Eötvös Loránd Tudományegyetem
Informatikai Kar

Eseményvezérelt alkalmazások fejlesztése II

A .NET keretrendszer és a C# programozási nyelv

groberto@inf.elte.hu
http://people.inf.elte.hu/groberto

Objektumorientált programozási nyelvek

A Smalltalk nyelv

- 1980-ban a *Smalltalk* volt az első tisztán objektumorientált nyelv, amely számos jellemzőjét bevezette a nyelvcsaládnak
  - teljesen hordozható kódot biztosít *virtuális gép*en történő futtatás segítségével
  - minden implementált elem (változók, konstansok, metódusok) egy objektum, és egyben egy osztály példánya, az osztályok egymástól örökölnek, és egy *teljes származtatási hierarchiá*ban vannak
  - a memóriakezeléshez *szemétgyűjtő*t használ
  - *dinamikus programozás* támogatása, azaz a program futás közben képes manipulálni a programkódot

Objektumorientált programozási nyelvek

A virtuális gép

- A hordozhatóság akadálya, hogy a fordítással keletkezett alacsonyszintű programkód gépfüggő, ezért a *Smalltalk* programokat egy gépfüggetlen *köztes nyelv*re (*Intermediate Language*) fordították
- A köztes nyelvű program egy értelmező szoftver segítségével futtatható, amely futás közben alakítja gépi kóddá a programkódot, ezt az értelmezőt nevezzük virtuális gépnek (*Virtual Machine*)
- Ezt a félig fordított, félig értelmezett megoldást nevezzük *futásidejű fordítás*nak, vagy röpfordításnak (*Just In Time Compilation*)

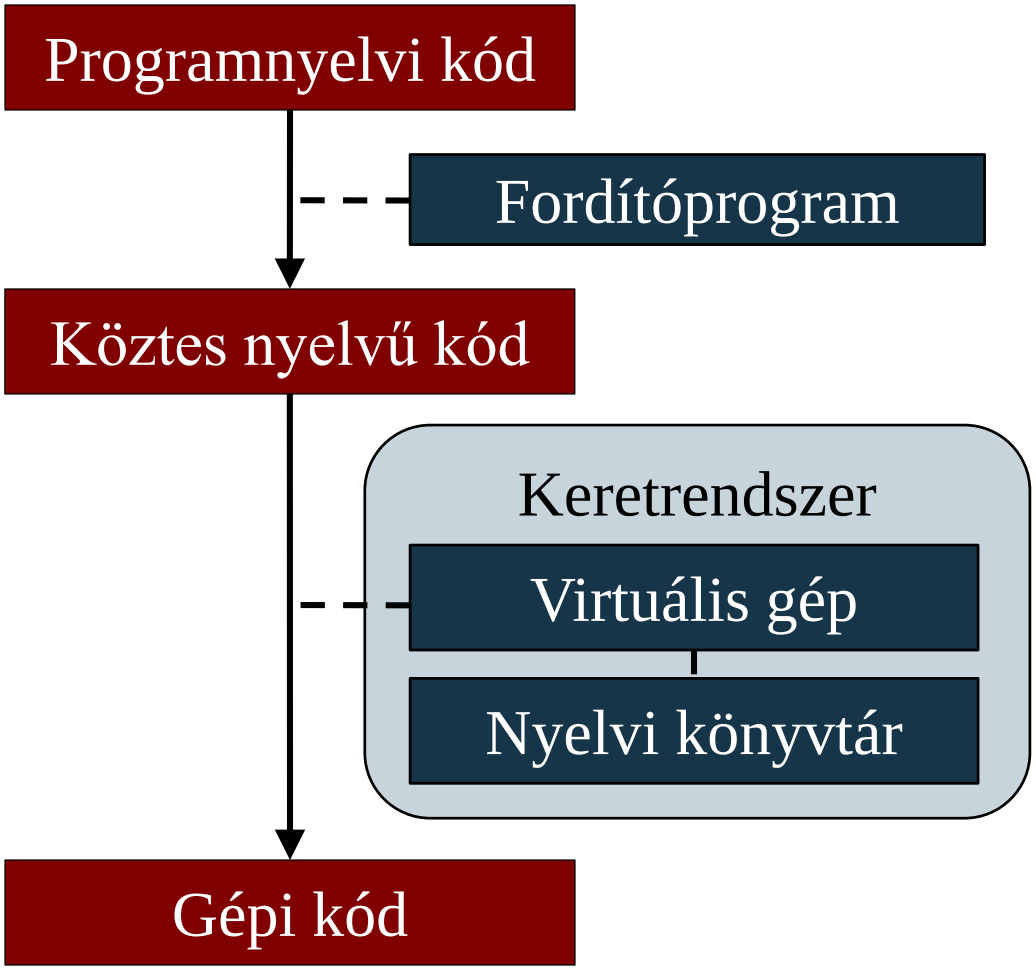
Objektumorientált programozási nyelvek

A szoftver keretrendszer

- Mivel a fordítás egy része, és az összeszerkesztés futási időben történik, ezért kell minden programkomponenst beágyazni a programba, a hivatkozások feloldása történhet futtatáskor is
  - csökkenthető a program mérete és a betöltés ideje
  - a kiemelt komponenseknek jelen kell lenniük a gépen
- *Szoftver keretrendszer*nek nevezzük a kiemelt programkönyvtár és a virtuális gép együttesét
  - tartalmazza az API gépfüggetlen, absztrakt lefedését
  - felügyeli a programok futásának folyamát
  - biztosítja a memóriakezelést, szemétgyűjtést

Objektumorientált programozási nyelvek

A szoftver keretrendszer



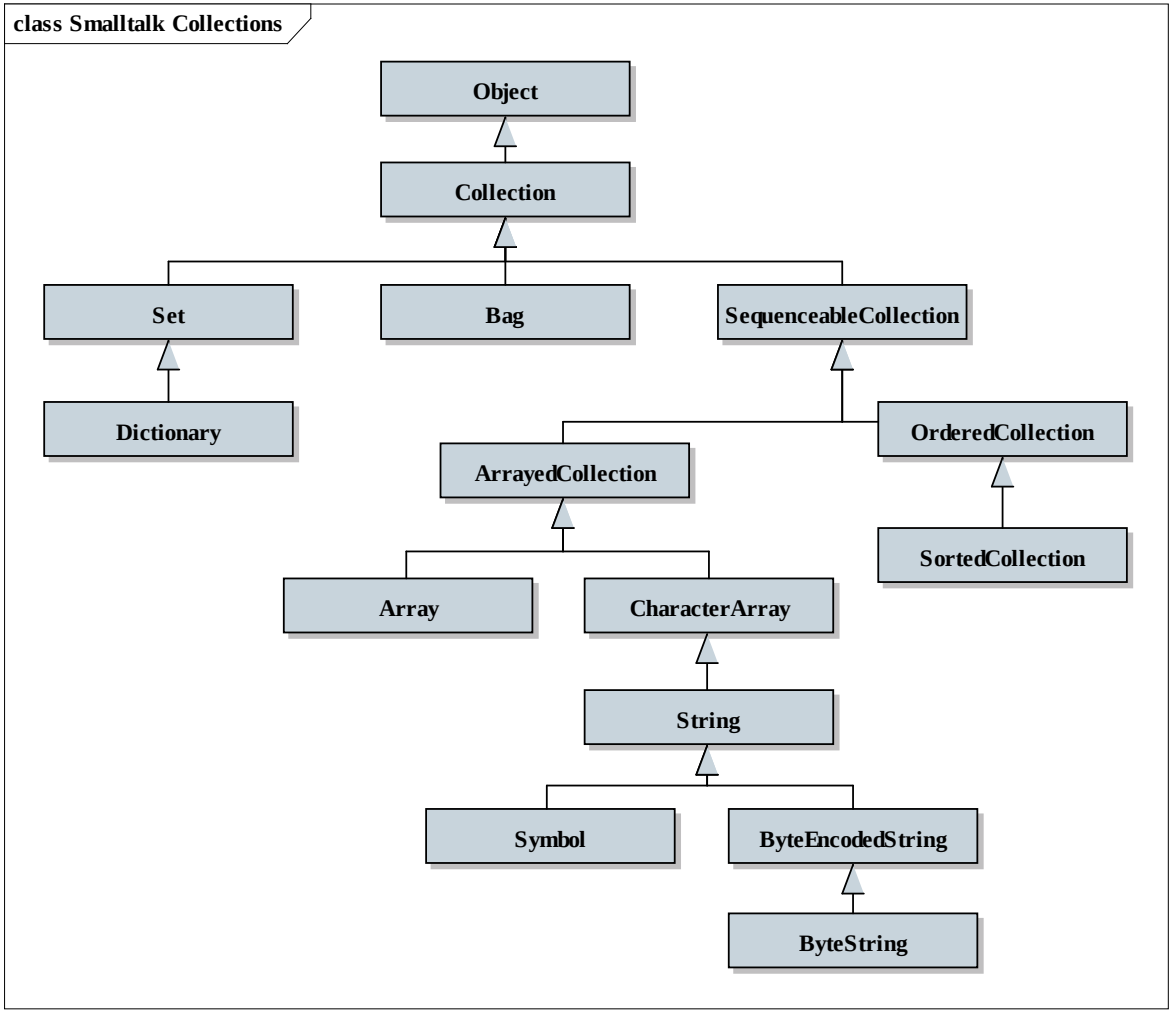
Objektumorientált programozási nyelvek

Teljes származtatási hierarchia

- Teljes származtatási hierarchiában van egy egyetemes ős osztály, minden más osztály ennek leszármazottja (akkor is, ha nincs megjelölve ősként)
  - az ős definiálja az alapértelmezett viselkedését minden objektumnak (pl. szöveggé alakítás, másolás,…)
  - minden beépített osztály egy származtatási hierarchia mentén van implementálva
  - ezen láncok mentén specializálódnak, illetve kiegészülnek az osztályok viselkedései
  - a hierarchia mentén találhatóak megkötések is a származtathatóságra és a felüldefiniálhatóságra

Objektumorientált programozási nyelvek

Teljes származtatási hierarchia



Objektumorientált programozási nyelvek

Memóriakezelés

- Az objektumorientált nyelvekben célszerű referenciákon, illetve mutatókon keresztül hivatkozni az objektumokra, hatékonysági és élettartam szabályozási okok folytán
  - mutatók esetén a létrehozást és törlést manuálisan kell megírni, ám a törlés sokszor elmarad, ezért a memóriában maradnak nem hivatkozott objektumok (a szemét)
- A virtuális gép feladata, hogy felügyelje a program által elfoglalt memóriaterületet, és a benne lévő memóriaszemetet eltávolítsa, ezt nevezzük *szemétgyűjtés*nek (*Garbage Collection*)
  - ciklikusan ellenőrzi a memóriát és a hivatkozásokat, és kiüríti a nem hivatkozott memóriaterületeket

Objektumorientált programozási nyelvek

Metaosztályok és dinamikus programozás

- Mivel minden objektum, maguk az osztályok is objektumok, és az ő viselkedési mintájukat is definiálni kell, erre a célra szolgálnak a *metaosztályok*
  - közös felületet biztosít osztálytulajdonságok és azok részleteinek lekérdezésére, metódusok futtatására, módosítására és példányosításra
  - az objektumok és osztályok lehetőséget adnak a metaosztály lekérdezésére és az osztályhoz tartozó *metaobjektum* létrehozására
- A metaobjektumokon át bármely osztályt módosíthatunk futás közben, ezáltal dinamikusan programozhatóvá válik a rendszer

Objektumorientált keretrendszerek

A Java

- 1991-ben indult el a *Java* fejlesztése a *Sun Microsystems*-nél, amelynek célja egy objektumorientált, általános célú, hordozható kódú, sokplatformos programozási nyelv megalkotása volt
  - könnyű programozhatóságot, ugyanakkor lassabb programfuttatást eredményezett, a programok a Java virtuális gépen (*JVM*) futottak
  - pár év alatt, különösen a mobil és webes alkalmazásoknak köszönhetően nagy népszerűségre tett szert
  - a *Microsoft* saját virtuális gépet, és nyelvi könyvtárat fejlesztett ki (*Virtual J++*), amely gyorsabb futtatást tett lehetővé

Objektumorientált keretrendszerek

A .NET keretrendszer

- 1998-tól a Microsoft új célja a futásidejű fordítás kiterjesztése a létező Microsoft nyelvekre (Visual C++, Visual Basic), ezáltal egy közös virtuális gépen futhatnak a programok
  - egységes köztes nyelv vezethető be, így a nyelvek tetszőleges mértékben kombinálhatóak egymással
  - egységes programkönyvtárak használhatóak
- 2001-re készült el a *.NET Framework*, és a dedikáltan ráépülő nyelv, a C#

  - azóta integrált része a Windowsnak
  - hozzáférést biztosít az összes Microsoft technológiához (*COM*, *ODBC*, *DirectX*)

Objektumorientált keretrendszerek

A .NET keretrendszer

- Egységes virtuális gép: *Common Language Runtime* (*CLR*)
- Egységes köztes nyelv: *Common Intermediate Language* (*CIL*)
- Egységes típusrendszer: *Common Type System* (*CTS*)
- Teljeskörű programkönyvtár
  - *Base Class Library* (*BCL*): gyűjtemények, I/O kezelés, adatkezelés, hálózat, párhuzamosítás, XML, …
  - *Framework Class Library* (*FCL*): WinForms, ASP.NET, LINQ, WPF, WCF
- Biztosítja a programok védettségét és hordozhatóságát
  - memóriakezelés felügyelete: *Managed Code*
  - köztes kód védelme: *Code Access Security* (*CAS*)

Objektumorientált keretrendszerek

A .NET keretrendszer

- 1.1 (2003): kompakt keretrendszer, kódbiztonság (CAS)
- 2.0 (2005): eléri a Java funkcionalitását (sablonok, parciális osztályok, névtelen metódusok), új adatkezelés (ADO.NET)
- 3.0 (2007): új technológiák a kommunikációra és megjelenítésre (WCF, WPF, WF, CardSpace)
- 3.5 (2008): funkcionális programozás, nyelvbe ágyazott lekérdezések (LINQ), AJAX támogatás
- 4.0 (2010): párhuzamosítás támogatása (PLINQ, TPL), szerződés alapú programozás (DbC)
- 4.5 (2012): nyelvi szintű párhuzamosítás, Modern UI

Objektumorientált keretrendszerek

A .NET keretrendszer

- Összesen 37 nyelv biztosít támogatást a .NET keretrendszer felé, az elsődleges nyelvek:
  - *C#*: a Visual J++ utódnyelve, amely eltávolodik a Java szintaxisától, több ponton visszatér a C++-hoz
  - *Visual Basic .NET*: a Visual Basic továbbfejlesztése
  - *C++/CLI (C++.NET)*: C++ szintaxis kiegészítve a .NET könyvtárakra memóriafelügyelettel
  - *F#*: funkcionális programozási nyelv
- További jelentős nyelvek: Cobra, IronScheme, IronPython, IronRuby, Scala

Objektumorientált keretrendszerek

A .NET keretrendszer

- A .NET keretrendszerhez kapcsolódó keretrendszerek:
  - *XNA Framework:* 3D játékprogramok fejlesztéséhez (XBox, Windows Phone)
  - *Silverlight*: gazdag internet alkalmazások, valamint mobil alkalmazások fejlesztéséhez (webböngészők, Windows Phone)
  - *Windows Runtime* (*WinRT*): mobil és táblagép alkalmazások fejlesztéséhez (*Windows RT*, *Windows Phone*)
- A .NET keretrendszert csak Microsoft platformokra fejlesztik, de alternatívát biztosít a *Mono*, amely számos más platformra (*Linux*, *OS X*, *iOS*, *Android*) biztosít támogatást

Objektumorientált keretrendszerek

Hátrányai

- A futásidejű fordítás miatt
  - szükséges a keretrendszer megléte
  - lassúbb programindítás (nagyságrendi elmaradás a fordított programokhoz képest)
  - a köztes kód teljes mértékben visszafejthető bármely felsőbb szintű nyelvbe (ez valamelyest kivédhető kódzavaró eszközökkel, de nem teljesen)
- A memóriafelügyelet miatt
  - megnövelt erőforrásigény (memória ellenőrzés miatt)
  - késleltetés a szemétgyűjtő futásakor (valós idejű alkalmazások implementálása lehetetlen)

A C# programozási nyelv

A nyelv lehetőségei

- A C# *tisztán objektumorientált programozási nyelv*, amely teljes mértékben a *.NET Framework*re támaszkodik
  - szintaktikailag nagyrészt C++, megvalósításában Java
  - egyszerűsített szerkezet, strukturált felépülés névterekkel
  - tisztán objektumorientált, minden típus egy .NET keretrendszerbeli osztály, vagy leszármazottja
  - támogatja a sablon-, eseményvezérelt, funkcionális programozást
  - a forrásfájl kiterjesztése: **`.cs`**
  - kódolás: *Unicode 3.0*

A C# programozási nyelv

A „Hello, World!” program

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

A C# programozási nyelv

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
    ```

A C# programozási nyelv

Névterek

- 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
  - a típusnév előtt is megadhatjuk a használandó névteret (így nem kell **`using`**), pl.: **`System.Collections.Stack s;`**
  - típusnév ütközés esetén mindenképpen ki kell írnunk a teljes elérési útvonalat

A C# programozási nyelv

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 mutatók, 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 C# programozási nyelv

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`** (**`Boolean`**)
  - egész számok (előjeles és előjel nélküli):
    - 8 bites: **`sbyte`** (**`SByte`**), **`byte`** (**`Byte`**),
    - 16 bites: **`short`** (**`Int16`**), **`ushort`** (**`UInt16`**)

- 32 bites: **`int`** (**`Int32`**), **`uint`** (**`UInt32`**)
- 64 bites: **`long`** (**`Int64`**), **`ulong`** (**`UInt64`**)

- lebegőpontos számok:
  - 32 bites: **`float`** (**`Single`**)
  - 64 bites: **`double`** (**`Double`**)
- tizedestört szám: **`decimal`** (**`Decimal`**)
- karakter: **`char`** (**`Char`**)

- Referencia szerinti primitív típusok:
  - objektum (ősosztály): **`object`** (**`System.Object`**)
  - szöveg: **`string`** (**`System.String`**)

A C# programozási nyelv

Primitív típusok

- Már a primitív típusok is intelligensek C#-ban, azaz támogatnak számos műveletet és speciális értéklekérdezést, 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)`**

A C# programozási nyelv

Példányosítás

- Változókat bármely (nem névtér) blokkon belül létrehozhatunk a programkódban típus, név és kezdőérték megadásával
  - pl.: `Int32 myInt = 10;`
  - kezdőérték megadása nem kötelező, de a változó addig nem használható fel, amíg nem kap értéket (fordítási hiba)
  - összetett típusok esetén a **`new`** operátort használjuk, pl.:
    `Stack<Int32> s = new Stack<Int32>();`
- Típusnév helyett használható a **`var`** kulcsszó, ekkor a típus automatikusan behelyettesítődik fordítási időben
  - pl.: `var myInt = 10;`
  - ez csupán rövidítés, nem gyengít a típusosságon

A C# programozási nyelv

Konstansok

- A konstansok is intelligens objektumok, pl. **`10.ToString()`**, **`"Hello World".SubString(0, 5)`**
- A konstansok a megfelelő automatikus típust kapják meg, ez módosítható karakterliterálok (**`L`**, **`U`**, **`F`**, …) segítségével, pl.:

```
10 // típusa Int32 lesz
10L // típusa Int64 lesz
10UL // típusa UInt64 lesz
10.0 // típusa Double lesz
10.0F // típusa Single lesz
```

- Lehet 8-as (**`0`** előtag, pl. **`0173`**), illetve 16-os számrendszert használni (**`0x`** előtag, pl. **`0x3de2`**)
- Valós számok megadhatóak kitevős formában (pl. **`13.2E5`**)

A C# programozási nyelv

Konstansok

- Elnevezett konstansoknál használnunk kell a **`const`** kulcsszót, ekkor kötelező a kezdőérték megadása (csak konstans lehet)
  - pl.: **`const Int32 myConstInt = 10;`**
  - osztályok tagjai is lehetnek konstansok
- Lehetőségünk van csak olvasható mezők létrehozására is a **`readonly`** kulcsszóval, amelyek inicializálással (kezdőérték megadásával), vagy a konstruktorban kaphatnak értéket (lehet változó is)
  - pl.:

```
private readonly Stack<Int32> s; // osztályban
…
s = new Stack<Int32>(); // konstruktorban
```

A C# programozási nyelv

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
    - 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

A C# programozási nyelv

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 int 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);
```

A C# programozási nyelv

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`**)

A C# programozási nyelv

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`*`>`**)

A C# programozási nyelv

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`**)

A C# programozási nyelv

Operátorok precedenciája

1. `++`, `--` (postfix), `[]`, `()`, `.`, **`new`**, **`typeof`**, **`checked`**, **`unchecked`**
2. `+`, `-` (unáris), `!`, `~`, `++`, `--` (prefix), (**`<típus>`**)
3. `*`, `/`, `%`
4. `+`, `-`
5. `<<`, `>>`
6. `>`, `<`, `>`, `<`, **`is`**, **`as`**
7. `==`, `!=`
8. `&`
9. `^`
10. `|`
11. `&&`
12. `||`
13. `?:`
14. `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `|=`, `^=`

Példa

Feladat: Kérjünk be két valós számot a konzol képernyőn, és írjuk vissza az összegüket.

- a valós számok közül használjuk az egyszeres pontosságút (**`Single`**), a változóink legyenek **`a`** és **`b`**
- a konzol képernyőről beolvasni a **`Console.ReadLine()`** utasítással tudunk, amely szöveget ad vissza, így azt konvertálnunk kell (**`Convert.ToSingle`**)
- a konzolra kiírni a **`Console.Write()`** és **`Console.WriteLine()`** utasításokkal tudunk bármilyen típusú változót, vagy konstansot
- a kiíráshoz szöveget is társítunk, amelyhez a **`+`** operátor segítségével tudunk további értékeket fűzni

Példa

Megoldás:

```
using System; // felhasznált névtér

namespace SimpleSummation { // saját névtér
   class Program {

      static void Main(string[] args) {
         Single a, b, c;
         // a két beolvasandó szám, valamint az
         // eredmény

         Console.Write("Kérem az első számot: ");
         a = Convert.ToSingle(Console.ReadLine());
         // beolvasás és konvertálás
```

Példa

Megoldás:

```
            Console.Write("Kérem a második számot:
                          ");
            b = Convert.ToSingle(Console.ReadLine());

            c = a + b; // összeadás elvégzése
            Console.WriteLine("A két szám összege: "
                              + c); // kiírás
            Console.ReadKey(); // várakozás
        }
    }
}
```

A C# programozási nyelv

Vezérlési szerkezetek

- *Szekvencia*: a `;` tagolja az utasításokat
- *Programblokk*: **`{ <utasítások> }`**
- *Elágazás*:
  - *kétágú elágazás*:

```
if (<feltétel>)
   <utasítás>; // igaz ág
else
   <utasítás>; // hamis ág
```

    - a feltétel logikai kifejezés
    - az **`else`** ág elhanyagolható
    - a csellengő **`else`** mindig az utolsó elágazáshoz tartozik

A C# programozási nyelv

Vezérlési szerkezetek

- *többágú elágazás*:

```
switch (<változó>){
    case <konstans> : <utasítások>; break;
    case <konstans> : <utasítások>; break;
    …
    default: <utasítások>; break;
}
```

  - 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

A C# programozási nyelv

Vezérlési szerkezetek

- *Ciklusok*:
  - *számláló ciklus*:

```
for (<inicializálás>; <feltétel>; <léptetés>)
    <utasítás>;
```

  - *elöltesztelő ciklus*:

```
while(<feltétel>)
    <utasítás>;
```

  - *hátultesztelő ciklus*:

```
do
    <utasítás>;
while(<feltétel>);
```

A C# programozási nyelv

Vezérlési szerkezetek

- *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 való ugrás (**continue**) is lehetséges

A C# programozási nyelv

Példa

Feladat: Adjuk meg egy pozitív egész szám faktoriálisát.

- alkalmazzunk összegzést, amelyben a szorzás műveletét használjuk
- készüljünk fel arra, hogy a felhasználó nem garantált, hogy pozitív számot ad meg, ezért egy elágazással előbb válasszuk le a hibás eseteket, és írjunk ki figyelmeztető üzenetet
- az eredményváltozót egy nagyobb értékhalmazban vesszük fel (**`Int64`**), hogy garantáltan elférjen az eredmény

A C# programozási nyelv

Példa

Megoldás:

```
static void Main(string[] args){
   Int32 number;
   Int64 sum; // az eredményhez nagyobb
              // értéktartományt veszünk

   Console.WriteLine("Kérek egy pozitív egész
                      számot: ");
   number = Convert.ToInt32(Console.ReadLine());

   if (number <= 0) {
      Console.WriteLine("Mondom, pozitív egész
                         számot!");
```

Példa

Megoldás:

```
      } else {
         // itt már programblokk szükséges, mivel
         // több utasítás is helyet kap

         sum = 1; // összegzés
         for (Int32 i = 1; i <= number; i++)
            sum *= i; // vagy sum = sum * i;

         Console.WriteLine("A szám faktoriálisa: " +
                                 sum);
      }
      Console.ReadKey();
   }
```

A C# programozási nyelv

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
```

  - ez is egy osztály a **System** névtérben:

```
public abstract class Enum : ValueType, …
```

A C# programozási nyelv

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 beágyazott osztályok, mezők, metódusok, események, illetve tulajdonságok (*property*), utóbbi lényegében a lekérdező (*get*) és beállító műveletek (*set*) absztrakciója
  - minden tagnak, és magának az osztályt is jelöljük a láthatóságát (**`public`**, **`private`**, **`protected`**, **`internal`**)
  - a nyílt rekurziót a **`this`** kulcsszó biztosítja

A C# programozási nyelv

Osztályok felépítése

- A C# osztály szerkezete:

```
<láthatóság> class/struct <osztálynév> {
    <láthatóság> <típus> <mezőnév>; // mező
    …
    <láthatóság> <típus> <metódusnév>
      ([ <paraméterek> ]) { <működés> } // metódus
    …
    <láthatóság> <típus> <tulajdonságnév> {
       [ get { <működés> } ]
       [ set { <működés> } ]
    } // tulajdonság
    …
    <láthatóság> event <delegált> <eseménynév>;
      // esemény
}
```

A C# programozási nyelv

Osztályok felépítése

- Pl. C++:

```
class Rational {
   private:
      int num;
      int denom;
   …
   public:
      Rational(int, int);
   …
};
…
Rational::Rational(int n, int d) {
   num = n; denom = d;
}
```

A C# programozási nyelv

Osztályok felépítése

- Pl. C#:

```
struct Rational {
    private Int32 num; // mező
    private Int32 denom;
        // mindenhol jelöljük a láthatóságot
    …
    public Rational(Int32 n, Int32 d) { // metódus
        num = n;
        denom = d;
        // a deklaráció és a definíció nem
        // választható el
    }
    …
} // nem kell a végén ; ☺
```

A C# programozási nyelv

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
  - lehetnek alapértelmezett paraméterek (csak a lista végén), továbbá a paraméterek átadhatóak név szerint (így az alapértelmezett sorrend felülírható)

A C# programozási nyelv

Osztályok felépítése

- Paraméterátadáskor az elemi típusok érték szerint, a referencia típusok cím szerint másolódnak
- A paraméterekre módosítókat is alkalmazhatunk:
  - a **`ref`** kulcsszó cím szerint veszi át az elemi típust, illetve cím címe szerint a referencia típust
  - az **`out`** kulcsszó kimenő paramétert jelöl, ekkor a változó kötelező, hogy értéket kapjon a változó azt alprogramban
  - a **`params`** kulcsszóval lehetőségünk van a paraméterek számát tetszőleges hosszúra venni, ekkor azok egy tömbbe képződnek le (ekkor a paraméterek típusa rögzített), pl.:

```
void Format(String f, params Object[] args){ … }
```

Osztályok felépítése

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

```
class Rational {
   …
   int getDenominator() { return denom; }
   void setDenominator(int value) {
      denom = (value == 0) ? 1 : value;
   }
   // publikus lekérdező és beállító művelet
}
```

A C# programozási nyelv

Osztályok felépítése

- ugyanez C#-ban:

```
struct Rational {
   …
   public Int32 Denominator {
      get { return denom; }
      set { denom = (value == 0) ? 1 : value; }
   } // változóhoz tartozó látható tulajdonság
}
…
Rational r = new Rational(10, 5);
r.Denominator = 10; // a 10 kerül a value-ba
```

- külön definiálható csak lekérdező, csak beállító, vagy mindkettőt elvégző tulajdonság

Osztályok felépítése

- a műveletek láthatósága külön szabályozható
- lehetőségünk van automatikus tulajdonságok létrehozására is, amelyek egy mező lekérdezésére/beállítására szolgálnak
  - ekkor elég a tulajdonságokat létrehozni, a mező automatikusan generálódik
  - nem kell törzset adnunk, csak jelöljük a műveleteket
  - nem történik semmilyen ellenőrzés a beállításnál
  - pl.:

```
struct Rational {
    …
    public Int32 Denominator { get; set; }
        // automatikus tulajdonság
}
```

A C# programozási nyelv

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>
}
```

A C# programozási nyelv

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

A C# programozási nyelv

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

A C# programozási nyelv

Elemi és referencia osztályok

- Pl.:

```
struct Rational { … } // elemi osztály
…
Rational r = new Rational(10, 5);
Rational t = r; // r érték szerint másolódik
t.Denominator = 10; // itt r.Denominator == 5
```

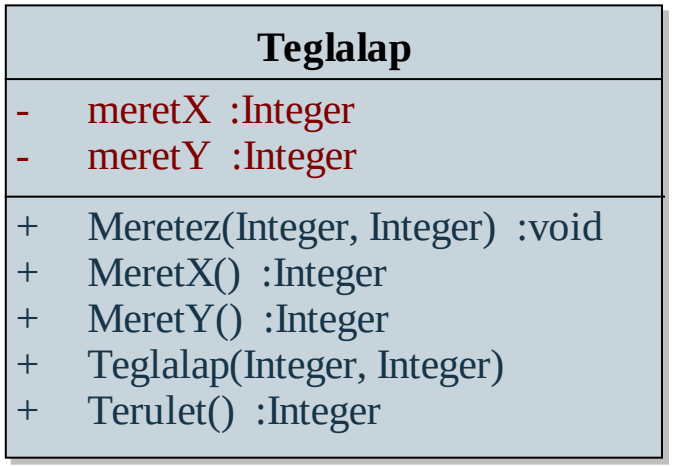
- Pl.:

```
class Rational { … } // referencia osztály
…
Rational r = new Rational(10, 5);
Rational t = r; // r cím szerint másolódik
t.Denominator = 10; // itt r.Denominator == 10
```

A C# programozási nyelv

Példa

- 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):



A C# programozási nyelv

Példa

- A téglalap osztály megvalósítása C++ nyelven:

```
class Teglalap{
private: // rejtett tagok
   int meretX;
   int meretY;
public: // látható tagok
   Teglalap(int x, int y) :
      meretx(x), meretY(y) { // konstruktor
   }
   // az alapértelmezett destruktort használjuk
   void Meretez(int x, int y) {
      meretX = x; meretY = y;
   }
   int MeretX() { return meretX; }
```

A C# programozási nyelv

Példa

```
    int MeretY() { return meretY; }
    int Terulet() { return meretX * meretY; }
}; // osztály vége


Teglalap t(10,20);
// az objektumpéldány létrehozása a konstruktor
// meghívásával
t.Meretez(20,20);
// módosító művelet meghívása az objektumra
cout << t.Terulet();
// lekérdező művelet meghívása az objektumra
```

A C# programozási nyelv

Példa

- A téglalap osztály megvalósítása C# nyelven:

```
class Teglalap{
   // a láthatóságokat tagonként adjuk meg
   private Int32 meretX;
   private Int32 meretY;

   public Teglalap(Int32 x, Int32 y)
   { // konstruktor
      meretX = x; meretY = y;
   }
   // az alapértelmezett destruktort használjuk
   public void Meretez(Int32 x, Int32 y){
      meretX = x; meretY = y;
   }
```

A C# programozási nyelv

Példa

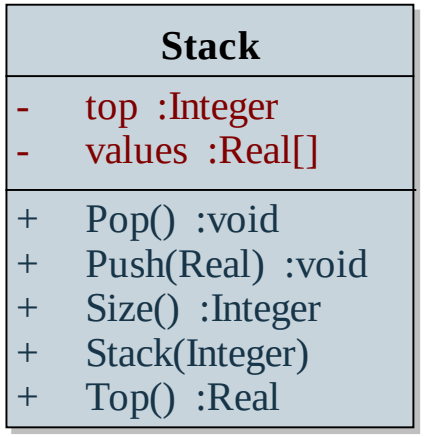
```
    // lekérdező műveletek, mint tulajdonságok:
    public Int32 MeretX { get { return meretX; }}
    public Int32 MeretY { get { return meretY; }}
    public Int32 Terulet {
       get { return meretX * meretY; }
    }
} // osztály vége

Teglalap t = new Teglalap(10,20);
// az objektumpéldány létrehozása
t.Meretez(20,20);
// módosító művelet meghívása az objektumra
Console.WriteLine(t.Terulet);
// lekérdező tulajdonság meghívása az objektumra
```

A C# programozási nyelv

Példa

- 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):



A C# programozási nyelv

Példa

- A verem osztály megvalósítása C++ nyelven:

```
class Stack {
private: // rejtett tagok
   vector<float> values;
   int top;
public: // látható tagok
   Stack(int maxSize){ // konstruktor
      values.resize(maxSize); // méret beállítás
      top = 0; // kezdetben üres
   }
   void Push(float v) {
      if (top < values.size()) {
         values[top] = v; top++; // méret növelés
      }
```

Példa

```
    }
    void Pop() {
       if (top > 0) top--; // méret csökkentése
    }
    float Top() {
       if (top > 0) return values[top];
       else exit(1); // hibajelzés
    }
    int Size() { return top; }
};

Stack* s = new Stack(10); // 10 méretű verem
s->Push(5.5); // elem behelyezése
…
```

A C# programozási nyelv

Példa

- A verem osztály megvalósítása C# nyelven:

```
class Stack {
   private Single[] values;
   private Int32 top;

   public Stack(Int32 maxSize){ // konstruktor
      values = new Single[maxSize];
      top = 0;
   }
   void Push(Single v) {
      if (top < values.Length) {
         values[top] = v; top++; // méret növelés
      }
   }
```

A C# programozási nyelv

Példa

```
    public void Pop() {
       if (top > 0) top--; // méret csökkentése
    }
    public Single Top { // lekérdező tulajdonság
       get {
          if (top > 0) return values[top];
          else throw new Exception(); // hibajelzés
       }
    }
    public Int32 Size { get { return top; } }
}

Stack s = new Stack(10); // 10 méretű verem
s.Push(5.5f); // elem behelyezése
…
```

A C# programozási nyelv

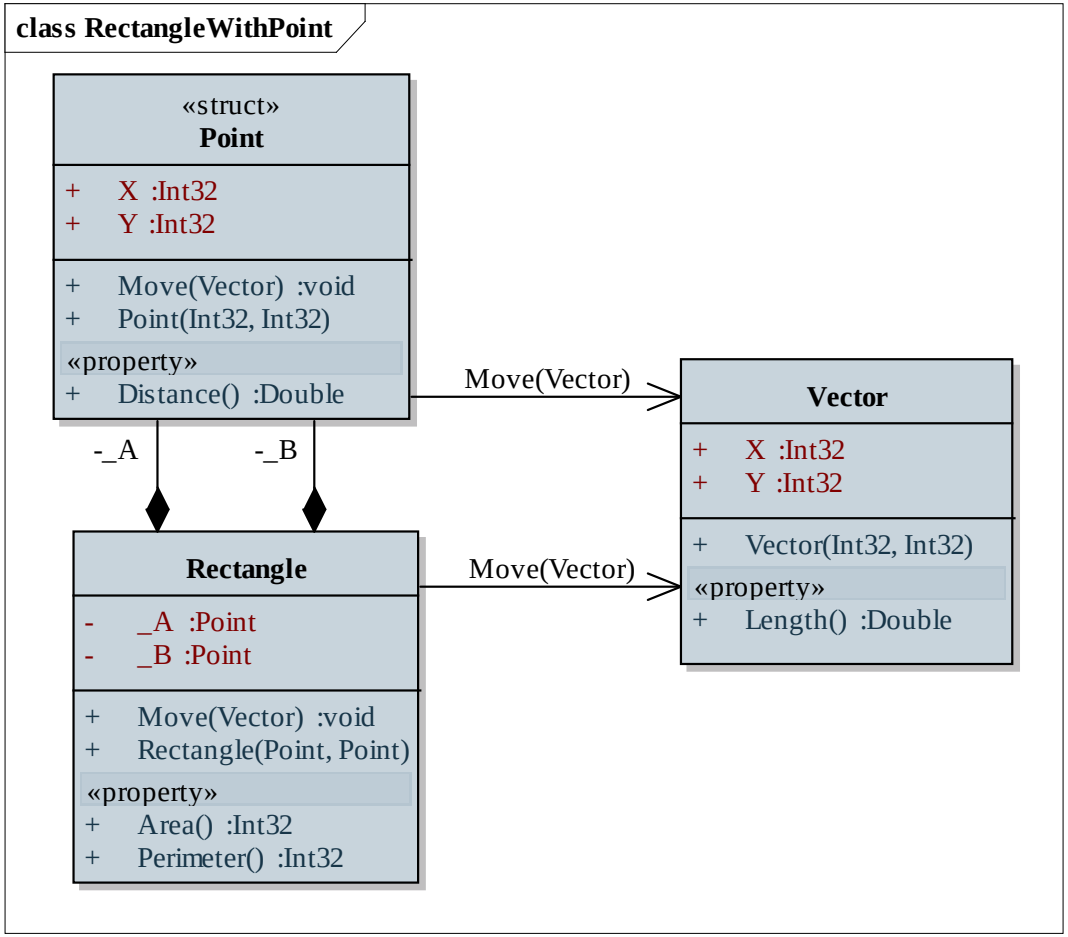
Példa

Feladat: 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

Példa

Tervezés:



A C# programozási nyelv

Példa

Megoldás:

```
struct Point { // érték szerint kezelt pont típus
   public Int32 X;
   public Int32 Y;

   …
}

class Vector { // cím szerint kezelt vektor típus
   public Int32 X;
   public Int32 Y;

   …
}
```

Példa

Megoldás:

```
class Rectangle {
   // cím szerint kezelt négyzet típus
   private Point _A;
   private Point _B;
   // a pontok érték szerint tárolódnak benne

   // két konstruktorműveletet definiálunk
   public Rectangle(Int32 aX, Int32 aY, Int32 bX,
                    Int32 bY) {
      // téglalap létrehozása koordináták alapján
      _A = new Point(aX, aY);
      _B = new Point(bX, bY);
   }
```

A C# programozási nyelv

Példa

Megoldás:

```
    …
    public void Move(Vector v) {
       _A.Move(v); _B.Move(v);
       // a vektor cím szerint adódik át
    }
    …
}

Rectangle rec = new Rectangle(10, 0, 20, 35);
  // téglalap létrehozása a 4 paraméteres
  // konstruktorral
rec.Move(new Vector(5, 5));
  // a létrehozott vektor cím szerint adódik át
```

A C# programozási nyelv

Generikus típusok

- Generikus programozásra futási időben feldolgozott sablon típusok (*generic*-ek) segítségével van lehetőség
  - osztály, metódus és delegált lehet sablonos, a sablon csak osztály lehet
  - a sablon fordításra kerül, és csak a futásidejű fordításkor helyettesítődik be a konkrét értékre
  - pl.:

```
struct Rational<T> {
    private T nom; // használható a T típusként
    …
    public Rational(T n, T d) { … }
    …
}
```

```
…
Rational<SByte> r1 = new Rational<SByte>(10,5);
Rational<Int64> r2 = new Rational<Int64>(10,5);
// különböző értékkészletű racionálisok
```

- 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
  - korlátoznál legfeljebb egy osztály, viszont tetszőleges számú interfész megadható

A C# programozási nyelv

Generikus típusok

- Pl.:

```
class Rational<T> where T : struct, IComparable,
          IFormattable, IConvertible { …
   // T elemi osztály, amire használható a fenti
   // interfészek összes művelete
}
…
Rational<Int64> r1 = new Rational<Int64>(10, 5);
// az Int64 megvalósítja az összes interfészt
Rational<String> r2 = new Rational<String>("10",
                                           "5");
// fordítási hiba, mivel a String nem valósítja
// meg valamennyi interfészt
```

A C# programozási nyelv

Tömbök

- A tömbök osztályként vannak megvalósítva (**`System.Array`**), de egyszerűsített szintaxissal kezelhetőek, pl.:

```
Int32[] myArray = new Int32[10]; // létrehozás
myArray[0] = 1; // első elem beállítása
```

  - referencia szerint kezeltek, méretnek változó is megadható, az értékek inicializálhatóak
  - akár több dimenziósak is lehetnek, pl.:

```
Int32[,] myMatrix = new Int32[10,5]; // mátrix
myMatrix[0, 0] = 1; // első sor első eleme
```

- Fontosabb műveletei:
  - hossz lekérdezés (**`Length`**, **`LongLength`**, **`GetLength`**)
  - dimenziószám lekérdezése (**`Rank`**)

A C# programozási nyelv

Tömbök

- Statikus műveletként számtalan lehetőségünk van, pl.:
  - másolás (**`Copy`**), átméretezés (**`Resize`**)
  - rendezés (**`Sort`**), fordítás (**`Reverse`**)
  - lineáris keresés (**`Find`**, **`IndexOf`**, **`LastIndexOf`**), bináris keresés (**`Binary Search`**)
- Lehetőség van a közvetlen inicializálásra is, pl.:

```
Int32[] myArray = new Int32[] {1, 2, 3, 4};
    // a tömb 4 hosszú lesz
```

- A tömböknél (és más gyűjteményeknél) alkalmazott indexelő művelet természetesen megvalósítható saját típusokra is (paraméteres tulajdonságként)

A C# programozási nyelv

Gyűjtemények

- A gyűjtemények a **`System.Collections`** névtérben találhatóak, a legtöbb gyűjteménynek van általános és sablonos változata is, pl.:
  - dinamikus tömbök: **`ArrayList`**, **`List<T>`**, **`SortedList`**, **`SortedList<Key, Value>`**
  - láncolt listák: **`LinkedList<T>`**
  - verem: **`Stack`**, **`Stack<T>`**
  - sor: **`Queue`**, **`Queue<T>`**
  - asszociatív tömb: **`Hashtable`**, **`Dictionary<Key, Value>`**, **`SortedDictionary<Key, Value>`**
  - halmaz: **`HashSet<T>`**, **`SortedSet<T>`**

A C# programozási nyelv

Gyűjtemények

- A nem sablonos gyűjteményekbe bármilyen elemeket helyezhetünk
- A dinamikus tömbök indexelhetőek, és változtatható a méretük (bárhova beszúrhatunk, bárhonnan törölhetünk), pl.:

```
List<Int32> intList = new List<Int32>();
   // üres tömb létrehozása
intList.Add(1); … // elemek hozzáadása
intList.Insert(0, 100); // beszúrás az elejére
…
intList.Remove(100); // elem törlése
for (Int32 i = 0; i < intList.Count; i++)
   Console.WriteLine(intList[i]);
   // lekérdezés
intList.Clear(); // kiürítés
```

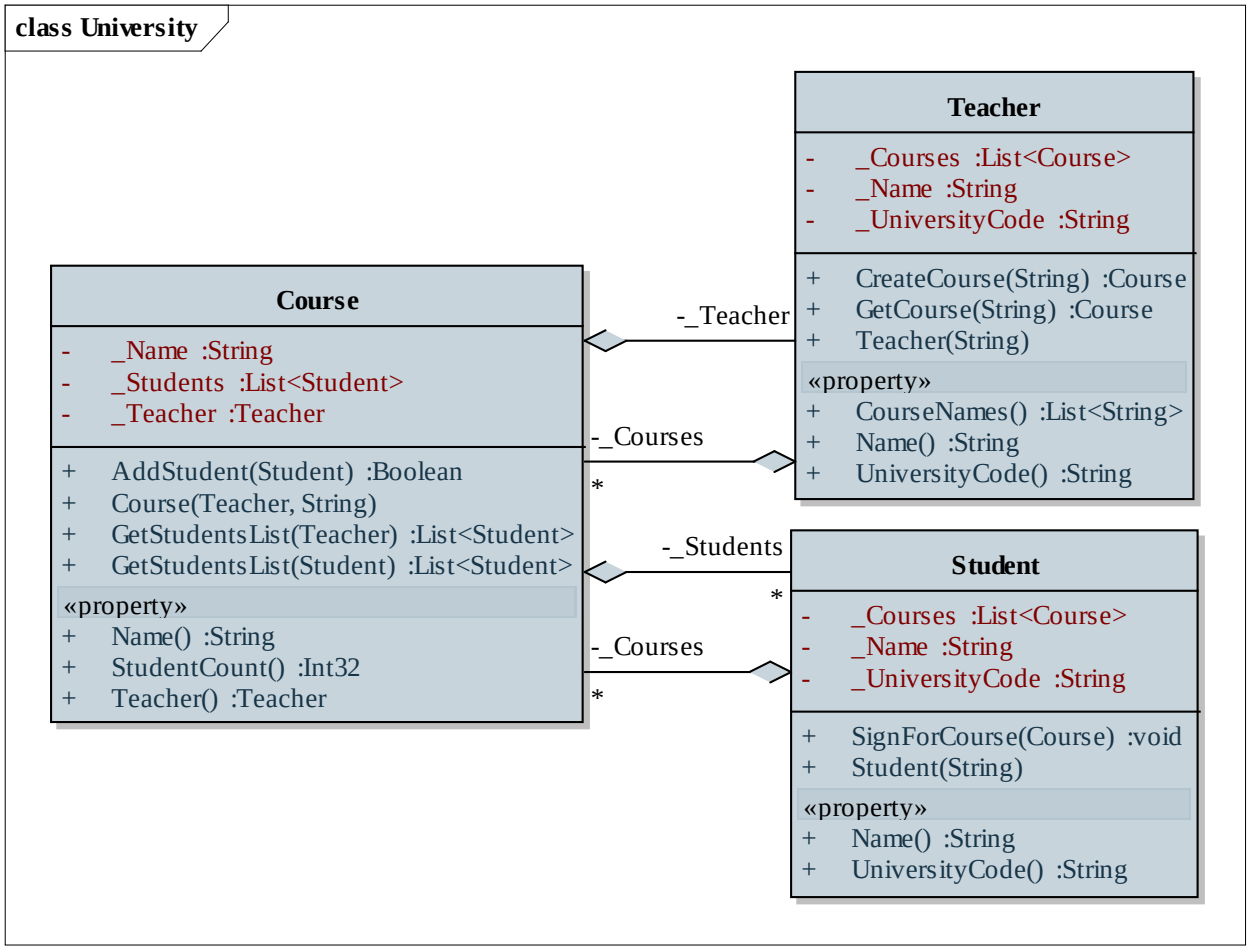
A C# programozási nyelv

Példa

Feladat: 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

Tervezés:


class University
Teacher
- _Courses :List<Course>
- _Name :String
- _UniversityCode :String
+ CreateCourse(String) :Course
+ GetCourse(String) :Course
+ Teacher(String)
«property»
+ CourseNames() :List<String>
+ Name() :String
+ UniversityCode() :String
Course
- _Name :String
- _Students :List<Student>
- _Teacher :Teacher
+ AddStudent(Student) :Boolean
+ Course(Teacher, String)
+ GetStudentsList(Teacher) :List<Student>
+ GetStudentsList(Student) :List<Student>
«property»
+ Name() :String
+ StudentCount() :Int32
+ Teacher() :Teacher
-_Teacher
-_Courses
*
-_Students
*
-_Courses
*
Student
- _Courses :List<Course>
- _Name :String
- _UniversityCode :String
+ SignForCourse(Course) :void
+ Student(String)
«property»
+ Name() :String
+ UniversityCode() :String

A C# programozási nyelv

Példa

Megoldás:

```
class Course { // egyetemi kurzus típusa
   …
   public List<Student> GetStudentsList(Teacher
                                        teacher) {
      // hallgatók listájának lekérdezése
      if (teacher == _Teacher)
          // ha a kurzus oktatója
          return _Students;
      // akkor lekérdezheti a hallgatók listáját,
      // és módosíthatja is azt
   else return new List<Student>();
      // különben egy üres listát kap
}
```

A C# programozási nyelv

Példa

Megoldás:

```
public List<Student> GetStudentsList(Student
                                     student) {
   if (_Students.Contains(student))
      // ha hallgatója a kurzusnak
      return new List<Student>(_Students);
      // akkor megkapja a lista másolatát, így
      // nem tudja módosítani az eredetit
   else
      return new List<Student>();
      // különben egy üres listát kap
}
}
```

Példa

Megoldás:

```
class Teacher { // egyetemi tanár típusa
   …
   public List<String> CourseNames {
      // a kurzusok neveinek lekérdezése
      get {
         List<String> courseNames =
            new List<String>();
         foreach (Course course in _Courses)
            courseNames.Add(course.Name);
            // új lista a kurzusnevekből
         return courseNames;
      }
   …
```

Példa

Megoldás:

```
    public Course CreateCourse(String name){
       Course c = new Course(this, name);
       // nyílt rekurzió használata
       _Courses.Add(c); return c;
    }
    public Course GetCourse(String name) {
       foreach (Course course in _Courses)
          if (course.Name == name)
             // ha sikerült megtalálnunk
             return course; // akkor visszaadjuk
          return null; // különben nullát adunk
    }
 }
```

A C# programozási nyelv

Osztályszintű tagok

- Lehetőségünk van *statikus osztályok*, *mezők, tulajdonságok és műveletek* létrehozására a **`static`** kulcsszó használatával
  - az osztályszintű tagok nem látják az objektumszintű tagokat, és nem használhatnak nyílt rekurziót
  - az osztályszintű tagokat csak az osztálynév megadásával érhetjük el
  - lehetőségünk van *osztályszintű konstruktor* megadására, ennek nem lehet láthatósága, illetve paraméterezése és mindig automatikusan hívódik meg, amely csak egyszer a program futása során
  - a teljes osztály is megjelölhető statikusnak, akkor csak statikus tagokat tartalmazhat, és nem példányosítható

A C# programozási nyelv

Osztályszintű tagok

- Pl.:

```
static class NumClass { // statikus osztály
   private static Int32 nr = 10;
      // statikus mező 10 kezdőértékkel
   public static Int32 Nr{ get { return nr; } }
      // statikus tulajdonság
   public static void Increase() { nr++; }
      // statikus metódus
}

Console.WriteLine(NumClass.Number) // eredmény: 10
NumClass.Increase();
Console.WriteLine(NumClass.Number) // eredmény: 11
```

A C# programozási nyelv

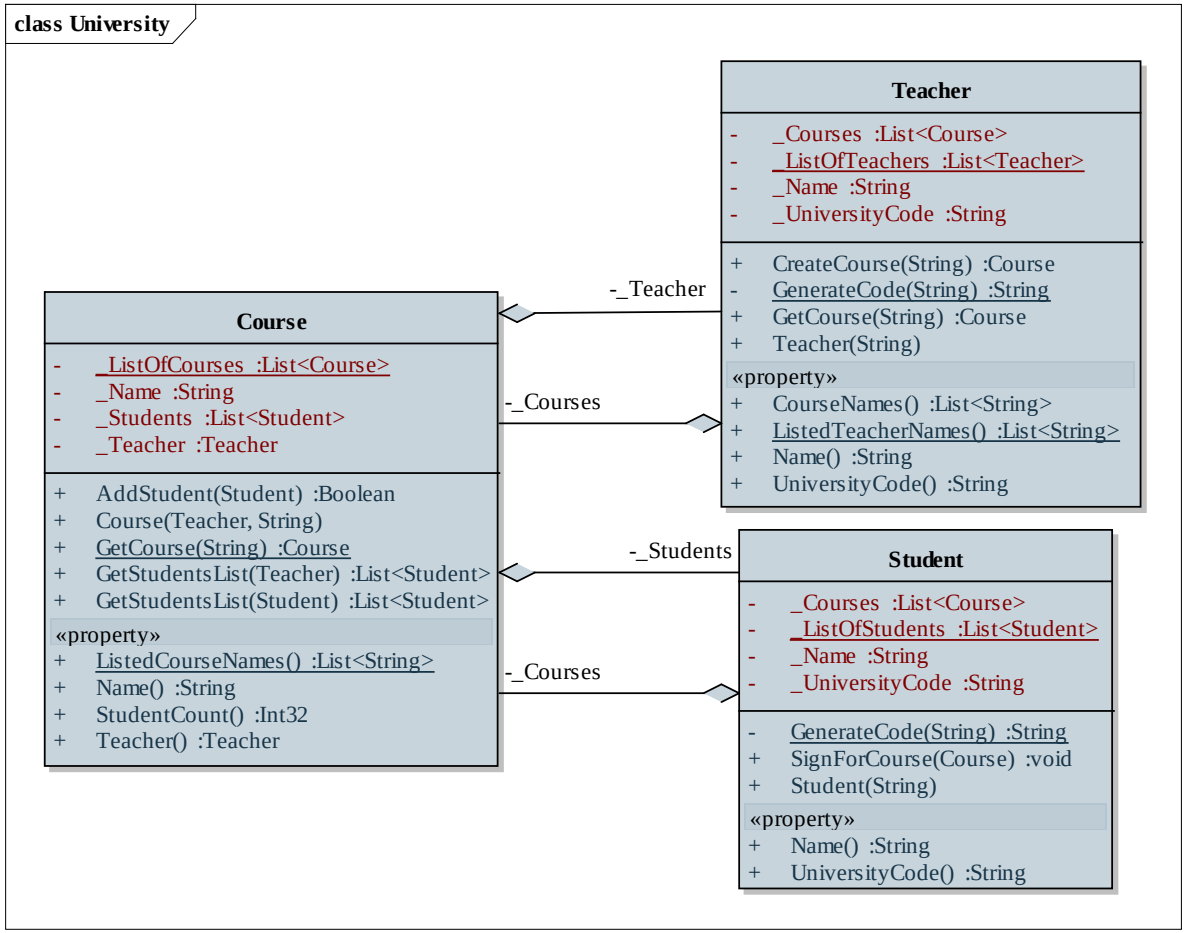
Példa

Feladat: Az egyetemi oktatók és hallgatók esetén hasznos lenne az azonosítót úgy generálni, hogy két azonos ne forduljon elő a rendszerben.

- ehhez a hallgatónak, és az oktatónak is ismernie kell az eddig kiadott azonosítókat, vagyis lényegében az eddig létrehozott objektumokat, ezt megoldhatjuk úgy, hogy egy osztályszintű mezőbe listázzuk a létrehozott hallgatókat és oktatókat
- hasonló módon legyen mód a rendszerben tárolt kurzusok és oktatók nevének lekérdezésére, valamint egy konkrét kurzus lekérdezésére objektum példányosítása nélkül, ezt statikus metódusokon keresztül érjük el

Példa

Tervezés:



A C# programozási nyelv

Példa

Megoldás:

```
class Student {
   public Student(String name){
      …
      _UniversityCode = GenerateCode(name);
      _ListOfStudents.Add(this);
   }
   …
   private static List<Student> _ListOfStudents
           = new List<Student>();
   private static String GenerateCode(String name)
   {  // kódgenerálás, amihez ismerni kell a többi
      // oktató kódját is
      …
```

Példa

Megoldás:

```
class Course {
   …
   private static List<Course> _ListOfCourses
           = new List<Course>();
   // statikus lista, az összes kurzusnak

   public static List<String> ListedCourseNames {
      // statikus tulajdonság, a kurzusnevek
      // lekérdezésére
      get { … }
      // elérheti a statikus mezőket
   }
   …
```

Példa

Megoldás:

```
static void Main() { // maga a főprogram is
   // statikus, mivel csak egy lehet belőle
   …
   Console.WriteLine("Kurzusok:");
   foreach (String name in
            Course.ListedCourseNames) {
      // statikus tulajdonság lekérdezése
      Course course = Course.GetCourse(name);
      // statikus metódus meghívása
      Console.WriteLine(course.Name +
          ", létszám: " + course.StudentCount);
   }
}
```

A C# programozási nyelv

Operátorok megvalósítása

- Az operátorokat osztályszintű metódusként kell definiálnunk az **`operator`** kulcsszó használatával:

```
<láthatóság> static
<típus> operator <jel> (<paraméterek>)
{
   <törzs>
}
```

  - a paraméterek száma az operandusok száma, és legalább egyiknek saját típusúnak kell lennie
  - a típus a visszatérési érték, vagyis az eredmény típusa
  - túlterheléssel több művelet is rendelhető az osztályon belül egy operátorhoz

A C# programozási nyelv

Operátorok megvalósítása

- Pl.:

```
class Rational {
    …
    public static Rational operator +(Rational a,
       Rational b){ // összeadás operátor
       …
    }
}

Rational r1 = new Rational(3);
Rational r2 = new Rational(5);
… r1 + r2 … // a + művelet alkalmazható
```

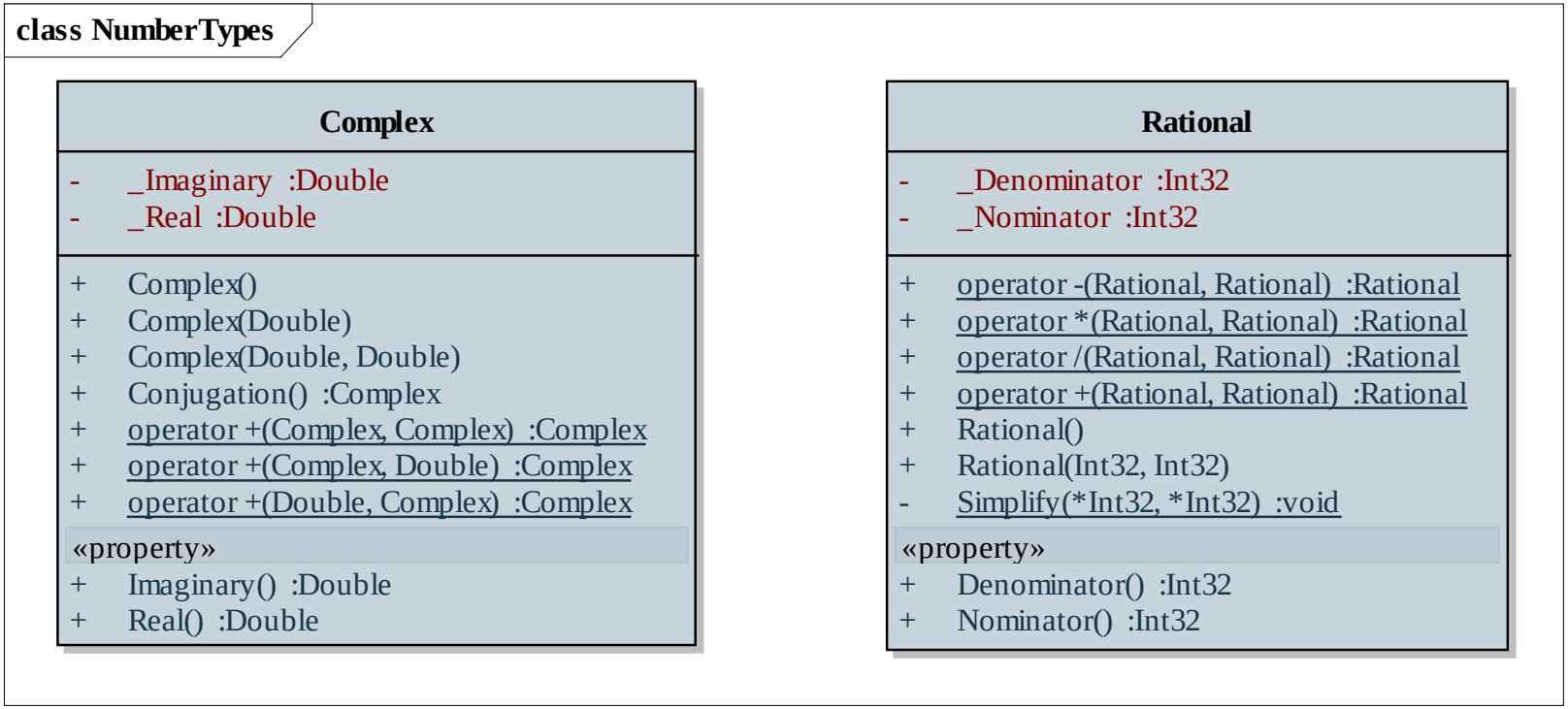
Példa

Feladat: Készítsük el a komplex számok, valamint a racionális számok osztályait operátorok segítségével.

- komplex számok esetén értelmezzük az összeadás (**+**) műveletét komplex-komplex, komplex-valós, valós-komplex értékekkel, valamint komplex szám konjugálását
- racionális számok esetén értelmezzük a négy alapműveletet (**+**, **-**, *****, **/**) racionális számok között, és ügyeljük arra, hogy a racionális szám nevezője nem lehet 0
- a racionális számot tartsuk mindig a legegyszerűbb formában, amihez valósítsunk egy egyszerűsítő műveletet (Euklideszi algoritmussal), amely paraméterben kapott nevezőt és számlálót adja vissza egyszerűsítve

Példa

Tervezés:



Példa

Megoldás:

```
class Complex { // komplex szám osztálya
    …
    public static
    Complex operator +(Complex a, Complex b) { … }
    // komplex bal és jobbérték

    public static
    Complex operator +(Complex a, Double b) { … }
    // komplex bal-, valós jobbérték
    …
}
```

Példa

Megoldás:

```
class Rational { // racionális szám osztálya
   …
   public static Rational
   operator +(Rational first, Rational second){
      …
      Simplify(ref result._Nominator, ref
          result._Denominator); // egyszerűsítés
      return result;
   }
   static private void Simplify(ref Int32 first,
                                ref Int32 second)
   { … } // cím szerinti paraméterátadás
   …
```

A C# programozási nyelv

Öröklődés

- A nyelv alapvető építőeleme az öröklődés és a polimorfizmus
  - az implementációban az osztályoknál jelölnünk kell, mely ősosztályból származtatunk:

```
class <osztálynév> : <ősosztály> {
    <további tagok>
}
```

  - a .NET keretszerben az osztályok egy teljes származtatási hierarchiában vannak
  - minden osztály automatikus őse az **`Object`**, így megkapja annak műveleteit (pl.: **`Equals(…)`**, **`ToString()`**), tehát csak leszármazott osztályt hozhatunk létre
  - az ős tagjaira a **`base`** kulcsszón keresztül hivatkozhatunk

A C# programozási nyelv

Öröklődés

- az öröklődésnek nincs külön láthatósága, a tagok láthatósággal együtt kerülnek a leszármazotthoz
- csak egyszeres öröklődés van, a többszörös öröklődés interfészekkel váltható ki

- A konstruktor automatikusan öröklődik
  - a paraméteres nélküli konstruktor automatikusan meghívódik a leszármazottban
  - lehetőségünk van az ős konstruktorának explicit meghívására is **`<konstruktor>(<paraméterek>) : base(<átadott paraméterek>)`** formában
- A destruktor automatikusan öröklődik és hívódik meg

A C# programozási nyelv

Öröklődés

- Pl.:

```
class BaseCl /* : Object */ { // ősosztály
   public Int32 Value;
   public BaseCl(Int32 v) { value = v; }
}
…
class DerivedCl : BaseCl { // leszármazott
   public BaseCl(Int32 v) : base(v) { }
      // ős konstruktorának meghívása
}
…
BaseCl b = new DerivedCl(1); // polimorfizmus
Object[] oArray = new Object[] { 1, "Hello World",
    new BaseCl(1), new DerivedCl(2) };
```

A C# programozási nyelv

Öröklődés

- Polimorfizmus során lehetőségünk van a típusazonosításra (**`is`**), valamint a biztonságos típuskonverzióra (**`as`**)
  - hibás típus esetén **`null`** értéket kapunk
  - explicit típuskonverzió is alkalmazható, ám ez kivételt válthat ki
  - pl.:

```
foreach(BaseCl item in oArray)
   if (item is BaseCl)
      // csak a leszármazott példányokra
      Console.WriteLine(
         (listItem as DerivedCl).Value);
         // konverzió után lekérdezhető az érték
```

A C# programozási nyelv

Öröklődés

- Öröklődés során a műveletek és tulajdonságok felüldefiniálhatóak, illetve elrejthetőek
  - elrejteni bármely műveletet, tulajdonságot lehet, amennyiben az új művelet megkapja a **`new`** kulcsszót (polimorfizmusnál nem érvényesül)
  - felüldefiniálni csak a virtuális (**`virtual`**) és absztrakt (**`abstract`**) műveleteket, tulajdonságokat lehet
  - absztrakt metódusok törzs nélküliek, absztrakt tulajdonságoknál csak azt kell jelezni, hogy lekérdezésre, vagy értékadásra szolgálnak-e
  - absztrakt tagot tartalmazó osztály is absztrakt (szintén jelölnünk kell)

- a felüldefiniálást is jelölnünk kell az **override** kulcsszóval
- a felüldefiniáló művelet meghívhatja az eredetit a **base** kulcsszóval

- Pl.:

```
class BaseCl { // ősosztály
   public void StandardMethod() {
      // lezárt (nem felüldefiniálható) művelet
      Console.WriteLine("BaseStandard");
   }
   public virtual void VirtualMethod() {
      // virtuális (felüldefiniálható) művelet
      Console.WriteLine("BaseVirtual");
   }
}
```

A C# programozási nyelv

Öröklődés

- Pl.:

```
class DerivedCl : BaseCl {
    public new void StandardMethod() {
        // művelet elrejtés
        Console.WriteLine("DerivedStandard");
    }
    public override void VirtualMethod() {
        // művelet felüldefiniálás
        base.VirtualMethod();
            // a felüldefiniált művelet meghívása
        Console.WriteLine("DerivedVirtual");
    }
}
```

A C# programozási nyelv

Öröklődés

- Pl.:

```
DerivedCl dc = new DerivedCl();
dc.StandardMethod(); // eredmény: DerivedStandard
dc.VirtualMethod();
   // eredmény:
   // BaseVirtual
   // DerivedVirtual

…
BaseCl bc = new DerivedCl();
bc.StandardMethod(); // eredmény: BaseStandard
bc.VirtualMethod();
   // eredmény:
   // BaseVirtual
   // DerivedVirtual
```

A C# programozási nyelv

Öröklődés

- Pl.:

```
abstract class BaseCl { // absztrakt ősosztály
   public abstract Int32 Value { get; }
      // absztrakt lekérdező tulajdonság,
      // felüldefiniálandó
   public abstract void AbstractMethod();
      // absztrakt metódus, felüldefiniálható
   public virtual void VirtualMethod() {
      Console.WriteLine(Value);
   }
}
…
BaseCl b = new BaseCl();
// hiba: absztrakt osztály nem példányosítható
```

A C# programozási nyelv

Öröklődés

```
class DerivedCl : BaseCl {
   public override Int32 Value {
      get { return 1; }
   } // tulajdonság felüldefiniálás
   public sealed override void AbstractMethod() {
      VirtualMethod();
      Console.WriteLine(2 * Value);
   }
}
…
BaseCl bc = new DerivedCl();
bc.AbstractMethod();
// eredménye:
// 1
// 2
```

A C# programozási nyelv

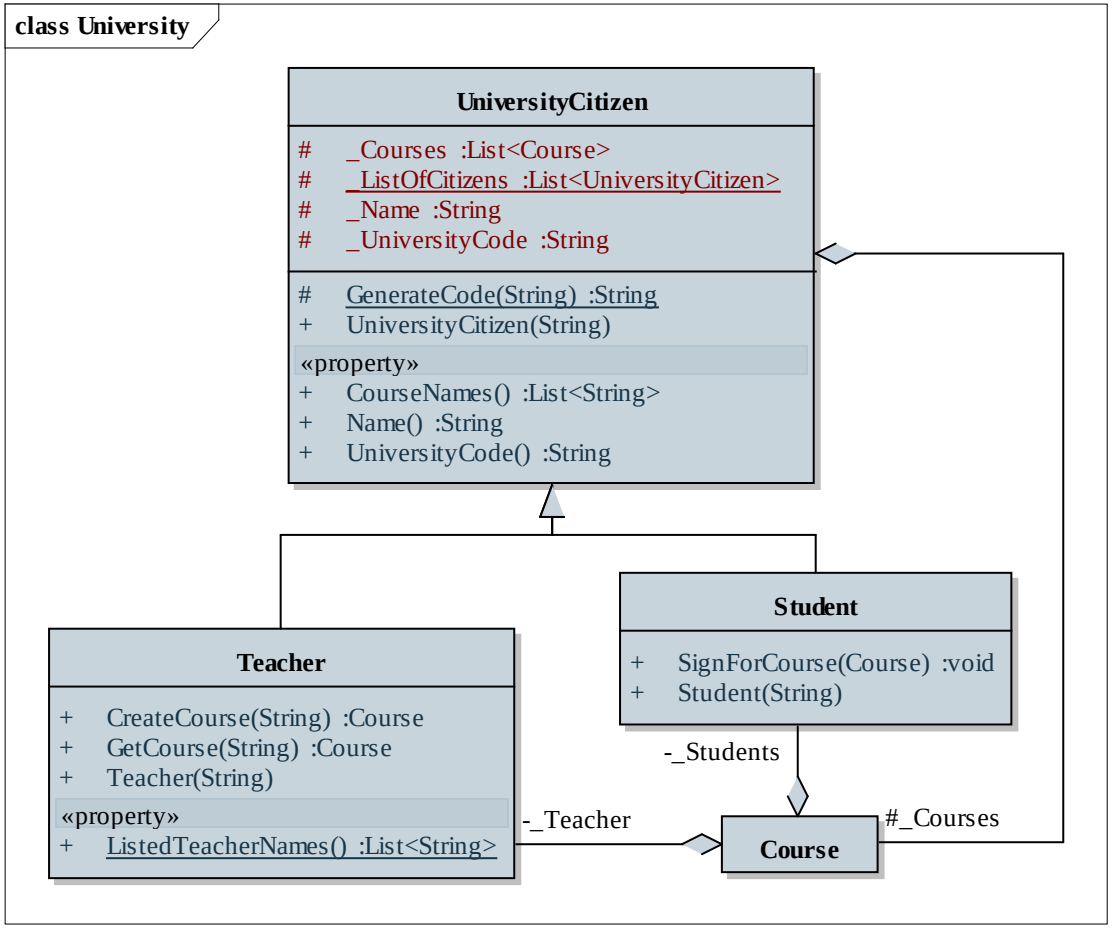
Példa

Feladat: Az egyetemi oktatót és hallgatót általánosíthatjuk egy egyetemi polgár osztályba.

- az egyetemi polgár (**`UniversityCitizen`**) tartalmazhatja a nevet, azonosítót, illetve a kurzusok listáját (védett láthatósággal), valamint az ehhez tartozó lekérdező tulajdonságokat
- ez egy absztrakt osztály lesz, ennek leszármazottai a hallgató és az oktató
- az egyetemi polgárba helyezzük a statikus listát, és az azonosító generálás védett láthatósággal, a polgár konstruktorába helyezzük az alap tevékenységeket

Példa

Tervezés:



Példa

Megoldás:

```
abstract class UniversityCitizen{
   public UniversityCitizen(String name){
      …
      _ListOfCitizens.Add(this); // aktuálisan
      // egy hallgató, vagy oktató példány lesz
   }
   …
}
class Teacher : UniversityCitizen{
   public Teacher(String name) : base(name)
      // ős konstruktorának meghívása
   { /* egyéb tevékenység nem kell * / }
}
```

A C# programozási nyelv

Példa

```
static void Main(string[] args){
   List<UniversityCitizen> cits =
        new List<UniversityCitizen>();
   cits.Add(new Student("Huba Hugó"));
   cits.Add(new Teacher("Kis Ferenc"));
   (cits[1] as Teacher).CreateCourse("Lazulás");
   // típusmegfeleltetés
   …
   foreach (UniversityCitizen cit in citizens){
      …
      if (citizen is Student) // típusazonosítás
         Console.WriteLine(", hallgató");
   }
   …
```

Interfészek

- A többszörös öröklődés számos hibához vezethet, ezért tiltott
- Feloldására lehetőségünk van olyan osztályokat definiálni, amelyek csak publikus absztrakt tagokat (tulajdonságokat és metódusokat) tartalmaznak, ezeket nevezzünk *interfészek*nek
- Interfészeket az **`interface`** kulcsszóval kell jelölnünk, és azt mondjuk, hogy az *osztály megvalósítja az interfészt*
  - az interfészben minden publikus, és absztrakt, ezért nem is írjuk ki a kulcsszavakat, felüldefiniáláskor sem
  - az interfészek elnevezését általában I-vel kezdjük
  - egy osztály tetszőleges sok interfészt valósíthat meg, az interfész megvalósíthat más interfészeket is

A C# programozási nyelv

Interfészek

- Interfészeket nem csak referencia szerinti, hanem érték szerinti osztályok is megvalósíthatnak, pl.:

```
interface IValuePrinter { // interfész
   // minden művelet public abstract
   void PrintIntValue();
   void PrintFloatValue();
}

struct AnyStruct : IValuePrinter {
   // interfészt megvalósító osztály
   public void PrintIntValue() {…}
   public void PrintFloatValue() {…}
   // definiálni kell minden interfész műveletet
}
```

A C# programozási nyelv

Interfészek

- Pl.:

```
interface IAllValuePrinter{ // újabb interfész
   void PrintAllValues();
}

abstract class BaseClass { // absztrakt osztály
   private Int32 _IntValue;

   public BaseClass() { _IntValue = 1; }
   public void PrintIntValue(){
      Console.WriteLine(_IntValue);
   }
}
```

A C# programozási nyelv

Interfészek

```
class DerivedClass : BaseClass,
                     IValuePrinter,
                     IAllValuePrinter {
   // öröklődés és több interfész megvalósítása
   // egyszerre
   // a PrintIntValue művelet már megvan az
   // öröklődésnek köszönhetően, csak a többi kell
   …
   // interfész megvalósítás:
   public void PrintFloatValue() {…}
   public void PrintAllValues() {…}
}
```

A C# programozási nyelv

Példa

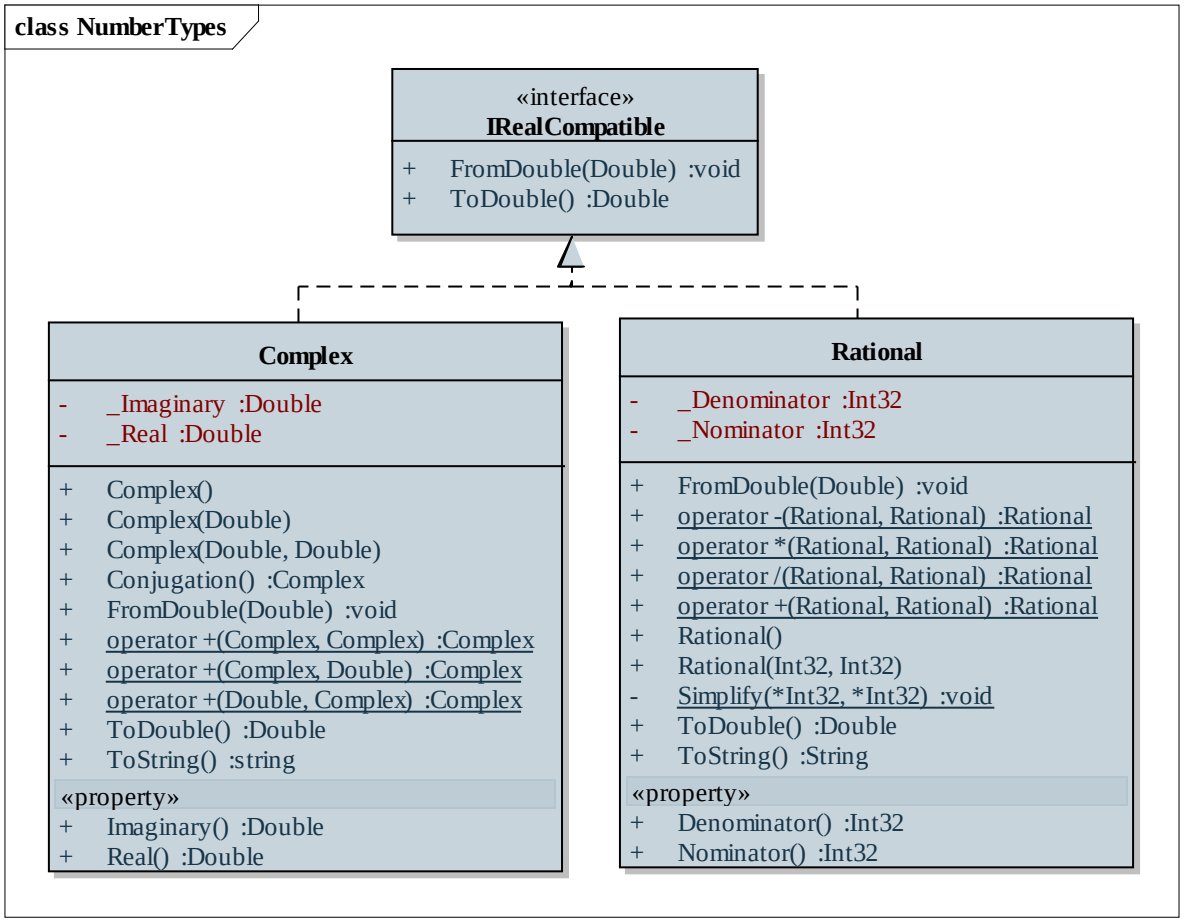
Feladat: A racionális, illetve komplex számok kompatibilisek a valós számokkal, ezért célszerű lenne egy olyan felületet adni nekik, ami a konverziót mindkét irányba elvégzi.

- létrehozunk egy interfészt, ami a valóssá alakítás, illetve valósról átalakítás metódusait tartalmazza
- a két osztályban ezt megvalósítjuk, így egy közös adatszerkezetben tárolva is működni fog a művelet az aktuális számra
- definiáljuk felül az **`Object`**-ből örökölt szöveggé alakítást is, hogy megfelelően tudjuk szöveges formában kiírni az értékeket

A C# programozási nyelv

Példa

Tervezés:



Példa

Megoldás:

```
interface IRealCompatible { // interfész
   Double ToDouble();
   void FromDouble(Double val);
} // csak absztrakt műveleteket írunk

class Rational : IRealCompatible {
   …
   // felüldefiniálások:
   public Double ToDouble() {…}
   public void FromDouble(Double val) {…}
   public Strint override ToString() {…}
}
```

A C# programozási nyelv

Kivételkezelés

- A .NET keretrendszerben minden hiba kivételként jelenik meg
- A kivétel általános osztálya az **`Exception`**, csak ennek példánya, vagy leszármazottja dobható
- Kivételt kezelni egy kivételkezelő (**`try-catch-finally`**) szakasszal tudunk:

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

A C# programozási nyelv

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 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 a mindenképpen (el nem fogott kivétel esetén is) lefuttatandó utasításokat tartalmazza (nem kötelező)

A C# programozási nyelv

Kivételkezelés

- Lehetőségünk van különböző típusú kivételek elfogására is, amennyiben több **`catch`** ágat készítünk a szakaszhoz
  - az ágak feldolgozása sorrendben történik, csak egy fut le
  - amennyiben biztosan el akarunk kapni bármilyen kivételt, kapjuk el az általános **`Exception`** típust is
  - pl.:

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

- A kivételek üzenettel rendelkeznek, amelyet a kivétel **`Message`** tulajdonságán keresztül kérhetünk le
- Kivételt kiváltani a **`throw`** utasítással tudunk:
  `throw new <kivétel típusa>(<kivétel paraméterei>);`
  - csak az **`Exception`**, vagy leszármazottjának (pl. **`ArgumentException`**, **`NullReferenceException`**, **`IndexOutOfRangeException`**) példánya dobható (mi is származtathatunk újat)
  - egy **`catch`** ágban, ha továbbdobnánk az elfogott kivételt, elég a **`throw;`** utasítás

A C# programozási nyelv

Előfordítási direktívák

- A nyelv tartalmaz előfordítási direktívákat, amelyek előzetesen kerülnek feldolgozásra, így lehetőséget adnak bizonyos kódsorok feltételes fordítására, hibajelzésre, környezetfüggő beállítások lekérdezésére, pl. **`#if`**, **`#define`**, **`#error`**, **`#line`**
- Mivel nem választható szét a deklaráció a definíciótól, a kód tagolását a *régió*k segítik elő, amelyek tetszőleges kódblokkokat foghatnak közre:

```
#region <név>
…
#endregion
```

  - nem befolyásolják a kódot, csupán a fejlesztőkörnyezetben érhetőek el

A C# programozási nyelv

Megjegyzések

- Az egyszerű *megjegyzések* a fordításkor törlődnek
  - sor végéig tartó: **`// megjegyzés`**
  - két tetszőleges pont között: **`/* megjegyzés */`**
- A *dokumentációs megjegyzés*ek fordításra kerülnek, és utólag előhívhatóak a lefordított tartalomból
  - osztályok és tagjaik deklarációjánál használhatjuk őket
  - céljuk az automatikus dokumentálás elősegítése és a fejlesztőkörnyezetben azonnal segítség megjelenítése
  - a **`///`** jeltől a sor végéig tart, belül XML szintaxisú blokkok adhatóak meg, amelyek meghatározzák az információ jelentését (pl. **`summary`**, **`remarks`**, **`exceptions`**)

A C# programozási nyelv

Megjegyzések

- pl.:

```
/// <summary>
/// Racionális szám típusa.
/// </summary>
/// <remarks>Két egész szám hányadosa.</remarks>
struct Rational {
    …
    /// <summary>
    /// Racionális szám példányosítása.
    /// </summary>
    /// <param name="n">Számláló.</param>
    /// <param name="d">Nevező.</param>
    public Rational(Int32 n, Int32 d) { … }
    …
```

A C# programozási nyelv

Fájlkezelés

- Az adatfolyamok kezelése egységes formátumban adott, így azonos módon kezelhetőek fájlok, hálózati adatforrások, memória, adatbázisok, stb.
  - az adatfolyamok ősosztálya a **`Stream`**, amely binárisan írható/olvasható
- Szöveges adatfolyamok írását, olvasását a **`StreamReader`** és **`StreamWriter`** típusok biztosítják
  - létrehozáskor megadható az adatfolyam, vagy közvetlenül a fájlnév
  - csak karakterenként (**`Read`**), vagy soronként (**`ReadLine`**) tudunk olvasni, így konvertálnunk kell

- amennyiben a műveletek során hiba keletkezik, **`IOException`**-t kapunk

- Pl.:

```
StreamReader reader = new StreamReader(„in.txt”);
   // fájl megnyitása
while (!reader.EndOfStream) // amíg nincs vége
{
   Int32 value = Int32.Parse(reader.ReadLine());
      // sorok olvasása, majd konvertálás
   …
}
reader.Close(); // bezárás
```

A C# programozási nyelv

Erőforrások felszabadítása

- A referencia szerinti változók törlését a szemétgyűjtő felügyeli
  - adott algoritmussal adott időközönként pásztázza a memóriát, törli a felszabadult objektumokat
  - sok, erőforrás-igényes objektum példányosítása esetén azonban nem mindig reagál időben, így nő a memóriahasználat
  - a **`GC`** osztály segítségével lehetőségünk manuális futtatásra
- A manuális törlésre (destruktor futtatásra) nincs lehetőségünk felügyelt blokkban, de erőforrások felszabadítására igen, amennyiben az osztály megvalósítja az **`IDisposable`** interfészt, és benne a **`Dispose()`** metódust

A C# programozási nyelv

Erőforrások felszabadítása

- Emellett a C# nyelv tartalmaz egy olyan blokk-kezelési technikát, amely garantálja a **`Dispose()`** automatikus futtatását:

```
using (<objektum példányosítása>){
   <objektum használata>
} // itt automatikusan meghívódik a Dispose()
```

- Pl.:

```
using (StreamReader reader = …){
   // a StreamReader is IDisposable
   …
}
// itt biztosan bezáródik a fájl, és
// felszabadulnak az erőforrások
```

A C# programozási nyelv

Példa

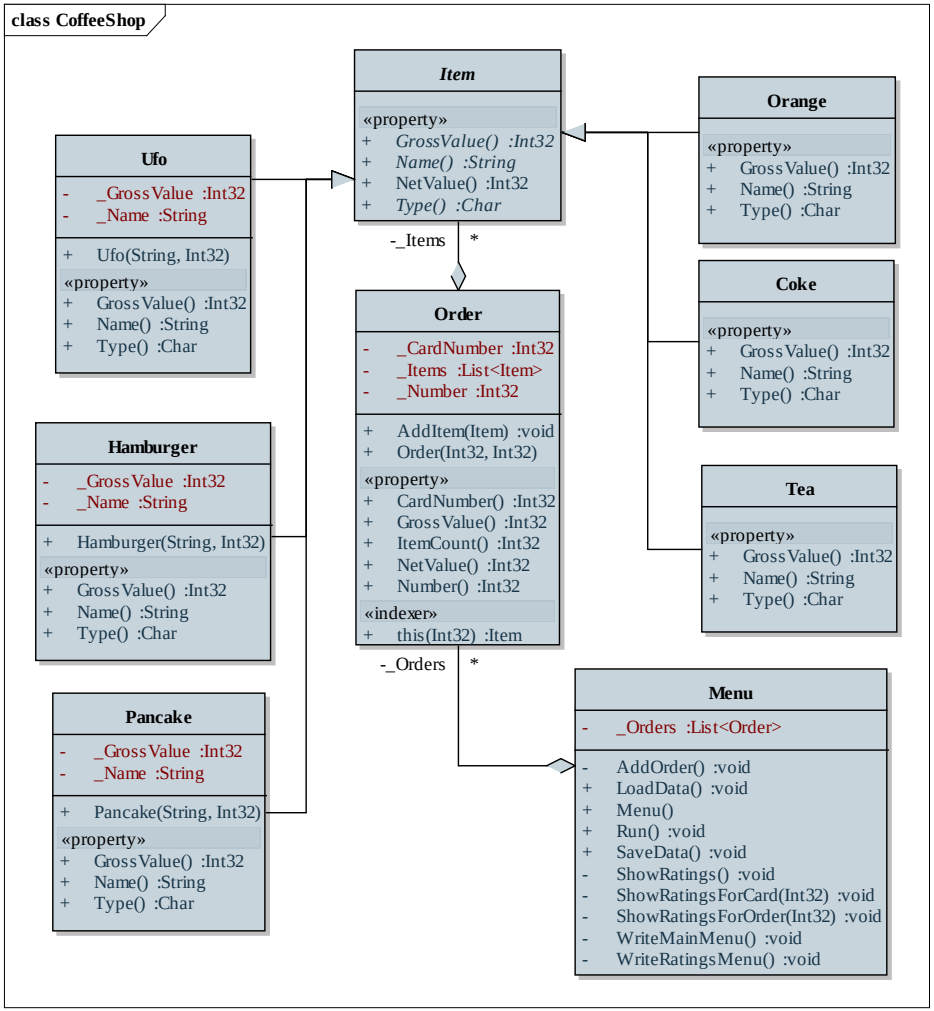
Feladat: Készítsünk programot, amely Marika néni kávézójának eladási nyilvántartását végigköveti.

- a kávézóban 3 féle étel (hamburger, ufo, palacsinta 10%-os áfával), illetve 3 féle ital (tea, narancslé, kóla 25%-os áfával) közül lehet választani
- az ételek ezen belül különfélék lehetnek, amelyre egyenként lehet árat szabni, és elnevezni, az italok árai rögzítettek
- minden rendelés több ételből és több italból áll, és sorszámmal rendelkezik, esetlegesen törzsvásárlói számmal, amelyet hagyhatunk üresen is
- lehessen lekérdezni egy adott rendelésre, egy adott törzsvásárlóra, illetve összesítve a bruttó, nettó összeget

A C# programozási nyelv

Példa

Tervezés:



Példa

Megoldás:

```
public void LoadData() {
   try {
      StreamReader reader =
         new StreamReader("data.dat");
      …
      if (!reader.EndOfStream) {
         row = reader.ReadLine();
         while (!reader.EndOfStream){
         String[] rowArray = row.Split(';');
            // a sort felbontjuk a ; mentén
         orderNumber = Int32.Parse(rowArray[0]);
            // a sor elemeit átalakítjuk a
            // megfelelő típusra
```

Példa

Megoldás:

```
            …
        }
        reader.Close();
    }
    catch (Exception) {
        Console.WriteLine("Az adatok betöltése
                           sikertelen!");
    }
}
```

A C# programozási nyelv

Objektum inicializálások

- Az *objektum inicializálás* lehetővé teszi, hogy megspóroljuk a konstruktor elkészítését, ugyanis automatikusan le tudja generálni a konstruktort
  - az inicializálásban az objektumnak tetszőleges publikus tulajdonságának, vagy mezőjének adhatunk értéket
  - pl.:

```
class Person {
   public String Name;
   public Int32 Age;
}

Person j = new Person{ Name = "John" };
Person t = new Person{ Name = "Tom", Age = 30 };
```

A C# programozási nyelv

Anonim típusok

- A *névtelen típusok (anonymous types)* lehetővé teszik egyszerű, rekordszerű objektumok létrehozását, amelyeknek nem rendelkeznek önálló típussal
  - a névtelen típusra csak a **`var`** kulcsszóval hivatkozhatunk
  - a létrehozását objektum inicializálással kell elvégeznünk, tetszőleges tulajdonságneveket megadva
  - a megadott tulajdonságok publikusak, és csak olvashatóak lesznek
- Pl.: **`var j = new { Name = "John", Age = 25 };`**
- A névtelen típusokhoz a fordítóprogram generál osztályt, ez közvetlenül az **`Object`** leszármazottja lesz

A C# programozási nyelv

Bővítő metódusok

- Az objektumorientált koncepció szerint amennyiben egy osztályhoz új metódust akarunk hozzávenni, akkor
  - az eredeti osztályban kell felvennünk, vagy
  - származtatni kell egy új osztályt az eredetiből
- Amennyiben ez kényelmetlen, vagy nem megoldható (pl. beépített osztályoknál), akkor a *bővítő metódusok (extension methods)* biztosítják a megoldást
  - kívülről úgy használhatóak, mintha az eredeti osztályban lennének, de valójában egy másik osztályban találhatóak (egy statikus osztály statikus metódusai)
  - lehetőséget adnak kész osztályok kiegészítésére

A C# programozási nyelv

Bővítő metódusok

- Pl.:

```
public static class StringExtensions
{ // String bővítő műveletek osztálya
    public static String First(this String str,
                               Int32 n){
        return str.SubString(0, n);
    } // visszaadja az első valamennyi karaktert
}

String s = "hello world";
String h = s.First(5);
// innentől használható, mint a String egy
// metódusa, csak egy paraméterrel
// másként: StringExtensions.First(s, 5);
```

A C# programozási nyelv

Lamda-kifejezések

- A *lambda-kifejezések (lambda-expressions)* funkcionális programozásból átvett elemek, amelyek egyszerre függvényként és objektumként is viselkednek
- A λ-kifejezést az **=>** operátorral jelöljük, tőle balra a paraméterek, jobbra a művelet törzse írható le, pl.:

```
a => a * a // négyzetre emelés
x => x.Length < 5 // 5-nél rövidebb szövegek
(x, y) => x + y; // összeadás
() => 5; // konstans 5
```

- A λ-kifejezést elmehetjük változóként is, típusa a sablonos **Func<...>** lesz, pl.:

```
Func<String, Boolean> lt5 = x => (x.Length < 5);
```

A C# programozási nyelv

Lamda-kifejezések

- Az eltárolt kifejezés bármikor futtathatjuk, mint egy függvényt, pl.: `Boolean l = lt5("Hello!"); // l hamis lesz`
- A λ-kifejezések tetszőlegesen összetett utasítássorozatot is tartalmazhatnak, nem csak egy kifejezés kiértékelését, ekkor a tartalmat blokkba kell helyezni, pl.:

```
Func<Int32, Int32> pow2 = x => {
   x = x * x;
   return x;
};
```

- A λ-kifejezések speciális típusa az akció (**`Action`**), amely egy paraméter és visszatérési érték nélküli tevékenység, pl.:

```
Action hello = () => { Console.Write("Hello!"); };
```

A C# programozási nyelv

Lamda-kifejezések

- A λ-kifejezések további előnye, hogy használhatják a lokálisan elérhető változókat, pl.:

```
void InvokeAction(Action act){
   act(); // akciót végrehajtó metódus
}
…
Int32 i = 1;
Action act = () => { // akció létrehozása
   while (i < 10) { // az i paraméter kívülről jön
      Console.WriteLine(i); i++;
   }
}; // itt i értéke 1
InvokeAction(act); // akció végrehajtása
Console.WriteLine(i); // itt i értéke 10
```

A C# programozási nyelv

Nyelvbe ágyazott lekérdezések

- A *nyelvbe ágyazott lekérdezések (Language Integrated Query)* lényege, hogy objektumorientált környezetben valósíthassunk meg lekérdező utasításokat
  - hasonlóan a relációs adatbázisok SQL nyelvéhez
  - pl.:

```
List<Int32> nrList =
   new List<Int32> { 1, 2, 3, 4 };
var numQuery = from i in numberList // honnan
               where i < 4 // feltétel
               select i; // mit
```

  - a lekérdezés eredménye egy speciális gyűjtemény (**`IEnumerable`**) lesz, ennek köszönhetően a tartalmát felsorolhatjuk (**`foreach`** segítségével)

Nyelvbe ágyazott lekérdezések

- Valójában a háttérben a **`System.Linq.Enumerable`** típusban definiált kiegészítő metódusok futnak le a gyűjteményre, amelyek λ-kifejezést fogadnak paraméterként, és azt alkalmazzák az elemekre
  - pl.:

```
var numQuery = numberList
               .Where(i => i < 4)
               .Select(i => i);
```

  - így még nagyobb szabadságunk van a lekérdezések megfogalmazáshoz (pl. az identitás **`Select`** elhagyható)
  - az **`Enumerable`** osztály ezen felül további műveleteket biztosít sorozatok generálására (**`Range`**, **`Repeat`**, **`Empty`**)

A C# programozási nyelv

Nyelvbe ágyazott lekérdezések

- A LINQ használatának előnye az úgynevezett *késleltetett végrehajtás* (deferred execution), amely lehetővé teszi, hogy a kifejezés nem a híváskor, hanem az eredmény bejárásakor (a bejáró léptetésekor) hajtódjon végre
  - a lekérdezés ilyenkor nem egyszerű felsoroló típust, hanem lekérdezés eredményt (**`IQueriable`**) ad, amely lényegében a kifejezést tárolja, viszont szintén bejárható (megvalósítja az **`IEnumerable`**-t)
  - így bármikor módosítjuk a forrás gyűjteményt, vagy valamilyen külső változót, a feldolgozáskor a módosított gyűjteményen fut le a kiértékelés

A C# programozási nyelv

Nyelvbe ágyazott lekérdezések

- Pl.:

```
var numList = new List<Int32> { 1, 2, 5, 6 };
var numQuery = numList
                  .Where(i => i < 4);
   // a lekérdezés nem fut le, csak eltárolódik
   // az eredmény IQueriable lesz
…
numList.Add(3); // módosítjuk a gyűjteményt
…
foreach (Int32 n in numQuery){
   // a kifejezés itt fut le, amikor bejárjuk
   Console.WriteLine(n);
} // eredmény: 1 2 3
```

A C# programozási nyelv

Nyelvbe ágyazott lekérdezések

- A késleltetett végrehajtás persze bizonyos esetekben ront a teljesítményen (mivel minden bejáráskor lefut a kiértékelés), ha azonnali végrehajtást szeretnénk, az eredményt konvertálnunk kell (**`ToArray`**, **`ToList`**, **`ToDictionary`**)

- Pl.:

```
var numList = new List { 1, 2, 5, 6 };
var numQuery = numList.Where(i => i < 4).ToList();
   // a lekérdezés eredményét listába gyűjtjük
numList.Add(3); // módosítás
…
foreach (Int32 n in numQuery){
   Console.WriteLine(n);
} // eredmény: 1 2
```

A C# programozási nyelv

Nyelvbe ágyazott lekérdezések

- Kiegészítő metódusokon keresztül további lehetőségek is elérhetőek a lekérdezésekben, pl.:
  - statisztikai függvények (**`Sum`**, **`Average`**, **`Min`**, **`Count`**, ...)
  - keresések (**`Any`**, **`All`**, **`FirstOrDefault`**, ...)
  - elem lekérdezés (**`First`**, **`Last`**, **`ElementAt`**)
  - valahány elem vétele (**`Take`**), vagy kihagyása (**`Skip`**), speciális lekérések (**`Distinct`**, ...)
  - csoportosítás (**`GroupBy`**), konkatenálás (**`Concat`**)
  - halmazműveletek (**`Union`**, **`Intersect`**, **`Except`**)
  - összekapcsolások (**`Join`**, **`GroupJoin`**)
  - sorba rendezés (**`OrderBy`**, **`OrderByDescending`**), akár többszörösen (**`ThenBy`**), fordított sorrend (**`Reverse`**)

A C# programozási nyelv

Nyelvbe ágyazott lekérdezések

- Pl.:

```
Int32[] s1 = { 1, 2, 3 }, s2 = { 2, 3, 4 };
Int32 sum = s1.Sum(); // számok összege
Int32 evenCount = s1.Sum(x => x % 2 == 0 ? 1 : 0);
   // megadjuk, mit összegezzen, így a páros
   // számok számlálása lesz
var union = s1.Union(s2);
   // két gyűjtemény uniója: { 1, 2, 3, 4 }
var evens = union.Select(x => x % 2 == 0);
   // páros számok kiválogatása
Int32 evenCount =
   s1.Union(s2).Sum(x => x % 2 == 0 ? 1 : 0);
   // unió, majd a páros számok számlálása
```

A C# programozási nyelv

Nyelvbe ágyazott lekérdezések

- Pl.:

```
Person[] people = new Person[] {
   new Person { Name = "James", Age = 31}, ...
};
var peopleQuery = people.Where(p => (p.Age > 20))
            // 20 év felettiek
           .OrderByDescending(p => p.Age)
            // kor szerint csökkenő sorrendben
           .ThenBy(p => p.Name)
            // majd név szerint
           .Select(p => new {p.Name, Senior =
                              p.Age > 30} );
            // az eredmény anonim típusú a Name és
            // Senior attribútumokkal
```

A C# programozási nyelv

Példa

Feladat: Módosítsuk az egyetemi polgárok kezelését speciális nyelvi lehetőségek és LINQ használatával, cseréljük le deklaratív lekérdezésekre minden olyan bejárást, amely azzal könnyen megfogalmazható.

- a tulajdonságokat automatikussá tesszük, privát írási hozzáféréssel
- a lineáris keresésekből szűrő lekérdezéseket (**`Where`**), a név lekérdezésből transzformációs lekérdezéseket (**`Select`**) készítünk, az eredményt listává alakítjuk (**`ToList()`**)
- a kiírásokat egyben végezzük el, ehhez aggregációt (**`Aggregate`**) alkalmazunk, ahol sörtöréssel (**`Environment.NewLine`**) szeparáljuk az értékeket

Példa

*Megoldás (**Course.cs**):*

```
public static List<String> ListedCourseNames {
   get {
      return _ListOfCourses.Select(course =>
         course.Name).ToList();
      // csak a nevet kérdezzük le a listából
   }
}
public static Course GetCourse(String name) {
   return _ListOfCourses.Where(couse => couse.Name
       == name).FirstOrDefault();
   // leválogatjuk azokat, ahol a kurzusnév
   // egyezik, és azokból az elsőt adjuk vissza
}
```