

Alkalmazott Modul III

5. előadás

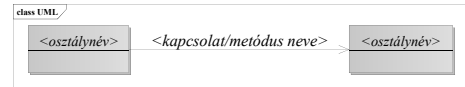
Objektumorientált programozás: osztályszerkezetek megvalósítása

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

Osztályszerkezetek megvalósítása

Objektumok közötti kapcsolatok

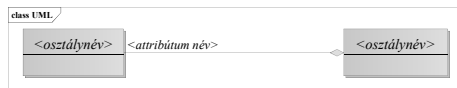
- Az alkalmazásokban rendszerint több osztály szerepel, amelyek objektumai kommunikálhatnak egymással, ezért az osztályok között relációkat állíthatunk fel, amely lehet:
- egyszerű kommunikáció (asszociáció)*: az osztály meghívja más osztály (látható) metódusát, hivatkozik rá a műveletek végrehajtása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



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 destruktorként semmisít meg
- az objektum élettartama megegyezik a tartalmazó objektum élettartamával, ezért mindig elérhetővé válik



Osztályszerkezetek megvalósítása

Példa

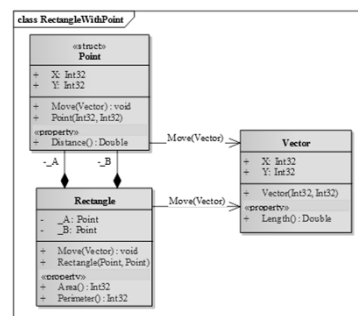
Feladat: Ábrázoljuk a téglalapot két ellentétes sarokpontja koordinátájával, és adjunk meg egy eltolási műveletet, amely tetszőleges vektorral arrébb tudja helyezni a téglalapot.

- 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;

    ...
}
```

ELTE TTK, Alkalmazott modul III

5:7

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);
    }
}
```

ELTE TTK, Alkalmazott modul III

5:8

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
```

ELTE TTK, Alkalmazott modul III

5:9

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 oktatóknak 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 lekérdezhető
- az adatok kölcsönös eltávolását hivatkozások segítségével valósítjuk meg, hiszen minden objektum élettartama független kell, hogy legyen

ELTE TTK, Alkalmazott modul III

5:10

Osztályszerkezetek megvalósítása

Példa

Tervezés:



ELTE TTK, Alkalmazott modul III

5:11

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
    }
}
```

ELTE TTK, Alkalmazott modul III

5:12

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
}
```

ELTE TTK, Alkalmazott modul III

5:13

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;
        }
    }
    ...
}
```

ELTE TTK, Alkalmazott modul III

5:14

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
}
```

ELTE TTK, Alkalmazott modul III

5:15

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

ELTE TTK, Alkalmazott modul III

5:16

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 rekurzió 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

ELTE TTK, Alkalmazott modul III

5:17

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
```

ELTE TTK, Alkalmazott modul III

5:18

Osztályszintű tagok	
Előnyök	
<ul style="list-style-type: none"> Az osztályszintű mezők előnyei: <ul style="list-style-type: none"> 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 objektumként eltárolni értékeket (pl. <code>Int32.MaxValue</code> minden egész számra érvényes mező) 	5:19
ELTE TTK, Alkalmazott modul III	

Osztályszintű tagok	
Statikus metódusok	
<ul style="list-style-type: none"> Analóg módon lehetőségünk van <i>osztályszintű metódusok és tulajdonságok</i> létrehozására is <ul style="list-style-type: none"> az osztály példányosítása nélkül elérhetőek bárhol a programban, így nincs szükségünk az objektumpéldányok kezelésére, ha a működést biztosítani akarjuk emiattnem férhetnek hozzá objektumszintű mezőkhöz, illetve magához az objektumhoz (<code>this</code>), 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) 	5:20
ELTE TTK, Alkalmazott modul III	

Osztályszintű tagok	
Statikus metódusok	
<ul style="list-style-type: none"> Pl.: <pre> 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 </pre> 	5:21
ELTE TTK, Alkalmazott modul III	

Osztályszintű tagok	
Statikus konstruktor	
<ul style="list-style-type: none"> Mivel bármely metódus lehet osztályszintű, maga a konstruktor is lehet az <ul style="list-style-type: none"> a statikus konstruktor feladata a statikus mezők inicializálása, valamint olyan műveletek elvégzése, amelyet bármely objektum létrehozása előtt meg kell tenni a statikus konstruktor a rendszer automatikusan hívja meg az osztályra történő első hivatkozás előtt, emiattnem paraméterezhető, nem adható neki láthatóság, direkt nem futtatható, és csak egyszer futhat a program során a statikus mellett objektumkonstruktorai is lehetnek az osztálynak 	5:22
ELTE TTK, Alkalmazott modul III	

Osztályszintű tagok	
Statikus mezők	
<ul style="list-style-type: none"> Pl.: <pre> 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 </pre> 	5:23
ELTE TTK, Alkalmazott modul III	

Osztályszintű tagok	
Statikus osztályok	
<ul style="list-style-type: none"> A statikus tulajdonság általánosítható az osztályra is, létrehozhatunk statikus osztályokat is <ul style="list-style-type: none"> csak statikus tagjai lehetnek nem példányosíthatóak, inkább globális érték, és műveletgyűjteménynek tekinthetőek lehet statikus konstruktorra, amely lefut az első taghivatkozás előtt Pl.: <pre> static class NumClass { ... } </pre> 	5:24
ELTE TTK, Alkalmazott modul III	

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

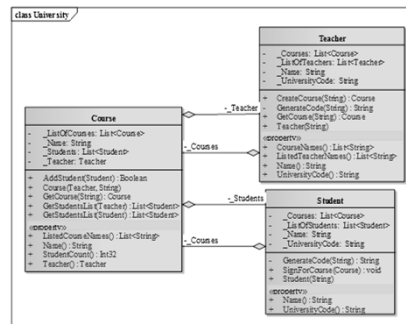
ELTE TTK, Alkalmazott modul III

5:25

Osztályszerkezetek megvalósítása

Példa

Tervezés:



ELTE TTK, Alkalmazott modul III

5:26

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
    }
    ...
}
```

ELTE TTK, Alkalmazott modul III

5:27

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
    }
    ...
}
```

ELTE TTK, Alkalmazott modul III

5:28

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);
    }
}
```

ELTE TTK, Alkalmazott modul III

5:29

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 nével 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

ELTE TTK, Alkalmazott modul III

5:30

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

ELTE TTK, Alkalmazott modul III

5:31

Operátorok definiálása

Definiálás és használat

- Például:

```
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
```

ELTE TTK, Alkalmazott modul III

5:32

Operátorok definiálása

Szabályok

- Operátor írásakor tartanunk kell magunkat pár szabályhoz:
 - operátorok meghívása csak rögzített formában történhet, és a műveleti precedencia már rögzített (halmazos 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 helyezük más osztályba az operátort)
 - a paraméterek lesznek az operátor operandusai (infix művelet esetén a bal, illetve jobb operandusa), a visszatérési érték az eredménye

ELTE TTK, Alkalmazott modul III

5:33

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 alpműveletet (+, -, *, /) racionális számok között, és ügyeljünk 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

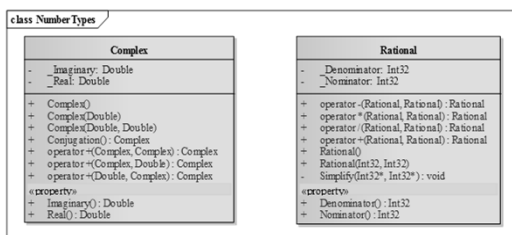
ELTE TTK, Alkalmazott modul III

5:34

Osztályszerkezetek megvalósítása

Példa

Tervezés:



ELTE TTK, Alkalmazott modul III

5:35

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
```

ELTE TTK, Alkalmazott modul III

5:36

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
    ...
}
```