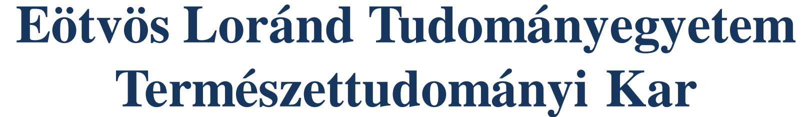



# Alkalmazott Modul III

## 5. előadás

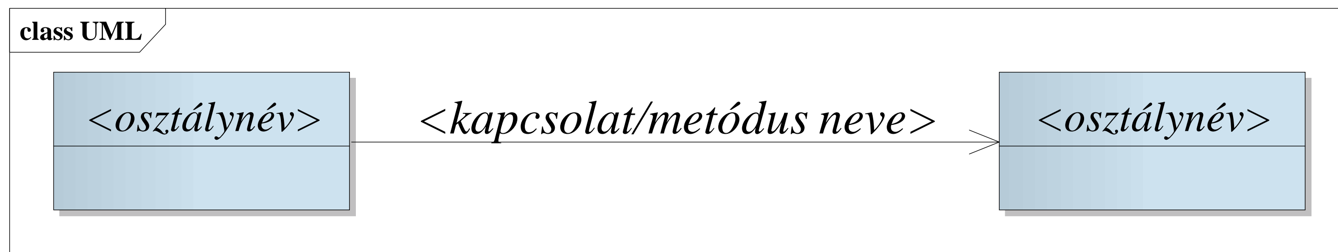
# Objektumorientált programozás: osztályszerkezetek megvalósítása

© 2011.10.17. Giachetta Roberto
groberto@inf.elte.hu
http://people.inf.elte.hu/groberto

# Osztályszerkezetek megvalósítása

## Objektumok közötti kapcsolatok

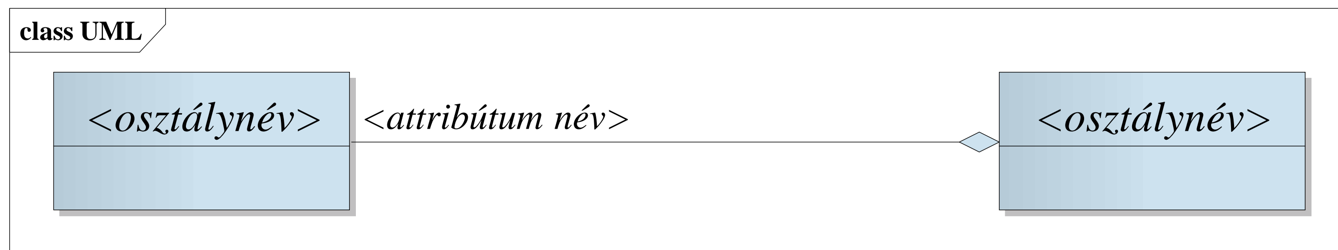
- Az alkalmazásokban rendszerint több osztály szerepel, amelyek objektumai kommunikálhatnak egymással, ezért az osztályok között relációkat állíthatunk fel, amely lehet:
  - *egyszerű kommunikáció* (*asszociáció*): az osztály meghívja más osztály (látható) metódusát, hivatkozik rá a műveletek végrehajtása során, paraméterként, vagy visszatérési értékként



# Osztályszerkezetek megvalósítása

## Objektumok közötti kapcsolatok

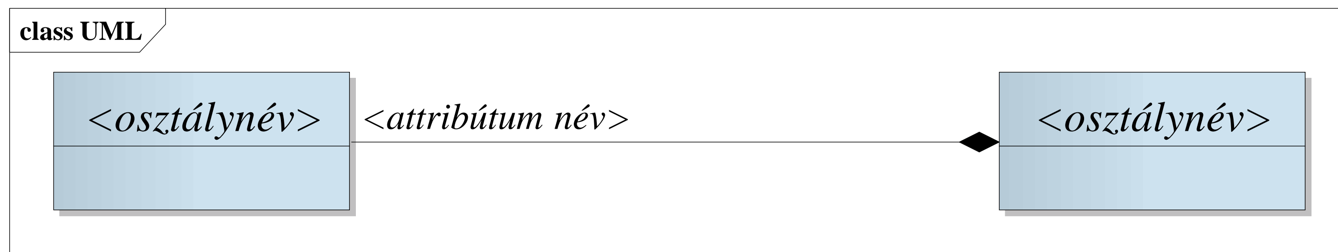
- *hivatkozás* (*aggregáció*): az osztály egy, vagy több példánya meg van hivatkozva a másik osztály egy, vagy több mezőjében
  - a hivatkozáshoz mutatón történő kezelés, vagyis referencia típus szükséges
  - a tényleges objektum nincs benne, ezért az élettartama nem függ tőle, így nem garantált, hogy elérhető



# Osztályszerkezetek megvalósítása

## Objektumok közötti kapcsolatok

- *tartalmazás* (*kompozíció*): az osztály egy példányát a másik osztály tartalmazza egy, vagy több mezőjében
  - többnyire érték típust követel, de akkor is érvényes, ha egy mezőt az objektum hoz létre, és a destruktor semmisít meg
  - az objektum élettartama megegyezik a tartalmazó objektum élettartamával, ezért mindig elérhetővé válik



# Osztályszerkezetek megvalósítása

## Példa

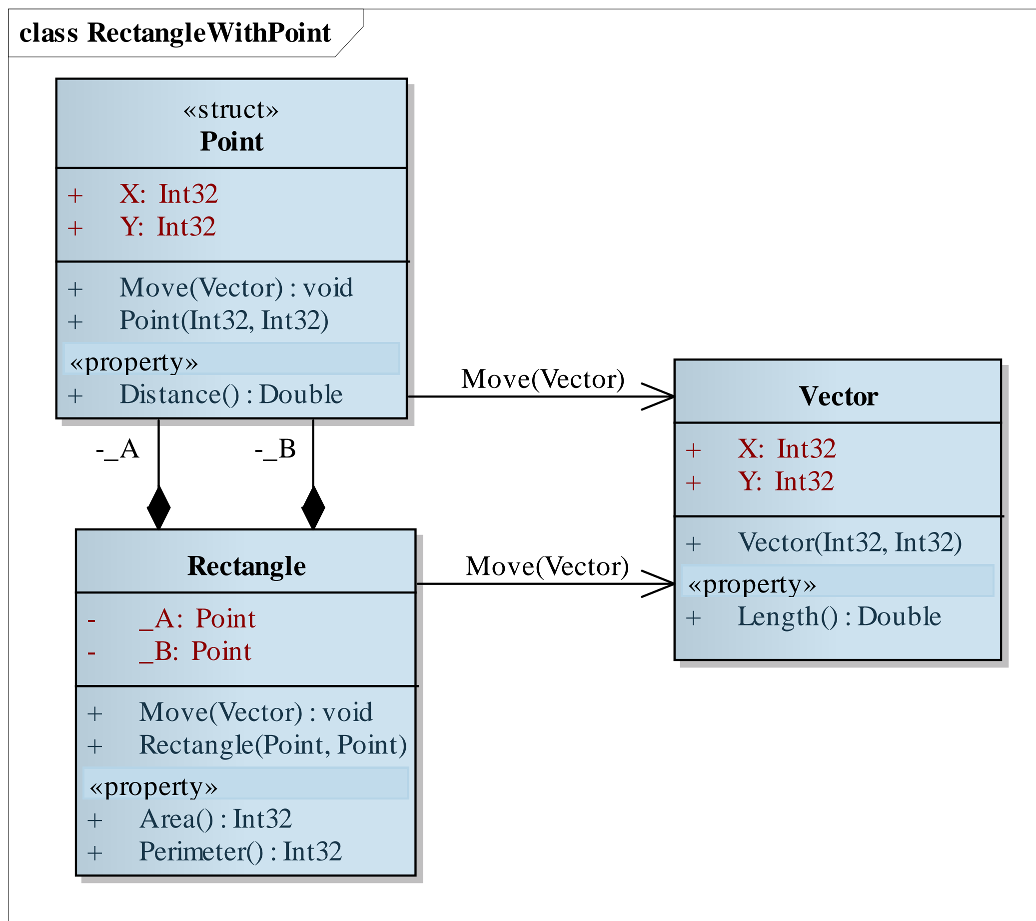
*Feladat*: Ábrázoljuk a téglalapot két ellentétes sarokpontja koordinátájával, és adjunk meg egy eltolási műveletet, amely tetszőleges vektorral arrébb tudja helyezni a téglalapot.

- 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

# Osztályszerkezetek megvalósítása

## Példa

*Tervezés:*



# Osztályszerkezetek megvalósítása

## 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;

    …
}
```

# Osztályszerkezetek megvalósítása

## 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);
   }
```

# Osztályszerkezetek megvalósítása

## 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
```

# Osztályszerkezetek megvalósítása

## Példa

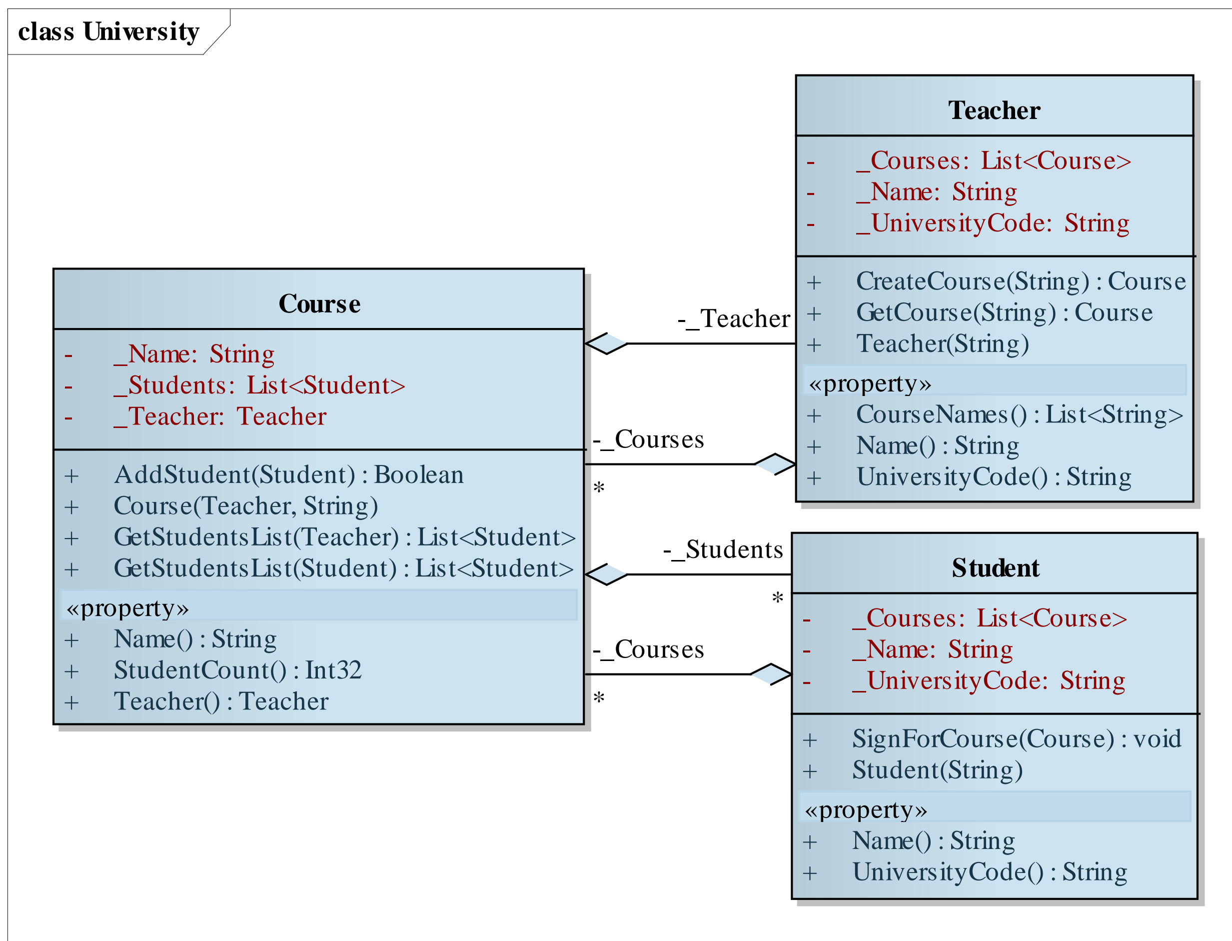
*Feladat*: Egy egyetemi kurzust egy oktató tart, és több hallgató veheti fel, és ennek megfelelően a kurzusnak tisztában kell lennie hallgatóival és oktatójával, ugyanakkor az oktatónak és a hallgatóknak is tisztában kell lenniük kurzusaikkal.

- 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

# Osztályszerkezetek megvalósítása

## Példa

*Tervezés:*



# Osztályszerkezetek megvalósítása

## 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
}
```

# Osztályszerkezetek megvalósítása

## 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
    }
  }
```

# Osztályszerkezetek megvalósítása

## 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;
      }
   …
```

# Osztályszerkezetek megvalósítása

## 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
    }
}
```

# Osztályszintű tagok

## Élettartam és láthatóság

- A procedurális programozás csak lokális, illetve globális adatok kezelését tette lehetővé mind a láthatóság, mind az élettartam tekintetében
- Az objektumorientált programozás magával hozta az objektumszintű élettartam bevezetését osztályszintű láthatóság mellett, mivel a hasonló típusú objektumok láthatják egymás rejtett tagjait
    - ugyanakkor a globális adatok megszűntek, minden adat élettartamát az őt tartalmazó objektum határozza meg
    - időnként szükség lenne olyan mezőkre, amelyek láthatóságban és élettartamban is osztályszinten működnének, így az osztály objektumai egy közös adatot használnának

# Osztályszintű tagok

## Statikus mezők

- Lehetőségünk van *osztályszintű*, *statikus mezők* létrehozására a `static` kulcsszó használatával, az így megjelölt mezők
  - az osztály teljes működése (általában a program futása) során jelen vannak
  - már az osztályban értéket kaphatnak
  - bármely osztálybeli példány számára elérhetőek az implementáción belül, de kívülről, illetve nyílt rekurzión keresztül nem (hiszen nem a példányhoz tartoznak)
  - az osztály példányosítása nélkül használhatóak az osztályon keresztül történő hivatkozással
  - tehát csak egy van belőlük jelen a teljes osztályban, függetlenül attól, hány objektumot hozunk létre a típusból

# Osztályszintű tagok

## Statikus mezők

- Pl.:

```
class NumClass {
   public static Int32 Number = 0;
   // osztályszintű mező 0 kezdőértékkel
   public NumClass() { Number++; }
   // osztályszintű mező elérése a konstruktorban
}

Console.WriteLine(NumClass.Number) // eredmény: 0
NumClass n1 = new NumClass();
Console.WriteLine(NumClass.Number) // eredmény: 1
NumClass n2 = new NumClass();
Console.WriteLine(NumClass.Number) // eredmény: 2
```

# Osztályszintű tagok

## Előnyök

- Az osztályszintű mezők előnyei:
  - a lokális és objektumszintű élettartam mellett megjelenik az osztályszintű élettartam
  - az osztályon keresztül bárhol elérhetjük a (látható) mezőket, objektumpéldányok és paraméterek használata nélkül, ugyanakkor mégse globális az elérés, így elkerülhetőek a névütközések és a hatókörön való túllépés
  - az objektumok az osztályon belül közvetlenül kommunikálhatnak egymással
  - konstans, illetve általános jellemzők esetén nem szükséges objektumonként eltárolni értékeket (pl. `Int32.MaxValue` minden egész számra érvényes mező)

# Osztályszintű tagok

## Statikus metódusok

- Analóg módon lehetőségünk van *osztályszintű metódusok* és *tulajdonságok* létrehozására is
  - az osztály példányosítása nélkül elérhetőek bárhol a programban, így nincs szükségünk az objektumpéldányok kezelésére, ha a működést biztosítani akarjuk
  - emiatt nem férhetnek hozzá objektumszintű mezőkhöz, illetve magához az objektumhoz (`this`), csak osztályszintű mezőkhöz
  - olyan működést valósítunk meg osztályszinten, amely független az objektumoktól, vagy kötődik az osztályszintű mezőkhöz (sokszor segédműveletek, amelyek paraméterben kapják az objektumokat, vagy mezőiket)

# Osztályszintű tagok

## Statikus metódusok

- Pl.:

```
class NumClass {
   private static Int32 _Number = 10;
   // osztályszintű mező 0 kezdőértékkel
   public static Int32 Number{
      get { return _Number; }
   } // osztályszintű tulajdonság
   public static void Increase() { _Number++; }
   // osztályszintű metódus
}

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

# Osztályszintű tagok

## Statikus konstruktor

- Mivel bármely metódus lehet osztályszintű, maga a konstruktor is lehet az
  - a statikus konstruktor feladata a statikus mezők inicializálása, valamint olyan műveletek elvégzése, amelyet bármely objektum létrehozása előtt meg kell tenni
  - a statikus konstruktort a rendszer automatikusan hívja meg az osztályra történő első hivatkozás előtt, emiatt nem paraméterezhető, nem adható neki láthatóság, direkt nem futtatható, és csak egyszer futhat a program során
  - a statikus mellett objektumkonstruktorai is lehetnek az osztálynak

# Osztályszintű tagok

## Statikus mezők

- Pl.:

```
class NumClass {
   public static Int32 Number;
   // osztályszintű mező 0 kezdőértékkel
   static NumClass() { Number = 10; }
   // osztályszintű konstruktor
   public NumClass() { Number++; }
   // osztályszintű mező elérése a konstruktorban
}

// ezen a ponton lefut a statikus konstruktor
Console.WriteLine(NumClass.Number) // eredmény: 10
NumClass n1 = new NumClass();
Console.WriteLine(NumClass.Number) // eredmény: 11
```

# Osztályszintű tagok

## Statikus osztályok

- A statikus tulajdonság általánosítható az osztályra is, létrehozhatunk statikus osztályokat is
  - csak statikus tagjai lehetnek
  - nem példányosíthatóak, inkább globális érték, és műveletgyűjteménynek tekinthetőek
  - lehet statikus konstruktora, amely lefut az első taghivatkozás előtt
- Pl.:

```
static class NumClass {
   …
}
```

# Osztályszintű tagok

## Példa

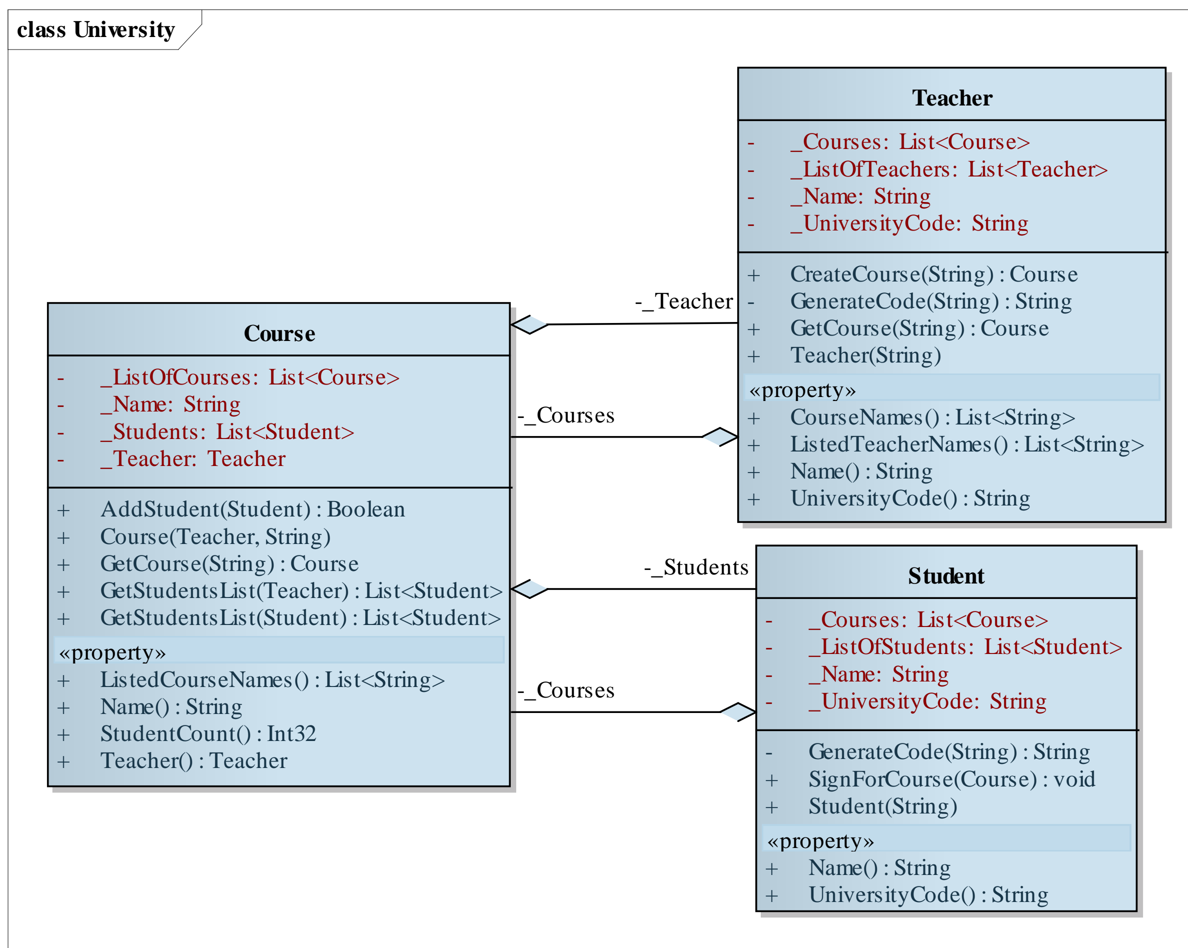
*Feladat:* Az egyetemi oktatók és hallgatók esetén hasznos lenne az azonosítót úgy generálni, hogy két azonos ne forduljon elő a rendszerben.

- 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

# Osztályszerkezetek megvalósítása

## Példa

*Tervezés:*



# Osztályszintű tagok

## 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
        …
```

# Osztályszintű tagok

## 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
   }
   …
```

# Osztályszintű tagok

## 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);
   }
}
```

# Operátorok definiálása

## Az operátor, mint metódus

- Az *operátor* olyan művelet, amelyet a nyelvben megadott műveleti jelek segítségével futtathatunk le
  - tehát csak a megadott módon és alakban hívhatóak meg, és megadott precedencia, illetve asszociativitás szerint hajtódik végre
  - objektumorientált környezetben ezek a műveletek ugyanúgy metódusként jelennek meg, mint bármely más művelet
- Osztályainkhoz nem csupán névvel ellátott metódusokat készíthetünk, hanem a nyelv által ismert operátorokat, mint műveleteket is megvalósíthatjuk
  - ezután a műveleti jel alkalmazható lesz a típusunkra
  - a műveleteket tetszőleges viselkedéssel ruházhatjuk fel

# Operátorok definiálása

## Szintaxis

- Az operátorokat osztályszintű metódusként kell definiálnunk az **operator** kulcsszó használatával és a műveleti jel megadásával, a következő formában:

```
<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

# Operátorok definiálása

## Definiálás és használat

- Pl.:

```
public class NumClass {
   private Int32 _Number;
   public OperatorClass(Int32 n) { _Number = n; }

   public static NumClass operator +(NumClass a,
      NumClass b){ // összeadás operátor
      return new NumClass(a._Number + b._Number);
   } // a megfelelő eredmény visszaadása
}

NumClass n1 = new NumClass(3);
NumClass n2 = new NumClass(5);
… n1 + n2 … // a + művelet alkalmazható
```

# Operátorok definiálása

## Szabályok

- Operátor írásakor tartanunk kell magunkat pár szabályhoz:
  - operátorok meghívása csak rögzített formában történhet, és a műveleti precedencia már rögzített (halmozott használatuk esetén)
  - a legtöbb műveleti jel definiálható, de bizonyosak nem (pl. `=`, `new`, `typeof`, …)
  - a paraméterek száma rögzített, típusuk szabályozható, de célszerű, hogy legalább az egyik paraméter az adott osztály példánya legyen (különben helyezzük más osztályba az operátort)
  - a paraméterek lesznek az operátor operandusai (infix művelet esetén a bal, illetve jobb operandusa), a visszatérési érték az eredménye

# Operátorok definiálása

## 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

# Osztályszerkezetek megvalósítása

## Példa

*Tervezés:*

**class NumberTypes**

**Complex**

- \- _Imaginary: Double
- \- _Real: Double

---

- \+ Complex()
- \+ Complex(Double)
- \+ Complex(Double, Double)
- \+ Conjugation() : Complex
- \+ operator +(Complex, Complex) : Complex
- \+ operator +(Complex, Double) : Complex
- \+ operator +(Double, Complex) : Complex

«property»

- \+ Imaginary() : Double
- \+ Real() : Double

**Rational**

- \- _Denominator: Int32
- \- _Nominator: Int32

---

- \+ operator -(Rational, Rational) : Rational
- \+ operator *(Rational, Rational) : Rational
- \+ operator /(Rational, Rational) : Rational
- \+ operator +(Rational, Rational) : Rational
- \+ Rational()
- \+ Rational(Int32, Int32)
- \- Simplify(Int32*, Int32*) : void

«property»

- \+ Denominator() : Int32
- \+ Nominator() : Int32

# Operátorok definiálása

## 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

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

# Operátorok definiálása

## 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
   …
```