehetséges kombinációkat az alábbi táblázatban foglaljuk össze:

Tag neve	Megjelenő gombok
AbortRetryIgnore	Megszakítás,Újra, Ismét
OK	Ok
OKCancel	Ok és Mégse
RetryCancel	Ismét és Mégse
YesNo	Igen és Nem
YesNoCancel	Igen, Nem, Mégse

A MessageBoxIcon felsorolás tagjaival az ablakon megjelenő ikont állíthatjuk be (több elemhez is ugyanaz az ikon tartozik, kompatibilitási okok miatt):

- Asterisk, Information 
- Error, Hand, Stop 
- Exclamation, Warning 

- None – nincs ikon
- Question, 

Ezen kívül beállíthatjuk, hogy melyik gomb legyen az alapértelmezett (pl. DefaultButton1) és címet is adhatunk az ablaknak.

```
DialogResult valasz;
valasz = MessageBox.Show("Folytassuk?", "A folytatás",
MessageBoxButtons.YesNo, MessageBoxIcon.Question);
if (valasz == DialogResult.Yes)
{
    MessageBox.Show("Jó, folytatjuk");
}
else
{
    MessageBox.Show("Akkor abbahagyjuk...");
}
```

A MessageBox.Show() metódus visszatérési értékéből megtudjuk, hogy a felhasználó melyik gombbal zárta be az ablakot. A DialogResult felsorolás tagjai azonosítják a lehetséges gombokat (Abort, Cancel, Ingore, No, OK, Retry, Yes).

6 Kivételkezelés

A C# programozási nyelv a futásidejű hibák kiküszöbölésére alkalmas eszközt is tartalmaz. Ennek az eszköznek a segítségével a programjainkba hibakezelő (más néven kivételkezelő) kódrészleteket is beépíthetünk. A kivételkezelő eljárásoknak programjainkban az a lényege, hogy elkerüljük a futásközben fellépő hibák során felbukkanó hibaüzeneteket, és megvédjük programjainkat a váratlan leállástól.

6.1 Hibakezelés

A programkészítés során még a gyakorlott programozóknál is előfordul, hogy valamilyen utasítást, operátort vagy egy műveleti jelet nem a C# programnyelv szabályai szerint írt le. A leggyakoribb hiba talán az, hogy az utasítás végéről lefelejtik a pontosvesszőt. Ilyenkor a program az előző utasítás folytatásaként kezdi el értelmezni a következő utasítást, amelynek eredménye fordítási hiba lesz. Ezek a hibák mivel a program fordítása során keletkeznek, így megakadályozzák a program futtatását.

A fordítóprogram által észlelt hibákat szintaktikai hibáknak nevezzük. Egy program addig nem futtatható, amíg szintaktikai hibát tartalmaz.

Előfordulhat olyan eset is, amikor a fordító csak valamilyen figyelmeztetést küld a programíró számára. A figyelmeztetések olyan hibára hívják fel a programíró figyelmét, amelyek nem akadályozzák meg a program futtatását, de esetleg rontják programunk hatékonyságát vagy esetleg csak rontják a forrásprogram áttekinthetőségét (pl. deklarálunk egy változót, de azt sehol nem használjuk a programunkban, azaz felesleges tárhelyet foglalunk!).

A szintaktikai hibák mellett a programunk logikai hibákat is tartalmazhat, de ez nincs befolyással a program futtatására (a program fut csak bizonyos input adatokra nem azt az outputot adja, amit kellene).

A program futása során fellépő (nem külső okból származó) hibákat szemantikai hibának nevezzük. Megelőzésükről a programkódban kell megfelelő ellenőrző, hibakezelő utasításokkal gondoskodni.

6.2 A try és catch

A ”**try**” parancs segítségével a programok egyes részeihez tudunk hibakezelő eljárásokat rendelni, amelyek a futásközbeni hiba esetén a vezérlést az általunk megírt hibakezelő utasításokra adják át, ezzel kiküszöbölve a program leállását.

A ”**catch**” parancs segítségével azokat a kivételeket tudjuk elfogni, amelyek a ”**try**” blokkban keletkeznek. Itt lehet meghatározni, milyen utasítások kerüljenek végrehajtásra, és hogyan kezeljék a felmerülő hibákat. A ”**catch**” segítségével különböző hibák egyidejű kezelését is meg lehet valósítani.

Abban az esetben, ha nem használjuk a hibakezelő eljárásokat, a programunk leáll, és a hibakezelést vagy az operációs rendszer vagy pedig a .NET rendszer veszi át. Ennek általában az a következménye, hogy különböző (esetleg angol nyelvű) hibaüzeneteket ír a programunk a képernyőre, amely ”éles” programok esetén elfogadhatatlan (nem várható el a felhasználótól, hogy a programozó hanyagsága miatt megtanuljon legalább alapszinten angolul). Fokozottan érvényes ez azokra a programokra, ahol a felhasználó adatbevitelt valósít meg, amint ez a következő példában történik.

```
int oszto=0;
double hanyados;
try
{
   hanyados=10/oszto;
}
catch (ArithmeticException ar)
{
   MessageBox.Show(Convert.ToString(ar));
}
```

Amennyiben az ”oszto” változó értéke 0 a végrehajtás pillanatában, akkor a kivételkezelés működésbe lép, a ”**catch**” blokkban elkapjuk a hibát és kiíratjuk üzenetablakban az okát.

A fenti programrészletben a ”**try**” után kapcsos zárójelek közt adjuk meg a hibát okozható programrészletet. A hiba abban az esetben lép fel, ha az ”oszto” értéke 0. A ”**catch**” hibakezelő utasításait is kapcsos zárójelek közé tesszük, ezzel jelezve a fordítónak, hogy hol kezdődik és hol végződik a hibakezelés.

A ”**catch**” blokkjában lehetőségünk van a különböző okok miatt keletkezett hibák szétválasztására, valamint a hiba típusának meghatározására is. A ”**catch**” parancs a kivételt akár paraméterként is fogadhatja **catch**(Exception e). A paraméterként megadott változó ”**System.Exception**” típusú, amelyből ki lehet olvasni a hiba okát és a megfelelő hibakezelőt indíthatjuk el.

A képernyőn megjelenő hibaüzenet az esetek többségében nem túl beszédes, vagy ha igen akkor gyakran túlságosan sok, nehezen érthető információt tartalmaz. Természetesen a hibakezelés során nem az a célunk, hogy a felhasználót hosszú, számára értelmezhetetlen üzenetekkel terheljük, mert így gyakorlatilag ugyanazt tesszük, mint az eredeti hibaablak (amelyben megjelenik az eredeti hibaüzenet). Azt sem szabad elfelejteni, hogy a felhasználó tulajdonképpen nem is nagyon tudja kezelni a program futása közben fellépő hibákat, még akkor sem, ha kiíratjuk a képernyőre a hiba valamennyi paraméterét. A gyakorlatban sokkal jobb megoldásnak számít, ha megállapítjuk a hiba okát, majd pedig a megfelelő hibakezelés aktivizálásával meg is szüntetjük azt, anélkül hogy a felhasználót nehéz helyzetbe hoznánk.

A gyakorlatban általában a ”**catch**” blokkok sorrendje sem mindegy. A helyes megközelítés az, ha azokat a hibákat az általános hibák elé helyezzük, melyekre jó eséllyel számítani lehet a programban. Ilyen lehet például a felhasználói adatbekérésnél fellépő hiba vagy a matematikai műveleteknél előforduló nullával való osztás esete.

A kivétel típusokat a ”System” névtérben (namespace) találhatjuk meg.

6.3 A finally blokk

A programunk írása során előfordulhat olyan eset is, hogy egy programrészlet hibás és hibátlan működés esetén is mindenképpen lefusson. A leggyakrabban a fájlkezelésnél találunk erre példát, mivel a megnyitott fájlt hiba esetén is mindenképpen le kell zárni. Gondoljunk bele milyen hibát okozhatna, ha a fájl bezárása nélkül próbálnánk újra megnyitni a fájlt. Az ilyen típusú problémákra nyújthat megoldást a ”**finally**” parancs.

A ”**finally**” blokkban elhelyezett kód tehát minden esetben lefut – kivéve, ha a program végzetes hibával áll meg –, függetlenül az előtte keletkezett hibáktól. A következő példában a matematikai műveleteknél gyakran fellépő nullával való osztás esetét mutatjuk be a ”**finally**” blokk használatával:

```
int oszto=0;
double hanyados;
try
{
   hanyados=10/oszto;
}
catch (ArithmeticException ar)
{
   MessageBox.Show(Convert.ToString(ar));
}
finally
{
   MessageBox.Show("A program ezen része mindenképpen lefut");
}
```

6.4 Kivételek feldobása

A C# programozási nyelv lehetőséget ad a programozó által definiált kivételek használatára is. A kivételek dobása a ”**throw**” paranccsal történik.

```
throw (exception);
throw exception;
```

A program bármely szintjén dobhatunk a kivételt a ”**throw**” parancs segítségével, és egy tetszőleges ”**catch**” blokkal el is tudjuk kapni. Amennyiben nem kapjuk el sehol a programunkban, akkor az a Main() függvény szintjén is megjelenik, majd végül az operációs rendszer lekezeli a saját hibakezelő eljárásával, ami persze a legtöbbször a programunk leállásával jár.

6.5 Checked és unchecked

A C# programozási nyelv tartalmaz két további parancsot a kivételek kezelésére és a hibák javítására. Az egyik a ”**checked**” a másik pedig az ”**unchecked**” parancs. Amennyiben az értékadáskor a változóba olyan értéket kívánunk elhelyezni, amely az adott változó értéktartományában nem fér el, ”**OverflowException**” hibával áll meg a programunk. Szerencsés esetben az ilyen jellegű hibákat el tudjuk kapni, a megfelelő catch{} blokk alkalmazásával, de az ”**unchecked**” parancs használatával megakadályozhatjuk a kivétel keletkezését is, mivel ilyen esetben elmarad a hibaellenőrzés.

```
unchecked
{
   byte a=300;
}
```

A fenti példában a byte típusú változó maximum 255-ig tud értéket tárolni. Az értékadás ennek ellenére megtörténik, és a változóba bekerül a csonkított érték, persze csak akkora, amekkora még elfér benne.

A ”**checked**” alkalmazásával pontosan az előbbi folyamat ellenkezőjét érhetjük el. Az ellenőrzés mindenképpen megtörténik, és kivétel keletkezik a hiba miatt.

A ”**checked**” és ”**unchecked**” parancsok nem csak blokként használhatóak, hanem egy kifejezés vagy értékadás vizsgálatánál is. Ekkor a következő alakban írhatjuk őket:

```
checked(kifejezés, művelet, értékadás);
unchecked(kifejezés, művelet, értékadás);
```

A bemutatott 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 programjainkban.

A kivételkezelés fejezetben leírtak alapján könnyen beláthatjuk, hogy a hibakezelés illetve a kivételek kezelése nagyon fontos és hasznos lehetőség a C# programozási nyelvben. Ezek nélkül nehéz lenne elképzelni hibátlanul működő programokat. Természetesen meg kell vallanunk, hogy a kivételkezelést alkalmazó programok sem tökéletesek, de jó esély van arra, hogy programunk nem áll le végzetes hibával és nem keserítik sem a programozó, sem pedig a felhasználók életét.

7 Állománykezelés

A C# nyelv programozása során elsősorban a hagyományos C nyelvből megismert fájlkezelési technikáit szokás használni. Abban az esetben, ha a fájl megnyitásakor valamilyen hiba lép fel, akkor a programunk többnyire valamilyen hibaüzenettel leáll. A C# programozási nyelv persze lehetővé teszi az állománykezelés során a be- és kiviteli műveleteknél fellépő műveletek hiba ellenőrzését.

A programozási nyelvekben az állományokat tartalmuk alapján szöveges (text), típusos és típusnélküli fájlok csoportjára osztjuk. A szöveges fájlokat soros, míg a másik kettőt soros és közvetlen eléréssel egyaránt elérhetjük.

Fájlkezelés, állomány fajták

- Szöveges
- Bináris

Fel kell hívni a figyelmet még arra is, hogy fájlkezelés során az állomány elejére be kell szúrnunk a következő sort, mert csak ezután lesz képes programunk a fájlok kezelésére.

```
using System.IO;
```

7.1 Szöveges állományok kezelése

7.1.1 Szöveges fájlok írása, bővítése, olvasása

A következő példában a ”c:\” könyvtárban lévő ”teszt.txt” fájlt hozzuk létre a ”**File.CreateText**” metódus segítségével, amennyiben még nincs ilyen nevű fájl a könyvtárban, ha igen, akkor nem történik semmi.

```
string utvonal = @"c:\teszt.txt";
if (!File.Exists(utvonal))    //ha nem létezik a fájl az adott
könyvtárban,  csak akkor ír bele
{
 using (StreamWriter sw = File.CreateText(utvonal))  //Létrehoz egy
szövegfájlt írásra
   {
       //ide jönnek az utasítások
   }
}
```

A következő részben már ellenőrzés nélkül írunk a fájlba a ”**File.AppendText**” metódus segítségével. Amennyiben a fájl nem létezik akkor létrehozásra kerül, ezért nem szükséges az

ellenőrzés, ha pedig létezik akkor megmarad a fájl tartalma és a megadott résszel kibővül az állomány a végén.

```
using (StreamWriter sw = File.AppendText(utvonal))
```

Ebben a programrészletben egy fájlt nyitunk meg olvasásra a ”**File.OpenText**” metódus segítségével és addig olvasunk belőle, amíg ”**null**” értéket nem olvasunk. A beolvasott adatokat kiíratjuk üzenetablakban a képernyőre.

```
string utvonal = @"c:\teszt.txt";
using (StreamReader sr = File.OpenText(utvonal))
{
   string s = "";
   while ((s = sr.ReadLine()) != null)
  {
      MessageBox.Show(s);
  }
}
```

7.1.2 Bináris fájlok írása, olvasása

Az előző részben már szóltunk a szöveges állományok kezeléséről. Vannak azonban olyan esetek, amikor egyszerűbb lenne a munkánk, ha nem kellene szöveges állományból illetve állományba konvertálni az adatainkat (pl. számok kezelése során). Előfordulhat olyan eset is, hogy nem praktikus a tároláshoz szöveges állományokat használnunk, mert egy külső egyszerű kis szövegszerkesztő programmal meg lehet nézni a tartalmát (pl. jelszavak tárolásánál, persze lehet kódolni őket, de ez már újabb programozási problémákat vet fel).
Az ilyen jellegű problémák feloldására találták ki a bináris állományokat, amelyek tartalmát nem lehet szövegszerkesztő programokkal megtekinteni illetve lehet bennük közvetlenül számokat tárolni, mindenféle konverzió nélkül. A bináris adatok jellemzője, hogy megtartják az adattípus eredeti tárolási formáját, azaz nem alakítják szöveggé.
A következő kis példaprogramunkban 10 darab egész számot tárolunk:

```
FileStream File = new FileStream(args[0], FileMode.CreateNew);
BinaryWriter bw = new BinaryWriter(File);
for (int i = 0; i < 10; i++)
{
    bw.Write(i);
}
bw.Close();
File.Close();
```

A 2. táblázatban a ”FileMode” lehetséges értékeit mutatjuk be:

2. táblázat A FileMode értékei

Érték	Leírás
Append	Megnyit egy létező fájlt vagy újat készít.
Create	Új fájlt készít. Ha a fájl már létezik, a program törli és új fájlt hoz létre helyette.
CreateNew	Új fájlt hoz létre. Ha a fájl már létezik, kivételt vált ki, amit illik lekezelnie a programozónak.
Open	Megnyit egy létező fájlt.
OpenorCreate	Megnyit egy fájlt, vagy ha nem létezik létrehozza.
Truncate	Megnyit egy létező fájlt és törli a tartalmát.

A ”FileStream” objektumot létrehozása után azonnal fel kell készítenünk a bináris adatok fogadására. Ezt úgy tudjuk megoldani, hogy a BinaryWriter típust kapcsoljuk az objektumhoz.

```
FileStream File = new FileStream(FileMode.Create);
BinaryWriter  bw = new BinaryWriter(File);
   //A következő lépésben az objektumba kötzvetlenül tudunk írni adatokat, a Write tagfüggvénnyel
bw.Write(i);
   //Ha befejeztük az írást a fájlba, akkor gondoskodnunk kell a megnyitott folyamok bezárásáról
bw.Close();
File.Close();
   //Ha állományba kiírtunk valamit, akkor a fájlból olvasáskor elöltesztelős ciklust használunk
FileStream File = new FileStream(FileMode.Open);
BinaryReader  br = new BinaryReader(File);
lstLista.Items.Clear();
while (br.PeekChar() != -1)
{
   lstLista.Items.Add(br.ReadInt32());
}
br.Close();
File.Close();
```

while (br.PeekChar() != -1) sorban olvassuk be a BinaryReader osztály PeekChar() tagfüggvényével az adatfolyam következő karakterét. Ez a tagfüggvény mindig a következő

karaktert adja vissza, egy kivételtől eltekintve, ha elértük az adatfolyam végét, ebben az esetben ”-1”-et ad vissza.

7.2 Könyvtár műveletek

Új könyvtár létrehozása:

```
System.IO.Directory.CreateDirectory (string)
```

Könyvtár tartalmának mozgatása egyik (source) helyről a másikra (destination):

```
System.IO.Directory.Move(source, destination);
```

Üres könyvtár tartalmának törlése:

```
System.IO.Directory.Delete (string)
```

Alkönyvtár létrehozása:

```
System.IO.DirectoryInfo.CreateSubdirectory (string)
```

Visszaadja az adott könyvtárban lévő alkönyvtárak neveit:

```
System.IO.Directory.GetDirectories (string)
```

A paraméterében megadott könyvtár létezését vizsgálja

```
System.IO.Directory.Exists(string)
```

Visszaadja az aktuális alkönyvtárat

```
System.IO.Directory.GetCurrentDirectory()
```

A gyökérkönyvtárat adja vissza

```
System.IO.Directory.GetDirectoryRoot(string);
```

A gépen található logikai meghajtókat listázza ki

```
System.IO.Directory.GetLogicalDrives();
```

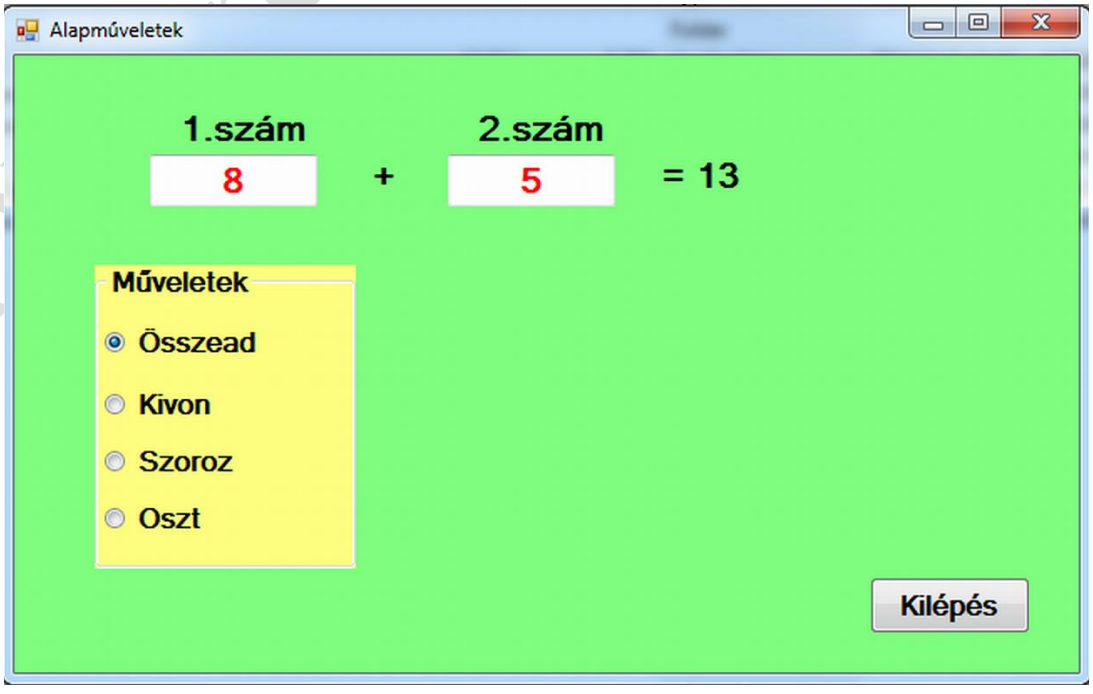
8 MINTA feladatok

A minta feladatok összeállítása olyan elemekből épül fel, amelyek esetleg önállóan más projektben is használhatóak. Szerepel benne alapműveletek elvégzését bemutató program, listakezelést bemutató és állománykezelést (szöveges és bináris) valamint könyvtárkezelési utasításokat bemutató program.

8.1 Példa: Alapműveletek elvégzését bemutató program

Példánkban az alapműveletek elvégzését bemutató programot készítünk. Egy csoportból választhatjuk ki, hogy összeadni, kivonni, szorozni vagy osztani szeretnénk, melyet radiogombok segítségével valósítunk megy GoupBox komponensbe ágyazva őket. Minden esetben két egész számmal dolgozunk, amelyekkel a kiválasztott művelettípust hajtjuk végre, majd megjelenítjük annak eredményét. Az osztásnál fontos felhívni a figyelmet arra, hogy le kell védenünk azt az esetet, ha a felhasználó a második szövegdobozba 0-át ír vagy üresen hagyja, így a feladat során be tudjuk mutatni az utólagos hibakezelés megvalósítását gyakorlati feladat formájában. Ugyancsak az osztás esetén mutatjuk be, hogyan lehet az eredményeket különböző számú tizedes jegyek formájában megjeleníti. A kilépés gombra kattintva az alkalmazásunk fejezze be a program futtatását! Nem kell visszakérdezni, hogy valóban ki akar-e lépni a programból!

1. Hozzunk létre új projektet „Alapmuveletek” néven!
2. Tervezzük meg az űrlapot az alábbiak szerint:



3. Az űrlapon szereplő komponensek tulajdonságait állítsuk be az alábbiak szerint:

Komponens	Text	Name
Form	Alapműveletek	frmMuvelet
GroupBox	Műveletek	grpMuveletek
Label	1.szám	lblSzam1
Label	2.szám	lblSzam2
Label	+	lblMuveletijel
Label	= 13	lblEredmny
TextBox	8	txtSzam1
TextBox	5	txtSzam2
RadioButton	Összead	rgOsszead
RadioButton	Kivon	rgKivon
RadioButton	Szoroz	rgSzoroz
RadioButton	Oszt	rgOszt
Button	Kilépés	btnKilepes

Az első és második szövegdobozba beírt számokkal (ha nem változtattuk meg, akkor az eredeti 8 és 5-tel) számol a program. Mivel a szövegdobozba beírt értéket szövegesen tárolja a program azt át kell konvertálni egész számmá és tároljuk egy általunk deklarált változóban:

```
x = int.Parse(txtSzam1.Text)
```

Mivel a konverzió során gyakran léphet fel hiba, pl. azért mert nem egész számot írunk vagy üresen hagyjuk a szövegdobozt, stb., így kénytelenek vagyunk a fellépő hibalehetőséget lekezelni egy „try-catch” blokkal. Itt fontos megjegyeznünk, hogy a „try” és „catch” után midig ki kell tenni az összetett utasítás zárójelét, még akkor is, ha csak egy utasítás kerül a blokkba. Ennek a működése úgy történik, hogy ha a „try” blokkban a konverzió során hiba lép fel, akkor automatikusan átkerül a vezérlés a „catch” blokkra és azokat az utasításokat hajtja végre. jelen esetben ez egy üzenetet küld a felhasználónak és visszaállítja a változó valamint a szövegdoboz értékét az aláértelmezettre. Ez ugyanígy működik a második megadott szám esetében is.

```
try
{
   x = int.Parse(txtSzam1.Text);
}
```

```
catch
{
    MessageBox.Show("Hibás adat az első szövegdobozban!");
    x = 8;
    txtSzam1.Text = "8";
}
```

Ezt követően az x és y változóban tároljuk tovább a két a felhasználó által a szövegdobozokban megadott értéket.

A következő eseménysor, amikor a radiogombokat járjuk végig. Le kell ellenőríznünk, hogy melyik van közülük kiválasztva, hiszen a négy közül mindig csak egy lehet kiválasztott állapotban. Ezt onnan tudjuk, hogy megvizsgáljuk a „checked” állapotát és amelyik „true” állapotban van az van kiválasztva. Mindegyik esetben be kell állítanunk a műveleti jelet a megfelelő értékre és ki kell számolni az „osszeg”, „kulonbseg”, „szorzat” és „hanyados” értékeit. Az „lblEredmeny” címke szövegét az „=” jel és a megfelelő változót („osszeg”, „kulonbseg”, „szorzat” és „hányados”) fűzzük össze. Mivel ezek egész típusúak, így a „ToString() metódussal” átalakítjuk szöveggé, így már két szöveget könnyen össze tudunk fűzni.

A hányados számolása már egy kicsit bonyolultabb módon történik, hiszen meg kell vizsgálni, hogy a szövegdobozban nem 0 érték van-e, mert ha igen, akkor hibaüzenetet kell küldeni a felhasználónak, hogy 0-val nem lehet osztani! Ezen kívül két egész szám eredménye a C#-ban egész számot eredményez, így törtszámot nem tudnánk kiíratni, ezért mind az „x”, mind pedig „y” változóra rákényszerítjük a „double” típust, hogy tört számot kapjunk eredményként.,

```
double hanyados = (double)x / (double)y
```

Ezt követően meg kell vizsgálni, hogy az osztás maradéka nulla-e vagy sem, hiszen ez határozza meg, hogy kell-e tizedes jegyeket kiíratni.

```
if (x % y != 0)
    lblEredmeny.Text = " = " + hanyados.ToString("### ### ##0.000");
else
    lblEredmeny.Text = " = " + hanyados.ToString("### ### ###");
```

A kiírásnál ún. maszkoló karaktereket használunk (0 vagy #). Alapvetően a két karakter a legtöbb esetben helyettesítheti egymást, mint a példa is mutatja. Az eltérés akkor van, ha 0 és

1 közötti tört értéket kapunk, mert ekkor a tizedes előtt mindenképpen 0 ke3ll, hogy álljon, mert akkor jelenik meg korrektül az érték (pl. 0.6) egyébként ha #-et használjuk, akkor (.6) jelenik meg. A #-ek azért vannak jobbról balra hármasával Space-el elválasztva, hogy ezresenként tördelje a nagyobb számokat.

Ezt követően a radiogombok „CheckedChanged” eseményét használjuk:

```
private void rgOsszead_CheckedChanged(object sender, EventArgs e)
```

Mivel a végrehajtandó utasításokat már megírtuk a:

```
private void btnSzamol_Click(object sender, EventArgs e)
```

eseményben, így itt már elég csak hivatkozni rá. Így járunk el mind a négy radiogomb esetén.

Ugyanezt az eseményt hívjuk meg mindkét szövegdoboz értékének változása esetén:

```
private void txtSzam1_TextChanged(object sender, EventArgs e)

private void txtSzam2_TextChanged(object sender, EventArgs e)
```

4. frmMuvelet.cs:

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

namespace Szamolas
{
    public partial class frmMuvelet : Form
    {
        public frmMuvelet()
        {
            InitializeComponent();
        }
        private void btnKilepes_Click(object sender, EventArgs e)
        {
            Application.Exit();
        }
        private void btnSzamol_Click(object sender, EventArgs e)
        {
            int x, y;
            try
            {
                x = int.Parse(txtSzam1.Text);
            }
            catch
            {
                MessageBox.Show("Hibás adat az első szövegdobozban!");
                x = 8;
                txtSzam1.Text = "8";
            }
```

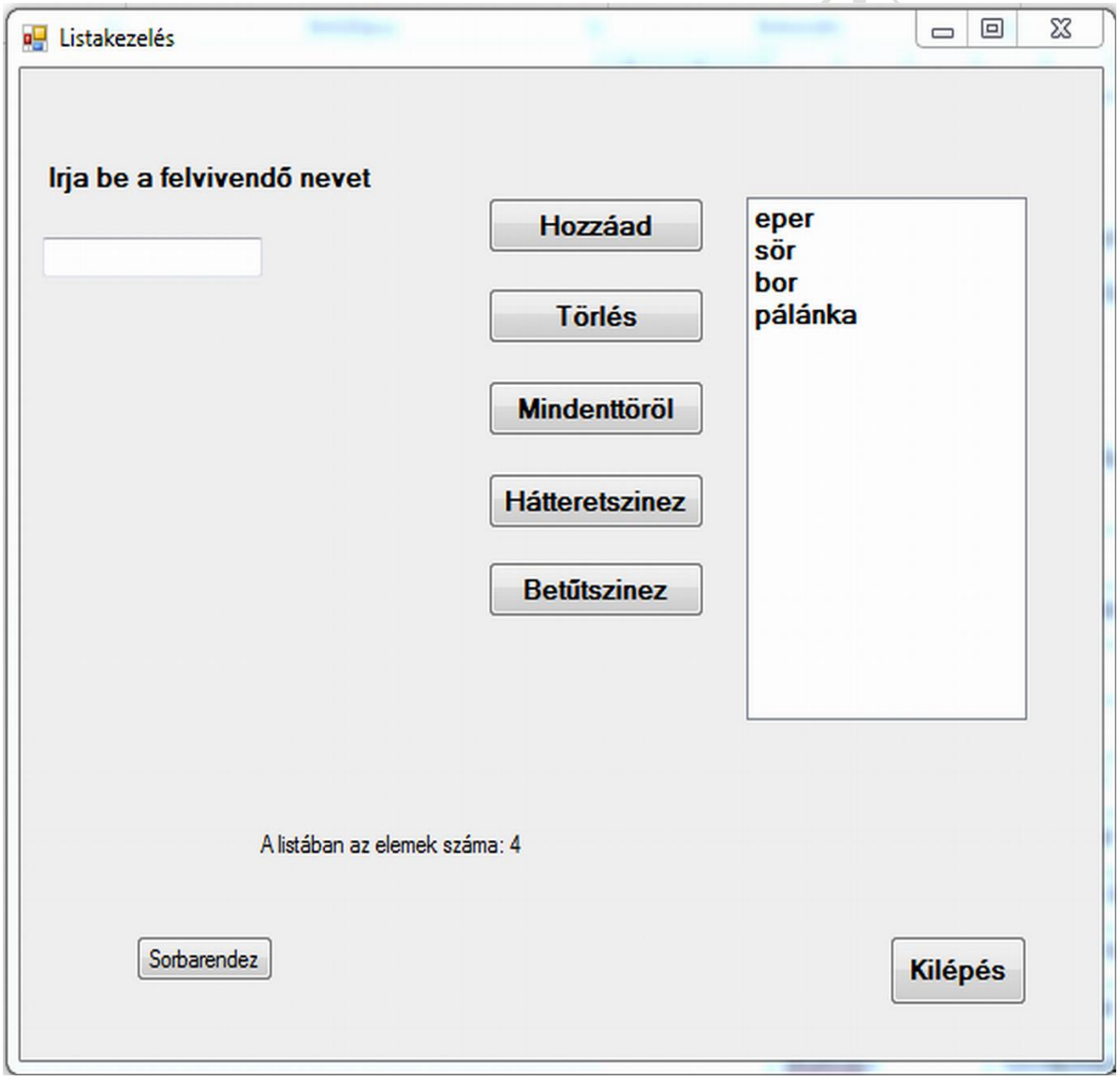
```
            try
            {
                y = int.Parse(txtSzam2.Text);
            }
            catch
            {
                MessageBox.Show("Hibás adat a második szövegdobozban!");
                y = 5;
                txtSzam2.Text = "5";
            }
            if (rgOsszead.Checked)
            {
                lblMuveletijel.Text = "+";
                int osszeg = x + y;
                lblEredmeny.Text = " = " + osszeg.ToString();
            }
            if (rgKivon.Checked)
            {
                lblMuveletijel.Text = "-";
                int kulonbseg = x - y;
                lblEredmeny.Text = " = " + kulonbseg.ToString();
            }
            if (rgSzoroz.Checked)
            {
                lblMuveletijel.Text = "*";
                int szorzat = x * y;
                lblEredmeny.Text = " = " + szorzat.ToString();
            }
            if (rgOszt.Checked)
            {
                lblMuveletijel.Text = "/";
                if (y != 0)
                {
                    double hanyados = (double)x / (double)y;
                    if (x % y != 0)
                        lblEredmeny.Text = " = " + hanyados.ToString("### ###
                              ##0.000");
                    else
                        lblEredmeny.Text = " = " + hanyados.ToString("### ### ###");
                }
                else
                {
                    MessageBox.Show("Nullával nem osztunk VAZZE!");
                    y = 5;
                    txtSzam2.Text = "5";
                }
            }
        }
        private void rgOsszead_CheckedChanged(object sender, EventArgs e)
        {
            btnSzamol_Click(sender, e);
        }
        private void rgKivon_CheckedChanged(object sender, EventArgs e)
        {
            btnSzamol_Click(sender, e);
        }
        private void rgSzoroz_CheckedChanged(object sender, EventArgs e)
        {
            btnSzamol_Click(sender, e);
        }
        private void rgOszt_CheckedChanged(object sender, EventArgs e)
        {
```

```
            btnSzamol_Click(sender, e);
        }
        private void txtSzam1_TextChanged(object sender, EventArgs e)
        {
            btnSzamol_Click(sender, e);
        }
        private void txtSzam2_TextChanged(object sender, EventArgs e)
        {
            btnSzamol_Click(sender, e);
        }
    }
}
```

8.2 Példa: Listakezelő program

Példánkban a listakezelésének elemeit fogjuk bemutatni. Megnézzük, hogyan lehet listába felvinni adatokat, törölni onnan (nem csak egyet egyszerre) valamint, hogyan lehet a teljes lista tartalmát törölni. A következő rézben megnézzük, hogyan lehet a lista hátterét illetve betűszínét megváltoztatni ColorDialógus ablak használatával. Valamint végezetül azt is megnézzük, hogyan lehet a lista elemeit ABC szerint rendezni, ez azt is jelenti, hogy sajnos számokat nem tud rendezni a lista ezzel a módszerrel, csak szöveges adatokat! A kilépés gombra kattintva az alkalmazásunk fejezze be a program futtatását! Nem kell visszakérdezni, hogy valóban ki akar-e lépni a programból!

1. Hozzunk létre új projektet „Listakezelo” néven!
2. Tervezzük meg az űrlapot az alábbiak szerint:



3. Az űrlapon szereplő komponensek tulajdonságait állítsuk be az alábbiak szerint:

Komponens	Text	Name
Form	Könyvtárak kezelése	frmLista
ListBox	---	lstLista
Button	Hozzáad	btnHozzaad
Button	Törlés	btnTorol
Button	Mindent töröl	btnMinden
Button	Hátteret színez	btnHatter
Button	Betűt színez	btnBetu
Button	Sorbarendez	btnSorbarendez
Button	Darabszám	btnDarabszam
Button	Kilépés	btnKilepes
TextBox	----	txtNev
Label	Írja be a felvivendő nevet!	lblNev
Label	----	lblDarab
Button	Kilépés	btnKilepes

A feladat első lépésében a szövegdobozba begépelt nevet kell felvinni a listába. Ezt megelőzően többféle ellenőrzést kell elvégezni. Meg kell nézni, hogy a beírt szó előtt vagy után nincsenek-e üres karakterek. Ha vannak azokat el kell távolítani:

```
txtNev.Text = txtNev.Text.Trim().
```

Ellenőrizni kell, hogy a felvinni kívánt név szerepel-e már listában (Contains() metódus), hogy a duplikátumokat kiszűrjük. Ráadásul nem tudhatjuk, hogy a felhasználó kis vagy nagy betűkkel írta be a szöveget (pl. CAPS LOCK bekapcsolva maradt), azért mindet kisbetűssé alakítjuk a „ToLower()”. metódussal.

```
if (lstLista.Items.Contains((txtNev.Text.ToLower()))).
```

Azt is ellenőrizni kell, hogy nem üres szöveget akar-e felvinni a felhasználó a listában, mert ezt a lehetőséget is ki kell zárni. Amennyiben a bevitt szöveg minden kritériumnak megfelelt, akkor lehet felvinni a listába (Add() metódus), de a felvitel előtt kisbetűssé kell alakítani, akármilyen formában is írta be a felhasználó. A felvitelt megkönnyíthetjük azzal, hogy a felhasználó amint begépeli a szöveget a szövegdobozba nem kell az egérrel a „Hozzáad”

feliratú gombra kattintani, hanem az ENTER lenyomására is fel tudja vinni a szöveget a listába. Ehhez viszont ki kell üríteni minden felvitel után a szövegdobozt és a fókuszt ráállítani. txtNev.Text = ""; és txtNev.Focus();

Íme a kódrészlet, ami az ENTER kezeléséért felelős.

```
private void txtNev_KeyPress(object sender, KeyPressEventArgs e)
{
   if (e.KeyChar==13)
   {
     btnHozzaad_Click(sender, e);
   }
}
```

A törlésnél meg kell oldani, hogy egyszerre több elemet is ki tudjon törölni a felhasználó, ehhez „One” helyett „MultiExtended”-re állítani a „SelectionMode” tulajdonságot. Ilyenkor mind a Ctrl, mind pedig a Shift billentyű használható a kijelöléshez. A kijelölés után a program ellenőrzi, mely elemek vannak kijelölve és indexük alapján ezeket törli (RemoveAt() metódus). A „gyujto” változóba számoljuk, hogy van-e egyáltalán egy elem is kijelölve törlésre, mert ha nincs akkor hibát jelezni a metódus végrehajtása.

A „Mindent töröl” feliratú gomb törli az összes elemet (Clear() metódus) a listából. Ezt nem szabad ellenőrizetlenül kiadni, mindenképpen rá kell kérdezni a felhasználónál, hogy tényleg „Törölni akarja valamennyi elemet?” vagy csak véletlenül kattintott rá és csak „Igen” válasz esetén hajtsuk végre a törlést!

A listában lévő elemek sorba rendezését a lista „Sorted” tulajdonságának „true”-ra állításával tehetjük meg. Figyelem az eredeti sorrendet ezt követően már nem tudjuk visszaállítani!

A „Darabszám” feliratú gomb rejtett a felhasználó elől, mert ezt a metódust minden művelet után meghívjuk (Hozzáadás, Törlés, Mindent töröl,– Form Load– hogy induláskor is korrekt értéket mutasson).

```
lblDarab.Text = "A listában az elemek száma: " +
        lstLista.Items.Count.ToString();
```

Így mindig az aktuális elemszámot fogja mutatni!

A „Hátteret színez” feliratú gomb esetén a „ColorDialog” ablakot hívjuk meg, de előtte elmentjük a lista háttér színét, hogy ha a felhasználó a „Mégsem” gombra kattint, akkor az eredeti szín visszaállítható legyen, mert egyébként fekete lenne a háttér, mivel ez az alapszíne a Dialógusablaknak. `lstLista.BackColor = colorDialog.Color;`

A „Betűt színez” feliratú gomb esetén a ColorDialógus ablakot hívjuk meg, de előtte elmentjük a lista betű színét, hogy ha a felhasználó a „Mégsem” gombra kattint, akkor az eredeti szín visszaállítható legyen, mert egyébként fekete lenne a háttér, mivel ez az alapszíne a Dialógusablaknak. `lstLista.ForeColor = colorDialog.Color;`

4. frmLista.cs:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
namespace Listakezelo
{
    public partial class frmLista : Form
    {
        public frmLista()
        {
            InitializeComponent();
        }
        private void btnKilépés_Click(object sender, EventArgs e)
        {
            Application.Exit();
        }
        private void btnHozzaad_Click(object sender, EventArgs e)
        {
           txtNev.Text = txtNev.Text.Trim();

           if (lstLista.Items.Contains((txtNev.Text.ToLower())))
           {
               MessageBox.Show("Ilyen elem már van a listában");
               txtNev.Focus();
           }
           else
           {
               if (txtNev.Text != "")
               {
                   lstLista.Items.Add((txtNev.Text.ToLower()));

                   txtNev.Text = "";
                   txtNev.Focus();
                   btnDarabszam_Click(sender, e);
               }
               else
               {
                   MessageBox.Show("Üres szöveget nem viszek fel!");
               }
           }
        }
        private void txtNev_KeyPress(object sender, KeyPressEventArgs e)
        {
            if (e.KeyChar==13)
            {
            btnHozzaad_Click(sender, e);
            }
        }
        private void btnDarabszam_Click(object sender, EventArgs e)
        {
```

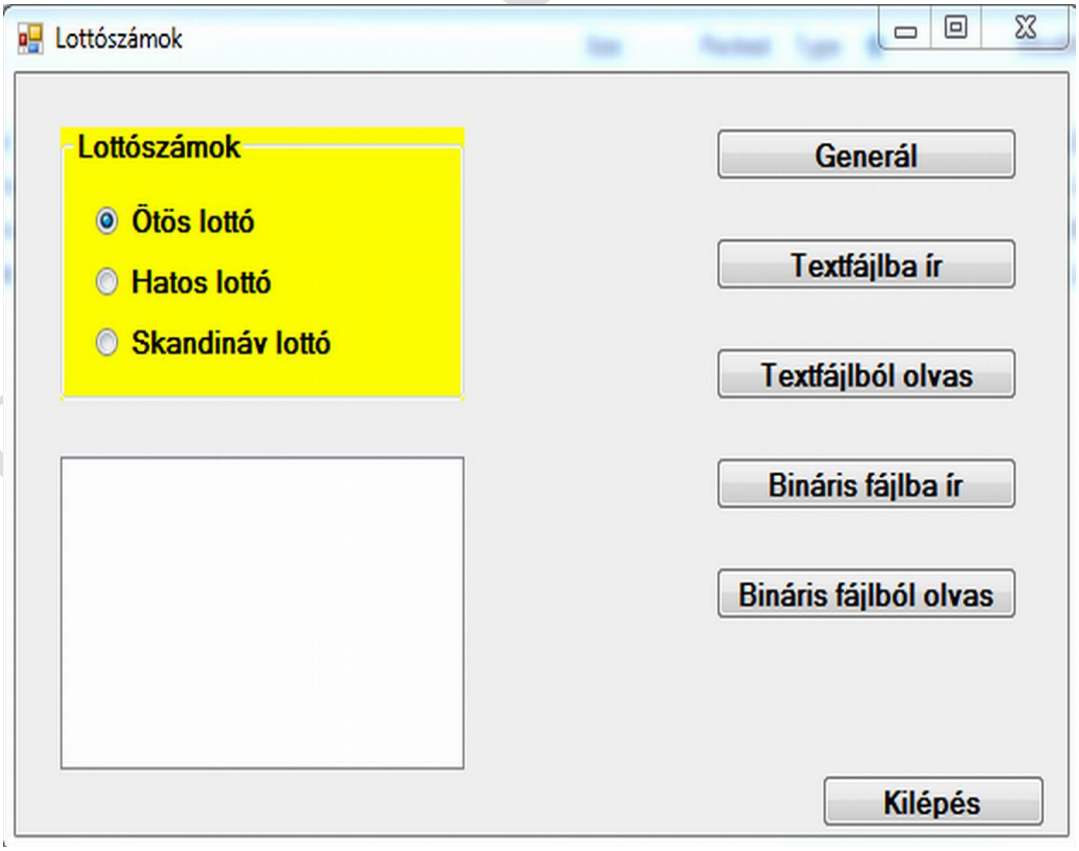
```
            lblDarab.Text = "A listában az elemek száma: " +
                lstLista.Items.Count.ToString();
        }

        private void frmLista_Load(object sender, EventArgs e)
        {
            btnDarabszam_Click(sender, e);
        }
        private void btnMinden_Click(object sender, EventArgs e)
        {
            DialogResult valasz;
            valasz = MessageBox.Show("Valóban törölni akar minden elemet?",
               "Törlés", MessageBoxButtons.YesNo, MessageBoxIcon.Question,
                MessageBoxDefaultButton.Button2);
            if (valasz==DialogResult.Yes)
            lstLista.Items.Clear();
        }
        private void btnTorol_Click(object sender, EventArgs e)
        {
            int i=0, gyujto=0;
            while (i <= lstLista.Items.Count - 1)
            {
                if (lstLista.SelectedIndex == i)
                {
                    lstLista.Items.RemoveAt(lstLista.SelectedIndex);
                    gyujto = 1;
                    i--;
                }
                i++;
            }
            if (gyujto == 0)
            {
                MessageBox.Show("Válassszon legalább egy elemet");
            }
            btnDarabszam_Click(sender, e);
        }
        private void btnSorbarendez_Click(object sender, EventArgs e)
        {
            lstLista.Sorted = true;
        }
        private void btnHatter_Click(object sender, EventArgs e)
        {
            colorDialog.Color = lstLista.BackColor;
            colorDialog.ShowDialog();
            lstLista.BackColor = colorDialog.Color;
        }
        private void btnBetu_Click(object sender, EventArgs e)
        {
            colorDialog.Color = lstLista.ForeColor;
            colorDialog.ShowDialog();
            lstLista.ForeColor = colorDialog.Color;
        }
    }
}
```

8.3 Példa: Véletlen számok generálása lottószámokhoz, majd kiírásuk fájlba és visszaolvasásuk fájlból (szöveg, bináris)

Példánkban szerepeljen három rádiógomb (RadioButton) egy konténerben (GroupBox), amelyben a felhasználó kiválaszthatja, hogy ötös, hatos vagy skandináv lottóhoz kíván véletlen számokat generálni. Az első esetben 5 számot generálunk 90-ből, a másodikban 6-ot 45-ből, míg a harmadikban 7-et 35-ből. A generálást a „Generál” feliratú gombra kattintással indítjuk, melyeket egy „ListBox” komponensben jelenítünk meg. A felhasználónak lehetősége van szöveges és bináris fájlba kiírni illetve onnan visszaolvasni az adatokat igény szerint. Mivel a fájlban lévő adatok száma nem ismert, hiszen nem tudhatjuk, hogy utoljára hány számot írtunk ki az állományba, így a számok beolvasását elöltesztelő ciklussal oldjuk meg. Kiíráskor két lehetőségünk is van egyrészt a listában lévő számok darabszámát kell meghatározni és ezt követően már előírt lépésszámú ciklussal is kiírhatjuk a számokat állományba, de itt is alkalmazhatjuk a már jól bevált elöltesztelő ciklust.

A „Kilépés” feliratú gombra kattintva az alkalmazásunk fejezze be a program futtatását! Egy üzenet ablakban kérdezzünk vissza, hogy valóban be szeretné-e zárni az alkalmazást a felhasználó és csak „igen” választása esetén lépjünk ki a programból.

1. Hozzunk létre Windows alapú új projektet „Veletlenfajlba” néven !
2. Tervezzük meg az űrlapot az alábbiak szerint:

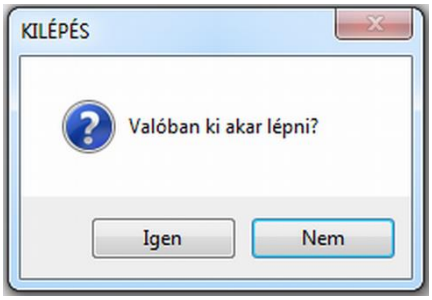


3. Az űrlapon szereplő komponensek tulajdonságait állítsuk be az alábbiak szerint:

Komponens	Text	Name
Form	Lottószámok	frmLotto
GroupBox	Lottószámok	grpLotto
RadioButton	Ötös lottó	rgOtos
RadioButton	Hatos lottó	rgHatos
RadioButton	Skandináv lottó	rgHetes
ListBox	-----	lstSzamok
Button	Generál	btnGeneral
Button	Textfájlba ír	btnFajlbair
Button	Textfájlból olvas	btnFajbololvas
Button	Bináris fájlba ír	btnBinarisbair
Button	Bináris fájlból olvas	btnBinarisbololvas
Button	Kilépés	btnKilepes
openFileDialog	----	openFileDialog1
saveFileDialog	----	saveFileDialog1

A feladat kódja tartalmaz egy eljárásszintű metódust, amellyel a különböző típusú lottószámokból tudunk véletlenszerűen generálni megadott darabszámot. Két bemenő paramétere van a tartomány felsőhatára, amelyből húzhatunk, valamint a kihúzható számok darabszáma (mindkettő egész típusú), `void lottoszamok(int szam, int darab)`.

A „Kilépés” feliratú gomb kódját a `private void btnKilepes_Click(object sender, EventArgs e)` eljárás valósítja meg, ahol a „DialogResult” osztály által visszaadható értékek közül a „Yes”-t vizsgáljuk, mert ez esetben választotta a felhasználó az „igen” gombot. A rafináltabb felhasználók megpróbálkozhatnak az űrlap jobb felső sarkában



lévő „x” gomb vagy az F4 funkcióbillentyű megnyomásával, hogy kikerüljék a fenti eseményt. Ezért került beépítésre a következő eljárás: `protected override void OnClosing(CancelEventArgs e),` amely az ablak bezárásakor kerül meghívásra, így mindkét említett eseményt egyszerre levédi a program. Itt fontos az `e.Cancel = true;` parancs, mert egyéb esetben ugyanazt csinálja a program mindkét gomb lenyomására.

A fájlba írásnál és olvasásnál is függetlenül attól, hogy szöveges vagy bináris állománnyal dolgozunk beállítjuk a Fájldialógus ablak címét, a kezdeti könyvtárat, hogy alapértelemzés szerint honnan illetve hova olvasson vagy írjon illetve beállítunk egy fájlszűrőt, hogy az ablak csak.az adott kiterjesztésű állományokat mutassa (*.txt vagy *.* illetve (*.dat vagy *.*). A *.* azt jelenti, hogy az összes fájlt mutassa függetlenül a nevétől és kiterjesztésétől. Ezzel a három beállítssal sok kellemetlen plusz lépéstől tudjuk megóvani a felhasználót, azaz felhasználóbarátabbá tesszük a programunkat. A „ShowDialog()” metódus szolgál a fájldialógus ablakok menyitására. Mindenesetben a fájlműveletek előtt ellenőrízni kell, hogy a felhasználó adott-e meg fájlnevet illetve, hogy az OK gombot nyomta-e meg. Mégsem esetén ugyanis nem kell fájlműveletet végezni, hiszen a felhasználó közben meggondolta magát, míg fájlnév nélkül az operációs rendszer nem engedi meg fájl létrehozását! Ezt követően már természetesen különböznek a feltételek belső utasításai aszerint, hogy írunk vagy olvasunk és aszerint is, hogy szöveg vagy bináris állománnyal van dolgunk.

Szövegfájlba íráskor használjuk a TextWriter osztályt:

```
TextWriter tw = File.CreateText(saveFileDialog1.FileName)
```

Szövegfájlból olvasáskor pedig használjuk a TextReader osztályt:

```
TextReader tr = File.OpenText(openFileDialog1.FileName)
```

Itt fontos megemlíteni, hogy alaból ezeket az osztályokat a C# nem ismeri fel, ezért a kód elején vegyük fel hiatkozásként a using-ok közé a using System.IO-t.

Szövegfájlból beolvasásnál addig olvasunk, amíg a beolvasott érték már nem null érték:

```
while ((sor = tr.ReadLine()) != null).
```

Bináris állományból való beolvasáskor viszont más lesz a leállási feltétel:

```
while(br.PeekChar()!=-1).
```

Figyelem, minden visszaolvasás előtt a listát űrítsük ki a „Clear()” metódussal, mert egyébként a korábbi beolvasások eredményeit is benne hagyja, ami nagyon zavaró lehet a felhasználó számára!

Minden állomány kezelésnél kulcskérdés az állományok lezárása, ez szövegfájlnál mindeképpen kötelező, bináris nál ajánlott, addig ugyanis üresnek mutatja az operációs rendszer a fájl méretét. Ezt a „Close()” metódussal tehetjük meg.

4. A feladat példakódja: frmLotto.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.IO;
using System.Windows.Forms;
namespace Fajlkezeles
{
    public partial class frmLotto : Form
    {
        public frmLotto()
        {
            InitializeComponent();
        }
        private void btnKilepes_Click(object sender, EventArgs e)
        {
            DialogResult valasz;
            valasz = MessageBox.Show("Valóban ki akar lépni?", "KILÉPÉS",
                     MessageBoxButtons.YesNo,
                MessageBoxIcon.Question, MessageBoxDefaultButton.Button2);
            if (valasz == DialogResult.Yes)
                Application.Exit();
        }
        protected override void OnClosing(CancelEventArgs e)
        {
            e.Cancel = true;
            base.OnClosing(e);
            btnKilepes_Click(btnKilepes, e);
        }
        void lottoszamok(int szam, int darab)
        {
            int veletlen;
            Random rnd = new Random();
            int i = 1;
            while (i <= darab)
            {
                veletlen = rnd.Next(1, szam + 1);
                if (!lstLista.Items.Contains(veletlen))
                {
                    lstLista.Items.Add(veletlen);
                    i++;
                }
            }
        }
        private void btnGeneral_Click(object sender, EventArgs e)
        {
            lstLista.Items.Clear();
            if (rgOtos.Checked)
                lottoszamok(90, 5);
            if (rgHatos.Checked)
                lottoszamok(45, 6);
            if (rgHetes.Checked)
```

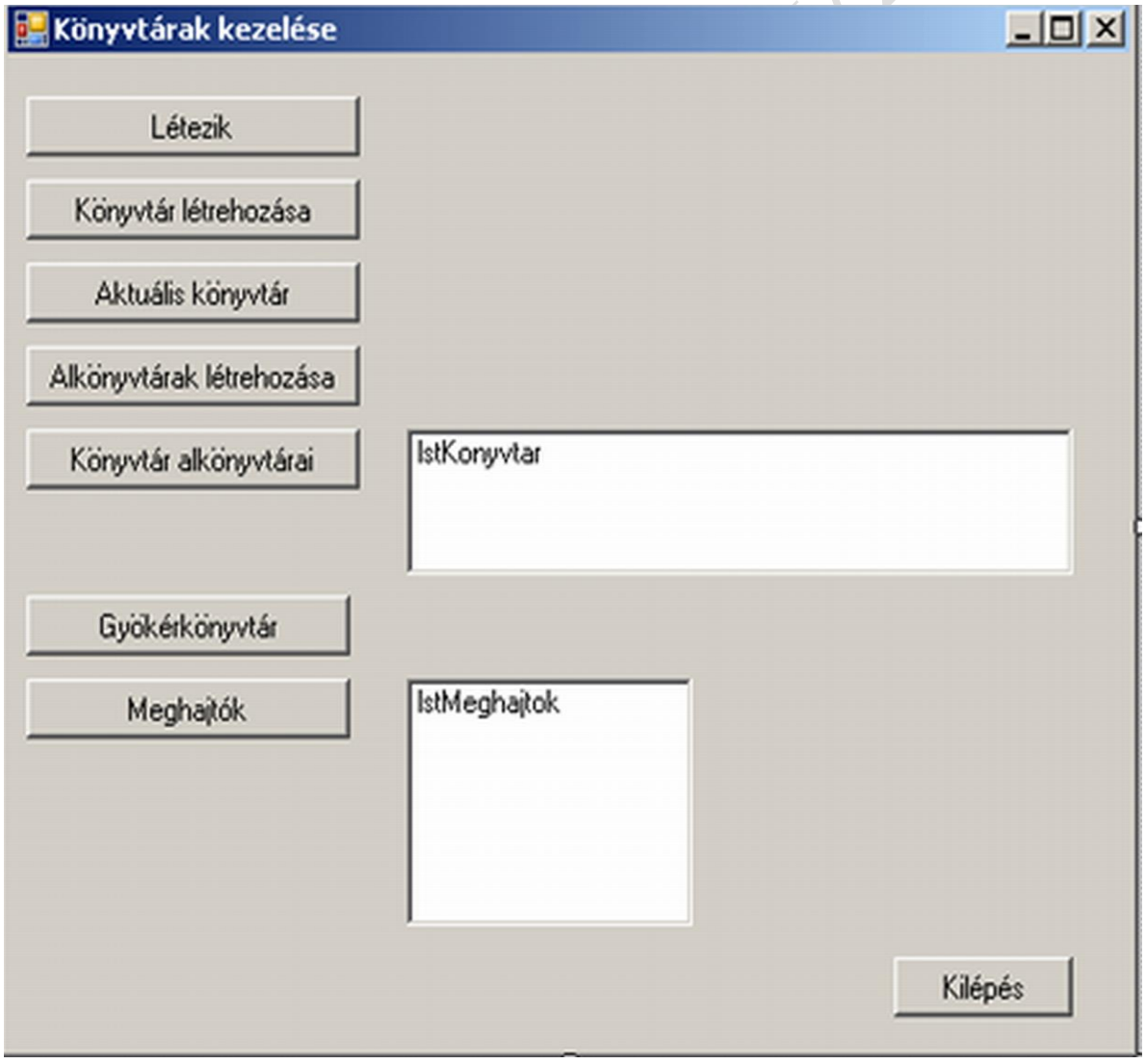
```
            lottoszamok(35, 7);
        }
        private void btnFajlbair_Click(object sender, EventArgs e)
        {
            saveFileDialog1.Title = "Lottószámok mentése";
            saveFileDialog1.InitialDirectory = "c:\\";
            saveFileDialog1.Filter = "Szövegfájlok (*.txt)|*.txt|Minden
                                      fájl(*.*)|*.*";
            if (saveFileDialog1.ShowDialog() == DialogResult.OK &&
                saveFileDialog1.FileName != "")
            {
                int i = 0;
                TextWriter tw = File.CreateText(saveFileDialog1.FileName);
                while (i < lstLista.Items.Count)
                {
                    lstLista.SelectedIndex = i;
                    tw.WriteLine(lstLista.SelectedItem);
                    i++;
                }
                tw.Close();
            }
        }
        private void btnFajbololvas_Click(object sender, EventArgs e)
        {
            lstLista.Items.Clear();
            openFileDialog1.Title = "Lottószámok beolvasása";
            openFileDialog1.InitialDirectory = "c:\\";
            openFileDialog1.Filter = "Szövegfájlok (*.txt)|*.txt|Minden
                                      fájl(*.*)|*.*";
            if (openFileDialog1.ShowDialog() == DialogResult.OK &&
                openFileDialog1.FileName != "")
            {
                TextReader tr = File.OpenText(openFileDialog1.FileName);
                string sor = "";
                while ((sor = tr.ReadLine()) != null)
                    lstLista.Items.Add(sor);
                tr.Close();
            }
        }
        private void btnBinarisbair_Click(object sender, EventArgs e)
        {
            saveFileDialog1.Title = "Lottószámok mentése";
            saveFileDialog1.InitialDirectory = "c:\\";
            saveFileDialog1.Filter = "Szövegfájlok (*.dat)|*.dat|Minden
                                      fájl(*.*)|*.*";
            if (saveFileDialog1.ShowDialog() == DialogResult.OK &&
                saveFileDialog1.FileName != "")
            {
                int i = 0;
                FileStream fs = new FileStream(saveFileDialog1.FileName,
                    FileMode.Create);
                BinaryWriter bw = new BinaryWriter(fs);
                while (i < lstLista.Items.Count)
                {
                    lstLista.SelectedIndex = i;
                    bw.Write((int)lstLista.SelectedItem);
                    i++;
                }
                bw.Close();
            }
        }
        private void btnBinarisbololvas_Click(object sender, EventArgs e)
```

```
        {
            lstLista.Items.Clear();
            openFileDialog1.Title = "Lottószámok beolvasása";
            openFileDialog1.InitialDirectory = "c:\\";
            openFileDialog1.Filter = "Szövegfájlok (*.dat)|*.dat|Minden
                                      fájl(*.*)|*.*";
            if (openFileDialog1.ShowDialog() == DialogResult.OK &&
                openFileDialog1.FileName != "")
            {
                FileStream fs = new FileStream(openFileDialog1.FileName,
                       FileMode.Open, FileAccess.Read);
                BinaryReader br = new BinaryReader(fs);
                while(br.PeekChar()!=-1)
                    lstLista.Items.Add(br.ReadInt32());
                br.Close();
            }
        }
    }

}
```

8.4 Példa: Könyvtár műveletek használata

Ebben a példában a könyvárakkal és a velük kapcsolatos műveleteket mutatjuk be. Megnézzük, hogyan tudunk könyvtárat illetve alkönyvtárat létrehozni. Természetesen ehhez ellenőrizni kell, hogy nincs-e már ilyen nevű könyvtár az adott környezetben. Megtanuljunk hogyan kérdezhető le az aktuális könyvár, ahol állunk illetve hol található a gyökérkönyvtár. Ki tudjuk listáztatni egy adott könyvtár alkönyvtárait. Végezetül pedig meg tudjuk nézni, hogy milyen meghajtók érhetők el az adott gépen, ez akkor különösen érdekes, ha hálózati meghajtók is rendelkezésre állnak, ráadásul azok már betűjelekhez vannak rendelve. A kilépés gombra kattintva az alkalmazásunk fejezze be a program futtatását! Nem kell visszakérdezni, hogy valóban ki akar-e lépni a programból!

1. Hozzunk létre új projektet Konyvtarak néven!
2. Tervezzük meg az űrlapot az alábbiak szerint:



Az űrlapon szereplő komponensek tulajdonságait állítsuk be az alábbiak szerint:

Komponens	Text	Name
Form	Könyvtárak kezelése	frmKonyvtar
ListBox		lstKonyvtar
ListBox		lstMeghajtok
Button	Létezik	btnLetezik
Button	Könyvtár létrehozása	btnKonyvtar
Button	Aktuális könyvtár	btnAktualis
Button	Alkönyvtárak létrehozása	btnAlkonyvtar
Button	Könyvtár alkönyvtárai	btnAllista
Button	Gyökérkönyvtár	btnGyoker
Button	Meghajtók	btnMeghajtok
Button	Kilépés	btnKilepes

Példánkban a könyvtárak kezelését mutatjuk be. A létezik gombra kattintva a felhasználó ellenőrizheti létezik-e már a ”proba” nevű könyvtár a ”c:\” meghajtón.

```
if (Directory.Exists("c:\\proba"))
```

A könyvtár létrehozása gombbal létre tudjuk hozni a tervezett könyvtárat.

```
Directory.CreateDirectory("c:\\proba");
```

A következő gomb az aktuális könyvtárat adja eredményül.

```
Directory. GetCurrentDirectory()
```

Az alkönyvtárak létrehozása gombbal az adott alkönyvtárba tudunk további könyvtárakat létrehozni. Ez tulajdonképpen megegyezik a könyvtár létrehozása paranccsal, csak az elérés abszolút útvonalát is meg kell hozzá adni hozzá.

A következő gombra kattintva a mellette lévő listaablakban az adott könyvtár („c:\proba”) alkönyvtárait tudjuk listáztatni egy „foreach” ciklus segítségével. Előtte a listaablak tartalmát ki kell üríteni.

```
foreach (string s in Directory.GetDirectories("c:\\proba"))
```

```
    lstKonyvtar.Items.Add(s);
```

A gyökérkönyvtár gombra kattintva az adott útvonal gyökérkönyvtárát adja vissza.

```
Directory.GetDirectoryRoot("c:\\proba\\aa\\11")
```

A meghajtók gombra kattintva a mellette lévő listaablakban a logikai meghajtókat listázza ki. Előtte a listaablak tartalmát itt is ki kell üríteni.

```
lstMeghajtok.Items.AddRange(Directory.GetLogicalDrives());
```

3. frmKonyvtar.cs:

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Globalization;
using System.IO;
namespace DirectoryTest
{
public class frmKonyvtar : System.Windows.Forms.Form
  {
  public frmKonyvtar()
  {
    InitializeComponent();
  }
  private void btnLetezik_Click(object sender, EventArgs e)
  {
   if (Directory.Exists("c:\\proba"))
      MessageBox.Show("c:\\proba könyvtár már létezik");
   else
      MessageBox.Show("c:\\proba könyvtár még nem létezik");
  }
  private void btnKonyvtar_Click(object sender, EventArgs e)
  {
     Directory.CreateDirectory("c:\\proba");
  }
  private void btnAktualis_Click(object sender, EventArgs e)
  {
     MessageBox.Show("Az aktuális könyvtár" +Directory.
GetCurrentDirectory());
  }
  private void btnAlkonyvtar_Click(object sender, EventArgs e)
  {
   Directory.CreateDirectory("c:\\proba\\a\\1\\2\\3");
   Directory.CreateDirectory("c:\\proba\\b");
   Directory.CreateDirectory("c:\\proba\\c");
   }
  private void btnAlLista_Click(object sender, EventArgs e)
  {
   lstKonyvtar.Items.Clear();
   foreach (string s in Directory.GetDirectories("c:\\proba"))
      lstKonyvtar.Items.Add(s);
  }
  private void btnGyoker_Click(object sender, EventArgs e)
  {
     MessageBox.Show(Directory.GetDirectoryRoot("c:\\proba\\aa\\11"));
```

```
  }

  private void btnMeghajto_Click(object sender, EventArgs e)
  {
    lstMeghajtok.Items.AddRange(Directory.GetLogicalDrives());
  }

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

9 Feladatok önálló gyakorlásra

1. Írjunk programot, mely a testsúly (kérje be a program) és a testmagasság (kérje be a program) alapján meghatározza a testtömegindexet (tti), és kiírja, hogy milyen testsúly osztályba tartozik az adott illető. a testtömeg osztályokat lásd lentebb.

$$Testtomegindex = Testtomeg[kg]/testmagassg^2[m^2]$$

   Ha a tti=0–15: Súlyos sovány ság
   Ha a tti=15,1–16: Mérsékelt sovány ság
   Ha a tti=16,1–18,5: Enyhe sovány ság
   Ha a tti=18,6–25: Normális testsúly
   Ha a tti=25,1–30: Túlsúlyos
   Ha a tti=30,1– :Elhízás

2. Írjunk programot, mely bekér egy számot, és eldönti, hogy osztható-e 3-mal, 4-gyel vagy 9-cel.
3. Írjunk programot, amely bekéri két pont koordinátáit, majd kiszámolja azok távolságát. A távolság értéket MessageBox-ban jelenítsük meg! A távolság a két pont közé eső szakasz hossza, melyet a pontok koordinátáiból könnyedén kiszámolhatunk. A gyökszámítást az alábbi függvény végzi: Math.Sqrt

$$\sqrt{(x1-x2)*(x1-x2)+(y2-y1)*(y2-y1)}$$

4. Írjon egy programot, ami leosztályoz egy maximálisan 100 pontos dolgozatot az 50, 65, 80, 90 ponthatárok szerint! A határérték a jobb jegyhez tartozik. Ha a pontszám negatív vagy száznál nagyobb, akkor a program írja ki, hogy hibás az adat! Switch szerkezetet használjon az elágazáshoz! A pontszámot a felhasználótól kérje be!
5. Készítsen egy alkalmazást, amely mezőgazdasági jóslást végez. A program kérje be az elvetett búza mennyiségét tonnában. Ez alapján számolja ki egy véletlenszerűen generált szorzóval (5–15) a várható hozamot, és írja ki a mennyiségét. A szorzó alapján elemezze és írja ki, hogy milyen év várható: átlag alatti (5–8), átlagos év (9–12), átlag feletti (13–15).
6. Készítsünk az egészség megőrzéséhez használható programot. A programunk kérje be a kilégzéskor keletkező CO_2 és O_2 mennyiségét! Számoljuk ki a respirációs kvócienst! *Magyarázat:* Az anyagcsere folyamán a keletkezett CO_2 és a felhasznált O_2 hányadosa, vagyis a légzési hányados. (RQ = kilégzett CO_2.Belégzett O_2 aránya). Az értékének a kiszámításához használhatjuk a következő képletet: RQ= $CO_{2/}$ $O_{2.}$ Az RQ akkor

megfelelő, ha értéke 0,8-as értéket mutat. Ha ennél kevesebb, akkor a szervezet a zsírokból nyeri az energiát. Ha ennél több, akkor a szénhidrátokból. Ezt írassuk is ki egy MessageBox segítségével!

7. Készítsünk programot, amely bekér két számot, majd a kettő közötti számtartományban kiír három darab véletlen számot egy MessageBox-ban.
8. Készítsünk programot, amely dinnyék csomagolásához végez számításokat. A dinnyéket szalaggal kell átkötni úgy, hogy kétszer körbe érje őket, és a masni készítéséhez számolunk még 60 cm-t. A program kérje be a dinnye átmérőjét (mindnél azonos), és a dinnyék számát! Számítsa ki, és írja ki egy MessageBox-ban, hogy a felhasználó által megadott számú dinnye csomagolásához hány méter szalagra van szükség.
9. Készítsünk programot, amely segíti a burkoló mesterek munkáját. A szükséges csempe mennyiségének a kiszámításához a program kérje be a terület szélességét, valamint a magasságát méterben, majd számolja ki, hogy 20cm*20 cm méretű csempék esetén hány darabra van szükség a munka elvégzéséhez (a plusz 10%-ot az illesztések miatt illik rászámolnunk).
10. Készítsünk egy alkalmazást, amely bekér két egész számot, majd eldönti, hogy melyik a nagyobb. A két számot int típusú változókban tároljuk el. Amennyiben a két megadott szám azonos értékű, a bekérést ismételjük meg. A megoldáshoz használjunk feltételes elágazást. Rossz adatok megadásakor az ismétlést folytassuk mindaddig, amíg helyes adatokat nem kapunk.
11. Kérjünk be a felhasználótól három egész számot, majd döntsük el, hogy melyik a legnagyobb, és a legkisebb érték és ezeket írassuk is ki a képernyőre!
12. Készítsünk egy programot, amely bekér három egész számot a felhasználótól. A bekért számokra úgy tekintünk, mint egy háromszög oldalaira. Döntsük el, hogy a háromszög szerkeszthető-e. A háromszög abban az esetben szerkeszthető, ha bármely két oldal hosszának az összege nagyobb a harmadik oldal hosszánál.
13. Készítsünk egy programot, amely bekér három egész számot a billentyűzetről. A bekért számokra úgy tekintünk, mint egy háromszög oldalaira. Számítsuk ki a háromszög területét. A terület kiszámításához használhatjuk a Hérón képletet.

$s = \frac{a+b+c}{2}$ $\mathcal{T} = \sqrt{(s(s-a)(s-b)(s-c))}$

14. Generáljunk tíz darab 1–6 közé eső véletlen számot. A program ezután mondja meg hányszor volt hatos a generált érték és írja ki egy MessageBox-ban!

15. Kérjünk be egy mondatot a felhasználótól, majd írjuk ki szóközök nélkül. A mondatot kezeljük karaktertömbként!
16. Készítsünk egy alkalmazást, amely paraméterként kap egy egész számot (int), majd kiírja a hét azonos sorszámú napját a képernyőre. Az 1-es érték jelenti a hétfőt, a 2-es a keddet, a 7-es a vasárnapot. Amennyiben a megadott szám nem esik az 1–7 intervallumba, a program írjon hibaüzenetet a képernyőre.
17. Készítsünk alkalmazást, amely beolvassa egy személy életkorát majd a kapott adattól függően kiírja egy MessageBox-ban azt a korosztályt, amibe az életkor tulajdonos tartozik. Gyermek (0–6), Iskolás (7–22), Felnőtt (22–64), 65-től nyugdíjas!
18. Írjunk olyan programot, amely véletlen, két számjegyű értékeket generál és generálás után azonnal kiírja a képernyőre egymás mellé, vesszővel elválasztva az elemeket!
19. Kérjünk be két egész számot a felhasználótól a 10–90-es intervallumból. Amennyiben a beírt számok ezen kívül eső egész számok lennének, úgy addig ismételjük a bekéréseket, amíg megfelelő értékeket nem kapunk. A két számot fogjuk fel, mint életkorok, egy apa és a fia életkorait. Adjuk meg, hány éves volt az apa, amikor a fia megszületett. Amennyiben az apa fiatalabb volt ekkor, mint 18, vagy idősebb, mint 50, akkor a program írja ki egy MessageBox-ban, hogy „Ez azért már nehezen hihető”.
20. Készítsünk a faktoriális kiszámítására alkalmas programot! A matematikában egy n nem negatív egész szám faktoriálisának az n-nél kisebb vagy egyenlő pozitív egész számok szorzatát nevezzük.

$$4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$$

21. Készítsen programot, amely bekér egy szöveget (szót) a felhasználótól és azt írja ki visszafelé!
22. Készítsük el a azt a programot, amely kiírja a képernyőre a Fibonacci számok közül az első 15-öt. A Fibonacci számok a matematikában az egyik legismertebb rekurzív sorozat elemei. Az első két elem a nulla és az egy, a további elemeket minden esetben az előző két szám összegéből képezzük.

    0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377

ISBN 978-963-318-651-0