Bevezetés a C# nyelvű objektumorientált programozásba

Segédlet

Szoftvertechnológia és -technikák

tárgyhoz

Benedek Zoltán

2019

BME Automatizálási és Alkalmazott Informatikai Tanszék

Tartalom

Bevezető 4
Visual Studio 4
.NET verziók 4
Visual Studio telepítése 4
Új solution létrehozása 5
Új solution létrehozása Visual Studio 2017 alatt 6
Új solution létrehozása Visual Studio 2019 alatt 6
A fejlesztés környezete 7
.NET/C# Hello World console app 9
Fordítás, futtatás és nyomkövetés 9
Solutionkezelés 11
Néhány hasznos funkció 11
Dokumentáció 13
Objektumorientált programozás alapjai C# nyelven - ShapesDemo példa 14
Osztály és objektum fogalma, egységbezárás 14
Konstruktor 18
Destruktor 19
Láthatóság szabályozása 19
Property 21
Statikus tagok 22
Öröklés, virtuális függvények 23
Absztrakt osztályok 27
Konstruktorok öröklésnél 27
Ős tagfüggvény hívása a felüldefiniáló függvényből 28
Behelyettesíthetőség, egységes kezelés , polimorfizmus 29
Generikus típusok használata 31
Kivételkezelés 31
Interface 33
Modern nyelvi eszközök 36
Property (tulajdonság) 36
Delegate (delegát, metódusreferencia) 38

Event (esemény) 41
Attribute 43

Bevezető

Jelen jegyzet egy rövid bevezetőként szolgál a C# nyelv alapjaiba azok számára, akik már egy másik nyelv – pl. C++, Java – kontextusában az objektumorientált nyelvek alapfogalmaival már megismerkedtek.

Visual Studio

Verzió: Visual Studio 2017-től már megfelel, de a mindenkori legfrissebb verzió javasolt, jelen jegyzet írásakor ez a 2019-es verzió.

A Windows platformon történő .NET alkalmazásfejlesztés messze legszélesebb körben elterjedt fejlesztőkörnyezete a Microsoft Visual Studio. A terméknek több változata létezik. A tárgy keretében a Professional vagy „Community Edition" használata javasolt. Az utóbbi alapvetően a Professional verzió funkcióit nyújtja egy-két kényelmi funkciótól eltekintve, ingyenesen. A BME hallgatók és dolgozók számára azonban oktatási céllal szabadon használható a Professional vagy ennél is komolyabb változatok is: a Visual Studio valamennyi változata a https://azureforeducation.microsoft.com/devtools honlapon hallgatók számára ingyenesen hozzáférhető (Azure Dev Tools for Teaching program).

Megjegyzés: jelen útmutató bizonyos képernyőmentései még egy korábbi Visual Studio verzióval készültek, a felületeken lényegi változás az újabb verziókban nincs.

.NET verziók

A .NET-re fejlesztett alkalmazások futtatásához szükség van arra, hogy az adott számítógépen a .NET keretrendszer megfelelő változata telepítve legyen. A fontosabb változatok:

.NET Framework

- Sokáig csak ez létezett
- Csak Windows platform
- Legtöbb szolgáltatást/funkciót jegyzet írásakor még ez nyújtja, de ez meg fog változni
- Legfrissebb változata (a jegyzet írásakor): .NET Framework 4.8

.NET Core

- Pár éve jelent meg először
- Keresztplatformos implementáció (Windows, Linux, Mac)
- Egyelőre szűkebb funkcionalitás – de folyamatosan bővül
- A Microsoft ezt fejleszti jelenleg aktívan
- Legfrissebb változata (a jegyzet írásakor): .NET Core 2.2

A két változat a jövőben egyesül, .NET 5 néven jelenik majd meg 2020-ban. Jelen útmutató a C# nyelv alapjaira fókuszál, egyszerű konzol (Console) alkalmazások keretében kerülnek az ismeretek bemutatásra. Erre a célra a „.NET Framework" és a „.NET Core" egyaránt megfelel, a két változatban számunkra nem lesz különbség.

Visual Studio telepítése

A Visual Studio telepítésekor gondoskodnunk kell arról, hogy vagy a .NET Framework, vagy a .NET Core telepítve legyen. Ha Visual Studio 2019-et használunk: a Visual Studio Installerben az „ASP.NET and web

development” és/vagy a „.NET desktop development” Workload-ot válaszuk ki. Az első gondoskodik a .NET Core, a második a .NET Framework és az ezekhez szükséges fejlesztőkörnyezeti komponensek telepítéséről gondoskodik. Ha más Visual Studio verziót használunk, akkor a Workload-ok nevei eltérhetnek, de jó eséllyel intuitíven is ki tudjuk választani a megfelelőket név alapján.

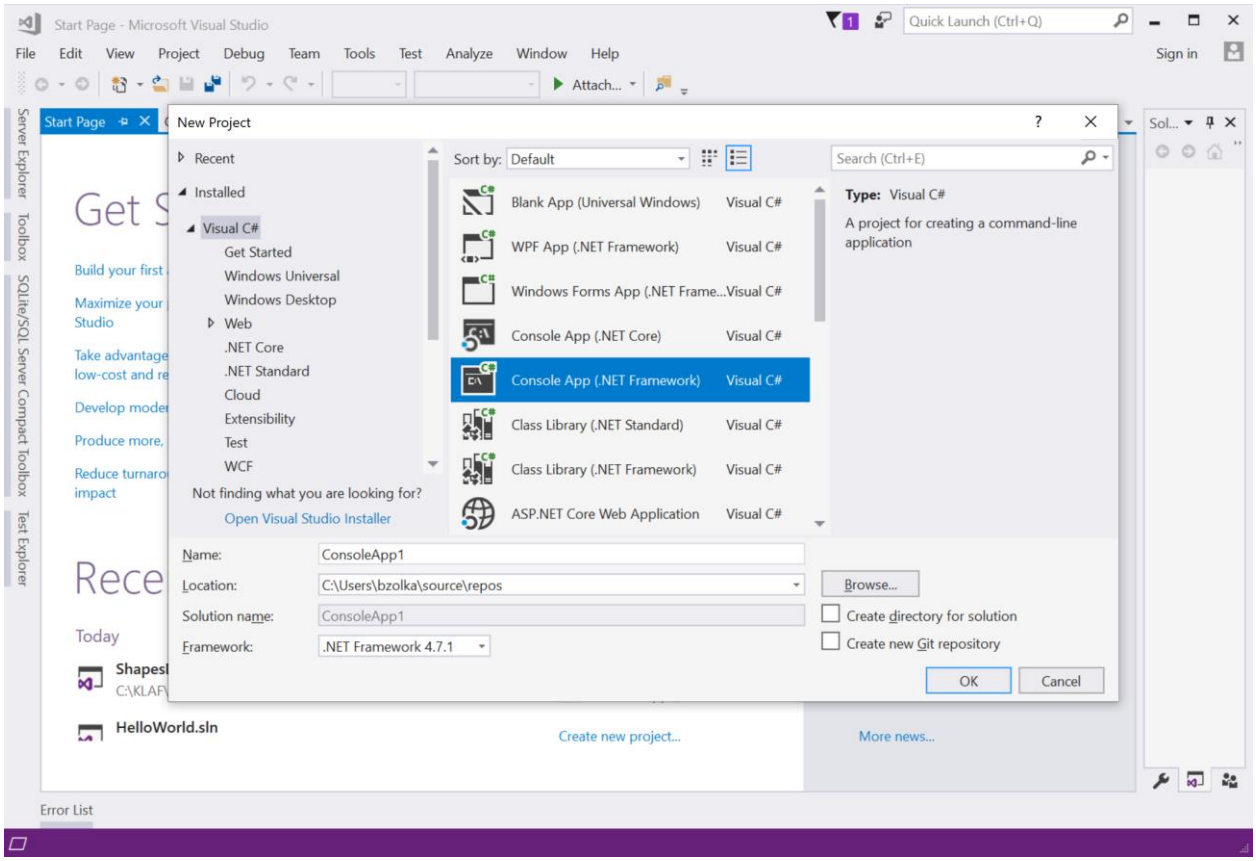
Megjegyzés: jelen jegyzethez mellékelt példakódok jelen pillanatban .NET Framework-öt igényelnek, így mindenképpen javasolt a „.NET desktop development” Workload telepítése (is).

Új solution létrehozása

Alkalmazásunk elkészítéséhez első lépésben egy új **solution**t kell létrehozni. A solution egy munkakörnyezetet fog össze. A solutionön belül hozhatók létre a **project**ek, a forrásfájlok (.cs, .cpp, .vb) pedig ezen projectekhez tartoznak. Minden project kimenete egy a project típusától függő lefordított állomány, amely lehet futtatható program (.exe), dinamikusan linkelhető könyvtár (.dll, .NET környezetben szerelvény/assembly a neve), C/C++ statikus könyvtár (.lib) stb. Egy projekt kimenete lehet egy website is. Ennek megfelelően egy solutionhöz akkor célszerű több projectet adni, ha az alkalmazásunkat több komponensre bontjuk, és ezeket egy közös környezetben szeretnénk kifejleszteni. Lényeges, hogy egy projekten belül csak egy adott nyelvet használhatunk (tehát nem keverhetjük a C#-ot a Visual Basickel, vagy C++-szal).

Új solution létrehozása Visual Studio 2017 alatt

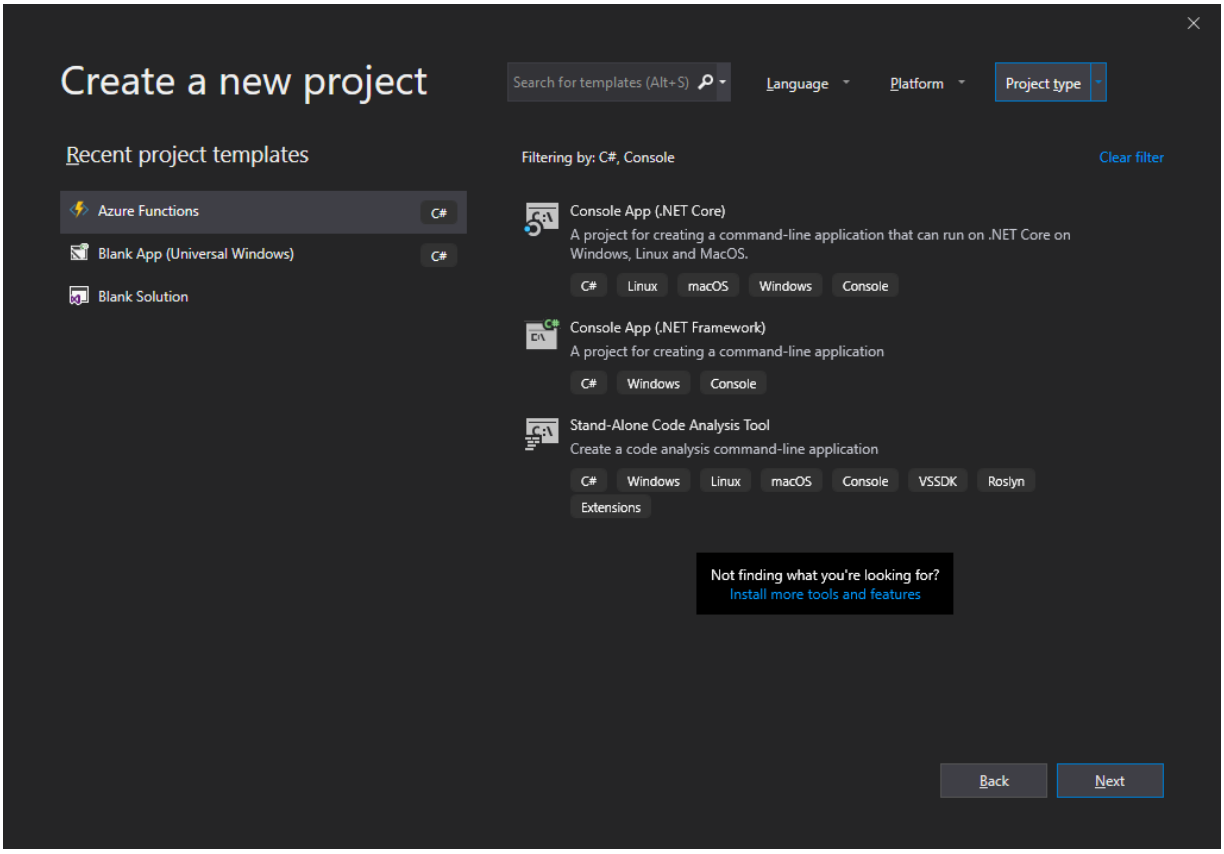
Új solution megalkotásához gyakorlatilag a solution első projectjét kell létrehozni: válasszuk ki a „File/New/Project” menüt. A megjelenő ablakban a project típusának válasszuk ki a „C#”-ot, Template-nek pedig a „Console App (.NET Core)”-t vagy a „Console App (.NET Framework)”-öt annak megfelelően, mely .NET változatra kívánjuk az alkalmazást fejleszteni. A „Name” mezőben adjuk meg a projekt nevét, a „Location” mezőben solutionünk fájljainak elérési útvonalát, „Solution Name” mezőben pedig a solution nevét az alábbiaknak megfelelően:



Új project létrehozása VS 2017 alatt

Új solution létrehozása Visual Studio 2019 alatt

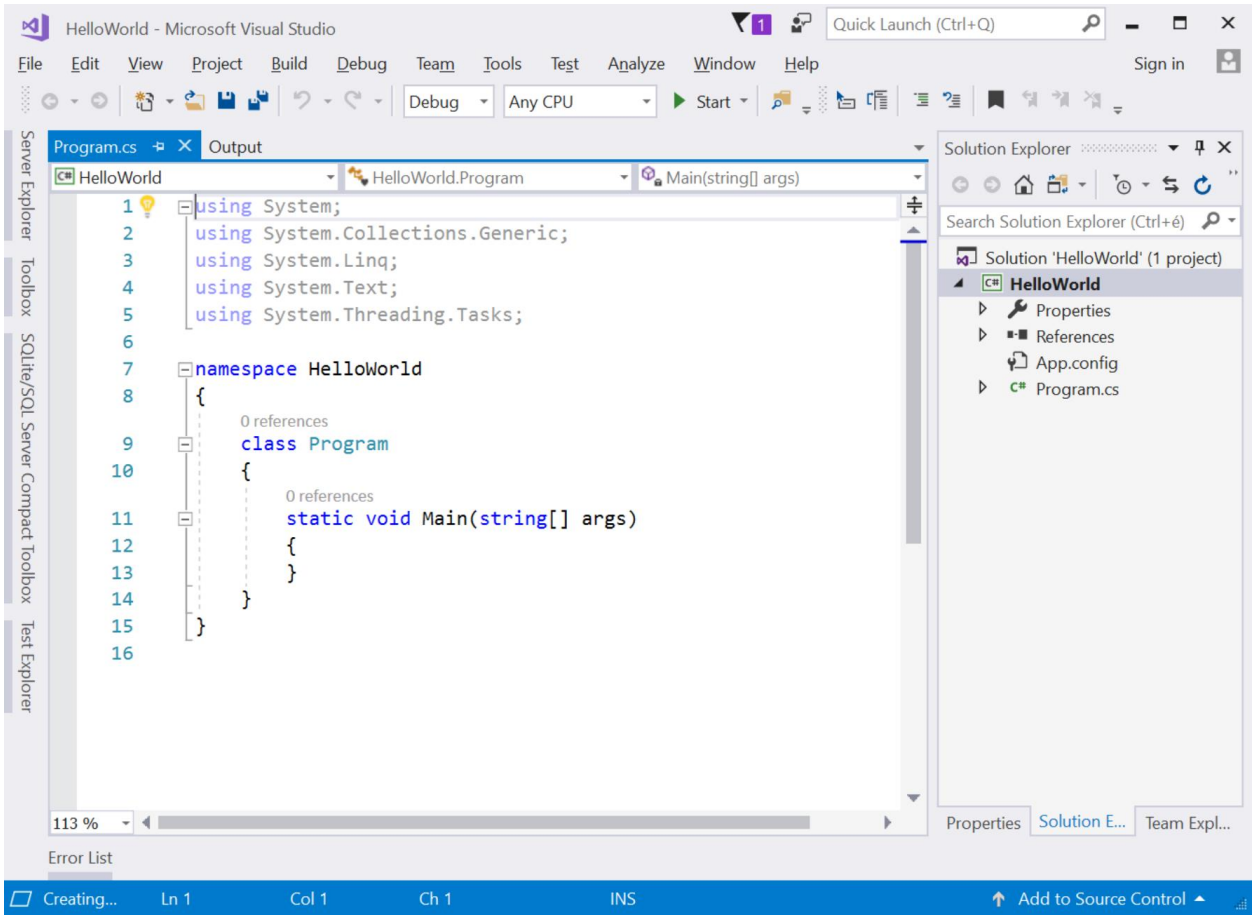
A Visual Studio indítását követően egy „Start” képernyő fogad minket, melyen a „Create a new project” lehetőséget válaszuk ki. Ekkor a „Create a new project” varázsló ablak jelenik meg. Itt könnyű elveszni a lehetőségek tengerében. Az ablak felső részében érdemes a „Language” szűrőben a C# nyelvet, a „Project type” szűrőben a „Console” lehetőséget kiválasztani. Ez követően template-nek válasszuk ki a „Console App (.NET Core)”-ot vagy a „Console App (.NET Framework)”-öt annak megfelelően, mely .NET változatra kívánjuk az alkalmazást fejleszteni.



A Next gombra kattintva a projekt létrehozásához már csak a projekt nevét, helyét és a solution nevét kell megadni. A solution nevének egy egyszerű esetben a projekt nevét adjuk meg.

A fejlesztés környezete

A solution létrehozását követően ismerkedjünk meg a fejlesztőkörnyezet használatának alapjaival.



A fejlesztőeszköz ablakai új solution létrehozását követően

A fontosabb képernyőnézetek a következők:

- **Dokumentumszerkesztő**: a képernyő legnagyobb részét adja, az aktuálisan kiválasztott forrásfájl, illetve fejlécfájl szövege szerkeszthető itt. Ha több fájl is meg van nyitva, tabfülekkel aktiválhatjuk az egyes fájlokat. A fenti képernyőn egy forrásfájl van megnyitva, a Program.cs.

- **Solution Explorer:** a „Solution Explorer” tabfülre kattintva aktiválható, a fenti képernyő jobb oldalán látható. Ebben a solution alatti projecteket, illetve az ezekhez tartozó forrásfájlokat láthatjuk. Egy fájlon duplán kattintva az adott fájl megnyílik a dokumentumszerkesztő ablakban.

- **Class View**: a „Class View” tabfülre kattintva aktiválható (ha nincs ilyen tabfül, a View menüből jeleníthető meg). Ebben a solutionhöz tartozó forrásfájlokban definiált osztályok nevének felsorolását láthatjuk. Az egyes osztályok, tagfüggvények és tagváltozók nevén kattintva a dokumentumszerkesztőben megnyílik a definíciót tartalmazó fájl, és a kurzor a definícióra ugrik. Ez a funkció remekül használható a solutionben való navigációra.

- **Output Window**: az „Output” tabfülre kattintva aktiválható. A fordítás során itt tudjuk nyomon követni a fordítás folyamatát, illetve itt kapunk információt az esetleges hibákról és figyelmeztetésekről.

.NET/C# Hello World console app

Mielőtt jobban elmélyülnénk a C# nyelv objektumorientáltságot szolgáló konstrukcióiba, érdemes egy minimalisztikus Hello World konzol alkalmazás elkészítésének alapjaival megismerkedni. Ehhez számos online anyag elérhető, pl. egy lehetőség a következő: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/hello-world-your-first-program

Illetve, mi is nézzük meg, hogy a korábban a varázsló által létrehozott solution-ben mi a Program.cs fájl tartalma, illetve egészítsük ki a parancssor argumentumok kiírásával:

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello world!");
            for (int i = 0; i < args.Length; i++)
                Console.WriteLine(args[i]);
            Console.ReadKey();
        }

    }
}
```

A kód tömör magyarázata a következő:

- A programunk egy osztályból áll, melynek neve `Program`.
- Programunk belépési pontja egy tetszőleges osztály **`Main`** nevű, egy `string[]` paraméterrel rendelkező **statikus** tagfüggvénye. Jelen esetben a `Program.Main`. A paraméter egy string tömb, ebben kapjuk meg a parancssor argumentumokat.
- Az osztályunk a `HelloWord` névtérben található (`namespace HelloWorld`).
- A konzolra (szabványos kimenet) írni a **`Console`** osztállyal tudunk, pl. annak **`WriteLine`** statikus tagfüggvényével. Mivel a `WriteLine` statikus, nem kell a `Console` osztályt példányosítani, hanem kényelmesen, az osztály nevén keresztül a „.” operátorral meghívható (C++-ban a „::” operátort használtuk).
- Billentyű lenyomásra várni a `Console.ReadKey()` művelettel tudunk.
- Karaktersorozatok kezelésére nem a `char*`, hanem a **`string`** típus használható. Ez a .NET osztálykönyvtár **`System.String`** osztályára képződik le.
- Tömböket a `[]` operátorral deklarálhatunk. Minden tömb tárolja a saját hosszát, ami a `Length` tagváltozón keresztül érhető el. A tömbök 0 indexel kezdődnek (mint pl. C++-ban).

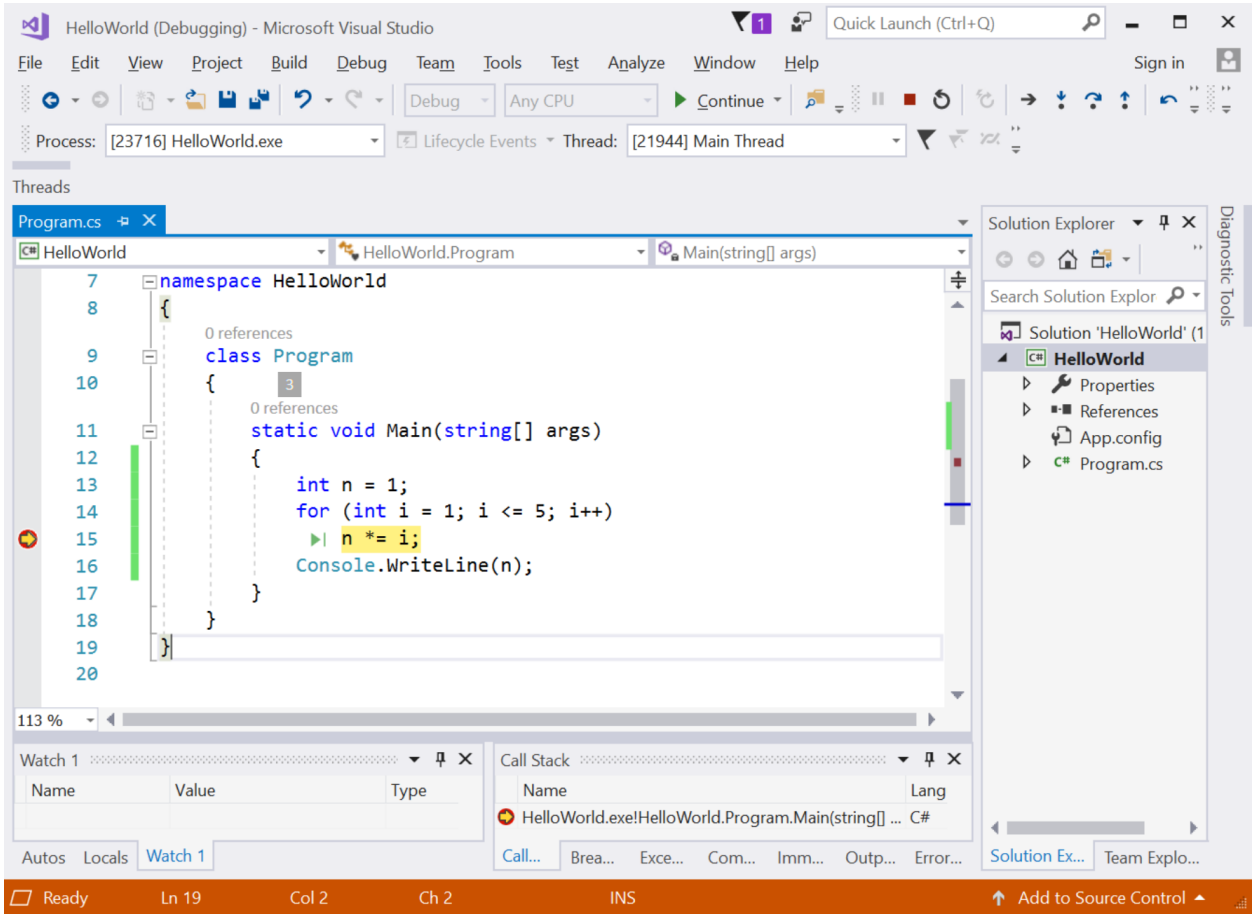
Egy alternatív online Hello World tutorial a következő: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/hello-world-your-first-program

Fordítás, futtatás és nyomkövetés

A „Build\Build HelloWorld” menü kiválasztásával (CTRL+SHIFT+B, vagy F6 konfigurációfüggően) fordíthatjuk le (build) a solutiont. Ez alatt azt a folyamatot értjük, amelynek eredményeként elkészülnek a solution projektjeinek kimenetei. Ez esetünkben a HelloWorld.exe elkészítését jelenti. Ha a build során a fordító hibát, figyelmeztetést állapít meg, akkor arról elsődlegesen az „Error List”, valamint az

„Output” ablakban értesülünk. Mindkét ablakban a hibaszövegen kattintva a dokumentumszerkesztőben a kurzor a hibát okozó sorra ugrik.

A „Debug\Start Debugging” paranccsal (vagy az eszközsáv gombjával, vagy az F5 gombbal) elindíthatjuk az alkalmazást nyomkövetési módban. Ha az előző build óta valamely forrás- vagy fejlécfájl megváltozott, akkor a futtatást automatikusan build előzi meg. A nyomkövetési módban történő indítás lényege az, hogy az esetlegesen beszúrt töréspontoknál megáll a futás, és megvizsgálható az egyes változók tartalma.[1] Ez a hibakeresés során nélkülözhetetlen funkció. Töréspont beszúrásához a dokumentumszerkesztő ablakban álljunk arra a sorra, amelyikre be kívánjuk szúrni a töréspontot, majd a jobb egérkattintásra feljövő menüben válasszuk ki a „Breakpoint/Insert Breakpoint” menüelemet (az F9 billentyű használata jóval egyszerűbb). A törésponttal ellátott sorokat a sorok elején ikon jelzi. Gyakorlásképpen tegyünk egy töréspontot az n*=i sorra, és indítsuk el az alkalmazást (F5 billentyű). Amikor a futás a töréspontnál megáll, visszatérhetünk a fejlesztőkörnyezethez:



52. ábra. *Töréspont alkalmazása*

Ekkor egy változó tartalmát megtekinthetjük úgy, hogy beírjuk a változó nevét a „Watch” ablakba („Watch” tabfüllel aktiválható), illetve a kurzorral ráállunk a változó nevére. Az utóbbi esetben a változó értéke tooltipben jelenik meg. Mindkét esetre mutat példát az előző ábra. Lehetőségünk van a programot lépésenként is futtatni a „Debug/Step Over” (F10) vagy a „Debug/Step Into” (F11) menüvel. A

[1] Ennek előfeltétele, hogy a kimeneti állomány tartalmazzon nyomkövetési infomációt. Ezt legkönnyebben úgy tudjuk elérni, hogy a *Debug* solution konfigurációt választjuk ki (a solution konfigurációkról a következő pontban lesz szó). Új solution létrehozása után automatikusan a *Debug* solution konfiguráció lesz kiválasztva.

különbség az, hogy ha az aktuális sor függvényhívás, és a függvény forrása elérhető, akkor a „Step Into” belelép a hívott függvénybe, a „Step Over” pedig átlépi azt. Néha jól használható a „Step Out” funkció (Shift+F11), amivel kiléphetünk az aktuális függvényből, vagyis a futás a hívó függvény következő soránál áll meg. A „Debug\Start Debugging” (F5) funkcióval a következő töréspontig, illetve kilépésig futtathatjuk a programot. Nyomkövetés közben a „Debug/Stop Debugging” menüvel (vagy az eszközsávon található  gombbal) bármikor megszakíthatjuk a program futását.

Solutionkezelés

A solution projectjeihez új forrásfájlokat a következőképpen adhatunk hozzá. Kattintsunk jobb gombbal a kiválasztott project nevén a Solution Explorerben, majd a menüben válasszuk ki az „Add/New Item” menüelemet. A megjelenő ablakban a „Categories” listában válasszuk ki a gyökeret vagy a „Code” elemet, majd a „Templates” listában a „Code File”, végül pedig a „Name” mezőben adjuk meg az újonnan létrehozandó fájl nevét. Az „Add” gomb hatására az új fájl létrejön és hozzáadódik a projecthez. Ha meglevő fájlokat kívánunk hozzáadni a solutionhöz, hasonlóképpen induljunk el, csak a felbukkanó menüben az „Add/Existing Item” menüelemet válasszuk ki, ezt követően pedig a megjelenő ablakban adjuk meg a felvenni kívánt fájlokat. Ha egy új C# osztályt szeretnénk valamelyik projecthez hozzáadni, akkor is ugyanígy kezdjük, a felbukkanó menüben az „Add/Class”-t válasszuk ki. A megjelenő ablakban megadhatjuk az osztály nevét.

Az eszközsávon található legördülőablakban (

) választhatjuk ki az aktuális solution konfigurációt. Minden solutionkonfigurációhoz egyedi fordító-, stb. beállítások tartoznak. Alapesetben a solution egy *Debug* és egy *Release* konfigurációval rendelkezik, de újak is létrehozhatók. A *Debug* konfigurációk kiválasztva és buildelve olyan kimeneti (esetünkben .exe) fájl keletkezik, amelynél töréspontok alkalmazásával lehetséges a nyomkövetés, a kimeneti fájl azonban nagyméretű és nem optimalizált. Ezt a konfigurációt a fejlesztés és a tesztelés során célszerű használni. Ha már megszabadultunk a bugoktól (legalábbis azt hisszük), a telepítéshez a *Release* konfigurációt célszerű alkalmazni. Ez kisméretű, optimalizált (így gyorsabb), de nem nyomkövethető kimenetet eredményez. Az egyes projektekre a „Project Properties” ablakban a fordítási, linkelési stb. beállítások minden solutionkonfigurációra egyedileg beállíthatók. A „Project Properties” ablak megnyitásához kattintsunk a projekt nevén jobb egérgombbal a Solution Explorerben és válasszuk ki a „Properties” menüelemet.

Néhány hasznos funkció

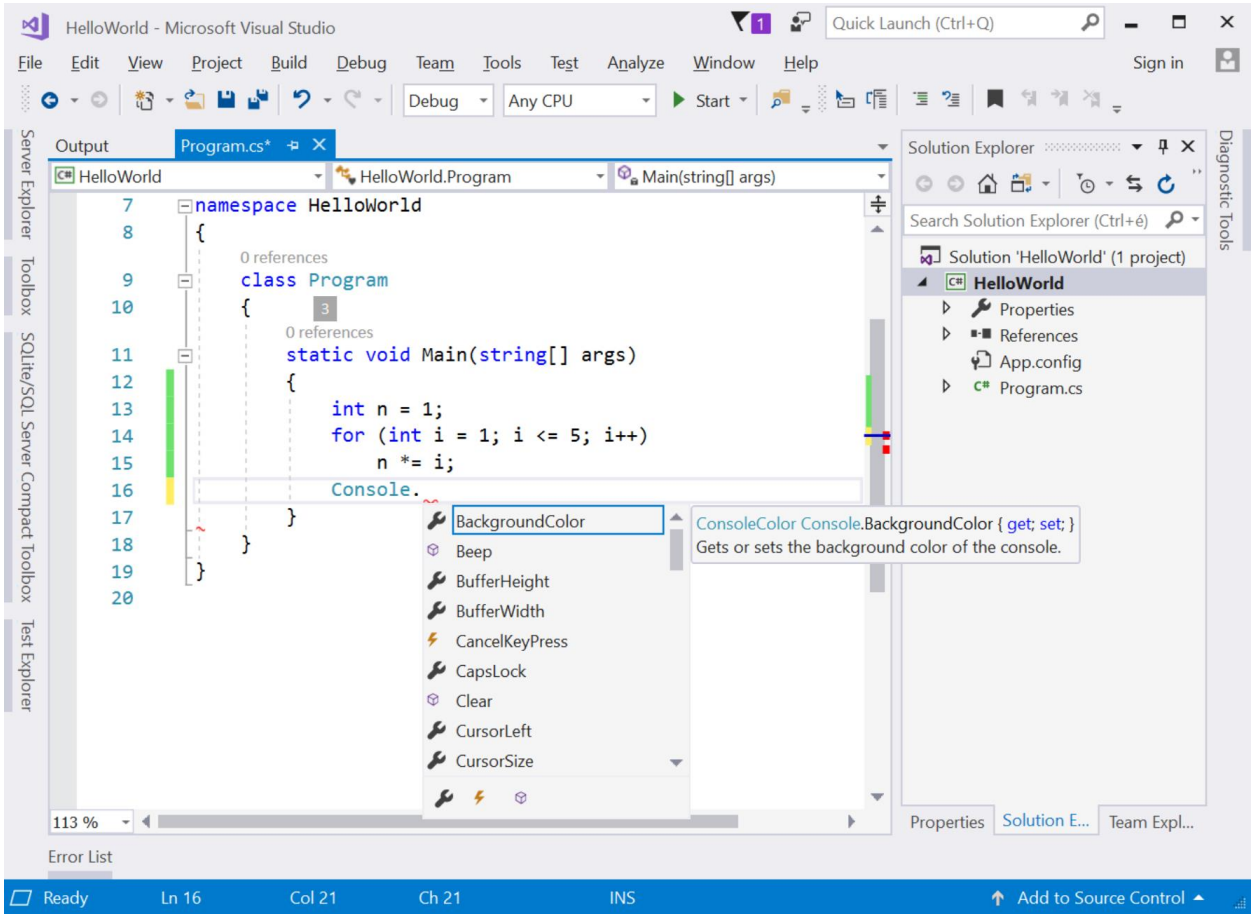
A következőkben felsorolunk néhány olyan hasznos szolgáltatást és funkciót, amelyet a kezdeti ismerkedést követően célszerű kipróbálni (ezek előadáson/gyakorlaton ismertetésre kerültek):

- **Display quick info:** a dokumentumszerkesztőben egy függvény, változó, osztály vagy struktúra nevére az egérrel ráállva tooltipben rövid leírást kapunk róla. Ez általában az adott elem deklarációjának megjelenítését jelenti.

- **Go to declaration:** a dokumentumszerkesztőben egy függvény, változó, osztály vagy struktúra nevén jobb gombbal kattintva megjelenő menüből elérhető funkció, mellyel az adott elem deklarációjára lehet ugrani.

- **Display word completion:** amikor a dokumentumszerkesztőben elkezdünk beírni egy nevet, megjelenik az adott hatókörben érvényes, az általunk beírt névvel kezdődő definíciók listája. A definíciók között a kurzorbillenyűkkel mozoghatunk, és az Enter billentyűvel választhatjuk ki a

beszúrni kívánt elemet. Ha az adott névvel csak egy definíció kezdődik, a lista meg sem jelenik, a definíció automatikusan beszúródik. A szolgáltatás közismert alternatív neve **intellisense**. Az alábbiakban erre látunk példát:



54. ábra. *Automatikus szókiegészítés*

Ez a funkció óriási segítség a mindennapi munkában. Lehetővé teszi, hogy hosszú, beszédes függvény-, osztály- és változóneveket használjunk, mégse kelljen azokat a használat során végig begépelni.

- **Display parameter description:** a dokumentumszerkesztőben egy függvény, illetve tagfüggvény nevét követően a „(”-et leírva tooltipben megjelenik a függvény paraméterlistája (ahogy az a függvénydeklarációban szerepel). Ez is nagy segítség, hiszen amennyiben egy függvény meghívásakor nem emlékszünk pontosan a megadandó paraméterekre, nem kell kikeresni a dokumentációból.

- **Display object member list:** a dokumentumszerkesztőben egy struktúra-, illetve objektumváltozó nevét követően a „.”-ot leírva automatikusan listázódnak a tagfüggvényei és a tagváltozói.

- **Error list:** a fordítási és linkelési hibák, figyelmeztetések táblázatos listája. A „View/Other Windows/Error List” menüvel jeleníthető meg. Egy adott során duplán kattintva az adott hibára, illetve figyelmeztetésre ugorhatunk.

- **Find in Files (CTRL+SHIFT+F):** valamennyi fájlban keres a megadott szövegre.

Dokumentáció

Jelen dokumentáció fókusza a C# nyelv, és nem a .NET keretrendszer. Online dokumentáció többek között itt érhető el:
https://docs.microsoft.com/en-us/dotnet/csharp/, melyen belül kezdőknek kiemelten hasznos:

- Get started with C#
- Tutorials (ezen belül az alapok)
- C# concepts
- C# programming guide
- …

Online formában számos útmutató, leírás érhető el. Egy osztály, művelet vonatkozásában célszerű arra a „Google”-ben rákeresni, a találatok között tipikusan az első helyen jelenik meg a referencia leírás.

Objektumorientált programozás alapjai C# nyelven - ShapesDemo példa

A fejezetben a következő – nagyrészt Java-ból vagy C++-ból már ismert - fogalmakat kívánjuk áttekinteni, illetve ezek C#-beli szintaktikáját megismerni:

- Objektum, osztály
- Tagváltozó, tagfüggvény
- Konstruktor, destruktor (dispose)
- Láthatóság szabályozása
- Statikus tagfüggvény
- Tulajdonság (property) – csak .NET
- Öröklés:
    - Ősosztály, gyerekosztály
    - Virtuális függvény, polimorfizmus, heterogén kollekciók
- Névtér
- Interfész
- Kivételek

A fogalmak egy átfogó, nagyobb példán keresztül kerülnek ismertetésre. **A példa forráskódja – illetve ennek valamennyi fontosabb köztes állapota – megtalálható a jelen jegyzetet tartalmazó zip állományban.**

Feladat: Írjunk olyan C# nyelvű konzol alkalmazást, mely támogatja különböző síkbeli alakzatok (téglalap, kör, stb.) kezelését. Minden alakzat rendelkezik egy koordinátával (x, y), valamint olyan további paraméterekkel, melyek lehetővé teszik területük kiszámítását. Tegyük lehetővé a memóriában tárolt alakzatok paramétereinek kényelmes képernyőre listázását (a területtel együtt), valamint az alakzatok adatfolyamba (pl. fájl) kiírását és onnan való betöltését.

Osztály és objektum fogalma, egységbezárás

Vegyük fel a szükséges struktúrákat `Rect.cs`-be, és a `Circle.cs`-be:

```
// Rect.cs
namespace HelloWorld
{
    public struct Rect
    {
        public int x;
        public int y;
    }
}
```

```
// Cirlce.cs
namespace HelloWorld
{
    public struct Circle
    {
        public int x;
        public int y;

        public int r;
```

```
    }
}
```

Írjuk meg az alábbi függvényeket:

- **`PrintName`**: kiírja az alakzat típusát (téglalapnál „Rect”, körnél „Circle”).
- **`PrintParams`**: kiírja az alakzat paramétereit (téglalapnál x és y, körnél x, y, r)
- **`GetArea`**: visszaadja az alakzat területét
- **`Print`**: kiírja az alakzat típusát, paramétereit és területét egyben

Tagfüggvényként írjuk meg: a függvényt bevisszük a struktúrába, melyen dolgozik. Így nem kell a függvénynek átadni a változót/objektumot paraméterül, melyen dolgozik (pl. a `GetArea` függvénynek nincs `Rect` paramétere).

A `Rect` struktúra:

```
// Rect.cs
public struct Rect
{
    public int x;
    public int y;
    public int w;
    public int h;

    public double GetArea() { return w * h; }

    public void PrintName() { Console.Write("Rect"); }

    public void PrintParams()
    {
        Console.WriteLine(" x: " + x.ToString() + ", y: " + y.ToString()
             + ", w: " + w.ToString() + ", h: " + h.ToString());
    }

    public void Print()
    {
        PrintName();
        PrintParams();
        Console.WriteLine(" Area: " + GetArea());
    }
}
```

Figyeljük meg a `PrintParams` függvényben, hogy a +operátor egy olyan stringgel tér vissza, mely a balérték és jobbérték string összefűzve. Egy másik érdekesség: az `x` és `y` tagváltozók típusa int, amely egy beépített típus. Ugyanakkor ezek a beépített típusok is a `System.Objectből` származnak, így öröklik a `ToString()` stringé alakító műveletet. Tanulság: a `ToString` minden típusra használható, egyszerű típusokra is.

A `Circle` struktúra:

```
// Circle.cs
public struct Circle
{
    public int x;
    public int y;
    public int r;

    public double GetArea() { return r * r * 3.14; }

    public void PrintName() { Console.Write("Circle"); }

    public void PrintParams()
```

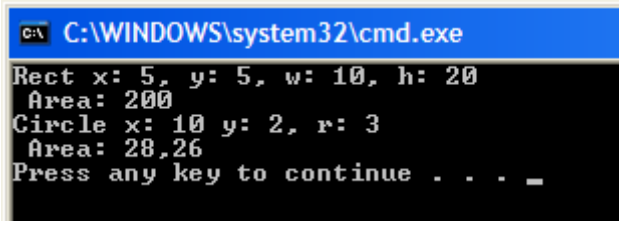
```
    {
        Console.WriteLine(
             string.Format(" x: {0} y: {1}, r: {2}", x, y, r)
         );
    }

    public void Print()
    {
        PrintName();
        PrintParams();
        Console.WriteLine(" Area: " + GetArea());
    }

}
```

A `GetArea`, `PrintName` és `PrintParams` függvényeket a kör viselkedésének megfelelően írtuk meg. A `Print` törzse változatlan. A `PrintParams`-ban most a `string.Format`-ot használtuk formázott string előállítására (nézzük meg a `System.String` osztály `Format` statikus tagfüggvényének dokumentációját). Egy kis kitérő: a `string` referencia típus (ennek jelentésére rövidesen kitérünk). Amikor a `Rect` struktúrában a + operátorral fűztünk össze stringeket, minden egyes + esetén egy új string jött létre, mely feleslegesen foglalja a memóriát és terheli a szemétgyűjtőt. Ez a probléma a `string.Format` esetén nem jelentkezik, így nagyobb számú string összefűzésére ez a módszer javasolt (vagy a `StringBuilder` osztály használata).

Használjuk a `Rect` és `Circle` struktúráinkat:

```
class Program
{
    static void Main(string[] args)
    {
          Rect r1;
          r1.x = 5;
          r1.y = 5;
          r1.w = 10;
          r1.h = 20;
          r1.Print();

          Circle c1;
          c1.x = 10;
          c1.y = 2;
          c1.r = 3;
          c1.Print();
    }
}
```

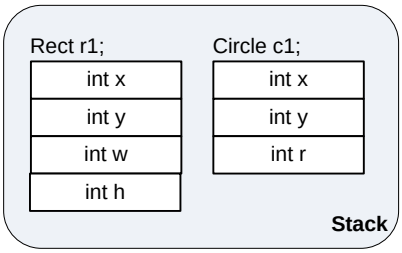
A kimenet a következő:



A tagfüggvény pont úgy használható a „.” operátorral, mint a tagváltozó. Meghívja a „.” előtti struktúrára/objektumra a „.” utáni műveletet. A tagfüggvény implicit módon megkapja a

struktúrát/objektumot, elérhetők a tagváltozók (lásd pl. `GetArea` tagfüggvény implementáció korábban).

Mi a memóriakép az előző példa esetében a `Main` függvényből való kilépés előtt? Mivel r1 és c1 struktúrák, a memória verem (stack) részén foglalnak helyet közvetlenül, lokális változóként:



A következő lépésben a `Rect` és a `Circle` struktúradefiníciókban cseréljük le a **`struct`** kulcsszót **`class`**-ra (minden más változatlan):

```
// Rect.cs
public class Rect
{ ... }

// Circle.cs
public class Circle
{ ... }
```

Ez egy lényeges változást eredményez: a `stuct` ún. érték típust definiált, a `class` pedig referencia típust. A lényegi jellemzők a következők:

- **Érték típus (value type):** int, char, decimal, double, float, bool, enum, stb. egyszerű típusok tartoznak ide, illetve amit mit definiálunk a **`struct`** kulcsszóval. Inline módon foglalnak helyet összetett típusokban. Lokális változónál a vermen foglalódik nekik hely. Függvényparaméter átadáskor pont úgy kezelődnek, mint C++-ban vagy Java-ban az int, vagy a hasonló „elemi" típusok: másolat készül az eredeti adatról. Gyors az allokációjuk. Korlátozások: nem örökölhetnek, nem lehet belőlük származni. Interfészt viszont implementálhatnak (lásd később).

- **A referencia típus (reference type):** két részből áll: egy hivatkozásból (referencia, mutató), mely alapértelmezésben null, és magából a hivatkozott objektumból, mely a felügyelt heapen foglal helyet (és a garbage collector gyűjti be, ha már nincs rá hivatkozás). Ez utóbbinak nekünk kell a **`new`**-val helyet foglalni. Ilyenek a .NET beépített osztályai (pl. string, File, stb.), a tömbök, illetve azok a típusok, melyeket mi hozunk létre a **`class`** kulcsszóval. Az interfészek is ide tartoznak, később lesz róluk szó.

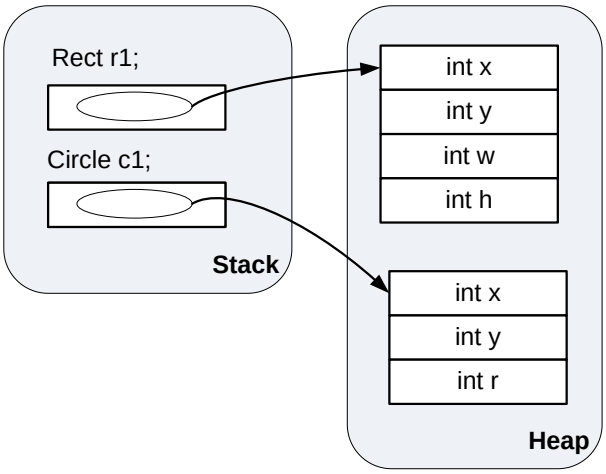
Ez esetben a `Rect` és a `Circle` már nem struktúra, hanem osztály. Ennek példányai pedig az objektumok: `r1` és `c1`. Pontosabban az `r1` és `c1` valójában egy-egy hivatkozás (referencia, mutató), melynek kezdőértéke **`null`**. A tényleges objektumnak a `new` operátorral kell helyet foglalni:

```
class Program
{
    static void Main(string[] args)
    {
        Rect r1; // (*1)
        r1 = new Rect();//(*2)
        r1.x = 5;
```

```
        r1.y = 5;
        r1.w = 10;
        r1.h = 20;
        r1.Print();

        Circle c1 = new Circle();
        c1.x = 10;
        c1.y = 2;
        c1.r = 3;
        c1.Print();
    }
}
```

- A (*1) sorban `r1` egy hivatkozás (referencia, mutató), értéke null.
- A (*2) sorban lefoglalunk egy Rect objektumnak helyet, és az `r1`-et ráállítjuk.
- Ez követően `r1`-el érhető el a hivatkozott/mutatott objektum a „.” operátorral (olyan, mint C++-ban a ->, vagy referencia esetén a „.”).

A memóriakép a `Main` függvény végén a következő:



Lényeges, hogy a GC csak a felügyelt heapen takarít. A stack-en az objektumok foglalása/felszabadítása hasonló módon történik, mint a C++ lokális változók/paraméterek esetében (stackpointer állítás, push, pop).

Konstruktor

Írjunk egy inicializáló `Init` függvényt a `Rect`-hez és a `Circle`-höz is. Nézzük most csak a `Circle`-re:

```
public class Circle
{
    ...
    public Init(int x, int y, int r)
    {
        this.x = x;
        this.y = y;
        this.r = r;
    }
    ...
}
```

Megjegyzés: a paramétereknek pont olyan nevet adtunk, mint a tagváltozóknak: névütközés lenne, de a this a tagfüggvényben egy implicit hivatkozást jelent az objektumra, melyre az adott tagfüggvényt meghívták. Ezzel lehet megkülönböztetni.

Probléma: elfelejtetjük meghívni az `Init` függvényt, az objektum ekkor inkonzisztens állapotban lesz (`x`, `y` és `r` nulla, `r` esetében ez nem elfogadott – nem szeretnénk hogy létezhessen nulla sugarú kör). Megoldás: Írjunk inkább konstruktort. A konstruktor az objektum létrehozásakor automatikusan lefut. Ebben inicializálhatjuk az objektumot, hogy már a létrehozásakor is konzisztens állapotban legyen. Közönséges tagfüggvény, neve megegyezik az osztály nevével, és nincs visszatérési típus. Túlterhelhető (overload, több verzió eltérő paraméterlistával).

Írjuk át konstruktorra:

```
public class Circle
{
    ...
    public Circle(int x, int y, int r)
    {
        this.x = x;
        this.y = y;
        this.r = r;
    }
    ...
}
```

A `Rect` hasonló módon átírható.
Használjuk is konstruktorainkat:

```
class Program
{
    static void Main(string[] args)
    {
        Rect r1; // (*1)
        r1 = new Rect(5, 5, 10, 20);
        r1.Print();

        Circle c1 = new Circle(10, 2, 3);
        c1.Print();
    }
}
```

Destruktor

Akkor fut le, amikor az objektumot a GC begyűjti. Szintaktika hasonló a C++-hoz (~osztálynév), de a C++-szal ellentétben nagyon ritkán kell használni, nem is írunk a példánkban. Míg C++-ban a destruktor szerepe általában az, hogy az objektum által dinamikusan lefoglalt erőforrásokat (tipikusan memóriát) felszabadítsuk (memória esetén delete). Erre .NET környezetben nincs szükség, hiszen a GC begyűjti az objektumunk által dinamikusan lefoglalt felügyelt erőforrásokat is (a delete nem is létezik). Szerepe .NET-ben a nem felügyelt erőforrások felszabadítása a dispose mintával karöltve, erről a későbbiekben lesz szó.

Láthatóság szabályozása

A célunk az, hogy az objektumaink mindig konzisztens állapotban legyenek. Ne lehessen kívülről elrontani belső állapotukat, még véletlenül se. A fenti példánkra ez nem teljesül, a következő lefordul és fut is:

```
class Program
{
    static void Main(string[] args)
    {
        Circle c1 = new Circle(10, 2, 3);
        c1.r = -10;
        c1.Print();
    }
}
```

Márpedig egy kör sugara nem lehet -10. A megoldást a láthatóság szabályozása jelenti. Ennek lényege, hogy a tagváltozóink, tagfüggvényeink előtti public, private, stb. kulcsszavakkal adjuk meg, ki férhet hozzá. A lehetőségek:

- **`private`**: csak az adott osztály tagfüggvényei számára látható (ez az alapértelmezett)
- **`public`**: minden osztály számára látható
- **`protected`**: az adott osztály és leszármazottai számára látható
- **`internal`**: csak az adott szerelvényen (assembly) belül látható (ez a későbbi tanulmányok során lesz érthető)
- **protected internal**: a protected és az internal kombinációja

Alakítsuk át céljainknak megfelelően a `Circle` osztályt:

```
public class Circle
{
    private int x;
    private int y;
    private int r;

    public int GetR()
    {
        return r;
    }

    public void SetR(int value)
    {
        if (value < 0)
            throw new ArgumentNullException("value");
        r = value;
    }
    ...
}

class Program
{
    static void Main(string[] args)
    {
        Circle c1 = new Circle(10, 2, 3);
        c1.x = -10; // fordítási hiba
        c1.SetX(-10); // futás közben hiba
        c1.Print();
    }
}
```

A megoldás lényege a következő:

- Legyenek az `x`, `y`, `r` tagváltozók privátok. Ezzel elérjük, hogy a következő sor le se fordul:

```
        c1.r = -10;
```

- Az `r`, `x`, `y` lekérdezéshez írjuk egy publikus `GetR` függvényt (a fenti példa csak az `r`-re mutatja a megoldást, a többi analóg módon megtehető).

- Az `r`, `x`, `y` állításhoz írjunk egy `SetR` függvényt. Ha a hívó érvénytelen értéket ad meg, dobjunk kivételt. Így a kódunk lefordul ugyan, de futás közben hibát jelez, nem marad rejtve a hiba (ami a legnagyobb baj lenne). Érvénytelen paraméter esetén a dobott kivétel típusa `ArgumentException` legyen.
- Konvenció: a privát védett tagok nevét kezdjük kisbetűvel (pl. `saveData`()), a publikusakét naggyal (pl. `SaveData`())

Property

A property (tulajdonság) egy új nyelvi elem, a C++ és Java nem támogatják. Mint az előző példánkban láttuk, az osztályaink tagváltozóit általában nem célszerű publikussá tenni, mert akkor nem garantálható az objektumok konzisztens állapota (pl. kör sugarának negatív érték is adható). Erre klasszikus megoldás az, ha a tagváltozókat védetté tesszük, majd lekérdezésükhöz egy `GetXXX`, szabályozott beállításukhoz egy `SetXXX` tagfüggvényt vezetünk be (mint az előző példánkban a `GetR` és `SetR`). A C# nyelvben erre egy szintaktikailag egyszerűbb megoldás is van, ennek neve property (tulajdonság). Valósítsuk meg a sugár elérését propertyvel:

```
public class Circle
{
    ...
    private int r;

    public int R
    {
        get { return r; }
        set
        {
            if (value < 0)
                throw new ArgumentNullException("value");
            r = value;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        Circle c1 = new Circle(10, 2, 3);
        c1.R = 10; // A set ág hívódik
        Console.WriteLine(c1.R); // A get ág hívódik
        c1.R = -10; // A set ág hívódik: hibajelzés futás közben

        c1.Print();
    }
}
```

Gyakorlatilag nem történt más, mint a korábbi `GetR` és `SetR` függvényünket összeolvasztottuk egy R nevű propertyben. A főbb jellemzők a következők:

- A definíciója a tagváltozókéhoz hasonló, de megadunk egy **`get`**`{}` és egy **`set`**`{}` ágat is.
- A `get{}` ág a tulajdonság lekérdezésekor fut le. Ha nem írunk `get{}` ágat, akkor a tulajdonság nem kérdezhető le.
- A `set{}` ág a tulajdonság állításakor fut le. Ha nem írunk `set{}` ágat, akkor a tulajdonság nem állítható be. A `set` ágban a **`value`** kulcsszó tartalmazza az új, beállítandó értéket.

- A tulajdonság egy közönséges nyelvi elem lett, osztályoknak most már nem csak tagváltozói és tagfüggvényei lehetnek, hanem tulajdonságai is.
- Egy objektum adott tulajdonságának elérése olyan szintaktikával történik, mintha az objektum tagváltozója lenne.

Önálló feladat: a fenti mintájára tegyük elérhetővé az `x` és `y` védett koordinátákat `X` és `Y` nevű propertyk segítségével a `Circle` és `Rect` esetében is. A koordináták 0 vagy nagyobb értéket vehetnek fel.

Statikus tagok

A jelentése ugyanaz, mint C++-ban vagy Java-ban:

- **Statikus tagváltozó**: Az osztály minden objektumára közös változó (nem objektumonként foglalódik neki tárhely).
- **Statikus tagfüggvény**: Objektum nélkül hívható, az osztály nevén keresztül (pl. `Console.WriteLine`).

Statikus tagváltozó és tagfüggvény a static kulcsszóval definiálható.

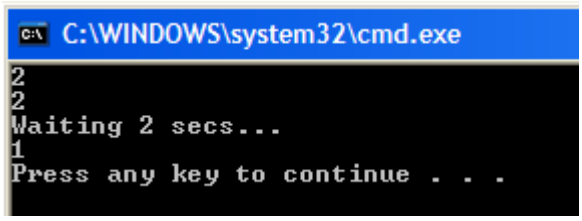
Feladat: írjunk olyan osztályt, amely nyilvántartja, hány objektum létezik belőle.

```
class InstaceCounter
{
    static int count = 0;

    public InstaceCounter() { count++; }

    ~InstaceCounter() { count--; }

    public static int Count
    {
        get { return count; }
    }
}
```

Egy kicsit tornáztassuk is meg:

```
class Program
{
    static void Main(string[] args)
    {
        InstaceCounter ic1 = new InstaceCounter();
        InstaceCounter ic2 = new InstaceCounter();
        Console.WriteLine(InstaceCounter.Count);
        ic1 = null;
        Console.WriteLine(InstaceCounter.Count);
        GC.Collect();
        // Elaltatjuk a hívót 2000 msec-ra
        Console.WriteLine("Waiting 2 secs...");
        System.Threading.Thread.Sleep(2000);
        GC.Collect();
        Console.WriteLine(InstaceCounter.Count);
    }
}
```

A program kimenete a következő:



Az osztályunk neve `InstanceCounter`. A `count` nevű védett statikus tagváltozóban tartja nyilván az objektumok számát. Ezt a konstruktorban megnöveljük, a destruktorban csökkentjük eggyel. A példánk remekül szemlélteti a Garbage Collector (GC) működését is:

- A `Main` függvény elején létrehozunk két objektumot, így az első kiíráskor a darabszám 2.
- Az `ic1` referenciát `null`-ra állítjuk, így az általa korábban hivatkozott objektumra már nem hivatkozik senki, begyűjthető a GC által. Ennek ellenére a következő sorban a darabszám még mindig 2. Ennek oka az, hogy a GC még nem gyűjtötte be az objektumunkat, hiszen az csak időnként fut le a háttérben, és csak akkor takarít, ha „szűkében vagyunk" a memóriának.
- A `GC.Collect()`-tel kikényszerítjük a GC lefutását, majd várunk 2 másodpercet (a gyakorlatban ne hívjunk `GC.Collectet`, mert a GC működése önhangoló, a `GC.Collect` hívásával feleslegesen terheljük a rendszert).
- Az utolsó sorban a darabszám így már 1.

Öröklés, virtuális függvények

Térjünk vissza a korábbi, alakzatos példánkhoz. A következő problémákat vegyük észre:

- Az x és y tagváltozó minden alakzat osztályban megjelenik.
- Bár a Print törzse ugyanaz, minden alakzat osztályba meg kellett írni. Ugyanez igaz az `X` és `Y` propertykre.

Az ilyen **kódduplikáció** a fejlesztők legősibb ellensége: egyrészt többet kell kódolni, de ennél is sokkal nagyobb baj, hogy hosszú távon nagyon meg tudja nehezíteni a kód karbantartását. Ha hibás a kód, sok helyen kell javítani, vagy ha változik, bővül, azt is sok helyen kell átvezetni. Bár esetünkben ez nem jelentős, de ha további alakzatokat vezetünk be, akkor már számottevővé válik.

A probléma megoldását az **öröklés** jelenti. Vezessünk be egy `Shape` nevű ősosztályt, ebbe vegyük fel a közös tagváltozókat és tagfüggvényeket (`x`, `y`, `X`, `Y`, `Print`). A `Rect` és a `Circle` származzon a `Shape`-ből: így örökli a `Shape` tagjait: olyan, mintha megírtuk volna a leszármazottakban külön-külön. Ennek megfelelően a leszármazottakból töröljük is a `Shape` osztályba átemelt kódot.

```
// Shape.cs
public class Shape
{
    private int x;
    private int y;

    public int X
    {
        get { return x; }
        set
        {
            if (value < 0)
                throw new ArgumentNullException("value");
            x = value;
        }
    }
```

```
    public int Y
    {
        get { return y; }
        set
        {
            if (value < 0)
                throw new ArgumentNullException("value");
            y = value;
        }
    }

    public void Print()
    {
        PrintName();
        PrintParams();
        Console.WriteLine(" Area: " + GetArea());
    }
}
```

```
// Circle.cs
public class Circle: Shape
{
    private int r;

    public int R
    {
        get { return r; }
        set
        {
            if (value < 0)
                throw new ArgumentNullException("value");
            r = value;
        }
    }

    public double GetArea() { return r * r * 3.14; }

    public void PrintName() { Console.Write("Circle"); }

    public void PrintParams()
    {
        Console.WriteLine(
             string.Format(" x: {0} y: {1}, r: {2}", x, y, r)
         );
    }

    public Circle(int x, int y, int r)
    {
        this.x = x;
        this.y = y;
        this.r = r;
    }
}
```

Mint látható, a leszármazott osztályban az osztálynév nevét követő „:” után adjuk meg az ősosztály nevét. .NET környezetben, így C# nyelven is egy osztálynak csak egy őse lehet (szemben a C++-szal), ugyanakkor több interfészt is implementálhat (erről később).

Ha beírjuk a fenti kódot, próbálkozásunk kudarcba fullad: számos fordítási hibát eredményez. Ahhoz, hogy a kódunk forduljon, el kell végezzük még a következő változtatásokat:

- **Probléma 1:** A `Shape` ősben a `Print` hívja a `PrintName` és `PrintParams` függvényeket. Azonban egy osztályban csak olyan függvény hívható, mely az adott osztályban létezik (vagy örökölte). A `Shape` esetében így fel kell vegyük ezeket a függvényeket, valamilyen alapértelmezett viselkedéssel.
- **Probléma 2:** A leszármazott `Circle` osztály `PrintParams` függvényében és konstruktorában használjuk az ős `x` és `y` tagváltozóit. Azonban ezek az ősben `private`-ok, így a leszármazottban sem érhetők el. A problémára két megoldás létezik:

  a, Az `x` és `y` tagváltozókat az ősben protected láthatóságúvá alakítjuk, így a leszármazottak is elérhetik.

  b, A leszármazottban nem az örökölt `x` és `y` tagváltozókat, hanem az örökölt `X` és `Y` propertyket használjuk (ezek publikusak). Ez elegánsabb megoldás, hiszen így a leszármazott is csak validált módon férhet koordinátákhoz. Így ezt a megoldást választjuk.

A javított megoldásunk a következő:

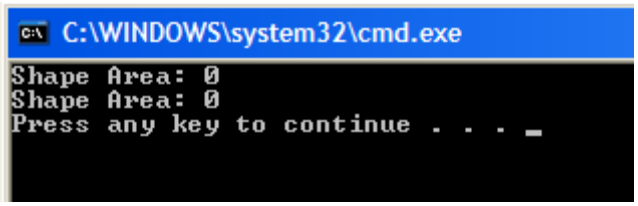
```
public class Shape
{
    ...

    // Probléma 1 megoldása
    public void PrintName() { Console.Write("Shape"); }
    public void PrintParams() { }
    public double GetArea() { return 0; }
}
```

```
public class Circle: Shape
{
    ...
    public double GetArea() { return r * r * 3.14; }

    public void PrintParams()
    {
        Console.WriteLine(
             string.Format(" x: {0} y: {1}, r: {2}", X, Y, r)
         );
    }

    public Circle(int x, int y, int r)
    {
        this.X = x;
        this.Y = y;
        this.R = r;
    }
}
```

A Rect osztályt is alakítsuk át gyakorlásképpen a `Circle` osztály mintájára. Így a kódunk már lefordul, használjuk is osztályainkat:

```
static void Main(string[] args)
{
    Rect r1; // (*1)
    r1 = new Rect(5, 5, 10, 20);
    r1.Print();

    Circle c1 = new Circle(10, 2, 3);
    c1.R = 10; // A set ág hívódik
    c1.Print();
}
```

A kimenetet látva azonban kezdeti lelkesedésünk gyorsan alábbhagy:



A kimenetben nem a `Rect` és a `Circle` paraméterei jelennek meg, a terület pedig nulla. Mi lehet a gond? Aki ismeri C++ kapcsán a virtuális függvények működését, hamar rájön a baj okára, ami a következő. Nézzük a `Circle` osztály `Print` függvényét:

```
public void Print()
{
    PrintName();
    PrintParams();
    Console.WriteLine(" Area: " + GetArea());
}
```

A problémát az okozza, hogy a `Print`-ben a `Shape` osztály `PrintName` és `PrintParams` függvénye hívódik meg, és nem a leszármazotté, amely már a megfelelő területet és paramétereket írná ki. A C++ nyelvben jártas fejlesztők számára jól ismert szabály ugyanis C# nyelven is él: **ha egy tagfüggvényt olyan pointeren (illetve C#-ban referencián) keresztül hívunk meg, mely ős típusú, akkor az ősosztálybeli függvény hívódik meg**. De hol van itt egyáltalán referencia? A `Print` törzsében írjuk ki explicit a „this„-t (ami implicit módon mindig ott van), és máris megkerült a keresett referenciánk: ez maga a `this`:

```
public void Print()
{
    this.PrintName();
    this.PrintParams();
    Console.WriteLine(" Area: " + GetArea());
}
```

Ha azt szeretnénk, hogy futás közben nézze meg a kód, hogy milyen típusú objektumra mutat a referencia (jelen esetben a this), akkor a meghívott függvényt virtuálisnak kell definiálni: ehhez az ősben a **`virtual`** kulcsszót, a leszármazottakban a felüldefiniálás helyén pedig az **`override`** kulcsszót kell C# nyelven használni:

```
public class Shape
{
    ...
    // Probléma 1 megoldása
    public virtual void PrintName() { Console.Write("Shape"); }
    public virtual void PrintParams() { }
    public virtual double GetArea() { return 0; }
}
```

```
public class Circle: Shape
{
    public override double GetArea() { return r * r * 3.14; }
    public override void PrintName() { Console.Write("Circle"); }
    public override void PrintParams()
    {
        Console.WriteLine( string.Format(" x: {0} y: {1}, r: {2}", X, Y, r)  );
    }
}
```

Megjegyzés Java fejlesztők számára: Java nyelvben minden (nem private és nem final) művelet automatikusan virtuális. Így azok számára, akik korábban csak Java nyelven fejlesztettek, elsőre szokatlan lehet, hogy .NET környezetben explicit virtuálissá kell tenni a függvényeket, ha felül akarjuk írni őket (különben, mint a példában látható, könnyen hibás viselkedést tapasztalhatunk. A Java nyelv ebben a tekintetben egyszerűbb, ugyanakkor a .NET megközelítésének is vannak előnyei:

- A virtuális függvények hívása kicsit lassabb, mint a nem virtuális függvényeké (ennek csak akkor van jelentősége, ha egy függvényt sokszor hívunk, és a törzse nagyon gyorsan lefut).
- A kód kifejezőbb: a virtual kulcsszót látva a kódban azonnal egyértelmű a kódban, hogy az adott függvény írójának az volt a szándéka, hogy az adott függvényt a leszármazottak felülírják/felülírhassák (míg a hiánya a fordítottját jelzi). C#-ban az alapértelmezett a nem virtuális „viselkedés”, a gyakorlatban pedig erre is van gyakrabban szükség.

Absztrakt osztályok

Jelen megoldásunkban bele lehet kötni a következőkbe (másba is, de haladjunk sorjában ☺):

- **A `Shape` osztály példányosítható.** Ez azt jelenti, hogy lefordul a következő:
  ```
  Shape s = new Shape();
  ```
  Ez nem szerencsés, hiszen a valóságban `Shape` objektumok létrehozásának nincs értelme, csak a leszármazottainak.
- **A `Shape.GetArea` implementált, nullával tér vissza.** Pedig a `Shape` egy absztrakt valami, melynek területe nem is értelmezett.

Szerencsére mindkét problémára van megoldás. Tegyük a `Shape` osztályt az **`abstract`** kulcsszóval absztraktá, így nem lesz példányosítható. Másrészt alakítsuk a `GetArea` függvényt is absztraktá szintén az `abstract` kulcsszóval, így nem kell neki törzset megadni. Az absztrakt függvények automatikusan virtuálisak is, ezeket a leszármazottban felül kell definiálni (vagy ha nem tesszük, maradnak absztraktak). Azok az osztályok, melyeknek van legalább egy absztrakt tagfüggvényük, kötelezően absztraktak kell legyenek. Az átalakított megoldásunk a következő:

```
public abstract class Shape
{
    ...
    public abstract double GetArea();
}
```

Gyakorlásképpen próbáljuk meg a `Shape` osztályt példányosítani:

```
Shape s = new Shape(); // Ez már nem fog menni szerencsére
```

Konstruktorok öröklésnél

Jelen esetben a leszármazott osztályaink felelősek a konstruktorukban az őstől örökölt tagok inicializálására is (`x` és `y`):

```
public class Circle: Shape
{
   ...
    public Circle(int x, int y, int r)
    {
        X = x;
        Y = y;
        R = r;
```

```
    }
}
```

Ez két okból nem szerencsés.

- Elfelejthetjük a leszármazott osztályban ezt megtenni, így az ős részei inicializálatlanok maradnak.
- Az őst inicializáló kódot minden leszármazottban duplikálni kell, amit szívből utálunk.
- Előfordulhat, hogy a leszármazott nem is éri el az ős inicializálandó tagjait , mert azok private-ok (és közvetve sem érhetők el).

A megoldás egyszerű: az ős inicializálását tegyük az ős felelősségévé, a leszármazottban csak használjuk ezt a kódot. Ennek megfelelően vezessünk be egy megfelelő konstruktort a `Shape` osztályban, a leszármazott konstruktorában pedig hívjuk meg a **`base`** kulcsszó használatával. Így a leszármazott konstruktor törzsébe már csak a saját tagjainak inicializálását kell megtennünk.

```
public abstract class Shape
{
    ...
    public Shape(int x, int y)
    {
        X = x;
        Y = y;
    }
}
```

```
public class Circle: Shape
{
    ...
    public Circle(int x, int y, int r): base(x, y)
    {
        this.R = r;
    }
}
```

Egy kiegészítő gondolat a konstruktorokkal kapcsolatban: az osztályok alapértelmezésben rendelkeznek alapértelmezett (paraméter nélküli) konstruktorral, de ha írunk egy másikat (nem defaultot), akkor ez már nem áll rendelkezésünkre. Semmi gond, ha szükségünk van rá, írjuk meg.

Ős tagfüggvény hívása a felüldefiniáló függvényből

Nézzük meg, hogyan történik a Circle osztályban az alakzatok paramétereinek kiírása:

```
public class Circle: Shape
{
    ...
    public override void PrintParams()
    {
        Console.WriteLine(
             string.Format(" x: {0} y: {1}, r: {2}", X, Y, r)
         );
    }
}
```

Nem túl szimpatikus megoldás, hogy a `Shape` ősosztályhoz tartozó paramétereket a leszármazott osztály írja ki:

- Az őst inicializáló kódot minden leszármazottban duplikálni kell. Ha például bevezetünk egy új tagot a `Shape`-ben, és már sok `Shape` leszármazottunk van, az új tag kiírásához valamennyi `Shape` leszármazottat módosítani kell. Kellemetlen.
- Előfordulhat, hogy a leszármazott nem is éri el az ős inicializálandó tagjait , mert azok `private`-ok (és közvetve sem érhetők el).

A megoldás: az ős paramétereinek kiírását tegyük az ős felelősségévé, a leszármazottban csak hívjuk meg ezt a kódot:

```
public class Shape
{
    ...
    public virtual void PrintParams()
    {
        Console.Write( string.Format(" x: {0}, y: {1}", x.ToString(), y.ToString()) );
    }
}

public class Circle: Shape
{
    ...
    public override void PrintParams()
    {
        // Előbb kiírjuk az ős paramétereit (x, y)
        base.PrintParams();
        // Majd a sajátainkat
        Console.WriteLine("r: " + r);
    }
}
```

Behelyettesíthetőség, egységes kezelés , polimorfizmus

A feladatkiírásunk része volt, hogy támogatni kell a különböző típusú alakzatok memóriában való tárolását, kényelmes listázását. Legyen ez egy új osztály, a `ShapeManager` feladata. Természetesen követelmény, hogy a jövőben kényelmesen tudjunk új alakzattípusokat (pl. háromszög) bevezetni, ehhez a meglévő osztályainkat ne kelljen átírni. Így nem választhatjuk azt a megoldást, hogy a `ShapeManager` egy `Rect` tömbben tárolja a téglalapokat, egy `Circle` tömbben pedig a köröket. Egy közös listában kell a különböző típusú alakzatokat tárolni. Szerencsére C#-ban is, mint minden objektumorientált környezetben él a **behelyettesíthetőségi szabály: őspointerrel (illetve C#-ban ősreferenciával) lehet leszármazottra is mutatni.**

Gyerekből szülőre (pl. `Rect`-re hivatkozás `Shape` referenciával) automatikus a konverzió, hiszen egy téglalap valójában alakzat is, rendelkezik annak minden tulajdonságával:

```
Rect r1 = new Rect(10, 20, 10, 20);
Shape s1 = r1; // Automatikus Rect -> Shape
Shape s2 = new Circle(1, 2, 3); // Automatikus Circle -> Shape
```

Szülőről gyerekre nincs automatikus konverzió (pl. `Shape`→`Rect`). Ez egy „veszélyes” művelet. Előfordulhat ugyanis, hogy a `Shape` hivatkozásunk valójában nem is egy `Rect` objektumra mutat, hanem egy másik leszármazottra (pl. `Circle`-re). Az ilyen irányú konverzióra is van lehetőség természetesen, de explicit ki kell írni. Ha a konverzió rossz (az objektum nem kompatibilis a céltípussal), futás közben `InvalidCastException` kivételt kapunk.

```
Shape s = new Rect(10,20,10, 10); // Automatikus Rect -> Shape
Rect r = (Rect)s; // Explicit cast, ok
Circle c = (Circle)s; // Explicit cast, InvalidCastException
```

Az „**is**” operátorral tudjuk ellenőrizni, hogy egy típus kompatibilis-e egy másikkal (bool-lal tér vissza) :

```
// is operátor
Shape s = new Rect(10, 20, 10, 10);
if (s is Rect)
{
    Rect r = (Rect)s; // Explicit cast, biztos OK, hiszen ellenőriztük
    // ...
}
```

Az „**as**” operátor működése hasonló, de ez megkísérli a konverziót. Ha nem lehetséges, `null`-lal tér vissza (de nem dob kivételt):

```
// as operátor
Shape s = new Rect(10, 20, 10, 10);
Rect r = s as Rect;
if (r != null)
{
    ...
}
```

Ennyi kitérő után írjuk meg a `ShapeManager` osztályunkat :

```
public class ShapeManager
{
    ArrayList shapes = new ArrayList();

    public void AddShape(Shape shape)
    {
        shapes.Add(shape); // Shape -> object aut konv.
    }

    public void PrintShapes()
    {
        object o;
        for (int i = 0; i < shapes.Count; ++i)
        {
            o = shapes[i];
            ((Shape)o).Print();
        }
    }
}
```

Használjuk is:

```
class Program
{
    static void Main(string[] args)
    {
        ShapeManager sm = new ShapeManager();
        Rect r1 = new Rect(10, 10, 20, 20);
        sm.AddShape(r1); // aut. Rect -> Shape konv
        sm.AddShape(new Circle(1, 2, 3));
        sm.AddShape(new Circle(3, 4, 6));
        sm.PrintShapes();
    }
}
```

Az osztály rövid magyarázata:

- Az elemek tárolására a `System.Collections` névtér `ArrayList` osztályát használtuk. Ez egy dinamikusan nyújtózkodó tömb (a közönséges .NET tömbök fix hosszúságúak!), `object` elemtípussal. Az `object` a `System.Object`-nek felel meg. .NET-ben minden ebből származik

implicit módon, így `object`-ként bármire lehet hivatkozni (olyasmi, mint C-ben a `void*`). Ennek következtében `ArrayList` típusú tárolóban bármilyen típusú objektum tárolható.

- Az `AddShape` művelet `Shape` típusként veszi át az újonnan tárolandó objektumot, így az automatikus konverzió miatt `Rect`, `Circle` és bármilyen más `Shape` leszármazott átadható.
- A `PrintShape` írja ki a tárolt elemek paramétereit. **Figyeljük meg, hogy minden elemet explicit `Shape`-re kell castolni. Ugyanis a `shapes[i]` típusa object, az object osztályban pedig nem létezik a `Print()` tagfüggvény!**

Lényeges gondolat, hogy a `ShapeManager` osztály tetszőleges `Shape` leszármazott osztállyal használható változtatás nélkül. Ezt a behelyettesítési szabály teszi lehetővé.

Generikus típusok használata

Az előző példánk `ShapeManager` osztályában az alakzatok tárolására az `ArrayList` osztályt használtuk. Az elemtípus ez esetben object, így az elemeket használatkor előbb `Shape`-re kell castolni. Ez körülményes, valamint nem típusbiztos (beletehetünk ugyanis véletlenül egy teljesen más típust is, pl. egy stringet is a többi alakzat mellé!). C++-ban a megoldást a sablonok (template-ek) alkalmazása jelenti. Hasonló (bár sok szempontból jelentősen eltérő!) megoldást a .NET is támogat, ezeket generikus típusoknak nevezzük (hasonlóan a Java nevezéktanhoz). Ezek szintaktikája hasonlít a C++-hoz (bár néhány tekintetben jelentősen egyszerűbb) és a Java-hoz. Egyelőre csak a .NET osztálykönyvtár által nyújtott generikus tárolóosztályok felhasználásával – ezek közül is a **`List<T>`** generikus listával ismerkedjünk meg. A típus a `System.Collections.Generic` névtérben definiált, és az `ArrayList`-hez hasonló dinamikusan nyújtózkodó tömböt valósít meg, de típusbiztosan. Alakítsuk át a `ShapeManager` osztályunkat, hogy az `ArrayList` helyett a `List<T>`-t használja:

```
using System.Collections.Generic;
...

public class ShapeManager
{
    List<Shape> shapes = new List<Shape>();

    public void AddShape(Shape shape)
    {
        shapes.Add(shape); // Shape -> object aut konv.
    }

    public void PrintShapes()
    {
        for (int i = 0; i < shapes.Count; ++i)
            shapes[i].Print();
    }
}
```

Lényeges egyszerűsítés, hogy a `PrintShape`-ben már nem kell az elemeket `Shape`-re konvertálni a `Print` hívása során, hiszen az elemeink típusa - a `shapes` tagváltozónk definiálásakor a `List`-nek megadott `Shape` generikus paraméternek megfelelően - `Shape`.

Kivételkezelés

A koncepció lényege ugyanaz, mint C++-ban. A **`throw`** kulcsszóval dobhatunk kivételt. A dobott kivétel hatására addig ugrunk felfelé a hívási fában, amíg egy megfelelő **`try-catch`** blokk el nem kapja. A kivételek .NET-ben az **`Exception`** osztályból kell származzanak. Maga a .NET Framework is számos kivételt definiál. Egy egyszerű példa:

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            double d;
            Console.WriteLine("Enter a nonzero number: ");
            d = Double.Parse(Console.ReadLine());
            if (d == 0)
                throw new Exception("The number can not be zero.");

            Console.WriteLine("The inverse is: " + 1 / d);
        }
        catch (FormatException e)
        {
            Console.WriteLine("Error! " + "Invalid number format.");
        }
        catch (Exception e)
        {
            Console.WriteLine("Error! " + e.ToString());
        }
        finally
        {
            // Cleanup - nothing to cleanup now
        }

        Console.WriteLine("Done.");
        Console.ReadKey();
    }
}
```

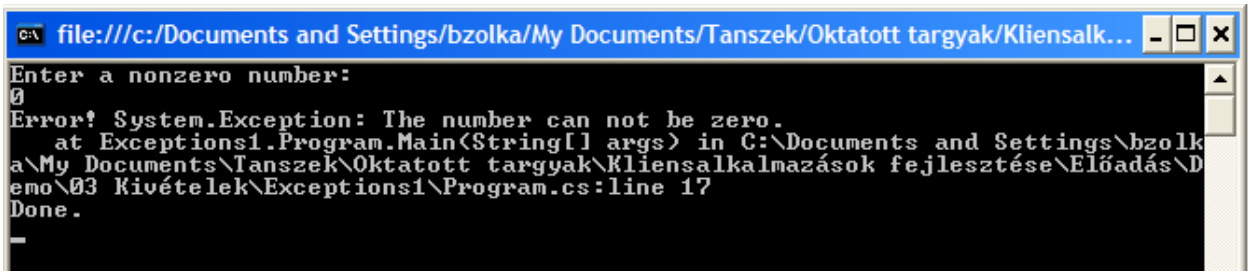
A példa rövid magyarázata:

- A `Console.ReadLine`-nal beolvasunk egy sort. Ezt a stringet `a Double.Parse`-szal megpróbáljuk `double`-re konvertálni, majd kiírjuk az inverzét.
- A műveleteket egy **`try-catch`** blokkba tettük. Ez esetünkben több `catch` ággal rendelkezik. Amennyiben a `Double.Parse` érvénytelen formátumú stringet kap paraméterként, `FormatException`-t dob (nézzük meg az online dokumentációját). Ezt a „`catch (FormatException e)`” kivételkezelő ágban elkapjuk, és jelezzük a felhasználó számára, hogy érvénytelen formátumú számot adott meg.
- Az inverz számítása során el kell kerüljük a nullával osztást (amely egyébként maga is kivételt eredményezne). Ennek érdekében megvizsgáljuk, hogy nulla-e a beolvasott szám, és ha igen, akkor kivételt dobunk:

  ```
  if (d == 0)
     throw new Exception("The number can not be zero.");
  ```

  A dobott kivételt a „`catch (Exception e)`” ág kapja el. Idézzük fel C++/Java-ból, hogy egy kivételkezelő ág azon kivételeket kapja el, amely paramétere kompatibilis a dobott kivétellel:
  - A „`catch (FormatException e)`” a `FormatException`-t és leszármazottait kapja el.
  - A „`catch (Exception e)`” az `Exception`-t és leszármazottait kapja el. Mivel minden kivétel az `Exception`-ből származik, ez gyakorlatilag minden kivételt elkap.

  A catch paraméterében megkapjuk a dobott kivételt: a „`catch (Exception e)`” esetében az „`e`” paraméter tartalmazza az általunk dobott kivételt, amikor nullával osztanánk. Ennek tartalmát `az e.ToString()`-gel tudjuk legegyszerűbben stringé alakítani majd megjeleníteni:



- Újdonság a C++-hoz képest, hogy megadható egy **`finally`** ág is. Az ebbe beleírt kód minden esetben lefut: ha volt kivétel, ha nem. Akkor is, ha a `try` blokkból `return`-nel lépünk ki. Így remekül használható olyan takarításra, amely minden esetben le kell fusson. Tipikus példa: a `try` blokkban nyitjuk meg az írandó/olvasandó fájlunkat/adatfolyamunkat (`FileStream` osztály), a `finally`-ban zárjuk le. Így garantált, hogy a fájlunk nem marad nyitva.

A kivételkezelésnek két nagy előnye van a hagyományos hibakezeléshez képest (melynek során pl. a hibát a függvények a visszatérési értékükben hibakóddal jelzik):

- Kevésbé körülményes, szépen különválik a normál, és a hibakezelésért felelős kód (nem kell minden függvényhívás után hibakódot vizsgálni).
- A hibák nem maradnak rejtve akkor sem, ha nem kezeljük. A hagyományos hibakezelés esetén ha nem vizsgáljuk meg a függvény visszatérési értékét, a hiba észrevétlen marad. Kivételkezelés esetében ha nem kezeljük a kivételt, az alkalmazás a hibainformációk kiírását követően kilép.

Interface

Az interfész egy olyan nyelvi elem a .NET nyelvekben, amely Java-ban igen, C++-ban viszont nem létezik. Az interfész nem más, mint egy művelethalmaz a műveletek szignatúrájának és a visszatérési érték típusának pontos megadásával. A műveletekhez implementációt (törzset) nem adhatunk. Szintaktikailag olyan, mint egy osztálydefiníció, de a `class` helyett az **`interface`** kulcsszót használjuk:

```
public interface IComparable
{
   // Összehasonlítja az objektumot a paraméterként kapott másikkal.
   int CompareTo( Object obj );
}

public interface IMySerializable
{
   void WriteToStream(Stream s);
   void LoadFromStream(Stream s);
}
```

A példánkban az `IComparable` interfészhez egy, az `IMySerializable`-hez pedig két műveletet tartozik. Az interfészre úgy is gondolhatunk, mint egy osztályra, melyben csupa absztrakt tagfüggvény található (de nem kell kiírni az `abstract` kulcsszót).

Az alábbiakban a `Person` osztály mindkét interfészt **implementálja**:

```
public class Person: IComparable, IMySerializable
{
  int CompareTo(Object obj) { ... }
  void WriteToStream(Stream s) { ... }
  void LoadFromStream(Stream s) { ... }
}
```

Egy osztály által implementált interfészeket az osztály neve utáni „:”-ot követően kell megadni, az ősosztály megadásához hasonlóan. Egy osztálynak csak egy őse lehet, de tetszőleges számú interfészt implementálhat.

Szabály1: Ha egy osztály implementál egy interfészt, akkor valamennyi műveletét implementálnia kell. Ha netán elfelejtenénk valamelyik függvény megírni (pl. a fenti `Person` osztályban a `LoadFromStream`-et), akkor fordítási hibát kapunk. Vagyis ha egy osztály implementál egy interfészt, akkor garantált, hogy valamennyi műveletét implementálja.

Szabály2: Egy osztályra (pontosabban objektumaira), hivatkozhatunk bármelyik általa implementált interfészként. Példa:

```
IComparable ic = new Person();
```

Ez pontosan olyan, mint amikor egy osztály objektumára lehet ősosztályként hivatkozni, vagyis működik a behelyettesíthetőségi szabály. Természetesen egy interfészen keresztül csak az interfészben is levő műveletek hívhatók az objektumra.

De mi értelme is van az interfészek használatának? Egyrészt **az ősosztályokhoz hasonlóan lehetővé teszik a különböző típusú objektumok egységes kezelését**. Ennek egy következménye: **lehetővé teszik széles körben használható osztályok és függvények megírását**. A következőkben írjunk meg egy ilyen sorrendező függvényt. Célunk tehát, hogy objektumok minél szélesebb körére használható legyen. Első gondolatunk talán az lenne, hogy akkor a rendezendő elemek listáját kapja meg `object` tömbként, hiszen minden típus `object` is. A sorrendezés során azonban a `Sort` törzsében össze kell tudjuk hasonlítani az elemeket. Az `object`-ek azonban nem összehasonlíthatók. Vagyis valójában „működik rájuk” az operator=, de ez a memóriabeli címük szerint hasonlítja össze az objektumokat, de ez nekünk nem lesz jó: számokat értékük szerint, stringeket ABC sorrendben, stb. szeretnénk sorrendezni. Az `object` típusnál egy kicsit többet kell feltételezni az elemekről. Az interfészek adják a megoldást: írjunk olyan `Sort` függvényt, ahol az elemek típusa `IComparable`. Miért is jó ez?

- Mivel az elemek implementálják az `IComparable` interfészt, a `CompareTo` meghívható rájuk (emlékezzünk, az implementáló osztályokban kötelező megírni).
- **A sorrendező osztályunk bármilyen, az `IComaparable` interfészt implementáló osztállyal használható lesz, így kellően általános**. Ha egy adott osztály elemeit sorrendezni szeretnénk, csak implementálnunk kell az `IComaparable` interfészt (meg kell írni a `CompareTo`-t), és máris sorrendezhető.

Írjuk meg az általános sorrendező függvényt:

```
public class Sorter
{
    public static void Sort(List<IComparable> items)
    {
        for (int i = 1; i < items.Count; i++)
        {
            for (int j = items.Count - 1; j >= i; j--)
            {

                if (items[j].CompareTo(items[j - 1]) < 0) // obj is smaller
                {
                    IComparable tmp = items[j];
                    items[j] = items[j - 1];
                    items[j - 1] = tmp;
                }
```

```
            }
        }
    }
}
```

Most nézzük meg, hogyan kell implementálni az `IComparable` interfészt a `Shape` osztályunkban ahhoz, hogy az alakzatokat terület szerint tudjuk sorrendezni:

```
public abstract class Shape: IComparable
{
    ...
    public int CompareTo(object other)
    {
        return this.GetArea().CompareTo(((Shape)other).GetArea());
    }
}
```

Megjegyezzük, hogy az általános sorrendező függvény megírására a példánkban nincs szükség, mert az általunk a `ShapeManaget` osztályban az alakzatok tárolására használt `List<Shape>` ezt már „beépítve” támogatja:

```
public class ShapeManager
{
    List<Shape> shapes = new List<Shape>();

    ...
    public void SortShapes()
    {
        shapes.Sort();
    }
}
```

```
static void Main(string[] args)
{
    ShapeManager sm = new ShapeManager();
    sm.AddShape(new Rect(10, 10, 20, 20));
    sm.AddShape(new Circle(1, 2, 3));
    sm.AddShape(new Circle(3, 4, 6));
    sm.SortShapes();
    // Terület szerint sorrendezve írja ki
    sm.PrintShapes();

    Console.ReadKey();
}
```

A sorrendezés mintájára írhatnánk egy általános adatfolyamba kiíró és beolvasó függvényt. Ez bármely osztállyal használgató lenne, amely implementálja a korábbi példánkban bemutatott `IMySerializable` interfészt. Ha egy osztály sorrendezni és adatfolyamba írni/betölteni is szeretnénk: semmi akadálya, csak implementálnia kell mindkét interfészt.

Az interfészek nagyon hasonlóak az ősosztályokhoz. Implementáció nem tehető bele (ennyiben „kevesebb” egy ősosztálynál), azonban egy osztály több interfészt is implementálhat, míg őse csak egy lehet (ennyivel „több”). A sorrendező függvényünket megcsinálhattuk volna úgy is, hogy az `IComparable` egy osztály, egy `CompareTo` absztrakt függvénnyel. Ez esetben viszont a `Shape`-nek már nem lehetett volna más őse. Jelen esetben ez nem okozott volna problémát, de sokszor már adott, mi kell legyen az ősosztályunk. Egy másik példa: a korábbi `Person` osztályunk implementálja az `IComparable` és `IMySerializable` interfészeket. Ha ezek osztályok lennének, ez nem lenne lehetséges (többszörös öröklést igényelne).

Két érdekességet említünk még meg az interfészek kapcsán (ezeket nem kell tudni). .NET-ben interfészben nemcsak művelet lehet, hanem `property` és `event` deklaráció is. Az event-ekről csak később tanulunk. A propertyk esetében a `get` és a `set` törzsét természetesen nem adhatjuk meg az interfészben (csak majd az implementáló osztályban):

```
interface ISupportName
{
    string Name { get; }
}
```

Az `ISupportName` interfészt implementáló osztályok kötelezően kell rendelkezzenek egy `Name` nevű tulajdonsággal, de csak egy getterrel, mivel a „set;”-et nem adtuk meg az interfészben, csak egy „get;”-et. A másik érdekesség pedig, hogy az interfészek is lehetnek generikusak, de erre itt nem mutatunk példát.

Modern nyelvi eszközök

.NET nyelvekben – így C#-ban is - megjelentek olyan nyelvi eszközök, melyek más objektumorientált platformokon több esetben nem találhatók meg, viszont a fejlesztés hatékonyságát megnövelik. Foglaljuk ezeket össze röviden:

- **Generikus típusok**: A C++-os sablonok megfelelői, az előző fejezetben láttunk példát a használatukra. A legtöbb objektumorientált nyelv (C++, Java, C#) támogatja.
- **Interfész:** Az előző fejezetben tárgyaltuk. A Java és a C# támogatja. C++-ban is „szimulálható” olyan absztrakt osztály definiálásával, amely csupa absztrakt (tisztán virtuális) tagfüggvényt tartalmaz.
- **Property (tulajdonság)**: Egyszerűsített szintaktikát biztosít a getter és setter tagfüggvények kiváltására. Tipikusan tagváltozókhoz való szabályozott hozzáférésre, illetve számított értékek visszaadására használjuk. .NET specifikus (Java, C++ nem támogatja).
- **Delegate (metódusreferencia)**: A C-beli függvénypointerek objektumorientált megfelelője. .NET specifikus (Java, C++ ?).
- **Event (esemény)**: A delegate-ekre épülve lehetővé teszi, hogy osztályok eseményeket definiáljanak, és erre objektumok feliratkozzanak. .NET specifikus (Java, C++ nem támogatja).
- **Attribute (attribútum)**: metaadatokat fűzhetünk vele nyelvi elemekhez (pl. osztályokhoz, tagfüggvényekhez, tagváltozókhoz, stb.)
- **Lambda expressions (lambda kifejezések):** Jelen jegyzetben nem térünk ki rá.

Property (tulajdonság)

Az előző nagyfejezetben egy példán keresztül ismertettük a tulajdonságok definiálásának módját és főbb jellemzőit. Itt ezt nem ismételjük meg, de egy újabb kódrészlettel illusztráljuk. Egy dolgot emelnénk csak ki: figyeljük meg, hogy az `Age` propertyt számított mező definiálására mutat példát.

```
class Person
{
    private string name;
    private int yearOfBirth;

    // Declare a Name property of type string:
    public string Name
    {
        get { return name; }
        set
        {
```

```
            if (value == null)
                    throw new ArgumentNullException("The Name property can not be null.");
            name = value;
        }
    }

    public int YearOfBirth
    {
        get
        {
            return yearOfBirth;
        }
        set
        {
            if (value < 1800 || value > 5000)
                    throw new ArgumentException(„Invalid yearOfBirth value.");
            yearOfBirth = value;
        }
    }

    // Számított, csak olvasható érték.
    public int Age
    {
        get { return DateTime.Now.Year-yearOfBirth; }
    }

    // Ez kell ahhoz, hogy konzisztens legyen!
    // A property-n keresztül állítjuk, hogy meglegyen a validáció.
    public Person(string myName, int yearOfBirth)
    {
        Name = name;
        YearOfBirth = yearOfBirth;
    }
 }

class Program
{
    static void Main(string[] args)
    {
        Person p1 = new Person("Béla", 1980);
        p1.YearOfBirth = 1995; // set-et hív
        Console.WriteLine("Név: {0}, kor: Age:{1}", p1.Name, p1.Age); // get-et hív

        p1.Age = 20; // Fordítási hiba, nincs set
        p1.YearOfBirth = 1000; // Futás közbeni hiba
        ...
    }
}
```

Auto-implemented property

.NET-ben gyakran fordul elő, hogy olyan property-t készítünk, mely lekérdezéskor pusztán visszaadja egy tagváltozó értékét, beállításkor egyszerűen minden validáció nélkül beállítja azt. Pl.:

```
class Person
{
    public string name;

    public string Name
    {
        get { return name; }
        set { name = value; }
```

```
    }
}
```

Jelen tudásunk alapján még nem érthető, mi értelme lehet ennek, hiszen az egészet kiválthatnánk egy normál publikus tagváltozó használatával. Bizonyos .NET „modulok” viszont kifejezetten építenek a property-k meglétére (pl. Form-ok estében az adatkötés), így mégiscsak nem is ritkán találkozunk hasonlóval. C# 3.0-tól kezdve lehetőség van auto-implementált property írására, amivel a fenti példa tömörebb formába hozható:

```
class Person
{
    public string Name
    {
        get; set;
    }
}
```

Ha a get és a set kulcsszavak után nem adunk meg implementációt, akkor auto-implementált property keletkezik. Az osztály a property-ez generál egy láthatatlan (kódból nem is elérhető) tagváltozót, és a property lekérdezésekor ennek értékét adja vissza, illetve ezt állítja. További lehetőségek:

- Lehetőség van csak get vagy csak set megadására.
- Arra is lehetőségünk van, hogy a get vagy a set ágra vonatkozóan szigorítsuk a láthatóságot. Pl. az alábbi osztályban a property-t a külvilág csak olvasni tudja, míg a saját metódusai írni is:

```
class Person
{
    public string Name
    {
        get;
        private set;
    }
}
```

Delegate (delegát, metódusreferencia)

Olyan, mint a C-ben a függvénypointer, csak objektumorientált, illetve a C-vel szemben nemcsak egy, hanem több függvényre (metódusra) is lehet vele mutatni (hivatkozni). A delegate-ek használatának egyik előnye az, hogy futási időben dönthetjük el, hogy több metódus közül éppen melyiket szeretnénk meghívni.

A delegate-ek (hasonlóan a C függvénypointerekhez) **típusosak**, egy delegate objektummal a típusának megfelelő szignatúrájú és visszatérési értékű metódusra lehet hivatkozni. Amikor delegate-ekkel dolgozunk, első lépésben egy delegate típust kell definiálni. Ez annak felel meg, mint amikor C-ben a typedef kulcsszóval egy függvénypointer típust definiáltunk. Ennek megfelelően egy **delegát típus** definiálásával egy olyan típust definiálunk, amelynek változóival rámutathatunk egy vagy több olyan metódusra, amely kompatibilis (paraméterlistája és visszatérési típusa) a delegát típusával. Pl.:

```
delegate bool FirstIsSmallerDelegate(object a, object b);
```

Itt a FirstIsSmallerDelegate egy delegate típus. Ebből pont úgy hozhatunk létre változókat (lokális, tagváltozók), vagy szerepeltethetjük függvényparaméterben, mintha egy közönséges osztály lenne, pl.:

```
FirstIsSmallerDelegate fis1;
```

Itt a fis1 delegate változó (objektum) értéke null. Értéket így adhatunk neki:

```
fis1 = new FirstIsSmallerDelegate(FirstIsSmaller_Complex);
```

A delegate típus „konstruktorának” a visszahívandó függvény nevét kell megadni. Itt feltettük, hogy a hívó kód osztályában létezik egy FirstIsSmaller_Complex nevű olyan függvény, amely kompatibilis a FirstIsSmallerDelegate delegate típussal (van két object paramétere és bool-lal tér vissza.)

Amikor értéket adunk egy delegate objektumnak, lehet az egyszerűsített szintaktikát is használni, amikor csak a függvény nevét adjuk meg:

```
fis1 = FirstIsSmaller_Complex;
```

A delegát meghívásával a delegate objektum által hivatkozott metódus automatikusan meghívódik:

```
bool res = fis1(obj1, obj2); // meghívódik a FirstIsSmaller_Complex
```

A delegátok használatának egyik előnye az, hogy futási időben dönthetjük el, hogy több metódus közül éppen melyiket szeretnénk meghívni.

Példa

Az alábbi példában a **Sorter** osztály **HyperSort** függvénye egy általános sorrendező művelet. Tetszőleges típusú elemeket tud sorrendezni. Ehhez paraméterként megkapja az elemlistát, valamint az elemeket összehasonlítani képes metódusreferenciát.

```
class Complex
{
    public double Re, Im;

    //  Konstruktor
    public Complex(double re, double im)
    {
        this.Re = re;
        this.Im = im;
    }
 }

delegate bool FirstIsSmallerDelegate(object a, object b);

class Sorter
{
    // A lista rendezése egy parametérkent kapott delegate segítségével.
    // tetszőleges típusra használni szeretnénk. A Complex forráskója
    // nincs meg, nem tudjuk megoldani, hogy implementálja az
    // IComparable-t. Megoldás: átadjuk az összehasonlító "függvényt" is.
    public static void HyperSort(ArrayList list,
        FirstIsSmallerDelegate firstIsSmaller)
    {
        for (...)
        {
           ...
                if ( firstIsSmaller(list[j], list[j - 1]) )
                  ...
            }
        }
    }

}

class Program
{
    static void Main(string[] args)
    {
        ArrayList list = new ArrayList();
        list.Add(new Complex(1, 2)); // Egy elem, biztosra megyünk :).
```

```
        // ...

        // Metódusreferencia statikus tagfüggvényre
        Sorter.HyperSort(list, new FirstIsSmallerDelegate(FirstIsSmaller_Complex));
        // Egyszerűsített szintaktikával is lehet, ekkor csak a függvény nevét írjuk ki
        Sorter.HyperSort(list, FirstIsSmaller_Complex);

        // Metódusreferencia objektum tagfüggvényre (kicsit erőltetett,
        // itt ez is lehetne statikus
        Comparers comps = new Comparers();
        Sorter.HyperSort(list, new FirstIsSmallerDelegate(comps.FirstIsSmaller_Complex));

        // A delegate egy típus, lehet lokális változó, tagváltozó is.
        FirstIsSmallerDelegate fis1 = new FirstIsSmallerDelegate(FirstIsSmaller_Complex);
        bool isFIS = fis1(new Complex(1, 1), new Complex(2, 2));

        // Minden delegate egy MultiCastDelegate leszármazott. Több metódusreferenciát
        // is tud tárolni. A += operátorral vehetők fel újak. Az alábbi példában kétszer
        // is meghívódik a FirstIsSmaller_Complex, nincs sok értelme. Majd az event-eknél
        // látjuk, miért jó ez.
        fis1 += FirstIsSmaller_Complex;
        fis1(new Complex(1, 1), new Complex(2, 2));
    }

    public static bool FirstIsSmaller_Complex(object a, object b)
    {
        Complex ca = (Complex)a;
        Complex cb = (Complex)b;
        return Math.Sqrt(...);
    }
}

class Comparers
{
    public bool FirstIsSmaller_Complex(object a, object b)
    {
        // mint a Program osztályban
        --||--
    }

    public bool FirstIsSmaller_Person(object a, object b)
    {
        ...
    }
}
```

Egy másik megközelítés az lehet, ha nem metódusreferenciát használunk, hanem az sorrendezendő objektumok típusának kell egy interfészt implementálniuk (pl. IComparable), amely két elem összehasonlítását elvégzi. Az olyan nyelvek esetében, melyek nem támogatják a függvénypointer/metódusreferencia koncepcióját, ezt szokás használni. .NET környezetben mindkét megoldás használható, az esettől függ, melyiket célszerűbb választani. Az interfész alapú megközelítés kicsit „egységbezártabb” megközelítést jelent, a delegate alapú kicsit rugalmasabb:

- Lehet statikus függvényre is metódusreferencia, nem kell hozzá objektum.
- Akkor is használható, ha nem tudjuk megoldani, hogy implementálja az adott interfészt, pl. „IComparable”-t.

A delegate-ekről leírás példákkal többek között itt található: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/using-delegates

Event (esemény)

Az alkalmazások többsége manapság már eseményvezérelt. Ez azt jelenti, hogy az alkalmazás bizonyos elemei eseményeket váltanak ki bizonyos sorrendben (például egy gomb megnyomása a felhasználói felületen), amelyekre az alkalmazás más részei reagálnak.

Az eseménykezelés .NET-ben, mint ahogy általában minden más programozói nyelvben (feltéve, hogy támogatja azt), **a Publisher/Subscriber tervezési mintára épül. Ennek lényege, hogy tetszőleges osztály publikálhatja eseményeknek egy csoportját, amelyekre tetszőleges más osztályok objektumai előfizethetnek. Amikor a publikáló osztály objektuma elsüti az eseményt, az összes feliratkozott objektum értesítést kap erről.** Megjegyzés: bár sokkal ritkább, lehetőség van osztály szinten is eseményeket definiálni (statikus esemény), illetve eseményekre osztály szinten feliratkozni.

.NET-ben az eseménykezelés hátterében a delegátok állnak. Az eseményt publikáló osztály tulajdonképpen egy multicast delegátból egy speciális tagváltozót definiál az **event** kulcsszóval, amely a feliratkozott osztályok egy-egy metódusára mutat. Az event egy közönséges nyelvi elem lett, osztályoknak most már nem csak tagváltozói, tagfüggvényei és tulajdonságai lehetnek, hanem eseményei is.

Amikor az esemény elsül, a feliratkozott osztályok megfelelő metódusai a delegáton keresztül meghívódnak. A feliratkozott osztályok azon metódusait, amelyekkel az egyes eseményekre reagálnak, eseménykezelő metódusoknak nevezzük.

Minden eseménynek van egy típusa és van egy neve. A típusa annak a delegátnak a típusa, amelyik tulajdonképpen megfelel magának az eseménynek.

Példa

Az alábbi példában a **Logger** egy általános célú naplózó osztály. Ez egy event-tet definiál **Log** néven. A Logger osztály esetében naplózni a WriteLine függvénnyel lehet, ami nem csinál mást, mint elsüti a Log eseményt (a null vizsgálattal előbb megvizsgálja, van-e legalább egy előfizető). Az **App** osztály az alkalmazást reprezentálja. A konstruktorában két előfizető műveletet is beregisztrál a **+= operátorral**: a saját **writeConsole** tagfüggvényét (amely a konzolra naplóz), valamint egy **FileLogListener** objektum **WriteToFile** tagfüggvényét (fájlba naplóz). Eseményről leíratkozni a **-= operátorral** lehet (lásd App.Cleanup művelet).

```
public delegate void LogHandler(string msg);

class Logger
{
    // Osztálynak .NET-nem nem csak tagváltozója és tagfüggvénye
    // lehet, hanem event-je is!
    public event LogHandler Log;
    // Ide jöhet a többi, de most nincs több

    public void WriteLine(string msg)
    {
        // Esemény elsütése (null vizsgálat: meg kell nézni, van-e előfizető)
        if (Log != null)
            Log(msg);
    }
}
```

```
/// <summary>
/// Fájlba naplózó osztály.
/// </summary>
class FileLogListener
{
    /// <summary>
    /// Szöveg fájlba írására a StreamWriter osztályt használjuk ("nyers" bájtok
    /// fájba írására a FileStream használatos).
    /// </summary>
    private StreamWriter streamWriter;

    /// <summary>
    /// A konstruktorban inicializáljuk a StreamWritert.
    /// </summary>
    /// <param name="filePath"></param>
    public FileLogListener(string filePath)
    {
        streamWriter = new StreamWriter(filePath, true);
    }

    /// <summary>
    /// Kiírja a fájl adatfolyamba a szöveget.
    /// </summary>
    /// <param name="msg">A naplózandó szöveg.</param>
    public void WriteToFile(string msg)
    {
        streamWriter.WriteLine(msg);
    }

    /// <summary>
    /// Lezárja a naplózó objektumot.
    /// </summary>
    public void Close()
    {
        streamWriter.Close();
    }
}


class App
{
    Logger log = new Logger();
    FileLogListener fileLogListener = new FileLogListener();

    public App()
    {
        // Feliratkozás a Log eseményre (a writeConsole és fileLogListener.WriteToFile
        // műveleteklet regisztráljuk be)
        log.Log += new LogHandler(writeConsole);
        log.Log += new LogHandler(fileLogListener.WriteToFile);

        // Egyszerűsített formával is feliratkozhatunk, csak a kezelőfüggvény nevét
        // adjuk meg. A fenti két sorral ekvivalens:
        log.Log += writeConsole;
        log.Log += fileLogListener.WriteToFile;

    }

    public void Process()
    {
        log.WriteLine("Process begin...");
        //...
        log.WriteLine("Process end...");
    }
```

```
    public void Cleanup()
    {
        // Leiratkozás eseményről
        log.Log -= new LogHandler(writeConsole);
        log.Log -= new LogHandler(fileLogListener.WriteToFile);
    }

    void writeConsole(string msg)
    {
        Console.WriteLine(msg);
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        App app = new App();
        app.Process();
        app.Cleanup();
    }
}
```

Miben más az event, mint a delegate?

- Egy delegate objektumból akkor lesz event, ha elé írjuk az event kulcsszót.
- Ez event osztályok tagváltozója lehet csak (így pl. lokális event objektum nincs, lokális delegate objektum létezik viszont.)
- Nem lehet az = operátort használni, csak a += és -= -t (így egy külső objektum nem tudja kitörölni a feliratkozottakat a listáról)
- Csak a tartalmazó osztály sütheti el.

Az események minden .NET nyelvben elérhetők, csak más szintaktikával.

Az event-ekről leírás példákkal többek között itt található: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/index

Attribute

Az attribútumok segítségével deklaratív jelleggel metaadatokat közölhetünk a kód bizonyos részeire vonatkozóan. Az attribútum is tulajdonképpen egy osztály, melyet hozzákötünk a program egy megadott eleméhez (típushoz, osztályhoz, interfészhez, metódushoz, …). Ezeket a metainformációkat a program futása közben mi magunk is kiolvashatjuk az úgynevezett reflection mechanizmus segítségével (ezzel részletesen egy későbbi gyakorlat foglalkozik), de általában az attribútumokkal csak a CLR számára szeretnénk információkat közölni. A .NET attribútumoknak a Java nyelvben az annotációk felelnek meg.

Az attribútumok funkciója a legkülönbözőbb féle lehet. A Serializable attribútum segítségével például egy osztályról jelezhetjük, hogy az binárisan sorosítható, azaz tetszőleges adatfolyamba (akár egy file-ba, akár egy hálózati adatfolyamba) az állapota bináris formában elmenthető:

```
[Serializable] // Jelezzük, hogy az osztály sorosítható
class User
{
```

```
    string name;

    [NonSerialized] // Jelezzük,hogy ezt a mezőt nem kell sorosítani
    string password;

    // Jelezzük a jogosultsági feltételeket
    [PrincipalPermission(SecurityAction.Demand, Role = "Admin")]
    public static void DeleteUser(int userId)
    {
    }

    // ...
}

class Program
{
    static void Main(string[] args)
    {
        User user = new User();
        // felparaméterezzük ...

        //  Sorosítás (serialization) egy file stream-be
        BinaryFormatter formatter = new BinaryFormatter();
        FileStream stream1 = new FileStream("Dump.dat", FileMode.Create);
        formatter.Serialize(stream1, user);
        stream1.Close();

        //  Deszerializalas a file stream-ből
        FileStream stream2 = new FileStream("Dump.dat", FileMode.Open);
        User u = (User)formatter.Deserialize(stream2);
        stream2.Close();

    }
}
```

Tudunk olyan kódot írni, amellyel lekérdezhetjük, hogy :

- a User osztálynak milyen attribútumai vannak
- a User osztálynak milyen tagváltozói/tagfüggvényei vannak, ezeknek milyen attribútumai (a BinaryFormatter is ezt csinálja, valamint a jogosultságellenőrző is).

További információ az attribútumokról: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/attributes/ (bár ebből a Custom Attributes résszel nem foglalkoztunk)