Belépő a tudás közösségébe

Szakköri segédanyagok tanároknak

C# nyelv fontosabb lehetőségei

Dr. Illés Zoltán

A kiadvány „A felsőoktatásba bekerülést elősegítő készségfejlesztő és kommunikációs programok megvalósítása, valamint az MTMI szakok népszerűsítése a felsőoktatásban” (EFOP-3.4.4-16-2017-006) című pályázat keretében készült 2018-ban.

Eötvös Loránd Tudományegyetem
Informatikai Kar

C#: egy nyelv – ezernyi lehetőség

C# nyelv fontosabb lehetőségei

Felelős kiadó

ELTE Informatikai Kar
1117 Budapest, Pázmány Péter sétány 1/C.

ISBN szám

ISBN XXXXXXXX

A kiadvány „A felsőoktatásba bekerülést elősegítő készségfejlesztő és kommunikációs programok megvalósítása, valamint az MTMI szakok népszerűsítése a felsőoktatásban” (EFOP-3.4.4-16-2017-006) című pályázat keretében készült 2018-ban

1.Bevezetés, a C# nyelv fejlődése

Mai rohanó világunkban a gyors gazdasági, társadalmi változások mellett is szembeötlő, mennyire gyors, milyen dinamikus a változás az informatikában.

Nem telik el úgy hónap, hogy az informatika valamelyik területén be ne jelentenének valamilyen újdonságot, legyen az hardver vagy szoftver.

Új nyelvek, új környezetek jelennek meg, ami állandó az a változás! A nyelveket, szoftver fejlesztési környezeteket tekintve az utóbbi időkben a sok változás mellett stabil és az egyik legnépszerűbb, legsokoldalúbb környezetét jelenti a Visual Studio (VS) fejlesztőrendszer a C# programozási nyelvvel.

A VS részeként több klasszikus fejlesztési nyelv mellett, mint a Basic vagy C++ nyelvek, 2002-ben megjelent az új C# (ejtsd: angolosan „szí sárp") programozási nyelv.

A nyelv mottójának talán az tekinthető, ötvözzük a Basic egyszerűségét a C++ hatékonyságával!

Ebben a tananyagban a fejlesztési eszköztárnak az alapját, a C# programozási nyelvet, annak alapvető lehetőségeit ismertetjük.

Egy következő kötetben segíteni szeretnénk az informatikus tanár kollégákat abban, hogy akár az iskolában a diákjaiknak is feladható, gyakorlatias példákon keresztül mélyedhessenek el a C# nyelv rejtelmeiben.

Megjegyzés

A megoldásokat Visual Studio 2017 Enterprise (verzió 15.5.4.) segítségével készítettük, 4.7 .Net Framework-öt használva. Ebben az anyagban nem használjuk ki ennek a kiadásnak lehetőségeit, az ingyenesen letölthető Visual Studio 2017 Community kiadás használata nem jelent semmilyen korlátozást!

Remélem, hogy ez a tankönyv széles olvasótábornak fog hasznos információkat nyújtani. A programozással most ismerkedőknek egy új, hatékony világot mutat meg, míg a programozásban már jártas Olvasók talán ennek a könyvnek a segítségével megérzik azt, hogy ez a nyelv az igazi.

Befejezésül remélem, hogy a magyarázatokhoz mellékelt példaprogramok jól szolgálják a tanulási folyamatot, és az anyag szerkesztése folytán nem kerültek bele „nemkívánatos elemek".

1.1. A nyelv rövid története, fejlődése

A C# programozási nyelv a Microsoft új fejlesztési környezetével, a 2002-ben megjelent Visual Studio.NET programcsomaggal, annak részeként jelent meg.

Bár a nyelv hosszú múlttal nem rendelkezik, mindenképpen elődjének tekinthetjük a C++ nyelvet, a nyelv szintaktikáját, szerkezeti felépítését.

A C, C++ nyelvekben készült alkalmazások elkészítéséhez gyakran hosszabb fejlesztési időre volt szükség, mint más nyelvekben, például a MS Visual Basic esetén. A C, C++ nyelv komplexitása, a fejlesztések hosszabb időciklusa azt eredményezte, hogy a C, C++ programozók olyan nyelvet keressenek, amelyik jobb produktivitást eredményez, ugyanakkor megtartja a C, C++ hatékonyságát.

Erre a problémára az ideális megoldás a C# programozási nyelv. A C# egy modern objektumorientált nyelv, kényelmes és gyors lehetőséget biztosítva ahhoz, hogy .NET keretrendszerben alkalmazásokat készítsünk, legyen az akár számolás, akár kommunikációs vagy webes alkalmazás.

A nyelv 2002-es megjelenése óta gyakorlatilag a Visual Studio-val,.NET keretrendszerrel együtt több kiadást is megért. Minden egyes kiadásban jellemzően bővültek mind a környezet, mind a C# nyelv lehetőségei, a jelen VS 2017 környezet például a C# nyelv 7. verzióját tartalmazza.

Megjegyzés

Ez a leírás nem törekszik a teljes környezet, nyelv, fejlődés részletes tárgyalására, arra talán az egyik leghitelesebb lehetőség a https://docs.microsoft.com/en-us/dotnet/csharp/ oldalról indulva érhető el.

1.2. A nyelv fontosabb jellemzői

A C# az új .NET keretrendszer bázisnyelve. Tipikusan ehhez a keretrendszerhez tervezték, nem véletlen, hogy a szabványosítási azonosítójuk (CLI-ECMA-335, C#- ECMA-334) is csak egy számmal tér el egymástól.

A C# nyelv legfontosabb elemei, jellemzői a következők:

- Professzionális, Neumann-elvű. Nagy programok, akár rendszerprogramok írására alkalmas.
- A program több fordítási egységből – modulból – vagy fájlból áll. Minden egyes modulnak vagy fájlnak azonos a szerkezete. Névterek, osztályok (partial) nem feltétlenül csak egy állományban helyezkedhetnek el.
- Egy sorba több utasítás is írható. Az utasítások lezáró jele a pontosvessző (;). Minden változót deklarálni kell. Változók, függvények elnevezésében az ékezetes karakterek használhatóak, a kis- és nagybetűk különbözőek.
- A keretrendszer fordítási parancsa parancssorból is egyszerűen használható. (pl. *csc /out:alma.exe alma.cs*).
- Minden utasítás helyére összetett utasítás (blokk) írható. Az összetett utasítást a kapcsos zárójelek közé *{}* írt utasítások definiálják.
- Objektumorientált, egyszeres öröklés, interface-ek használata.
- Nincs mutatóhasználat; biztonságos a tömb (vektor) használata, delegáltak, események.
- Érték, referencia (*ref*) és output (*out*) függvényparaméterek.
- Destruktor definiálható, a keretrendszer szemétgyűjtési algoritmusa hívja meg ha van. Szükség esetén az IDisposable interface *Dispose* metódusa újradefiniálható.
- Operátorok definiálásának lehetősége, *property*, *indexer* definiálás, aszimmetrikus elérés.
- Extension metódusok (függvénykiterjesztés), attribútumok definiálása.
- Kivételkezelés, Párhuzamos végrehajtású szálak, aszinkron metódusok definiálhatósága.
- Generic (típusparaméter, ko-,kontra variáns paraméterek),Lambda kifejezések használata.
- LINQ (Language INtegrated Query) implementálása. Segítségével nyelvi lehetőségként tudunk SQL-hez hasonló formájú lekérdezéseket végezni különböző adatforrásokon.

1.3. A Visual Studio környezet áttekintése

A Visual Studio 6.0 fejlesztőrendszer átdolgozásaként 2002-ben jelent meg a Microsoft újabb fejlesztőeszköze ami a Visual Studio.NET nevet kapta. Ettől kezdve a fejlesztőkörnyezet verziójaként a megjelenés évszáma jelent meg az elnevezésben. Az első ilyen változat volt a Visual Studio 2003, ami 2003-ban jelent meg, míg a legfrissebb verzió a Visual Studio 2017, ami 2017-ben jelent meg. A könyvben tárgyalt nyelvi jellemzők bemutatása során nem térünk ki az egyes verziók közti különbségek bemutatására, mivel annak nincs nagy jelentősége.

Az új eszköz a korábbi fejlesztőrendszerekkel ellentétben nem a hagyományosnak tekinthető ún. Win32 alapú, hanem a .NET környezet alatt futtatható alkalmazásokat készít. Ez azt jelenti, hogy az új eszközzel készült alkalmazások csak olyan operációs rendszer alatt futtathatók, melyek támogatják a .NET keretrendszert (.NET Framework). A fordító a forráskódot nem natív, hanem egy köztes kódra fordítja le. Ezt a köztes kódot MSIL (Microsoft Intermediate Language) néven szokták említeni. Ebben a keretrendszerben több nyelv is támogatott, például BASIC vagy F# nyelven is készíthetünk

alkalmazásokat. Az így lefordított programok futtatását a .NET keretrendszer Common Language Infrastructure (CLI) futtató alrendszere biztosítja. (ECMA-335)

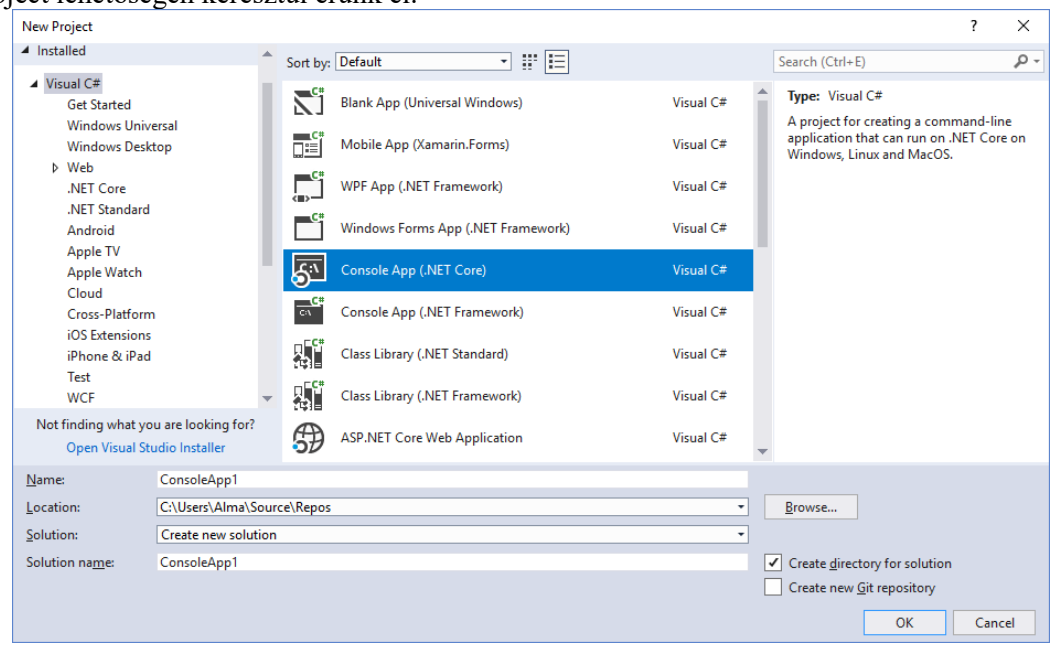
A keretrendszer a fejlesztőeszköz számára fordítási, fejlesztési idejű szolgáltatásokat végez (intellisence, szintaktikus ellenőrzés), míg az így lefordított alkalmazásoknak futási idejű környezetet biztosít (Common Language Runtime-CLR). A fordítás (Compile-Build) során elkészült EXE kód valójában menedszelt, MSIL kód, amit a CLR először helyben fordít (JIT-Just In Time compile), majd futtat. A Win32, vagy MFC alapú C++ alkalmazások kivételével menedzselt kódú fordítási eredményt kapunk.

A keretrendszer osztálykönyvtára, a telepített csomagoktól függően, több alkalmazás típus készítését segíti, néhány fontosabb a következő:

- Windows, Univerzális Windows alkalmazás, Windows Form, Console alkalmazás.
- ASP.NET, webes alkalmazások, felhő (Cloud) szolgáltatások.
- Mobil alkalmazások, kereszt platform (Xamarin).
- .NET Core alkalmazások (Linux és MacOS rendszeren is futtatható)

Az osztálykönyvtárak használatához egy fejlesztő nyelvre van szükség. A Visual Studio alapértelmezésben, szintén a telepítés függvényében, több nyelvet biztosít a fejlesztők számára, úgy mint a C#, Visual Basic, C++, F# vagy Python.

Egy feladat megoldásának indítása, természetesen a megfelelő tervezési lépések után, a fejlesztőkörnyezetben egy projekt létrehozását jelenti. Ma ez gyakorlatilag szinte minden környezetben így van. Az alábbi képen látjuk az induló projekt készítés ablakát., amit a File menüpont New Project lehetőségen keresztül érünk el.



1. ábra.

A fenti dialógus ablakban látható a projekt létrehozásának lépése. A nyelv, projekt sablon, könyvtár helyének megadása mellett látható, hogy a csoportmunkát, verzió követést támogató un. Git tárhelyet is hozzárendelhetnénk az alkalmazáshoz. Ebben a tananyagban csak a nyelvi eszközök rövid lehetőségeit mutatjuk meg, így ezen kódrészletek közvetlenül konzol alkalmazásban be is illeszthetők. Ezért az előző dialógus ablak által mutatott ConsoleApp1 projekt akár a későbbi kódrészletek kipróbálási helye is lehet.

I.4. A program szerkezete

Egy C# program tetszőleges számú fordítási egységből (modulból) állhat. Szokás ezen modulok vagy fájlok összességét projektnek nevezni.

A program ezen modulokban definiált kód (osztályok) összessége.

Egy forrásfájl tartalmazza a névtérhivatkozásokat *(using)*, névtér, osztálytípus-, változó- és függvénydefiníciókat. A névtér *(namespace)* az a logikai egység, amiben az azonosítónknak egyedinek kell lennie. Az állományokban megengedett névtéren kívül is osztályokat definiálni, de javasolt ennek kerülése.

Névtér szerkezete:

```
namespace új_névtérnév
{
        class új_osztálynév
        {
                Típusdefiníció;
                Függvénydefiníció;
        }
…
}
```

Függvények definíciójának formája:

```
visszaadott_típus név(argumentumlista, ha van)
{
        változó definíciók,
        deklarációk
        és utasítások
}
```

Az osztályok egyikében kell egy *Main* nevű függvénynek szerepelnie, amely futtatáskor az operációs rendszertől a vezérlést megkapja. A nyelv megengedi, hogy több típusban (osztályban) is definiáljunk ilyen függvényt. Ebben az esetben fordításkor kell megmondani, hogy melyik típus *Main* függvénye legyen a főprogram.

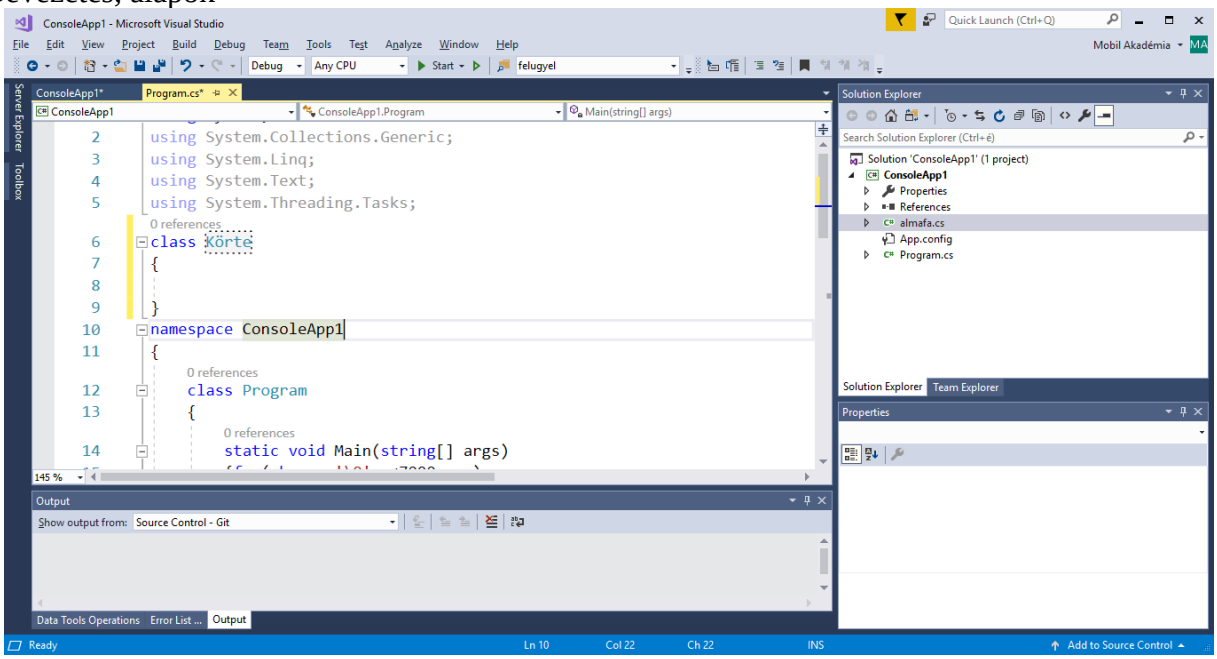
A programban használt neveknek betűvel vagy _ jellel kell kezdődniük, majd a második karaktertől tetszőleges betű és szám kombinációja állhat. A nyelvben a kis- és nagybetűk különbözőek, így például a következő két név sem azonos:

```
alma
aLma
```

Az azonosítónevek hossza általában 32 karakter lehet, de sok rendszerben az igényeknek megfelelően lehet ezen változtatni.

A .NET keretrendszer 16 bites unikód karakterábrázolást használ, amiben a magyar ékezetes karakterek is benne vannak. A nyelvi fordító ezeket az ékezetes betűket is megengedi, így az alábbi név (változó, típus, függvény) is megengedett:

```
körte
```



2. ábra

A 2. ábrán látható, hogy ha meghagyjuk a kezdeti javaslatokat (1.ábra), akkor milyen munkafelülettel indul a Visual Studio keretrendszer. Mivel nem változtattunk a javasolt ConsoleApp1 elnevezésen, így ilyen nevű lett a megoldás(Solution), a projekt sőt a névtér is!

Bár nem kötődik a C# nyelvhez, de jellemző típus, változó elnevezési konvenció, hogy az összetett, helyközt tartalmazó kifejezésekből úgy készítünk változó nevet, hogy a helyközöket elhagyjuk, és a szavakat nagy kezdőbetűvel írjuk. Az OsztályElnevezésSzavanként**NagyBetűvel**, míg az egyszerű változóElnevezésKisKezdőBetűvel kezdődik. Ezt CamelCase formának is szokták nevezni.

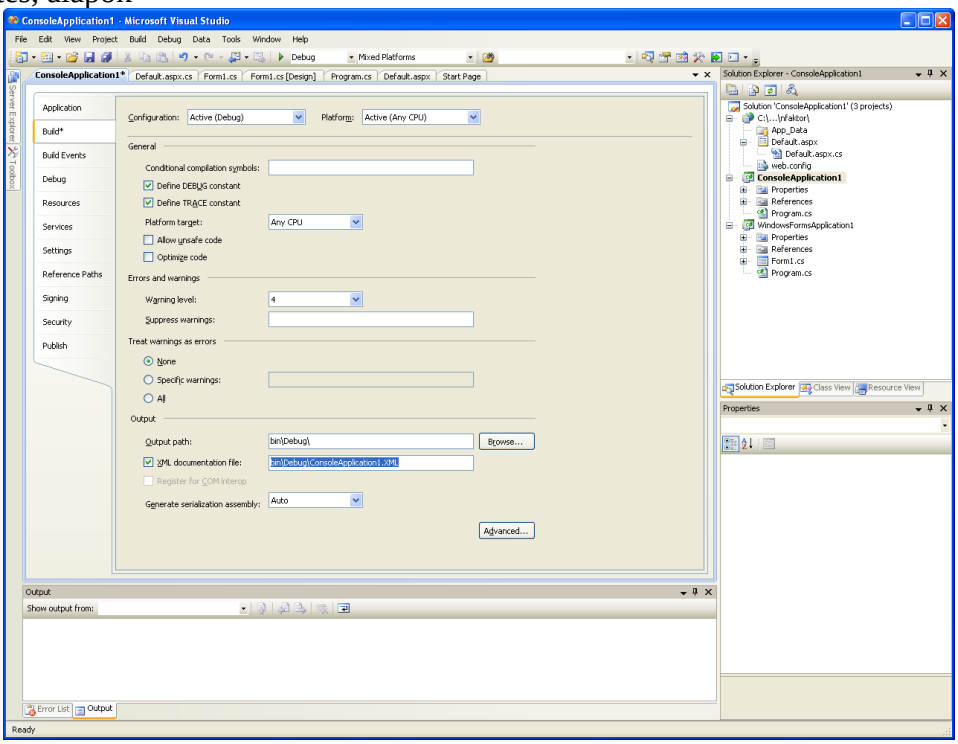
Azonosítók használatára nem megengedettek a C#-ban a nyelv által használt kulcsszavai vagy azonosítói, például ***for, while, operator***, stb. A teljes lista megtalálható a nyelv referencia leírásaiban.

A nyelvben található néhány speciális kulcsszó, melyeket lehet használni akár változónévnek is (de nem tanácsolt), de bizonyos helyzetekben nyelvi kulcsszóként értelmezhetők. Például definiálhatok egy *partial* nevű egész típusú változót, de osztálydefiníció előtt használva a *partial* kulcsszót, azt jelöljük, hogy az osztálynak csak egy része van az aktuális helyen kifejtve. A másik rész egy másik fájlban van.

Egy forrásállomány szerkesztése során magyarázatokat is elhelyezhetünk a /* és */ jelek között. Az ilyen megjegyzés több soron át is tarthat. Egy soron belül a // jel után írhatunk magyarázatot.

A mai fejlesztőkörnyezetekben egyáltalán nem ritka, hogy valamilyen speciális megjegyzést arra használjanak fel, hogy ennek a programszövegből történő kigyűjtésével a forrásállománynak vagy magának a programnak egy dokumentációját kapják. Ezt a kigyűjtést a fordító végzi el.

A C# fordítónak ha a */doc:fájlnév* formában adunk egy fordítási paramétert, akkor /// karakterek utáni információkat XML fájlba (a megadott névvel) kigyűjti. Ha nem parancssori fordítót használunk, ami a Visual Studio környezet használatakor gyakran (majdnem mindig) előfordul, akkor ezt a beállítást a projekt menüpont Tulajdonságok ablakában az *XML Documentation File* paraméter értékeként állíthatjuk be.



1. ábra

Ha a keretrendszerben egy változó vagy függvény definíciója elé beszúrjuk a /// karaktereket, akkor azt a szövegszerkesztő automatikusan kiegészíti a következő formában:

Példa:

```
    class Program
    {
        /// <summary>
        /// Osztály adatmező.
        /// </summary>
        int a;
        /// <summary>
        /// maxhal függvény, kiválasztja a paraméter tömb
        /// legnagyobb értékét
        /// </summary>
        /// <param name="h">
        /// h a paraméter tömb
        /// </param>
        /// <returns></returns>
        static public int maxhal(int[] h)
        {
            int db = h.Length;
            int max=h[0];
            for (int i = 1; i < db; i++)
                if (h[i] > max) max = h[i];
            return max;
        }
    }
```

Az előbbi példakódhoz a beállításoknak megfelelően az alábbi XML részlet készül el.

Példa:

```
<?xml version="1.0" ?>
- <doc>
- <assembly>
<name>ConsoleApp1</name>
```

```
  </assembly>
- <members>
- <member name="F:ConsoleApplication1.Program.a">
  <summary>Osztály adatmező.</summary>
  </member>
-<member name="M:ConsoleApplication1.Program.maxhal(System.Int32[])">
  <summary>maxhal függvény, kiválasztja a paraméter tömb legnagyobb értékét</summary>
  <param name="h">h a paraméter tömb</param>
  <returns />
  </member>
  </members>
  </doc>
```

A /** ....*/ forma is megengedett, szintén dokumentációs célokra. Mivel az előző módszer nem gyűjti ki az ilyen formájú megjegyzést, nem is használják gyakran.

A /// jel után megadott megjegyzéseket, paraméterleírásokat nem csak mi olvashatjuk el, hanem a fejlesztési környezet is, és azt az aktuális hívás írásánál kis sárga ablakban segítségül kiírja a beírt sorhoz.

Ezek alapján nézzük meg a szokásosnak nevezhető bevezető programunk forráskódját.

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

namespace ConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hajrá Fradi!");
        }
    }
}
```

Programunk rövid magyarázataként annyit mondhatunk, hogy a forráskódunk a legegyszerűbb esetben egyetlen állományból áll, a projekt neve lesz a névtér neve (ConsoleApp1), és a névtérben kapunk egy *Program* osztályt. Ez a kód egy Program.cs állománynév alatt található. (Természetesen átírhatjuk az osztály és a fájl nevét is!)

Ebben az osztályban egyetlen függvényt definiálunk, a programot jelentő Main függvényt. Mivel egyetlen osztálytípus egyetlen példánya sem létezik a létrehozásig (a definíció még nem létrehozás!), ezért ahhoz, hogy a függvényünk példány nélkül is létezzen, a *static* jelzővel kell a hozzáférési szintet meghatározni.

A C stílusú nyelvek hagyományaként a C# nyelvnek sem részei a beolvasó és kiíró utasítások, így könyvtári szolgáltatásokra kell hagyatkoznunk, ami a *System* névtér része, és ennek a névtérnek a *Console* osztálya biztosítja a klasszikus képernyőre írás, *Console.WriteLine()* és billentyűzetolvasás, *Console.ReadLine()* feladatát. A using utasítás(ok) a programunk számára szükséges azon névtereket adják meg, melyekben automatikusan keresni kell a hivatkozott szolgáltatásokat. A fenti első példakódban négy névtérhivatkozást látunk (az üres konzolalkalmazás projekt kódjához ez automatikusan hozzáadódik indításkor). Ezen névterekből csak az elsőt használjuk ki (System), hiszen az ebben található Console osztályt használjuk. A maradék három hivatkozást akár ki is törölhetnénk, hiszen nem használjuk ki semmilyen szolgáltatását. Egy szolgáltatást teljes névvel is elérhetünk (System.Console.WriteLine), ebben az esetben a névterek használatára vonatkozó hivatkozásunkat akár el is hagyhatnánk a forrásfájl elejéről.

I.5. Parancssori környezet használata

A tetszőleges szövegszerkesztővel megírt forráskódot például mentsük egy *alma.cs* (a nyelv a *cs* kiterjesztést szereti…) fájlba, majd az alábbi utasítással fordíthatjuk le:

```
csc alma.cs
```

A fordítás eredménye egy *alma.exe* állomány lesz, amit futtatva a képernyőn láthatjuk az eredményt.

A parancssori fordítónak a */?* paraméterrel történő futtatása, mint egy segítség funkció, megadja a fordító fontosabb kapcsolóit.

Ezek közül a legfontosabbak a következők:

/t:exe alapértelmezés, exe állományt fordít.

/t:library a fordítandó állomány könyvtár (dll) lesz.

/out:név a fordítás eredményének nevét határozhatjuk meg, alapértelmezésben ez a név megegyezik a fordított fájl nevével.

/r:valami.dll egy könyvtár felhasználása fordításkor, ha több könyvtári állományt kell hozzáfordítani, akkor az állománynevek közé vesszőt kell tenni.

/main:osztálynév a nyelv megengedi, hogy több osztály is tartalmazzon *Main* függvényt, ekkor ezzel a kapcsolóval mondhatjuk meg, hogy a sok *Main* függvény közül melyik legyen a „főprogram”.

A keletkezett exe állományról el kell mondani, hogy az nem natív bináris kód(processzor szintű utasításokat tartalmaz), hanem menedzselt kód aminek futtatásához szükséges az a .NET keretrendszer!

Általában elmondható, hogy az új fejlesztőkörnyezet minden egyes nyelvi eszköze olyan kódot fordít, aminek szüksége van erre a keretrendszerre. Természetesen, mint az élet minden területén, úgy itt is vannak kivételek. Ilyen kivétel például a C++ nyelv, ahol alapértelmezés szerint meg kell mondani, hogy a programunkat menedzselt vagy natív kódra fordítsa-e a fordítónk.

II. A C# nyelv alaptípusai

Egy programozási nyelvben az egyik legfontosabb tulajdonság az, hogy programkészítés során hogyan és milyen típusú adatokat használhatunk. Meg kell jegyezni, hogy van néhány olyan programozási nyelv is (pl. PhP), ahol ez a terület nem igazán fontos, típusok gyakorlatilag nincsenek, adatot szükség szerinti típusban a programozó rendelkezésére bocsát.

A C# szigorúan típusos nyelv, ebben a nyelvben csak ennek figyelembevételével használhatunk saját változókat.

II.1. Változók definiálása

A nyelvben a változók definiálásának alakja:

típus változónév ;

Azonos típusú változókat egymástól vesszővel elválasztva definiálhatunk. A típusok ismerete nélkül nézzünk néhány példát változók definiálására.

Példa:

```
char ch;              // ch változó karakter típusú
int egy, tizenegy;    // az egy és tizenegy egész típusú
```

Változók lehetnek külsők, azaz függvényen kívüliek, vagy belsők, azaz függvényen belüliek. A függvényen kívüli változók is természetesen csak osztályon belüliek lehetnek. Gyakran hívják ezeket a változókat adatmezőknek is.

Belső változók, az adott blokkon (függvényen) belüli lokális változók lehetnek dinamikus vagy statikus élettartamúak. Módosító jelző nélküli definiálás esetén dinamikusnak vagy automatikusnak nevezzük, s ezek élettartama a blokkban való tartózkodás idejére korlátozódik. Ha a blokk végrehajtása befejeződött, a dinamikus változók megszűnnek.

Statikus élettartamú belső változót a *static* szó használatával (ahogy külső változó esetén igen) nem definiálhatunk. Láttuk, a függvények lehetnek statikusak (lásd *Main* függvény), melyekre ugyanaz igaz, mint az osztályváltozókra, ezen függvények élettartama is a programéval egyezik meg, más szóval ezen függvények a program indulásakor jönnek létre.

A változókat két kategóriába sorolhatjuk, az osztálytípusú változók referencia típusúak, melyek mindig a dinamikus memóriában helyezkednek el (az irodalomban ezt gyakran *heap*-nek nevezik), míg minden más nem osztálytípusú, az úgynevezett értéktípusú (*value type*) változó. Az értéktípusú változókat szokták *stack változóknak* is nevezni.

Az értéktípusú változók kezdőértékét, az inicializálás hiányában, *0*, *false*, *null* értékre állítja be a fordító.

A változók definiálásakor rögtön elvégezhető azok direkt inicializálása is.

II.2. Állandók definiálása

Állandók definiálását a típusnév elé írt *const* típusmódosító szócska segítségével tehetjük meg. A definíció alakja:

const típus név = érték;

Példa:

```
const float g=9.81;    // valós konstans
g=2.3;                 // !!! HIBA
…
const int min=0;       // egész konstans
```

II.3. Változók inicializálása

Változók kezdő értékadása, inicializálása azok definiálásakor is elvégezhető a következő formában:

típus változó = érték;

Példa:

```
char s='a'; int []a={1,2,3};
char[] b="Egy";
```

A változók, konstansok és függvények használatára nézzük a következő példát.

Példa:

```
// A valtozo.cs fájl tartalma:
using System;
// A kiíráshoz szükséges deklarációt tartalmazza.
class változó
{
     static int alma=5;              //alma változó definíciója
                                     //és inicializálása
     static float r=5.6F             //statikus valós típusú változó
                                     //valós konstanst zárhat az
                                     //F (float) betű
     const float pi=3.1415;          //konstans definíció

     static void Main()
     {
          Console.WriteLine( "Teszt");       //eredmény Test
          int alma=6;                        //helyi változó
          Console.WriteLine( alma ) ;        //eredmény 6
          Console.WriteLine( változó.alma ); //a külső takart (5)
                                             //változóra hivatkozás
          Console.WriteLine( terület(r));    //a terület függvény
                                             // meghívása
     }

     static float terület(float sugár)       // függvénydefiníció
     {
          return(pi*sugár*sugár);
     }
}
```

Bár még nem definiáltunk függvényeket, így talán korainak tűnhet ez a példa, de inkább felhívnám még egyszer a figyelmet az ékezetes karakterek használatára. Egy osztályon belül a függvények definiálási sorrendje nem lényeges. Sok környezetben furcsa lehet amit a fenti példában látunk, hogy előbb használjuk, és csak ezután definiáljuk a függvényünket. (terület függvény)

II.4. Elemi típusok

***char** – karakter típus (2 byte hosszú)*

A karakter típus 16 bites karakterábrázolást (unikód) használ. Általában igaz, hogy minden alaptípus mérete rögzített.

A karakter típusú változó értékét aposztróf (') jelek között tudjuk megadni.

Példa:

```
char c;
c='a';
```

A backslash (\) karakter speciális jelentéssel bír. Az utána következő karakter(eke)t, mint egy escape szekvenciát dolgozza föl a fordító. Az így használható escape szekvenciák a következők:

\a	-	a 7-es kódú csipogás
\b	-	backspace, előző karakter törlése
\f	-	formfeed, soremelés karakter
\r	-	kocsi vissza karakter
\n	-	új sor karakter (soremelés+kocsi vissza)

Az új sor karakter hatása azonos a formfeed és a kocsi vissza karakterek hatásával.

\t	-	tabulátor karakter
\v	-	függőleges tabulátor
\\	-	backslash karakter
\'	-	aposztróf
\"	-	idézőjel
\?	-	kérdőjel
\uxxyy		xx és yy unikódú karakter

***string** – karaktersorozat*

A *System.String* osztálynak megfelelő típus. Szövegek között a + és a += szövegösszefűzést végző operátorok ismertek. A [] operátor a szöveg adott indexű karakterét adja meg. Az indexelés 0-val kezdődik. A @ karakter kiküszöböli a szövegen belüli „érzékeny” karakterek hatását. Ilyen például a backslash (\) karakter.

Példa:

```
char c='\u001b';           // c= a 27-es kódú (Esc) karakterrel
string s="alma";
string n="alma"+"barack";
s+="fa";                   //s= almafa
char c1=s[2];              // c1= 'm'
char c2="szia"[1];         // c2='z'
string f="c:\\programok\\alma.txt";
string file=@"c:\programok\alma.txt";
```

A *String* osztály a fenti lehetőségeken kívül egy sor függvénnyel teszi használhatóbbá ezt a típust. Ezen függvények közül a legfontosabbak:

Length

Csak olvasható tulajdonság, megadja a szöveg karaktereinek a számát:

```
string s="alma";
int i=s.Length;        //i=4
```

CompareTo(string)

Két szöveg összehasonlítását végzi. Ha az eredmény *0*, akkor azonos a két szöveg:

```
string str1="egyik", str2="másik";
int cmpVal = str1.CompareTo(str2);
if (cmpVal == 0)      // az értékek azonosak
     {…}
else if (cmpVal > 0)  // str1 nagyobb mint str2
     {…}
else                  // str2 nagyobb mint str1
```

Equals(string)

Megadja, hogy a paraméterül kapott szöveg azonos-e az eredeti szöveggel:

```
string str1="egyik", str2="másik";
if (str1.Equals(str2){…}   // azonos
else{…}                    // nem azonos
```

IndexOf(string)

Több alakja van, megadja, hogy az eredetiben melyik indexnél található meg először a paraméterül kapott szöveg. A visszaadott érték *–1* lesz, ha nincs benn a keresett szöveg:

```
int i="almafa".IndexOf("fa");   //4
```

Insert(int,string)

A második paraméterül kapott szöveget, az első paraméterrel megadott indextől beszúrja az eredeti szövegbe:

```
string s="almafa alatt";
string ujs=s.Insert(4," a ");   // alma a fa alatt
```

A szövegtípus további szolgáltatásait, függvényeit a szövegosztály (*System.String*) online dokumentációjában érdemes megnézni. Meg kell említeni még azt, hogy a szövegosztály függvényei nem módosítják az eredeti szöveget, a szövegobjektumot, így az előző *s* szöveges változó értéke változatlan marad. Ha magát a szövegobjektumot is módosítani akarom, például az adott szöveg egy karakterét kicserélni egy másikra, akkor a string típus helyett például a StringBuilder osztályt kell választani.

Példa:

```
string s="alma";
s[2]='f';          // hiba, s string nem módosítható
StringBuilder sb = new StringBuilder("alma");
                   //a StringBuilder konstruktorát kell meghívni
                   // az sb="alma"; utasítás hibás!!!
sb[2] = 'f';
sb.Insert(4, " űrbázis");
Console.WriteLine(sb); // eredmény: alfa űrbázis
```

Szöveges feldolgozási feladatok során a reguláris kifejezésekkel megadható szövegminták segítségét is igénybe vehetjük. A .NET Framework a Perl5 kompatibilis reguláris kifejezéshasználatot támogatja. A *Regex* osztály szolgáltatja a reguláris kifejezést, míg a *Match* a találati eredményt. Az osztályokat a *System.Text.RegularExpressions* névtérben találjuk.

Példa:

```
using System;
using System.Text.RegularExpressions;
class reguláris
```

```
{
    public static void Main()
    {
        Regex reg_kif = new Regex("fradi");
        // a fradi keresése
        Match találat = reg_kif.Match("A Fradi jó csapat?");
        if (találat.Success)
        {
            // (hol találtuk meg
            Console.WriteLine("A találat helye: " + találat.Index);
        }
    }
}
```

A fenti példa nem ad eredményt, hiszen alapértelmezésben a kis- és nagybetű különbözik.

***int** – egész típus(4 byte)*

Az egész típusú értékek négy bájtot foglalnak – a ma leggyakrabban használt nyelvi környezetben –, így értelmezési tartományuk -2^{31}, $2^{31}- 1$ között van.

long	hosszú egész (8 byte)
short	rövid egész (2 byte)
sbyte	előjeles (signed) byte
float	valós (4 byte)
double	dupla pontosságú valós (8 byte)
decimal	„pénzügybeli” típus (16 byte), 28 helyiérték

Mindkét egész típus előjeles, ha erre nincs szükségünk, használhatjuk az előjel nélküli változatukat, melyek: *uint, ushort, byte, ulong.*

Valós számok definiálásakor a 10-es kitevőt jelölő *e* konstans használható.

Példa:

```
float a=1.6e-3;   // 1.6 * 10^-3
```

***void** – típus nélküli típus*

Az olyan függvénynek (eljárásnak) a típusa, amelyik nem ad vissza értéket.

Példa:

```
void növel(ref int mit)
{
    mit++;
}
```

Az iménti függvény paraméterének jelölése referencia szerinti paraméter-átadást jelent, amiről a *Függvények* c. fejezetben részletesen szólunk.

***bool** – logikai típus (1 byte)*

A logikai típus értéke a *true* (igaz) vagy *false* (hamis) értéket veheti fel. Ahogy a nyelv foglalt alapszavainál láthattuk, ezeket az értékeket a *true* és *false* nyelvi kulcsszó adja meg.

Általában elmondható, hogy míg a nyelv nem definiál sok alaptípust, addig az egyes implementációk, főleg azok grafikus felület alatti könyvtárai ezt bőségesen pótolják, és gyakran több időt kell az 'új típusok' megfejtésének szentelni, mint a függvények tanulmányozásának.

Az alaptípusok valójában a System névtér megfelelő struktúráinak felelnek meg. Például a *double* valós szám egy System.Double típusnak, míg az *int* egész egy System.Int32 típusnak felel meg.

II.5. Felsorolás típus

A felsorolás típus gyakorlatilag nem más, mint egész konstans(ok), szinonimák definiálása. Felsorolás típust névtéren vagy osztályon belül definiálhatunk.

Ennek a kulcsszava az *enum*. A kulcsszó után a felsorolás típus azonosítóját meg kell adni. A *System.Enum* osztály szinonimáját adja az ezen kulcsszó segítségével definiált típus.

Példa:

```
enum szinek {piros,kék,zöld,sárga};
enum betuk {a='a', b='b'};
betuk sajátbetű=betuk.a;        // változó kezdőértéke a
enum valami { x="alma"};        // hiba
```

Ekkor a 0,1, … értékek rendelődnek hozzá a felsorolt nevekhez, és a nevek kiírásakor ezen számokat is látjuk. A kezdőértékadás lehetőségével élve nem kell ezeket feltétlenül elfogadnunk, hanem mi is megadhatjuk az értéküket.

Példa:

```
class Szinek
{
    enum újszinek {piros=4, kék=7, zöld=8, sárga=12};

    public static void Main()
    {
        Console.WriteLine(újszinek.zöld); // kiírja a színt
        //ciklus a színeken történő végighaladásra
        for(újszinek i=újszinek.kék; i<újszinek.sárga; i++)
        {
            //kiírja a színeket
            Console.WriteLine(i);
        }
    }
}
```

A fenti ciklus eredményeként azon egészek esetén, ahol az egész értékéhez egy „nevet” rendeltünk hozzá, a név kerül kiírásra, egyébként a szám. Az eredmény a következő lesz:

```
kék
zöld
9
10
11
```

Mivel ez a típus a *System.Enum* megfelelője, ezért rendelkezik annak jellemzőivel is. Ezek közül a két legjellemzőbb a *GetHashCode()* és a *ToString()* függvény. Az előbbi a belső tárolási formát, a megfelelő egész számot adja meg, míg a *ToString()*, mint alapértelmezett reprezentáció, a szöveges alakot (kék,zöld, stb) adja meg.

Példa:

```
újszinek s=újszinek.piros;
Console.WriteLine(s.GetHashCode());      // eredmény: 4
```

A felsorolás típus alapértelmezésben egész típusú értékeket vesz fel. Ha ez nem megfelelő, akár *byte* (vagy tetszőleges egész szinonima) típusú értékeket is felvehet úgy, hogy a típusnév után kettősponttal elválasztva megadom a típusnevet. (Nem adhatom meg a valós vagy karakter típust, az enum egész szám alapú!)

Példa:

```
enum színek:byte {piros,kék,zöld,sárga};
…
```

```
enum hibás:float {rossz1, rossz2};       // ez fordítási hiba
```

Előfordulhat, hogy olyan felsorolásadatokra van szükségem, melyek nem egy egész konstanshoz, hanem egy bithez kötődnek, hiszen ekkor a logikai és/vagy műveletekkel a felsorolásadatok kombinációját határozhatjuk meg. Ebben az esetben a *[Flags]* attribútumot kell az *enum* szócska elé szúrni.

Példa:

```
[Flags] enum alma
{piros=1,jonatán=2,zöld=4,golden=8};
…
alma a=alma.piros | alma.jonatán;
Console.WriteLine(a);      // piros, jonatán
a=(alma) System.Enum.Parse(typeof(alma),"zöld,golden");
// a felsorolás típus egyszerű értékadása helyett
// a könyvtári hívás lehetőségét is használhatjuk
//
Console.WriteLine(a.GetHashCode());
…
// eredmény 12 lesz
```

Bithez kötött értékek esetén kötelező minden egyes konstans névhez megadni a neki megfelelő bites ábrázolást!

Az alaptípusokat a .NET keretrendszer biztosítja, így ennek a fejlesztési környezetnek egy másik nyelvében pontosan ezek a típusok állnak rendelkezésre. Az természetesen előfordulhat, hogy nem ezekkel a nevekkel kell rájuk hivatkozni, de a nyelvi fordító biztosan ezen értelmezés szerinti kódot (köztes kód) fordítja le a keretrendszer, mint futtató környezet számára.

Az alaptípusok zárásaként meg kell jegyezni, hogy a mutató típus is értelmezett, ahogy a C++ világában, de ennek a használata csak olyan C# programrészben megengedett, amely nem biztonságos környezetben helyezkedik el *(unsafe context)*. Erről röviden a könyv XIV. fejezetében olvashat.

II.6. Null értéket felvevő típusok

A fejezet elején említettük, hogy a keretrendszerben megkülönböztetünk érték (value) illetve referencia típusokat.

A referencia típusok esetében alapértelmezés, hogy amikor nincs értékük (nem hivatkoznak egyetlen címre sem), akkor ezt a *null* értékkel (sehova nem mutat) jelöljük. Ebben az esetben tehát a változók értéke (például string típus esetén) alapértelmezésként lehet *null*.

Érték típusok esetén (int, float,…) a *null* azzal a jelentéssel bír, hogy nincs még értéke, nincs inicializálva. Ezen *nullable* típusok abban segítenek, hogy adattáblák (ahol a *null* mezőérték megengedett) típusillesztésénél nincs nyelvi probléma.

Ennek az értelmezésnek a nyelvi megjelenését úgy jelöljük, hogy egy kérdőjelet írunk a típusnév után. Ez valójában a Nullable<*típus*>, amit rövidítve jelölünk *típus?* formában.

Példa:

```
char? c;          // c karakter null értéket felvevő típus
string? s="alma"; // fordítási hiba, mert a string
                  // maga is referencia típusú!!!
int? n=null;      // a null értékkel inicializáljuk n-t
                  // int? a Nullable<int> rövidítése
```

A null értéket felvehető típusnak van két hasznos metódusa. A HasValue metódus logikai értéket ad vissza annak megfelelően, hogy null vagy nem null a változó értéke. A Value metódus a változó értékét adja vissza ha van, ha nincs akkor pedig kivételt dob.

II.7. Implicit típusok

Korábban említettük, hogy a C# nyelv szigorúan típusos. Ez a programozási nyelvekben azt szokta jelenteni, hogy egy változó definíciója tartalmazza annak típusjelzését is. Az implicit típusok használatakor igazából egy kényelmi szolgáltatásról beszélünk. A var kulcsszó használatakor nem kell kiírni a változó típusát, azt a fordító a jobboldali inicializáló kifejezésből kitalálja.

Példa:

```
var i= 1;          // i egész típusú lesz
var f= 2.4;        // f valós szám lesz
var s= "alma";     // s string típusú lesz
var v= new [] {1,2,3,4,5};// v egész tömb lesz
var f= "almafa";  // hiba, f már definiált
```

A var kulcsszó segítségével **csak lokális változókat** definiálhatunk! A for, foreach, using utasításoknál is használható. A nyelv implicit típusmeghatározási segítsége, majd látni fogjuk a LINQ kapcsán (X. fejezet), hogy nagyon hasznos tud lenni, de az egyszerű egész, valós, karakter vagy string típusok esetén én azt tudom tanácsolni, hogy inkább ne használjuk. A kis példasorok írásakor is, hacsak lehet, kerülni fogjuk a használatát.

II.8. Nem nevesített (anonymous) típusok

Az implicit típusokhoz hasonlóan azt mondhatjuk, hogy ez a lehetőség is elsősorban a tömörebb írásmód használatában van segítségünkre. Ha nincs szükségünk a program során konkrét névvel adott összetett típusra (lásd IV. fejezet), akkor definiálhatunk egy olyan konkréten meg nem adott típusú (implicit) változót, amelyet egy saját összetett konstrukció eredményeként kapunk.

Példa:

```
var anonim = new {név=”Zoli”, kor=25};
// anonim összetett anonymous típusú lesz, két mezővel
Console.WriteLine(anonim.név);      // Zoli
```

Talán furcsálljuk ezen két utóbbi lehetőséget (implicit, anonymous típusok), és azt kérdezhetjük, hogy miért kell ezeket a lehetőségeket bevezetve „feláldozni az igazi nyelvi típusosságot”? Erre azt válaszolhatjuk, hogy egyrészt a típusosság megmarad, másrészt az így kapott egyszerűbb írásmód a LINQ használatánál jóval könnyebb, átláthatóbb, egyszerűbb kódot eredményez.

III. Kifejezések, műveletek

A nyelv jellemzője a korábban (például C++ nyelvben) megismert, megszokott kifejezésfogalom, amit tömören úgy is fogalmazhatunk, hogy a vezérlési szerkezeteken kívül a nyelvben minden kifejezés.

Egy kifejezés vagy elsődleges kifejezés, vagy operátorokkal kapcsolt kifejezés.

Elsődleges kifejezések:

- azonosító
- (kifejezés)
- elsődleges kifejezés[]
- függvényhívás
- elsődleges kifejezés.azonosító

A kifejezéseket egy-, két- vagy háromoperandusú operátorokkal kapcsolhatjuk össze.

Egy kifejezésen belül először az elsődleges kifejezések, majd utána az operátorokkal kapcsolt kifejezések kerülnek kiértékelésre.

A nyelvben használható operátorok prioritási sorrendben a következők.

III.1. Egyoperandusú operátorok

new

A dinamikus helyfoglalás operátora. A helyfoglalás operátorának egyetlen operandusa az, hogy milyen típusú objektumnak foglalunk helyet. Sikeres helyfoglalás esetén a művelet eredménye a lefoglalt memória címe. Sikertelen helyfoglalás esetén a *null* mutató a visszatérési eredmény.

Példa:

```
int[]a;
a = new int;       //egy egész tárolásához
                   //elegendő hely foglalása
a = new int[5]     //öt egész számára foglal helyet
```

A *new* utasítás gyakran szerepel a tömbökkel összefüggésben, amiről később részletesen szólunk. Minden objektum a new operátor segítségével készül, valójában minden típus, így például az egyszerű egész szám definiálás is. Annak igazi alakja:

```
System.Int32 szám=new System.Int32();
// Ennek rövidített, általánosan használt alakja:
// int szám;
```

A .NET keretrendszer egyik legfontosabb tulajdonsága az, hogy a dinamikusan foglalt memóriaterületeket automatikusan visszacsatolja a felhasználható memóriatartományba, ha arra a memóriára a programnak már nincs szüksége. Sok nyelvi környezetben erre a *delete* operátor szolgál.

-	aritmetikai negálás
!	logikai negálás (not)
~	bitenkénti negálás,
++,--	inc, dec. (eggyel való növelés, csökkentés)
(típus) kifejezés	típuskényszerítés (cast)

Példa:

```
double d=2.3;
int i=(int) d;     // i=2
```

A ++ és -- operátor szerepelhet mind pre-, mind postfix formában. Prefix esetben a változó az operátorral változtatott értékét adja egy kifejezés kiértékelésekor, míg postfix esetben az eredetit.

Példa:

```
a= ++b;          // hatása: b=b+1; a=b;
a= b++;          // hatása: a=b; b=b+1;
```

Az egyoperandusú operátorok és az értékadás operátora esetén a végrehajtás jobbról balra haladva történik, minden más operátor esetén azonos precedenciánál balról jobbra.

III.2. Kétoperandusú operátorok

*	szorzás
/	osztás
%	maradékképzés

E három operátorral egészek esetén igaz, hogy az *(a/b)*b+a%b* kifejezés az *a* értékét adja.

A *%* operátor esetén az operandusok lehetnek valósak is, és a prioritásuk azonos. Ebben az esetben kivonásokat végez a fordító, amíg a kisebbítendő (a) nagyobb, mint a kivonandó (b).

Példa:

```
double a= 7.3;
double b= 2.6;
double c= a%b;          // c értéke 2.1 lesz
```

+	összeadás, string esetén azok összefűzése
–	kivonás
<<	bitenkénti balra léptetés
>>	bitenkénti jobbra léptetés

A jobbra léptetés operátorához példának tekintsük az *a >> b*; utasítást. Ekkor ha az *a* unsigned, akkor balról nullával, különben pedig az előjelbittel tölt a bitenkénti jobbra léptetés operátora.

Példa:

```
int a= –1, b;
b= a >> 2;              // b értéke -1 marad
```

<,>	kisebb, nagyobb relációs operátorok
<=, >=	kisebb vagy egyenlő, ill. a nagyobb vagy egyenlő operátorok
==, !=	egyenlő, nem egyenlő relációs operátorok
&	bitenkénti és
^	bitenkénti kizáró vagy
|	bitenkénti vagy
&&	logikai és
||	logikai vagy

III. Kifejezések, műveletek

A logikai müveletek során a C++ nyelvben gyakran elkövetett hiba, hogy elágazásban eltévesztik azt, hogy az értékadás és az egyenlőségvizsgálat különböző operátorokat jelent. A C# nyelv fordítója figyel erre az esetre, ahogy a következő példa mutatja.

Példa:

```
int a= 2;
if (a=3)          // hiba, az értékadás nem logikai művelet
```

?? alapértelmezett vagy

A ?? operátor a jelentését tekintve nem igazán új, például a Perl nyelvben is már ismert, erre a jelentésre a logikai vagy operátort (||) használja. A C# nyelvben a *null* értéket felvehető típushoz kötődik. Ha a változónak még nincs értéke, azaz *null* értéke van, akkor a jobboldali operandus legyen az értéke.

Példa:

```
int? a= 2, b=null;
int c= a ?? 5;          // c értéke 2 lesz
int d= b ?? 3;          // d értéke 3 lesz
```

A ?? operátor hatását az alábbi háromoperandusú kifejezéssel is jelölhetnénk. A háromoperandusú operátorról bővebben a következő fejezetben olvashatunk.

Példa:

```
c= a.HasValue ? a.Value: 5;
```

:: névtér (globális) alias operátor

A keretrendszer alapértelmezésben a projekt névvel egy névteret hoz létre. Nem feltétlenül szükséges minden osztályt ebben a névtérben definiálni, sőt, névtéren kívül is definiálhatok osztályokat (VII. fejezet). A névtéren kívüli osztály neve megegyezhet egy névtéren belüli osztállyal. Ekkor a global::osztálynév formában érhetjük el a külső típust.

Példa:

```
using System;

//Névtéren kívüli alma osztály
public class alma
{
    public void hello()
    {
        Console.WriteLine("Külső (globális térben lévő alma");
    }
}
namespace Foci
{
    public class alma
    {
        public void hello()
        {
            Console.WriteLine("Belső alma");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            alma al = new alma();
            al.hello();      // belső alma
```

```
            global::alma al1 = new global::alma();
            al1.hello();     // külső alma
        }
    }
```

=>

Lambda kifejezés operátora. A kifejezés delegált típust (VI.5) határoz meg, így mielőtt használnánk, definiálni kell ilyen változót is. A System.Linq névtér tartalmaz paraméter nélküli, 1,2,3 vagy 4 paraméteres delegáltat *Func* néven. Amennyiben ez megfelelő, akkor használhatjuk, hiszen a névtér alapértelmezett módon használatba van véve.

Példa:

```
using System.Linq;
…
int i;
Func<int,int> f1 = (x) => { return x+5; };
i= f1(3);
Console.WriteLine(i);  // eredmény: 8
```

A Lambda kifejezésről bővebben a VI.7. fejezetben olvashatunk.

is

Logikai operátor, egy objektumról megmondja, hogy a bal oldali operandus a jobb oldali típusnak egy változója-e.

Példa:

```
int i=3;
if (i is int) Console.WriteLine("Bizony az i egész!");
     else Console.WriteLine("Bizony az i nem egész!");
```

as

Kétoperandusú típuskényszerítés. A bal oldali változót a jobb oldali referencia típusra alakítja, ha tudja. Ha sikertelen az átalakítás, akkor eredményül a *null* értéket adja.

Példa:

```
double d=2.5;
Object o= d as Object;
Console.WriteLine(o);      // eredmény: 2.5
// Object-re mindent lehet alakítani, aminek van toString kiíró
//függvénye. Erről bővebben később esik szó.
```

typeof

Egy *System.Type* típusú objektumot készít. Ennek az objektumnak a mezőfüggvényeivel, tulajdonságaival *(FullName, GetType())* tudunk típusinformációhoz jutni.

Példa:

```
class Teszt
{
     static void Main(){
          Type[] t =    {
               typeof(int),
               typeof(string),
               typeof(double),
               typeof(void)
                         };
          for (int i = 0; i < t.Length; i++)
          {
               Console.WriteLine(t[i].FullName);
          }
     }
}
```

A program futása a következő eredményt produkálja:

```
System.Int32
System.String
System.Double
System.Void
```

A típuskonverzióval kapcsolatban elmondható, hogy minden értéktípusból létrehozhatunk egy neki megfelelő *Object* típusú objektumot. Ezt gyakran *boxing*-nak, csomagolásnak, míg az ellenkező kicsomagoló műveletet, amihez a zárójeles típuskonverzió operátort kell használni, *unboxing*-nak nevezi a szakirodalom. (Használatánál legyünk óvatosak, ha eltévesztjük a típust kivételt dob a típuskényszerítés!)

Példa:

```
int i=5;
Object o=i;       // boxing, becsomagolás
int j=(int) o;    // unboxing, kicsomagolás
```

Az *Object* típus, ami a *System.Object* típusnak felel meg, minden típus őseként tekinthető. Ennek a típusnak a függvényei ily módon minden általunk használt típushoz rendelkezésre állnak.

Az *Object* típus tagfüggvényei a következők:

ToString()

Megadja az objektum szöveges reprezentációját. Ha erre van szükség, például kiírásnál, akkor ezt automatikusan meghívja. Ha ezt egy új típushoz újradefiniáljuk, akkor ez hívódik meg kiírás esetén.

Példa:

```
Console.WriteLine(o);      // indirekt ToString hívás
Console.WriteLine(o.ToString());
```

GetType()

Megadja az objektum típusát, hasonló eredményt ad, mint a korábban látott *typeof* operátor.

Equals(object)

Megadja, hogy két objektum egyenlő-e, logikai értéket ad eredményül.

Példa:

```
int i=5;
object o=i;
Console.WriteLine(o.Equals(5));
```

Az *Equals* függvénynek létezik egy statikus változata is, aminek a formája: *Object.Equals(object, object)*

GetHashCode()

Megadja az objektum *hash* kódját, ami egy egész szám. Tetszőleges saját utódtípusban újradefiniálhatjuk, amivel a saját típusunk *hash* kódját tudjuk számolni.

III.3. Háromoperandusú operátor

? : háromoperandusú operátor

Az operátorral képezhető kifejezés alakja a következő*: e1 ? e2 : e3* , ahol a *?* és a *:* az operátor jelei, míg *e1,e2,e3* kifejezések.

A kifejezés értéke *e2* ha *e1* igaz (nem 0), különben *e3*.

Példa:

```
char c;
int a;
a=((c>='0' && c<='9') ? c-'0' : -1);
```

Az *a* változó vagy a 0-9 vagy –1 értéket kapja.

A háromoperandusú operátor valójában egy logikai elágazás utasítás. A hatékony kifejezéskészítés, tömörebb írásmód kiváló eszköze.

III.4. Kétoperandusú értékadó operátorok

= értékadás operátora

A nyelvben az értékadás sajátos, önmaga is kifejezés. Az egyenlőségjel bal oldalán olyan kifejezés, változó állhat, amely által képviselt adat új értéket vehet fel. Gyakran hívja a szakirodalom ezt balértéknek is (*lvalue*). Az egyenlőségjel jobb oldalán egy értéket adó kifejezés kell, hogy álljon (jobbérték, *rvalue*). Ez gyakorlatilag azt jelenti, hogy a jobbértékre semmilyen korlátozás nincs.

Az értékadás során a bal oldal megkapja a jobb oldali kifejezés értékét, és egyben ez lesz a kifejezés értéke is.

Példa:

```
…
char []d=new char[80]; // másoljuk az s forrásszöveget d-be
while (i<s.Length)
{
    d[i]=s[i]; i++;
}
char []c={'L','a','l','i'};
d=c;     // hiba!! mivel d már egy 80 karakteres
         //területet jelöl, ezért új területet már nem
         // jelölhet, d nem lehet balérték
```

Az értékadás operátora jobbról balra csoportosít, ezért pl. az *a=b=2;* utasítás hatására *a* és *b* is 2 lesz.

Az egyenlőségjel mellett még néhány, az aritmetikai operátorokkal 'kombinált' értékadó operátor is használható.

Ezek az operátorok a következők:

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

Jelentésük: az *e1 operátor e2* kifejezés, ahol *e1* és e2 is kifejezések, az *e1 = e1 op e2* kifejezésnek felel meg, ahol *op* az egyenlőségjel előtti operátor.

Példa:

```
int a=2;
a *= 3;       // a= a*3, azaz a értéke 6 lesz
string b="alma";
b+="fa";      // b=almafa
```

III. Kifejezések, műveletek

A típusok egymás közti konverziójával kapcsolatban azt lehet mondani, hogy a C vagy C++-ban ismert automatikus konverziók nem léteznek. Egy egész típusú változót egy valósba gond nélkül beírhatunk, míg fordítva már hibát jelez a fordító. A konverziós lehetőségek széles skáláját nyújtja a fejlesztőeszközünk.

Konverziókkal kapcsolatban két esetet szoktak megkülönböztetni. Ezek közül az első szám szöveggé alakítása. Ez gyakorlatilag nem igényel semmilyen segítséget, a számtípushoz tartozó osztály *ToString* függvényét hívja meg a fordító. Ehhez nem kell semmit sem tenni.

Példa:

```
int a=2;
string s="Az eredmény:" + a;
```

A másik eset, mikor szövegből szeretnénk számot kapni, már nem ilyen automatikus, de semmiképpen nem lehet bonyolultnak nevezni. Többféle módszer lehet az alakításra, de a legáltalánosabb talán a *Convert* osztály használata. Az osztály konvertáló függvényei azonos névkonvencióval az alábbi formájúak:

```
Convert.ToCTStípusnév
```

ahol a CTS (Common Type System) típusnév az alábbi lehet: *Boolean*, *Int16* (short int), *Int32* (rendes int), *Int64* (hosszú int), *Float*, *Double*, *String*, *Decimal*, *Byte*, *Char*.

Példa:

```
string s="25";
int i=Convert.ToInt32(s);  // i=25 lesz
```

Érdekességként megemlítem, hogy létezik a *Convert.ToDateTime(object)* függvény is, ami egy könyvtári dátumtípust készít a paraméterül kapott objektumból.

Ha bármelyik konvertáló függvény hibás paramétert kap, nem tud eredményt produkálni, akkor *InvalidCastException* kivételt vagy más, a hiba okára utaló kivételt generál. A kivételkezelésről később részletesen is fogunk beszélni. (VIII. fejezet)

Általában igaz, hogy grafikus alkalmazások során a beírt (textbox-ba) adataink szövegesek, így minden esetben az azokkal történő számolás előtt konvertálnunk kell, ezért a konvertáló függvények használata elég gyakori.

Hasonló konvertáló szolgáltatásokat kapunk, ha az alaptípusok *Parse* függvényét használjuk *(int.Parse(szöveg), double.Parse(szöveg)).*

Gyakran előfordul, hogy az alapműveletek nem elégítik ki a számolási igényeinket. A *System* névtér *Math* osztálya a leggyakrabban használt számolási műveleteket biztosítja. A leggyakrabban használt függvények a következők:

Math.Sin(x)	sin(x), ahol az x szög értékét radiánban kell megadni
Math.Cos(x)	cos(x)
Math.Tan(x)	tg(x)
Math.Exp(x)	ex
Math.Log(x)	ln(x)
Math.Sqrt(x)	x négyzetgyöke
Math.Abs(x)	x abszolút értéke
Math.Round(x)	kerekítés a matematikai szabályok szerint
Math.Ceiling(x)	felfelé kerekítés
Math.Floor(x)	lefelé kerekítés
Math.Pow(x,y)	hatványozás, x^y
Math.PI	a PI konstans (3.14159265358979323846)
Math.E	az e konstans (2.7182818284590452354)

Példa:

```
double dd=Math.Sin(Math.PI/2);
Console.WriteLine(dd);      // értéke 1.
dd=Math.Pow(2,3);
Console.WriteLine(dd);      // 8
```

Matematikai, statisztikai feladatoknál a véletlenszámok használata gyakori igény. A *System* névtér *Random* osztálya nyújtja a pszeudo-véletlenszámok generálásának lehetőségét. Egy véletlenszám-objektum létrehozását a rendszeridőhöz (paraméter nélküli konstruktor) vagy egy adott egész számhoz köthetjük. A véletlenobjektum *Next* függvénye a következő véletlen egész számot, a *NextDouble* a következő valós véletlenszámot adja. A *Next* függvénynek három, míg a *NextDouble* függvénynek egy változata van, amit a következő példa is bemutat.

Példa:

```
Random r=new Random();
//r véletlenszám objektum rendszeridő alapú létrehozása
Random r1=new Random(10);
// r1 véletlenobjektum generátor a 10 értékből indul ki
//
int v= r.Next();
// véletlen egész szám 0 és MaxInt (legnagyobb egész) között
//0 lehet az érték, MaxInt nem
//
int v1=r.Next(10);
// véletlen egész szám 0 és 10 között, 0<=v1<10
//
int v2=r.Next(10,100);
// véletlen egész szám 10 és 100 között, 10<=v2<100
//
double d=r.NextDouble();
// véletlen valós szám 0 és 1 között, 0<=d<1
```

IV. Összetett adattípusok

IV.1. Tömbök

A feladataink során gyakori igény az, hogy valamilyen típusú elemből többet szeretnénk használni. Amíg ez a „több” kettő vagy három, addig megoldás lehet, hogy két vagy három változót használunk. Amikor viszont tíz, húsz adatra van szükségünk, akkor ez a fajta megoldás nem igazán kényelmes, és sok esetben nem is kivitelezhető. Lehetőség van azonos típusú adatok egy közös névvel való összekapcsolására, és az egyes elemekre index segítségével hivatkozhatunk. Ezt az adatszerkezetet tömbnek nevezzük. A tömbök elemeinek elhelyezkedése a memóriában sorfolytonos. A tömb helyett gyakran használjuk a vektor szót, annak szinonímájaként.

A C# nyelv minden tömb- vagy vektortípus definíciót a *System.Array* osztályból származtat, így annak tulajdonságai a meghatározóak.

A tömbök definíciójának formája a következő:

típus[] név;

Az egyes tömbelemekre való hivatkozás, a tömbök indexelése mindig 0-val kezdődik. Az index egész típusú kifejezés lehet, a típus pedig tetszőleges.

Egy tömb általános definíciója nem tartalmazza az elemszám értékét. Bár a nyelv környezetében nem lehet mutatókat definiálni, de ez gyakorlatilag azt jelenti. A vektor definíciója egy referencia típus létrehozása, és a nyelv minden referencia típusát a *new* operátorral kell konkrét értékekkel inicializálni (példányosítani). Ekkor kell megmondani, hogy az adott vektor hány elemű lesz. Minden vektornak, ellentétben a C++ nyelvvel, a létrehozása után lekérdezhető az elemszáma a Length tulajdonsággal.

Példa:

```
int[] számok;              // egész vektordefiníció
számok = new int[10];      // 10 elemű lesz az egész vektorunk
                           // az index 0 és 9 között van
számok = new int[20];      // most meg 20 elemű lett
int db = 15;
számok = new int[db];      // egy változó adja a hosszát
```

Mivel a nyelv minden dinamikus referencia típusának memóriabeli felszabadítását a rendszer automatikusan végzi – ezt az irodalom gyakran szemétgyűjtési algoritmusnak (*garbage collection*) nevezi –, ezért a példabeli 10, 20 elemű tömb memóriahelyét automatikusan visszakapja az operációs rendszer.

A vektorelemek a fenti dinamikus létrehozás után 0 kezdőértéket kapnak. Ha egyéb elemekkel szeretnénk az inicializációt elvégezni, kapcsos zárójelek közé írva adhatjuk meg azokat.

típus[] név={érték, érték,…};

Példa:

```
int[] n={3,4,5,6};
int hossz= n.Length;       // vektorhossz meghatározás
```

Ha logikai vektort definiálunk, akkor a vektorelemek kezdőértékadás hiányában logikai hamis (*false*), míg ha referencia vektort definiálunk, akkor a referenciaelemek a *null* kezdőértéket kapják meg.

Példa:

```
int[] számok = new int[5] {1, 2, 3, 4, 5};
string[] nevek = new string[3] {"Ali", "Pali", "Robi"};
int[] számok1 = new int[] {1, 2, 3, 4, 5};
string[] nevek1 = new string[] {"Ali", "Pali", "Robi"};
```

A C# nyelvben nem csak a fenti egydimenziós tömb definiálható. Ezenkívül még definiálható két- vagy többindexű, multidimenziós tömb, illetve tömbök tömbje.

Multidimenziós tömb:

Példa:

```
string[,] nevek;
nevek= new string[2,4];
```

A tömb dimenzióját(indexeinek számát) a *Rank* tulajdonsággal kérdezhetjük le.

```
Console.WriteLine(nevek.Rank);           // 2
// az egyes dimenziókbeli méretet a GetLength adja
Console.WriteLine(nevek.GetLength(0));   // 2
Console.WriteLine(nevek.GetLength(1));   // 4
// a vektorhossz (Length) tulajdonság itt is elérhető
// a tömb összes elemeinek a számát adja meg
Console.WrteLine(nevek.Length);          // 8
```

Természetesen a normál vektornak *(int[] v)* is lekérdezhetjük a dimenzió értékét.

Példa:

```
int[,] számok= new int[3, 2] { {1, 2}, {3, 4}, {5, 6} };
string[,] párok = new string[2, 2] { {"Miki","Ani"}, {"Mari","Albert"} };
int[,] számok1 = new int[,] { {1, 2}, {3, 4}, {5, 6} };
int[,] számok2 = { {1, 2}, {3, 4}, {5, 6} };
string[,] párok1 = { {"Miki", "Ani"}, {"Mari", "Albert"} };
```

Használat:

```
számok[1,1] = 667;
```

Iménti példában egész és string típusú tömbök különböző inicializálására látunk példákat. Az ilyen tömbhasználat gyakori olyan esetben, mikor tömbökkel kapcsolatos függvényt kell írnunk, és a függvény teszteléséhez szükségünk van paraméterként adatokra.

Tömbök tömbje (Vektorok vektora):

A más nyelvekbeli analógiát tekintve, a multidimenziós vektorban minden sorban azonos darabszámú elem van, ez tehát a szabályos, négyzetes mátrix stb. definíciónak felel meg. A vektorok vektora pedig valójában egy mutató vektor–definíció, ahol először megmondjuk, hogy hány elemű a mutató vektor, majd ezután egyenként meghatározzuk, hogy az egyes elemek milyen vektorokat jelentenek.

Példa:

```
byte[][] meres = new byte[5][];
for (int x = 0; x < meres.Length; x++)
{
    meres[x] = new byte[4];
}
```

```
int[][] számok= new int[2][]
{
    new int[] {3,2,4},
    // ez a három elemű vektor lesz a számok első vektora
    new int[] {5,6}
    // ez a kételemű vektor lesz a számok második vektora
};

int[][] számok1 =
new int[][] { new int[] {1,2},new int[] {2,3,4},
              new int[] {5,6,7,8} };
int[][] számok2 = { new int[] {2,3,4}, new int[] {5,6,7,8,9} };
```

Használat:

```
számok[1][1] = 5; // az eredeti 6 értéket 5-re átírjuk
```

Természetesen mindkét esetben – multidimenziós, vektorok vektora – nem–csak kétdimenziós esetről beszélhetünk, hanem tetszőleges méretű vektorról.

Példa:

```
int[,,] három = new int[4,5,3];
```

Lehetőségünk van a kétféle vektordefiníció kevert használatára is. A következő példa egy olyan egyszerű tömböt (tömbök tömbjét) definiálja, aminek minden eleme egy 3 indexes tömb, például a tér egy pontjának valamilyen jellemzőjét írja le. Ezt a jellemzőt szintén nem egy egyszerű adattal adjuk meg, hanem egy 2 dimenziós szabályos mátrixszal.

Példa:

```
int[][,,][,] mix;
```

Ezek után írjunk egy rövid példaprogramot, amely bemutatja a korábban megbeszélt vektorjellemzők használatát:

Példa:

```
//vektor.cs
using System;

class vektorpelda
{
    public static void Main()
    {
        // Sima vektordefiníció, a definíció pillanatában
        // 5 elemű vektor lesz
        // A vektorelemeknek külön nem adtunk kezdőértéket,
        // ezért azok 0 értéket kapnak.
        int[] számok = new int[5];

        // Kétdimenziós vektor, sima 5x4-es szöveges,
        //ezen elemek inicializálás hiányában null értéket kapnak.
        string[,] nevek = new string[5,4];

        // Vektorok vektora, ezt az irodalom gyakran
        //"jagged array", (csipkés, szaggatott vektor) néven említi
        byte[][] pontok = new byte[5][];

        // Az egyes vektorok definiálása
        for (int i = 0; i < pontok.Length; i++)
        {
            pontok[i] = new byte[i+3]; // elemről elemre nő az elemszám.
        }
```

```
            // Egyes sorok hosszának kiírása
            for (int i = 0; i < pontok.Length; i++)
            {
              Console.WriteLine("Az {0} sor hossza: {1}", i, pontok[i].Length);
                 // A {} jelek közötti számmal hivatkozhatunk az első paraméter
                 // utáni további értékekre. {0} jelenti a nulladik
                 // változót a sorban, jelen esetben az i változót, stb.
                 // Használata nagyon hasonlít a C printf használatához.
            }
        }
    }
```

A program futása után a következő eredményt kapjuk:

```
Az 0 sor hossza: 3
Az 1 sor hossza: 4
Az 2 sor hossza: 5
Az 3 sor hossza: 6
Az 4 sor hossza: 7
```

Példa:

```
string hónapnév(int n)
{
     string[]név={"Nem létezik","Január",
              "Február","Március",
              "Április","Május","Június",
              "Július","Augusztus",
              "Szeptember","Október",
              "November","December"};
     return(((n<1)||(n>12)) ? név[0] : név[n]);
}
```

Az iménti példa meghatározza egy tetszőleges sorszámú hónaphoz annak nevét.

IV.2. Struktúra

Gyakran szükségünk van az eddigiektől eltérő adatszerkezetekre, amikor a leírni kívánt adatunkat nem tudjuk egy egyszerű változóval jellemezni. Elég, ha a talán legkézenfekvőbb feladatot tekintjük: hogyan tudjuk a sík egy pontját megadni? Egy lehetséges megadási mód, ha a pontot mint a sík egy derékszögű koordinátarendszerének pontját tekintem, és megadom az X és Y koordinátákat.

A struktúradefiniálás az ilyen feladatok megoldása esetén lehetővé teszi, hogy az összetartozó adatokat egy egységként tudjuk kezelni.

A struktúradefiníció formája a következő:

```
struct név {
                hozzáférés típus mezőnevek;
                …
                hozzáférés függvénydefiníció ;
                };
```

Hivatkozás egy mezőre: *változónév.mezőnév*

A struktúrákon azonos típus esetén az értékadás elvégezhető, így ha például *a* és *b* azonos típusú struktúra, akkor az *a=b* értékadás szabályos, és az *a* minden mezője felveszi *b* megfelelő mezőértékeit.

Példa:

```
struct tanuló
{
     string név;
```

```
        int kor;
};                  // egy struktúra függvénydefiníciót
                    // is tartalmazhat, de az esetk nagy részében
                    // ahogy ebben a példában is, csak adatmezői
                    // vannak
tanuló zoli;
tanuló[] tanulók=new egy[10];   // 10 elemű struktúra vektor
```

A fenti definíció teljesen jó, csak a hozzáférési szint megadásának hiányában minden mező privát, azaz a struktúra mindkét adata csak belülről látható. A struktúra alapértelmezett mezőhozzáférése privát (*private*). Minden taghoz külön meg kell adni a hozzáférési szintet.

Struktúra esetén a megengedett hozzáférési szintek:

private	csak struktúrán belülről érhető el
public	bárki elérheti ezt a mezőt
internal	programon belülről (assembly) elérhető

Példa:

```
struct személy
{
    public string név;
    private int kor;
};
```

Az egyes mezőkre hivatkozás formája:

Példa:

```
személy st=new személy();
System.WriteLine( st.név); // név kiírása
```

Struktúradefiníció esetén a nyelv nem csak adatmező, hanem függvénymező definiálást is megenged. Azt a függvénymezőt, aminek ugyanaz a neve, mint a struktúrának, konstruktornak nevezzük. Paraméter nélküli konstruktor nem definiálható, azt mindig a környezet biztosítja. A struktúra értéktípusú adat, így a vermen és nem a dinamikus memóriában jön létre. Általában elmondható, hogy a struktúra majdnem úgy viselkedik, mint egy osztály, de funkcionalitását tekintve megmarad a különböző típusú adatok tárolásánál.

A struktúrákkal kapcsolatosan érdemes megjegyezni néhány alapvető tulajdonságot:

- Struktúrából nem származtathatunk.

Példa:

```
struct szilva
{
    public string név;
    public int tömeg;
}
struct befőtt : szilva       // fordítási hiba
{
    public int üvegméret;
}
```

- Egy struktúra adatmezőt a definiálás pillanatában nem inicializálhatunk.

Példa:

```
struct alma
{
```

```
        string nev="jonatán";  // fordítási hiba
        …
}
```

- Struktúra adatmezőket az alapértelmezett (*default)* konstruktor nem inicializál. A kezdőérték beállításáról, ha szükséges, saját konstruktorral vagy egyéb beállítási móddal kell gondoskodni.

Példa:

```
struct pont
{
    public int x,y;
}
pont p=new pont();
p.x=2; p.y=4;
```

- Bár a struktúra érték típusú, azért a *new* operátorral kell biztosítani a saját konstruktorral történő inicializációt. Ez ebben az esetben is a vermen fog elhelyezkedni.

Befejezésül a struktúra definiálására, használatára, struktúra vektorra nézzünk egy teljesebb példát:

Példa:

```
using System;
struct struktúra_példa
{
    public int kor;
    public string név;
}
class struktúra_használ
{
    public static void Main()
    {
        struktúra_példa sp=new struktúra_példa();
        sp.kor=5;
        sp.név="Éva";
        // struktúra_példa vektor
        struktúra_példa [] spv=new struktúra_példa[5];
        int i=0;
        // beolvasás
        while(i<5)
        {
            spv[i]=new struktúra_példa();
            Console.WriteLine("Kérem az {0}. elem nevét!",i);
            string n=Console.ReadLine();
            spv[i].név=n;
            Console.WriteLine("Kérem az {0}. elem korát!",i);
            n=Console.ReadLine();
            spv[i].kor=Convert.ToInt32(n);
            i++;
        }
        // kiírás
        for(i=0;i<5;i++)
        {
            Console.WriteLine(spv[i].név);
            Console.WriteLine(spv[i].kor);
        }
    }
}
```

Befejezésül a II.8 pontban említett anonymous típus kapcsán kell megemlíteni, hogy valójában a névtelen típus leginkább egy struktúra definiálására, típusnév nélküli használatára hasonlít

Nézzük egymás mellett az előző struktúra_példa típust, és annak névtelen megvalósítását.
Példa:

```
// klasszikus használat
struct struktúra_példa     // típusdefiníció
{
    public int kor;
    public string név;
}
// sp példány definíció
struktúra_példa sp=new struktúra_példa();
sp.kor=45;
sp.név="Éva";

// névtelen típussal, mintha gyakorlatilag az
// előző struktúra_példa típus lenne

var sp1=new { kor=46, név=”Zoli” };
```

IV. 3. Adatsorok - Tuples

Érdekes adatforma jelent meg a C# legújabb 7. verziójában. Ez a Tuples. A szótári jelentés alapján ez egyfajta rekord, egy adatsor (n-es lista). Ez gyakorlatilag a struktúrához hasonlít leginkább, azzal a különbséggel, hogy ebben az esetben nem kell előre, egzakt módon egy struktúrát definiálni, rögzített nevekkel, típusokkal, hanem az igényeknek megfelelően összerakom az adatsoromat a konkrét adatokkal. Gyakorlatilag azt is mondhatjuk, hogy ez a névtelen összetett típusnak az általánosítása lenne.

Nem ismeretlen ez a nyelvek világában, elsősorban funkcionális nyelvi környezetekben volt ismert szerkezet.

A Tuples adat összeállításnak általános formája:

(adat1, adat2, adat3,…)

Az adatsorban szereplő mezők számát minden esetben az aktuális igényekhez szabjuk. Ezen forma használata, ahogy említettük a legújabb 7.0 nyelvi verziótól érhető el, de a dinamikus változásra jellemző, hogy például ezen adatok összehasonlíthatósága (==,!=) csak a legújabb (jelenleg 7.3) nyelvi verziótól használható! Elmondhatjuk tehát, hogy ennek az eszköznek az értelmezése fokozatosan bővül jelenleg.

Ezek után nézzünk egy kódrészletet, amiben jobban látható ezen lehetőség használata.

Példa:

```
// 7.0 C# óta elérhető a Tuple típus
            // Ez nem beépített alap nyelvi elem, a NuGet manageren keresztül kell a
projekthez
            // adni a System.ValueTuple csomagot. Ha ez megtörtént, megjelenik a
referenciák között
            // ez a ValueTuple típus!
            var t = (5, "Alma"); // Névtelen tuple definícó
            Console.WriteLine($"Első tuple mező: {t.Item1}, Második mező: {t.Item2}");
            int x = 5; string s = "Alma";
            var t1 = (x, s);
            Console.WriteLine($"Első tuple mező: {t1.Item1}, Második mező: {t1.Item2}");
            // 7.1 C# után a mezőnév változó is használható!
```

```
            Console.WriteLine($"Első tuple mező: {t1.x}, Második mező: {t1.s}");
            // 7.3 óta az összehasonlíthatóság is él
            if (t1 == t) Console.WriteLine($"Azonosak!");
```

Ha egy projektet mondjuk a 7.1 nyelvi verzióval készítettünk el először, akkor az utolsó sorban látott azonosságvizsgálatnál hibát jelez a VS környezet és tanácsolja, hogy állítsuk be a legújabb 7.3-as nyelvi verziót!

Zárásképpen nézzük meg, hogyan használhatjuk ezt a lehetőséget függvények eredményeként.

Példa:

```
static public (int,string) adatsor()
        {
            return (3, "barack");
           }
    …
// Tuples eredményt adó függvény hívása
            var t2 = adatsor();
               Console.WriteLine($"Elem1: {t2.Item1}, Elem2: {t2.Item2}");
```

V. Utasítások

A programvezérlés menetét az utasítások szekvenciája szabja meg. Ahogy korábban is láttuk, az operációs rendszer a *Main* függvénynek adja át a vezérlést, majd a függvénytörzs egymás után következő utasításait (szekvencia) hajtja végre, ezután tér vissza a vezérlés az operációs rendszerhez.

Az utasítások fajtái:

- összetett utasítás
- kifejezés utasítás
- elágazás utasítás
- ciklus utasítás
- ugró utasítás

V.1. Összetett utasítás

Ahol utasítás elhelyezhető, ott szerepelhet összetett utasítás is.

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

V.2. Kifejezés utasítás

A kifejezés mint utasítás jellemzően egy matematikai műveletet jelent. Az alábbi példa kifejezése egy változó értékét növeli egyesével egy ciklusban.

Példa:

```
int k=5;
for(i=0; i<10; i++)
     k++;
```

V.3. Elágazás utasítás

Az elágazások két típusa használható, a kétfelé és a sokfelé elágazó utasítás.

A kétfelé elágazó utasítás formája:

if (kifejezés) utasítás;
if (kifejezés) utasítás; else utasítás;

Először a kifejezés kiértékelődik, majd annak igaz értéke esetén a kifejezés utáni utasítás, hamis értéke esetén az *else* – ha van – utáni utasítás hajtódik végre.

Egymásba ágyazás esetén az *else* ágat a legutóbbi *else* nélküli *if*hez tartozónak tekintjük.

Példa:

```
if (c=='a')
     Console.WriteLine("Ez az a betű");
else
     Console.WriteLine("Nem az a betű");
```

Ha az igaz ágon összetett utasítást használunk, akkor utána pontosvessző nem kell, illetve ha mégis van, az az *if* lezárását jelentené, és hibás lenne az utasításunk az *if* nélküli *else* használata miatt.

Példa:

```
if (c=='a')
     { Console.WriteLine("Ez az a betű");}
else
     Console.WriteLine("Nem az a betű");
```

A többirányú elágazás utasítása, a *switch* utasítás formája:

```
switch (kifejezés) {
        case érték1: utasítás1 ;
        case érték2: utasítás2 ;
        …
        default: utasítás ;
        };
```

A *switch* utáni kifejezés nem csak egész értékű lehet, hanem szöveg *(string)* is. Ha a kifejezés értéke a *case* után megadott értékkel nem egyezik meg, akkor a következő *case* utáni érték vizsgálata következik. Ha egy *case* utáni érték azonos a kifejezés értékével, akkor az ez utáni utasítás végrehajtódik. Ha egy *case* ágban nincs egyértelmű utasítás arra vonatkozóan, hogy hol folytatódjon a vezérlés, akkor fordítási hibát kapunk.

A C++ nyelvet ismerők tudhatják, hogy abban a nyelvben egy *case* ág végét nem kellett vezérlésátadással befejezni, és ebből gyakran adódtak hibák.

Minden elágazáságat, még a *default* ágat is, kötelező valamilyen vezérlésátadással befejezni! (*break, goto, return…*)

Példa:

```
switch (parancs)
{
     case "run":
          Run();
          break;
     case "save":
          Save();
          break;
     case "quit":
          Quit();
          break;
     default:
          Rossz(parancs);
          break;
}
```

A *case* belépési pont után csak egy érték adható meg, intervallum vagy több érték megadása nem megengedett, hiszen ezek az 'értékek' gyakorlatilag egy címke szerepét töltik be. Ha két értékhez rendeljük ugyanazt a tevékenységet, akkor két címkét kell definiálni.

Példa:

```
…
switch (a)
{
case 1:
case 2:
     Console.WriteLine("Egy és kettő esetén…");
     break;
default:Console.WriteLine("Egyébként…");
     break;
```

```
};
…
```

V.4. Ciklus utasítás

A ciklus utasításnak négy formája alkalmazható a C# nyelvben, ezek a *while*, a *do*, a *for* és a *foreach* ciklus utasítások.

V.4.1. „while” utasítás

A *while* ciklus utasítás formája:

while (kifejezés) utasítás;

Először a zárójelben lévő kifejezés kiértékelődik, ha értéke igaz, akkor a zárójeles kifejezést követő utasítás – amit szokás ciklusmagnak nevezni –, végrehajtódik. Ezután újból a kifejezés kiértékelése következik, igaz érték esetén pedig a ciklusmag végrehajtása. Ez az ismétlés addig folytatódik, amíg a kifejezés hamis értéket nem ad.

Mivel az utasítás először kiértékeli a kifejezést és csak utána hajtja végre a ciklusmagot (ha a kifejezés igaz értéket adott), ezért a *while* ciklus utasítást elöltesztelő ciklusnak is szokás nevezni.

Példa:

```
while(true)
{
     Console.Writeline("Végtelen ciklus!");
     Console.WriteLine("Ilyet ne nagyon használj!");
}
…
static void Main()              // betűnkénti kiírás
{
     string szöveg[]="Szöveg!";
     int i=0;
     while(i<szöveg.Length)     // a string végéig
     {
          Console.WriteLine(szöveg[i]);
          i++;
     }
}
```

V.4.2. „do” utasítás

Az utasítás formája:

do utasítás; while (kifejezés) ;

A ciklusmag – a *do* és a *while* közötti rész – végrehajtása után kiértékeli a kifejezést, és amíg a kifejezés igaz értéket ad, megismétli a ciklusmag végrehajtását. A *do* ciklust szokás hátultesztelő ciklusnak is nevezni.

Példa:

```
string név;
do
{
```

```
        Console.WriteLine("Ki vagy?");
        név=Console.ReadLine();
    }
    while (név!="Zoli");
    Console.WriteLine("Szia {0}!",név);

    do
        Console.WriteLine("Biztos egyszer lefut!");
    while(false);
```

V.4.3. „for" utasítás

Az utasítás formája:

for (kifejezés1; kifejezés2; kifejezés3) utasítás;

Ez az utasítás a következő alakhoz hasonlít:

```
kifejezés1;
while (kifejezés2)     {
            utasítás;
            kifejezés3;
            };
```

Mindegyik kifejezés elmaradhat. Ha *kifejezés2* is elmarad, akkor *while (true)*-ként tekinti a ciklust. A kifejezések közötti pontosvessző nem maradhat el.

A *kifejezés1* típusdefinícós kifejezés utasítás is lehet. Nagyon gyakran látható forráskódban az alábbi forma:

Példa:

```
for (int i=0; i<10; i++)
    Console.WriteLine("Hajrá Fradi!");
```

Az i változó csak a cikluson belül látható! Ez a fajta ciklus használat felel meg például a Basic *For…Next* ciklusának.

V.4.4. „foreach" utasítás

Az utasítás formája:

foreach (azonosító in vektor) utasítás;

Ez az utasítás egyenértékű a vektor-elemeken történő végiglépdeléssel. A vektort általánosabban egy felsorolható típussal is helyettesíthetjük. Az azonosító, mint ciklusváltozó felveszi egyenként e vektor, felsorolható típus elemeit.

Példa:

```
public static void Main()
{
    int paros = 0, paratlan = 0;
    int[] v = new int [] {0,1,2,5,7,8,11};
    foreach (int i in v)
    {
        if (i%2 == 0)
        paros++;
        else
        paratlan++;
    }
```

```
          Console.WriteLine("Találtam {0} páros, és {1} páratlan
 számot.",paros, paratlan);
     }
```

A *foreach* ciklus az általunk definiált vektorokon kívül az ehhez hasonló könyvtári adatszerkezeteken is (*ArrayList, Queue* stb.) jól használható. Ezzel kapcsolatos példát a *Helyi könyvtárak használata* fejezetben láthatunk.

A *foreach* utasítás nemcsak közvetlen vektortípuson, mint például a fenti *v* vektoron alkalmazható, hanem minden „végigjárható” típuson.

.

V.5. Ugró utasítások

V.5.1. „break” utasítás

Hatására befejeződik a legbelső *while, do, for* vagy *switch* utasítás végrehajtása. A vezérlés a következő utasításra adódik.

Példa:

```
  int i=1;
       while(true)                // látszólag végtelen ciklus
       {
            i++;
            if (i==11) break;     // Ciklus vége
            Console.WriteLine( i);
       }
```

A *break* a többirányú elágazás *(switch)* utasításban is gyakran használt, így kerülhetjük el, hogy a nem kívánt *case* ágak végrehajtódjanak.

V.5.2. „continue” utasítás

A hatására a legbelső *while, for, do* ciklus utasításokat vezérlő kifejezések kerülnek kiértékelésre. (A ciklus a következő ciklusmag végrehajtásához készül.)

Példa:

```
  int i=1;
       while(true)                // 10 elemű ciklus
       {    i++;
            if (i<=10) continue;  // következő ciklusmag

            if (i==11) break;     // Ciklus vége
            Console.WriteLine( i);
       }
```

A fenti példában a *continue* utasítás miatt a *Console.WriteLine(i)* utasítás egyszer sem hajtódik végre.

A break, continue utasítások szabad használata nem tartozik a klasszikus, struktúrált programozási stílus közvetlen eszköztárába, így használatuk kerülendő.

V.5.3. „return” utasítás

Az utasítás formája:

return ;
return kifejezés;
return (kifejezés);

A vezérlés visszatér a függvényből, a kifejezés értéke a visszaadott érték.

V.5.4. „goto” utasítás

Az utasítás formája:

goto címke;

A vezérlés arra a pontra adódik, ahol a *címke:* található.

Példa:

```
goto tovább;
Console.WriteLine("Ezt a szöveget sohase írja ki!");
tovább:;
…
int i=1;
switch (i)
{
     case 0:
          nulla();
          goto case 1;      // egy case 1: sor valójában címke
     case 1:
          egy();
          goto default;
     default:
          valami();
          break;
}
```

A *goto* utasításról zárásképpen meg kell jegyezni, hogy a strukturált programkészítésnek nem feltétlenül része ez az utasítás, így használata sem javasolt. A könyv későbbi példaprogramjaiban sem fordul elő egyszer sem ez az utasítás.

V.5.5. „yield” utasítás, iterátor

A foreach utasítás ismertetésénél láttunk egy klasszikus megvalósítást a felsorolhatóságra. Az így kapott konstrukciót iterátornak is nevezzük.

Ennek a lehetőségnek egy egyszerűbb megvalósítását biztosítja a yield utasítás. A GetEnumerator metódusban a yield return utasítással adhatjuk meg a felsorolható adatsorozatban az elemek aktuális értékét.

Példa:

```
namespace Foci
{
    public class adatsor
    {
        int[] elemek;
        public adatsor()
        {
            elemek = new int[5] { 12, 44, 33, 2, 50 };
        }
        public  IEnumerator GetEnumerator()
        {
            foreach(int i in elemek)
             yield return i;
```

```
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            adatsor asor = new adatsor();
          // asor olyan típus, melynek elemein végiglépdelhetünk
            foreach (int i in asor)
                Console.WriteLine(i);
        }
    }
}
```

A yield return segítségével egy aktuális lépés során visszaadott értéket tudunk a külső hívó felé visszaadni. A következő lépésre, elem visszaadásra (*yield return*), a következő elem kérésekor (az asor objektum következő elemének) kerül sor.

A GetEnumerator az *IEnumerable* interfész egyetlen metódusa. Az adatsor osztálydefiníciója a következő formában is használható.

```
public class adatsor:IEnumerable
```

A felsorolhatóságot nem feltétlenül kell kötnünk egy osztály definícióhoz. Egy függvényre is ráhúzhatjuk a felsorolhatóságot, ekkor a függvény eredmény típusa *IEnumerable*.

Példa:

```
public static IEnumerable faktoriális(int meddig)
{
    int fakt=1;
    for (int i = 1; i <= meddig; i++)
        yield return fakt *= i;
}
static void Main(string[] args)
{
  // faktoriális sorozata 1-től 10-ig.
    foreach (int i in faktoriális(10))
        Console.WriteLine(i);
}
```

Ha az iterációt nem egy egyszerű ciklussal valósítjuk meg, szükség lehet a lépéssorozat megszakítására. Ezt biztosítja a *yield break* utasítás.

V.5.6. „using” utasítás

A *using* kulcsszó kétféle nyelvi környezetben szerepelhet. Egyrészt ún. direktíva, könyvtári elemek, adott névtér, osztály használataként. Ennek formája:

using névtér_vagy_osztály;

Példa:

```
using System; // a keretrendszer fő névterének használata
```

```
// a C++ #include-hoz hasonlóan megmondja a
// fordítónak, hogy ennek a névtérnek a típusait is
// használja. Ezen típusokat azért enélkül is teljes
// névvel (System.Console…) használhatjuk
```

A másik eset a *using* utasítás. Egy ideiglenesen használt objektum esetén a *using* utasítás után a keretrendszer automatikusan meghívja az objektum *Dispose* metódusát. A *using* utasítás alakja a következő:

using (objektum) { utasítások;}

Példa:

```
using (Objektumom o = new Objektumom())
{
    o.ezt_csinald();
}
```

Ez a használat az alábbi kódrészletnek felel meg, a nem ismert nyelvi elemeket később ismertetjük:

```
…
Objektumom o = new Objektumom();
try
    {o.ezt_csinald();}
finally
{
    if (o != null) ((IDisposable)o).Dispose();
}
```

A using utasítás file műveletek esetén is gyakran használt, mintegy blokkolja az állományt a using blokk idejére.

A *Dispose* utasítást és környezetét bővebben a destruktorokkal kapcsolatban részletezzük.

V.5.7. „lock" utasítás

Ha egy kritikus utasítás blokk végrehajtásakor egy referencia blokkolására van szükség a biztonságos végrehajtáshoz, akkor ezt a *lock* utasítással megtehetjük.

A *lock* kulcsszó egy referenciát vár, ez lehet akár a *this*, majd utána következik a kritikus blokk. Ennek formája:

lock(ref) utasítás;

Példa:

```
object zár=new object();
lock(zár)
{
    a=5; // ehhez a blokkhoz egyszerre csak egy végrehajtási
         // szál (hívás) fér hozzá, amíg „zár" zárva van
} // innentől zár nyitva
```

A *lock* utasításnak, ahogy láttuk, a leggyakrabban használt paramétere a *this*, ha a védett változó vagy függvényutasítás nem statikus. (A *this* mindig az aktuális osztálypéldány referenciája.)

V. Utasítások

VI. Függvények

VI.1. A függvények paraméterátadása

Ha egy függvénynek adatot adunk át paraméterként, akkor alapvetően két különböző esetről beszélhetünk. Egy adat érték szerint és hivatkozás szerint kerülhet átadásra. Az első esetben valójában az eredeti adatunk értékének egy másolata kerül a függvényhez, míg a második esetben az adat címe kerül átadásra.

VI.1.1 Érték szerinti paraméterátadás

Általánosan elmondhatjuk, hogyha nem jelölünk semmilyen paraméterátadási módszert, akkor érték szerinti paraméterátadással dolgozunk. Ekkor a függvény paraméterében, mint formális paraméterben, a függvény meghívásakor a hívóérték másolata helyezkedik el.

Tekintsük meg a következő maximum nevű függvényt, aminek az a feladata, hogy a kapott két paramétere közül a nagyobbat adja vissza eredményként.

Példa:

```
class maxfv
{
    static int maximum(int x,int y)
    {
        return (x>y?x:y);
    }

    static void Main()
    {
        Console.WriteLine(maximum(4,3));
    }
}
```

Az így megvalósított paraméterátadást érték szerinti paraméterátadásnak nevezzük, a *maximum* függvény két (*x* és *y*) paramétere a híváskor megadott két paramétert kapja értékül. Érték szerinti paraméterátadásnál, híváskor konstans érték is megadható.

A függvény hívásának egy sajátos esete az, mikor egy függvényt saját maga hív meg. Szokás ezt rekurzív függvényhívásnak is nevezni. Erre példaként nézzük meg a klasszikus faktoriálisszámoló függvényt.

Példa:

```
class faktor
{
    static int faktorialis(int x)
    {
        return (x<=1?1: (x* faktorialis(x-1)));
    }
    static void Main()
    {
        Console.WriteLine(faktorialis(4));
    }
}
```

VI.1.2. Referencia (cím)és eredmény(out) szerinti paraméterek

Amikor egy függvény paramétere nem a változó értékét, hanem a változó tárolási helyének címét, hivatkozási helyét kapja meg a függvény meghívásakor, akkor a paraméterátadás módját cím szerinti paraméterátadásnak nevezzük.

A cím szerinti paraméterátadást gyakran hivatkozás (*reference*) szerinti paraméterátadásnak is szokás nevezni.

Ha a paraméternév elé a *ref* (hivatkozás) kulcsszót beszúrjuk, akkor a hivatkozás szerinti paraméterátadást definiáljuk. Ezt a kulcsszót be kell szúrnunk a függvénydefinícióba és a konkrét hívás helyére is!

Példaként írjunk olyan függvényt, ami a kapott két paraméterének értékét felcseréli.

Példa:

```
…
// a ref kulcsszó jelzi, hogy referencia
// paramétereket vár a függvény
void csere(ref int x, ref int y)
{
     int segéd=x;
     x=y; y=segéd;
}
static void Main()
{
     int a=5, b=6;
     Console.WriteLine("Csere előtt: a={0}, b={1}.", a , b);
     // függvényhívás, a ref kulcsszó itt is kötelező
     csere(ref a,ref b);
     Console.WriteLine("Csere után: a={0}, b={1}.", a , b);
}
```

A *csere(a,b)* hívás megcseréli a két paraméter értékét, így az *a* változó értéke 6, míg *b* értéke 5 lesz.

Egy függvény definíció szerint egyértelmű (egy) eredményt ad. A gyakorlatban sokszor előfordul, hogy két vagy többféle adatot kell szolgáltatni. Ekkor egy struktúra vagy újabban egy Tuples lehet az eredmény. Ennél sok esetben egyszerűbb lehet, hogy a bemeneti paraméterek mellett eredmény (out) paramétert adunk meg. Az out kulcsszót jelezni kell a függvény fejlécében és híváskor is.

Példa:

```
static public int eredményparaméter(out string név)
 {
     név = "Zoli";
     return 2;
 }
….
string t3;
     Console.WriteLine($"Eredmény: {eredményparaméter(out t3)}");
     Console.WriteLine(t3);
```

VI.1.3. Függvényeredmény paramétere

A referencia szerinti paraméterátadáshoz hasonlóan, a függvénybeli változó értéke kerül ki a hívó változóba. A különbség csak az, hogy ebben a változóban a függvény nem kap értéket. Ennek a paraméternek a helyére, híváskor konstans adatot nem adhatunk meg!

Az eredmény paramétert az *out* kulcsszóval definiálhatjuk. A híváskor is a paraméter elé ki kell írni az *out* jelzőt. A használata akkor lehet hasznos, amikor egy függvénynek egynél több eredményt kell adnia.

Példa:

```
using System;
public class MyClass
{
```

```
        public static int TestOut(out char i)
        {
            i = 'b';
            return -1;
        }

        public static void Main()
        {
            char i;       // nem kell inicializálni a változót
            Console.WriteLine(TestOut(out i));
            Console.WriteLine(i);
        }
    }


    Képernyő eredmény:
    -1
    b
```

VI.1.4. Tömbök paraméterátadása

A nyelv a tömböt annak nevével, mint referenciával azonosítja, ezért egy tömb paraméterátadása nem jelent mást, mint a tömb referenciájának átadását. Egy függvény természetesen tömböt is adhat eredményül, ami azt jelenti, hogy az eredmény egy tömbreferencia lesz. Egy tömb esetében sincs másról szó, mint egyszerű változók esetében, a tömbreferencia – egy referenciaváltozó –, érték szerinti paraméterátadásáról.

Az elmondottak illusztrálására lássuk a következő sematikus példát.
Példa:

```
void módosít(int[] vektor)
{
    // a vektort, mint egy referenciát kapjuk paraméterül
    // ez érték szerint kerül átadásra, azaz ez a referencia
    //nem változik, változik viszont a 0. tömbelem, és ez a
    // változás a hívó oldalon is látszik
    vektor[0]=5;
}
```

Ahogy a fenti példán is látható, a vektor paraméterként a referenciájával kerül átadásra. Ez érték szerinti paraméterátadást jelent. Ha egy függvényben a mutatott értéket (vektorelemet) megváltoztatom, akkor a változás a külső, paraméterként átadott vektorban is látható lesz!

A nyelvben a vektor tudja magáról, hogy hány eleme van *(Length)*, ezért – ellentétben a C++ nyelvvel –, az elemszámot nem kell átadni.

Tömb esetében is alkalmazhatjuk a referencia vagy *out* paraméter átadásának lehetőségét. Ennek illusztrálására nézzük a következő példákat:

Példa:

```
        class Refteszt
    {
        public static void feltölt(ref int[] arr)
        {
            // Ha hívó fél még még nem készítette el,
            //akkor megtesszük itt.
            if (arr == null)
            arr = new int[10];
```

```
            // néhány elem módosítás
            arr[0] = 123;
            arr[4] = 1024;
            // a többi elem értéke marad az eredeti
        }

        static public void Main ()
        {
            // Vektor inicializálása
            int[] vektor = {1,2,3,4,5};

            // Vektor átadása ref, paraméterként:
            feltölt(ref vektor);

            // Kiírás:
            Console.WriteLine("Az elemek:");
            for (int i = 0; i < vektor.Length; i++)
                Console.WriteLine(vektor[i]);
        }
    }
```

VI.2. A Main függvény paraméterei

A parancsok, programok indításakor gyakori, hogy a parancsot valamilyen paraméter(ek)rel indítjuk. Ilyen parancs például a DOS *echo* parancsa, amely paramétereit a képernyőre „visszhangozza”, vagy a *type* parancs, mely a paraméterül kapott fájl tartalmát írja a képernyőre.

Ezeket a paramétereket parancssor-argumentumoknak is szokás nevezni. Amikor elindítunk egy programot (a *Main* függvény az operációs rendszertől átveszi a vezérlést), akkor híváskor a *Main* függvénynek egy paramétere van. Ez a paraméter egy szöveges *(string)* tömb, amelynek elemei az egyes paraméterek. A paramétertömböt gyakran *args* névre keresztelik**.**

A parancssor-argumentumok használatára nézzük az imént már említett *echo* parancs egy lehetséges megvalósítását.

Példa:

```
static void Main(string[] args)
{
    int i=0;
    while (i<args.Length)
    {
        Console.WriteLine(args[i]);
        i++;
    }
}
```

Ha a fenti programot lefordítjuk, és a neve *parancs.exe* lesz, akkor a következőképpen próbálhatjuk ki:

```
c:\parancs.exe fradi vasas újpest
```

A program futtatása után az alábbi eredményt kapjuk:

```
fradi
vasas
újpest
```

A parancssori paraméterek Visual Studio környezetet használva a projekt *Tulajdonság* ablakának *Debug* csoportban is megadhatók.

A *Main* függvényt, abban az esetben, ha a parancssori paramétereket nem akarjuk feldolgozni, a paraméterei nélkül is deklarálhatjuk:

```
static void Main()
    {…}
```

VI.3. Függvények változó számú paraméterrel

A feladatmegoldások során gyakran előfordulhat, hogy előre nem tudjuk eldönteni, hány elemet kell a függvénynek feldolgoznia. Tehát nem tudjuk, hogy hány paraméterrel készítsük el a kívánt függvényünket. A *params* kulcsszó segítségével egy vektorparamétert definiálhatunk a függvénynek, ami ezeket a paramétereket tartalmazza majd.

Nézzünk egy példát ilyen függvény definiálására, majd a használatára.

Példa:

```
public static void intParaméterek(params int[] list)
{
    for ( int i = 0 ; i < list.Length ; i++ )
        Console.WriteLine(list[i]);
    Console.WriteLine();
}
…
    intParaméterek(1, 2, 3);
```

Egy változó paraméterszámú függvénynek lehetnek fixen megadott paraméterei is. A fixen megadott paramétereknek kötelezően meg kell előzniük a változó paramétereket.

VI.4. Függvénynevek átdefiniálása

Egy-egy feladat esetén gyakorta előfordul, hogy a megoldást adó függvényt különböző típusú vagy különböző számú paraméterrel jellemezhetjük. Természetesen az ezt megoldó függvényt azonos névvel szeretnénk elkészíteni minden esetben, hisz ez lenne a legkifejezőbb. A lehetőséget gyakran függvény „overloading” névvel is megtaláljuk az irodalomban, vagy a polimorfizmus kapcsán kerül megemlítésre.

Határozzuk meg két szám maximumát! Ha el szeretnénk kerülni a típuskonvertálást mondjuk egész típusról valósba és fordítva, akkor külön az egész és külön a valós számok esetére is meg kell írnunk a *maximum* függvényt.

Kényelmetlen dolog lenne, ha mondjuk *egészmax* és *valósmax* néven kellene megírni a két függvényt. Mindkét esetben szeretnénk a *maximum* függvénynevet használni, hogy ne nekünk kelljen eldönteni azt, hogy az egészek vagy valósak maximumát akarjuk-e meghatározni, hanem döntse el a fordító a paraméterlista alapján automatikusan, hogy melyik függvényt is kell az adott esetben meghívni.

Példa:

```
// valós maximum függvény
double maximum(double x,double y)
{
```

```
        if (x>y)
            return x;
        else
            return y;
    }

    // egész maximum függvény
    int maximum(int x,int y)
    {
        if (x>y)
            return x;
        else
            return y;
    }
    …
    int a=2,b=4;
    double c=3.123, d=2;
    Console.WriteLine(maximum(4.1,2));  // valós hívás
    Console.WriteLine(maximum(4,2));    // egész hívás
    Console.WriteLine(maximum(a,b));    // egész hívás
    Console.WriteLine(maximum(c,d));    // valós hívás
    …
```

Meg kell jegyezni, hogy ezt a tulajdonságot függvénysablon *(generic)* tulajdonság segítségével elegánsabban oldhatnánk meg, amit a típusparaméterek fejezetben meg is mutatunk.

VI.5. Delegáltak, események

Ahogy korábban is szó volt róla, a biztonságosabb programkód készítésének érdekében a mutató aritmetika a C# nyelvben nem engedélyezett. Ebbe beletartozik az is, hogy a függvénymutatók sem lehetnek kivételek.

Ez utóbbi esetben viszont olyan nyelvi tulajdonságok nem implementálhatók, mint például egy menüponthoz hozzárendelt függvény, amit a menüpont választása esetén kell végrehajtani. Ezen utóbbi lehetőséget hivatott a delegált típus biztosítani. Ez már csak azért is fontos, mert többek között a Windows programozás *„callback”* jellemző paramétere is ezt használja.

A C++ környezetben ezt a lehetőséget a függvénymutató biztosítja.

A delegált valójában egy függvénytípus-definíció, aminek alakja a következő:

delegate típus delegáltnév(típus paraméternév,…);

A delegált referencia típus. Ha paramétert is megadunk, akkor a paraméter nevét is meg kell adni.

Példa:

```
delegate int pelda(int x, string s);
```

Az előbbi példában tehát a *pelda* delegált típust definiáltuk. Ez olyan függvénytípus, amelyiknek egy egész és egy szöveg paramétere van, és eredményül egy egész számot ad.

Egy delegált objektumnak vagy egy osztály statikus függvényét, vagy egy osztály példányfüggvényét adjuk értékül. Delegált meghívásánál a hagyományos függvényhívási formát használjuk. Ezek után nézzünk egy konkrét példát:

Példa:

```
using System;
class proba
```

```
{
    public static int negyzet(int i)
    {
        return i*i;
    }
    public int dupla(int i)
    {
        return 2*i;
    }
}
class foprogram
{
    delegate int emel(int k);  // az emel delegált definiálása

    public static void Main()
    {
        emel f=new emel(proba.negyzet);
                        // statikus függvény lesz a delegált
        Console.WriteLine(f(5));        // eredmény: 25
        proba p=new proba();
        emel g=new emel(p.dupla);
                    // normál példányfüggvény lesz a delegált
        Console.WriteLine(g(5));        // eredmény: 10
    }
}
```

A delegáltakat a környezet két csoportba sorolja. Ha a delegált visszatérési típusa *void*, akkor egy delegált több végrehajtandó függvényt tartalmazhat (*multicast*, összetett delegált), ha nem *void* a visszatérési típus, mint a fenti példában, akkor egy delegált csak egy végrehajtandó függvényt tud meghívni (*single cast*, egyszerű delegált).

Az egyszerű és összetett delegáltakra nézzük a következő példát:

Példa:

```
using System;
class proba
{
    public static int negyzet(int i)
    {
        return i*i;
    }
    public int dupla(int i)
    {
        return 2*i;
    }
    public int tripla(int i)
    {
        return 3*i;
    }
    public void egy(int i)
    {
        Console.WriteLine(i);
    }
    public void ketto(int i)
    {
        Console.WriteLine(2*i);
    }
}
class foprogram
{
    delegate int emel(int k);
```

```
        delegate void meghiv(int j);

        public static void Main()
        {
            emel f=new emel(proba.negyzet);
            Console.WriteLine(f(5));
            proba p=new proba();
            emel g=new emel(p.dupla);
            Console.WriteLine(g(5));
            emel h=new emel(p.tripla);
            g+=h;                   //g single cast, g a tripla lesz
            Console.WriteLine(g(5));        // eredmény: 15
            meghiv m=new meghiv(p.egy);     // multicast delegált
            m+=new meghiv(p.ketto);
            m(5);                   // delegált hívás, eredmény 5,10
        }
    }
```

Ha delegált típust (függvénymutató) deklarálunk, akkor értékadás esetén annak típusa határozza meg, hogy a fordítónak melyik aktuális függvényt is kell az azonos nevűek közül választania.

```
    delegate double vmut(double,double);
    // vmut olyan delegált amelyik egy valós értéket visszaadó,      // és két valós
paramétert váró függvényt jelent

    delegate int emut(int,int);
    // emut olyan delegált, amelyik egy egész értéket visszaadó      // , és két egész
paramétert váró függvényt jelent
    vmut mut=new vmut(maximum);// valós hívás
    Console.WriteLine(mut(3,5));
```

A *multicast* delegált (típus) definiálási lehetőséget, igazítva az igényeket az eseményvezérelt programozási környezethez (mind a Windows, mind az X11 ilyen), az egyes objektumokhoz, típushoz úgy kapcsolhatjuk, hogy esemény típusú mezőt adunk hozzájuk. Ezt az alábbi formában tehetjük meg:

```
public event meghiv esemeny;
```

ahol a *meghiv* összetett delegált. Az esemény objektum kezdő értéke *null* lesz, így az *automatikus meghívás (esemeny(…))* nem ajánlott!

Az eseményhez zárásképpen meg kell jegyezni, hogy az egész keretrendszer a felhasználói tevékenységet ehhez a lehetőséghez rendeli akkor, amikor klasszikus Windows alkalmazást akarunk készíteni.

Példaként nézzük meg egy Windows alkalmazás esetén a form tervező által generált kódot. A form *(this)* objektuma egy választómező *(ComboBox), ampl* névre hallgat. Ennek könyvtári eseménye, a *SelectedIndexChanged* végrehajtódik (a keretrendszer hajtja végre), ha módosítunk a választáson.

Mikor ehhez az eseményhez egy eseménykezelő függvényt definiálunk *ampl_SelectedIndexChanged* néven, akkor az alábbi sort adja a programhoz a tervező:

Példa:

```
  this.ampl.SelectedIndexChanged += new
System.EventHandler(this.ampl_SelectedIndexChanged);
```

VI.6. Névtelen (anonymous) metódusok

A névtelen típusok mintájára definiálhatunk olyan „függvényeket” is melynek nem adunk nevet. Rögtön feltehetjük a kérdést, hogy mi értelme van ilyen definíciónak?

Csak önmagában nincs is értelme, viszont ha delegált, esemény végrehajtásra gondolunk akkor ebben az esetben a valódi név csak a delegált, esemény inicializálásánál használt, később már csak a delegált, esemény nevét használjuk. Ilyen esetekben megtehetjük, hogy kihagyjuk a klasszikus függvénydefiníciós lépést, és csak a törzset, a valódi tevékenységet adjuk meg a delegált végrehajtási utasításának.

A névtelen függvénydefiníció alakja a következő:

delegáltnév=delegate[paraméter] {függvénytörzs };

A további magyarázatok helyett nézzünk a névtelen metódusok használatára egy rövid példát.

Példa:

```
namespace anonymfv
{
    class Program
    {
        public delegate int módosít(int x);
        public delegate string üdv();

        public int dupláz(int mit)
        {
            return 2*mit;
        }
        public void paraméter_minta(módosít m)
        {
            Console.WriteLine("A paraméterül kapott delegált hívása!");
            Console.WriteLine(m(6));
        }
        static void Main(string[] args)
        {
            Program p=new Program();
            //Klasszikus delegált definíció
            módosít mf = new módosít(p.dupláz);
            //
            //Névtelen metódussal inicializált delegált
            üdv d1 = delegate { return "Hajrá Fradi!"; };
            //Névtelen paraméteres metódussal inicializálva
            módosít mf1 = delegate(int x) { return x * x; };
            Console.WriteLine("Hívások eredménye {0}, {1}.", mf(5), mf1(5));
            Console.WriteLine(d1());
            p.paraméter_minta(delegate(int x) { return 3 * x; });
          // Közvetlen, névtelen metódus a paraméter.
        }
    }
}
```

VI.7. Lambda kifejezések

A Lambda kifejezés a matematikából ismert (Lambda-kalkulus, λ-kalkulus) függvény absztrakciós művelet. Egy formális függvény leírási módszer, egyféle műveletet ismer, a változó behelyettesítést. A

VI. Függvények

Lambda kalkulust használva bizonyította be egymástól függetlenül Alonzo Church és Alan Turing, hogy a „Döntési probléma” néven ismert feladatra nem lehet megoldást adni (1936). Jelenleg a Lambda kalkulus leginkább a funkcionális nyelvekben játszik jelentős szerepet

Láttuk az előző fejezetben, a névtelen metódusok használatával egy kódrészletet, függvényt kapcsolhatunk közvetlenül egy delegálthoz. A Lambda kifejezések használatával ezt még tömörebben, kifejezőbb formában írhatjuk le.

A Lambda kifejezés segítségével tömören egy delegált típushoz tudunk egy függvényhozzárendelést adni. A kifejezés operátora a C# nyelvben a => . A kifejezés általános formája:

```
( x ) => f(x);
```

A baloldali operandust szokás paraméternek is nevezni, míg a jobboldali kifejezés megadja a paraméterhez tartozó helyettesítési értéket (a hozzá tartozó függvényt).

Nézzük a nyelvben használható jellemző formákat.

Példa:

```
// Konkrétan megadott típusok a baloldalon,
// teljes blokk a jobbon!
(int x) => {return x*x;}    // egy paraméteres kifejezés
(int a, int b) => { return a+b; }    // két paraméteres kifejezés
//
// Implicit típusok a baloldalon, a fordító kitalálja a konkrét
// használati típusokból a és b típusát.
// Ha a jobboldali kifejezés egyszerű, akkor elhagyható a return
// és a {} blokkhatárolók is.
(a,b) => a+b  // ugyanaz mint az első példa
x => db+=x    // egy paraméternél elhagyható a baloldalról
              // a zárójel
() => db+1    // paraméter nélküli kifejezés
```

Predikátumnak, állításnak nevezzük a Lambda kifejezést ha az logikai értéket ad vissza, illetve projekciónak ha a visszatérési típus eltér a paraméter típusától.

Ahhoz, hogy használni tudjuk a kifejezéseket nem elég az előző példában látott módon beírni egy programba valamelyik sort, ugyanis ez csak kifejezés. Ahogy önmagában az a+b jellegű kifejezés sem teljes utasítás, úgy a Lambda kifejezés is csak egy utasítás rész.

Legegyszerűbb módon úgy használhatjuk ezt a lehetőséget, hogy a System.Linq névtér előredefiniált delegáltjait vesszük elő.

Az alábbi delegáltak ismertek ebben a névtérben:

```
public delegate T Func<T>();                    // paraméter nélküli delegált
public delegate T Func<P1,T>(P1 p1);            // 1 paraméteres delegált
public delegate T Func<P1,P2,T>(P1 p1, P2 p2);  // 2 paraméteres delegált
public delegate T Func<P1,P2,P3,T>(P1 p1,P2 p2,P3 p3);
// 3 paraméteres delegált
public delegate T Func<P1,P2,P3,P4,T>(P1 p1,P2 p2,P3 p3,P4 p4);
// 4 paraméteres delegált
```

Ezen delegáltakban a P1,P2,P3,P4 típusparaméterek, amiknek használatáról a IX fejezetben olvashatunk bővebben.

Akinek vagy nem felel meg a Func név, vagy 5 vagy több paraméteres delegáltra van szüksége, az ennek mintájára létrehozhatja saját delegáltjait.

Példa:

```
// „Gyári” 1 paraméteres delegált használat
int i;
```

```
Func<int,int> f1 = (x) => { return x*x; };
i= f1(3);          // 9
…
// Két paraméteres delegált (P, P1), T az eredmény típusa
public delegate T Almafa<T, P, P1>(P p,P1 p1);
// delegált nem lehet függvényen belül!!!
…
Almafa<int,string, bool> fa = (x, y) => { if (y) return x.Length;
                                          else return 0; };
int starking= fa(„starking”,true);
Console.WriteLine(starking);         // 8
```

VII. Osztályok

C#-ban az osztályok az összetett adatszerkezetek (struktúrák) egy természetes kiterjesztése. Az osztályok nemcsak adattagokat, hanem operátorokat, függvényeket is tartalmazhatnak. Az osztályok használata esetén általában az adatmezők az osztály által éppen leírni kívánt esemény „állapotjelzői”, míg a függvénymezők az állapotváltozásokat írják le, felhasználói felületet biztosítanak. A függvénymezőket gyakran metódusoknak is nevezik. Osztálymezőket lehetőségünk van zárttá nyilvánítani *(encapsulation),* ezzel biztosítva az osztály belső „harmóniáját”.

Lehetőségünk van az eredeti osztály megtartása mellett, abból a tulajdonságok megőrzésével egy új osztály származtatására, amely *örökli (inheritance)* az ősosztály tulajdonságait (az osztályok elemeit). A C++ nyelvben lehetőségünk van arra, hogy több osztályból származtassunk utódosztályt *(multiple inheritance),* esetünkben ezt a jellemzőt a CLR nem támogatja, így ennek a fejlesztőkörnyezetnek egyik nyelve sem támogatja a többszörös öröklés lehetőségét. Lehetőségünk van *interface* definícióra, amiket egy osztály implementálhat. Egy osztály több *interface*-t is implementálhat.

Egy osztálytípusú változót, az osztály megjelenési alakját gyakran objektumnak is szokás nevezni.

Az osztályok használatához néhány jótanácsot, a nyelv „jobbkéz-szabályát” a következő pontokban foglalhatjuk össze.

Ha egy programelem önálló értelmezéssel, feladattal, tulajdonságokkal rendelkezik, akkor definiáljuk ezt az elemet önálló osztályként.

Ha egy programrész adata önálló objektumként értelmezhető, akkor definiáljuk őt a kívánt osztálytípus objektumaként.

Ha két osztály közös tulajdonságokkal rendelkezik, akkor használjuk az öröklés lehetőségét.

Általában is elmondható, hogy ha a programokban az osztályok közös vonásokkal rendelkeznek, akkor törekedni kell univerzális bázisosztály létrehozására. Gyakran *ezt absztrakt bázisosztálynak* is nevezzük.

Az osztályok definiálásakor kerüljük a „nyitott” (publikus) adatmezők használatát.

VII.1. Osztályok definiálása

Az osztályok deklarációjának alakja a következőképpen néz ki:

```
osztálykulcsszó osztálynév [: szülő, …]
{
osztálytaglista
};
```

Az osztálykulcsszó a *class*, *struct* szó lehet. Az osztálynév az adott osztály azonosító neve. Ha az osztály valamely bázisosztályból származik, akkor az osztálynév, majd kettőspont után az ősosztály megadása történik.

Az osztályok tagjainak öt elérési szintje lehet:

- private: csak az adott osztályon belülről érhetjük el
- public: bárhonnan elérhetjük, módosíthatjuk
- protected: az osztályon kívülről nem elérhetjük el, míg az utódosztályból igen
- internal: a készülő program osztályaiból érhetjük el
- protected internal: elérhetjük a programon belülről, vagy az osztály utódosztályából

VII. Osztályok

Ha egy „osztály” definiálásakor a *struct* szót használjuk, akkor abban elsősorban adatok csoportját szeretnénk összefogni, ezért gyakran tanácsként is olvashatjuk, hogy ha hagyományos értelemben vett struktúrát, mint összetett adatszerkezetet szeretnénk használni, akkor azt *„struct típusú osztályként”* definiáljuk.

Az egyes mezőnevekre való hivatkozás ugyanúgy történik, ahogy korábban a struktúrák esetében.

Példa:

```
class Példa
{
    private int szám; // privát mező
    public void módosít(int új) { szám = új; }
    public int lekérdez() { return szám; }
}
```

Mivel a szám az osztály privát mezője, ezért definiáltunk az osztályba két függvénymezőt, amelyek segítségével be tudjuk állítani, illetve ki tudjuk olvasni az értékét. Ahhoz, hogy ezt a két függvényt az osztályon kívülről el tudjuk érni, publikusnak kellett őket deklarálni.

```
class Program
{
    static void Main(string[] args)
    {
        Példa első = new Példa(); // létrehozunk egy első nevű Példa típusú változót
        első.szám = 4; // fordítási hiba
        első.módosít(5);
        Console.WriteLine(első.lekérdez());
    }
}
```

Osztályt egy osztályon vagy egy függvényen belül is definiálhatunk, ekkor azt belső vagy lokális osztálynak nevezzük. Ennek a lokális osztálynak a láthatósága hasonló, mint a lokális változóké.

Minden osztályhoz automatikusan létrejön egy mutató, aminek a neve *this*, és az éppen aktuális osztálypéldányra mutat. Így, ha egy osztályfüggvényt meghívunk, amely valamilyen módon például a privát változókra hat, akkor az a függvény a *this* mutatón keresztül tudja, hogy mely aktuális privát mezők is tartoznak az objektumhoz. A *this* mutató konkrétabb használatára a későbbiek során láthatunk példát.

Statikus mezők nem az objektumhoz, hanem az osztályhoz kötődnek, melyeket a *static* jelző kiírásával definiálhatunk. Ekkor az adott mező minden objektum esetében közös lesz. A statikus mező kezdőértéke inicializálás hiányában *0, null, false* lesz.

Példa:

```
class Példa  {
     public static int a; // a kezdőértéke 0
     public static string név = "Katalin";
     …
              };
…
Példa.a = 8;      // statikus mező beállítása 8-ra
```

Statikus mezők a *this* mutatóra vonatkozó utalást nem tartalmazhatnak, hiszen a statikus mezők minden egyes objektum esetén (azonos osztálybelire vonatkozóan) közösek. Tehát például statikus függvények nem statikus mezőket nem tudnak elérni! Fordítva természetesen problémamentes az elérés, hiszen egy normál függvényből bármely statikus mező, függvény elérhető.

VII.2. Konstruktor- és destruktor függvények

Osztályok esetén egy függvény kapja meg az inicializálásával járó feladatot. Ez a függvény az osztálypédány (objektum) „születésének” pillanatában automatikusan végrehajtódik, és konstruktornak vagy konstruktor függvénynek nevezzük. A konstruktor neve mindig az osztály nevével azonos. Ha ilyet nem definiálunk, a keretrendszer egy paraméter nélküli automatikus konstruktort definiál az osztály számára.

A konstruktor egy szabályos függvény, így mint minden függvényből, ebből is több lehet, ha mások a paraméterei.

Az osztály referencia típusú változó, egy osztálypéldány létrehozásához kötelező a *new* operátort használni, ami egyúttal a konstruktor függvény meghívását végzi el. Ha a konstruktornak vannak paraméterei, akkor azt a típusnév után zárójelek között kell megadni.

VII.2.1. Statikus konstruktor

A statikus mezőknek az inicializálását végezheti a statikus konstruktor. Definiálása opcionális, ha nem definiáljuk, a keretrendszer nem hozza létre.

Ha definiálunk egy statikus konstruktort, akkor annak meghívása a program indulásakor megtörténik, még azelőtt, hogy egyetlen osztálypéldányt is definiálnánk.

Statikus konstruktor elé nem kell hozzáférési módosítót, visszatérési típust írni. Ennek a konstruktornak nem lehet paramétere sem.

Példa:

```
class statikus_konstruktor
{
     public static int x;
     static statikus_konstruktor()
     {
          x = 2;
     }
     public int getx()
     {
          return x
     }
}
class program
{
     public static void Main()
     {
          // valahol itt a program elején kerül meghívásra az
          // osztály statikus konstruktora, az x értéke 2 lesz!!!
          Console.WriteLine(statikus_konstruktor.x);       // 2
          statikus_konstruktor s=new statikus_ konstruktor();
          // dinamikus példány
          Console.WriteLine(s.getx()); // természetesen ez is 2 lesz
     }
}
```

VII.2.2. Statikus osztály

Definiálhatunk olyan osztályt is, ami maga is statikus. Ilyen osztálynak csak statikus mezői lehetnek. Egy ilyen mező lehet akár egy statikus kunstruktor is! Az ilyen osztályokból nem készíthetünk példányokat, nem származtathatunk belőlük újabb típusokat.

Példa:

```
public static class statikus_osztály
{
    static int x;
    static statikus_osztály()
    {
        x=5;
    }
    public static int szoroz(int mit)
    {
        return x*mit;
    }
}
```

A könyvtári szolgáltatások között talán a leggyakrabban használt osztályunk (System.Console) is statikus.

VII.2.3. Függvények kiterjesztése (Extension methods)

Egy statikus osztály csak statikus tagokkal (adat vagy függvény) rendelkezhet. Ez a programunk számára azt jelenti, hogy ezen adatok, függvények ha publikusak, bárki számára elérhetők

Ilyen statikus osztályban megengedett az a függvénydefiníciós forma is, mikor az első paramétert megelőzi a ***this*** kulcsszó. (Lehetnek további paraméterek is!)

public static típus függvénynév(**this** típus név, tov.paraméterek)
{
függvénytörzs;
};

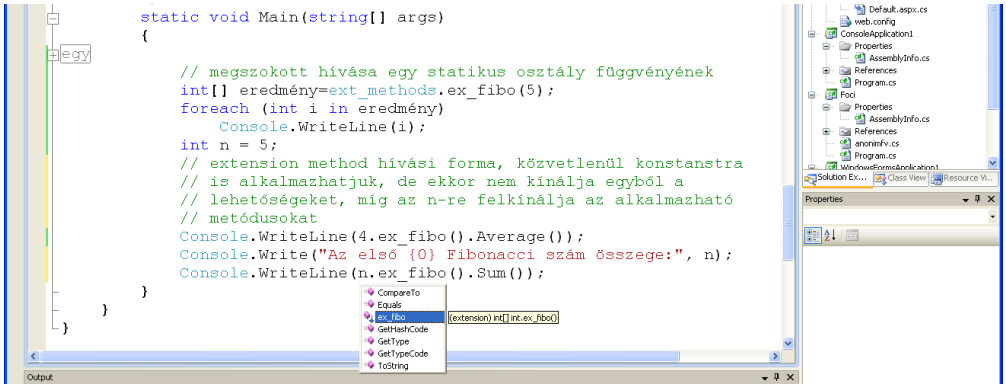
Az így definiált függvényt meghívhatjuk „hagyományos” módon is, de úgy is mint az első paraméternek, mint objektumnak egy függvényeként!

Nézzünk egy konkrét példát. Készítsük el azt a függvényt, amelyik egy ***n*** egész számhoz megadja az első n Fibonacci számot.

Példa:

```
static class ext_methods_példa
{
 public static int[] ex_fibo(this int n) // első n fibonacci szám
//Az egész típusnak egy kiterjesztése
  {
      int[] eredmény = new int[n];
      eredmény[0] = eredmény[1] = 1;
      for (int i = 2; i < n; i++)
          eredmény[i] = eredmény[i - 1] + eredmény[i - 2];
      return eredmény;
  }
}
```

Ekkor az így definiált *ex_fibo* függvényünket egy egész típusú érték (n) kiterjesztés függvényének is nevezzük. Ennek kétféle hívását láthatjuk a következő képen.



11. ábra

A függvénykiterjesztés hívási sorrendje (balról jobbra) a fordítottja a rendes egymásbaágyazott függvényhívások sorrendjének. (jobbról balra)

VII.2.4. Privát konstruktor

Szükség lehet arra is, hogy egy osztályból kívülről ne tudjunk egyetlen példányt se létrehozni. A probléma úgy oldható meg, hogy privát konstruktort definiálunk. Egy népszerű példa az egyke programozási minta, ahol az osztályból csak egy példányt tudunk létrehozni.

Példa:

```
class Egyke
{
    private static Egyke példány = null;
    public static Egyke Példány
    {
        get
        {
            if (példány == null)
                példány = new Egyke();
            return példány;
        }
    }

    private Egyke()
    {
        // a konstruktortörzs üres
    }
}
class Program
{
    static void Main(string[] args)
    {
        Egyke a = new Egyke(); // Fordítási hiba
        Egyke b = Egyke.Példány;
    }
}
```

VII.2.5. Saját destruktor

Ahogy egy osztály definiálásakor szükségünk lehet bizonyos inicializáló kezdeti lépésekre, úgy az osztály vagy objektum „elmúlásakor” is sok esetben bizonyos lépéseket kell tennünk. Például, ha az osztályunk dinamikusan foglal magának memóriát, akkor azt használat után célszerű felszabadítani.

Azt a függvényt, amelyik az osztály megszűnésekor szükséges feladatokat elvégzi destruktornak vagy destruktor függvénynek nevezzük. A destruktor neve is az osztály nevével azonos, csak az elején ki kell egészíteni a ~ karakterrel. A destruktor meghívása automatikusan történik, és miután az objektumot nem használjuk, a keretrendszer automatikusan lefuttatja, és a felszabaduló memóriát visszaadja az operációs rendszernek.

Konstruktornak és destruktornak nem lehet visszaadott értéke. A destruktornak mindig publikus osztálymezőnek kell lennie.

Ha egy osztályhoz nem definiálunk konstruktort vagy destruktort, akkor a rendszer automatikusan egy alapértelmezett, paraméter nélküli konstruktort, illetve destruktort definiál hozzá. Ha viszont van saját konstruktorunk, akkor nem 'készül' automatikus.

A destruktor automatikus meghívásának a folyamatát szemétgyűjtési algoritmusnak nevezzük (garbage collection, GC). Ez az algoritmus nem azonnal, az objektum blokkjának, élettartamának a végén hívja meg a destruktort, hanem akkor, amikor az algoritmus „begyűjti" ezt a szabad memóriaterületet. Ha nem írunk saját destruktor függvényt, akkor is a garbage collector minden, az objektum által már nem használt memóriát felszabadít az automatikusan definiált destruktor függvény segítségével. Ez azt jelenti, hogy nincs túlzottan nagy kényszer saját destruktor definiálására.

VII.3. Konstans, csak olvasható mezők

Az adatmezők hozzáférhetősége az egyik legfontosabb kérdés a típusaink tervezésekor. Ahogy korábban már láttuk, az osztályon belüli láthatósági hozzáférés állításával *(private, public, ...)* a megfelelő adathozzáférési igények kialakíthatók. Természetes ezek mellett az az igény is, hogy a változók módosíthatóságát is szabályozni tudjuk.

VII.3.1 Konstans mezők

Egy változó módosíthatóságát a *const* kulcsszóval is befolyásolhatjuk. A konstans mező olyan adatot tartalmaz, amelyik értéke fordítási időben kerül beállításra. Ez azt jelenti, hogy ilyen mezők inicializáló értékadását kötelező a definíció során jelölni.

Példa:

```
class konstansok
{
    public const double pi=3.14159265;  // inicializáltuk
    public const double e=2.71828183;
}
```

Egy konstans mező eléréséhez nincs szükség arra, hogy a típusból egy változót készítsünk, ugyanis a konstans mezők egyben statikus mezők is. Ahogy látható, a konstans mezőt helyben inicializálni kell, ebből pedig az következik, hogy minden később definiálandó osztályváltozóban ugyanezen kezdőértékadás hajtódik végre, tehát ez a mező minden változóban ugyanaz az érték lehet. Ez pedig a statikus mező tulajdonsága.

VII.3.2. Csak olvasható mezők

Ha egy adatmezőt a *readonly* jelzővel látunk el, akkor ezt a mezőt csak olvasható mezőnek nevezzük. Ezen értékeket a program során csak olvasni, használni tudjuk. Ezen értékek beállítását egyetlen helyen, a konstruktorban végezhetjük el. Látható, hogy a konstans és a csak olvasható mezők közti leglényegesebb különbség az, hogy míg a konstans mező minden példányban ugyanaz, addig a csak olvasható mező konstansként viselkedik miután létrehoztuk a változót, de minden változóban más és más értékű lehet!

Példa:

```
class csak_olvashato
{
    public readonly double x;
    public csak_olvashato(double kezd)
    { x=kezd; }
```

```
}
class olvashatosapp
{
    public static void Main()
    {
        csak_olvashato o=new csak_olvashato(5);
        Console.WriteLine(o.x);         // értéke 5
        csak_olvashato p=new csak_olvashato(6);
        Console.WriteLine(p.x);         // értéke 6
        p.x=8;        // fordítási hiba, x csak olvasható!!!!
    }
}
```

Természetesen definiálható egy csak olvasható mező statikusként is, ebben az esetben a mező inicializálását az osztály statikus konstruktorában végezhetjük el.

VII.4. Tulajdonság, index függvény

Egy osztály tervezésekor nagyon fontos szempont az, hogy az osztály adattagjaihoz milyen módosítási lehetőségeket engedélyezzünk, biztosítsunk.

VII.4.1. Tulajdonság, property függvény

A C# nyelv a tulajdonság *(property)* függvénydefiniálási lehetőségével kínál egyszerű és kényelmes adathozzáférési és módosítási eszközt. Ez egy olyan speciális függvény, amelynek nem jelölhetünk paramétereket, még a zárójeleket sem *()*, és a függvény törzsében egy *get* és *set* blokkot definiálunk. Használata egyszerű értékadásként jelenik meg. A *set* blokkban egy *„value”* névvel hivatkozhatunk a beállítási értékadás jobb oldali kifejezés értékére.

Példa:

```
class Ember
{
    private string nev;
    private int kor;
    // a nev adatmező módosításához definiált Nev tulajdonság
    public string Nev // kis- és nagybetű miatt nev!=Nev
    {
        get      // ez a blokk hajtódik végre akkor,
                 // amikor a tulajdonság értéket kiolvassuk
        {
            return nev;
        }
        set      // ez hajtódik végre mikor a
                 // tulajdonságot írjuk
        {
            nev=value;
        }
    }
    public Ember(string n, int k)
    {
        nev=n; kor=k;     // nev, kor privát mezők elérése
    }
}
class program
{
    public static void Main()
    {
        Ember e=new Ember("Zoli", 16);
```

```
            //a Nev tulajdonság hívása
            Console.WriteLine(e.Nev);  // a Nev property get blokk hívása
            e.Nev="Pali";     // a Nev property set blokkjának hívá-
                              // sa a value változóba kerül a "Pali"
        }
    }
```

Ha a tulajdonság függvénynek csak *get* blokkja van, akkor azt csak olvasható tulajdonságnak *(readonly)* nevezzük.

Ha a tulajdonság függvénynek csak *set* blokkja van, akkor azt csak írható tulajdonságnak *(writeonly)* nevezzük.

Gyakran szükség lehet arra, hogy a tulajdonságok egyikéhez ne publikus legyen a hozzáférés. Ez általában a set blokkra vonatkozik, ugyanis amíg egy tulajdonság értékére mindenkinek szüksége lehet, de annak módosítását azért ne mindenki tudja elvégezni. Ezt az aszimmetrikus elérési lehetőséget elérhetjük ha a get, set blokkok elé írjuk a kívánt (protected, privát, internal) jelzőket.

Példa:

```
        public string Nev // kis- és nagybetű miatt nev!=Nev
        {
            get      //
            {
                return nev;
            }
            protected set     // az utódból engedem a módosítást
            {
                nev=value;
            }
        }
```

VII.4.2. Index függvény (indexer)

Az *indexer* függvény definiálása valójában a vektorhasználat és a tulajdonság függvény kombinációja. Gyakran előfordul, hogy egy osztálynak olyan adatához szeretnénk hozzáférni, aminél egy indexérték segítségével tudjuk megmondani, hogy melyik is a keresett érték.

Az *indexer* függvény esetében lényeges különbség, hogy van egy *index* paraméter, amit szögletes zárójelek között kell jelölni, és nincs neve, pontosabban a *this* kulcsszó a neve, ugyanis az aktuális típust, mint vektort indexeli.

Mivel az *indexer* az aktuális típust, a létező példányt indexeli, ezért az *indexer* függvény nem lehet statikus.

Lássuk az alábbi példát, ami a jellemző használatot mutatja.

Példa:

```
  class valami
  {
        int [] v=new int[10];
        …
        public int this[int i]
        {
            get
            {
                return v[i];
            }
            set
            {
                v[i]=value;       // mint a property value értéke
            }
```

```
        }
    }
    class program
    {
        static void Main()
        {
            valami a=new valami();
            a[2]=5;                // az indexer set blokk hívása
            Console.WriteLine(a[2]);        // 5, indexer get hívása
        }
    }
```

Az indexer esetében is, hasonlóan a tulajdonságdefinícó használatához, nem feltétlenül kell mindkét *(get, set)* ágat definiálni. Ha csak a *get* blokk definiált, akkor csak olvasható, ha csak a *set* blokk definiált, akkor csak írható *indexer* függvényről beszélünk.

Az indexerhasználat kísértetiesen hasonlít a vektorhasználathoz, de van néhány olyan különbség, amit érdemes megemlíteni.

- Az *indexer* függvény, és egy függvénynek pedig nem csak egész *(index)* paramétere lehet.

Példa:

```
    public int this[string a, int b]
    {
        get
        {
            return b;
        }
        set
        {
            nev=a;
            adat[b]=value;
        }
    }
```

- Az *indexer* függvény az előzőekből következően újradefiniálható *(overloaded).* A fenti *indexer* mellett a következő „hagyományos” is megfér:

Példa:

```
    public string this[int x]
    {
        get
        {
            return s[x];
        }
        set
        {
            s[x]=value;
        }
    }
```

- Az *indexert ref* és *out* paraméterként nem használhatjuk.

VII.5. Osztályok függvényparaméterként

Egy függvény paraméterei más egyszerű adattípushoz hasonlóan lehetnek osztálytípusok is. Alapértelmezés szerint az osztálytípusú változó is érték szerint adódik át.

Példa:

```
    class példa
    {
        private int x;
```

```
        public példa (int a)
        {
            Console.WriteLine( "Konstruktorhívás!");
            x=a;
        };
        public int X
        {
            get
            {
                return x;
            }
            set
            {
                x=value;
            }
        }
    }
    class program
    {
        static int négyzet(példa a)
        {
            a.X=5;
            return (a.X*a.X);
        }

        static void Main()
        {
            példa b=new példa(3);
            Console.WriteLine(négyzet(b));
        }
    }
```

A program futtatása a következőket írja a képernyőre:

```
Konstruktorhívás!
25
```

Mikor egy függvény paraméterként (érték szerint) egy osztályt kap, akkor a függvényparaméter egy értékreferenciát kap, és nem egy másolata készül el az eredeti osztályváltozónak.

Teljesen hasonló akkor a helyzet, mikor egy függvény osztálytípust ad visszatérési értékként.

VII.6. Operátorok újradefiniálása

A már korábban tárgyalt operátoraink az ismert alaptípusok esetében használhatók. Osztályok (új típusok) megjelenésekor az értékadás operátora automatikusan használható, mert a nyelv létrehoz egy számára alapértelmezett értékadást az új osztályra is. Ezen értékadás operátort felülbírálni, azaz egy újat definiálni nem lehet a C# nyelvben (ellentétben pl. a C++ nyelvvel).

Hasonlóan nem lehet az index operátort [] sem újradefiniálni, bár az *indexer* definiálási lehetőség lényegében ezzel egyenértékű.

A legtöbb operátor újradefiniálható, melyek egy- vagy kétoperandusúak. Az alábbi operátorok definiálhatók újra:

Egyoperandusú operátorok: +, -, !, ~, ++, --, true, false
Kétoperandusú operátorok: +, -, *, /, %, &, |, ^, <<, >>, <=, >=, ==, !=, <, >

A fenti hagyományosnak mondható operátorkészlet mellett még típuskonverziós operátor függvény is definiálható.

Az operátor függvény definíciójának formája:

```
static visszatérési_érték operator?(argumentum)
{
        // függvénytörzs
}
```

Az *operator* kulcsszó utáni *?* jel helyére az újradefiniálni kívánt operátor neve kerül. Tehát ha például az összeadás *(+)* operátorát szeretnénk felülbírálni, akkor a *?* helyére a + jel kerül.

Az irodalomban az operátor újradefiniálást gyakran operátor overloading-nak hívják.

Az operátorok precedenciája újradefiniálás esetén nem változik, és az operandusok számát nem tudjuk megváltoztatni, azaz például nem tudunk olyan / (osztás) operátort definiálni, amelynek csak egy operandusa van.

Az operátor függvények öröklődnek, bár a származtatott osztályban az ősosztály operátor függvényei igény szerint újradefiniálhatóak.

Az operátor függvény argumentumával kapcsolatosan elmondhatjuk, hogy egyoperandusú operátor esetén egy paramétere van, míg kétoperandusú operátor esetén két paramétere van a függvénynek.

Tekintsük első példaként a komplex számokat megvalósító osztályt, amelyben az összeadás operátorát szeretnénk definiálni oly módon, hogy egy komplex számhoz hozzá tudjunk adni egy egész számot. Az egész számot a komplex szám valós részéhez adjuk hozzá, a képzetes rész változatlan marad.

Példa:

```
class komplex
{
    private float re,im;
    public komplex(float x,float y) // konstruktor
    {
        re=x; im=y;
    }
    float Re
    {
        get
        {
            return re;
        }
        set
        {
            re=value;
        }
    }
    float Im
    {
        get
        {
            return im;
        }
        set
        {
            im=value;
        }
    }
    public static komplex operator+(komplex k,int a)
    {
```

```
            komplex y=new komplex(0,0);
            y.Re=a+k.Re;
            y.Im=k.Im;
            return y;
        }
        override public string ToString()   // Példány szöveges alak
        {
            string s="A keresett szám:"+re+":"+im;
            return s;
        }
    }
    class program
    {
        public static void Main()
        {
            komplex k=new komplex(3,2);
            Console.WriteLine("Komplex szám példa.");
            Console.WriteLine("Összeadás eredménye: {0}",k+4);
            // a k+4 komplex indirekt toString hívás
        }
    }
```

Az eredmény a következő lesz:

```
Komplex szám példa.
Összeadás eredménye: A keresett szám:7:2
```

A *k+4* összeadás úgy értelmezendő, hogy a *k* objektum + függvényét hívtuk meg a *k* komplex szám és a *4* egész szám paraméterrel, azaz *k.+(k,4)* függvényhívás történt.

Abban az esetben, ha a *komplex + komplex* típusú összeadást szeretnénk definiálni, akkor egy újabb operátor függvénnyel kell a *komplex* osztályt bővíteni. Ez a függvény a következőképpen nézhet ki:

```
    public static komplex operator+(komplex a, komplex b)
    {
        komplex temp=new komplex(0,0);
        temp.re= b.re+a.re;
        temp.im= b.im+a.im;
        return temp;
        // rövidebben így írható
        // return new komplex(a.re+b.re,a.im+b.im);
    }
```

Ahhoz, hogy az összeadás operátorát a korábban (az egyszerű típusoknál) megszokott módon tudjuk itt is használni, már csak az kell, hogy az *egész + komplex* típusú összeadást is el tudjuk végezni. (Az összeadás kommutatív!) Erre az eddigi két összeadás operátor nem ad lehetőséget, hiszen ebben az esetben, a bal oldali operandus mindig maga az aktuális osztály. A mostani esetben viszont a bal oldali operandus egy egész szám.

Ekkor a legkézenfekvőbb lehetőség az, hogy az *egész + komplex* operátorfüggvényt is definiáljuk. Figyelembe véve a Visual Studio szövegszerkesztőjének szolgáltatásait, ez gyorsan megy, így elkészítjük ezt a változatot is:

```
    public static komplex operator+(int a, komplex k)
    {
        komplex y=new komplex(0,0);
        y.Re=a+k.Re;
        y.Im=k.Im;
        return y;
```

```
    }
```

Ekkor a *Console.WriteLine("Összeadás eredménye: {0}",4+k);* utasítás nem okoz fordítási hibát.

Erre a problémára még egy megoldást adhatunk. Ez pedig a konverziós operátor definiálásának lehetősége. Ugyanis, ha definiálunk olyan konverziós operátort, amely a 4 egész számot komplex típusúra konvertálja, akkor két komplex szám összeadására vezettük vissza ezt az összeadást.

A konverziós operátoroknak készíthetünk implicit vagy explicit változatát is. Implicitnek nevezzük azt a konverziós operátorhívást, mikor nem jelöljük a forrásszövegben a konverzió műveletét, explicitnek pedig azt, amikor jelöljük.

Konverziós operátornál az operátor jel helyett azt a típust írjuk le, amire konvertálunk, míg paraméterül azt a típust adjuk meg, amiről konvertálni akarunk.

Visszatérve a fenti komplex szám kérdésre, az *egész + komplex* operátor helyett az alábbi operátort is definiálhattuk volna:

```
    public static implicit operator komplex(int a)
    {
        komplex y=new komplex(0,0);
        y.Re=a;
        return y;
    }
```

Ha az operátor szó elé az *implicit* kulcsszót írjuk, akkor az implicit operátor függvényt definiáljuk, tehát *4 + komplex* jellegű utasításnál a *4* számot implicit (nem jelölve külön) konvertáljuk komplex számmá.

```
    komplex k= new komplex(3,2);
    komplex k1= 5 + k1;    // implicit hívás
```

Előfordulhat, hogy az implicit és az explicit konverzió mást csinál, ekkor, ha akarjuk, az explicit verziót is definiálhatjuk.

```
    public static explicit operator komplex(int a)
    {
        komplex y=new komplex(0,0);
        y.Re=a+1;     // mást csinál, mint az előző
                      // nem biztos, hogy matematikailag is helyes!!!
        return y;
    }
```

Az explicit verzió meghívása a következőképpen történik:

```
    komplex k=(komplex) 5;
    Console.WriteLine(k.Re);        // 6
```

Természetesen egy komplex számhoz értékadás útján rendelhetünk egy valós számot is, mondjuk oly módon, hogy a komplex szám valós részét adja a valós szám, míg a képzetes rész legyen 0.

Két egyoperandusú operátor, a ++ és a -- esetében, ahogy az operátorok tárgyalásánál is láttuk, létezik az operátorok postfix illetve prefix formájú használata is:

```
    komplex k=new komplex(1,2);
    k++;          // postfix ++
    ++k;     // prefix forma
```

Ha definiálunk ++ operátor függvényt, akkor ezen két operátor esetében mindkét forma használatánál ugyanaz az operátor függvény kerül meghívásra.

```
    public static komplex operator++(komplex a)
```

```
    {
        a.Re=a.Re+1;
        a.Im=a.Im+1;
        return a;
    }
```

Befejezésül a *true, false* egyoperandusú operátor definiálásának lehetőségéről kell röviden szólni. A C++ nyelvvel ellentétben, ahol egy adott típust logikainak is tekinthetünk (igaz, ha nem 0, hamis, ha 0), a C# nyelvben a már korábbról ismert

```
    if (a) utasítás;
```

alakú utasítások akkor fordulnak le, ha az *a* változó logikai. A *true* és *false* nyelvi kulcsszavak nemcsak logikai konstansok, hanem olyan logikai egyoperandusú operátorok, melyeket újra lehet definiálni.

A *true, false* operátor logikai. Megkötés, hogy mindkét operátort egyszerre kell újradefiniálnunk. A jelentése pedig az, hogy az adott típust mikor tekinthetjük logikai igaznak vagy hamisnak.

Definiáljuk újra ezeket az operátorokat a már megismert komplex osztályunkhoz:

```
    public static bool operator true(komplex x)
    {
        return x.Re > 0;
    }
    public static bool operator false(komplex x)
    {
        return x.Re <= 0;
    }
```

Ez a definíció ebben az esetben azt jelenti, hogy egy komplex szám akkor tekinthető logikai igaz értékűnek, ha a szám valós része pozitív.

Példa:

```
    komplex k=new komplex(2,0);
    if (k) Console.WriteLine("Igaz");
```

Ekkor a képernyőre kerülő eredmény az *igaz* szó lesz!

Végül azt kell megemlíteni, hogy a logikai *true, false* operátorok mintájára, azokhoz hasonlóan csak párban lehet újradefiniálni néhány operátort. Ezek az operátorok a következők:

==, !=	azonosság, különbözőség megadása
<, >	kisebb, nagyobb
<=, >=	kisebb vagy egyenlő, nagyobb vagy egyenlő

VII.7. Interface definiálása

Egy osztály definiálása során a legfontosabb feladat az, hogy a készítendő típus adatait, metódusait megadjuk. Gyakran felmerül az az igény, hogy ne egy osztály keretében fogalmazzuk meg a legjellemzőbb tulajdonságokat, hanem kisebb tulajdonságcsoportok alapján definiáljunk. A keretrendszer viszont csak egy osztályból enged meg egy új típust, egy utódosztályt definiálni. Ez viszont ellentmond annak az elvárásnak, hogy minél részletesebben fogalmazzuk meg a típusainkat.

VII.7.1. Interface fogalma

A fenti ellentmondás feloldására alakult ki az a megoldás, hogy engedjünk meg olyan típust, interface-t definiálni, ami nem tartalmaz semmilyen konkrétumot, de rendelkezik a kívánt előírásokkal.

Az interfacedefiníció formája:

```
interface név
        {
        // deklarációs fejlécek
        }
```

Példa:

```
interface IAlma
{
    bool nyári();
    double termésátlag();
}
```

A fenti példában definiáltuk az *Ialma* interface-t, ami még nem jelent közvetlenül használható típust, hanem csak azt írja elő, hogy annak a típusnak, amelyik majd ennek az *IAlma* típusnak a tulajdonságait is magán viseli, kötelezően definiálnia kell a *nyári()*, és a *termésátlag()* függvényeket. Tehát erről a típusról azt tudhatjuk, hogy az interface által előírt tulajdonságokkal biztosan rendelkezni fog. Ezen kötelező tulajdonságok mellett természetesen tetszőleges egyéb jellemzőkkel is felruházhatjuk majd az osztályunkat.

Amikor könyvtári előírásokat, interface-eket implementálunk, akkor általában az elnevezések az *I* betűvel kezdődnek, utalva ezzel a név által jelzett tartalomra.

Egy interface előírásai közé nem csak függvények, hanem tulajdonságok és indexer előírás megadása és esemény megadása is beletartozhat.

Példa:

```
interface IPozíció
{
    int X { get; set; }
    int Y { get; }         // readonly tulajdonság
    int Z { set; }         // csak írható tulajdonság
    int this[int i] { get; set;} // read-write indexer
    int this[int i] { get; } // read-only indexer
    int this[int i] { set;} // write-only indexer
}
```

VII.7.2. Interface használata

Az *IAlma* előírások figyelembevétele a következőképpen történik. Az *osztálynév (típusnév)* után kettőspontot kell tenni, majd utána következik az implementálandó név.

Példa:

```
class jonatán: IAlma
{
    private int kor;
    public jonatán(int k)
    {
        kor=k;
    }
    public bool nyári()
    {
        return false;
    }
```

```
        public double termésátlag()
        {
            if ((kor>5) && (kor<30))
            return 230;
            else return 0;
        }
    }
```

VII.8. Osztályok öröklődése

Az öröklődés az objektumorientált programozás elsődleges jellemzője. Egy osztályt számaztathatunk egy ősosztályból, és ekkor az utódosztály az ősosztály tulajdonságait (függvényeit, …) is sajátjának tudhatja. Az örökölt függvények közül a változtatásra szorulókat újradefiniálhatjuk. Öröklés esetén az osztály definíciójának formája a következő:

```
class utódnév: ősnév
{
    // …
}
```

A C# nyelvben a .NET Frameworknek (Common Type System) köszönhetően minden mező automatikusan, mintha publikus öröklés lenne, megtartja ősosztálybeli jelentését. Ekkor az ősosztály publikus mezői az utódosztályban is publikus mezők, és a protected mezők az utódosztályban is protected mezők lesznek.

Az ősosztály privát mezői az utódosztályban is privát mezők maradnak az ősosztályra nézve is, így az ősosztály privát mezői közvetlenül az utódosztályból sem érhetők el. Az elérésük például publikus, ún. „közvetítő” függvény segítségével valósítható meg.

Egy őstípusú referencia bármely utódtípusra hivatkozhat.

Az elérési módok gyakorlati jelentését nézzük meg egy sematikus példán keresztül:

```
    class ős
    {
        private int i;          // privát mező
        protected int j;        // protected mezőtag
        public   int k;         // publikus mezők
        public void f(int j)
        { i=j; };
    }
    class utód: ős
    {
        …
    };
```

Ekkor az utódosztálynak „helyből” lesz egy *protected* mezője, a *j* egész változó, és lesz két publikus mezője, a *k* egész változó és az *f* függvény. Természetesen az utódosztálynak lesz egy *i* egészmezője is, csak az közvetlenül nem lesz elérhető.

Gyakran találkozunk azzal az esettel, mikor a könyvtárosztály *protected* mezőket tartalmaz. Ez azt jelenti, hogy a függvényt, mezőt olyan használatra szánták, hogy csak származtatott osztályból tudjuk használni.

A C# nyelvben nincs lehetőségünk többszörös öröklés segítségével egyszerre több ősosztályból egy utódosztályt származtatni. Helyette viszont tetszőleges számú interface-t implementálhat minden osztály.

```
    class utód: ős, interface1, …   {
                                        //
```

```
                                        };
```

Konstruktorok és destruktorok használata öröklés esetén is megengedett. Egy típus definiálásakor a konstruktor függvény kerül meghívásra, és ekkor először az ősosztály konstruktora, majd utána az utódosztály konstruktora kerül meghívásra, míg destruktor esetén fordítva, először az utódosztály majd utána az ősosztály destruktorát hívja a rendszer. Természetesen a destruktor hívására az igaz, hogy a keretrendszer hívja meg valamikor azután, hogy az objektumok élettartama megszűnik.

Paraméteres konstruktorok esetén az utódkonstruktor alakja:

```
        utód(paraméterek): base(paraméterek)
                                                {
                                                        // …
                                                }
```

Többszintű öröklés esetén először az őskonstruktorok kerülnek a definíció sorrendjében végrehajtásra, majd az utódbeli tagosztályok konstruktorai és legvégül az utódkonstruktor következik. A destruktorok hívásának sorrendje a konstruktorsorrendhez képest fordított. Ha nincs az utódban direkt őshívás, akkor a rendszer a paraméter nélküli őskonstruktorát hívja meg.

Ezek után nézzük a fentieket egy példán keresztül.

Példa:

```
  class a
  {
       public a()
       { Console.WriteLine( "A konstruktor";}
       ~a()
       { Console.WriteLine("A destruktor";}
  }
  class b: a
  {
       public b()
       { Console.WriteLine("B konstruktor";}
       ~b()
       { Console.WriteLine("B destruktor";}
  }
  class program
  {
       public static void Main()
       {
            b x=new b();  // b típusú objektum keletkezik, majd 'kimúlik'
            // Először az a majd utána a b konstruktorát hívja meg
            // a fordító
            // Kimúláskor fordítva, először a b majd az a
            // destruktora kerül meghívásra
       }
  }
```

VII.9. Végleges és absztrakt osztályok

A típusdefiníciós lépéseink során előfordulhat, hogy olyan osztályt definiálunk, amelyikre azt szeretnénk kikötni, hogy az adott típusból, mint ősből ne tudjunk egy másik típust, utódot származtatni.

Ahhoz, hogy egy adott osztályt véglegesnek definiáljunk, a *sealed* jelzővel kell ellátni.

Példa:

```
  sealed class végleges
  {
       public végleges()
```

```
        {
            Console.WriteLine( "A konstruktor" );
        }
}
    class utód:végleges        // fordítási hiba
    { }
```

Ha *protected* mezőt definiálunk egy *sealed* osztályban, akkor a fordító figyelmeztető üzenetet ad, mivel nincs sok értelme az utódosztályban látható mezőt definiálni.

Interface definiálás esetében csak előírásokat – függvény, tulajdonság formában – fogalmazhatunk meg az implementáló osztály számára. Gyakran előfordul, hogy olyan típust szeretnénk létrehozni, mikor a definiált típusból még nem tudunk példányt készíteni, de nemcsak előírásokat, hanem adatmezőket, implementált függvényeket is tartalmaz.

Ez a lehetőség az *abstract* osztály definiálásával valósítható meg. Ehhez az *abstract* kulcsszót kell használnunk az osztály neve előtt. Egy absztrakt osztály egy vagy több absztrakt függvénymezőt tartalmazhat (nem kötelező!!!). Ilyen függvénynek nincs törzse, mint az interface függvényeknek. Egy absztrakt osztály utódosztályában az absztrakt függvényt kötelező implementálni. Az utódosztályban ekkor az *override* kulcsszót kell a függvény fejlécbe írni.

Példa:

```
  abstract class os
  {
      private int e;
      public os(int i)
      {
          e=i;
      }
      public abstract int szamol();
      public  int E
      {
          get
          {
              return e;
          }
      }
  }
  class szamolo:os
  {
      public szamolo():base(3)
      {
          …
      }
      public override int szamol()    // kötelező implementáció
      {
          return base.E*base.E;
      }
  }
  class program
  {
      public static void Main()
      {
       os x= new os();  // hiba, mert absztrakt osztályból nem
                        // készíthetünk változót
          szamolo f = new szamolo();
          Console.WriteLine(f.szamol()); // eredmény: 9
      }
  }
```

VII.10. Virtuális tagfüggvények, függvényelfedés

A programkészítés során, ahogy láttuk, az egyik hatékony fejlesztési eszköz az osztályok öröklési lehetőségének kihasználása. Ekkor a függvénypolimorfizmus alapján természetesen lehetőségünk van ugyanazon névvel mind az ősosztályban, mind az utódosztályban függvényt készíteni. Ha ezen függvényeknek különbözők a paraméterei, akkor gyakorlatilag nincs is kérdés, hiszen függvényhíváskor a paraméterekből teljesen egyértelmű, hogy melyik kerül meghívásra. Nem ilyen egyértelmű a helyzet, amikor a paraméterek is azonosak.

VII.10.1. Virtuális függvények

A helyzet illusztrálására nézzük a következő példát. Definiáltuk a *pont* ősosztályt, majd ebből származtattuk a *kor* osztályunkat. Mindkét osztályban definiáltunk egy *kiir* függvényt, amely paraméter nélküli és az osztály adattagjait írja ki.

Példa:

```
class pont
{
    private int x,y;
    public pont()
    {
        x=2;y=1;
    }
    public void kiir()
    {
        Console.WriteLine(x);
        Console.WriteLine(y);
    }
}
class kor:pont
{
    private int r;
    public kor()
    {
        r=5;
    }
    public void kiir()
    {
        Console.WriteLine(r);
    }
}
class program
{
    public static void Main()
    {
        kor k=new kor();
        pont p=new pont();
        p.kiir();         //2,1
        k.kiir();         //5
        p=k;          //Egy őstípus egy utódra hivatkozik
        p.kiir();     //Mit ír ki?
    }
}
```

A program futásának eredményeként először a *p* pont adatai *(2,1),* majd a *kor* adata *(5)* kerül kiírásra.

A *p=k* értékadás helyes, hiszen *p* (*pont* típus) típusa a *k* típusának *(kor)* az őse. Azt is szoktuk mondani, hogy egy őstípusú referencia tetszőleges utódtípusra hivatkozhat. Ekkor a második *p.kiir()* utasítás is, a *p=k* értékadástól függetlenül a *2,1* értékeket írja ki! Miért? Mert az osztályreferenciák alapértelmezésben statikus hivatkozásokat tartalmaznak a saját függvényeikre. Mivel a pontban van *kiir*, ezért attól függetlenül, hogy időközben a program során (futás közbeni – dinamikus – módosítás után) a *p* már egy kör objektumot azonosít, azaz a kör *kiir* függvényét kellene meghívni, még mindig a fixen, fordítási időben hozzákapcsolt *pont.kiir* függvényt hajtja végre.

Ha azt szeretnénk elérni, hogy mindig a dinamikusan hozzátartozó függvényt hívja meg, akkor az ősosztály függvényét virtuálisnak *(virtual),* míg az utódosztály függvényét felüldefiniáltnak *(override)* kell nevezni, ahogy az alábbi példa is mutatja.

Példa:

```
class pont
{
    private int x,y;
    public pont()
    {
        x=2;y=1;
    }
    virtual public void kiir()
    {
        Console.WriteLine(x);
        Console.WriteLine(y);
    }
}
class kor:pont
{
    private int r;
    public kor()
    {
        r=5;
    }
    override public void kiir()
    {
        Console.WriteLine(r);
    }
}
```

Ez a tulajdonság az osztályszerkezet rugalmas bővítését teszi lehetővé, míg a programkód bonyolultsága jelentősen csökken.

VII.10.2. Függvényeltakarás, sealed függvény

Az öröklés kapcsán az is előfordulhat, hogy a bázisosztály egy adott függvényére egyáltalán nincs szükség az utódosztályban. Az alábbi példában a focicsapat osztálynak van *nevkiir* függvénye. Az utód *kedvenc_focicsapat* osztálynak is van ilyen függvénye, ezek virtuális függvények. A *new* hatására a *legkedvenc_focicsapat* osztály *nevkiir* függvénye eltakarja az ősosztály(ok) hasonló nevű függvényét.

Példa:

```
class focicsapat
{
    protected string csapat;
    public focicsapat()
    {
        csapat="UTE";
    }
    public void nevkiir()
    {
        Console.WriteLine("Kedves csapat a lila-fehér {0}",csapat);
    }
```

```
    }
    class kedvenc_csapat:focicsapat
    {
        public kedvenc_csapat()
        {
            csapat="Fradi";
        }
        new public void nevkiir()
        {
            Console.WriteLine("Kedvenc csapatunk a: {0}",csapat);
        }
    }
class legkedvenc_csapat : kedvenc_csapat
 {
     public legkedvenc_csapat()
     {
         csapat = "Debrecen";
     }
     new public void nevkiir()
     {
         Console.WriteLine("Legkedvenc csapatunk a: {0}", csapat);
     }
 }
  public class MainClass
  {
      public static void Main()
      {
        legkedvenc_csapat lk = new legkedvenc_csapat();
        lk.nevkiir();   // legkedvenc nevkiir hívás
        focicsapat cs = new legkedvenc_csapat();
        cs.nevkiir();   // kedvenc_csapat hívás
      }
  }
```

A *new* utasítás hatására egy öröklési sor megszakad, ezért a cs.nevkiir() illesztés a sorban csak a kedvenc_csapat osztályig jut, így a kedvenc_csapat osztályban definiált *nevkiir* függvényt hívja meg.

Ennek az ellenkezőjére is igény lehet, mikor azt akarjuk elérni, hogy az ősosztály függvényét semmilyen más formában ne lehessen megjeleníteni. Ekkor a függvényt, az osztály mintájára, véglegesíteni kell, azaz a *sealed* jelzővel kell megjelölni.

VIII. Kivételkezelés

Egy program, programrész vagy függvény végrehajtása formális eredményesség tekintetében három kategóriába sorolható.

1. A függvény vagy programrész végrahajtás során semmilyen „rendellenesség” nem lép fel.
2. A függvény vagy programrész aktuális hívása nem teljesíti a paraméterekre vonatkozó előírásainkat, így ekkor a „saját hibavédelmünk” eredményekénti hibával fejeződik be a végrehajtás.
3. A függvény vagy programrész végrehajtása során előre nem látható hibajelenség lép fel.

Ezen harmadik esetben fellépő hibajelenségek programban történő kezelésére nyújt lehetőséget a kivételkezelés *(Exception handling)* megvalósítása. Ha a második esetet tekintjük, akkor a „saját hibavédelmünk” segítségével, mondjuk valamilyen „nem használt” visszatérési értékkel tudjuk a hívó fél tudomására hozni, hogy hiba történt. Ez néha komoly problémákat tud okozni, hiszen például egy egész értékkel visszatérő függvény esetében néha elég körülményes olyan egész értéket találni, amelyik nem egy lehetséges valódi visszatérési érték. Így ebben az esetben is, bár a fellépő hibajelenség valahogy kezelhető, a kivételkezelés lehetősége nyújt elegáns megoldást.

A kivételkezelés lehetősége hasonló, mint a fordítási időben történő hibakeresés, hibavédelem azzal a különbséggel, hogy mindezt futási időben lehet biztosítani. A C# kivételkezelés a hibakezelést támogatja. Nem támogatja az ún. aszinkron kivételek kezelését, mint például billentyűzet- vagy egyéb hardvermegszakítás *(interrupt)* kezelése.

VIII.1. Kivételkezelés használata

A kivételkezelés implementálásához a következő új nyelvi alapszavak kerülnek felhasználásra:

try	mely kritikus programrész következik
catch	ha probléma van, mit kell csinálni
throw	kifejezés, kivételkezelés átadása, kivétel(ek) deklarálása
finally	kritikus blokk végén biztos végrehajtódik

A kivételkezelés használata a következő formában adható meg :

```
try      {
         // azon utasítások kerülnek ide, melyek
         // hibát okozhatnak, kivételkezelést igényelnek
         }
catch( típus [név])
         {
              // Adott kivételtípus esetén a vezérlés ide kerül
              // ha nemcsak a hiba típusa az érdekes, hanem az
              // is,hogy például egy indexhiba esetén milyen
              // index okozott 'galibát', akkor megadhatjuk a
              // típus nevét is, amin keresztül a hibát okozó
              // értéket is ismerhetjük. A név megadása opcionális.
         }
finally  {
              // ide jön az a kód, ami mindenképpen végrehajtódik
         }
```

A *try* blokkot követheti legalább egy *catch* blokk, de több is következhet. Ha több *catch* blokk van, azok sorrendje nem közömbös a típus miatt. Az elkapó blokkokat utódosztály- ősosztály sorrendben célszerű írni. Ha a *try* blokk után nincs elkapó *(catch)* blokk, vagy a meglévő *catch* blokk típusa nem egyezik a keletkezett kivétellel, akkor a keretrendszer kivételkezelő felügyelete veszi át a vezérlést.

A C# nyelvben a *catch* blokk típusa csak a keretrendszer által biztosított *Exception* osztálytípus, vagy ennek egy utódtípusa lehet. Természetesen mi is készíthetünk saját kivétel–típust, mint az *Exception* osztály utódosztályát.

Példa:

```
int i=4;
int j=0;
try
{
    i=i/j;   // 0-val osztunk
}
catch (Exception )
{
    Console.WriteLine("Hiba!");
}
finally
{
    Console.WriteLine("Végül ez a Finally blokk is lefut!");
}
```

A fenti példában azt láthatjuk, hogy a rendszerkönyvtár használatával, például a nullával való osztás próbálkozásakor, a keretrendszer hibakivételt generál, amit elkapunk a *catch* blokkal, majd a *finally* blokkot is végrehajtjuk.

A példa egész számokhoz kapcsolódik, így meg kell jegyezni, hogy gyakran használják az egész aritmetikához kapcsolódóan a *checked* és az *unchecked* módosítókat is. Ezek a jelzők egy blokkra vagy egy függvényre vonatkozhatnak.

Ha az egész aritmetikai művelet nem ábrázolható, vagy hibás jelentést tartalmaz, akkor a *checked* blokkban ez a művelet nem kerül végrehajtásra.

Példa:

```
int i=System.Int32.MaxValue;
checked
{
    i++;     // OverflowException kivétel keletkezik
}
…
int j=System.Int32.MaxValue;
unchecked
{
    j++;     // OverflowException kivétel nem keletkezik
}
Console.WriteLine(i);  // -2147483648 lesz a kiírt érték
                       // ez azonos a Syste.Int32.MinValue
                       // értékével
```

Természetesen nemcsak a keretrendszer láthatja azt, hogy a normális utasításvégrehajtást nem tudja folytatni, ezért kivételt generál, és ezzel adja át a vezérlést, hanem maga a programozó is. Természetesen az, hogy a programozó mikor látja szükségesnek a kivétel generálását, az rá van bízva.

Példa:

```
int i=4;
try
{
    if (i>3) throw new Exception(); // ha i>3, kivétel indul
```

```
    }
    catch (Exception )
    { // mivel az i értéke 4, itt folytatódik a végrehajtás
        Console.WriteLine("Hibát dobtál!");
    }
    finally
    {
        Console.WriteLine("Végül ez is lefut!");
    }
```

A kivételkezelések egymásba ágyazhatók.

Többféle abnormális jelenség miatt lehet szükség kivételkezelésre, ekkor az egyik „kivételkezelő” a másiknak adhatja át a kezelés lehetőségét, mondván „ez már nem az én asztalom, menjen a következő szobába, hátha ott a címzett” *(throw).* Ekkor nincs semmilyen paramétere a *throw*-nak. Ez az eredeti hibajelenség újragenerálását, továbbadását jelenti.

Példa:

```
    int i=4;
    try
    {
        if (i>3) throw new Exception(); // ha i>3, kivétel indul
    }
    catch (Exception )
    {
        Console.WriteLine("Hibát dobtál!");
        throw;        //a hiba továbbítása
        // ennek hatására , ha a program nem kezel további kivétel-
        // elkapást, a .NET keretrendszer lesz a kivétel elkapója.
        // így szabvány hibaüzenetet kapunk, majd a
        // programunk leáll
    }
```

VIII.2. Saját hibatípus definiálása

Gyakori megoldás, hogy az öröklődés lehetőségét használjuk ki az egyes hibák szétválasztására, saját hibatípust. Például készítünk egy *Hiba* osztályt, majd ebből származtatjuk az *Indexhiba* osztályt. Ekkor természetesen egy hibakezelő lekezeli az *Indexhibát* is, de ha a kezelő formális paraméterezése érték szerinti, akkor az *Indexhibára* jellemző plusz információk nem lesznek elérhetők! Mivel egy őstípus egyben dinamikus utódtípusként is megjelenhet, ezért a hibakezelő blokkokat az öröklés fordított sorrendjében kell definiálni.

Példa:

```
    public class Hiba:Exception
    {
        public Hiba(): base()
        { }
        public Hiba(string s): base(s)
        { }
    }
    public class IndexHiba:Hiba
    {
        public IndexHiba(): base()
        { }
        public IndexHiba(string s): base(s)
        { }
    }
        int i=4;
        int j=0;
        try
```

```
        {
            if (i>3) throw new Hiba("Nagy a szám!");
        }
        catch (IndexHiba h )
        {
            Console.WriteLine(h);
            // A konstruktor szöveg paraméterét adja meg.
            Console.WriteLine(h.Message);
        }
        catch (Hiba h )   // csak ez a blokk hajtódik végre
        {
            Console.WriteLine(h);
        }
        catch (Exception )
        {
            Console.WriteLine("Hiba történt, nem tudom milyen!");
        }
        finally      // és természetesen a finally is
        {
            Console.WriteLine("Finally blokk!");
        }
```

Ahogy korábban láttuk, minden objektum a keretrendszer *Object* utódjának tekinthető, ennek a típusnak pedig a *ToString* függvény a része, így egy tetszőleges objektum kiírása nem jelent mást, mint ezen *ToString* függvény meghívását.

A fenti példa így meghívja az *Exception* osztály *ToString* függvényét, ami szövegparaméter mellett kiírja még az osztály nevét és a hiba helyét is.

IX. Típusparaméter, Generikus típusok, metódusok

Egy függvény vagy osztály definiálásakor a legelőször feltett kérdés az szokott lenni, hogy milyen paramétere(i) legyen(ek) a függvénynek, milyen adatmezőket kell az osztályon belül léltrehozni. Ha mondjuk egy tevékenységnek kétféle típussal is meg kell birkóznia, akkor definiálhattunk két azonos nevű függvényt, mindegyiket a megfelelő paraméter fogadására felkészítve. (VI.4. Függvénynevek átdefiniálása)

Ez a megoldás egyszerű, csak az a legnagyobb baj vele, hogy nem a legtömörebb kódot eredményezi. A típusparaméter lehetőségének használata a hasonló esetekben tömörebb, sokkal kifejezőbb lehetőséget jelent. Ezt a lehetőséget gyakran „sablon” név alatt, -például C++ nyelvben (template)-, találjuk. A C# angol nyelvű környezete „Generic” névvel jellemzi ezt a tulajdonságot.

IX.1. Könyvtári típusparaméteres gyűjtemények

A könyvtári gyüjteményeket (tömb, lista, sor, ..) a *System.Collections* névtérben találhatjuk. Ezen hagyományos adattípusoknak megfelelő típusparaméterrel megvalósított változata a *System.Collections.Generic* névtérben található. Az üres projektváz is már a *System.Collections.Generic* névtérre való hivatkozást tartalmazza. Ez valószínűleg utalás akar lenni arra, hogy ha egy mód van rá, akkor javasolt a típusparaméteres változat használata.

A típusparaméter azonosítóját < > jelek között kell jelölni. Több típusparaméter is lehet, ekkor vesszővel kell elválasztani az azonosítókat.

A részletesebb magyarázat mellőzésével nézzük meg a könyvtári gyűjteményeket, és azok típusparaméteres megfelelőjét.

A Generic névtér típusaihoz az alábbi megfeleltetést adhatjuk a System.Collections névtérből.

System.Collections.Generic	System.Collections
Comparer<T>	Comparer
Dictionary<K,T>	HashTable
LinkedList<T>	-
List<T>	ArrayList
Queue<T>	Queue
SortedDictionary<K,T>	SortedList
Stack<T>	Stack
ICollection<T>	ICollection
IComparable<T>	System.IComparable
IDictionary<K,T>	IDictionary
IEnumerable<T>	IEnumerable
IEnumerator<T>	IEnumerator
IList<T>	IList

Példaként nézzük talán a beszédes nevűre változott Dictionary<K,T>, (szótár típus) használatát. Ennek a konstrukciónak, adatszerkezetnek az a jellemzője, hogy a több adatot tartalmazó szerkezetben az egyes elemekre nem (csak) számmal, hanem szöveges index-el tudunk hivetkozni. Ezt a típust gyakran asszociatív tömb névvel is nevezzük.

Példa:

```
static void Main(string[] args)
{
 Dictionary<string, string> szótár=new Dictionary<string,string>();
 szótár.Add("apple","alma");
 szótár.Add("grape","szőlő");
 szótár.Add("pear","körte");
 szótár.Add("peach","barack");
  // Az adatszerkezet kulcs-érték párjait egy anonym típusba
  // rakva végignézzük.
 foreach(var szó in szótár)
        Console.WriteLine("{0} : {1}",szó.Key,szó.Value);
  // Kiíratjuk, az adatpárok számát.
 Console.WriteLine(szótár.Count);
  // „Extension” metódus, egy tevékenységet végez ezen
  // (felsorolható) elemeken, egy transzformáció végrehajtásával.
  // Ezt a transzformációt jellemzően egy Lambda kifejezéssel adjuk
  // meg. Egy elemet helyettesítünk egy másik értékkel.
  //
 Console.WriteLine(szótár.Average(szó1 => szó1.Value.Length));
          // eredmény: 5
}
```

A típpusparaméterek bővebb lehetőségeiről, saját típusparamétert használó osztály, metódus használatáról a C# nyelvi füzet haladó, következő részében olvashatunk bővebben.

X. Adatlekérdezés: LINQ

A C# programozási nyelv talán legfontosabb tulajdonsága a *LINQ* (*L*anguage *IN*tegrated *Q*uery) lekérdező szolgáltatás. Ezt a programozási nyelvbe integrált adatlekérdezési lehetőséget már az elnevezéséből is ismerősnek halljuk. Méginkább így van ez akkor, amikor a tulajdonság jellemző kulcsszavait is leírjuk: *from, where, select*. Igen, a már régóta használt SQL nyelvből ismerősek ezek a szavak, és a C# nyelvbe integráltan azt a lehetőséget nyújtják, hogy lekérdező kifejezéseket alkossunk a segítségükkel. Ezeket a kifejezéseket gyakran *LINQ lekérdezés, vagy LINQ kifejezés* néven nevezzük.

A lekérdező kifejezések segítségével akár komplex szűrő, sorrendiséget megadó, csoportosításokat végző műveleteket adhatunk meg egyszerű formában. Ezek a kifejezések egy adatforráson végzik el a műveleteket. Ez a lekérdezés független az adatforrástól, ami lehet akár egy SQL adatbázis, egy XML dokumentum vagy egy .NET gyűjtemény(collection), ami a legegyszerűbb esetben egy szimpla tömb. Ebben a részben ezen legegyszerűbb esettel a tömbökkel foglalkozunk, a bővebb leírást a következő, haladó munkafüzet tartalmazza.

X.1. LINQ alapok

Mielőtt belemennénk a részletekbe, lássunk egy egyszerű, de teljes példát a LINQ használatára. A példa előtt elmondhatjuk, hogy egy lekérdezéses feladat végrehajtása jellemzően három részből áll.

1. Adatforrás definiálása
2. Lekérdező kifejezés megadása
3. A lekérdezés eredményének felhasználása

Példa:

```
   static void Main()
   {
// 1. adatforrás megadása. Ez legegyszerűbb esetben
                // egy tömb, jelen esetben egész tömb
      int[] halszám = new int[] { 4, 5, 7, 6, 3};
                //
// 2. Lekérdező kifejezés megadása.
      IEnumerable<int> halszámok =
         from hal in halszám
         where hal > 4
         select hal;

// 3. A lekérdezés végrehajtása. A kapott felsorolható
                // adatsoron (IEnumerable<int>) a foreach végiglépdel.
      foreach (int i in halszámok)
      {
         Console.WriteLine(i);
      }
                // Eredmény: 5, 7, 6
```

```
    }
```

X.2. Lekérdezés kifejezések készítése

A nyelvi elemek segítségével készített lekérdezések (LINQ) azt a célt szolgálják, hogy hatékony, könnyen olvasható lekérdezéseket, transzformációkat tudjunk készíteni.

A lekérdezések fontosabb jellemzői a következők:

- Könnyű használhatóság, sok C# nyelvi konstrukciót használnak.
- A lekérdezés eredménye szigorúan típusos (ahogy a nyelv is), bár sok esetben a fordítóra ráhagyjuk, hogy találja ki az eredmény típusát!
- A lekérdezés valójában a foreach utasításban hajtódik végre.
- Lekérdezést vagy kifejezés formában vagy metódus (extension method) formában írhatunk.
- Bár a fordító is átalakítja a kifejezést metódus formára, azért tanácsos a kifejezés formának a használata a jobb és könnyeb olvashatóság miatt.
- Lekérdezéseket, extension metódusokat felsorolható, lekérdezhető típusokon (IEnumerable<>, IQueryable<>) végezhetünk. Legegyszerűbb ilyen típus a tömb.

Egy lekérdezés utasítás jellemző formája kifejezéses alakban a következő:

eredmény = **from** név **in** forrás
[kiegészítő feltételek]
select elem vagy **group** elem;

Tehát a lekérdezés kifejezésforma egy from kulcsszóval kezdődik, és a select vagy group parancs zár. A kettő között lehet még from parancs, illetve a where, orderby, join, let parancs.

A metódus forma ehhez nagyon hasonlít:

eredmény = forrás.
[kiegészítő feltételek.]
Select(Lambda kifejezés);

A szűrési feltételek a Selecthez hasonlóan Lambda kifejezést várnak paraméterül.

Lássuk a legelső példánkat, mikor a legegyszerűbb adatforrásból (egész tömb) kérdezünk le adatokat.

Példa:

```
// egy tömb a forrás
  int[] halszám = new int[] { 4, 5, 7, 6, 3 };
// a legegyszerűbb lekérdezés, nincs feltétel, minden
// elemet megkapunk
  IEnumerable<int> halszámok =
      from hal in halszám  // nem hagyható el lekérdezésből
      select hal;

// Lekérdező kifejezés megadása. Eredmény típusát
// konkrétan megadjuk
  IEnumerable<int> halszámok1 =
```

```
            from hal in halszám
            where hal > 4    // elhagyható ha nem kell feltétel
            select hal;
        // A lekérdezés végrehajtása.
        foreach (int i in halszámok1)
        {
            Console.WriteLine(i);
        }
    // Eredmény: 5, 7, 6
    // A fenti kifejezés forma metódus (extension) hívásos
    // alakja
    // Eredmény típusát ráhagyjuk a fordítóra, hadd találja ki!
        var hsz = halszám.
                    Where(x => x > 4).
                    Select(x => x);
        foreach (int i in hsz)
        {
            Console.WriteLine(i);
        }
```

A lekérdezések bővebb kifejtése a C# nyelvi füzet haladó részében kerülnek tárgyalásra.

Irodalomjegyzék

1. Brian W. Kernigham, Dennis M. Ritchie : A C programozási nyelv
   Műszaki Könyvkiadó, Budapest 1985, 1988

2. Bjarne Stroustrup: C++ programming language
   AT&T Bell Lab, 1986

3. Tom Archer : Inside C#
   MS Press, 2001

4. Microsoft C# language specifications
   MS Press, 2001, 2007

5. John Sharp, Jon Jagger: Microsoft Visual C#.NET
   MS Press, 2002

6. Illés Zoltán: A C++ programozási nyelv
   ELTE IK, Mikrológia jegyzet, 1995-…

7. David S.Platt: Bemutatkozik a Microsoft.NET
   Szak Kiadó, 2001

8. Illés Zoltán: A C# programozási nyelv és környezete
   Informatika a felsőoktatásban 2002, Debrecen

9. David Chappell: Understanding .NET
   Addison-Wesley, 2002

10. Paolo Pialorsi, Marco Russo: Introducing Microsoft LINQ
    MS Press, 2007