# Alapok

## Visual Studio és C#

### Előkészületek

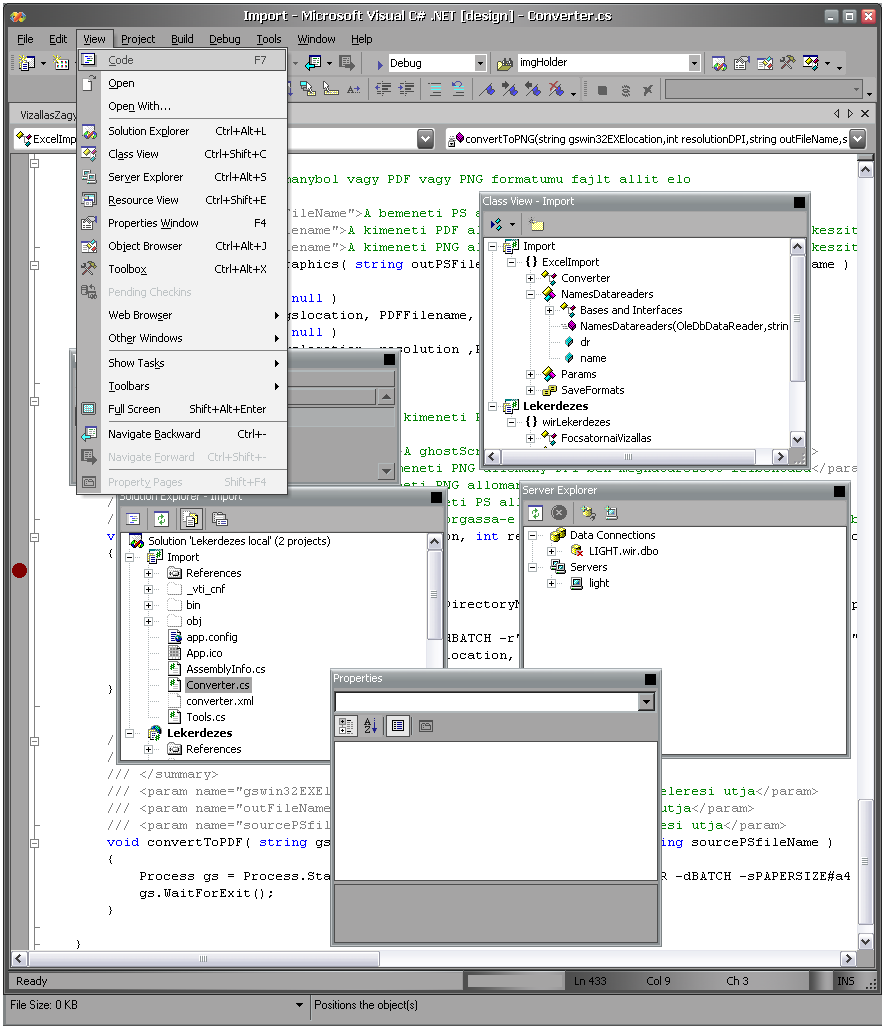
A labor elvégzéséhez Visual Studio 2015 szükséges.

### Visual Studio

A labor célja, hogy a .NET és a Visal Studio legalapvetőbb tulajdonságaival és funkcióival megismerkedhessenek a hallgatók.

* Labor elején mérjük fel hallgatóink felkészültségét!
  + Ki találkozott már a Visual Studioval?
  + Ki programozott már C#-ban?

Indítsuk el a Visual Studio-t. Tömören ismertessük a felület felépítését!



A labor során érdemes néha kitérni rá, hogy mi látható az egyes ablakokban, illetve mire lehet őket használni.

1. Hozzunk létre egy új Solution-t! (File-New-Project, azon belül Other Project Types/Visual Studio Solutions/Blank Solution)

A Solution hivatott arra, hogy összefogja a Project-eket. Egy Solution bárhány projectből állhat, egy project pedig bárhány fájlból. Fontos hangsúlyozni, hogy csak a fejlesztés megkönnyítésére lett létrehozva a project feletti egység, hiszen a tényleges outputokat (legyen az EXE, DLL stb.) általában a projectek alapján állítjuk elő, a solution „csak” összefogja ezeket.

* A Solution View (a beállítások függvényében) néha elrejti a Solution-t, amennyiben csak egy project van benne, így ha a hallgatói gépeken esetleg eltűnne a Solution megjegyezhetjük, hogy ez csak a látszat, és később, amikor újabb projectet adunk hozzá magától megjelenik.

1. Hozzunk létre az üres solution-ben egy üres C#-os projectet (File-***Add***-New Project / C# -Console Application). Az így létrehozott Project alapértelmezésben egy konzolos alkalmazás minimális kóddal.
   * Nézzük meg a project adatlapján, hol állítható be egy project típusa (konzolos alkalmazás/DLL/Windows), igény és idő szerint térjünk ki az egyéb beállításokra.
2. Nézzük át Program.cs felépítését, fussuk át a nyelvi elemeket benne:
   * névtér
   * osztály
   * konstruktor (vagyis a hiánya…, C#-ban az alapértelmezett konstruktort nem kötelező kiírni)
   * statikus Main függvény
3. Beszéljünk a C# elnevezési konvenciókról. Ez nem Java (!!!)

* ökölszabályként camelCasing-et alkalmazunk a nem publikus elemeknél, és PascalCasinget a publikus elmeknél, továbbá minden osztály és függvénynévnél.

1. Egészítsük ki a main függvényt

int a = 5;

int b = 7;

Console.WriteLine(a+b);

Console.ReadLine();

Használjuk ki a kódrészletet, hogy bemutassuk a következő funkciókat:

* + ***Intellisense***: A Console.WriteLine(…) beírása valójában sokkal kevesebb leütés a következő módon: conso.wr[lefele nyíl](…) (Tehát nem érdemes shift-et használni, az elválasztó karakterek automatikusan kiváltják a kiegészítést, néha pedig érdemes nyilakat is használni)
* Az intellisense a CTRL+. kombinációval hívható elő
  + ***CodeSnippet***: akinek az előző nem volt elég, a cw+TAB+TAB code snipettel még tovább gyorsíthat
* Figyeljük meg, hogy az intellisense külön jelöli a code snippeteket
  + ***Debugger, Watch ablak, Locals ablak, Call Stack ablak***
  + ***Edit And Continue***
  + ***ToString():*** minden objektum rendelkezik ilyen függvénnyel, ezért adhatunk meg bármilyen objektumot a WriteLine függvénynek.

1. Ha jól állunk idővel: Állítsuk be a project adatlapján, hogy alkalmazásunk bemeneti paraméterekkel induljon. Írassuk ki ezeket a paramétereket a konzolra for ciklussal (for code snippet), majd foreach-csel is (fore+TAB+TAB)
2. Adjunk hozzá egy új osztályt Person.cs néven a projecthez. C#-ban nem kötelez semmi arra, hogy minden osztályt külön fájlban definiáljunk, de tartani szoktuk magunkat ehhez a konvencióhoz. Az osztályban hozzunk létre két tulajdonságot „Name” és „Age” néven, illetve írjuk felül a ToString() függvényt, végül a Main függvényt írjuk át, hogy az új osztályt használja.

public class Person

{

private string name = „Névtelen”;

public string Name

{

get { return name; }

set { name = value; }

}

private int age = 17;

public int Age

{

get { return age; }

set { age = value; }

}

public override string ToString()

{

return name + ": " + age;

}

}

class Program

{

static void Main(string[] args)

{

Person p = new Person ();

p.Name = "Béla";

p.Age = 18;

Console.WriteLine(p.ToString());

}

}

1. Egészítsük ki a programunkat egy eseménykezelővel, mely jelzi az életkor megváltozását. (sárga kiemelés jelöli az új sorokat)

public class Person

{

public override string ToString()

{

return name + ": " + age;

}

private string name = "Névtelen";

public string Name

{

get { return name; }

set { name = value; }

}

private int age = 17;

public int Age

{

get { return age; }

set

{

if (AgeChanging != null) AgeChanging(age, value);

age = value;

}

}

public event AgeChangingEventHandler AgeChanging;

}

public delegate void AgeChangingEventHandler(int oldAge, int newAge);

class Program

{

static void Main(string[] args)

{

Person person = new Person();

person.AgeChanging += new AgeChangingEventHandler(person\_AgeChanging);

person.Name = "Béla";

person.Age = 18;

Console.WriteLine(person.ToString());

}

static void person\_AgeChanging(int oldAge, int newAge)

{

Console.WriteLine("Age changed from "+oldAge+" to "+ newAge);

}

}

1. A most elkészült alkalmazást alakítsuk át egy komponensé: A project tulajdonságai között állítsuk át, hogy nem konzolos alkalmazást szeretnénk, hanem egy Class Library-t, ami egy DLL-t fog eredményezni. A Main() függvényt kommentezzük ki.
2. Miután elkészült a komponensünk, hozzunk létre egy új konzolos alkalmazást a solution-ön belül. A most elkészült projecthez adjuk hozzá referenciaként a másik projectet (references ikonon jobb gombbal, majd Projects). Ilyen módon meg tudjuk hívni a komponensünk egy függvényét. Tegyünk is így: példányosítsuk a Person osztályunkat, gyakorlatilag a korábbi Main függvény másolásával.

* Tipp: a második project akár VB#-ban is készülhet

## Gyűjtemények, generikus típusok

Ebben a fejezetben a legsűrűbben használt dinamikus tömböket fogjuk áttekinteni (az Array létrehozása után nem átméretezhető!).

1. Tiszta lappal indulunk . Hozzunk létre egy új konzolos alkalmazást!
2. A legsűrűbben a típusos System.Collections.Generics.List-et használjuk fejlesztés közben. Pontosan úgy kell használni, mint egy hagyományos tömböt, annyi extrával, hogy újabb elemek hozzáadása egyszerűen az Add() hívással történik:

using System.Collections.Generics;

List<string> sList = new List<string>( );

sList.Add( "Hello" );

sList.Add( "World" );

string hello = sList[ 0 ];

Mivel számos esetben nem megfelelő nekünk az List az Array helyett, így egyetlen hívással típusos Array-é konvertálhatjuk:

string[] strArr = sList.ToArray();

A System.Collections.Generics.Dictionary segítségével a különösen sok elemet tartalmazó tömbökben való keresést egyszerűsíthetjük le. A Dictionary kulcs-elem párosokat használ a tároláshoz.

Dictionary<int,string> dict = new Dictionary<int,string>();

for( int i = 0; i < 10; i++ )

dict.Add( i, "Ertek: " + i.ToString() );

if( dict.ContainsKey( 3 ) )

Console.WriteLine( "Letezo ertek: " + dict[ 3 ] );

Mint látható a Dictionary feltöltése hasonló a List-nél látottakhoz. Keresés benne a tábla indexelésével történik, ahol az index értéke az általunk megadott kulcs.

1. IEnumerable interfész: a legtöbb dinamikus tömb implementálja ezt az interfészt, minek segítségével az összes benne található elemen végig lehet lépkedni. Tegyünk így, írassuk ki a konzolra a korábban használt Dictionary teljes tartalmát:

Console.WriteLine( "Kulcs - Ertek" );

foreach( KeyValuePair<int,string> v in dict )

Console.WriteLine( v.Key.ToString() + " " + v.Value );

* Használjuk ki az előző példákat, hogy áttekintsük a generikus típusok használatát is.
* Ha nagyon sok időnk maradt, játszadozzunk el a listák rendezési és szűrési lehetőségeivel.