Pázmány Péter Katolikus Egyetem
Információs Technológiai Kar

Programozás .NET környezetben

3. gyakorlat

Objektumorientált alkalmazások

© 2013.02.28. Cserép Máté
mcserep@caesar.elte.hu
http://mcserep.web.elte.hu

Objektumorientált alkalmazások
Osztályszintű tagok

- Lehetőségünk van *osztályszintű*, *statikus 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 a program futása során csak egyszer, automatikusan hívódik meg az osztály első használata előtt
  - a teljes osztály is megjelölhető statikusnak, akkor csak statikus tagokat tartalmazhat, és nem példányosítható

Objektumorientált alkalmazások
Osztályszintű tagok

- Pl.:

```
static 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
```


Objektumorientált alkalmazások
Osztályszintű tagok

- Pl.:

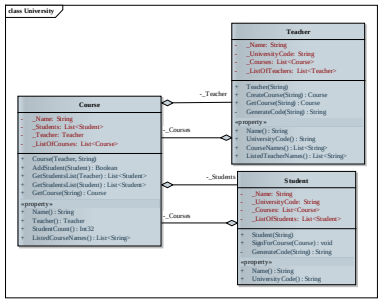
```
class NumClass {
   public static Int32 Number;
   // osztályszintű mező 0 kezdőértékkel
   static NumClass() { Number = 10; }
   // osztályszintű konstruktor
   public NumClass() { Number++; }
   // osztályszintű mező elérése a konstruktorban
}

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


Objektumorientált alkalmazások
Példa #1

- 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

Objektumorientált alkalmazások
Példa #1

Tervezés:



Objektumorientált alkalmazások
Példa #1

UniversityWithStatic

Objektumorientált alkalmazások
Operátorok megvalósítása

- 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

Objektumorientált alkalmazások
Operátorok megvalósítása

- Pl.:

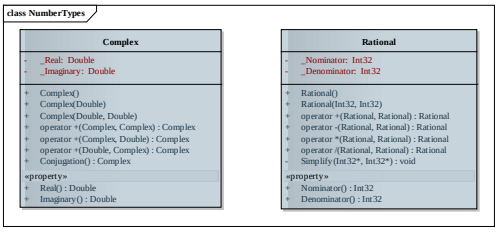
```
class NumClass {
   private Int32 _Number;
   public NumClass(Int32 n) { _Number = n; }

   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
```


Objektumorientált alkalmazások
Példa #2

- 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, a komplex szám konjugálását, valamint a konverziót valós és racionális értékekről.
  - 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

Objektumorientált alkalmazások
Példa #2

Tervezés:



Objektumorientált alkalmazások
Példa #2

NumberTypes

Objektumorientált alkalmazások
Öröklődés

- Lehetőségünk van az osztályok közötti kapcsolatot öröklődés segítségével megvalósítani
  - az általános osztály minden tulajdonságát örökli a speciálisabb osztály
  - az általánosabb objektumot *ősosztály*nak (*base class*), a speciálisabbat *leszármazott*nak (*subclass*, *derived class*) nevezzük
- Az implementációban az osztályoknál kell jelölünk úgy, hogy megadjuk, mely ősosztályból származtatunk

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

- Az öröklődés csak referenciaosztályokra (**class**) használható

Objektumorientált alkalmazások
Öröklődés

- Öröklődéskor használhatunk egy harmadik láthatóságot, a *védett*et (**protected**), amely által megjelölt tagok a leszármazottakban is láthatóak lesznek, de kívül nem
  - az osztálydiagramban # jellel jelölhetjük
- Pl.:

```
class BaseClass { // ős
   protected Int32 _IntValue; // mező

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


Objektumorientált alkalmazások
Öröklődés

```
class DerivedClass : BaseClass { // leszármazott
   // minden, ami a BaseClass-ben volt,
   // automatikusan bekerül ide

   private Single _FloatValue; // további mezők
   // további műveletek:
   public DerivedClass() { _FloatValue = 2; }
   public void PrintFloatValue() {
      Console.WriteLine(_FloatValue);
   }
   public void PrintAllValues() {
      PrintFloatValue();
      PrintIntValue(); // örökölt művelet
   }
}
```


Objektumorientált alkalmazások
Öröklődés

```
BaseClass bc = new BaseClass(); // ős példány
bc.PrintIntValue(); // ős műveleteit meghívhatjuk

DerivedClass dc = new DerivedClass();
// leszármazott példány
dc.PrintFloatValue();
// a leszármazott műveleteit meghívhatjuk
dc.PrintIntValue();
// az ős műveleteit is meghívhatjuk
dc.PrintAllValues();
// közvetetten is meghívhatjuk az ős műveleteit
```


Objektumorientált alkalmazások
Öröklődés

- Lehetőségünk direkt hivatkozni az ősből örökölt tagokra a **base** kulcsszón keresztül, pl.:

```
public void PrintAllValues() {
   PrintFloatValue();
   base.PrintIntValue(); // örökölt művelet
}
```

- A konstruktor automatikusan öröklődik
  - az ős paraméter nélküli konstruktora 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

Objektumorientált alkalmazások
Öröklődés

- Pl.:

```
class BaseClass { // ős
   private Int32 _IntValue; // rejtett mező

   public void BaseClass(Int32 v) {
      _IntValue = v;
   }
}

class DerivedClass : BaseClass { // leszármazott
   public DerivedClass (Int32 v) : base(v) {}
   // meghívjuk az ős konstruktorát a paraméterrel
}
```

Objektumorientált alkalmazások
Absztrakt osztályok

- Amennyiben az általános osztály csak a közös rész összefogására szolgál, és nem használjuk példányok létrehozására, akkor *absztrakt osztály*nak nevezzük
  - absztrakt osztályokat az **`abstract`** kulcsszóval kell megjelölnünk (osztálydiagramban dőlt betűvel jelöljük), innentől nem alkalmazható rá a **`new`** operátor, tehát nem lehet példányosítani
  - pl.:

```
abstract class BaseClass { // absztrakt ős
    …
}
```

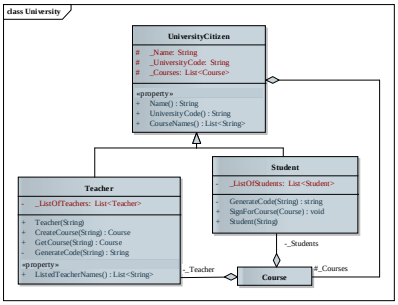
  - hasonlóan megszűntethető a példányosítás, ha a konstruktor(ok)nak nem publikus láthatóságot adunk

Objektumorientált alkalmazások
Példa #3

- 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
  - így megszűnik a mezők ismétlődésének jelentős része az oktató és hallgató osztályokban, amelyeket ezek után ugyanúgy használhatunk

Objektumorientált alkalmazások
Példa #3

Tervezés:



Objektumorientált alkalmazások
Példa #3

UniversityWithInheritance

Objektumorientált alkalmazások
Viselkedés elrejtés

- A leszármazott amellett, hogy örököl minden tagot az őstől, lehetősége van *elrejteni* az örökölt viselkedést, és újat definiálni
  - azaz lehetőségünk van ugyanolyan szignatúrájú metódusok létrehozására, amelyek más működést hajtanak végre a leszármazottban
  - akkor hasznos, amikor a speciális osztály más viselkedést kell, hogy biztosítson, pl. a területet másként számíthatjuk egy négyzet esetén, mint egy általános sokszög esetén
  - a leszármazottban jelölnünk kell, hogy szándékos az elrejtés a **`new`** kulcsszóval

Objektumorientált alkalmazások
Viselkedés elrejtés

- Pl.:

```
class BaseClass { // ős
   public void PrintMethod() {
      Console.WriteLine(1);
   }
}
class DerivedClass : BaseClass { // leszármazott
   public new void PrintMethod() {
      Console.WriteLine(2);
   } // elrejtő metódus
}

(new BaseClass()).PrintMethod(); // eredmény: 1
(new DerivedClass()).PrintMethod(); // eredmény: 2
```

Objektumorientált alkalmazások
Polimorfizmus

- Objektumorientált szerkezetben az objektumok típusa az osztály, amiből példányosítjuk őket, azonban öröklődés esetén az objektum típusának az osztály bármely őse is tekinthető, ezt *többalakúság*nak, vagy *polimorfizmus*nak nevezzük
  - ez az implementációban úgy jelenik meg, hogy egy általános osztályú hivatkozást ráállíthatunk egy speciális típusra is, pl.:
    ```
    BaseClass bc = new DerivedClass();
    ```
  - a változó *statikus típusa* a mutató típusa, a fordítóprogram ezt tudja értelmezni
  - a változó *dinamikus típusa* a ténylegesen létrehozott objektum típusa

Objektumorientált alkalmazások
Polimorfizmus

- A polimorfizmus korlátozza a tagelérést a statikus típusra, amit lehetőségünk van feloldani
  - az **`is`** operátor egy logikai kiértékelés, amely megadja, hogy az adott osztály példánya-e az objektum
  - az **`as`** operátor az adott osztály példányának tudja megfeleltetni az objektumot (amennyiben nem megfelelő az osztály **`null`** értéket kapunk)
  - pl.:

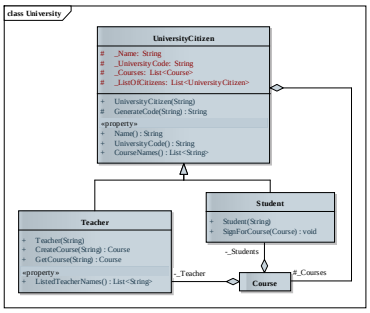
```
foreach(BaseClass listItem in bcList)
   if (listItem is DerivedClass)
      // csak a leszármazott példányokra
      Console.WriteLine(
         (listItem as DerivedClass).FloatValue);
```


Objektumorientált alkalmazások
Példa #4

- Tovább javíthatjuk az egyetemi polgárok kezelését, ha az azonosító generálást, és tárolást is az általános osztályba helyezzük.
  - az egyetemi polgárba helyezzük a statikus listát, és az azonosító generálás védett láthatósággal, a polgár konstruktorába helyezzük az alap tevékenységeket
  - kihasználva a polimorfizmust, az általános osztályban lévő statikus lista el tudja tárolni a leszármazott példányait is
  - típusazonosítás segítségével leválogathatjuk a speciális elemeket a listából, pl. az oktatók neveinek lekérdezéséhez
  - a főprogramban lehetőségünk van egy közös listában kezelni a hallgatókat és az oktatókat

Objektumorientált alkalmazások
Példa #4

Tervezés:



Objektumorientált alkalmazások
Példa #4

UniversityWithPolymorphism

Objektumorientált alkalmazások
Működés felüldefiniálás

- Ahhoz, hogy egy leszármazott osztály megváltoztathassa ősének egy viselkedését (metódus, vagy tulajdonság esetén), felül kell definiálnia a megfelelő metódust
  - az ősben jelezni kell, hogy megengedjük a felüldefiniálást, a felüldefiniálható metódusokat nevezzük *virtuális metódusok*nak
  - adhatunk olyan tagokat is, amelyeket a leszármazottban kötelező felüldefiniálni, ezeket *absztrakt* (vagy *tisztán virtuális*) *metódusok*nak nevezzük, ezeknek az ősben nem adunk törzset, csak deklarációt
    - absztrakt tag emiatt csak absztrakt osztályban szerepelhet, így csupán a hívási felület megadására szolgál

Objektumorientált alkalmazások
Működés felüldefiniálás

- A felüldefiniálható tagot a **`virtual`** kulcsszóval kell jelölnünk, míg az absztrakt tagot az **`abstract`** kulcsszóval, és ekkor az osztályt is absztraktként kell definiálnunk
  - a kulcsszó az összes leszármazott osztályra vonatkozik, nem csak a gyerek osztályra
  - osztálydiagramban dőlt betűvel jelöljük
- A felüldefiniáló tagot az **`override`** kulcsszóval kell jelölnünk
- A nem felüldefiniálható tagokat nevezzük *véglegesített*nek, vagy *leazár*tnak (*sealed*) nevezzük
  - alapértelmezetten minden tag lezárt, mezők mindig lezártak
  - amennyiben felüldefiniálásnál véglegesíteni akarunk, akkor a **`sealed`** kulcsszót kell alkalmaznunk az **`override`** mellett

Objektumorientált alkalmazások
Működés felüldefiniálás

- Pl.:

```
class BaseClass {
   public virtual void PrintValue() {
      // felüldefiniálható metódus
      Console.Write(1);
   }
}
class DerivedClass : BaseClass {
   public override void PrintValue() {
      Console.Write(2); // felüldefiniálás
   }
}
BaseClass bc = new DerivedClass();
bc.PrintValue(); // eredménye: 2
```


Objektumorientált alkalmazások
Működés felüldefiniálás

- Pl.:

```
abstract class BaseClass {
   public abstract void PrintValue();
   // a művelet absztrakt, emiatt az osztály is
}
class DerivedClass : BaseClass {
   public override void PrintValue() {
      Console.Write(2); // felüldefiniálás
   }
}

BaseClass bc = new DerivedClass();
// a művelet meghívható, mert ismert a szintaxisa
bc.PrintValue(); // eredménye: 2
```


Objektumorientált alkalmazások
Teljes származtatási hierarchia

- A C# programozási nyelv úgynevezett teljes származtatási hierarchiát épít fel, ez azt jelenti, hogy minden osztály egy egyetemes ősosztály (**`Object`**) valamilyen szintű leszármazottja
  - az ős definiálja az alapértelmezett viselkedést, pl. értékadás, egyenlőség lekérdezés, szöveggé alakítás
  - ezek a műveletek többnyire virtuálisak, ezért a leszármazott osztályok felüldefiniálhatják a különböző általános műveleteket, pl. a szöveggé alakítást (**`ToString`**)
  - amennyiben az új osztálynak nem adunk ős típust, automatikusan az egyetemes ős lesz az ősosztály
  - ennek köszönhetően az osztályok ábrázolhatóak egy úgynevezett öröklődési fában, aminek a gyökere az **`Object`**

Objektumorientált alkalmazások
Teljes származtatási hierarchia

- pl.:

```
class AnyClass {
   public AnyClass() {}
   public override string ToString() {…}
}
```

  jelentése igazából:

```
class AnyClass : Object {
   publuc AnyClass() : base() {}
   public override string ToString() {…}
}
```

- az érték szerinti osztályok is leszármazottak (ősük a **`ValueType`**), valamint a felsorolási típusok is (ősük az **`Enum`**), így szintén a az **`Object`** leszármazottai

Objektumorientált alkalmazások
Interfészek

- A többszörös öröklődés tiltott, ennek feloldására lehetőségünk van olyan osztályokat definiálni, amelyek csak publikus absztrakt tagokat (tulajdonságokat és metódusokat) tartalmaznak, ezeket nevezzünk *interfész*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 a megkülönböztethetőség érdekében konvencionálisan I-vel kezdjük
  - egy osztály tetszőleges sok interfészt valósíthat meg, az interfész pedig megvalósíthat más interfészeket is

Objektumorientált alkalmazások
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
}
```


Objektumorientált alkalmazások
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);
   }
}
```


Objektumorientált alkalmazások
Interfészek

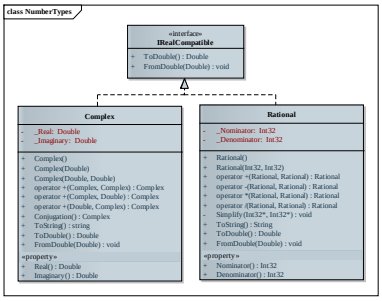
```
class DerivedClass : BaseClass,
                     IValuePrinter,
                     IAllValuePrinter {
   // öröklődés és több interfész megvalósítása
   // egyszerre
   // a PrintIntValue művelet már megvan az
   // öröklődésnek köszönhetően, csak a többi kell
   …
   // interfész megvalósítás: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


Objektumorientált alkalmazások
Példa #5

- 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

Objektumorientált alkalmazások
Példa #5

Tervezés:



Objektumorientált alkalmazások
Példa #5

NumberTypesWithInterface