A C# nyelv programozási alapjai

oktatási segédanyag, mely a

Társadalmi Megújulás Operatív Program
Határon átnyúló együttműködés a szakképzés és a felnőttképzés
területén c. pályázati felhívás keretében megvalósított

Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával című, ***TÁMOP-2.2.4-11/1-2012-0055*** kódszámú
projekt keretében valósult meg.

2013.

A projektek az Európai Unió támogatásával, az Európai
Szociális Alap társfinanszírozásával valósulnak meg.

Tartalomjegyzék

1. Bevezető 2
2. A .NET Framework alapjai 3
2.1 Előzmények 3
2.2 Jellemzők 4
2.3 Fordítás, futtatás 7
3. A Visual Studio rövid bemutatása....................10
3.2 Egy jól ismert újszülött: Helló Világ!....................13
3.3 Hasznos eszközök, kiegészítések 14
4. A C# nyelv fundamentumai....................17
4.1 Változók....................18
4.1. Referencia- és értéktípusok 19
4.2. Operátorok 24
4.3. Vezérlési szerkezetek....................28
4.4. Tömbök 30
4.5. Stringek....................31
5. Metódusok 34
5.1 A metódusok deklarálása és hívása 34
5.2 Paraméter átadási módok 35
6. Osztályok és objektumok 38
6.1 Alapelvek....................38
6.2 Láthatóság....................39
6.3 Egységbezárás (encapsulation), tulajdonságok (properties) ...41
6.4 Konstruktorok, példányosítás 42
6.5 Statikus módosító 44
6.4 Öröklődés 45
7. Struktúrák 47
8. Gyűjtemények, generikusok 49
8.1 ArrayList....................49
8.2 Sorok 50
8.3 Verem 51
8.4 Hashtable 51
8.5 SortedList....................53
8.6 Típusos lista....................53
8.7 Típusos sor 54
8.8 Típusos verem....................54
8.9 Dictionary....................54
8.10 Generikus metódusok....................55
9. Fájlkezelés....................57
9.1 Fájlrendszer és kezelése....................57
9.3 Szerializáció....................63
10. Kivételkezelés....................65

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

1. Bevezető

A .NET tervezése során az egyik elsődleges szempont a termelékenység volt. Az objektum-orientált szemlélet, az osztálykönyvtárak bősége, a platformsemlegességgel és egy hatékony fejlesztőeszközzel társítva olyan eszközt adhat a kezünkbe, amivel egyedül, vagy akár csapatban dolgozva is gyorsan tudunk látványos eredményeket elérni. A jegyzet az ismerkedést még csak most kezdőknek igyekszik útmutatást adni. Bemutatja a nyelv alapjául szolgáló .NET Framework alapjait, aminek a segítségével programunk különböző architektúrák között is hordozhatóvá válik. Tisztázza a nyelv alapjául szolgáló referencia- és értéktípusok közötti különbséget, a változók, operátorok és leggyakoribb adattípusok használatát. Az OO szemlélettel, az osztályok használatával most ismerkedőknek bemutatja a szemlélet alapjait, a példányosításnak, a metódusok készítésének legjellemzőbb módjait. Összeveti az osztályokat a nyelvben hasonlóan viselkedő, ám érték típusú struktúrákkal. A több más, korszerű nyelvben is meglévő generikusság jelentőségét külön fejezet tárgyalja. Adataink tárolásáról a fájlkezelést tárgyaló fejezet segítségével gondoskodhatunk. Az ehhez (és számos más) szituációhoz gyakran szükséges kivételkezelés zárja a bemutató anyagát. A tananyag példákkal igyekszik segíteni a befogadást, egyszerű, könnyen áttekinthető programokkal próbálja fogyaszthatóvá az elméletet.

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL

A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

2. A .NET Framework alapjai

A .NET Framework egy népszerű fejlesztői platform, amelynek segítségével *Windows, Windows Phone, Windows Server* vagy a felhő alapú *Windows Azure* által futtatható alkalmazások készíthetők. A keretrendszer lehetővé teszi olyan alkalmazások, vagy szolgáltatások futtatását, amelyek több programozási nyelven készülhetnek, platformsemlegesek lehetnek, illetve támogatják a gyors alkalmazásfejlesztést *(RAD – Rapid Application Development)*. A .NET tehát kiszolgálói oldalra, asztali felületre, mobil eszközökre, vagy akár (a .NET-re épülő XNA Framework segítségével például Xbox360) konzolra is enged fejleszteni alkalmazásokat.

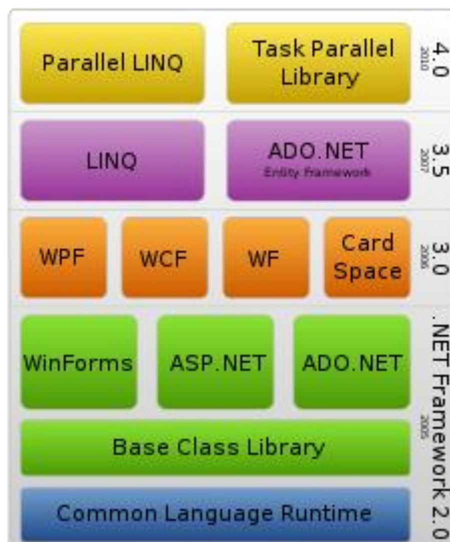
2.1 Előzmények

A fejlesztés előzményei az előző évezredig nyúlnak vissza. 1998-ban dőlt el a Microsoftnál, hogy nem használják tovább a Sun Java technológiáját. A cél egy olyan fejlesztői platform kidolgozása lett, ami felveszi a versenyt a Java-val. Ezért a meglévő Visual J++ termékük lett az alapja az új projektnek, amelynek a neve Next Generation Windows Services lett. Az első béta változat 2000. végén jelent meg .NET 1.0 beta 1 néven.

A szoftverplatform első végleges, hivatalos változata 2002-ben jelent meg. Az első jelentős változtatás a 2.0-s változatban szinte a komplett framework átírását okozta, az osztálykönyvtárak száma jelentősen bővült. Ez 2005-ben lett nyilvánosan is elérhető. A 2006-os, 3.0-s változat jelentősége többek között olyan újdonságokban rejlett, mint a szervízorientált architektúrát *(SOA)* megvalósító *WCF – Windows Communication Foundation*, vagy a felhasználói felületet jelentősen megújító *WPF – Windows Presentation Foundation. A* 2007. végén kiadott 3.5 változat egyik jellemző újdonsága a *LINQ* volt, amely a nyelvbe ágyazott lekérdezésekkel közelítette egymáshoz az objektum-orientált szemléletet és a relációs adatbázisokat, illetve az *XML*-t. A többprocesszoros gépek terjedésével megjelentek a 4.0-s változatban a párhuzamosítás eszközei. A 2012-ben debütált 4.5 számos újdonsága közül néhány: *Windows Store* alkalmazások fejlesztése Windows 8 alá, aszinkron programozási támogatás, hordozható osztálykönyvtárak támogatása.

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.



The .NET Framework Stack

1. ábra .NET Framework képességek a verziók függvényében[1]

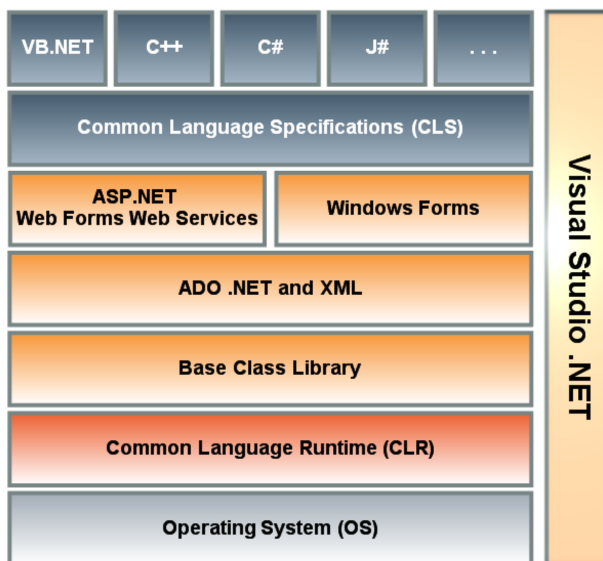
2.2 Jellemzők

A .NET Framework tulajdonképpen egy futtató környezet, ami az alkalmazások végrehajtását kezeli. Egyrészt tehát tartalmaz egy közös nyelvi futtató környezetet *(CLR – Common Language Runtime)*, amelyik a memóriakezelést és a rendszerszolgáltatások elérését biztosítja, másrészt pedig része egy olyan kiterjedt osztálykönyvtár *(Class Library)*, amely az alkalmazásfejlesztés minden fontos területén segíti a hatékony, biztonságos kódkészítést.

[1] A kép forrása: http://en.wikipedia.org/wiki/.NET_Framework

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.



2. ábra A klasszikus .NET Framework architektúra

Számtalan programozási nyelvet használhatunk a frameworkben. Tulajdonképpen bármelyik nyelvnek elkészíthető az implementációja .NET alá, mivel külső partnerek, gyártók is készíthetnek ilyet a keretrendszer számára.
Néhány az ismertebbek közül, amelyeknek már készült ilyen változata[2]:

- Visual C# .NET
- Visual Basic .NET
- Visual C++ .NET
- Python for Microsoft .NET
- Perl for Microsoft .NET
- Pascal for Microsoft .NET

A .NET által használt programozási nyelvek közös jellemzői:

- Objektum-orientáltság: azaz olyan objektumokkal képezzük le a megvalósítandó feladatot, amelyeknek vannak saját tulajdonságai, műveletei, egyfajta állapota.
- Eseményvezérlés: az objektumoknak lehetnek eseményeik és amikor ezek bekövetkeznek, kódot rendelhetünk hozzájuk.

[2] Forrás: http://msdn.microsoft.com/en-us/library/aa292164(v=vs.71).aspx

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

CLS

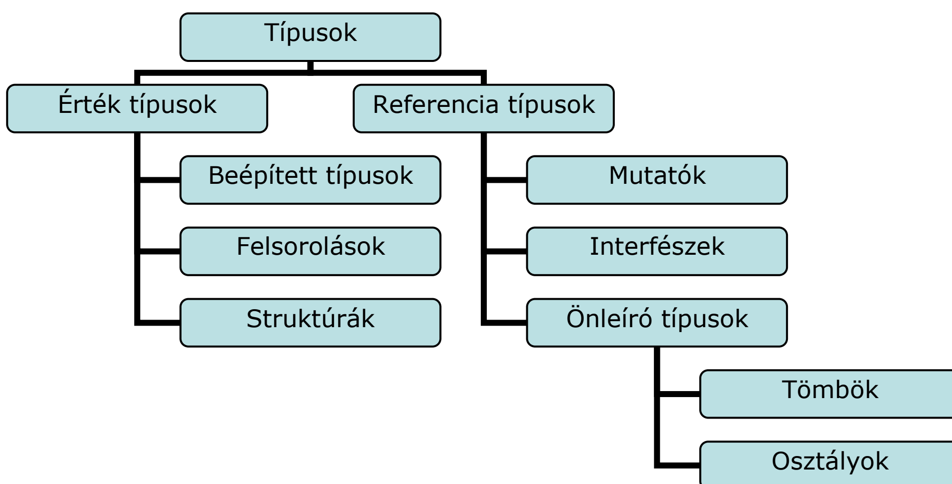
A *CLS (Common Language Specification)* a programozási nyelvekkel kapcsolatos elvárásokat tartalmazza, olyan szabvány, ami megadja a szolgáltatások formátumát, leírja a memóriakezelés módját, a függvényhívás, a kivételkezelés, stb. mikéntjét.

Memóriakezelés

Egy klasszikus programozási nyelv esetében a programozó feladata az általa készített objektumok számára a memóriában történő helyfoglalás, valamint annak biztosítása, hogy ezen területek használat utáni felszabadítása megtörténjen. A CLR ezt automatikusan biztosítja.

Nyelvfüggetlen típusok

A hagyományos nyelvek esetében a programozó által használt alaptípusokat a fordító kezeli, ezzel megnehezítve a program esetleges átültetését másik nyelvre. Ehhez képest a *Common Type System (CTS)* biztosítja, hogy az alapvető típusok .NET Frameworkben legyenek elkészítve, ezáltal valamennyi ezt használó nyelvben közösek legyenek.



3. ábra Az alapvető típusok osztályozása

Platformsemlegesség

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

A hordozható osztálykönyvtárak *(Portable Class Library)* segítségével Visual Studio 2012-ben készíthetünk olyan cross-platform alkalmazásokat is, amelyek módosítás nélkül is futtathatók több platformon, így például Windows 7, Windows 8, Silverlight, Windows Phone, vagy Xbox360 alatt[3].

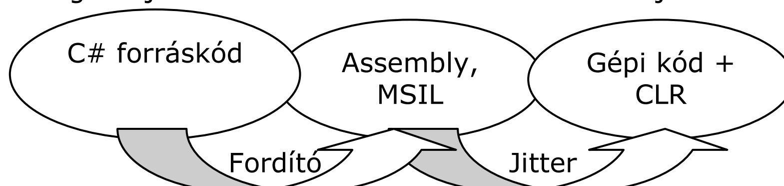
Kompatibilitás

Egy korábbi változatban megírt alkalmazás jellemzően változtatás nélkül képes futni az újabb .NET Frameworkben is.
Egyazon számítógépen azonban a keretrendszer több változata is telepítve lehet egyidejűleg. Ennek köszönhetően minden alkalmazás azon a keretrendszeren működhet, amelynek a verziójára eredetileg is készült.

2.3 Fordítás, futtatás

Az „ortodox” programozási nyelveken megírt programok úgynevezett natív kódra fordítódnak. Ez azt jelenti, hogy a futtatást végző számítógép processzora által közvetlenül is végrehajtható utasításokat eredményez. A .NET esetében a fordítás eredménye egy menedzselt kód. Ez egy különleges nyelvet eredményez, aminek a futtatása nem közvetlenül történik, hanem egy köztes eszköz segítségével.
Natív kódra fordít például a C++ nyelv, menedzselt kódot eredményez a JAVA (byte kód), illetve a .NET is.
A natív kódokat általában hardver-közelibbnek tekinthetjük, ami azt is eredményezheti, hogy a menedzselt kódhoz képest nagyobb teljesítményű. A menedzselt kód cserébe többféle hardveren futhat, illetve biztonságosabb, kevesebb hibalehetőséget nyújtó programokat eredményezhet.
A fordítás és futtatás menete:

- Elkészítjük a forráskódot, például C# nyelven.
- A fordító segítségével (csc.exe) elkészül egy köztes kód *(Microsoft Intermediate Language – MSIL).*
- Mivel a köztes kód nem közvetlenül futtatható, ezért az alkalmazás első futtatása alkalmával a CLR egy úgynevezett futásidejű fordító *(Just In Time – JIT, vagy Jitter)* segítségével a végrehajtandó részeket natív utasítássá alakítja.



[3] Forrás: http://msdn.microsoft.com/en-us/library/gg597391.aspx

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

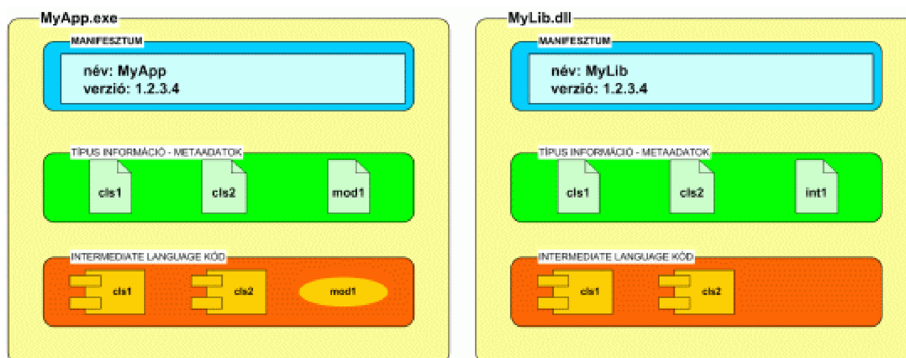
Assembly

A programunk első fordítása alkalmával egy *assembly* (szerelvény) keletkezik, ami tartalmazza egyrészt magát a kódot köztes formában, másrészt pedig úgynevezett meta adatok is helyet foglalnak benne.

Az assembly részei:

- Köztes kód (MSIL)
- Manifest
- Meta adatok

A *manifest* írja le magát az assemblyt: név, verziószám, függőségek más *assemblyk*től, biztonsági előírások, stb. Ezeket az információkat a CLR használja fel a működéséhez.
A meta adatok a futtató környezet számára tartalmaznak információkat. Itt találhatók az alkalmazásban használt típusok, osztályok, interfészek, felsorolások, struktúrák, stb.



4. ábra Az assembly belső szerkezete[4]

Egy assembly egy, vagy több fájlból állhat, jellemzően .exe, vagy .dll kiterjesztéssel.

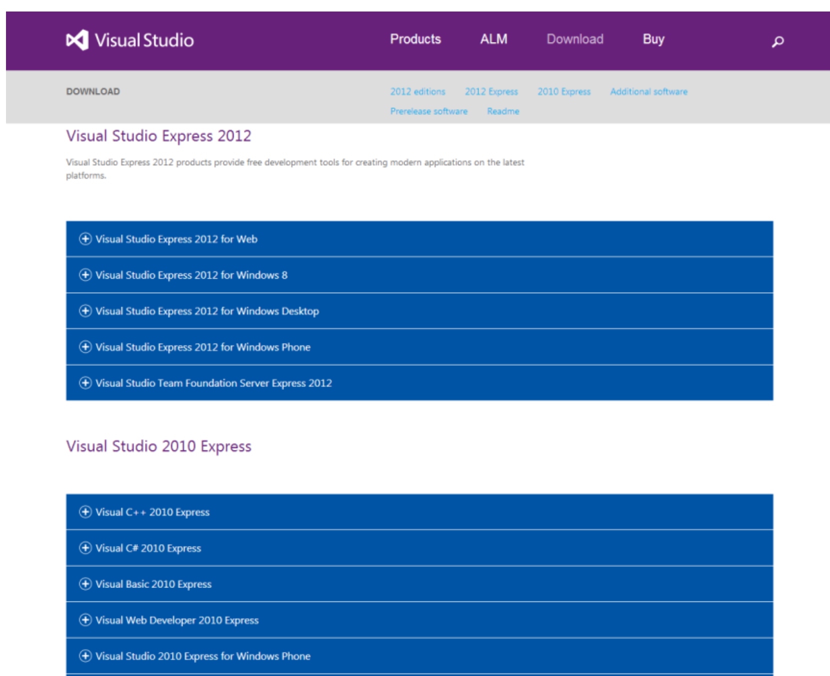
Nem csupán fejlesztők kerülhetnek kapcsolatba a frameworkkel. Mivel számos alkalmazás igényli a futásához, ezért a legtöbb modern Windows rendszer már feltelepítve, transzparens módon tartalmazza a keretrendszer valamely változatát. Természetesen egyes program telepítésekor is jelezhetnek, hogy ilyen eszközre van szükségük. Ilyenkor jellemzően a webről, néhány dialógusablakon keresztül megtörténhet a keretrendszer telepítése, a korábbi változatok megőrzése mellett is.

[4] Forrás: http://prog.hu/cikkek/860/Bevezetes+a+dot-net-be/oldal/6.html

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

Ahhoz, hogy elkezdhesd a fejlesztést, elsőnek töltsd le a legfrissebb .NET keretrendszert! Ez jelenleg a 4.5-ös verzió. Ehhez a változathoz legalább Windows Vista operációs rendszer szükséges SP2 szervízcsomaggal, vagy újabb változat. Ezt követően töltsd le a Visual Studio integrált fejlesztőeszközt, majd kezdjük el a munkát! A Visual Studio 2012-es változataihoz legalább Windows 7, vagy újabb operációs rendszer szükséges A program ugyan feltelepül Windows Vistára, de sajnos nem fog indulni!



5. ábra Visual Studio Express változatok[5]

[5] A programok a http://www.microsoft.com/visualstudio/eng/downloads oldalról tölthetőek le.

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

3. A Visual Studio rövid bemutatása

A framework alá programot fejleszteni már parancssorból is lehetséges. A korábban említett *csc.exe* segítségével akár Jegyzettömbből is dolgozhatnánk, de a *Visual Studio* integrált fejlesztőeszközének segítségével lényegesen hatékonyabban és gyorsabban kódolhatunk.
A programnak több különböző változata ismert. A tanulási célra ingyenes Express 2012-es verziói elsősorban a célplatform alapján szelektáltak:

- Visual Studio Express 2012 for Web,
- Visual Studio Express 2012 for Windows 8,
- Visual Studio Express 2012 for Windows Desktop,
- Visual Studio Express 2012 for Windows Phone,
- valamint Visual Studio Team Foundation Server Express 2012 verziók állnak rendelkezésünkre.

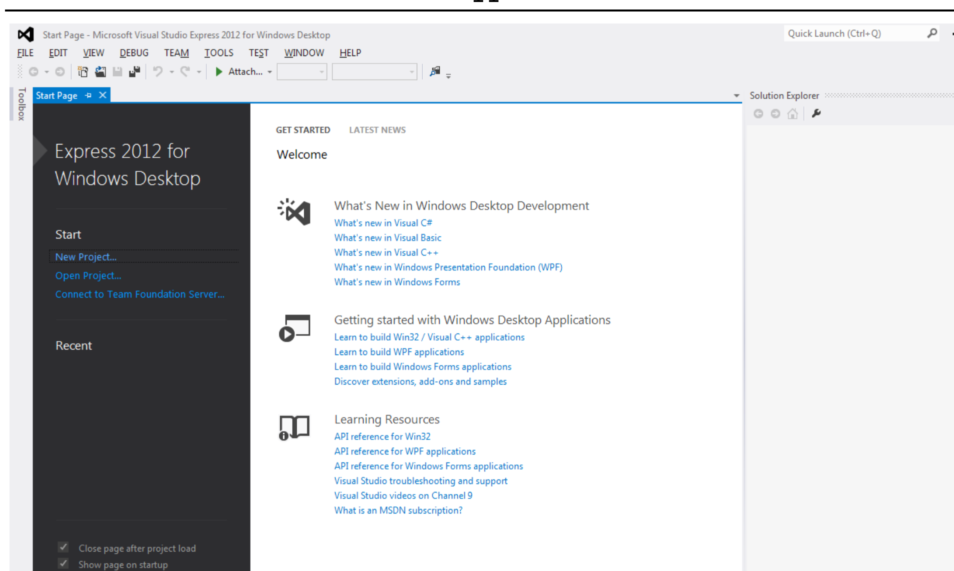
A korábbi, 2010-es Express esetében a választott nyelv is szempont lehetett a megfelelő verzió kiválasztásához. A fizetős változatokkal jelen jegyzetben nem foglalkozunk, valamennyi általunk tárgyalt ismeret elérhető a fentiek segítségével is.

3.1 A Visual Studio kezelőfelülete

A jegyzetben a *Visual Studio Express 2012 for Windows Desktop* változaton keresztül fogunk ismerkedni a C# nyelv alapjaival.

A telepítés követően az első indításkor a Windows platformon megszokott elemek fogadják az ismerkedőt: a felső sorban menüpontok, jobb oldalt az ablakban gyorskereső, a menük alatt testre szabható toolbar (eszköztár) várja az újdonsült programozó-palántát. A toolbar és az eszközöket biztosító ablakok helye megváltoztatható, vagy átméretezhető, esetleg rögzíthető is.
Első indításkor a Start page ablaka fogad minket:

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.



Új alkalmazás készítéséhez válasszuk ki a *New Project…* pontot (vagy válasszuk menüből a *FILE/New Project*-et)!

A *New Project* ablak baloldalán feltelepített, vagy online elérhető sablonok közül választhatjuk ki alkalmazásunk nyelvét és típusát.
Jelöljük be az *Installed / Templates / Visual C#* szakaszt!
Ehhez középen a következő projekttípusokat (sablonok, templatek) ajánlja a fejlesztőeszköz:

- *Windows Forms Application*: klasszikus Windows alkalmazások számára
- *WPF Application*: új, látványos grafikus felületű programok létrehozásához
- *Console Application*: parancssorban futtatható alkalmazásokhoz
- *Class Library*: osztálykönyvtárak számára készült sablon (.dll kiterjesztéssel).

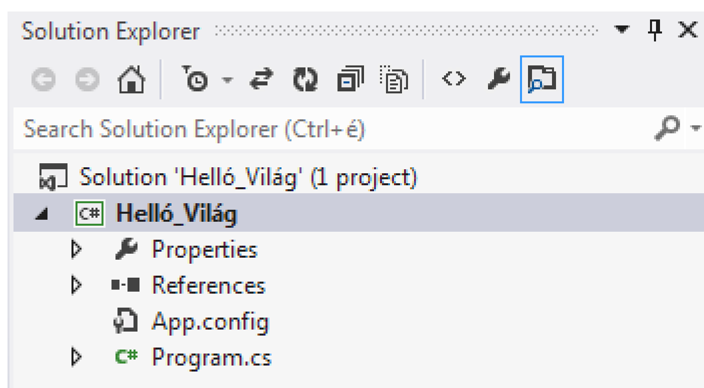
Mivel a nyelv elemeinek a megismeréséhez kezdetben elegendő, ezért válasszuk most a *Console Application* típust, az első létrehozandó programunk számára!

Alul a *Name* szakaszon adjuk első programunknak a Helló_Világ nevet!
A *Location* segítségével megadható a projektet tartalmazó mappa. Ez alapesetben a *Dokumentumok\Visual Studio 2012\Projects* mappába hoz létre alkönyvtárakat.

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

A Studio a készítendő program névtere számára a *Solution*-ban is meghatározhat egy nevet. Egy Solution több projektet is tartalmazhat szükség szerint. Ez alapesetben megegyezik a projektünk nevével.
A jóváhagyást követően elkészül a sablon alapján az üres alkalmazás váza.
Vegyük szemügyre a frissen elkészült programvázat!
Középen a szerkesztőablak található, amely több fájlt is kezel, ezek fülekről érhetők el. A jelenleg elérhető egyetlen megnyitott állomány itt a Program.cs nevet viseli. Ez lesz első alkalmazásaink helye.
Jobboldalt, alapesetben fent egy *Solution Explorer* nevű ablak látszik:

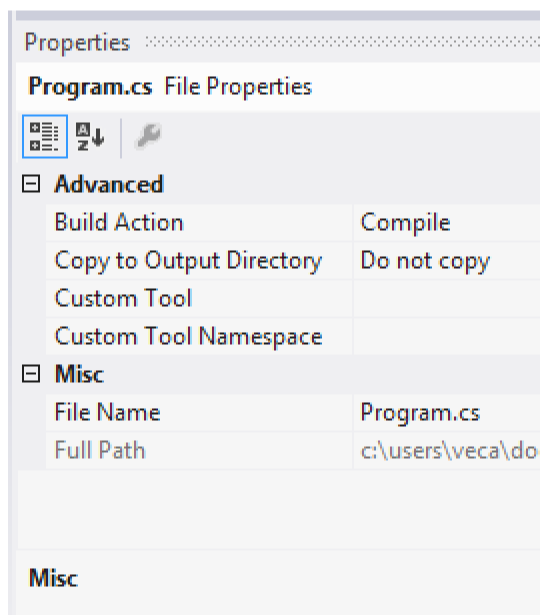


6. ábra Solution Explorer

Ez tartalmazza a Solution-ünkben található projektek (jelenleg csak egy van) elemeit. Itt jobb egér gomb segítségével új mappákat, projekteket, osztályokat adhatunk az eddigiekhez.

A fejlesztőeszköz jobb alsó sarkában (kezdetben) a kiválasztott elem tulajdonságait írja le a Properties ablak:

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.



7. ábra Properties ablak

Ez a kiválasztott elem tulajdonságait írja le csoportokba szedve, vagy ábécé sorrendben. Grafikus alkalmazások vezérlőit kiválasztva a rajtuk értelmezhető események (és a rájuk feliratkozott metódusok) is kiválaszthatók, megváltoztathatók az ablak segítségével.

Baloldalt, egyelőre összecsukva a *Toolbox* ablak pihen. Innen válogathatunk, ha grafikus alkalmazásainkhoz tervezési időben szeretnénk vezérlőket beszerezni. Jelenleg üres, mivel parancssorban ilyenek nem várhatók.
Grafikus alkalmazásoknál itt egy másik, *Data Sources* ablak is megjelenhet, amely az alkalmazásunk számára szükségessé váló helyi, vagy távoli adatkapcsolatok bekötését végezheti.

3.2 Egy jól ismert újszülött: Helló Világ!

Első alkalmazásunk elkészítése előtt még vegyük részletesen szemügyre a Program.cs fájlunk tartalmát!
A *using* kezdetű sorok olyan direktívákat tartalmaznak, amelyek néhány ismert, gyakran használt névtér tartalmát teszik rövidebben elérhetővé.
A *namespace* bekezdés egy alapértelmezett, saját névteret hoz létre alkalmazásunk számára. A névtérbe az összetartozó objektumokat gyűjtjük, gyakorlatilag egy kódszervezési technika. Több, akár egymásba ágyazott *(nested)* névteret is tartalmazhat az alkalmazásunk.

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.
MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

Mivel objektum-orientált nyelvet tisztelhetünk a C#-ban, ezért a névtérben egy osztály foglal helyet *Program* néven, aminek egy *Main* nevű metódusa is elkészült. Ezekről később, az *„Osztályok és objektumok”* című fejezetben még szót ejtünk.
Első klasszikus, „Helló Világ” szöveget kiíró programunk elkészítéséhez írjuk be a Main metódusba a következőket:

```
Console.WriteLine("Helló Világ!");
Console.ReadKey();
```

Az első utasítás valósítja meg a konzolon kiíratást, a második egy billentyű lenyomásáig felfüggeszti a program befejezését.

A program elindítását többféle módon kezdeményezhetjük. Az F5 lenyomásával úgynevezett *Debug* módban indíthatjuk a programunkat el. Ilyenkor az alkalmazás olyan kiegészítő információkkal fordul le, amelyek megkönnyítik a hibakeresést. Ezzel egyenértékű a *toolbar*-on a Start gombra kattintás, ha mellette a *Solution Configuration „Debug”* módban áll. Debug nélkül is elindítható az alkalmazás, az úgynevezett *Release* módban, vagy a *Ctrl-F5* megnyomásával.
Az eredmény az első C# programunk lesz, amelyet .exe kiterjesztéssel a projekt mappájában is megtalálhatunk a *bin* könyvtárban. A *Debug* és a *Release* változatok külön mappákban foglalnak helyet a fordítások után.

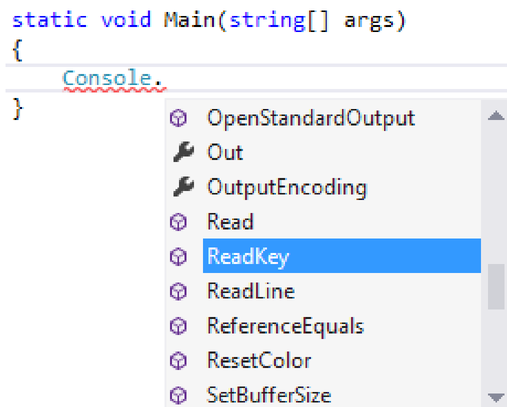
3.3 Hasznos eszközök, kiegészítések

A Visual Studio számos olyan hasznos segítséget nyújt a fejlesztőknek, ami érdemben gyorsíthat a munkán.

IntelliSense

Ilyen eszköz például az *IntelliSense*, amely a nyelv elemeit könnyen elérhetővé teszi beírás közben, legyenek azok osztályok, metódusok, tulajdonságok, vagy saját objektumaink.

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.
MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.



8. ábra IntelliSense munka közben

A kiválasztott metódus esetében egy rövid leírást, magyarázatot is kaphatunk annak használatáról.

Code Snippet

A Code Snippetek olyan kicsi, de gyakran használt kódrészletek, amelyeket általában billentyűkombinációval gyorsan előhívhatunk. Az előző kódban szereplő Console.WriteLine() utasítást mennyivel gyorsabb például beírni a következő módon:

```
cw +TAB+TAB (Két TAB egymás után)
```

Több ötletet is nyerhetünk a *Tools/Code Snippet Manager* menüpontban, ahol nyelvenként vannak csoportosítva a *snippet*ek.

Névterek beszúrása, felesleges névterek eltávolítása

Az első programunk számos olyan névteret hivatkozik, amire nincs feltétlen szükségünk:

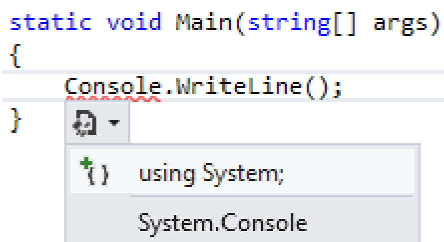
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

A nem használt névterek eltávolíthatók a kódból jobb egérgomb segítségével: a felugró gyorsmenüből válasszuk az *Organize Usings / Remove Unused Usings* menüpontokat!

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

Ezután a kódunkban csak a *using System;* hivatkozás marad meg, mivel ez a névtér tartalmazza a *Console* osztályt, amelynek a metódusát a kiíráshoz felhasználtuk!
Amennyiben viszont ezt is törölnénk (próbáljuk ki!), a program nem fog lefordulni, hiszen nem állapítható meg, hogy a *Console* osztály melyik névtérhez tartozik. Ez sem jelent gondot: az osztály nevére kattintva megkeresi az őt magába foglaló névteret, majd felajánlja a következő lehetőségeket:



9. ábra Kiválasztott elem névtérhez kötése

Az első esetben *(using System;)* visszaírja a használt névterek közé a program a System-et. Használjuk ki ezt akkor, ha többször fogunk hivatkozni a benne lévő elemekre, osztályokra!
A *System.Console* választása esetén csak a választott helyen egészíti ki a kódot így:

```
System.Console.WriteLine()
```

Ez akkor hasznos opció, ha csak alkalomszerűen van szükségünk az osztályra.

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

4. A C# nyelv fundamentumai

A nyelv alapvetően a C szintaktikai jellemzőit viseli magán: a nyelv *case-sensitive*, tehát megkülönböztetjük egymástól a kis- és nagybetűket a forráskódban. Egyrészt ha az általunk létrehozott szerkezeteket nem erre tekintettel hívjuk meg, akkor hibát kapunk:

```
string név=”Kovács Tamás”;
Console.WriteLine(Név);
```

Másrészt az utasítások sem fognak lefordulni, ha az írásmódjukban eltérünk:

```
console.WriteLine(név);
```

Ez szintén fordítási hibát fog eredményezni.

A blokkok elejét és végét itt is a kapcsos zárójelek *{ }* jelölik ki.
Mint látni fogjuk, a változódeklaráció és az alapvető vezérlési szerkezetek használata is hasonlóságot mutat.
Egysoros megjegyzések készíthetők a // használatával. Többsoros megjegyzésekhez a /* */ szerkezet is bevethető. A Ctrl-E, C gombokkal *(Comment out the selected lines)* az egérrel kijelölt forráskód gyorsan átalakítható megjegyzéssé, ennek ellenkezője a Ctrl-E, U kombináció *(Uncomment the selected lines)*, amelyekkel könnyen tudjuk ki-bekapcsolni a programunk kijelölt részét.

A fordítási hibák elkerülése érdekében, valamint az egységes, jól olvasható kód készítésének érdekében érdemes áttekinteni az azonosítókra vonatkozó névadási konvenciókat a kis-nagybetűk tekintetében.[6]

Az összetett szavak esetében *Pascal Case* írásmódnak nevezzük a szóhatárokon nagy kezdőbetűs jelölést. A Camel Case ettől annyiban tér el, hogy az első szó minden betűje kicsivel írandó.

Azonosító típusa	**Írásmód**	**Példa**
osztály	Pascal Case	Console
felsorolás típusa	Pascal Case	ErrorLevel
felsorolás értékei	Pascal Case	FatalError
esemény	Pascal Case	ValueChanged
kivétel osztály	Pascal Case	WebException
interfész	Pascal Case, I-vel kezd	IDisposable

[6] Forrás: http://msdn.microsoft.com/en-us/library/vstudio/ms229043(v=vs.100).aspx

metódus	Pascal Case	ToString
névtér	Pascal Case	System.IO
paraméter	Camel Case	typeName
tulajdonság	Pascal Case	BackColor

A Java nyelvvel szemben tehát eltérés például a metódusok írásmódja, a nyelvet ismerők ügyeljenek erre!
A saját objektumok elnevezése jellemzően Camel Case-ben történik, a Microsoft alkalmazásfejlesztői részlege a Simonyi Károly által bevezetett úgynevezett *Hungarian notation* rendszert használta, illetve használja.[7]

4.1 Változók

A programjaink egyik leggyakrabban alkalmazott elemei a változók, amelyek segítségével nevesítve tudunk értékeket tárolni és azokkal műveleteket végezni a számítógép memóriájában. Használatához általában meg kell adnunk a típusát és a változó nevét.
A következő táblázat azokat a beépített típusokat tartalmazza, amiket C# nyelven használhatunk:

C# típus	**.NET típus**	**Méret (byte)**	**Leírás**
byte	System.Byte	1	Előjel nélküli 8 bites egész szám (0..255)
char	System.Char	2	Egy Unicode karakter
bool	System.Boolean	1	Logikai típus, értéke igaz(1), vagy hamis(0)
sbyte	System.SByte	1	Előjeles 8 bites egész szám (-128..127)
short	System.Int16	2	Előjeles 16 bites egész szám (-32768..32767
ushort	System.UInt16	2	Előjel nélküli 16 bites egész szám (0..65535)
int	System.Int32	4	Előjeles 32 bites egész szám (–2147483647.. 2147483647).
uint	System.UInt32	4	Előjel nélüli 32 bites egész szám (0..4294967295)
float	System.Single	4	Egyszeres pontosságú lebegőpontos szám
double	System.Double	8	Kétszeres pontosság lebegőpontos szám
decimal	System.Decimal	8	Fix pontosságú 29 jegyű szám
long	System.Int64	8	Előjeles 64 bites egész szám
ulong	System.UInt64	8	Előjel nélküli 64 bites egész szám
string	System.String	-	Unicode karakterek sorozata
object	System.Object	-	Minden típus őse

[7] http://en.wikipedia.org/wiki/Hungarian_notation

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.
MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

A táblázat második oszlopa a .NET-es megfelelője a típusoknak, a forráskódban ezt is használhatjuk típusmegjelölésnek.
Mivel a C# erősen típusos nyelv, ezért a létrehozott változó típusának ismertnek kell lennie fordításkor.
A változók neve betűvel, vagy aláhúzás jellel (_) kezdődhetnek és folytatódhat számokkal is. A nyelvben használhatunk magyar ékezetes elnevezéseket is.

Deklaráció

A deklaráláskor megadjuk az adott változónak (illetve objektumnak, lásd később) a típusát és a nevét, amivel hivatkozunk rá:

```
int életkor;
```

Definíció

Definiáláskor a már deklarált változónknak értéket adunk. Természetesen történhet egyszerre a deklarációval is.

```
életkor = 18;
char betű = ’A’;
```

Hatókör

A változókra hivatkozni azon a blokkon belül lehetséges, amelyiken belül létrehozták azt. Másképp fogalmazva a változó lokális lesz a blokkjára nézve. Hagyományos programozási nyelvekben megismert globális változókat ne keressünk.

4.1. Referencia- és értéktípusok

A *Common Type System* által használt típusok alapvetően kétféle csoportba oszthatók: referencia és értéktípusokra bonthatjuk őket. Ezek vizsgálatához elsőnek nézzük meg, milyen memóriaszerkezeteket használnak programjaink.
A verem *(stack)* olyan memóriaterület, amely LIFO (last-in-first-out) elven működik, tehát azon adatokhoz férünk hozzá elsőnek, amit legutolsónak tettünk bele. Kifejezések kiértékeléséhez oly módon használható, hogy az elvégzendő műveletek operandusait a verembe helyezve a szükség műveletek elvégzése után az eredmény is visszakerül a *stack*-be.
Heap-nek, azaz halomnak nevezik a program futtatásához lefoglalt, általa szabadon felhasználható memóriaterületet.

Az érték- és referenciatípusok között az egyik fő különbség az, hogy amíg az értéktípusra hivatkozunk, elsősorban azok értékéről beszélünk. Referenciatípus használatánál azonban - mivel ezek általában összetett szerkezetek - általában nem írhatók le egyetlen értékkel az adatok (illetve műveletek), ezért egy ilyen, referencia típusú változó azt a memóriaterületet jelöli, ahol azok elhelyezkednek. Ennek köszönhetően referencia típus esetében másoláskor a hivatkozás másolódik, ezért előfordulhat az, hogy ugyanazon szerkezetre két, vagy több változó is mutat, s az érték megváltozása mindegyikre hatással van.
Az érték típus másolásakor az érték egy új helyre másolódik, ezért annak változása nem érinti a korábbi értékét.

A következő program egy érték típusú struktúrát és egy referencia típusú osztályt hoz létre, majd ezeket egy-egy metódus módosítja. Kérdés, hogy a módosítás, s így a program futtatásának eredménye mi lesz?

```
struct Struktúra
 {
     public int adat;
 }

 class Osztály
 {
     public int adat;
 }

 class Főprogram
 {
     public static void StruktúraMódosítás(Struktúra s)
     {
         s.adat = 2;
     }
     public static void OsztályMódosítás(Osztály o)
     {
         o.adat = 2;
     }
     public static void Main()
     {
         Struktúra struktúra = new Struktúra();
         Osztály osztály = new Osztály();
         struktúra.adat = 1;
         osztály.adat = 1;
         StruktúraMódosítás(struktúra);
         OsztályMódosítás(osztály);
```

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

```
            Console.WriteLine("struktúra.x = " +
struktúra.adat);
            Console.WriteLine("osztály.x = " + osztály.adat);
            Console.ReadKey();
        }
    }
```

A program futtatásának eredményeként a struktúra adata nem változott, mivel a másolat változtatása nem hatott ki az eredetire. Az osztály objektumának viszont megváltoztatta az értékét a módosítás.

Referencia típusok:

- *String*
- *Osztály*
- *Interfész*
- *Delegált*

Ezek a típusok minden esetben referenciaként tárolódnak.

Érték típusok:

- Beépített típusok
  - *sbyte*
  - *byte*
  - *short*
  - *int*
  - *uint*
  - *long*
  - *float*
  - *double*
  - *decimal*
  - *char*
  - *bool*
  - *date*, stb.
- *Felsorolás*
- *Struktúra*

Az érték típusok tárolása akkor történik a veremben, ha azok lokálisak, egy metóduson belül. Ha egy referenciatípus, mint amilyen egy osztály belsejében adattagként foglalnak helyet, úgy a halomban (*heap*) lesznek tárolva.

Akár érték-, akár referenciatípusról van szó, minden típus a *System.Object*-ből származik. Ennek eredményeképp több metódusuk és tulajdonságuk hívható, vagy használható fel:

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL

A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

- *GetType()* : a változó típusát adja vissza eredményül *(System.Type)*
- *ToString()* : a teljes minősített típusnevet adja vissza, felhasználható a standard kimeneten a változó leírására is.
- *Equals(Object)* : egyenlőségvizsgálat másik objektummal

A létrehozott változók érték-, vagy referencia jellege lekérdezhető a *GetType().IsValueType* tulajdonság segítségével, amely *true* esetén érték, *false* visszaadásakor azonban referencia lesz.

Null értékű változók

A referencia típusú változók értéke értékadás előtt *null*. Arra is van lehetőség, hogy ezt mi magunk állítsuk be, ha jelölni szeretnénk, hogy a változó beállítatlan, vagy például kivételkezelés előtt.
Az érték típusok esetében ez nem járható út, mindenképpen értéket kell adnunk nekik használat előtt.
A .NET 2.0-s változatától kezdődően azonban az érték típusú változó helyett használható egy speciális struktúra, a *Sytem.Nullable* is. Egy nullázható típus a szokásos intervallum értékein túl egy plusz értéket, a *null*-t is tartalmazhatja.

Deklarálása: Nullable<típus> változónév, vagy <típus>? változónév
A változók értékadása a szokásos módon történhet, vagy adható *null* érték is. A változó értékének lekérdezése a változó nevével, vagy a *Value* tulajdonságával történhet. Azt, hogy van-e értéke, a *HasValue* tulajdonság segítségével kérdezhető le.

```
Nullable<int> x = null;
//A Value tulajdonság csak olvasható!
//x.Value = 20;
x = 20;
if (x.HasValue) Console.WriteLine(x.Value);
else Console.WriteLine("Nincs értéke a változónak!");
```

A *GetValueOrDefault()* metódus segítségével a nullázható változók értékét úgy adhatjuk tovább, hogy *null* értékük esetében a típusra jellemző alapértelmezett érték másolódik.
Ez numerikus típusoknál 0, karakternél a 0 értékű bájt, logikai típusnál pedig a *false*, azaz hamis érték.

```
bool? x = null;
bool xMásolat = x.GetValueOrDefault();
//xMásolat értéke: False
```

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.
MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

```
Console.WriteLine(xMásolat);
int? y = null;
int yMásolat = y.GetValueOrDefault();
//yMásolat értéke: 0
Console.WriteLine(yMásolat);
char? z = null;
char zMásolat = z.GetValueOrDefault();
//zMásolat értéke: '\0'
Console.WriteLine(zMásolat);
```

Boxing és unboxing

Az érték típusokat jellemzően kis méretű adatoknál használjuk, mivel kevesebb helyet foglalhatnak és kevesebb adminisztrációval jár a használatuk, mint referencia típusú megfelelőjük.
Azonban, mint szó volt róla, minden típus a *System.Object*-ből származik. Ez teszi lehetővé azt, hogy az érték típus része lehessen a közös típusrendszernek, valamint a korábban felsorolt metódusokat, tulajdonságokat használjuk rajtuk. *Boxing*-nak (bedobozolás) nevezzük azt a jelenséget, ami lehetővé teszi, hogy tetszőleges érték típust objektumként használjunk. A *boxing* során a kiválasztott érték típusú változónak a *heap*-ben lefoglalnak egy olyan objektumot, ami az eredeti érték típusának megfelelő, majd belemásolódik az eredeti érték. A használata hasznos lehet az egységes típusrendszeren túl bizonyos gyűjtemények, például *ArrayList* használata esetén is. (A gyűjteményekről később szót ejtünk.)

```
int életkor = 23;
//Bedobozolás
object oÉletkor = életkor;
//Összehasonlíthatóak?
if (életkor.Equals(oÉletkor)) Console.WriteLine("Egyenlőek");
//Ki is íratható
Console.WriteLine(oÉletkor);
```

A kidobozolás típuskényszerítéssel történhet *(cast)*, de természetesen csak olyan típussal érdemes próbálkozni, ami kompatibilis az eredetivel.

```
int évjárat = 2013;
//Bedobozolás
object oÉvjárat = évjárat;
//Kidobozolás
int másolat = (int)oÉvjárat;
//int->double konverzió megengedett
double másolat2 = (int)oÉvjárat;
```

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

```
double fok = 12.3;
object oFok = fok;
//De a double->int konverzió System.InvalidCastException kivételt dob
int másolat3 = (int)oFok;
```

Mivel a bedobozolás egy új memóriaterületre történik, ezért az új objektum változtatása nem hat ki a régi érték változónkra.

4.2. Operátorok

A programjainkban használt utasítások többnyire különböző típusú kifejezéseket kiértékelve hajtódnak végre. Ezen kifejezések egy, vagy több operandusból és az őket összekapcsoló operátorokból állnak. Ebben a fejezetben a legfontosabb operátorokat fogjuk áttekinteni.

Értékadó operátorok

A legtipikusabb műveletek egyike. Az operátor jele az egyenlőségjel. Tőle balra a változó, jobbra a számára adandó kifejezést kell megadnunk. A kifejezésnek a típusa meg kell, hogy egyezzen a változó típusával, vagy felülről kompatibilisnek kell lennie vele (implicit konverzió).

```
int x = 10;
double y = x;
```

Típuseltérés esetében az egyik lehetséges út a *cast*olás (típuskényszerítés).

```
double pi = 3.141592654;
//Castolás adatvesztéssel
int p = (int)pi;
```

További lehetőségeink szöveg átalakítása esetén lehet a parse-olás.

```
string szöveg = Console.ReadLine();
//Konverzió parse-olással
int életkor = int.Parse(szöveg);
```

A Convert osztály metódusait pedig általánosabban is használhatjuk.

```
int rendben = 1;
//Konverzió osztályfüggvény segítségével
bool ok = Convert.ToBoolean(rendben);
```

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

Az értékadó utasítások rövidebb formában is használhatók.
A matematikai operátorok esetében rövidebb és gyorsabb eredményt kaphatunk ezek használatával.

```
int a = 12;
// a = a + 3;
a += 3;
```

Az inkrementáló és dekrementáló operátorok használhatók prefix és (kicsit bonyolultabban, de ugyanazon eredményt szolgáltató) postfix formában is.

```
int számláló = 5;
//Növelés prefix módon
++számláló;
//Növelés postfix alakban
számláló++;
//Csökkentés prefix módon
--számláló;
//Csökkentés postfix alakban
számláló--;
```

Feltételes és logikai operátorok

A feltételes ÉS operátor jele a &&.

```
int x = 12;
if (x<100 && x>=10) Console.WriteLine("x kétszámjegyű szám");
```

A feltételes VAGY operátort a || jelzi.

```
int x = 12;
if (x>99 || x<10) Console.WriteLine("x kívül esik a [10-99]
tartományon");
```

A tagadást is a C szintaktikájú nyelvekben megszokott módon, a felkiáltójel (!) használatával kell megoldani.

```
int a = 2000;
if (!(a>2000)) Console.WriteLine("'a' nem nagyobb, mint 2000");
```

A logikai operátorok a feltételesek rövidebb párja: a logikai ÉS operátort az & , a logika vagy operátort pedig a | biztosítja. A logikai kizáró vagy (XOR) jele a ^. Mivel ez utóbbi operátorok bitszintűek *(bitwise)*, ezért az úgynevezett rövidzár-kiértékelés nem használható

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

rajtuk, tehát minden esetben kiértékelődik mindkét oldala a feltételnek.
Mivel bitszintűek, ezért alkalmasak bitenkénti műveletek elvégzésére is.

Az úgynevezett feltételes operátor három operandussal rendelkezik.

Szintaktikája:
feltétel : igaz esetben lefutó utasítás : hamis esetben lefutó utasítás

Relációs és típustesztelő operátorok

Idetartoznak az ismert numerikus összehasonlító operátorok: >, <, >=, <=, az egyenlőséget a ==, ennek hiányát a != biztosítja.

Az *is* operátor segítségével egy objektum futásidejű típusának kompatibilitása egy adott típussal megállapítható.

```
int a = 1;
//'a' kompatibilis az object-tel
if (a is object) Console.WriteLine("'a' kompatibilis");
```

Az *as* operátor egy könnyed módja a *cast*-olásnak: a segítségével a kompatibilis, vagy nullázható típusok konvertálhatók.

```
string állat = "Kecske";
object oÁllat = állat;
//object felhasználása string-ként
string sÁllat = állat as string;
//object felhasználása StringBuilder-ként (lásd később)
StringBuilder sbÁllat = állat as StringBuilder;
```

Biteltoló operátorok

Az egész típusokkal (változók, vagy konstansok) végezhető biteltolások segítségével az egész érték bináris alakja csúsztatható valamelyik irányba el. A balra eltolás a <<, a jobbra csúsztatás a >> operátorok segítségével végezhetők. Az eltolás mértéke az operátor jobb oldali operandusa lesz.

```
//2 a 0. hatványon (1) 2 bittel eltolva 2 a 2. hatványon, azaz 4
Console.WriteLine(1 << 2);
int x = 11;
```


MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

```
//11-ben a 2 megvan ötször (1-el jobbra tolva egész osztás 2-vel)
Console.WriteLine(x>>1);
```

Matematikai operátorok

A matematikai operátorok alapműveletek elvégzésre teszik alkalmassá programjainkat.
Ilyen műveletek lehetnek:

- Összeadás (+)
- Kivonás (-)
- Szorzás (*)
- Egész- és valós osztás (/)
- Maradék (%)

```
int a = 5;
//Egész kifejezések között egész osztás lesz értelmezve.
Console.WriteLine("5 / 2 = " + a/2);
//Ha legalább az egyik operandus nem egész, valós osztást kapunk.
Console.WriteLine("5 / 2.0 = " + a / 2.0);
```

Precedencia

Az operátorok kiértékelési sorrendjét nemcsak felsorolásuk rendje, hanem azok fontossága is befolyásolja. Az a + b * c kifejezés kiértékelésénél például elsőként a szorzás lesz elvégezve, csak ezt követi az összeadás.
Ennek alapján a következő precedencia sorrend állítható fel a használható operátoraink között:[8]

- A legelsőként kiértékelve a ( ) zárójelek lesznek, de idetartoznak a következők is: x.y f(x) a[x] x++ x-- new typeof checked unchecked
- Unáris operátorok (előjelek) ! ~ ++x --x
- * / % (szorzás, osztás, maradék)
- + - (összeadás, kivonás)
- << >> (biteltolás)
- > < >= <= as is
- == !=
- & (logikai ÉS)
- ^ (logikai kizáró vagy)

[8] Forrás: MSDN Operator precedence and associativity (http://msdn.microsoft.com/en-us/library/aa691323(v=vs.71).aspx)

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.
MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

- | (logikai vagy)
- && (feltételes ÉS)
- || (feltételes VAGY)
- ?: (feltételes operátor)
- = *= /= += -= <<= >>= &= ^= |=

4.3. Vezérlési szerkezetek

A strukturált programozás értelmében a program által megvalósítandó feladatokat kisebb részekre bontjuk. Ezen feladatok utasításai meghatározott sorrendben hajtódnak végre.
Azon utasításokat nevezzük vezérlési szerkezeteknek, amelyek a program utasításainak végrehajtását, azok sorrendjét befolyásolják.

Szekvencia

Az adott sorrendben egymást követő utasítások egy szekvenciát alkotnak. Ez a legegyszerűbb szerkezet tehát nem más, mint egy lépéssorozat.

Feltételes elágazások, szelekciók

Az utasítás, vagy utasítások végrehajtását egy logikai kifejezés értékétől tesszük függővé. Előírhatunk utasítást, vagy utasításokat arra nézve is, ha az általunk támasztott feltétel hamis.
Az *if – else* szerkezet, vagy többirányú elágazás esetében a *switch – case* szerkezet írja le. Természetesen a feltételes elágazások egymásba is ágyazhatóak. Eltérés a C++-hoz képest, hogy a *switch – case* szerkezet esetében a *break* hiányában sem hajtjuk végre az alatta lévő elágazás utasításait (kivéve, ha az adott ágon nincs is utasítás).

```
Console.WriteLine("Add meg a jelenlegi hőmérsékletet");
int hőmérséklet;
hőmérséklet = int.Parse(Console.ReadLine());
if (hőmérséklet == 20) Console.WriteLine("Szobahőmérséklet");
else if (hőmérséklet > 20) Console.WriteLine("Meleg van");
else Console.WriteLine("Hideg van");
```

A fenti programrészlet megkérdezi az aktuális hőfokot, majd ennek ismeretében ír ki valamilyen reakciót rá.

```
Console.WriteLine("Hányas jegyet kaptál?");
int jegy = int.Parse(Console.ReadLine());
switch (jegy)
```

```
{
case 1: Console.WriteLine("elégtelen"); break;
case 2: Console.WriteLine("elégséges"); break;
case 3: Console.WriteLine("közepes"); break;
case 4: Console.WriteLine("jó"); break;
case 5: Console.WriteLine("jeles"); break;
default:  Console.WriteLine("Sajnos  ezt  a  jegyet  nem  ismerem");
break;
}
```

Az előző részlet pedig bekéri az érdemjegyet, majd szövegesen értékeli azt a *switch – case* szerkezet segítségével.

Ciklusok, iterációk

Az ismétlődő tevékenységek leírását alapvetően négyféle módon tehetjük meg.

Elöltesztelő ciklus

A while kulcsszó segítségével írhatjuk le. A kulcsszó után megadott feltétel esetében végrehajtja a mögött lévő utasítást. Kezdettől hamis feltétel esetében hozzá sem kezd a végrehajtáshoz, hanem azonnal az ezt követő utasításokra kerül a sor.

```
int i = 1;
// A ciklus kiírja a 2 első néhány hatványát.
while (i <= 256)
{
	Console.WriteLine(i);
	i *= 2;
}
```

Hátultesztelő ciklus

A do – while szerkezet hasonló az előzőhöz, de mivel az ismétlés feltételét fogalmazza meg, legalább egyszer biztosan lefut a ciklus magja.

```
char betű;
//A begépelt betűket az első csillag karakterig kiírja
do
{
	betű = (char) Console.Read();
	Console.WriteLine(betű);
} while (betű != '*');
```

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

Előírt lépésszámú ciklus

A for ciklus az egyik legnépszerűbb ciklusféleség. A ciklus addig hajtja végre az ismétlendő utasítás(oka)t, ameddig egy feltétel igaz marad. Jellemző felhasználása az ismert, vagy kiszámítható ismétlésszámú feladatok esetében.

```
int összeg = 0;
//Az első száz szám összege
for (int i = 1; i <= 100; i++)
      összeg += i;
//A cikluson kívül az 'i' változó nem hivatkozható
Console.WriteLine(összeg);
```

A ciklusváltozó deklarálható a ciklus fejrészében is, de ebben az esetben a cikluson kívül nem hivatkozható.

Foreach ciklus

A *foreach* segítségével tömbök, gyűjtemények kényelmes bejárása valósítható meg, mivel nem szükséges számon tartani az aktuálisan érintett elem sorszámát, vagy az elemek darabszámát. Paraméterei a bejárandó gyűjtemény egy elemtípusa és maga a gyűjtemény.

```
string gyümölcsös = "Almafa, körtefa";
//Kiíratjuk a string betűit szóközökkel ritkítva.
foreach (char betű in gyümölcsös)
      Console.Write(betű+" ");
```

A példa a később részletesebben tárgyalt szöveges *string* típus karaktereit járja be. Az egyes elemek típusa *char* lesz.

4.4. Tömbök

A tömb segítségével egyazon típusú adatból többet is képesek vagyunk együtt kezelni.

Deklarálása:
típus[] tömb_neve

Az elemek elérése egydimenziós tömb esetén egyetlen értékkel, a tömbindex megadásával történhet. Az indexelés 0-tól kezdődik. Készíthetők többdimenziós, vagy hiányos tömbök is.

```
//Egydimenziós tömb lefoglalása, az elemek értéke nulla.
```

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

```
int[] ötösLottó = new int[5];
for (int i=0; i< ötösLottó.Length; i++)
      Console.Write(ötösLottó[i]+" ");

Console.WriteLine();
//Új egydimenziós tömb, előre feltöltve.
int[] hatosLottó = { 2, 5, 12, 22, 25, 29 };
for (int i = 0; i < hatosLottó.Length; i++)
      Console.Write(hatosLottó[i] + " ");

//Kétdimenziós tömb lefoglalása, egyedi értékadás.
string[,] nevek = new string[2, 20];
nevek[0, 0] = "Kovács";
nevek[1, 0] = "Tamás";
```

A tömb hosszát a *Length* tulajdonsággal kérdezhetjük le.
A dimenziószám a *Rank* segítségével adható vissza. 1-es eredmény esetében egy, 2-nél kétdimenziós tömbről van szó, és így tovább.
Rendezhető elemek esetében a *Sort()* metódussal alapesetben növekvő rendbe terelhetők a tömb elemei.
Ezeken kívül is még számos metódus, tulajdonság segítheti a tömbbel való munkánkat, amelyek keresnek a tömbben, átméretezik, vagy éppen törlik azt.

4.5. Stringek

A *string* olyan adatszerkezet, amely szövegek tárolására alkalmas, Unicode kódolással. A [] operátor segítségével hivatkozhatóak tömbszerűen a benne lévő karakterek.

```
//Deklarálás definíció nélkül
string szöveg1;
//Üres szöveg definiálása
string szöveg2 = String.Empty;
//Értékadás
string szöveg3 = "Üdvözlünk!";
//Konstans változó
const string szöveg4 = "Ne változtasd meg!";
//Hiba:
//szöveg4 += "!";
Console.WriteLine(szöveg4);
//Hivatkozás egy elemére tömbszerűen
Console.WriteLine("Első betűje:"+szöveg4[0]);
//Hossza:
Console.WriteLine("A szöveg hossza:"+szöveg4.Length);
```

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

Noha referencia típusú, mégis könnyen összehasonlítható konstans szövegekkel, vagy más *string* típusú változókkal, mivel az == és a != operátorok az értékükre és nem a címükre hivatkozik.
A + operátorral összefűzhetők szövegek.

```
string név1 = "Zsuzsa";
string név2 = "Zsuzsanna";
if (név1 == név2) Console.WriteLine("A két név teljesen egyforma");
else Console.WriteLine("A két név különbözik");
//név1 = "Zsuzsa"+"nna";
név1 += "nna";
```

Escape karaktereket tartalmazhat a szöveg. A @ operátor segítségével azonban előírható, hogy az ezt követő szövegben nincs ilyen karakter. Ez később, a fájlkezelésnél leegyszerűsítheti az útvonalak megadását.

```
string útvonal1 = "C:\\saját mappa\\saját fájl.txt";
string útvonal2 = @"C:\saját mappa\saját fájl.txt";
//A két útvonal ugyanazt a tartalmat hordozza.
if (útvonal1 == útvonal2) Console.WriteLine("A két útvonal megegyezik");
```

A *string* típusú objektum nem megváltoztatható *(immutable)*. A *string* létrehozását követően már nem módosítható. Minden művelet, amely látszólag ezt megteszi, valójában új *string* objektumot készít ilyenkor. A következő példában a szövegek konkatenálásánál (összefűzésénél) új objektum születik, amely a változóhoz rendelődik. A korábbi objektumot a *CLR Garbage Collector*nak nevezett szemétgyűjtő rutinja fogja kitakarítani a tárból.

```
string állatkert = "kutya ";
állatkert += "macska ";
állatkert += "elefánt ";
állatkert += "oroszlán";
Console.WriteLine(állatkert);
```

4.6 StringBuilder osztály

A *StringBuilder* osztály segítségével kifejezetten változtatható szövegeket készíthetünk.

```
//Üres név létrehozása
StringBuilder üresnév = new StringBuilder();
```

```
//Alapesetben 16 karakteres a kapacitása
Console.WriteLine(üresnév.Capacity);

//A névnek kezdőérték és kapacitás is meg lesz adva.
StringBuilder név = new StringBuilder("Kovács János", 25);
//Hozzáfűzünk a névhez, közben kiíratjuk a kapacitást.
//Ha elértük a kapacitás végét, az megduplázódik.
for (int i = 0; i < 50; i++)
{
név.Append(".");
Console.WriteLine(név.Capacity);
}
//Beszúrás a 0. pozícióba
név.Insert(0, "Dr. ");
Console.WriteLine(név);
```

Alapesetben a szöveg kapacitása 16 karakter lehet, ami bővíthető, vagy automatikusan duplázódik, ha elérte a határát.
A változtatáshoz számos metódust használhatunk:

- *Insert()*: beszúrás adott pozícióba
- *Append()*: hozzáfűzés a végéhez
- *Remove()*: adott pozíciótól adott számú karakter eltávolítása
- *Replace()*: szövegrészlet cseréje
- *Clear()*: tartalom törlése

Hasznos tulajdonságok:

- *Length*: hossz
- *Capacity*: jelenlegi kapacitás
- *MaxCapacity*: maximális kapacitás

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

5. Metódusok

A metódusok olyan nevesített kódrészek, amelyek egy általunk megadott utasítássorozatot hajthatnak végre.
Minden metódus egy osztály, vagy egy struktúra részeként (lásd később) születhet és adhat vissza értékeket is a futtatása eredményeként. A paramétereket zárójelben, egymástól vesszővel elválasztva adhatjuk meg.
Minden C# alkalmazás tartalmaz (legalább egy) *Main* metódust, ami a programunk belépési pontját határozza meg. A programunk indítása esetén az i lévő utasítások fognak végrehajtódni.

A metódus nevének megadásakor hagyományosan szavanként nagy kezdőbetűt használunk, már az első szótól kezdődően.

5.1 A metódusok deklarálása és hívása

```
módosítók visszaadott_érték_típusa metódus_neve(paraméterek)
{
      utasítások;
}
```

Amennyiben a metódus nem ad vissza értéket, úgy a visszatérés típusául *void*-ot kell megadnunk. Amennyiben van tényleges visszaadott érték, a metódus törzsében azt *return* kulcsszó után kell megadnunk.

A metódusok hívása visszatérésük típusától többféleképp történhet. Void metódusok esetében hívhatjuk utasításszerűen, a nevükkel és szükség esetén konkrét paraméterek megadásával. Tényleges értékeket visszaadó metódus hívásánál azonban érdemes kifejezésben felhasználni őket, hogy az eredményt legyen, ami feldolgozza.

```
//static = osztálymetódus, az osztályhoz tartozik
//int = egész értéket ad vissza eredményül

static int AzÉletÉrtelme()
{
return 42;
}

public static void Main()
{
//metódus hívása

Console.WriteLine(AzÉletÉrtelme());
```

```
Console.ReadKey();
}
```

A fenti metódus működéséhez nem igényel paramétert, egész értéket ad vissza, ezért hívása is praktikusan kifejezésben történhet.

```
static void Diszítősor(char betű, int hányszor)
{
	for (int i = 0; i < hányszor; i++)
	{
	   Console.Write(betű);
	}
}

public static void Main()
{
	//metódus hívása
	Diszítősor('*', 40);
	Console.ReadKey();
}
```

Ez a metódus két bemenő paramétert igényel a futásához, viszont nem ad vissza eredményt (nincs *return* sem), így utasításszerűen hívtuk meg.

Több azonos nevű metódust is készíthetünk egyazon osztályban, vagy struktúrán belül is, ha azok paramétereik számában, vagy típusában eltérnek egymástól. A paraméterek alkotják a névvel együtt a metódus úgynevezett *szignatúráját*.
Ilyen esetekben híváskor az a metódus kapja meg a vezérlést, amelyik a híváshoz használt paraméterekkel számban és típusban is pontosan megegyezik.

5.2 Paraméter átadási módok

Az érték típusú paraméterek átadásánál alapesetben egy másolat készül az eredeti változóból a metódus számára.

```
static void Életkorom(int év)
{
	if (év > 18) év = 18;
	Console.WriteLine("A metóduson belül az életkorom:"+ év);
}

public static void Main()
{
```

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

```
        int életkor = 21;
        Console.WriteLine("Életkorom a metódus előtt:"+életkor);
Életkorom(életkor);
        Console.WriteLine("Életkorom a metódus után:"+életkor);
        Console.ReadKey();
}
```

A fenti metódus hiába igyekszik változtatni az életkorunkat 18 felett, annak érvényessége a metóduson belül marad.

A paraméter azonban nemcsak érték, de referencia alapján is odaadható. Ehhez mind a szignatúrában, mind híváskor jelezzük az adott paraméter előtti *ref* kulcsszóval ezt a tényt!

```
static void Életkorom(ref int év)
{
        //Mivel az év referencia alapján kap értéket,
        //ha megváltoztatjuk, az kihat a hívó félre is.
        if (év > 18) év = 18;
        Console.WriteLine("A metóduson belül az életkorom:"+ év);
}

public static void Main()
{
        int életkor = 21;
        Console.WriteLine("Életkorom a metódus előtt:"+életkor);
        Életkorom(ref életkor);
        //Így könnyű 18-nak lenni :)
        Console.WriteLine("Életkorom a metódus után:"+életkor);
        Console.ReadKey();
}
```

Természetesen ilyenkor már nem hívhatjuk a metódust konstans számmal, hiszen annak nincs referenciája sem.

Előfordulhat az is, hogy a metódus eredményeként nem egyetlen, de több értéket is szeretnénk visszaadni. Ezt már most is megoldhatnánk a *ref* használatával, de van erre egy praktikusabb megoldás, az *out* kulcsszó. Az *out* bár hasonlóan használható, mint a *ref*, de nem igényli, hogy a kapott paraméterének már legyen definiált értéke.

```
static void Eredmény(out int x)
{
        x = 12;
}
public static void Main()
{
```

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

```
        //a változónak nem adunk értéket
        int a;
        //Metódushívás
        Eredmény(out a);
        Console.WriteLine("'a' értéke a metódus hívása után: "+a);
        Console.ReadKey();
}
```

Ügyeljünk rá, hogy az int típust váró metódusnak nem adhatunk *Nullable* típust *(int? a)*, mivel az nem egyezik a várt *int* típussal!
Az *out*-tal tehát inicializálatlan eredményváltozóknak adhatunk értéket. Ha a változóknak volt korábbi értéke, természetesen a metódus azokat felülírja.

Abban az esetben, ha a metódus számos bemenő adatot kíván a működéséhez, célszerű lehet azokat úgynevezett *paramétertömb*ben átadni a metódusnak. Ilyenkor a tömb típusával megegyező elemeket vár a metódus a tömb formájában. Amennyiben viszont kihasználjuk, hogy valamennyi típus közös őse a *System.Object,* ezt használva tömbtípusnak, bármilyen adatot átadhatunk a metódusunknak. Erre mutat példát a következő programrészlet.

```
static void ParaméterTömbFüggvény(params object[] tömb)
{
        foreach (object egy_elem in tömb)
        {
          Console.Write("Az elem típusa:" + egy_elem.GetType());
          Console.WriteLine(", értéke:"+ egy_elem.ToString());
        }
}
public static void Main()
{
        object[]  vegyesTömb  =  {"szöveg",  1,  2,  3,  'A',  new
SystemException()};
        ParaméterTömbFüggvény(vegyesTömb);
        Console.ReadKey();
}
```

A metódusokhoz köthető egyéb módosítókat (statikusság, láthatóság, stb.) az osztályok áttekintésénél tárgyaljuk.

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

6. Osztályok és objektumok

6.1 Alapelvek

Az objektum-orientált programozás (OOP) fő alapelve, hogy ne válasszuk el az adatokat az őket felhasználó, működtető kódoktól, hanem kezeljük őket egy egységben.
A feladatunkban egységesen viselkedő, egyetlen modellel leírható entitást (az adatok és a rajtuk végezhető műveletekkel) **osztály**nak nevezzük.
Az osztály **tulajdonság**ait, állapotát általában adatszerkezetekkel biztosítjuk, a rajtuk végezhető műveletet pedig jellemzően függvényekkel, **metódus**okkal biztosítjuk.

Deklarálása:

```
attribútumok módosítók class OsztályNeve {
 OsztályTörzse;
}
```

Példa

Tegyük fel, hogy programot szeretnénk írni az autónk legfontosabb adatainak tárolására, üzemeltetésének leírására!
Ebben az esetben az autó számára létrehozhatunk egy Autó osztályt (használjuk a tanult névkonvenciót: az osztályok neve kezdődjön nagybetűvel). Az autónkat elsősorban utazásra szeretnénk használni, ezért fontos számunkra, hogy azonosítható legyen, nyilvántartsuk a fogyasztását és azt, hogy mennyi üzemanyag van jelenleg is benne.

Az Autó osztály tulajdonságai ekkor lehetnek a következők:

- Rendszám (szöveges adat)
- Átlagfogyasztás: l/100 km (numerikus információ)
- Üzemanyag tartály mérete: l (numerikus adat)
- Aktuális üzemanyag mennyiség (numerikus információ)

Az autónk működtetése során üzemanyagot éget el. Ha tehát használjuk, csökken a benne lévő üzemanyag mennyiség. Készíthetünk rá olyan metódust, ami ezt megteszi, tehát csökkenti a mennyiséget a megtett távolság és a fogyasztás arányában. A metódusunk bemenő paramétere lehet a megtett távolság. Eredménye lehet a metódusnak egy logikai eredmény, ami jelzi, hogy az adott távolságot sikerült-e megtenni (mert például volt elegendő üzemanyag a tartályban).
Ha fogytán az üzemanyagunk, tankolhatunk bele egy adott mennyiséget, figyelembe véve az üzemanyag tartály méretét is. Ez is

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

adhat vissza eredményt a siker tényéről egy logikai érték segítségével.

Az Autó osztály metódusai, amelyek műveleteket végeznek tehát:

- bool Utazás(int távolság)
- bool Tankolás(int mennyiség)

Ezt megvalósíthatja a következő kódrészlet:

```
class Autó
    {
        //Tulajdonságok (adattagok)
        string rendszám;
        double átlagFogyasztás;
        int üzemanyagTartály;
        double üzemanyagMennyiség;

        //Metódusok
        bool Utazás(int távolság)
        {
            double              üzemanyagSzükséglet             =
átlagFogyasztás/100*távolság;
            if (üzemanyagSzükséglet <= üzemanyagMennyiség)
            {
                üzemanyagMennyiség -= üzemanyagSzükséglet;
                return true;
            }
            else return false;
        }

        bool Tankolás(int tankoltMennyiség)
        {
            if   (üzemanyagMennyiség   +   tankoltMennyiség   <=
üzemanyagTartály)
            {
                üzemanyagMennyiség += tankoltMennyiség;
                return true;
            }
            else return false;


        }
    }
```

6.2 Láthatóság

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

Az Autó osztályunk üzemanyag-mennyiségét most már szabályozni tudjuk: az *Utazás()* metódussal fogyasztani, a *Tankolás()*-sal növelni tudjuk a benne lévő mennyiséget, hasonlóan a való életbeli viselkedéshez. Fontos azonban megakadályozni azt, hogy az *üzemanyagMennyiség* adattag értékét közvetlenül megváltoztatva szabálytalanul manipulálni lehessen az autóban lévő mennyiséget. Ezért – összhangban az objektumorientált programozás elveivel – csak annyit engedjünk kívülről láttatni osztályunkból, ami a használathoz feltétlenül szükséges.
Az adattagok és a metódusok esetében a következő láthatósági módosítókat *(access modifiers)* használhatjuk azok deklarálása előtt:

- *public*
- *private*
- *protected*
- *internal*
- *internal protected*

A *public* esetében a tag mindenhonnan látható, ahonnan az osztály is. Tehát akár másik osztályból, vagy másik *assembly*-ből meghívható, ahonnan hivatkoznak rá.
Private választásával csak azon az osztályon (vagy struktúrán) belül érhető el a tag, ahol őt létrehozták.
Protected az elérés akkor, ha nemcsak az adott osztály, de a leszármazott osztály is láthatja az adott tagot. Az öröklődésről a fejezet végén részletesebben is beszélünk.

Az eddig ismertetett módosítók általában is ismertek valamennyi objektum-orientált nyelv eszköztárában. A most következő két módosító azonban specifikus a .NET-re nézve.

Az *internal* esetében bármely kódra kiterjed a láthatóság, de csak az adott szerelvényen, *assembly*-n belül. Idegen *assembly*-ből nem lesz hozzáférhető az így deklarált tag.
Az *internal protected* tagok az előzőek keverékeként viselkednek, azaz a vele egy *assembly*-ben lévő kódok számára látható, vagy egy leszármazott osztályban, ami akár másik szerelvényben is elhelyezkedhet.

A konkrét osztályunkra visszatérve a feladat megoldható például úgy, hogy az adattagokat *private* módosítóval elzárjuk a külvilág, illetve más osztályok elől, az őket kezelő metódusok viszont ha publikus, akkor rajtuk keresztül használhatók maradnak ugyan az adatok, de megszűnik a közvetlen változtatás, beavatkozás lehetősége.

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.
MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

```
using System;
using System.Text;
namespace Helló_Világ
class Autó
{
        //Tulajdonságok (adattagok)
        private string rendszám;
        private double átlagFogyasztás;
        private int üzemanyagTartály;
        private double üzemanyagMennyiség;

        //Metódusok
        public bool Utazás(int távolság)
…
        public bool Tankolás(int tankoltMennyiség)
…
}
```

6.3 Egységbezárás (encapsulation), tulajdonságok (properties)

Az objektum-orientált paradigmák közé tartozik az egységbezárás jellemző. Ez azt jelenti, hogy az adatokat és a rajtuk végezhető műveleteket egyrészt egy helyről tesszük elérhetővé, másrészt őket változtatni is csak a megadott „üzleti logika" alapján engedjük. Tehát kívülről nemcsak a hozzáférést tudjuk korlátozni, de arra is ügyelhetünk, hogy „értelmes" információkat őrizzünk bennük.
Az előző, láthatóságról írt részben lezártuk tehát az adattagokat és őket csak az üzemeltető metódusok változtatják, megakadályozva, hogy megváltoztassák, vagy akár csak megnézzék az autóban lévő üzemanyag mennyiséget.
Vannak azonban olyan szituációk, amikor el kell őket látni kezdőértékkel, vagy le kellene kérdezni például azt, hogy mennyi üzemanyag van aktuális a járműben.
A kívülről rejtett adattagokhoz ellenőrzött hozzáférést biztosító metódusokat, vagy más megoldásokat hívják *accessor*oknak, *mutator*oknak, vagy más terminológia szerint *getter*eknek, *setter*eknek. Ezek egyrészt lekérdezést, másrészt beállítást tesznek lehetővé az egyes adattagokra nézve.
Az autónk üzemanyagtartályába például csak 0, vagy annál nagyobb mennyiségű üzemanyag lehet. Ugyanakkor több sem lehet a mennyisége, mint az üzemanyagtartály maximális kapacitása (ezzel gondolom, a pénzügyőrök is egyetértenek). Ahhoz tehát, hogy megváltoztassuk a fenti adattag értékét, ezeket figyelembe kell vennünk.
Java nyelven és C#-ban is erre lehetőség van a következő metódusokkal:

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával"* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

```
public double getÜzemanyagMennyiség()
{
	return üzemanyagMennyiség;
}
public void setÜzemanyagMennyiség(double újMennyiség)
{
	if (újMennyiség > 0 && újMennyiség<= üzemanyagTartály)
	üzemanyagMennyiség = újMennyiség;
}
```

A getÜzemanyagMennyiség() metódussal lekérdezhető, a setÜzemanyagMennyiség() segítségével pedig beállítható a paraméterben adott értékre az adattag kezdőértéke.
C#-ban azonban (a *COM* múltnak köszönhetően) van még egy lehetőségünk.
A tulajdonságok *(properties)* segítségével lehetőségünk van az adattagok ellenőrzött lekérdezésére és beállítására, mintha csak egy-egy változót változtatnánk. A tényleges művelet elvégzésekor azonban egy-egy metódus fog lefutni, amelyben megadhatjuk az előző feltételeket is az adattag értékére vonatkozóan.

Ehhez hozzuk létre a következőket az Autó osztályunkban:

```
public double ÜzemanyagMennyiség
{
	get
	{
	 return üzemanyagMennyiség;
	}
	set
	{
	 if   (value   >   0   &&   value   <=   üzemanyagTartály)
üzemanyagMennyiség = value;
	}
}
```

A tulajdonság neve esetünkben szinte pontosan megegyezik az adattag nevével (üzemanyagMennyiség), csak a nagy kezdőbetű utal rá, hogy ez tulajdonság.

6.4 Konstruktorok, példányosítás

Noha már számos dolgot állítottunk az Autó osztályunkról, mindez idáig csak formális leírás volt és egy konkrét autót sem hoztunk létre belőle. Itt az ideje, hogy elkezdjük *példányosítani* az osztályunkat,

azaz kézzel fogható objektumokat készíteni az osztály felhasználásával.
A konstruálás az osztályon kívül a *new* kulcsszó segítségével a következőképpen történhet:

```
Autó sajátAutónk = new Autó();
```

Ezután már akár a következő módon is használhatjuk az osztályból létrehozott példányunkat:

```
sajátAutó.ÜzemanyagMennyiség = 55;
if (sajátAutó.Utazás(1000)) Console.WriteLine("Az utazás sikeres volt");
```

A metódusok és tulajdonságok a . operátorral választódnak el objektumunk nevétől.
Igen ám, de mi a helyzet a beállítatlan adatokkal? Mi lett az autónk rendszáma? Mennyi a fogyasztása? Számos adat megadása még hiányzik, ráadásul elég kényelmetlen volna minden autó létrehozásakor ezeket egyesével beállítani és el is felejtkezhetünk valamelyikről ezek közül.
Ezt a problémát orvosolhatjuk *konstruktor*(ok) segítségével. Ezek olyan speciális metódusok, amelyek a példányosításkor automatikusan lefuthatnak, beállítva az induláshoz szükséges adatokat akár paraméterek segítségével is. A konstruktorok könnyen felismerhetők: nevük kötelezően megegyezik az osztály nevével és nincs visszatérési típusuk, még *void* sem.
A konstruktort hozzuk létre az osztályon belül:

```
//Konstruktor
public Autó(string újRendszám, double újÁtlagFogyasztás, int újÜzemanyagTartály, double újÜzemanyagMennyiség)
{
	rendszám = újRendszám;
	átlagFogyasztás = újÁtlagFogyasztás;
	üzemanyagTartály = újÜzemanyagTartály;
	üzemanyagMennyiség = újÜzemanyagMennyiség;
}
```

Tulajdonképpen már akkor is volt konstruktorunk, amikor explicit még nem is hoztunk létre egyet sem. Ilyenkor az ősosztályának az alapértelmezett konstruktora, végső soron a System.Object

konstruktora hívódik meg, igaz ezekhez nem tartozik paraméter. Erre utalt a példányosításkor a *new Autó()* rész.
Mostantól azonban, hogy van saját konstruktorunk, ezt kell használni. Ez azt jelenti, hogy a korábbi híváshoz képest megváltozik a konstruálás módja.
Ahhoz hogy autókat hozhassunk létre, meg kell adnunk a konstruktorban megadott számú és típusú paraméter konkrét értékét az egyes autóinkra.

```
Autó sajátAutó = new Autó("ABC-123", 6.9, 55, 20);
```

A példában szereplő autó rendszáma ABC-123, 6.9 literes átlagfogyasztással rendelkezik 100 km-en átlagosan és az 55 literes üzemanyagtartálya jelenleg 20 literig van feltöltve.

Konstruktorból többet is készíthetünk, ez lehetővé teszi, hogy több, vagy kevesebb adat megadásával is készíthessünk autókat.

```
//Másik konstruktor
public  Autó(string  újRendszám,  double  újÁtlagFogyasztás,  int
újÜzemanyagTartály)
{
      rendszám = újRendszám;
      átlagFogyasztás = újÁtlagFogyasztás;
      üzemanyagTartály = újÜzemanyagTartály;
      üzemanyagMennyiség = 5;
}
```

Ezzel a konstruktorral már lehetővé válik olyan autó létrehozása is, amelyiknél a jelenlegi üzemanyag szint beállítása ha konstruáláskor elmarad, automatikusan 5 literre állítja annak értékét.
A konstruktorok mellett destruktorok is készíthetők, amelyek segítségével leírható egy feleslegessé vált objektumpéldány megsemmisítésének módja. A deklarálása az ~Osztálynév() formában lehetséges. A .NET Frameworkban működő *Garbage Collector* automatikusan gyűjti azokat az objektumokat (referencia típus), amelyek feleslegessé válnak, ezért egy null értékre állítást követően várható azok megsemmisítése. A GC szemétgyűjtése azonban indeterminisztikus módon hívódik, tehát nem tudjuk előre megmondani, hogy pontosan mikor fog lefutni. Ezt gyorsíthatjuk fel a *GC.Collect()* metódus meghívásával.

6.5 Statikus módosító

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.
MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

A példaosztályunkban eddig használt adattagok és metódusok úgynevezett példányadatok és –metódusok voltak. Ez azt jelenti, hogy több autó példányosításakor mindegyiknek különböző terület foglalódik a tárban. Minden autónknak tehát különböző rendszáma, fogyasztása, stb. lehet. Ha azonban olyan jellemzőt kívánunk használni, ami minden autó esetében közös, ezen érdemes változtatni.
Tegyük fel, hogy bevezetjük az Autó osztályban a költségek kiszámítása érdekében az üzemanyag árának tárolását!
Mivel ez az érték gyakorlatilag független attól, hogy melyik autónkról beszélünk (feltételezve, hogy valamennyi autónk azonos üzemanyagtípust használ), ezért ez az adat lehet *statikus.*

```
private static int üzemanyagÁr;
```

Ez az adattag statikus, ezért ez nem az egyes példányokhoz fog tartozni, hanem magához az osztályhoz, osztályadat lesz.
Amennyiben ezzel beállítjuk az üzemanyag árát, innentől kezdve minden autóra ugyanaz az érték fog vonatkozni.
Nemcsak adattagok, hanem metódusok, vagy akár osztályok is lehetnek statikusak, ezek természetesen nem példányosíthatók.

6.4 Öröklődés

Egy osztály képes más osztályból öröklődni. A szülő osztály meghatározása a gyerek osztály deklarációjakor a következő módon történhet:

```
class Szülő
{
}
class Gyerek : Szülő
{
}
```

A gyerek (leszármazott) osztály megörökli a szülő osztály nem-privát adatait és metódusait. Ezeket saját adatokkal és metódusokkal is bővítheti. Az örökléssel tehát egy meglévő típust, illetve osztályt bővíthetünk ki. A C# - szemben például a C++ nyelvvel – csak egyetlen szülőt enged meg a gyerekosztály számára.
A következő példában leszármaztatjuk Autó osztályunkat. Az új, Teherautó nevű osztály a *kapacitás* adattaggal bővíti a meglévő funkcionalitást.

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

A konstruktorban felhasználhatjuk azt a tényt, hogy szinte valamennyi paramétert (a kapacitás kivételével nyilván) már feldolgozta a szülő osztály konstruktora. Ezt a *:base(paraméterek)* hivatkozás biztosítja.

```
class Teherautó : Autó
{
//Új adattag
private int kapacitás;
//A konstruktor hívásakor továbbadja a kapott paramétereket a szülőosztály konstruktora számára
public Teherautó(string újRendszám, double újÁtlagFogyasztás, int újÜzemanyagTartály, double újÜzemanyagMennyiség, int újKapacitás)
: base(újRendszám, újÁtlagFogyasztás, újÜzemanyagTartály, újÜzemanyagMennyiség)
       {
       kapacitás = újKapacitás;
       }
}
```

A példányosításkor készíthetünk Autó, illetve most már Teherautó típusú objektumokat is. A gyerekosztály típusa lehet megegyező a szülőével, de fordítva nem lehetséges.

```
Teherautó kisTeherautónk = new Teherautó("DEF-456", 8.5, 65, 25, 300);
//Ez megengedett
Autó másikTeherautónk = new Teherautó("GHI-789", 9.2, 70, 40, 400);
//Ez nem megengedett
//Teherautó harmadikTeherAutó = new Autó("JKL-012", 8.8, 65, 50);
```

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

7. Struktúrák

A struktúrák hasonlítanak az osztályokra abban a tekintetben, hogy ők is képesek adattagokat, műveleteket, tulajdonságokat tárolni. Az osztályokkal szemben a struktúrák azonban nem referencia, hanem érték típusúak. Az osztály objektumaival szemben egy struktúra típusú változó nem referenciát tartalmaz, hanem közvetlenül tárolja a hozzá tartozó értékeket. Nem igényel heap foglalást sem a használatuk. Amikor egy objektumpéldányt létrehozunk, az a heap-ben lefoglalódik. Egy struktúra létrehozásakor az a stack-be (verembe) kerül. Másoláskor a struktúra értéke másolódik.
Elsősorban kisméretű adatszerkezeteknél ajánlott a használata a kisebb helyfoglalás miatt, különösen, ha ilyen szerkezetből sokat kell lefoglalnunk a tárban.

Definiálása:

attribútumok módosítók struct StruktúraNeve {
 struktúra_törzse;
}

A struktúrák is tartalmazhatnak konstruktorokat, de ezek nem lehetnek paraméter nélküliek.
Nincs öröklés. Egy struktúra nem származhat másik struktúrából, vagy osztályból és ő maga sem lehet szülője egy másiknak.

```
struct Háromszög
    {
        //Adattagok
        private int a;
        private int b;
        private int c;

        //Paraméter nélküli konstruktor nem megengedett
        //public Háromszög(){}
        //Paraméteres konstruktor viszont igen
        public Háromszög(int újA, int újB, int újC)
        {
            a = újA;
            b = újB;
            c = újC;
        }

        //Tulajdonságok
        public int A
        {
            set { if (value > 0) a = value; }
```

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

```
        }
        public int B
        {
            set { if (value > 0) b = value; }
        }
        public int C
        {
            set { if (value > 0) c = value; }
        }
        //Metódus
        public int Kerület()
        {
            return a + b + c;
        }
    }
class Főprogram
    {
    public static void Main()
      {
       Háromszög derékszögűHáromszög = new Háromszög(2, 4, 5);
       derékszögűHáromszög.A = 3;
       Console.WriteLine("Kerülete:"+
derékszögűHáromszög.Kerület());
      }
    }
```

A fenti struktúra privát adattagokat, struktúrát, tulajdonságokat és metódust is tartalmaz. Hívásakor először konstruálunk, aztán megváltoztatjuk az A tulajdonságán keresztül az adattagját, majd kiíratjuk a metódus segítségével a kerületét.

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

8. Gyűjtemények, generikusok

A gyűjtemények segítségével nagy számban ismétlődő adataink memóriában tartására, vagy éppen különleges hozzáférésű adattárolási megoldásokra tehetünk szert. Ilyenek lehetnek a korábban tanult tömbhöz képest olyan új módszerek, mint a sor, verem, vagy épp a szótáron alapuló gyűjtemények.
A gyűjtemények a *System.Collection* névtérben találhatók.
Néhányat a következő tábla mutat be közülük:

Gyűjtemény neve	Leírása
ArrayList	Dinamikus méretű, index alapú tömb
SortedList	Rendezett kulcs/érték párok
Queue	Sor
Stack	Verem
Hashtable	Kulcs alapján hozzáférhető kulcs/érték párok
StringDictionary	Sztring párok táblája

8.1 ArrayList

Ennek a gyűjteménynek a segítségével objektumok tárolására nyílik index alapon lehetőség. A gyűjteménybe tehát egyszerű típusok ugyanúgy felvehetők, mint struktúrák, vagy osztályobjektumok.

Jellemzői:

- Dinamikusan, igény szerint növelhető a mérete
- Index alapon hozzáférhetőek az elemei
- Mivel objektumokat tárol, ezért boxing / unboxing műveleteket igényel a tárolóba helyezéskor, illetve kivételkor.

A használatához szükséges, fontosabb metódusok

- *Add()*: objektum hozzáadása a gyűjtemény végéhez
- *AddRange()*: több elem(tömb, gyűjtemény) felvétele
- *Clear()*: a gyűjtemény elemeinek törlése
- *Insert()*: adott pozícióba elem beszúrása
- *InsertRange()*: adott helyre több elem beszúrása
- *Remove()*: adott elem első előfordulását törli
- *RemoveAt()*: adott indexű elemet töröl
- *RemoveRange()*: elemtartomány törlése
- *IndexOf()*: adott elem indexét adja vissza, vagy -1-et, ha nincs benne a keresett elem
- *Contains()*: adott elem benne van-e a listában?
- *Sort()*: elemek rendezése

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

Hasznos tulajdonságok:

- *Count*: elemszám
- *Capacity*: a gyűjtemény jelenlegi kapacitása
- *Item*: adott indexű elem elérése/beállítása

```
//Létrehozás
ArrayList tömblista = new ArrayList();

//Elemek beszúrása a végére
tömblista.Add("Helló Világ");
tömblista.Add(new Autó("ABC-123", 7.9, 55));
tömblista.Add(2013);

//Elem beszúrása az első pozícióra
tömblista.Insert(0, true);

//Adott pozícióból törlés (az Autó objektum)
tömblista.RemoveAt(2);

//Elemszám
Console.WriteLine(tömblista.Count);

//Kapacitás
Console.WriteLine(tömblista.Capacity);

//Hivatkozás indexszel
Console.WriteLine(tömblista[1]);

//Bejárás és elemkiíratás
foreach (object tömblista_eleme in tömblista)
Console.WriteLine(tömblista_eleme);
```

8.2 Sorok

FIFO (First-in-first-out) hozzáférést megvalósító sorok készítését támogató gyűjtemény. A segítségével olyan adatszerkezetet készíthetünk, amelyben a legelsőnek betett elemhez történhet először hozzáférés.
Jellemzője, hogy mivel itt is objektumokat tárol a gyűjtemény, ezért boxing, unboxing segítségével kerül be a gyűjteménybe elem, vagy kerül ki onnan. Fontos tulajdonsága a gyűjteményben lévő elemszámot visszaadó *Count*.

Egyedi, rá jellemző metódusai:

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.
MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

- Enqueue(): elem hozzáadása a sor végére
- Dequeue(): eleme kivétele a sor elejéről
- Peek(): kukucskálás: megnézzük az elemet a sor elején, de nem vesszük ki azt

```
//Létrehozás
Queue sor = new Queue();

string szöveg = "ABCDEFG";
//A szöveg betűit egyesével tesszük be a sorba
foreach(char betű in szöveg)
	sor.Enqueue(betű);

//A sor hossza
Console.WriteLine(sor.Count);

//Egyesével kivesszük és kiírjuk a berakott elemeket
while(sor.Count>0)
	Console.Write(sor.Dequeue());
```

8.3 Verem

A verembe is bármilyen objektum helyezhető, a hozzáférés azonban az előző fordítottja. A *LIFO (last-in-first-out)* elv alapján azokat az elemeket vehetjük elsőként ki, amit legutoljára beraktunk. A *Count* property itt is az aktuális elemszámról tájékoztat. A tagfüggvények nevei azonban megváltoztak, ezzel is jelezve a másféle hozzáférést.
A *Push()* metódussal betenni, a *Pop()* segítségével pedig az utoljára berakott elemet lehet kipattintani a vermünkből.

```
//Létrehozás
Stack verem = new Stack();
//Egyesével növekvő számokat pakolunk a vermünkbe.
for (int i = 1; i <= 10; i++)
	verem.Push(i);

//A kukucskálás ezúttal a legutolsó elemet mutatja
Console.WriteLine(verem.Peek());

//Egyesével kivesszük és kiírjuk a berakott elemeket
while(verem.Count>0)
	Console.Write(verem.Pop()+" ");
```

A példaprogram eredményeként a betett számok csökkenő sorrendben íródnak ki a konzolra.

8.4 Hashtable

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.
MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

A *Hashtable* olyan szótár alapú gyűjtemény, ahol az elemeket nem egy egyszerű *indexer* teszi elérhetővé, hanem egy általunk megadott tetszőleges kulcs alapján folyik a megadott elem keresése.
Ennek ellenére szükség esetén továbbra is elérhető az index alapú hozzáférés benne.
Jelentős változás az előzőekhez képest ezért, hogy a gyűjtemény elemei nem objektumok, hanem *DictionaryEntry* elemek! Ezen elempárok kulcs és érték feléhez a hozzáférés a *Value* és *Key* tulajdonságokon keresztül történhet.
Az elemek hozzáadása is az *Add(kulcs, érték)* metódus segítségével, vagy indexszerűen történhet.

További elérhető metódusok:

- Clear(): minden elem törlése a gyűjteményből
- Contains(), ContainsKey(), ContainsValue(): megadott kulcs/érték megtalálható-e a gyűjteményben
- Remove(): adott kulcsú elem eltávolítása

```
//Létrehozás
Hashtable szótár = new Hashtable();

//Elemek hozzáadása
szótár.Add("alma", "apple");
szótár.Add("körte", "pear");
szótár.Add("barack", "peach");
szótár.Add("rossz kulcs", "rossz érték");

//Elemszám
Console.WriteLine("A szótár elemeinek száma:"+szótár.Count);

//Elem eltávolítása kulcs megadásával
szótár.Remove("rossz kulcs");

//Tartalmaz adott kulcsot a szótár?
if (szótár.ContainsKey("alma"))
{
	Console.Write("Az alma kulcs megtalálható a szótárban, értéke:");
	//Kulcshoz tartozó érték
	Console.WriteLine(szótár["alma"]);
}

Console.WriteLine("A szótár teljes tartalma:");

//Bejárás, kulcs és érték kiíratása
foreach (DictionaryEntry szótárelem in szótár)
```

```
        Console.WriteLine(szótárelem.Key+":"+szótárelem.Value);
```

8.5 SortedList

Ez a gyűjtemény alkalmas olyan Hashtábla létrehozására, amely már beszúráskor is képes kulcs szerint rendezni. Az elemek hozzáférése történhet kulcs, vagy index szerint is.

A gyűjtemény jellegzetes metódusai:

Metódus neve	**Leírása**
GetKey()	Kulcs kiolvasása index szerint
GetKeyList()	A kulcsok rendezett listáját visszaadja
GetValueList()	Visszaadja az értékek listáját
IndexOfKey()	A megadott kulcsú elem indexe
IndexOfValue()	Adott érték első előfordulásának indexe
GetByIndex()	Elem visszaadása index szerint
SetByIndex()	Elem beállítása index alapján

A következő példaprogram az előző szótárt rendezett

```
//Létrehozás
SortedList rendezettSzótár = new SortedList();
rendezettSzótár.Add("barack", "peach");
rendezettSzótár.Add("körte", "pear");
rendezettSzótár.Add("alma", "apple");

//A szótár bejárása a kulcsok ABC rendjében
foreach(DictonaryEntry elem in rendezettSzótár)
        Console.WriteLine(elem.Key+":"+elem.Value);
```

A C# 2.0-s változatának egy komoly újítása volt a generikusok bevezetése. Ennek felhasználásával típusbiztos adatstruktúrákat hozhatunk létre, ami a teljesítményre is jótékony hatással van. Természetesen a hozzáféréskor sem kell foglalkozunk az esetleges visszacastolással. A gyűjteményeknél alkalmazni kívánt típusokat a < , > zárójelek között kell megadnunk. A *System.Collections.Generic* névtérben találjuk az osztályait.

8.6 Típusos lista

A *List<>* hasonlít a korábban tárgyalt *ArrayList*-hez, azzal a komoly eltéréssel, hogy ez már típusos. Természetesen ezért az elemei sem objektumok, hanem a megadott típusból állnak. Így új elemeket is csak az adott típusból vehet fel:

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

```
//Létrehozás
List<int> egészLista = new List<int>();
//Ez megengedett
egészLista.Add(1);
egészLista.Add(2);
//Ez már nem
//egészLista.Add("a");
```

8.7 Típusos sor

A következő programrészlet egy olyan sort *(queue)* hoz létre, amelynek az elemei csak *string* típusú objektumok lehetnek.
Amikor kiveszünk elemet, azt nem szükséges visszacastolni, mert ismert a várható elem típusa.

```
Queue<string> szövegesSor = new Queue<string>();
szövegesSor.Enqueue("első");
szövegesSor.Enqueue("második");

//Elem kivételekor nem szükséges castolni
string elsőBerakottElem = szövegesSor.Dequeue();
```

8.8 Típusos verem

A most bemutatni kívánt programrészlet egy *char* (karakter) típusú elemeket befogadó vermet hoz létre. A veremből kivételkor mégis egy számnak (int) adjuk az eredményt. Mi lesz a program futásának eredménye?

```
Stack<char> betűVerem = new Stack<char>();
betűVerem.Push('a');
betűVerem.Push('b');
betűVerem.Push('c');
//Elvileg kivételkor ez típuseltérés
int szám = betűVerem.Pop();
//Miért működik mégis?
Console.WriteLine(szám);
```

8.9 Dictionary

Ez a szótár hasonlít a korábban tárgyalt *HashTable*-re. Ő is kulcs/érték párok tárolására ad módot. Itt is lehetőségünk van indexszerűen, vagy az *Add()* metódussal gyarapítani a gyűjtemény méretét. Fontos különbség azonban egyrészt, hogy a kulcs és az érték típusát itt előre meg kell határoznunk, később attól eltérni nem lehet.

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

Másfelől a gyűjtemény bejárásakor az elemek típusa a korábbitól eltérően itt nem *DictionaryEntry* lesz (bár a neve alapján erre következtethetnénk), *hanem KeyValuePair<kulcs_típus, érték_típus>* lesz.
Ezt demonstrálja a következő rövid részlet is.

```
//Típusos szótár létrehozása
Dictionary<int, string> személyekSzótár = new Dictionary<int,
string>();
//Elemet hozzáadhatunk így is
személyekSzótár.Add(1234, "Kovács Tamás");
//És így is
személyekSzótár[1235] = "Szabó Sándor";

//Bejáráskor az elemek típusa KeyValuePair lesz
foreach(KeyValuePair<int, string> személy in személyekSzótár)
   Console.WriteLine(személy.Key+":"+személy.Value);
```

8.10 Generikus metódusok

A metódusok alapesetben csak olyan megadott típusú adatokkal képesek működni, amelyeket paraméterként megkapnak, és amelyeket esetleg visszaadnak. Előfordulhat azonban olyan általános probléma, amely többféle szituációban, több típus esetében is előfordulhat.

Két változó cseréjére például megírhatjuk a következő metódust:

```
public static void Csere(ref int a, ref int b)
{
	int cs = a;
	a = b;
	b = cs;
}
```

A metódus kiválóan működik egész típusokra, referencia alapon megkapja a változókat, majd megcseréli őket.
Azonban ez csak az egész típusokra lesz alkalmazható, pedig a változók cseréje hasonló módon kivitelezhető lenne más típusok esetében is.
A megoldás egy olyan generikus metódus lesz, ahol nem adunk meg konkrét típust, csak jelöljük azt, pl. T-vel.

```
public static void Csere<T>(ref T a, ref T b)
{
```

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

```
        T cs = a;
        a = b;
        b = cs ;
}
```

Innentől a T (mint *template*) jelöli a konkrét típust (lehet tetszőlegesen máshogy is jelölni), bárhol szükség van a típus nevére, ezzel fogjuk helyettesíteni.
Amikor viszont meghívjuk a metódusunk, meg kell adni az aktuális paraméterekhez használni kívánt típust:

```
double x = 1;
double y = 2;

Csere<double>(ref x, ref y);

Console.WriteLine("x a csere után:" + x);
Console.WriteLine("y a csere után:" + y);
```

A generikus metódusnak köszönhetően bármilyen típusú adatok cseréjére módunk nyílik innentől, a megfelelő típus megjelölése mellett.

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

9. Fájlkezelés

Az általunk eddig létrehozott objektumok élettartama a program befejeztével véget ér. Az adatok perzisztens tárolásához érdemes belemélyednünk a fájlkezelés rejtelmeibe is.
Valamennyi, ebben a fejezetben használt osztály és metódus a *System.IO* névtérben található, ezért ajánlott ennek behívása a *using* segítségével a most következő programokba.

9.1 Fájlrendszer és kezelése

Először tekintsük át, hogy milyen lehetőségeink vannak a programunkból elérhető fájlrendszerek használatára!
Nézzük át, hogyan történhet a meghajtók kezelése, majd foglalkozzunk azzal, hogyan kezelhetjük a rajta lévő fájlokat, könyvtárakat. Ezek kezelése, másolása, a hozzájuk kapcsolódó információk lekérdezése lesz a feladata ennek a fejezetnek.

A munkánkhoz információs és segédosztályokat is biztosít a .NET:

- Információs osztályok
  - *DriveInfo*
  - *FileSystemInfo*
    - *FileInfo*
    - *DirectoryInfo*

- Segédosztályok
  - *Path*
  - *Directory*
  - *File*

Az információs osztályok közül a *DriveInfo* osztály írja le a meghajtókhoz kapcsolódó műveleteket és tulajdonságokat.
A *FileSystemInfo* ősosztály a fájlrendszert alkotó bejegyzések, a fájlok és könyvtárak információit és a rajtuk végezhető közös műveleteket tartalmazza. Ebből az ősosztályból olyan osztályok származnak (*FileInfo* és *DirectoryInfo)*, amelyek specializáltak az adott bejegyzés fajtájára.

Elsőként nézzük meg, hogyan szerezhetünk a programunkból információt a körülöttünk lévő meghajtókról!

```
DriveInfo[] minden_meghajtó = DriveInfo.GetDrives();
foreach (DriveInfo egy_meghajtó in minden_meghajtó)
Console.WriteLine("Név:" + egy_meghajtó.Name + " , típusa:" +
egy_meghajtó.DriveType);
Console.ReadKey();
```

A *DriveInfo.GetDrives()* statikus metódus visszaadja a futtatást végző gépen elérhető logikai meghajtókat egy *DriveInfo* típusú tömbben. Ezen tömbelemek tulajdonságai ezután a *foreach* ciklusban kényelmesen lekérdezhetők.

Elérhető fontosabb tulajdonságok:

- Name: a meghajtó neve (pl. C:\ )
- DriveType: a meghajtó típusa (Fixed, CDRom, Network, stb.)
- VolumeLabel: a meghajtó kötetcímkéje
- TotalSize: a meghajtó teljes mérete bájtokban
- TotalFreeSpace: a meghajtón lévő összes szabad hely bájtokban
- AvailableFreeSpace: a futtatást végző felhasználó által használható szabad hely (kvóta használatakor eltérhez az előzőtől)
- DriveFormat: a meghajtó által használt fájlrendszer (például FAT32, vagy NTFS)

Ha kiválasztottuk a meghajtót, vegyük szemügyre a tartalmát! Listázzuk ki a meghajtó gyökérkönyvtárának tartalmát, külön a fájlokat és a mappákat!

```
DirectoryInfo dir_info = new DirectoryInfo("C:\\");
Console.WriteLine("Fájlok listája:");
Console.WriteLine("---------------");
foreach (FileInfo fájl in dir_info.GetFiles())
Console.WriteLine(fájl.Name);

Console.WriteLine("Könyvtárak listája:");
Console.WriteLine("---------------");
foreach (DirectoryInfo könyvtár in dir_info.GetDirectories())
Console.WriteLine(könyvtár.Name);
```

A *DirectoryInfo* osztályt úgy példányosítjuk, hogy a konstruktorának paraméterként azt az útvonalat kell megadnunk, ahol a vizsgálni kívánt könyvtár megtalálható (C:\). A *GetFiles()* metódus *FileInfo* típusú bejegyzéseket gyűjt az adott mappában. A hasonlóan működő *GetDirectories()* által szolgáltatott elemek pedig *DirectoryInfo* típussal rendelkeznek. Az itt használt *Name* tulajdonságok a bejegyzések nevét fogják kiírni a konzolra, fájl és könyvtár esetében egyaránt.

Ezen kívül hasznos tulajdonságok lehetnek még:

- *Exists*: létezik-e az adott bejegyzés a könyvtárban?

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

- *FullName*: a teljes név az elérési útvonalával együtt
- *Extension*: fájlok esetében a név kiterjesztés része
- *Parent*: a megadott alkönyvtár szülőkönyvtára
- *Root*: a megadott könyvtár gyökere
- *CreationTime*: a bejegyzés létrehozási ideje
- *LastAccessTime*: a bejegyzés utolsó hozzáférésének ideje
- *LastWriteTime*: a bejegyzés utolsó módosításának időpontja

A DirectoryInfo osztály lényegesebb metódusai:

- *Create()*: hozzuk létre a megadott könyvtárat!
- *Delete()*: töröljük a könyvtárat, ha az üres!
- *GetFiles()*: a jelenlegi könyvtárban lévő fájlok listája
- *GetDirectories()*: a jelenlegi könyvtárban lévő mappák listája
- *MoveTo()*: a könyvtár mozgatása új helyre
- *ToString()*: az útvonal visszaadása

A *FileInfo* osztály tulajdonságai részben megegyeznek a *DirectoryInfo*-éval, de természetesen van néhány speciális, csak erre jellemző tulajdonságok:

- *DirectoryName*: a kijelölt fájlt tartalmazó könyvtár elérési útvonalát adja vissza
- *IsReadOnly*: a fájl csak olvasható tulajdonságának lekérdezése, vagy beállítása is itt lehetséges
- *Length*: a fájl mérete bájtokban.

A *FileInfo* osztály metódusaival, hasonlóan a *DirectoryInfo*-hoz, létrehozhatunk (*Create()*), törölhetünk (*Delete()*), mozgathatunk (*MoveTo()*) állományokat. Ezeken kívül még számos műveletet is végezhetünk velük:

- *CopyTo()*: a fájl másolása
- *Encrypt(), Decrypt()*: a fájl titkosítása/feloldása a jelenlegi felhasználó számára
- *Open()*: a fájl megnyitása
- *Replace()*: a fájl tartalmának lecserélése egy másik fájléval.

A fájlok, könyvtárak és útvonalak kezelésére a *Path*, a *Directory* és a *File* osztály statikus metódusai, valamint tulajdonságai is felhasználhatók.

A következő kis kódrészlet előbb létrehoz egy könyvtárat a megadott útvonalon, majd belemásol egy fájlt, végül törli az eredeti helyéről az átmásolt állományt.

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

```
String forrásmappa = "C:\\";
String forrásnév = "állomány.txt";
String célmappa = "C:\\Saját\\";
Directory.CreateDirectory(célmappa);
File.Copy(forrásmappa + forrásnév, célmappa + forrásnév, true);
File.Delete(forrásmappa + forrásnév);
Console.ReadKey();
```

Természetesen gondot okozhat, ha az adott állomány a leírt útvonalon nem létezik, vagy nincs hozzáférési jogosultsága a programunknak. Ilyen esetben a problémának megfelelő típusú kivétel generálódik. Erről a kivételkezelésről szóló fejezetben részletesebben is írunk

9.2 Fájlok készítése, felhasználása

Az előző fejezetben még csak információkat gyűjtöttünk, vagy alapvető műveleteket (másolás, törlés) végeztünk állományokkal. Ahhoz azonban, hogy felhasználhassuk őket adataink tárolására, meg kell ismerkednünk a *stream*-ekkel is.
A *stream* az I/O műveletek alapjául szolgáló adatfolyam. Szekvenciális és közvetlen elérésű adatok kezelésére egyaránt alkalmas lehet. A technika nem csupán fájlok kezelésére alkalmas, a használatával a memóriában, vagy a hálózatban is megvalósíthatunk kommunikációt.
Erre mutat példát néhány folyamosztály:

- *CryptoStream*
- *FileStream*
- *MemoryStream*
- *NetworkStream*

A *Stream* osztály valamennyi folyam közös őse. Ennek köszönhetően valamennyi folyam kezelésében találhatunk közös elemeket. Minden folyamot megnyitással veszünk igénybe, majd az olvasási/írási műveletek következnek. A folyamok igénybevételének végét azok zárása fogja jelezni.
Ehhez kötődik néhány jellemző metódusnév:

- *Read()*: olvasás a stream-ből
- *Write()*: stream-be írás
- *Seek()*: pozícionálás közvetlen hozzáférés esetében
- *Close()*: bezárás

A *FileStream* osztály segítségével az adatainkhoz bájtonként férhetünk hozzá. Ehhez meg kell határozni a megnyitás, valamint az adatkezelés módját is.
A megnyitás módja a következők közül választható:

- *Open*: meglévő fájl megnyitása
- *OpenOrCreate*: fájl nyitása, ha nem létezik, létrehozza
- *Create*: mindenképpen új fájlt hoz létre
- *CreateNew*: újat hoz létre, de hiba (kivételt okoz), ha már létezik
- *Truncate*: létező fájl tartalmát törli
- *Append*: a létező fájlt a végén bővíti

A *Lock()* metódussal zárolható más folyamatok elől a fájlunk. Ennek feloldása az *Unlock()*-kal lehetséges.

```
FileStream file = new FileStream("C:\\állomány.txt",
FileMode.Open, FileAccess.Read);
int bájt = file.ReadByte();
while (bájt != -1)
{
	Console.Write((char)bájt);
	bájt = file.ReadByte();
}
file.Close();
```

A példaprogram a meglévő állományt olvasásra megnyitja, majd bájtonként olvas belőle a *ReadByte()* metódussal. A beolvasott tartalmat megjeleníti a konzolon. A fájl végét a beolvasott -1-es érték jelzi.

A fájl lehetséges hozzáférési módjai:

- *Read*: olvasásra
- *Write*: írásra
- *ReadWrite*: olvasásra és írásra is

A fenti program szimpla szöveges állományok esetében alkalmazható ugyan, de mi a helyzet a bináris adatainkkal? Kényelmetlen adattárolási módszer volna a meglévő adatszerkezeteink bájtonkénti mentése.
Erre igyekszik megoldást nyújtani a *BinaryWiter* osztály. Az osztály egyszerűbb típusaink tárolására nyújt megoldást. A numerikus, karakteres, szöveges adataink tárolását saját metódusokkal segíti. Ehhez elsőnek be kell csomagolnunk a *FileStream*-ünket egy *BinaryWriter* példány konstruktorába.

```
FileStream file = new FileStream("C:\\Users\\Veca\\adatok.dat",
FileMode.Create, FileAccess.Write);
BinaryWriter bináris = new BinaryWriter(file);
```

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

```
int szám = 2013;
bool választ = false;
char betű = 'A';
string szöveg = "Körtefa";
byte[] tömb = new byte[]{ 1, 2, 3, 4, 5 };
bináris.Write(szám);
bináris.Write(választ);
bináris.Write(betű);
bináris.Write(szöveg);
bináris.Write(tömb);
file.Close();
```

A megoldás azonban csak akkor ér valamit, ha a mentett adatainkat vissza is tudjuk olvasni a későbbiekben.
Ehhez a *BinaryReader* osztályt hívhatjuk segítségül.

```
FileStream file = new FileStream("C:\\Users\\Veca\\adatok.dat",
FileMode.Open, FileAccess.Read);
BinaryReader binOlvas = new BinaryReader(file);
int szám = binOlvas.ReadInt32();
Console.WriteLine(szám);
bool választ = binOlvas.ReadBoolean();
Console.WriteLine(választ);
char betű = binOlvas.ReadChar();
Console.WriteLine(betű);
string szöveg = binOlvas.ReadString();
Console.WriteLine(szöveg);
byte[] tömb = binOlvas.ReadBytes(5);
foreach(byte bájt in tömb)
Console.WriteLine(bájt);
file.Close();
```

A megfelelő adattípus kinyeréséhez a *BinaryReader* objektumunknak a megfelelő metódusát kell meghívni.

Metódus	Típus
ReadInt32()	int
ReadBoolean()	bool
ReadChar()	char
ReadString()	string
ReadBytes()	byte típusú tömb

Sajnos ennek a megoldásnak is vannak hátrányai.
Egyrészt a fájl visszaolvasásakor ugyanazt az olvasási sorrendet kell ismerni és alkalmazni, mint amit íráskor használtunk, másrészt összetett adatok tárolására nincsenek metódusai az osztálynak. Ezek mentéséről külön kell gondoskodnunk.

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

9.3 Szerializáció

Összetett szerkezetek, például objektumok tárolásának egyik legegyszerűbb módja a szerializáció. Ennek segítségével egy mozdulattal letárolhatjuk példányaink állapotát egy kijelölt fájlba. A visszatöltéshez deszerializáció szükséges.
Azokat az osztályokat, amelyek objektumait ilyen módon tárolhatjuk, a `[Serializable()]` attribútummal jelölhetjük az osztály deklarálása előtt.
A *System.Runtime.Serialization.Formatters.Binary* névtérben lévő *BinaryFormatter* osztály gondoskodhat objektumaink ilyen formában történő mentéséről, vagy beolvasásáról.
Az osztályoknál tárgyalt Autó osztályunk példányainak mentéséről és visszaolvasásáról ilyen módon egyszerűen gondoskodhatunk.

```
[Serializable()]
    class Autó
    {
     …
    }

    class Főprogram
    {
        public static void Main()
        {
            //szerializáció
            FileStream file = new FileStream("C:\\Users\\Veca\\adatok.dat", FileMode.Create, FileAccess.Write);
            BinaryFormatter formázott = new BinaryFormatter();
            Autó sajátAutó = new Autó("ABC-123", 6.9, 55, 20);
            formázott.Serialize(file, sajátAutó);
            file.Close();

            //deszerializáció
            FileStream fileOlvas = new FileStream("C:\\Users\\Veca\\adatok.dat", FileMode.Open, FileAccess.Read);
            BinaryFormatter formázottOlvas = new BinaryFormatter();
            Autó újAutó = (Autó)formázottOlvas.Deserialize(fileOlvas);
            file.Close();
```


MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

```
            Console.WriteLine(újAutó.getÜzemanyagMennyiség());
            Console.ReadKey();
        }
    }
```

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL

A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

10.Kivételkezelés

A programunk futtatása során váratlan, vagy kivételes események is bekövetkezhetnek. Ilyen esemény lehet egy szokatlan típusú bemenő adat, amit a programunk felhasználója megad, de kivétel következhet be akkor is, ha egy szükséges erőforrás nem található meg a futtatás közben, vagy nincs jogosultságunk elérni azt.
Magát a kivételt generálhatja a CLR, vagy egy általunk használt külső könyvtár, sőt bizonyos esetekben mi magunk is.
A kivételek kezelése a *try – catch – finally* struktúrák segítségével történhet.

Tekintsük a következő egyszerű kódot:

```
int számláló = 100;
Console.WriteLine(" A számláló értéke=" + számláló);

Console.Write("Add meg a nevező értékét:");
int nevező = int.Parse(Console.ReadLine());

Console.WriteLine(számláló+"/"+nevező+"="+ (számláló/nevező));

Console.ReadKey();
```

A részlet bekér szövegesen a konzolról egy egész értéket, majd kiírja a számláló és a nevező hányadosát a kijelzőre.

Mi történik akkor, ha a felhasználó 0-t ír be?
A program önálló futtatása esetén ekkor az egész program leáll egy hibaüzenettel. Ha a Visual Studio-ban futtatjuk a kódot, akkor a program futása félbeszakad, és a *Debug* menü segítségével dönthetünk arról, hogy félbehagyjuk-e a futtatást, vagy megpróbáljuk folytatni azt.
Az egész számok osztása esetén ilyenkor egy futásidejű kivétel generálódik, amely a *System.DivideByZeroException* nevet viseli. A neve alapján sejthető, hogy a kivételek hierarchikus rendbe szerveződnek. Minden kivétel *object* is egyúttal, a legtöbb kivétel az *Exception* osztályból származik. A fenti kivétel például a következő hierarchiából származik:

System.Object
    System.Exception
        System.ArithmeticExpression
            System.DivideByZeroException

Ahhoz, hogy erre az esetre felkészüljünk, a kritikus utasításokat öleljük körbe *try* blokkal! A blokkot közvetlenül egy *catch* szakasz

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

kövesse, amelyben meghatározható, hogy mely kivételt szeretnénk benne elkapni (*catch*), és feldolgozni!

```
int számláló = 100;
Console.WriteLine(" A számláló értéke=" + számláló);
Console.Write("Add meg a nevező értékét:");
try
{
       int nevező = int.Parse(Console.ReadLine());
       Console.WriteLine(számláló+"/"+nevező+"="+(számláló/nevező
));
}
catch (System.DivideByZeroException)
{
       Console.WriteLine("Nullával osztás történt");
}
```

Ebben az esetben a program futtatásánál a 0 beírásakor a keletkező kivételt elkapjuk, majd feldolgozzuk egy kiírás segítségével.
A program futása a kivétel feldolgozását követően a *try* blokk után folytatódhat.

A *try* blokkban lévő utasításokat úgy tekintjük: lehet, hogy akár egyszer sem fognak lefutni. Ezért az itt előforduló deklarációkkal óvatosan bánjunk, mivel a blokkon kívül ezek nem érvényesek! Ilyenkor ezeket érdemes a *try*-on kívül, előtte létrehozni.

Sajnos nem csupán egyféle problémával szembesülhetünk a program egy adott részén. Elképzelhető a fenti példában az is, hogy olyan jelsorozatot ír be a felhasználó, amit nem lehet sikeresen számmá alakítani (például az „alma” szó nehézkesen alakítható közvetlenül számmá…) *Catch* szakasz több is lehet a kódban, közvetlenül egymás után. Ilyenkor a kivétel az első, megfelelő kivételtípust elkapó *catch* szakasznak lesz odaadva. Ügyeljünk ezért arra, hogy a kivételeket feldolgozó *catch* részek olyan sorban legyenek felsorolva, hogy a legspecifikusabbaktól a legáltalánosabbak felé haladjon a sor! Ha nem így tennénk, az általánosabb kivételeket kezelő részek nem engednék szóhoz jutni a kisebb kivételekre specializálódott szakaszokat. A legáltalánosabb *Exception* kivételt például utoljára érdemes felsorolni emiatt a kódban, bár .Net Frameworkben a fordító sem engedné, hogy máshová tegyük!

```
try
{
       int nevező = int.Parse(Console.ReadLine());
```

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.
MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

```
	Console.WriteLine(számláló+"/"+nevező+"="+(számláló/nevező
));
}
//Nullával osztás elkapása
catch (System.DivideByZeroException)
{
	Console.WriteLine("Nullával osztás történt");
}
//Rosszul formázott adatbevitel elkapása
catch (System.FormatException)
{
	Console.WriteLine("Szám helyett szöveget kaptam!");
}
//Egyéb itt nem lekezelt kivétel elkapása, pl. túl nagy érték
beírása esetén
catch (Exception e)
{
	//A nevesített kivétel Message tulajdonsága leírja a
konkrét problémát
	Console.WriteLine("Általános kivétel történt:
"+e.Message);
}
```

A fenti kód háromféle kivétel elkapására készült fel: a nullával osztásra, a formázási kivételre, valamint minden más esetben az *Exception* kivételre írt *catch* fog reagálni.

Természetesen a kivétel feldolgozása közben is keletkezhet újabb kivétel, emiatt a *catch* szakaszban is lehet *try* blokk. Van arra is lehetőség, hogy a *throw* utasítás segítségével továbbdobjuk a kivételt más, specializáltabb kivétel-feldolgozóknak. Ez akkor is hasznos lehet, ha a fejlesztői környezetben a hiba feldolgozása mellett továbbra is szeretnénk debuggolni a hiba előfordulásakor.
A kivételkezelést kiegészítheti a *finally* kulcsszó, ami akkor is le fog futni, ha nem történt, illetve akkor is, ha történt kivételkezelés. Abban az esetben is számíthatunk rá, ha olyan kivétel keletkezett, amit nem dolgoztunk fel a *catch* blokkok valamelyikében. Ezért többre hivatott, mint ha a benne lévő utasításokat egyszerűen csak a *catch*-ek mögé írnánk. Használjuk olyan feladatokra, amelyek a sikeres lezárást, az erőforrások elengedését, vagy lehetőség szerint az adatok elmentését lehetővé teszi hiba esetén is.
A *throw* segítségével mi magunk is dobhatunk egy ismert kivételt, vagy akár készíthetünk saját kivételosztályt is.

```
try
{
```

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL
A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

```
      if (!adat.HasValue) throw new System.ArgumentException("Az
adat értéke nem lehet null!");
}
catch (System.ArgumentException e)
{
      Console.WriteLine(e.Message);
}
```

Az oktatási segédanyag a *„Mobil alkalmazásfejlesztés az informatikai tudás innovatív alkalmazásával”* című, *TÁMOP-2.2.4-11/1-2012-0055* kódszámú projekt keretében valósult meg.

MAGYARORSZÁG MEGÚJUL

A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.