C# alapok

Jánosi-Rancz Katalin Tünde

Sapientia EMTE
tsuto@ms.sapientia.ro

C# alapvető tulajdonságai

Letisztult, mint a Java; egyszerű, mint a VB6; hatékony és rugalmas, mint a C++

- Nincs szükség mutatókra (de lehetséges). Van delegált!
- Automatikus memóriakezelés a szemétgyűjtésen keresztül
- Osztály, az interfész, a struktúra, a felsorolás és a metódusreferenciák (C++)
- Operátor-túlterhelés, változó számú argumentum (C++)
- Az attribútumalapú programozás (VB6)
- Bővítő metódusok
- Típusos lekérdezések (LINQ) támogatása

C# alapvető tulajdonságai -2

- Névtelen típusok támogatása
- A lambda operátorok `x => x*x`
- Névtelen metódusok, metódusreferencia
- Új objektuminicializáló szintaxis

```
topLeft = new Point { X = 0, Y = 0};
```

- Egyetlen típus több kódfájlon keresztüli definiálásának képessége (partial)
- Generikus típusok és template mechanizmus (C++, Java)
- Elemek funkcionális nyelvekből (LISP, Haskell)

A C# fejlődése

Evolution of C# (dotnetcodetree)

	VS IDE Version	Key features						
C# 1.0	VS2002	Managed Code						
C# 2.0	VS2005	Partial Class	Generics	Anonymous Methods	Nullable Types	Covariance Contra-variance	Lamda Expression	
C# 3.0	VS2008	Anonymous Types	Extension Method	Lamda Expression	Implicit type (var)	LINQ	Expression Tree	
C# 4.0	VS2010	Named Arguments	Late Binding	Optional Parameters	More COM Support			
C# 5.0	VS2012	Caller information	Async programming					
C# 6.0	VS2015	String Interpolation	Expression Bodied methods	await in catch and finally block	Exception Filters	null conditional operator	auto property intializer	...
C# 7.0	VS2017	Tuples	Pattern Matching	Out Variables	Local functions	ref return and ref local	Throw Expression	...

C# 6.0: open source compiler, szintaxis egyszerűsítés

C# 7.0 - Tuple type - újdonság

```
//szintaxis:
(<datatype> [name1],<datatype> [name2]) methodName(parameters){
    return (<val1>,<val2>);}
</val2></val1></datatype></datatype>
```

```
//használat:
(int a,string b) = (1,"hello");
Console.WriteLine($"{a} {b}");
```

```
(string, string ) getfirstlastname(string name)
{
     string[] val = name.Split(' ');
     return (val[0],val[1]);
}

(string fname, string lname) = getfirstlastname("Kiss Pistike");
Console.WriteLine($"{fname} {lname}");
```

```
var result = getfirstlastname("Kiss Pistike");
var details = (Name:"Kiss Pistike", Age:30);
```

C# Alkalmazások fordítása

- Fejlesztőeszközök (IDE): Visual Studio, SharpDevelop, MonoDevelop
- akár a jegyzettömb és lefordíthatjuk a beépített csc fordítóval
- MS C# fordító: `csc.exe`
- Mono C# fordító: `mono-csc`
- Közös C# fordító kimeneti opciók:

Option	**Jelentése**
`/out:OUT`	kimeneti assembly neve
`/target:exe`	futtatható parancssori alkalmazás (default)
`/target:library`	egyfájlos *.dll assembly
`/target:winexe`	megakadályozza, hogy a konzolablak a háttérben jelenjen meg
`/lib:PATH1[,PATHn]`	specifies the location of referenced assemblies
`/main:CLASS`	specifies the class with the Main method (short: -m)
`/optimize[+|-]`	enables advanced compiler optimizations (short: -o)

- Automatikusan hivatkozik a `mscorlib.dll`-re

A C# alapvető építőelemei

Általános:

- minden adattag és metódus egy típusdefiníción belül kell szerepeljen
- C# -ban nem lehet globális függvényeket vagy globális adatelemeket létrehozni
- tartalmaznia kell egy osztályt, amely definiálja a `Main()` metódust
- A `Main()` metódust tartalmazó osztály = alkalmazásobjektum
- Variációk a `Main() metódusra`
  - `static void Main(string[] args)`
  - `static int Main(string[] args)`
  - `static void Main()`
  - `static int Main()`
- `Main` visszatérítési értéke: `%ERRORLEVEL%` változóba
- kódolási stílus: camel case, e.g. `Console.WriteLine`

Parancssori argumentumok feldolgozása

```
static int Main(string[] args)
{
  ...
  foreach(string arg in args)
    Console.WriteLine(”Arg: {0}”, arg);

  Console.ReadLine();
  return -1;
}
```

NB: a 0-dik argumentum az alkalmazás nevét

NB: `System.Environment.GetCommandLineArgs()`

A System.Console osztály

Néhány metódus:

- Write, WriteLine, Read, ReadLine

Numerikus adatok formázása:

```
Console.WriteLine(”{1}, {0}, {2}”, 10, 20, 30);
Console.WriteLine(”{0:c}”, 99999); // c pénznem
Console.WriteLine(”{0:f3}”, 99999); // f fixpontos
Console.WriteLine(”{0:e}”, 99999); // e exponenciális
Console.WriteLine(”{0:X}”, 99999); // x hexadecimális
```

```
20, 10, 30
£99,999.00
99999.000
9.999900e+004
1869F
```

több [és komplexebb] formázó példa

Szintaxis

- azontosítók: kis-nagybetű érzékeny, teljes szó, betűvel vagy aláhúzással kezdődik, Unicode megengedett
- 77 kulcsszó: `abstract`, `as`, `base`, `bool`, `break`, `byte`, `case`, `catch`, `char`, `checked`, `class`, `const`, `continue`, `decimal`, `default`, `delegate`, `do`, `double`, `else`, `enum`, `event`, `explicit`, `extern`, `false`, `finally`, `fixed`, `float`, `for`, `foreach`, `goto`, `if`, `implicit`, `in`, `int`, `interface`, `internal`, `is`, `lock`, `long`, `namespace`, `new`, `null`, `object`, `operator`, `out`, `override`, `params`, `private`, `protected`, `public`, `readonly`, `ref`, `return`, `sbyte`, `sealed`, `short`, `sizeof`, `stackalloc`, `static`, `string`, `struct`, `switch`, `this`, `throw`, `true`, `try`, `typeof`, `uint`, `ulong`, `unchecked`, `unsafe`, `ushort`, `using`, `virtual`, `volatile`, `void`, `while`
- példa: `@` használatra

```
class class {...}   // Illegal
class @class {...}  // Legal
```

Szintaxis -2

- 23 **contextual** azonosító , `@` nélkül használható változó névként is:
  - `add`, `ascending`, `by`, `descending`, `dynamic`, `equals`, `from`, `get`, `global`, `group`, `in`, `into`, `join`, `let`, `on`, `orderby`, `partial`, `remove`, `select`, `set`, `value`, `var`, `where`, `yield`

- literálok: hasonló mint Javaban, e.g `12 0.5f ’\n’ ”cloud”`
- elválasztók: hasonló mint Javaban, e.g. `; { }`
- operátorok: hasonló mint Javaban, e.g. `. () * =`

C# adattípusok

Típus	C# azonosító	CLS?	Suffix	Megj.
Boolean	bool	Yes		true | false
SByte	sbyte	No		előjeles 8 bit
Byte	byte	Yes		előjel nélküli 8 bit
Int16	short	Yes		
UInt16	ushort	No		
Int32	int	Yes		
UInt32	uint	No	U	
Int64	long	Yes	L	
UInt64	ulong	No	UL	
Char	char	Yes		egyetlen 16 bites Unicode karakter
Single	float	Yes	F	32 bites lebegőpontos
Double	double	Yes	D	64 bites lebegőpontos
Decimal	decimal	Yes	M	128 bites lebegőpontos
String	string	Yes		Unicode karaketerek halmaza
Object	object	Yes		az összes .NET típus ősosztálya

`float`(32 bit) vs `double`(64 bit) and `decimal`(128 bit)

Mikor melyiket használjuk?

- `decimal` - pénzügyi alkalmazásokban, nagyobb pontosság
- `float` - ahol elnézhető a kerekítési hiba, grafikus könyvtárakban
- `double` - minden más esetben
- `double`: approximately $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$ - 15, 16 a tízedes vessző után helyes
- `decimal`: approx. $\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$ - 28, 29 a tízedes vessző után helyes
- `float` és `double` támogatnak számos műveletet és speciális értéklekérdezést (NaN = not a number, ± 0 and $\pm\infty$:

  ```
  double.NaN, double.PositiveInfinity, -0.0
  ```
- konverziós műveletek: `Double.Parse(…)`
- `decimal`-al dolgozni kb. 10x lassubb mint `double`-al

Numerikus típusok operátorai

- hasonló mint C/C++/Java: `+, -, *, %, ++, --`
- túlcsordulás

```
int a = int.MinValue;
a--;
Console.WriteLine (a == int.MaxValue); // True // a=2147483647
```

- futásidejű túlcsordulás ellenőrzés (csak tesztelésre, teljesítményvesztéssel jár)

```
int a = 100000, b = 100000;
int c = checked(a * b); // futásideju kivételt kényszerít ki
    adatvesztés esetén
```

- `unchecked` túlcsordulás ellenőrzés leállítása, adatvesztés elfogadható

```
int x = int.MaxValue;
int x = int.MaxValue+1; // compile time error
int y = unchecked (x + 1); //No error
unchecked { int z = x + 1; }
```

Numerikus típusok operátorai -2

- bit operátorok, mint C/C++/Java: `~`, `&`, `|`, `<<`, `>>`
- NB: a 8 és 16-bites integrálok: C# implicit módon kibővít egy nagyobb adattípusra (szűkítő és bővítő konverziók)

```
short x = 1, y = 1;
short z = x+y;           // futásideju kivétel, mert a C# implicit
    módon kibovít minden short típust int típusra
short z = (short) (x+y); // ok
```

- egyenlőség és hasonlító operátorok, mint C/C++/Java:
  `==`, `!=`, `<`, `>`, `<=`, `>=`
- feltételes operátorok, mint Java:
  `&&`, `||`, `?:`

Típuskonverziók

A nyelv szigorúan típusos, tehát minden értéknek fordítási időben ismert a típusa, és nem enged meg értékvesztést

- az implicit (automatikus) típuskonverziók korlátozva vannak a nagyobb típushalmazba
  - byte $\longrightarrow$ short, ushort, int, ..., double
  - int $\longrightarrow$ long, float, decimal, double
  - float $\longrightarrow$ double
- nem lehet automatikus konverzióra támaszkodni olyan típusok között, ahol nem garantált, hogy nem történik értékvesztés, és ez fordítási időben kiderül
  - float $\longrightarrow$ int
  - kell explicit konverzió

Típuskonverziók 2

- az explicit típuskonverzió fordítási időben felügyelt, és kompatibilitást ellenőriz, pl.:

```
int x; double y = 2, string z;
x = (int)y; // engedélyezett
z = (string)y; // hiba, mert int és string nem kompatibilisek
```

- típuskonverzió a `Convert` osztály statikus metódusaival, több módon is

```
int x; double y = 2, string z;
x = Convert.ToInt32(y);
z = Convert.ToString(y); // z = y.ToString();
x = Int32.Parse(z); // x = Convert.toInt32(z);
```

A `System.Char` tagjai

- egyetlen karakternyi adat tárolására képes, pl ’a’
- a `char` 2 byte-os unicode karakter
- használhatunk speciális escape szekvenciákat is, mint C/C++/Java: `\n`, `\t`, `\\`, `\’`, etc
- megadhatjuk hexadecimálisan is: `char omegaSymbol=’\u03A9’;`
- példák:

```
char.IsDigit(’a’): False
char.IsLetter(’a’): True
char.IsWhiteSpace(”Hello There”, 5): True
char.IsWhiteSpace(”Hello There”, 6): False
char.IsPunctuation(’?’): True
```

A `System.String` típus

- alapvető string manipuláló metódusok:
  `Length(), Compare(), Contains(), Equals(), Format(), Insert(), PadLeft(), PadRight(), Remove(), Replace(), Split(), Trim(), ToUpper(), ToLower()`
- string összefűzés: + operátor

```
string s1 = ”Programming the ”;
string s2 = ”PsychoDrill (PTP)”;
string s3 = s1 + s2;
```

- vezérlőkarakterek, mint C-ben: \’, \”, \\, \n, \r, \t
- @ - ahogy le van írva betű szerint értelmezi a fordító:
  `@”C:\MyApp\bin\Debug”`
- string egyenlőség tesztelése: `s1.Equals(s2)`
- NB: a string megváltoztathatatlan, metódusai egy teljesen új formátumú string objektummal térnek vissza
- gyakori string módosítás? használj `System.Text.StringBuilder` - az objektum belső karakteradatait módosítja. Hatékony.

Értékek értelmezése string adatokból

Egy értéket annak sztringreprezentációjából olvassuk ki (pl. GUI listamező választását numerikus értékké konvertálni)

```
bool b = bool.Parse(”True”);
Console.WriteLine(”Value of b: {0}”, b);
double d = double.Parse(”99.884”);
Console.WriteLine(”Value of d: {0}”, d);
int i = int.Parse(”8”);
Console.WriteLine(”Value of i: {0}”, i);
char c = Char.Parse(”w”);
Console.WriteLine(”Value of c: {0}”, c);
Console.WriteLine();
```

Output:

```
Value of b: True
Value of d: 99.884
Value of i: 8
Value of c: w
```

A string.Format(...) metódus

```
DateTime dateRightNow = DateTime.Now;
String.Format(”{0:M/d/yyyy}”, dateRightNow); // 3/9/2008

decimal value = 123.456m;
string.Format(”{0:c}”, value);     // $123.46
```

```
public class StringSamples
{
  public void TryFormat()
  {
    string firstThing = ”first thing”;
    string secondThing = ”second thing”;

    string sampleTextVeryBAD = ”Something I have to write here” +
             firstThing + ” and here too ” + secondThing;

    string sampleTextGOOD = String.Format(”Something I have to write here {0}
          and here too {1}”, firstThing, secondThing);
  }
}
```

String interpoláció C# 6

```
int x = 4;
Console.Write($”Egy téglalapnak {x} oldala van”);
// Egy téglalapnak 4 oldala van
```

- bármilyen típus lehet a {} -ben

A DateTime értéktípus

```
DateTime myValue = DateTime.Now;

Console.WriteLine(myValue.ToString()); // 7/5/2016 2:13:30 AM
Console.WriteLine(myValue.ToShortDateString()); // 7/5/2016
Console.WriteLine(myValue.ToShortTimeString()); // 2:13 AM

Console.WriteLine(myValue.AddDays(3).ToShortDateString()); // 7/8/2016
Console.WriteLine(myValue.AddDays(-3).ToShortDateString()); // 7/2/2016

Console.WriteLine(myValue.Month.ToString()); // 7

// sajátos dátum létrehozása
DateTime myBirthDate = new DateTime(1999, 12, 5);
Console.WriteLine(myBirthDate);

TimeSpan myAge = DateTime.Now.Subtract(myBirthDate);
Console.WriteLine(myAge.TotalDays.ToString());
```

Tömbök kezelése C# -ban

- hansonló mint Java-ban:

```
int[] myInts = new int[3];              // --> 0 0 0
string[] myStrings = new string[3];     // --> null null null
```

- minden pozíció a megfelelő adattípus default értékére állítódik
- minden tömb a System.Array osztályból származik, ennek metódusait használhatjuk
- hasznos statikus metódusok: `Clear(), Reverse(), Sort()`
- hasznos metódusok: `CopyTo(), GetEnumerator()`
- hasznos propertik: `Length, Rank`
  - Rank = Dimenzió (1 dim. tömb 1), (2 dim. tömb 2)

C# tömbinicializáló szintaxis

- többféle lehetőség:

```
string[] stringArray = new string[] { ”one”, ”two”, ”three” };
bool[] boolArray = { false, false, true };
int[] intArray = new int[4] { 20, 22, 23, 0 };
```

- objektumtömb:

```
object[] array = {1, false, ”korte”, new DateTime(1969,3,24)};

foreach(object o in array) {
  Console.WriteLine(”type: {0}, value: {1}”, o.GetType(), o);
}
```

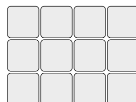
Output:

```
type: System.Int32, value: 1
type: System.Boolean, value: False
type: System.String, value: korte
type: System.DateTime, value: 24/03/1969 00:00:00
```

Munka a többdimenziós tömbökkel

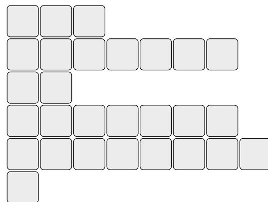
- derékszögű tömb: minden sor ugyanolyan hosszúságú

```
int[,] ra = new int[3,4];
for (int i = 0; i < 3; ++i)
  for(int j = 0; j < 4; ++j)
     ra[i,j] = i*j;
```



- 3D derékszögű tömb: `int [,,] cube = new int[3,3,3];`
- fűrészfogas (jagged) array: array of arrays; mindenik sornak különböző hossza lehet

```
int [][] ja = new int[6][];
ja[0] = new int[3];
ja[1] = new int[7];
...
ja[3][4] = 9;
...
```



- mennyi a `ja` rank-je ?

Tömbök mint paraméterek (és visszatérítési értékek)

- `static void PrintArray(int[] myInts)` ...
- `static string[] GetStringArray()` ...

Változók és paraméterek

- C# megköveteli a határozott hozzárendelést: inicializálatlan memóriához lehetetlen hozzáférni
  - kezdőérték megadása nem kötelező, de a lokális változó addig nem használható fel, amíg nem kap értéket (fordítási hiba)
  - függvény argumentumait meg kell adni mielőtt az meghívódna
  - minden más változó (pl. mezők és tömb értékek) automatikusan inicializálódnak a CLR által
  - a változónév első karaktere csak betű vagy (_) lehet, a többi karakter szám is. Konvenció: kisbetűvel kezdődnek

```
class Test {
  static int field;
  static void Main() {
    int localVar;
    Console.WriteLine(localVar);      // compile-time error
    ...
    int[] localArray = new int[2];
    Console.WriteLine(localArray[0]); // 0
    ...
    Console.WriteLine(field);         // 0
  }
}
```

Alapértelmezett értékek változóknak és mezőknek

- minden referencia típus: `null`
- minden numerikus és felsorolás típus: 0
- `char` típus: '\0'
- `bool` típus: `false`
- megkaphatjuk az alapértelmezett értékét **bármelyik** változónak a `default` kulcsszóval

```
decimal d = default (decimal);
```

Metódusok és paramétermódosítók

- szintaxis hasonló mint Java-ban
- az elemi típusok érték szerint, a referencia típusok cím szerint másolódnak
- paramétermódosítók:

Módosító	**Magyarázat**
(None)	átadása értékként történik
`out`	a kimeneti paramétereknek a meghívott metódusban értéket kell adni, a paraméter referenciaként adódik át. Ha nem kap értéket fordítási hiba
`ref`	a paraméter referenciaként adódik át; a hívó kezdetben hozzárendel egy értéket, amit a meghívott metódus mó-dosíthat
`params`	lehetővé teszi, hogy változó számú argumentumot egyetlen logikai paraméterként adjunk át. Egy metódusnak egyetlen params módosítója lehet, ez pedig az utolsó paramétere kell legyen

Az `out` módosító

- a paraméter referenciaként adódik át
- a kimeneti paramétereknek a meghívott metódusban értéket kell adni
- hasznos: lehetővé teszi, hogy egyetlen metódushívásból több visszatérítési érték is legyen
- egy metódusnak több `out` paramétere is lehet

```
void Add(int x, int y, out int ans){
  ans = x + y;
}
...
int ans;
Add(21, 21, out ans);
...
```

A `ref` módosító

- a paraméter referenciaként adódik át
- meg kell adnunk egy kezdőértéket, amit a meghívott metódus módosíthat

```
static void SwapStrings(ref string s1, ref string s2) {
  string tempStr = s1;
  s1 = s2;
  s2 = tempStr; // nincs fordítási hiba ha nem ad új értéket
}
...
string s1 = ”Flip”, s2 = ”Flop”;
SwapStrings(ref s1, ref s2);
Console.WriteLine(s1); // Flop
```

A `params` módosító

- lehetővé teszi, hogy változó számú argumentumot **egyetlen logikai paraméterként** adjunk át
- Egy metódusnak egyetlen params módosítója lehet, ez pedig az utolsó paramétere kell legyen

```
static double CalculateAverage(params double[] values) {
  double sum = 0;
  for (int i = 0; i < values.Length; i++)
    sum += values[i];
  return (sum / values.Length);
}
...
// Pass in a comma-delimited list of doubles...
double average;
average = CalculateAverage(4.0, 3.2, 5.7, 64.22, 87.2);
// ...or pass an array of doubles.
double[] data = { 4.0, 3.2, 5.7 };
average = CalculateAverage(data);
```

Opcionális paraméterek

- metódusok, konstruktorok és indexelők deklarálhatnak opcionális paramétereket
- egy paraméter opcionális ha megnevez egy **default értéket**

```
void Foo(int x = 23) {
  Console.Write(x);
}
...
Foo();  //23
```

- a default érték konstans kifejezés kell legyen, vagy paraméter nélküli konstruktora egy értéktípusnak
- opcionális paraméterek nem kaphatnak `ref` vagy `out` módosítót
- a kötelező paraméterek az opcionális paraméterek előtt kell legyenek

Nevesített argumentumok

- az eredeti metódus-deklarációban előírt sorrend helyett azonosíthatunk egy argumentumot a nevével

```
void Foo(int x, int y) {
  Console.WriteLine (x + ”, ” + y);
}
...
Foo(y:3, x:1); // sorrend nem fontos
Foo(1, y:2);   // nevesített és pozicionális argumentumok együtt
Foo(x:1, 2);   // error! pozicionális paraméterek a nevesítettek elott
              //kell legyenek
```

- hasznosak: kombinálva az opcionális paraméterekkel

```
void Foo(int x = 1, int y = 2, int z = 3) {
  Console.WriteLine(”x={0}, y={1}, z={2}”, x, y, z);
}
...
Foo(y:20);
```

Ciklusszerkezetek és ugró utasítások C# -ban

- hasonló Java/C++
- Ciklusszerkezetek
  - for loop
  - foreach/in loop
  - while loop
  - do/while loop

```
string[] carTypes = {”Ford”, ”BMW”, ”Yugo”, ”Honda” };
foreach (string c in carTypes)
  Console.WriteLine(c);
```

- warning: `foreach` bejáró ciklus **nem** használható itemek hozzáadásánál illetve törlésénél
- ugró utasítások: `break`, `continue`, `goto`, `return`, `throw`

Elágazó szerkezetek és a reláció/egyenlőség-operátorok

Nagyon hasonló mint Java-ban:

- `if/else` utasítás
- `switch` utasítás
  - minden eset, amelyik végrehajtható utasítást tartalmaz, lezáró break vagy goto-t kell tartalmazzon

```
string langChoice = Console.ReadLine();
switch (langChoice) {
  case ”C#”: Console.WriteLine(”C# is a fine language.”);
             break;
  case ”VB”: Console.WriteLine(”OOP, multithreading and more!”);
             break;
  default:   Console.WriteLine(”Well...good luck with that!”);
             break;
}
```

- reláció és egyenlőségvizsgáló operátorok `==, !=, <, >, <=, >=`

Névterek

- egy virtuális doboz, amelyben a logikailag összefüggő osztályok, metódusok vannak
- egy névtér típusai egyedi névvel kell rendelkezzenek
- hierarchikusan egymásba ágyazhatóak, ezt . jelöljük
- minden osztálynak névtérben kell elhelyezkednie

```
namespace Outer {
  namespace Middle {
    namespace Inner {
      class Class1 {}
      class Class2 {}
    }}}
    ...
using Outer.Middle;
... new Inner.Class1() ...
```

- Névteret magunk is definiálhatunk a `namespace` kulcsszóval

```
namespace MyNameSpace
{
}
```

Névterek -2

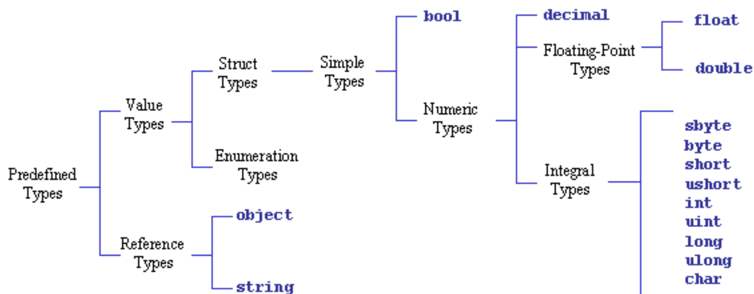
- A .NET könyvtárai is hierarchikus névterekben találhatóak
- Névtereket használni a `using <névtér>` utasítással lehet, ekkor a névtér összes típusa elérhető lesz
  - pl.: `using System;`
  - `using System.Collections.Generic;`
- az utasítás a teljes fájlra vonatkozik, így általában a névtér-használattal kezdjük a kódfájlt
- using nélkül: `System.Console.WriteLine(”Hello, World!”);`
- típusnév ütközés esetén mindenképpen ki kell írnunk a teljes elérési útvonalat
- névterek állneve

```
using PInf = System.Reflection.PropertyInfo;
class Program {
  PInf p;
}
```

Típuslétrehozás C# -ban

Típusosság

- Értéktípus (közvetlenül tartalmazzák értéküket, veremben jönnek létre)
  - összes numerikus, felsorolt, logikai, karakter típus, struktúrák
- Referenciatípus (tartalmuk referencián keresztül érhető el, halomban (heap) vannak lefoglalva)
  - osztályok, interfész, delegate típusok, stringek
  - olyan típusok, amelyek a System.Object–ből vagy bármely class kulcsszóval bevezetett szerkezetből származnak
- Mutatók



http://tldp.org/LDP/LGNET/85/ortiz.html

Példa:

```
static void Main(string[] args)
{
int x = 10; //lokális, verembe foglal helyet
}
```

```
class MyClass
{
private int x = 10; //x egy referenciatípuson belüli adattag, ezért a halomban foglal
     majd helyet
}
```

```
class MyClass
{
private int x = 10;  //halom
public void MyMethod()
{
int y = 10; //metódusban, de lokálisan, verem
```

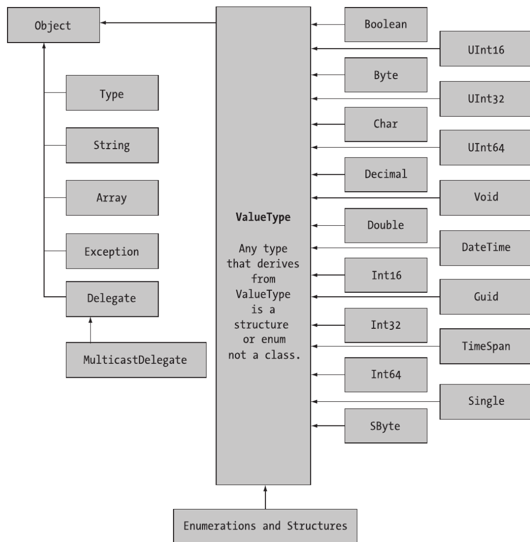
```
class MyClass
{
    public int x;
}
class Program
{
   public static void Main()
    {
        MyClass s = new MyClass();
        s.x = 10;

        MyClass p = s;
        p.x = 14;

        Console.WriteLine(s.x);
    }
    }
```

A rendszertípusok osztályhierarchiája

- minden típus a Sytem.Object nevű típusból származik



System.Object példák

- NB: int rövidítése System.Int32-nek
- minden típus a Sytem.Object kiterjesztése, így bármelyik nyilvános tagja meghívható

```
Console.WriteLine("12.GetHashCode() = {0}", 12.GetHashCode());
Console.WriteLine("12.Equals(23) = {0}", 12.Equals(23));
Console.WriteLine("12.ToString() = {0}", 12.ToString());
Console.WriteLine("12.GetType() = {0}", 12.GetType());
```

Output:

```
12.GetHashCode() = 12
12.Equals(23) = False
12.ToString() = 12
12.GetType() = System.Int32
```

Érték és referenciatípusok

Kérdés	Értéktipus	Referenciatípus
Hol történik a típus lefoglalása?	A veremben.	A felügyelt heapben.
Hogyan van jelképzve?	Lokális másolatokként.	A lefoglalt példány által elfoglalt memóriára mutatnak.
Mi az alaptípus?	A `System.ValueType`-ból kell származnia.	Bármelyik másik típusból származhat (kivéve a `System.ValueType`), amíg a típus nem "sealed".
Szolgálhat-e alaptípusként más típusnak?	Nem.	Igen, ha a típus nem sealed.
Mi az alapértelmezett paraméterátadási viselkedés?	Értékként adódik át (azaz egy másolat adódik át).	Referenciaként adódik át (azaz, a változó címe adódik át).
Felüldefiniáhatja-e a `System.Object.Finalize()` metódust?	Nem. Nem kerülhetnek a heapbe, ezért nem kell őket véglegesíteni.	Igen, közvetetten.
Lehet-e konstruktorokat definiálni a típus számára?	Igen.	Természetesen!
Mikor halnak meg?	Amikor kiesnek a meghatározó hatókörből.	Amikor az objektum szemétgyűjtött lesz.

Felsorolás típus - Enum

- értékek egymásutánja, ahol az értékek egész számoknak felelnek meg, hasonló mint C-ben

```
enum EmpType {
  Manager,       // = 0
  Grunt = 102,   //
  Contractor,    // = 103
  VicePresident // = 104
}...
EmpType emp = EmpType.Contractor; //hivatkozás a típusnéven át történik
```

- minden enum a `System.Enum` példánya: támogatja a `GetValues()` or `GetNames()` metódusokat
- Enum típust csakis metóduson kívül (osztályon belül vagy "önálló" típusként) deklarálhatunk, ellenkező esetben a program nem fordul le
- A tagok értékei alapértelmezetten int típusúak, de ezen változtathatunk (csakis a beépített egész numerikus típusokat használhatjuk mint byte, long, uint stb.)

Felsorolás típus - Példák

```
public enum BorderSide : byte { //figyelem a típus byte! Értékek 0-255
  Left=1, Right, Top=10, Bottom
}...
//egy enum példány explicit kasztolható az annak alapjául szolgáló  értékké
BorderSide bs1 = (BorderSide)11;//BorderSide.Bottom értékét hozzárendelni  OK
    Console.WriteLine(bs1); //Bottom

BorderSide bs2 = (BorderSide)100;  //ez NEM OK, de nem lesz exception
    Console.WriteLine(bs2); //100

byte b = (byte)BorderSide.Top;
    Console.WriteLine(b); //10

foreach (string s in Enum.GetNames(typeof(BorderSide)))
                Console.WriteLine(s);  //Left Right Top Bottom

foreach (byte b in Enum.GetValues(typeof(BorderSide)))
                Console.WriteLine(b);  //1 2 10 11
```

A struktúratípus

- a struktúra hasonló az osztályokhoz, viszont azoktól eltérően nem referencia-, hanem értéktípusok
- a struktúrák közvetlenül tartalmazzák a saját értékeiket, míg az osztályok „csak” referenciákat tárolnak.
- struktúra:
  - meghatározhat konstruktorokat, implementálhat interfészeket
  - tartalmazhat bármennyi adatmezőt, metódust, eseményt, és túlterhelt operátort
  - nem szolgálhat alap típusként
  - nem lehet virtual adatmezője
- hasznos amikor kulcs/értékpár szemantikára van szükségünk

```
struct Point {
  public int X, Y; //NB: rossz ötlet public mezoket használni
  public void Increment() { ++X; ++Y; }
  public void Decrement() { --X; --Y; }
  public void Display() {
    Console.WriteLine(”X: {0} Y: {1}”, X, Y);
  }}
```

Nullázható típusok C# -ban

- hogyan reprezentáljunk egy változót aminek nincs értéke?
- ez hasznos amikor AB-okkal foglalkozunk
- értéktípushoz nem lehet `null`-t rendelni

```
bool b = null;  //error CS0037: Cannot convert null to 'bool'
...             //because it is a value type
```

- a .NET 2.0-től lehet **nullable típust** létrehozni
- **csak** értéktípusoknál használható

```
bool? nullableBool = null;
int? nullableInt = 10;
Console.WriteLine(nullableInt.HasValue); // --> True
Console.WriteLine(nullableBool == null); // --> True
```

- a `?` egy rövidítése a `System.Nullable<T>` struktúra példányának létrehozására

A ? és ?? operátor

- ha az Exp1 értéke true, akkor az Exp2 értékelődik ki, ellenkező esetben az Exp3

```
Exp1 ? Exp2 : Exp3;
```

- rendelhetünk egy értéket egy nullable típusnak ha a kifejezés értéke null

```
int? myData = databaseRow.GetInt(3) ?? 100;
```

A ?. Null feltételes operátor - Elvis operátor

```
public class Person
{
public string FirstName { get; set; }
public int Age { get; set; }
}
```

```
//felt. hogy p Person típusú
var name = p?.FirstName; // ha p null, akkor name is null
//ha p nem null, akkor name a p.FirstName értéke lesz
var age = p?.Age;
// name string típusú, age int? típusú
```

```
//felt. hogy people  IList<Person> típusú
var thisName = people?[3]?.FirstName;
//ha a people null, thisName is null, ha people[3] null, thisName is null.
    Másképp thisName értéke a people[3].FirstName lesz. Ha people nem null,
    de kevesebb mint 4 eleme van, akkor people[3] OutOfRangeExceptiont ad.
```

```
VehicleTypeList?.Clear();
// ha a VehicleTypeList null, nem akad ki, h Cleart akarok a null-on
```

Osztálytípus - `class` Types

- nagyon hasonló a Java osztályaihoz
- adatmezőkből és olyan tagokból (konstruktor, tulajdonság, metódus, esemény) áll, amelyek műveleteket végeznek ezekkel az adatokkal
- referenciatípus

```
class Motorcycle
{
  public int driverIntensity;
  public string driverName;
  public Motorcycle() {}
  public Motorcycle(int intensity) : this(intensity, ””) {} // használj :this-t, hogy
  public Motorcycle(string name): this(0, name) {} // meghívj 1 általánosabb konstruktort
  public Motorcycle(int intensity, string name) {
    if (intensity > 10)
       intensity = 10;
    driverIntensity = intensity;
    driverName = name;
  }
}
```

- a `this` kulcsszóval férhetünk hozzá az aktuális osztály példányához
- objektum = egy adott osztálytípus `new` kulcsszóval létrehozott példányát; változó, amelynek típusa egy osztály

A konstruktor

- hasonló mint Java-ban
- konstruktor = egy új objektum kezdőértékkel való ellátója, (new)
- neve megegyezik az osztály nevével, nincs visszatérítési értéke
- ha nem írunk konstruktort, akkor a fordító generál egyet, de ha van egy non-default konstruktorunk, akkor a fordító nem generál
- egy osztálynak több konstruktora lehet, túlterhelhető

```
class Motorcycle {
  public int speed;
  public string brand;

  public Motorcycle() {}
  public Motorcycle(int speed) : this(speed, ””) {}

  public Motorcycle(string brand): this(0, brand) {}

  // fokonstruktor, ez végzi a lényeges munkát, a többi átpasszolja neki a feladatot
  public Motorcycle(int speed, string brand) {
    this.brand = brand;
    this.speed = Math.Min(300, speed);
  }
}
```

Objektum inicializálás

- lehetővé teszi, hogy megspóroljuk a konstruktor elkészítését, automatikusan le tudja generálni a konstruktort

```
class Point {
  public int X {get; set;}
  public int Y {get; set;}
  public int color;
  public Point() {}
  public Point(int x, int y) {
    X = x; Y = y;
  }
}
//paraméterek sorrendje nem fontos
Point p1 = new Point{color = 0, X = 1, Y = 2};
//keverhetjük a konstruktor hívást és az objektum inicializáló szintaxist
Point p2 = new Point(2, 3){color = 1};
```

- Gyűjtemény (collection) inicializálás

```
List<int> myGenericList = new List<int> { 0, 1, 2, 3, 4, 5};
List<Point> myListOfPoints = new List<Point> {
  new Point { X = 2, Y = 2 },
  new Point { X = 3, Y = 3 },
};
```

A beágyazott típus definiálása

```
public class OuterClass {
  ...
  public class PublicInnerClass {...}
  private class PrivateInnerClass {...}
  // NB: a nem beágyazott osztályok nem
       lehetnek private-ok
}
```

```
static void Main(string[] args)
{
// OK, mert public
OuterClass.PublicInnerClass inner;
inner= new OuterClass.PublicInnerClass();

// Fordítási hiba, mert private
OuterClass.PrivateInnerClass inner2;
inner2= new OuterClass.PrivateInnerClass();
}
```

- az inner class egy metódusa hozzáférhet az outer class private és protected tagjaihoz, ha van egy outer class példány referenciája

```
class OuterClass {
  private int privateMember;
  ...
  public class InnerClass {
    public void Foo(OuterClass oc) {
      ++oc.privateMember;
      Console.WriteLine("private member is now {0}", oc.privateMember);
    }
  }
}
...
new OuterClass.InnerClass().Foo(new OuterClass());
```

Mező módosítók

- Static módosító: `static`
- Hozzáférés módosítók: `public, internal, private, protected`
- Inheritance módosító: `new`
- Unsafe code módosító: `unsafe`
- Read-only módosító: `readonly`
- Threading módosító: `volatile`

A static kulcsszó

- osztályszintű mezők és metódusok definiálására használjuk
- statikus osztályok nem példányosíthatóak
- osztályszintről kell aktivizálni, nem pedig típuspéldányból:

```
Console.WriteLine(”Thanks...”);        //OK
Console c = new Console();
c.WriteLine(”I can’t be printed...”); //Error! WriteLine() is not
                                      //an instance level method!
```

- statikus metódust Osztálynév.Metódusnévvel érjük el
- statikus tagok csak statikus adattagot érnek el
- statikus mezők inicializálása a mezők deklarációs sorrendjében történik

```
class Foo {
  public static int X = Y;  // 0
  public static int Y = 3;  // 3
}
```

Statikus konstruktor - Static Constructors SC

- konstruktor = egy új objektum kezdőértékkel való ellátója
- **statikus konstruktor** helyet biztosít a statikus adatok kezdőértékeinek a beállításához (lehet AB-ból kell beolvasni őket)

```
class SavingsAccount {
  public double balance;
  public static double interestRate;
  public SavingsAccount(double balance) {
    this.balance = balance;
  }
  static SavingsAccount() {
    interestRate = 0.04;
  }
}
```

- tudnivalók:
  - egy osztály csak egy SC-t határozhat meg
  - egy SC nem vesz fel hozzáférési módosítót és nem vehet fel paramétereket
  - pontosan egyszer hajtódik végre
  - a SC végrehajtása megelőzi a példányszintű konstruktorét

Statikus osztályok

- csak static kulcsszóval jelölt tagokat vagy mezőket tartalmazhat

```
static class TimeUtilClass {
  public static void PrintTime() {
    Console.WriteLine(DateTime.Now.ToShortTimeString());
  }
  public static void PrintDate() {
    Console.WriteLine(DateTime.Today.ToShortDateString());
  }
}
```

- nem példányosítható, egyetlen példány létezik belőle

Az OOP alappillérei

Az OOP alappillérei

- **Egységbezárás - Encapsulation**
- Származtatás - Inheritance
- Polimorfizmus - Polymorphism

Egységbezárás. C# hozzáférés módosítók

- Hogyan rejti el egy objektum belső megvalósítási részleteit és védi meg az adatintegritást

C# hozzáférés módosító	Alkalmazási terület	Jelentés
`public`	Típusok vagy típustagok	Nincs hozzáférési korlátozás.
`private`	Típustagok vagy beágyazott típusok	Csak az az osztály férhet hozzá, amelyik létrehozta.
`protected`	Típustagok vagy beágyazott típusok	Az az osztály férhet hozzá, amelyik létrehozta és amelyek ebből származnak
`internal`	Típusok vagy típustagok	Csak ugyanabból az assemblyből.
`protected internal`	Típustagok vagy beágyazott típusok	Ugyanabból az assemblyből és a leszármazott osztályok.

- alapértelmezetten a típustagok (tulajdonságok, metódusok, konstruktorok, mezők) **implicit** `private`
- alapértelmezetten a típusok (osztály, interface, struktúra, felsorolás, metódusreferencia) **implicit** `internal`

Egységbezárás. Típustulajdonság - Type Properties

- az objektum belső adata ne legyen közvetlenül hozzáférhető egy objektumpéldányból
- .NET **tulajdonságok** használatával oldja meg az adatvédelmet
- a tulajdonságok C# -ban a Getter/Setter metódusokat váltják ki, úgy kezelhetőek, mint változók, azonban metódusok

```
class Book {
  int numberOfPages;

  public int NumberOfPages {
    get {
      return numberOfPages; //lekérdezo
    }
    set {
     if (value > 0 && value < 10000)
       numberOfPages = value; //Beállító, az átvett érték value néven érheto el
    }
  }
}
...
Book book = new Book();
book.NumberOfPages = 193;
```

- a hívó számára úgy tűnik mintha a nyilvános adatelemeket olvasná/ írná, háttérben pedig a get, set blokkok futnak le

Encapsulation. Type Properties -2

- `XXX` property-i `set_XXX` és `get_XXX` metódusokra lesznek leképezve (CIL)
- ezért ne használj `set_XXX` / `get_XXX` -t propertykkel egyszerre
- megadható a láthatósági szintje a `set` és `get` -nek

```
// az objektum felhasználók csak hozzáférhetnek,
// a létrehozó osztály és a származtatott típusok módosíthatják az
     értéket
public string SocialSecurityNumber {
  get { return empSSN; }
  protected set { empSSN = value; }
}
```

Egységbezárás. Csak olvasható és csak írható propertie-k

- Csak olvasható (csak írható propertiek): egyszerűen elhagyjuk a set (get) blokkot

```
public string SocialSecurityNumber {
  get { return empSSN; }
}
```

- a propertie-k statikusak is lehetnek
- csak osztályszinten lehet hozzáférni, az osztály egy példányából nem

```
class SavingsAccount {
  public double balance;
  public static double interestRate = 0.04;
  public static InterestRate {
    get { return interestRate; }
    set { interestRate = value;}
  }
...
```

Egységbezárás. Automatikus tulajdonságok

- ha egyszerűen csak szeretnénk beállítani egy mező get/set-jét további logika használata nélkül, akkor használjunk **automatikus** propertie-t
- a fordító biztosítja az implementációt
- példa: standard és **automatikus** property-re

```
class Car {
  private string name = string.Empty;
  public string Name {                  //
    get { return name; }                // standard property syntax
    set { name = value; }               //
  }
  public string PetName {get; set; }    // automatic property syntax
}
```

Automatikus tulajdonságok -2

- 1: az automatikusan generált propertie-k private háttérmezője **nem** látható a saját C# kódban, fordításidőben jöt létre
- 2: **nem** lehetséges read-only vagy write-only automatikus tulajdonságot készíteni (majd C# 6-tól)

```
public int MyReadOnlyProp { get; } // Read-only auto property? Error!
```

- **van** lehetőség arra, hogy megszorítsuk az automatikus tulajdonságok hozzáférését

```
public string PetName { get; protected set; }
```

Konstans adatok

- soha nem változnak meg a kezdeti értékadás után
- hasznos ha ismert értékek készletét adjuk meg
- a konstans adattag fordítás időpontjában definiálódik és menetközben nem lehet megmódosítni
- a `const` kulcsszóval definiálhatunk konstans adatot
- szabályok:
  - a konstansokat létrehozáskor kezdőértékkel kell ellátni
  - a konstans mezők **implicit** statikusak

```
class MyMathClass {
  public const double PI = 3.14;
}
...
Console.WriteLine(MyMathClass.PI);
```

Írásvédett mezők - Read-Only Fields

- a konstanshoz hasonló, de a hozzárendelt érték meghatározható futás közben (pl. egy konstruktor hatókörén belül)
- kezdeti értékadás után nem lehet módosítani

```
class MyMathClass {
  public readonly double MyConstant;
  public MyMathClass(double constant) {
    MyConstant = constant;
}
```

Részleges típusok - Partial Types

- partial típusok: szintaxis, amely megengedi, hogy különböző fileokba fejlessszük ugyanazt az osztályt vagy metódust
- hasznos ha nagy az osztály vagy egy része **machine generated**
- minden részhez oda kell tenni a `partial`-t
- ugyanolyan hozzáférhetőség
- delegate és enum nem lehet `partial`
- egy parciális osztály darabjainak ugyanabban az assemblyben kell lenniük

Employee.cs:

```
partial class Employee {
// Constructors...
// Properties...
}
```

Employee.Internal.cs:

```
partial class Employee {
  // Field data.
  private string empName;
  private int empID;
  private float currPay;
  private int empAge;
  private string empSSN;
  private static string companyName;
  ...
```

Partial Metódusok

- egyik helyen signatura, másik helyen az implementáció
- megszorítások:
  - csak partial osztályban definiálhatók
  - csak `void` típusú lehet
  - paraméterezhető (`this`, `ref`, vagy `params`—de NEM out módosító)
  - implicit `private`