Pázmány Péter Katolikus Egyetem
Információs Technológiai Kar

# Programozás .NET környezetben

## 2. gyakorlat

## C# alapismeretek

© 2013.02.21. Cserép Máté
mcserep@caesar.elte.hu
http://mcserep.web.elte.hu

### C# alapismeretek
**A nyelv lehetőségei**

- A C# *tisztán objektumorientált programozási nyelv*, amely teljes mértékben a *.NET Framework*re támaszkodik
  - hordozható kód, memóriafelügyelet
  - szintaktikailag nagyrészt C++, megvalósításában Java
  - egyszerűsített szerkezetet biztosít, nem választható el a deklaráció a definíciótól
  - strukturált felépülés névterekkel
  - lehetőséget ad komponens-alapú, elosztott, lokalizált, beágyazott fejlesztésre
  - 2.0-tól sablonok használata, elosztott típusok
  - 3.0-tól támogatja a funkcionális paradigmát
  - a forrásfájl kiterjesztése: **`.cs`**



### C# alapismeretek
**Azonosítók és literálok**

- Kódolás: Unicode 3.0
  - azonosítókat a szabványnak megfelelően lehet írni
  - a kulcsszavak a @ karakterrel használhatóak azonosítóként
- Számábrázolás:
  - egész számok alapértelmezésben 10-es számrendszerben ábrázolódnak, de lehet 8-as (**`0`** előtag, pl. **`0173`**), illetve 16-os számrendszert használni (**`0x`** előtag, pl. **`0x3de2`**)
  - hosszú (**`long`**) számokat **`L`** utótaggal (pl. **`132L`**), előjel nélküli (**`unsigned`**) számokat **`U`** utótaggal jelölünk (pl. **`12U`**)
  - valós számok alapértelmezetten dupla pontosságúak, egyszeres pontosságot **`F`** utótaggal jelölünk (pl. **`13.1F`**)
  - valós számok megadhatóak kitevős formában is (pl. **`3.1E4`**)



### C# alapismeretek
**A „Hello, World!” program**

```
namespace Hello // névtér
{
  class HelloWorld // osztály
  {
      static void Main() // statikus főprogram
      {
          System.Console.WriteLine("Hello, World!");
          // kiírás konzol képernyőre
      }
  }
}
```



### C# alapismeretek
**Névterek**

- A névterek biztosítják a rendszer strukturáltságát, lényegében csomagoknak felelnek meg
  - minden osztálynak névtérben kell elhelyezkednie, nincs globális, névtelen névtér, így a program szerkezete:
    **`namespace <nevek> { <osztályok> }`**
  - hierarchikusan egymásba ágyazhatóak, és ezt a névtérben pont elválasztóval jelöljük, pl.:

```
namespace Outer { ... }
namespace Outer.FirstInner { ... }
// a fenti névtéren belüli névtér
namespace Outer.FirstInner.DeepInner { ... }
// a belső névtéren belüli névtér
namespace Outer.SecondInner { ... }
```



### C# alapismeretek
**Névterek**

  - a .NET könyvtárai is hierarchikus névterekben találhatóak közvetlenül csak az aktuális névtérbeli osztályok láthatóak, de más névterekre is hivatkozhatunk
- 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 névtérhasználattal kezdjük a kódfájlt
  - a típusnév előtt is megadhatjuk a használandó névteret (így nem kell **`using`**), pl.: **`System.Collections.Stack l;`**
  - típusnév ütközés esetén mindenképpen ki kell írnunk a teljes elérési útvonalat

## C# alapismeretek
### Típusosság

- A nyelv három típuskategóriát különböztet meg:
  - *érték*: érték szerint kezelendő típusok, mindig másolódnak a memóriában, és a blokk végén törlődnek
  - *referencia*: biztonságos mutatókon keresztül kezelt típusok, amelyeknél csak a memóriacím másolódik, a szemétgyűjtő felügyeli és törli őket, amint elvesztik az összes referenciát
  - *mutató*: nem biztonságos mutatatók, amelyek csak felügyeletmentes (unsafe) kódrészben használhatóak
- Minden típus objektumorientáltan van megvalósítva
- Minden típus a teljes származtatási hierarchiának megfelelően egy .NET Framework-beli osztály, vagy annak leszármazottja, a keretrendszertől független típusok nem hozhatóak létre



## C# alapismeretek
### Primitív típusok

- A nyelv *primitív típusa*i két névvel rendelkeznek, egyik a C# programozási nyelvi név (amelynek célja a C++-beli elnevezések megtartása), a másik a .NET könyvtárbeli megfelelő típusnév
  - a .NET típusnév használatához szükségünk van a System névtérre
- Érték szerinti primitív típusok:
  - logikai: **`bool`** (**`System.Boolean`**)
  - egész számok (előjeles és előjel nélküli):
    - 8 bites: **`sbyte`** (**`System.SByte`**), **`byte`** (**`System.Byte`**),
    - 16 bites: **`short`** (**`System.Int16`**), **`ushort`** (**`System.UInt16`**)



## C# alapismeretek
### Primitív típusok

    - 32 bites: **`int`** (**`System.Int32`**), **`uint`** (**`System.UInt32`**)
    - 64 bites: **`long`** (**`System.Int64`**), **`ulong`** (**`System.UInt64`**)
  - lebegőpontos számok:
    - 32 bites: **`float`** (**`System.Single`**)
    - 64 bites: **`double`** (**`System.Double`**)
  - fixpontos szám: **`decimal`** (**`System.Decimal`**)
  - karakter: **`char`** (**`System.Char`**)
- Referencia szerinti primitív típusok:
  - objektum (ősosztály): **`object`** (**`System.Object`**)
  - szöveg: **`string`** (**`System.String`**)



## C# alapismeretek
### Primitív típusok

- Már a primitív típusok is intelligensek C#-ban, azaz támogatnak számos műveletet és speciális értéklekérdezést a típusokon, illetve változókon keresztül, pl.:
  - szöveggé alakítás bármely típusra: **`intValue.ToString()`**
  - speciális értékek lekérdezése: **`Int32.MaxValue`**, **`Double.NaN`**, **`Double.PositiveInfinity`**, **`String.Empty`**
  - konverziós műveletek: **`Double.Parse(myString)`**
  - karakter átalakító és lekérdező műveletek: **`Char.ToLower(myChar)`**, **`Char.IsDigit(myChar)`**
  - szöveg átalakító és lekérdező műveletek: **`myString.Length`**, **`myString.Find(char)`**, **`myString.Replace(oneChar, ahotnerChar)`**



## C# alapismeretek
### Példányosítás

- Változókat bármely (nem névtér) blokkon belül létrehozhatunk a programkódban a típus, a név, illetve a kezdőérték megadásával
  - pl.: **`Int32 myInt = 10;`**
  - a kezdőérték megadása nem kötelező, de a változó addig nem használható fel, amíg nem kap értéket (ezt a fordító ellenőrzi)
  - összetett típusok esetén használni kell a **`new`** utasítást a létrehozáshoz
- Konstansoknál használnunk kell a **`const`** kulcsszót, ekkor kötelező a kezdőérték megadása
  - pl.: **`const Int32 myConstInt = 10;`**



## C# alapismeretek
### Konstansok példányosítása

- A nem elnevezett konstansok a megfelelő automatikus típust kapják meg, ez módosítható karakterliterálok (**`L`**, **`U`**, **`F`**, …) segítségével, pl.:

```
10 // típusa Int32 lesz
10L // típusa Int64 lesz
10.0 // típusa Double lesz
10.5 // típusa Double lesz
10.5F // típusa Single lesz
```

- Ezen konstansok is megkapják a típus összes utasítását, így ők is intelligensek lesznek, pl.:

```
10.ToString() // eredménye: "10"
"Hello World".SubString(0, 5)
   // eredménye: "Hello"
```

## C# alapismeretek
### Típuskonverziók

- *Szigorúan típusos* a nyelv
  - minden értéknek fordítási időben ismert a típusa
  - az implicit (automatikus) típuskonverziók korlátozva vannak a nagyobb típushalmazba
    - pl.: **`byte`** ⟶ **`short`**, **`ushort`**, **`int`**, …, **`double`**
      **`int`** ⟶ **`long`**, **`float`**, **`decimal`**, **`double`**
      **`float`** ⟶ **`double`**
  - nem lehet automatikus konverzióra támaszkodni olyan típusok között, ahol nem garantált, hogy nem történik értékvesztés, és ez fordítási időben kiderül
    - pl.: **`float`** ⟶ **`int`**
    - ekkor explicit konverziót kell alkalmaznunk



## C# alapismeretek
### Típuskonverziók

- 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 double és string nem
kompatibilisek
```

- tetszőleges primitív típuskonverzióra a **`Convert`** osztály statikus metódusai használhatóak, illetve szövegre történő konverzió több módon is elvégezhető:

```
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);
```



## C# alapismeretek
### Operátorok

- Az operátorok olyan függvények, amelyek speciális meghívással rendelkeznek, vezérlőkarakteren, vagy kulcsszavakon keresztül
  - a meghívás rögzített formában történik (*prefix*, *postfix* vagy *infix* jelölés mellett)
  - az operátorok *precedenciá*val rendelkeznek, amely halmozás esetén megszabja a hívási sorrendet
  - az operandusok száma minden esetben rögzített, vannak egy-, két-, illetve háromoperandusú műveletek, és ez rögzített minden operátorhoz (a **`+`** és **`-`** operátortoknak van egy-, illetve kétoperandusú változata is)
- A beépített típusok előre definiált operátorokkal rendelkeznek



## C# alapismeretek
### Operátorok

- Aritmetikai:
  - pozitivitás (**`+a`**), negáció (**`-a`**)
  - értéknövelés (**`a++`**, **`++a`**), értékcsökkentés (**`a--`**, **`--a`**)
  - összeadás (**`a + b`**), kivonás (**`a - b`**), szorzás (**`a * b`**), osztás (**`a / b`**), maradékképzés (**`a % b`**)
  - túlcsordulás ellenőrzés (**`checked`**, **`unchecked`**)
- Értékadás:
  - egyszerű (**`a = b`**)
  - összetett (**`a += b`**, **`a -= b`**, **`a *= b`**, **`a /= b`**, **`a %= b`**, **`a <<= b`**, **`a >> b`**, **`a &= b`**, **`a |= b`**, **`a ^= b`**)
  - feltételes (**`a ? b : c`**)



## C# alapismeretek
### Operátorok

- Függvényhívás (**`a()`**)
- Logikai:
  - érték összehasonlítás (**`a < b`**, **`a > b`**, **`a <= b`**, **`a >= b`**, **`a == b`**, **`a != b`**)
  - tagadás (**`!a`**), és (**`a && b`**), vagy (**`a || b`**)
- Memória:
  - memóriajelenlét ellenőrzés (**`??`**)
  - referencia (**`&a`**), dereferencia (**`*a`**)
  - taghivatkozás (**`a.b`**), mutató általi taghivatkozás (**`a->b`**)
  - méretlekérdezés (**`sizeof a`**, **`sizeof(<típus>)`**)
  - memóriaterület lefoglalás (**`new <típus>`**)



## C# alapismeretek
### Operátorok

- Indexelés (**`a[b]`**)
- Típuskezelés:
  - típusazonosítás (**`is`**), típuskezelés (**`as`**)
  - explicit típuskonverzió (**`(<típus>) a`**)
  - típusazonosítás (**`typeof a`**, **`typeof(<típus>)`**)
- Bitenkénti:
  - eltolás balra (**`a << b`**), eltolás jobbra (**`a >> b`**)
  - komplementer képzés (**`~a`**), bitenkénti és (**`a & b`**), bitenkénti vagy (**`a | b`**), különbségképzés (**`a ^ b`**)

## C# alapismeretek
### Operátorok precedenciája

- Az operátorok precedenciája meghatározza, a műveletek halmozása esetén milyen sorrendben történik a végrehajtás
  - a precendencia típusfüggetlen, és rögzített
  - a magasabb precendenciájú hajtódik előbb végre
  - zárójelek használatával befolyásolhatjuk a végrehajtási sorrendet
- C# precedenciák:
  1. `++` (postfix), `--` (postfix), `[]`, `()`, `.`, `new`, `typeof`, `checked`, `unchecked`
  2. `+` (unáris), `-` (unáris), `!`, `~`, `++` (prefix), `--` (prefix), (`<típus>`)
  3. `*`, `/`, `%`
  4. `+`, `-`



## C# alapismeretek
### Operátorok precedenciája

5. `<<`, `>>`
6. `>`, `<`, `>`, `<`, `is`, `as`
7. `==`, `!=`
8. `&`
9. `^`
10. `|`
11. `&&`
12. `||`
13. `?:`
14. `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `|=`, `^=`

- Pl.:
  `a * b – c == 7` jelentése: `((a * b) – c) == 7`
  `c = a == b % 2` jelentése: `c = (a == (b % 2))`
  `++a++` jelentése: `++(a++)`



## C# alapismeretek
### Példa #1

# SimpleSummation



## C# alapismeretek
### Vezérlési szerkezetek

- *Szekvencia*: a `;` tagolja az utasításokat
- *Programblokk*: `{ <utasítások> }`
- *Elágazás*:
  - *kétágú elágazás*:
    ```
    if (<feltétel>) <utasítás>; // igaz ág
    else <utasítás>; // hamis ág
    ```
    - a csellengő `else` mindig az utolsó elágazáshoz tartozik
  - *többágú elágazás*:
    ```
    switch(<változó>){
       case <konstans> : <utasítások>; break;
       …
       default: <utasítások>; break;
    }
    ```



## C# alapismeretek
### Vezérlési szerkezetek

  - egy adott változó értéke függvényében kerülnek különböző ágak végrehajtásra
  - alkalmazható egész, karakter és szöveg típusú változókra
  - alapértelmezett (`default`) ág nem kötelező
  - a lezárás (`break`), visszatérés (`return`) vagy továbbadás (`goto`) kötelező
- *Ciklusok*:
  - *számláló ciklus*:
    ```
    for (<inicializálás>; <feltétel>; <léptetés>)
       <utasítás>;
    ```
  - *előtesztelő ciklus*:
    ```
    while(<feltétel>) <utasítás>;
    ```



## C# alapismeretek
### Vezérlési szerkezetek

- *utántesztelő ciklus*:
  ```
  do <utasítás>; while(<feltétel>);
  ```
- *bejáró ciklus*:
  ```
  foreach (<deklaráció> in <gyűjtemény>)
     <utasítás>;
  ```
  - egy gyűjtemény értékein tud végighaladni
  - a gyűjtemény olyan osztály, amely megvalósítja az `IEnumerable` interfészt (a `GetEnumerator()` metódussal)
- ciklusból kilépés bármikor lehetséges a felvételtől függetlenül (`break`), valamint feltétel kiértékeléshez történő ugrás (`continue`) is lehetséges

## C# alapismeretek
### Példa #2

# SimpleFactorial



## C# alapismeretek
### Tömbök

- A tömbök is objektumok, a **`System.Array`**, vagy leszármazottjának példányai, emiatt referenciaként tárolt és intelligens (pl. a méretét a **`Length`** tulajdonsággal érhetünk el)
- Deklarációra több lehetőség is adott:
  - **`<típus>[]`** szokványos egydimenziós tömb
  - **`<típus>[][]`** szokványos egymásba ágyazott tömb
  - **`<típus>[,]`** kétdimenziós tömb
- Definícióhoz létre kell hozni a tömböt a méret megadásával, vagy az értékek megadásával, pl.:

```
int[] t = new int[4]; // minden eleme 0 lesz
int[] t = new int[] {1, 2, 3, 4}; // t.Length == 4
int[,,] m = new int[10,5,2]; // 3 dimenziós
```



## C# alapismeretek
### Felsorolási típus

- A *felsorolási típus* (**`enum`**) értékek egymásutánja, ahol az értékek egész számoknak felelődnek meg (automatikusan 0-tól sorszámozva, de ez felüldefiniálható)
  - pl.: **`enum Munkanap { Hétfő, Szerda = 2, Csütörtök}`**
  - a hivatkozás a típusnéven át történik, pl.:
    **`Munkanap mn = Munkanap.Hétfő`**
  - egy változó több értéket is eltárolhat, így több értékre is igaz lehet, pl.:
    **`Munkanap mn = Munkanap.Hétfő | Munkanap.Szerda`**
  - többágú elágazásban használható feltételként
  - ez is egy osztály a **`System`** névtérben:
    **`public abstract class Enum : ValueType, …`**



## C# alapismeretek
### Osztályok

- A C# programozási nyelv tisztán objektum-orientált, ezért minden érték benne objektum, és minden típus egy osztály
  - az osztály lehet érték szerint kezelt (*struct*), vagy referencia szerint kezelt (*class*), előbbi élettartama szabályozott az őt tartalmazó blokk által, utóbbié független tőle
  - az osztály tagjai lehetnek mezők, metódusok, illetve tulajdonságok (*property*), utóbbi lényegében a lekérdező (*get*) és beállító műveletek (*set*) absztrakciója
  - minden tagot jelöl a láthatósággal, a látható tagokat a **`public`**, a rejtett tagokat a **`private`**, a védett tagokat a **`protected`** kulcsszóval
  - az osztályok tetszőlegesen egymásba ágyazhatóak



## C# alapismeretek
### Osztályok felépítése

- A C# osztály szerkezete:

```
class/struct <osztálynév> {
   <láthatóság> <típus> <mezőnév> = <érték>;
   …
   <láthatóság> <típus> <metódusnév>
     ([ <paraméterek> ]) { <működés> }
   …
   <láthatóság> <típus> <tulajdonságnév> {
      [ get { <működés> } ]
      [ set { <működés> } ]
   }
   …
}
```



## C# alapismeretek
### Osztályok felépítése

- A *mezők* típusból és névből állnak, illetve kaphatnak alapértelmezett értéket (csak referencia szerinti osztályban)
  - a mezők alapértelmezett értéket kapnak, amennyiben nem inicializáljuk őket
- A *metódusok* visszatérési típussal (amennyiben nincs, akkor void), névvel és paraméterekkel rendelkeznek
  - a konstruktor neve megegyezik a típussal, külön visszatérési típusa nincs, a desktruktort a **~** jellel jelöljük
  - a metódusok tetszőlegesen túlterhelhetőek
  - lehetnek alapértelmezett paraméterek (csak a lista végén), továbbá a paraméterek átadhatóak név szerint (így az alapértelmezett sorrend felülírható)

## C# alapismeretek
### Osztályok felépítése

- A tulajdonság egy könnyítés (szintaktikus cukorka) a programozónak a lekérdező és író műveletek absztrakciójára
  - a beállító tulajdonság esetén a **`value`** pszeudóváltozó veszi át az értéket
  - pl. megvalósítás C++-ban:

```
class SomeClass{
   private:
      int pIntValue; // rejtett változó
   …
   public:
      int getIntValue() { return pIntValue;}
      void setIntValue(int v) {pIntValue = v; }
      // látható lekérdező és beállító művelet
}
```



## C# alapismeretek
### Osztályok felépítése

- ugyanez C#-ban:

```
class SomeClass{
   private int pIntValue; // rejtett változó
   …
   public int IntValue{
      get { return pIntValue; }
      set { pIntValue = value; }
   } // változóhoz tartozó látható tulajdonság
}
…
SomeClass s = new SomeClass();
s.IntValue = 10; // a 10 kerül a value-ba
```

- külön definiálható csak lekérdező, csak beállító, vagy mindkettőt elvégző tulajdonság, és a láthatóságuk is



## C# alapismeretek
### Példa #3

# SimpleRational



## C# alapismeretek
### Elemi osztály

- Az *elemi osztály* (*struct*) egy egyszerűsített osztály, amely:
  - mindig érték szerint kezelődik, a példány automatikusan megsemmisül a blokk végén
  - nem szerepelhet öröklődésben (sem ősként, sem leszármazottként), de implementálhat tetszőleges számú interfészt
  - automatikus őse a **`System.ValueType`** (ami leszármazottja a **`System.Object`**-nek, ugyanakkor definiálja az érték szerinti kezelést)

```
[<láthatóság>] [<módosítók>] struct <név>
  [: <interfészek>] {
   <definíciók>
}
```



## C# alapismeretek
### Elemi osztály

- További megkötések:
  - nem lehet alapértelmezett konstruktora, és destruktora (az érték szerinti kezelés miatt)
  - nem lehet az elemeit inicializálni (minden elem az alapértelmezett értéket kapja meg)
  - nem alkalmazhatóak az **`abstract`**, **`virtual`**, **`sealed`**, **`protected`** kulcsszavak
- Általában egyszerű, rekordszerű szerkezethez használjuk, amelyek érték szerinti kezelése, másolása nem rontja a program teljesítményét
  - ugyanakkor tetszőleges mértékig növelhetjük a tulajdonságaikat



## C# alapismeretek
### Referencia osztály

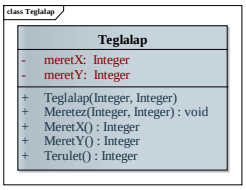
- A *referencia osztály* (*class*) a teljes értékű osztály, amely származtatásban is szerepelhet

```
[<láthatóság>] [<módosítók>] class <név>
  [: <ős>, <interfészek>]
{
   <definíciók>
}
```

  - csak egy őse lehet, de tetszőleges számú interfészt valósíthat meg
  - mezőit lehet közvetlenül inicializálni, vagy nulla paraméteres konstruktor segítségével
  - az öröklődés miatt lehet absztrakt osztály, és szerepelhetnek benne absztrakt és virtuális elemek

## C# alapismeretek
### Példa #4

- A téglalap osztály megvalósítása:
  - mezői a két oldal mérete, ezeket elrejtjük
  - műveletei az átméretezés, illetve a méretek és a terület lekérdezése, ezeket láthatóvá tesszük
- A téglalap osztály terve (UML osztálydiagramja):





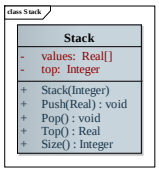

## C# alapismeretek
### Példa #4

SimpleRectangle



## C# alapismeretek
### Példa #5

- A verem (stack) osztály megvalósítása:
  - a verem egy tömb és elemszám segítségével írható le, amiket elrejtünk
  - műveletei a behelyezés (push), kivétel (pop), tetőelem (top), illetve méret (size) lekérdezés, ezeket láthatóvá tesszük
- A verem osztály terve (UML osztálydiagramja):






## C# alapismeretek
### Példa #5

SimpleStack



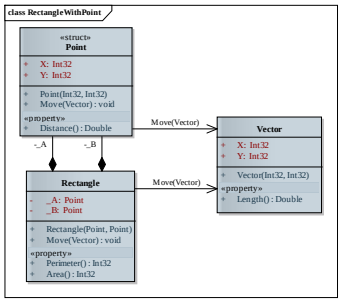
## C# alapismeretek
### Példa #6

- Ábrázoljuk a téglalapot két ellentétes sarokpontja koordinátájával, és adjunk meg egy eltolási műveletet, amely tetszőleges vektorral arrébb tudja helyezni a téglalapot.
  - a téglalap (**`Rectangle`**) osztály tartalmazni fogja a pont (**`Point`**) osztály két példányát, lekérdezhetjük a kerületét, és területét
  - a pontot ennek megfelelően kezelhetjük érték szerint, két egész számot tartalmaz, és lekérdezhetjük az origótól való távolságát
  - a négyzet eltolásához a pontban is megvalósítjuk az eltolás műveletét, ehhez szükséges a vektor (**`Vector`**) osztályt, amely szintén két egész számot tartalmaz



## C# alapismeretek
### Példa #6

*Tervezés:*



## C# alapismeretek
**Példa #6**

# RectangleWithPoints



## C# alapismeretek
**Nyílt rekurzió**

- Az objektum mindig tisztában van saját állapotával, vagyis elérheti mezőit, és azok aktuális értékét, és saját magából is meghívhatja metódusait
- Lényegében azt mondhatjuk, hogy az objektum rendelkezik saját maga felett, eléri saját magát, ezt *nyílt rekurzió*nak (*open recursion*) nevezzük
  - a nyílt rekurzióban az objektumon belül mindig elérhetjük az objektum hivatkozását (mutatón keresztül), a C#-ban erre a **`this`** kulcsszó használható
  - a nyílt rekurzió amellett, hogy lehetővé teszi a hivatkozáson keresztül is a tagok elérését (**`this.<tagnév>`**), lehetőséget ad saját magára való hivatkozás átadására (pl. metódus paraméterében)



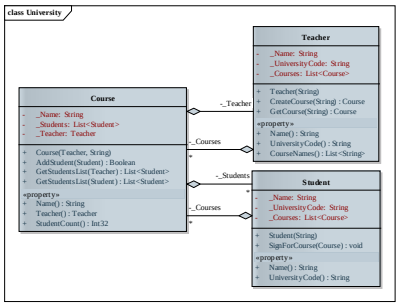
## C# alapismeretek
**Példa #7**

- Egy egyetemi kurzust egy oktató tart, és több hallgató veheti fel, és ennek megfelelően a kurzusnak tisztában kell lennie hallgatóival és oktatójával, ugyanakkor az oktatónak és a hallgatóknak is tisztában kell lenniük kurzusaikkal.
  - mind az oktató, mind a hallgató névvel, valamint egyetemi azonosítóval rendelkezik, az oktatótól lekérdezhetőek a kurzusok nevei
  - a kurzus hallgatóit csak az oktató módosíthatja, de az összes felvett hallgató lekérdezheti
  - az adatok kölcsönös eltárolását hivatkozások segítségével valósítjuk meg, hiszen minden objektum élettartama független kell, hogy legyen



## C# alapismeretek
**Példa #7**

*Tervezés:*






## C# alapismeretek
**Példa #7**

# University



## C# alapismeretek
**Interfész**

- Az *interfész* (*interface*) deklarációk halmaza, egy osztályfelület, amely nem példányosítható, csak arra szolgál, hogy osztályok közös tulajdonságait összefogja

```
[<láthatóság>] interface <név> {
    <deklarációk>
}
```

  - a többszörös öröklődés kiküszöbölésére szükséges
  - nem tartalmaz megvalósítást, azaz lényegében minden eleme absztrakt (de nem kell kiírni a kulcsszót)
  - tartalmazhat metódusokat és tulajdonságokat
  - minden eleme publikus, ezért nem adunk meg láthatóságot
  - a névkonvenció szerint az interfész nevét I betűvel kezdjük

## C# alapismeretek
**Példa #8**

# SimpleComplex



## C# alapismeretek
**Kivételkezelés**

- A program által kiváltott bármilyen nem szabályos tevékenység kivételként jelenik, és lekezelhető
- A kivétel általános osztálya az **`Exception`**, de a műveletek rendszerint ennek valamely speciálisabb változatát keltik
- Kivételt kezelni egy kivételkezelő (**`try-catch-finally`**) szakasszal tudunk, amelyben meg kell adnunk az elfogandó kivétel típusát, és kivétel esetén lefuttathatunk egy megadott utasítássorozatot:

```
try { <kivételkezelt utasítások> }
catch (<elfogott kivétel típusa>){
   <kivételkezelő utasítások>
}
finally { <mindenképp lefuttatandó utasítások> }
```



## C# alapismeretek
**Kivételkezelés**

- Kivételkezelő szakaszt bármely metóduson belül elhelyezhetünk a programban
  - ha a **`try`** blokkban kivétel keletkezik, akkor a vezérlés a **`catch`** ágra ugrik, az utána következő utasítások így nem futnak le
  - a program ellenőrzi, hogy a kivétel típusa egyezik-e, vagy speciális esete a **`catch`**-ben megadottnak, különben tovább dobja a kivételt
  - ha elfogta a kivételt, akkor futtatja a **`catch`** ág utasításait
  - a **`finally`** blokk használata nem kötelező, de amennyiben van, úgy az abban lévő utasításokat minden esetben lefuttatja (akkor is, ha keletkezett nem lekezelt kivétel)



## C# alapismeretek
**Kivételkezelés**

- Lehetőségünk van különböző típusú kivételek elfogására is, amennyiben több **`catch`** ágat készítünk a szakaszhoz
  - a kivétel típusát sorban egyezteti az ágakon, és az első találat ágát futtatja
  - amennyiben biztosan el akarunk kapni bármilyen kivételt, kapjuk el az általános **`Exception`** típust is
  - pl.:

```
try { // kivételkezelt utasítások
   …
} // kivételkezelő ágak:
catch (ArgumentException) { … }
catch (NullReferenceException) { … }
catch (Exception) { … }
```



## C# alapismeretek
**Kivételkezelés**

- A kivételek üzenettel rendelkeznek, amelyet a kivétel **`Message`** tulajdonságán keresztül kérhetünk le
- Kivételt mi is kiválthatunk tetszőleges pontján a programnak, amelyet egy őt meghívó metódusban kezelni tudunk
- Kivételt kiváltani a **`throw`** utasítással tudunk:

```
throw new <kivétel típusa>(<kivétel szövege>);
```

  - az utasításra a program futása megszakad, és a következő kivételelfogó utasításra kerül a vezérlés
  - a kivétel típusa lehet egy beépített kivételtípus (pl. **`ArgumentException`**, **`IndexOutOfRangeException`**, **`Exception`**), valamint mi is definiálhatunk kivétel típusokat
  - a kivétel szövegét nem kötelező megadni



## C# alapismeretek
**Generikus típusok**

- Generikus programozásra futási időben feldolgozott sablon típusok (*generic*-ek) segítségével van lehetőség
  - a sablon fordításra kerül, és csak a futásidejű fordításkor helyettesítődik be a konkrét értékre
  - a sablonos osztályt szokás szervernek, a példányosítását kliensnek nevezni
  - pl.:

```
class GenericClass<T>{
   public T item; // használható a T típusként
}
//...
GenericClass<int> gc = new GenericClass<int>();
gc.item = 1;
```

## C# alapismeretek
**Példa #9**

GenericSample



## C# alapismeretek
**Generikus típusok**

- Mivel szigorú típusellenőrzés van, ezért fordítási időben a sablonra csak az **`Object`**-ben értelmezett műveletek használhatóak, ezt a műveletkört növelhetjük megszorításokkal
  - a megszorítás (**`where`**) korlátozza a típus behelyettesítési értékeit, és ezáltal bővíti az alkalmazható metódusok és tulajdonságok körét
  - a behelyettesítéskor így csak az adott típus, vagy annak leszármazottja szerepelhet
  - korlátozásnál csak egy osztály, és mellette tetszőleges számú interfész adható meg, ekkor a behelyettesített típusnak valamennyit meg kell valósítania, ez fordítási időben van ellenőrizve



## C# alapismeretek
**Generikus típusok**

- Lehetséges megszorítások:
  - A T típus a Control leszármazottja és megvalósítja az ICloneable és az IDisposeable interfészeket.
    ```
    class GenericClass<T> where T :
            Control, ICloneable, IDisposable
    ```
  - A T típusnak referencia típusúnak kell lennie.
    ```
    class GenericClass<T> where T : class
    ```
  - A T típusnak érték típusúnak kell lennie.
    ```
    class GenericClass<T> where T : struct
    ```
  - A T típusnak rendelkezik alapértelmezett konstruktorral.
    ```
    class GenericClass<T> where T : new()
    ```
- Az előbbiek értelemszerűen vegyíthetőek is.