Eötvös Loránd Tudományegyetem
Természettudományi Kar

Alkalmazott Modul III

6. előadás

Objektumorientált programozás: öröklődés és polimorfizmus

© 2011.10.24. Giachetta Roberto
groberto@inf.elte.hu
http://people.inf.elte.hu/groberto

Öröklődés

Kódismétlődés objektum-orientált szerkezetben

- Az objektum-orientált programokban az osztályok definiálják az objektumok működési sémáját
  - sok esetben hasonló, de mégis eltérő viselkedésre van szükségünk különböző objektumoktól, ekkor azoknak külön osztályt kell definiálnunk akkor is, ha sok ismétlődés előfordul
  - emiatt *kódismétlődéssel* kell szembenéznünk, amelyet a procedurális programokban kódrészlet kiemeléssel, alprogramokkal oldottunk fel, itt is valamilyen hasonló eszközhöz kell folyamodnunk
  - pl.: az egyetemi oktató és az egyetemi hallgató is tartalmaz nevet, azonosítót, kurzuslistát, és vannak ugyanolyan műveleteik

Öröklődés

Osztályok közötti hasonlóságok

- Az osztályok között felfedezhetők az ismétlődések, amelyeknek két esete van:
  - két, vagy több osztály közös tagokkal rendelkezik, ekkor célszerű lenne a közös részt kiemelni
    - pl. a háromszög és a négyszög is rendelkezik pontokkal, kerülettel és területtel
  - egy osztály rendelkezik egy másik osztály valamennyi tagjával, és ezen felül továbbiakkal, ekkor mondhatjuk azt, hogy az osztály *speciális esete* a másiknak (illetve a másik *általános esete* ezen osztálynak)
    - pl. a háromszög és a négyszög speciális esete a sokszögnek, és minden sokszög rendelkezik kerülettel és területtel

Öröklődés

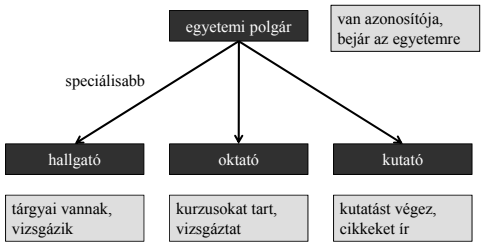
Általánosítás és specializáció

- Az első eset kombinálható a másodikkal, hiszen ha több osztálynak közös tulajdonságai vannak, akkor van olyan általánosabb osztály, amely azokat a tulajdonságokat tartalmazza
  - pl. az egyetemi hallgató és oktató közös tulajdonságait az egyetemi polgár osztály tartalmazza, így azok ennek speciálisabb változatai lesznek
- Egy osztálynak tehát lehet egy, vagy több speciálisabb változata, amely mindent tud, amit az általánosabb, és ezen felül még kiegészítheti tagjait tetszőleges számban
  - fordítva, egy osztálynak lehet egy (esetekben akár több) általánosabb változata, amelytől átveszi a viselkedést

Öröklődés

Általánosítás és specializáció

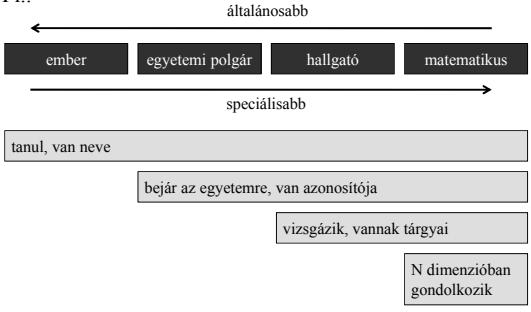
- Pl.:



Öröklődés

Általánosítás és specializáció

- Pl.:

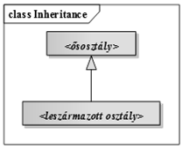


Öröklődés
Fogalmak

- A folyamatot, amelyben a speciálisabb osztály átveszi az általánosabb viselkedését, jellemzőit, *öröklődés*nek, vagy *származtatás*nak (*inheritance*) nevezzük
  - az öröklődés két irányát *specializáció*nak, illetve *általánosítás*nak nevezzük
  - az öröklődés lehet többszintű, a speciálisabb osztálynak lehetnek még speciálisabb osztályai
  - az általánosabb objektumot *ősosztály*nak (*base class*), a speciálisabbat *leszármazott*nak (*subclass*, *derived class*) nevezzük
  - ha csak egy szint a különbség, akkor *szülő*nek (*parent class*), illetve *gyerek*nek (*child class*) nevezzük az osztályokat

Öröklődés
Jelölés

- Az osztálydiagramban az öröklődést megfelelő relációval jelölhetjük:



- Az implementációban az osztályoknál kell jelölünk úgy, hogy megadjuk, mely ősosztályból származtatunk

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


Öröklődés
Működése

- Az öröklődés csak referenciaosztályokra (`class`) használható
- Minden tagot átvesz a leszármazott osztály, így azokat nem kell újra definiálnunk, és közvetlenül elérhetjük őket a leszármazott osztályban, illetve annak példányaiban
- Pl.:

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

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


Öröklődés
Működése

```
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
   }
}
```


Öröklődés
Működése

```
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
```


Öröklődés
Elérés és láthatóság

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

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

- Öröklődés esetén a tagok láthatóságát az eddigi 2 helyett 3 szinten szabályozhatjuk, a rejtett (`private`) láthatóság ugyanis a leszármazottak számára sem biztosít láthatóságot a tagra, ezért bevezetjük a védett (`protected`) láthatóságot
  - kívülről rejtett, de a leszármazott osztályokon belül látható tulajdonság, így közvetlenül tudunk hivatkozni rájuk
  - az osztálydiagramban # jellel jelölhetjük

Öröklődés
Rejtett láthatóságú tagoknál

- Pl.:

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

   public Int32 IntValue {
      get { return _IntValue; }
      set { _IntValue = value; }
   }
   public void BaseClass() { _IntValue = 0; }
} // az osztályon belül használható, kívül nem

class DerivedClass : BaseClass { // leszármazott
   public DerivedClass () { IntValue = 1; }
} // csak a tulajdonságon keresztül érjük el
```


Öröklődés
Védett láthatóságú tagoknál

- Pl.:

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

   public Int32 IntValue {
      get { return _IntValue; }
      set { _IntValue = value; }
   }
   public void BaseClass() { _IntValue = 0; }
} // az osztályon belül használható, kívül nem

class DerivedClass : BaseClass { // leszármazott
   public DerivedClass () { _IntValue = 1; }
} // közvetlenül is elérhetjük
```


Öröklődés
Konstruktor és destruktor művelet

- A konstruktor automatikusan öröklődik
  - a paraméteres nélküli konstruktor automatikusan meghívódik amikor a leszármazottból létrehozunk egy példányt
    - elsőként az ős konstruktora hívódik meg, aztán a leszármazotté
  - lehetőségünk van az ős konstruktorának explicit meghívására is **`<konstruktor>(<paraméterek>) : base(<átadott paraméterek>)`** formában
  - paraméteres konstruktorokra csak az explicit hívás használható
- A destruktor automatikusan öröklődik és hívódik meg

Öröklődés
Konstruktor és destruktor művelet

- 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
}
```


Öröklődés
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 konstruktornak nem publikus láthatóságot adunk

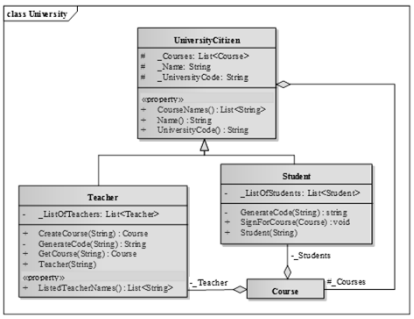
Öröklődés
Példa

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

Öröklődés
Példa

Tervezés:



Öröklődés
Példa

Megoldás:

```
abstract class UniversityCitizen { // polgár
   protected String _Name; // védett tagok
   protected String _UniversityCode;
   protected List<Course> _Courses;

   public String Name { get { return _Name; } }
   …
}

class Student : UniversityCitizen {
   // egyetemi hallgató, speciális osztály
   public Student(String name){ … }
   …
```


Öröklődés
Viselkedés elrejtés

- A leszármazott amellett, hogy örököl minden tagot az őstől, lehetősége van *elrejteni* az örökölt viselkedést, és újat definiálni
  - azaz lehetőségünk van ugyanolyan szintaxisú 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

Öröklődés
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
```


Polimorfizmus
Objektumok típusai

- Sokszor előfordul, hogy az általános osztály speciális eseteit szeretnénk együtt kezelni, pl. egy listában eltárolni a hallgatókat és oktatókat
- Objektumorientált szerkezetben az objektumok típusa az osztály, amiből példányosítjuk őket, azonban öröklődés esetén az objektum típusának az osztály bármely őse is tekinthető
  - hiszen annak viselkedését, értékeit birtokolja, és a konstruktora is lefut
  - ez az implementációban úgy jelenik meg, hogy egy általános osztályú hivatkozást ráállíthatunk egy speciális típusra is, pl.:

```
BaseClass bc = new DerivedClass();
```


Polimorfizmus
Statikus és dinamikus típus

- Ennek megfelelően egy objektumnak több alakja is lehet, így a jelenséget *többalakúság*nak, vagy *polimorfizmus*nak nevezzük
  - pontosabban altípusos polimorfizmusnak (*subtype polymorphism*)
- Polimorfizmus esetén két típust kell nyilvántartanunk:
  - a változó *statikus típusa* a mutató típusa, a fordítóprogram ezt tudja értelmezni
    - emiatt a változó csak ennek a típusnak a tagjai hívhatóak meg, mivel azok jelenléte garantált
    - lehet absztrakt osztály is, hiszen nem kerül példányosításra

Polimorfizmus
Statikus és dinamikus típus

- a változó *dinamikus típusa* a ténylegesen létrehozott objektum típusa
  - lehet a statikus típus, vagy annak bármely leszármazottja, nem lehet absztrakt osztály
  - ténylegesen az ő tagjai kerülnek meghívásra, azaz futás közben az ő viselkedése a meghatározó
- Pl.:

```
class BaseClass {
   // őszosztály egy egész értékkel
   public Int32 IntValue;
   public BaseClass() { IntValue = 1; }
}
```


Polimorfizmus
Statikus és dinamikus típus

```
class DerivedClass : BaseClass {
   // leszármazott egy egész és egy valós értékkel
   public Int32 FloatValue;
   public DerivedClass() {
      FloatValue = 2; IntValue = 2;
      // a konstruktor más értéket rendel hozzá
   }
}

BaseClass bc = new DerivedClass();
Console.WriteLine(bc.IntValue);
// eredménye: 2
Console.WriteLine(bc.FloatValue);
// hiba: nincs ilyen mező
```


Polimorfizmus
Adatszerkezetbe szervezés

- Mivel a változó által hivatkozott objektumot változtathatjuk a program során, a dinamikus típus is változhat, pl. :

```
BaseClass bc;
bc = new DerivedClass();
// most egy leszármazott objektumra mutat
bc = new BaseClass();
// most egy ős objektumra mutat
```

- A többalakúság lehetőséget ad, hogy egy adatszerkezetbe különböző típusú elemek kerüljenek, amelyek közös őssel rendelkeznek, pl.:

```
List<BaseClass> bcList = new List<BaseClase>();
bcList.Add(new DerivedClass());
bcList.Add(new BaseClass());
bcList.Add(new DerivedClass());
```


Polimorfizmus
Osztályazonosítás és megfeleltetés

- Ugyanakkor 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);
```

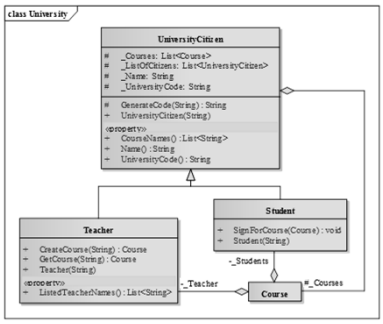

Polimorfizmus
Példa

Feladat: 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

Polimorfizmus
Példa

Tervezés:



Polimorfizmus
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 * / }
}
```


Polimorfizmus
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");
   }
   …
```


Polimorfizmus
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

Polimorfizmus
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 *leazárt*nak (*sealed*) nevezzük
  - alapértelmezetten minden tag lezárt, mezők mindig lezártak
  - amennyiben felüldefiniálásnál véglegesíteni akarunk, akkor a **`sealed`** kulcsszót kell alkalmaznunk az **`override`** mellett

Polimorfizmus
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
```


Polimorfizmus
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
```

Polimorfizmus
Dinamikus kötés

- Felüldefiniált metódus esetén polimorfizmus alkalmazása mellett nem a statikus, hanem a dinamikus típus szerint fut le a művelet, mivel a program futási idő alatt azonosítja be a futtatandó műveletet, ezt *dinamikus kötés*nek (*dynamic binding*) nevezzük
  - elrejtés esetén nem használható, akkor a statikus típus szerint fut le a művelet
  - akár az ősosztály nem felüldefiniált metódusában is meghívhatunk absztrakt műveletet, amit felüldefiniálunk, ekkor ugyanúgy a felüldefiniáló művelet fut le a leszármazott példányban
  - konstruktorra és destruktorra nem alkalmazható, mivel ott valamennyi lefut

Öröklődés
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`

Öröklődés
Teljes származtatási hierarchia

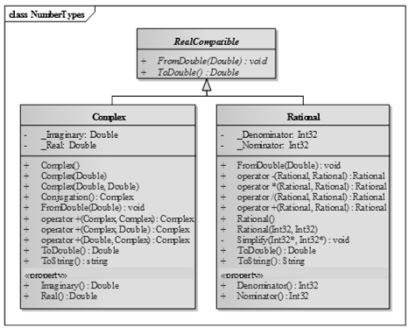
- pl.:
```
class AnyClass {
   public AnyClass() {}
   public override string ToString() {…}
}
```
jelentése igazából:
```
class AnyClass : Object {
   publuc AnyClass() : base() {}
   public override string ToString() {…}
}
```
- az érték szerinti osztályok is leszármazottak (ősük a `ValueType`), valamint a felsorolási típusok is (ősük az `Enum`), így szintén a az `Object` leszármazottai

Öröklődés
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 absztrakt osztályt, ami a valóssá alakítás, illetve valósról átalakítás absztrakt metódusait tartalmazza
- a leszármazott osztályokban ezt felüldefiniáljuk, így a dinamikus kötés segítségével 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

Öröklődés
Példa

Tervezés:



Öröklődés
Példa

Megoldás:
```
abstract class RealCompatible { // absztrakt ős
   public abstract Double ToDouble();
   public abstract void FromDouble(Double val);
   // csak absztrakt műveleteket írunk
}

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

Öröklődés
Többszörös öröklődés

- Általában egy szülőnek több gyereke is van, ám csak egyes esetekben engedélyezett, hogy egy gyereknek több szülője is legyen, ezt *többszörös öröklődés*nek (*multiple inheritance*) nevezzük
  - a gyerek megkaphatja minden szülője minden tulajdonságát, ami a leszármazottban ütközéshez (főként a mezőkben), vagy felesleges adattároláshoz vezethet, ezért a programozási nyelvek rendszerint nem engedélyezik
  - pl. egy négyzet lehet a rombusznak, valamint a téglalapnak is speciális esete, ugyanakkor teljesen felesleges lenne a téglalap különböző oldalainak, illetve a rombusz szögeinek eltárolása a négyzetben

Interfészek
A többszörös öröklődés feloldása

- A többszörös öröklődés tehát gondot jelenthet a mezőütközések miatt, amiatt a legtöbb nyelvben nem engedélyezett, viszont sokszor előfordul, hogy az osztályok viselkedését több absztrakt osztály szerint is meg akarjuk határozni
  - pl. a matematikus hallgató magában hordja a matematikus, és az egyetemi hallgató viselkedését is
  - a többszörös öröklődés nem jelent gondot olyan osztályok esetében, amelyek nem tartalmaznak mezőt, csak virtuális, vagy absztrakt tagokat
- Lehetőségünk van olyan osztályokat definiálni, amelyek csak publikus absztrakt tagokat (tulajdonságokat és metódusokat) tartalmaznak, ezeket nevezzünk *interfészek*nek

Interfészek
Működése

- Interfészeket az `interface` kulcsszóval kell jelölnünk, és azt mondjuk, hogy az *osztály megvalósítja az interfészt* (ezt szaggatott vonallal jelöljük az osztálydiagramban)
  - az interfészben minden publikus, és absztrakt, ezért nem is írjuk ki a kulcsszavakat, felüldefiniáláskor sem
  - az interfészek elnevezését a megkülönböztethetőség érdekében I-vel kezdjük
  - egy osztálynak csak egy szülője lehet, de tetszőleges sok interfészt valósíthat meg, az interfésznek nem lehet őse, de megvalósíthat más interfészeket is
  - mivel minden tag absztrakt, ezért nem jelent problémát a szintaxis ütközés, mivel úgyis felül kell definiálni a metódusokat

Interfészek
Érték szerinti típusoknál

- 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
}
```


Interfészek
Referencia szerinti típusoknál

- 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);
   }
}
```

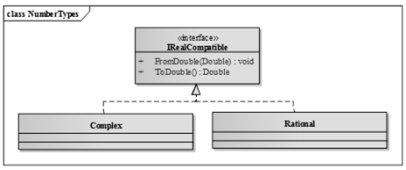

Interfészek
Referencia szerinti típusoknál

```
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() {…}
}
```

Interfészek
Példa

Feladat: A számkezelő osztályok őse igazából egy interfész, mivel csak absztrakt műveleteket tartalmaz, ezért megfogalmazhatjuk interfészként.

Tervezés:



Interfészek
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) {…}
}
```