Objektumorientált programozás

Rekord adatszerkezet – mint "előzmény"

- A rekord olyan összetett adatszerkezet, amely több, akár különböző típusú adatot foglal egységbe.
- Egy felhasználói típus, amely típusból aztán változókat deklarálhatunk
- A rekordnak neve van Pl: Tanulo
- A rekord mezőkből épül fel
  - A mezőknek a rekordon belül egyedi neve van
  - A mezők típusa tetszőleges

A rekord továbbfejlesztése:

- Ne csak a logikailag összetartozó adatokat zárjuk egységbe, hanem az adatokon dolgozó metódusokat is
- Ez lesz az osztály (class) – ami az objektumorientált programozás alapegysége
- A C#-ban egy másik hasonló, rekord-szerű szerkezet a struct → a különbségek majd később…

Az OOP története

- Előzmény: szoftverkrízis
  - 1968. NATO szoftverkonferencia
- Alan Kay – 1969-1972.
  - Smalltalk nyelv
  - Alapelvek kidolgozása
- A válságból kivezető út: az objektumorientált szemléletmód
  - Bonyolultság kezelése
  - „Biztonságos” kód
  - Újrafelhasználhatóság

Ami nem változott:

- Minden olyan program, ami megírható OO szemléletben, megírható hagyományos, strukturált programozási szemléletben is.
- Az OO program is a megismert vezérlési szerkezeteket használja a függvények törzsében.
- Az OO programok nem futnak gyorsabban.

Mitől jobb?

- A program jobban áttekinthető egységekből áll.
- Az egységek tesztelésével programunk hibátlan működése jobban biztosítható.
- Az adataink értékére vonatkozó bonyolultabb garanciák, invariánsok is fenntarthatók.
- A sokfős csoportmunka lehetősége.
- Újrafelhasználható kód.

OOP versus hagyományos nyelv

- Az OOP elvek használata mellett az eljárás-orientált nyelvek minden lehetősége lefedhető.

- Az OOP programozás biztonságosabb, kevesebb hibalehetőség a hagyományos programozással szemben

Programozási nyelvek osztályozása:

- **Hagyományos (eljárás-orientált)**:
  - nem alkalmazza az OOP, csak az eljárás-orientált programozási nyelvek elveit
  - Pl: Pascal, C.
- **OOP támogató:**
  - Alkalmazza az eljárás-orientált és az OOP elveket is
  - Pl: Delphi, C++
- **Tisztán OOP** :
  - nem alkalmazza az eljárás-orientált elveket, csak az OOP elveit
  - Pl: Java, C#

OOP alapelvek:

1. Egységbezárás (encapsulation)
2. Öröklődés (inheritance)
3. Polimorfizmus/sokalakúság (polymorphysm)

1. Alapelv: Egységbezárás

Az egységbezárás azt jelenti, hogy az összetartozó adatokat, és a hozzájuk tartozó tevékenységeket (műveleteket) egyetlen szerkezeti egységben fogjuk össze.

Alapfogalmak: osztály

Objektumosztály vagy egyszerűen osztály (class):

A valós világ elemeinek programozási modellje:

Összetartozó adatok, és a rajtuk dolgozó függvények, eljárások egysége.

- Adatokat tárol az adattagokban (data member) /mezőkben (field) / tagváltozókban
- Tevékenységeket végez az osztály metódusain/tagfüggvényein (function member) keresztül (amikor meghívjuk a metódust – ezt szokás üzenet küldésnek, kérésnek is nevezni)
- Például:
  - Objektumosztály: autó
  - Adatok: rendszám, tank mérete, gyártási év, tank töltöttség
  - Tevékenységek: tankol

Osztály deklaráció

```
class Tanulo
{
    string nev;
    int szulev;
    int evfolyam;
    char osztaly;
    double atlag;
}
```

Konvenció: az osztálynév nagybetűvel kezdődik.

Alapfogalmak: objektum

Az Objektumosztály (Class) tehát egy modell, egy terv, ami alapján az **osztály példányait (instance)**, **objektum**okat tudunk létrehozni, azaz **példányosítani**.

Példányosítás:

```
Tanulo t = new Tanulo();
```

- ⇨ A `new` végzi a helyfoglalást a memóriában és a típusra jellemző null-értékek beállítását, illetve a mező deklarációkor megadott kezdőértékek beállítását
- ⇨ A `Tanulo()` az osztály konstruktora, amely a példány alaphelyzetbe állítását végzi

Alapfogalmak: példányosítás

- A példányosításkor történik meg az adattagok számára a **helyfoglalás** a memóriában
- valamint az **inicializálás**, az objektum alaphelyzetbe állítása: a mezők kezdőértékeinek beállítása

Felelősség elve:

Egy objektum felelős azért, hogy az inicializálás után máris megfelelő értékkel rendelkezzenek a mezői és később se kerülhessen a mezőibe hibás érték.

A fordítóprogram csak a mező alaptípusát ellenőrzi értékadáskor!

Adatrejtés elve:

- az objektumoknak el kell rejteni a külvilág felől az adataikat
- az objektum adatait csak az objektum saját eljárásai (metódusai) kezeljék
- az objektumok adatai így az objektum „tudta" nélkül nem változtathatók meg

Védelmi szintek:

Mezőknek, és metódusoknak is van

- **Private** – az alapértelmezett védelmi szint, a legszigorúbb
  - ↳ Csak az osztályon belülről elérhetők
- **Protected**
  - ↳ Csak az osztályon belülről és a leszármazott osztályokból elérhetők
- **Public**
  - ↳ A program tetszőleges részéből elérhetők

A konstruktor

Egy speciális metódus:

- Neve megegyezik az osztály nevével
- Csak a `new` szó után hívható meg
- Nincs jelölt visszatérési típusa, még void sem!
- Minden osztálynak kell, hogy legyen konstruktora
- Az alapértelmezett konstruktor egy üres konstruktor
- **Ha nem írunk konstruktort**, **akkor a fordító hozzáadja az alapértelmezett konstruktort** az osztályhoz

Adattagok

⇨ Példányszintű mező

[védelmi szint] <típus> <mezőnév> [=kezdőérték]

⇨ Osztályszintű mező

static [védelmi szint] <típus> <mezőnév> [=kezdőérték]

felcserélhető

⇨ Konstans mező

const [védelmi szint] <típus> <mezőnév> [= kezdőérték]

felcserélhető

Példányszintű mezők

- Példányonként eltérő értékeket tartalmazhatnak
- A memóriában kezdetben 0 db van
- A példányosításkor a new operátor foglalja le a helyet számukra
- Hatókörük a védelmi szinttől függ
- Élettartamuk a példány élettartamától függ (a GC hatásköre)
- Hivatkozás a mezőkre:
  - Osztályon belül: mezőnév vagy this.mezőnév
  - Másik osztályból: példánynév.mezőnév

Osztályszintű mezők

- Értékük nem változik példányonként, az egész osztályra jellemző
- Hatókörük a védelmi szinttől függ
- Élettartamuk statikus: a program indulásakor bekerülnek a memóriába és végig ott maradnak
- Egy van a memóriában akkor is, ha sok példány van, akkor is, ha nulla példány van – nem a GC hatáskörébe tartoznak
- Hivatkozás a mezőkre:
  - Osztályon belül: mezőnév vagy osztálynév.mezőnév
  - Másik osztályból: osztálynév.mezőnév

Property - problémafelvetés

- Felelősség kérdése: **a mezőbe csak a megengedett értékek kerülhessenek**
- A fordító minden, a mező típusának megfelelő értéket „engedélyez”
- Mi van, ha ennél szigorúbb megkötéseink vannak?
- Ha egy mezőt public-ra állítunk, akkor az bárhonnan olvasható és írható
- Előfordulhat, hogy az olvasás/írás közül csak az egyiket akarnánk engedélyezni kívülről – **csak írható/csak olvasható mezők megvalósítása**

Property – mire jó?

- A mezők védelmére szolgál
- Virtuális mező, „mezőnek látszó izé”
- A property lehetővé teszi, hogy a mezőkhöz csak metódusokon keresztül férhessünk hozzá
- Formailag egyszerűbb, mint a metódusok használata: nincs szükség ( ) –re
- Használatkor úgy néz ki, mintha egy mezőt használnánk
- Egyetlen egységen belül lehet megvalósítani a mező olvasását és/vagy írását lehetővé tevő metódust

Property létrehozása

```
class Osztaly
{
        private típus _mezőnév;

        public típus mezőnév
        {
            get
            {
                return _mezőnév
            }
            set
            {
                _mezőnév = value;
            }
        }
 }
```

Property részei

- A **get** rész szolgál a kiolvasásra
- A **set** rész szolgál az értékadásra
- Jellemzően a set részben feltételt is szabunk, nem megfelelő érték esetén kivételt dobunk
- Nem kell mindkét résznek szerepelnie:
  - **Csak olvasható mező**: csak a get rész van
  - **Csak írható mező**: csak a set rész van
- Lehetséges olyan property is, amely mögött nincs tárolómező, hanem a kiolvasandó értéket a property törzsében generáljuk

Property és kettős védelmi szint

- Ha a property mindkét része egyforma védelmi szinttel rendelkezik, azt a külső részen kell deklarálni, a property típusa előtt
- Eltérő védelmi szintek esetén az egyik védelmi szintet kívül, a másikat belül kell megadni
- Kívül kell megadni a megengedőbb védelmi szintet, és belül kell szigorítani

Példa kettős védelmi szintre:

```
private int _kor;

public int kor
{
    get
    {
        return _kor;
    }

    protected set
    {
        if (value < 18 || value > 60)
            throw new ArgumentException("csak 18...60 lehet a kor");
        else
            _kor = value;
    }
}
```

2. Alapelv: Öröklődés

- Új objektumosztály fejlesztésekor egy már meglévőt fel tudunk használni – ha **ősként jelöljük meg**.
- Az öröklődés során átvesszük az ős objektumosztály **mezőit** és **metódusait**.
- Az osztály **bővíthető** új mezőkkel és metódusokkal.
- Az örökölt metódusok **módosíthatók**.
- A gyerek osztály **nem dobhat el elemeket** az ősosztályból.

Öröklődés – elnevezések:

- Amiből származtatunk:
  - **ősosztály** (base class) *VAGY*
  - **szülő osztály** (parent class) *VAGY*
  - super class
- A leszármazott:
  - **származtatott osztály** (derived class) *VAGY*
  - **gyerek osztály** (child class)

Öröklődés és védelmi szintek

- Az ősosztályban **private** hozzáférésű mezők **nem elérhetők** a gyerekosztályban (de öröklődnek, tehát helyet foglalnak a memóriában)
- Az ősosztályban **protected** hozzáférésű mezők **elérhetők a gyerekosztályokban**, de nem elérhetők más osztályokból – OOP-ban ez a leggyakoribb védelmi szint
- Az ősosztályban **public** hozzáférésű mezők **mindenhonnan elérhetők**, tehát a gyerekosztályokból is.

Öröklődés deklarálása

```
class Negyzet
{
    protected double _a;

}

    class Teglalap: Negyzet
{
    protected double _b;
}
```

//a Negyzet osztálynak 1 mezője van

//a Teglalap osztálynak 2 mezője van

Mezők újradefiniálása

- A **new** szóval felülírhatunk egy örökölt mezőt egy ugyanolyan nevűvel - NE HASZNÁLJUK!
- Ekkor az ősosztálybeli mezőt a **base** kulcsszóval érhetjük el.

Például:

```
class Elso
{
  protected  int a = 1;

}

class Masodik:Elso
{
    protected double a = 2.2;

    public void Kiir()
    {
        Console.WriteLine(a);
        Console.WriteLine(base.a);
    }
}
```

//2.2-t ír ki
//1-et ír ki

Metódusok újradefiniálása a new kulcsszóval

- Ha egy ősosztálybeli **protected**, vagy **public** metódussal azonos nevű és paraméterezésű metódust szeretnénk írni: a **new** kulcsszóval megtehetjük
- Metódusok esetén is használható a **base** kulcsszó

Típuskompatibilitás

- ⇨ A gyerekosztály a szülőosztálytól minden mezőt, metódust, property-t örököl
- ⇨ A gyerekosztály példányai képesek helyettesíteni az ősosztály példányait
- ⇨ Azt mondjuk, hogy **a gyerekosztály típuskompatibilis az ősosztályával**
- ⇨ Minden osztály kompatibilis minden „felmenőjével” (nemcsak a közvetlen ősével)

Típuskompatibilitás, értékadás

Egy értékadó utasítás helyes, ha a bal oldalon álló változó típusa egyezik a jobb oldalon álló kifejezés típusával, vagy ezen kifejezés típuskompatibilis vele.

Példa:

```
class Elso...

class Masodik:Elso...

class Harmadik: Masodik...
```

```
Elso e = new Elso();
Masodik m = new Masodik();
Harmadik h = new Harmadik();

e = m;  //helyes!
e = h;  //helyes!

m = e; //szintaktikai hiba!

m = (Masodik)e; //nincs szintaktikai hiba, de futási hiba előjöhet
```

Statikus és dinamikus típus

- **Statikus típus**: egy változó deklarációkor megadott típusa
- **Dinamikus típus**: egy változóban valóban létrejött példány típusa

Példa:

```
Elso e=new Elso(); //e statikus és dinamikus típusa is Elso
Elso e=new Masodik(); //e statikus típusa Elso, dinamikus típusa Masodik
```

Korai és késői kötés

- **Kötés**: az a folyamat, amikor a fordítóprogram a metódus hívást egy konkrét metódussal kapcsolja össze
- **Korai kötés**: a döntés fordítási időben születik meg, a példány statikus típusa alapján → gyorsabb futási sebesség
- **Késői kötés**: a döntés futási időben, a példány dinamikus típusa alapján → lassabb futási sebesség

Késői kötés alkalmazása

- C#-ban alapértelmezett a korai kötés
- A new kulcsszóval való újradefiniálás korai kötést eredményez
- Ha késői kötést szeretnénk azt a **virtual-override** kulcsszó párral tudjuk jelezni a fordítónak (**virtuális metódusok**)

Virtual - override

- Az **ősosztályban**, ahol először definiáljuk a metódust: elé kell tenni a **virtual** szót.
- A virtual szót tehát egyszer használjuk.
- A **gyerekosztályokban**, ahol szeretnénk felülírni a metódust: az **override** szót tesszük a metódus elé. NEM KÖTELEZŐ, a new is használható.
- Az override szót többször is használhatjuk, de csak olyan metódusra, amely meg volt jelölve a virtual szóval.

Megjegyzések

- Private metódus, property nem lehet virtual
- Static metódus nem lehet virtual
- Az override során csak a metódus törzsét módosíthatjuk (nevét, visszatérési típusát, paraméterezését nem!)

Konstruktorok öröklődése

- ⇨ **A konstruktorok nem öröklődnek!**
- ⇨ Egy példány elkészítésében az ősosztályok konstruktorai is részt vesznek.
- ⇨ **Minden ősosztályból lefut egy konstruktor**, mégpedig az öröklődési lánc sorrendjében

  Elso →Masodik → Harmadik

Konstruktor azonosítási lánc

1. Példányosított osztály konstruktorának azonosítása
2. Ősosztály konstruktorának kiválasztása:

   Csak **public** vagy **protected** konstruktor lehet!

   2.1 **this()** saját másik konstruktor meghívása (ha van)

   2.2 **base()** ősosztálybeli konstruktor meghívása (ha van)

   2.3 **paraméter nélküli konstruktor** meghívása (ha van)

   2.4 **szintaktikai hiba**, ha az előző 3 alapján nem kerül megívásra ősosztálybeli konstruktor

Az Object osztály…

- ⇨ Ha nem jelölünk ki ősosztályt: a fordító egy **alapértelmezett ősosztály**t jelöl ki, ez az **Object** nevű osztály (alias neve object)
- ⇨ A gyerekosztályok kompatibilisek az ősosztályukkal → **a C#-ban minden osztály típuskompatibilis a System névtér Object osztályával**
- ⇨ Object típusú változókba bármilyen típusú értéket elhelyezhetünk

…Object osztály

- Az Object osztály **metódusai**, amit minden más osztály örököl:
  - **GetType()** : megadja az osztály nevét és a névteret, amelyben az osztály van
  - **ToString()** : string alakban adja meg az adott példányt
  - **Equals(x)** : megadja, hogy az adott példány és az x példány egyenlő-e
  - **GetHashCode()** : egy int értéket ad, mely a példányra jellemző
- Saját osztályaink esetén a ToString() metódus felüldefiniálható, mert a **ToString()** metódus **virtuális**

A Main() függvény…

- A Neumann-elvű számítógép **soros működésű**
- Az utasítások a forráskódban megadott sorrendben kerülnek végrehajtásra
- A metódusok definiálásánál a sorrendnek nincs jelentősége, csak a meghívásnak
- Kell egy **kezdőpont, ami megadja, hogy melyik a legelső végrehajtandó utasítás**, onnantól kezdve egyértelmű, hogy melyik a következő
- A program kezdőpontját a **C# nyelvben a Main() nevű függvény** jelzi

…Main() függvény

- A C#-ban minden függvényt osztályba kell helyezni
- Mindegy, hogy melyik osztályban van a Main() függvény
- Nem lehet példányszintű, azaz **kötelezően static**
- Védelmi szintje: mindegy
- Legegyszerűbb alakja:

```
static void Main()
{
}
```

…Main() függvény

- Visszatérési típusa: **void** vagy **int**
- int visszatérési típus esetén a visszaadott érték jelzi, hogy a program futása sikeres volt-e, vagy sem
- 0 érték jelöli, hogy nem volt hiba, minden más érték valamilyen hibára utal
- Paramétere lehet egy string tömb, amiben például fájl-neveket adhatunk meg

Névterek (namespace)…

- A létrehozott osztályok csoportosítására szolgál
- Minden névtérnek van neve, a szokásos névadási szabályoknak megfelelően
- A névtér neve az osztály nevét kiegészíti → **minősített név**
- Pl. `Elolenyek.Cica cicc = new Elolenyek.Cica();`

```
namespace Elolenyek
{
    class Allat { }
    class Haziallat : Allat { }
    class Cica : Haziallat { }
}
```

…Névterek

- Ha nem akarjuk a teljes minősített nevet használni, a forráskód elejére beírjuk:

```
using Elolenyek;
Cica cicc = new Cica();
```

- A using után csak a névtér nevét adhatjuk meg, az osztálynév már nem szerepelhet!
- A névterek egymásba ágyazhatók tetszőleges mélységben
- A névterek segítenek elkerülni a névütközési problémákat

Egymásba ágyazott névterek

A minősített nevek:

```
Elolenyek.Haziallatok.Kutya k = new Elolenyek.Haziallatok.Kutya();
Elolenyek.Haziasitott.Galamb g = new Elolenyek.Haziasitott.Galamb();
```

```
namespace Elolenyek
{
    namespace Haziallatok
    {
        class Allat { }
        class Kutya : Allat { }
        class Cica : Allat { }
    }

    namespace Haziasitott
    {
        class Allat { }
        class Galamb : Allat { }
    }
}
```