Eötvös Loránd Tudományegyetem
Természettudományi Kar

Alkalmazott Modul III

4. előadás

Objektumorientált programozás: objektumok és osztályok

© 2011.10.10. Giachetta Roberto
groberto@inf.elte.hu
http://people.inf.elte.hu/groberto

Procedurális programozás

Koncepció

- A *procedurális programozás* a felülről lefelé tervezés elvét követi, vagyis a feladatot részfeladatokra bontja
  - a részfeladatokhoz az imperatív megközelítés elvén megoldó *algoritmusok*at rendel
  - az algoritmusok a *programkonstrukció*k, vagy vezérlési szerkezetek (szekvencia, elágazás, ciklus) segítségével épülnek fel
  - a megoldó algoritmusok az algoritmus állapotterét alkotó *változók* segítségével kommunikálnak egymással
  - a változók változtatásával a program állapotot vált, és a program futása az *állapotváltozás*ok sorozata, amelyek elvezetnek a végállapothoz

Procedurális programozás

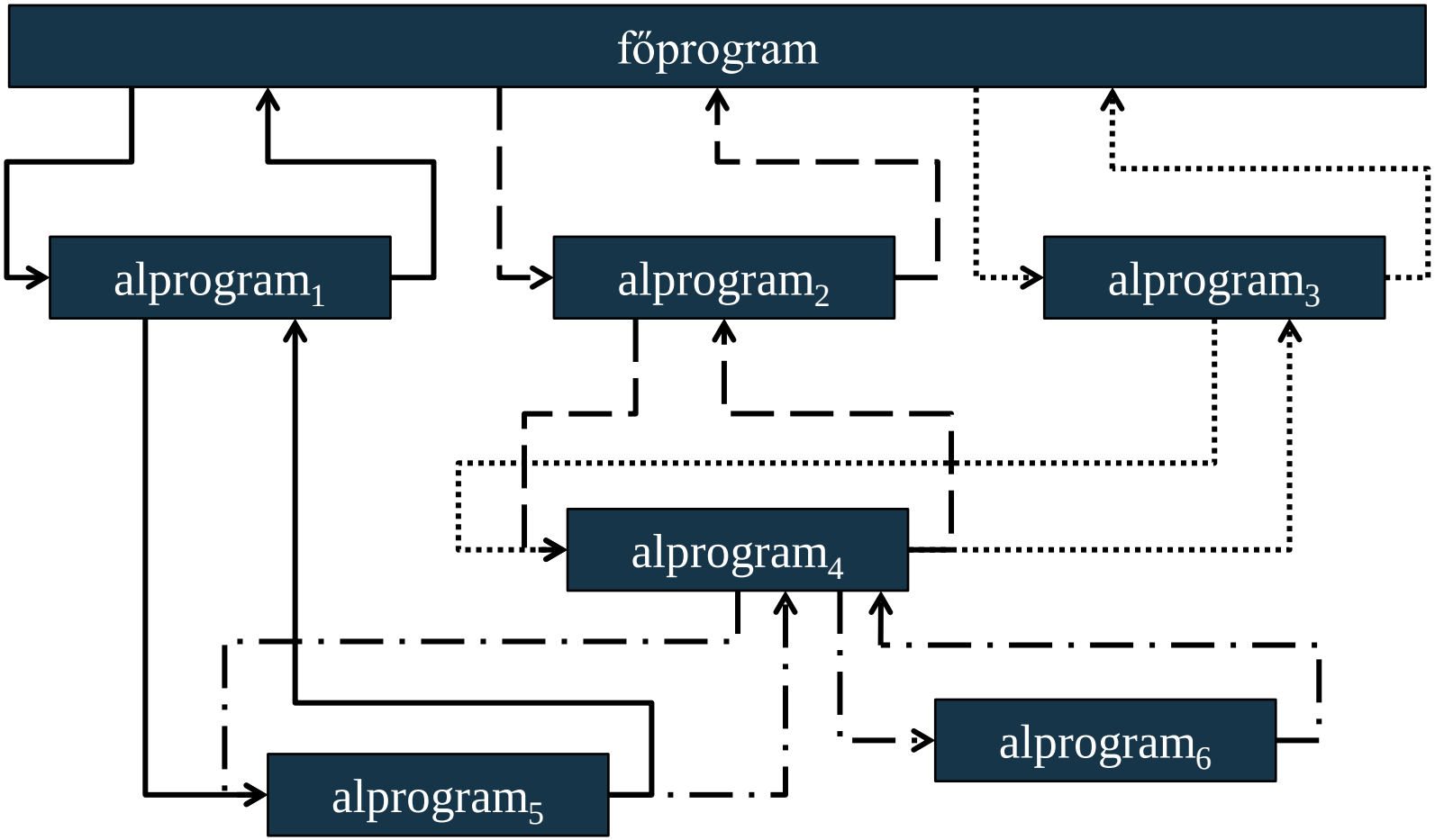
Alprogramok

- Az algoritmusok *alprogram*okként (*subroutine*) jelennek meg, és meghatározott részfeladatot végeznek el, lehetnek:
  - *eljárás*ok (*procedure*): valamilyen utasítássorozatot futtat, végeredmény nélkül
  - *függvény*ek (*function*): valamilyen matematikai számítást végez el, és megadja annak eredményét
- Alprogramok meghívhatnak más alprogramokat, és kommunikálhatnak velük adatok átadásával
- Van egy kiemelt szerepű alprogram, amelyet a program indítások futtat, ez a *főprogram*
  - feladata a teljes program vezérlése és az adatok összefogása, rá hárul a teljes felelősség

Procedurális programozás

Adatkezelés és vezérlés

- Az adatok a procedurális programozásban lehetnek:
  - *globálisak*: valamennyi alprogram számára láthatóak, és kezelhetőek, élettartamuk a program teljes futása
  - *lokálisak*: csak egy alprogram számára láthatóak, élettartamuk az alprogram futása
- A felső szintű alprogramok között áramló, illetve a program teljes működése során jelen levő adatokat a főprogram tárolja
- A lokális adatok áramlását az alprogramok között felügyelni, nyomon követni kell
  - a lokális adatok átvitele történhet *paraméter*ként, illetve *visszatérési érték*ként, amely során az adatok átmásolódnak


főprogram
alprogram1
alprogram2
alprogram3
alprogram4
alprogram5
alprogram6

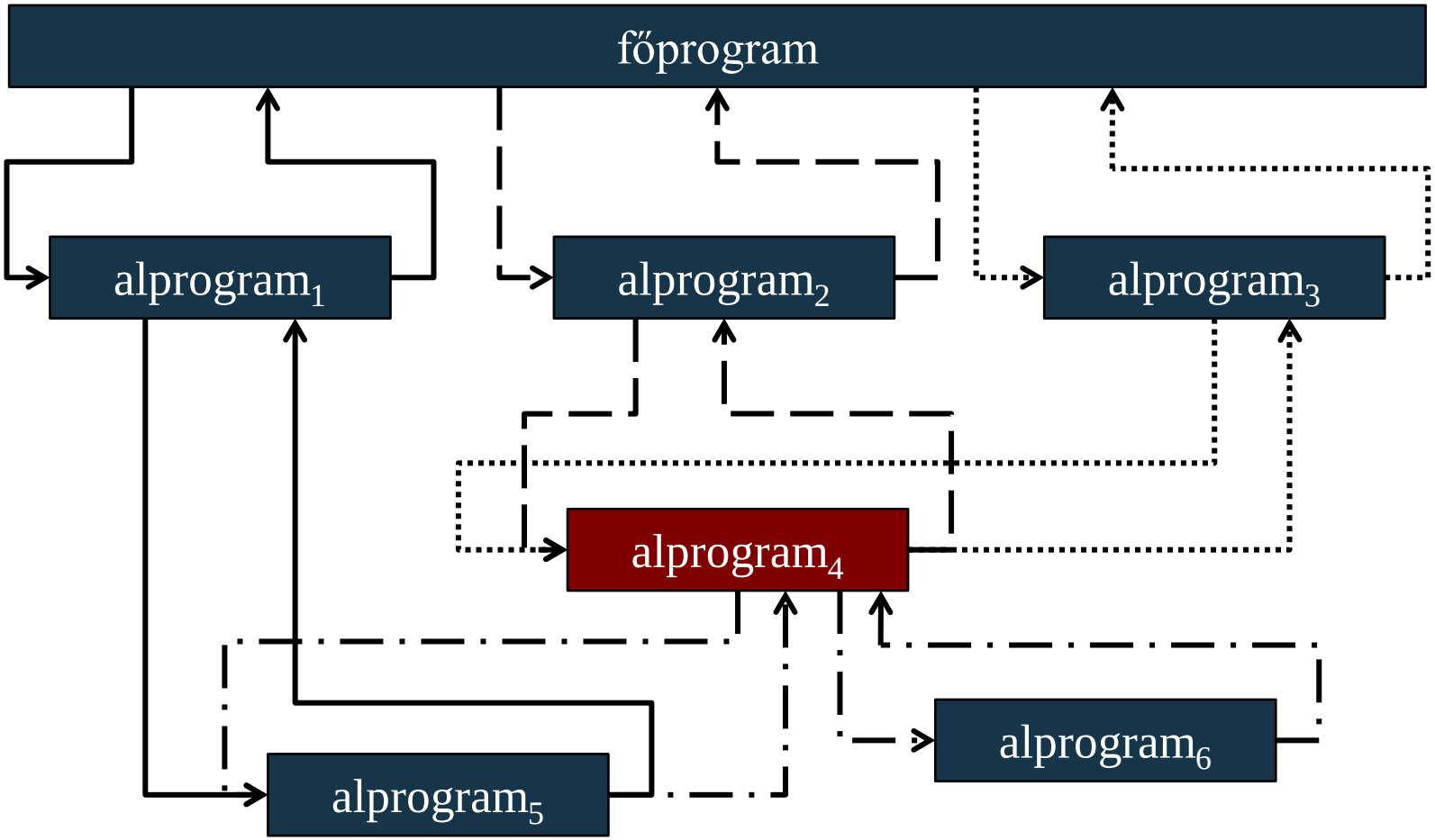
Procedurális programozás

Korlátok

- A programrészek csak az algoritmusok szerint különíthetőek el, aszerint nem, hogy melyik feladatkörben végzik tevékenységüket, tehát *nem tagolható* kellő mértékben a program
- Az *adatok élettartama* nem elég testre szabható, az adat vagy mindenki számára látható, vagy csak egy adott algoritmus számára, nem lehet programegység szerinti adatokat megkülönböztetni
- A *feladat módosítása* több funkció módosítását igényelheti, ami mind az alprogramokban, mind az adatkezelésben változásokat jelent, beleértve az általuk meghívott, és az őket meghívó alprogramokra is, tehát jelentős módosításokat kell végezni a programkódban

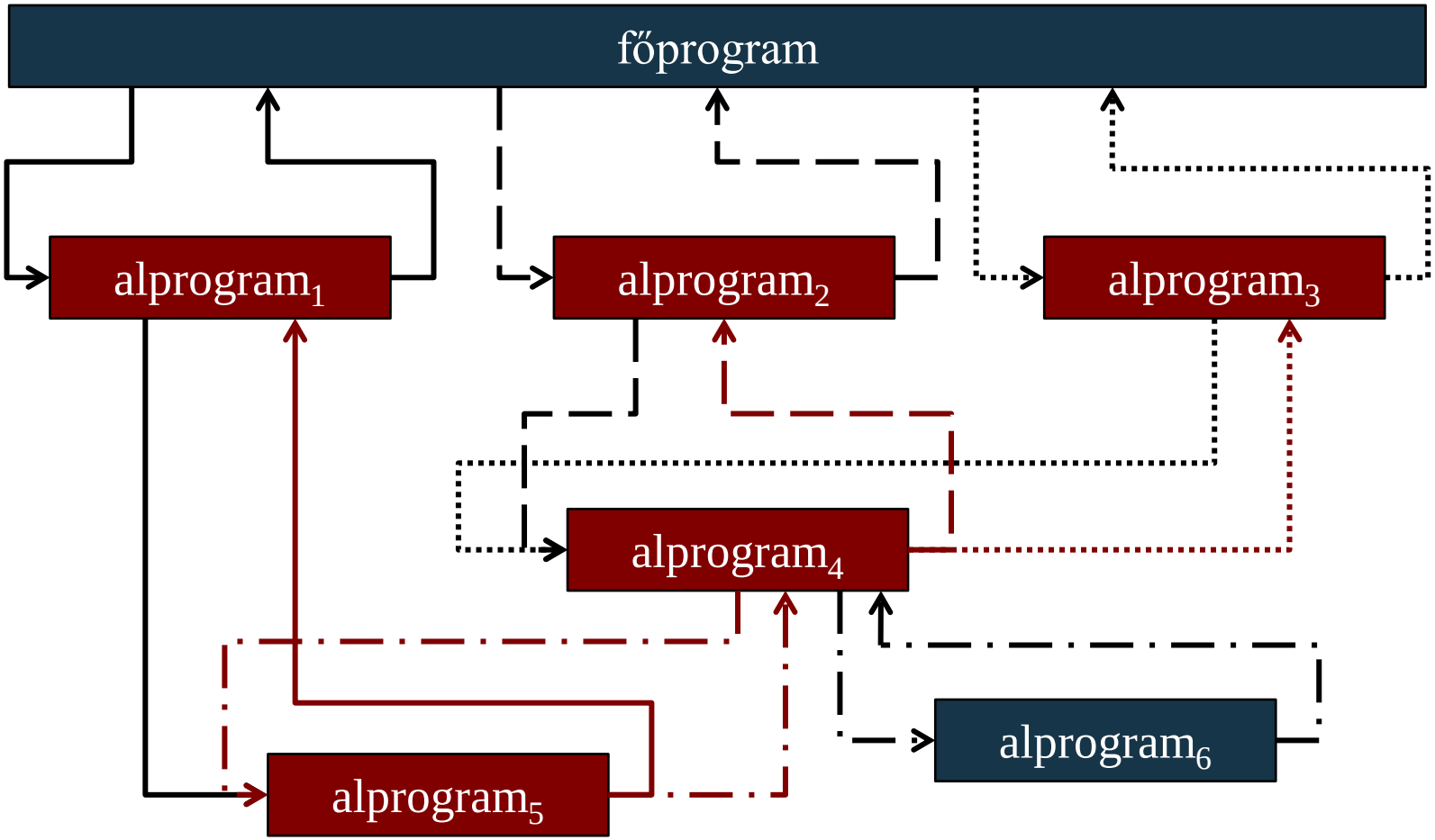
Procedurális programozás

A feladatmódosítás hatásai



Procedurális programozás

A feladatmódosítás hatásai



Objektumorientált programozás

Kialakulása

- A korlátok kiküszöbölése:
  - a programot *tagolni kell*, de nem a részfunkciók, hanem a betöltött szerep szerint, tehát a programegységeket úgy alakítjuk ki, hogy azok egy adott tárgykörért feleljenek
  - az összetartozó programrészek között *szorosabb kapcsolatot*, a nem összetartozó programrészek között *lazább kapcsolatot* kell biztosítani
  - meg kell határozni az adatoknak a *programegység szintű láthatóságát*, mivel a globális adatok túl általánosan, a lokálisak pedig túl szűkkörűen használhatóak
  - *függetleníteni* kell az adatokat a főprogramtól, valamint mentesíteni azt a teljes program vezérlésének feladatkörétől

Objektumorientált programozás

Kialakulása

- Megoldás a *felelősség továbbadása*, azaz az adatok továbbadása az őket kezelő programegységek számára
  - így egy programegység alá nem csak a műveletek kerülnek, hanem az általuk manipulált adatok – ezt nevezzük *egységbe zárás*nak (*enkapszuláció*nak) – , így kialakul a programegységen belül a szoros összetartás
  - a programegységek között így már kialakítható egy gyengébb kohézió azáltal, hogy csak bizonyos pontokon engedélyezünk hozzáférést a részegységhez, a többi funkciót, adatot pedig *elrejtjük*
- Ennek eredményeként egy módosítás a feladatban csupán egy programegység módosítását vonja maga után

Absztrakció

- Az *absztrakciós szint* megválasztása lehetővé teszi, hogy az adott problémát több szinten kezeljük, azaz az összes lehetséges tulajdonságnak egy alkalmas részhalmazát válasszuk ki, és azokat használjuk a megvalósításban
  - a számunkra lényeges tulajdonságok kiemelését, és ezáltal az objektum meghatározását nevezzük *absztrakció*nak
  - pl.: az autót ábrázolhatjuk rendszámmal, sebességgel és tömeggel, vagy ábrázolhatjuk, mint motor és karosszéria összessége, vagy akár minden egyes alkatrészét külön-külön vehetünk
  - pl.: egy háromszöget ábrázolhatunk a 3 pont koordinátáival, vagy a két oldallal és a közrezárt szöggel, stb.

Objektumorientált programozás

Objektumok

- *Objektum*nak (object) nevezzük a feladat egy adott tárgyköréért felelős programegységet, amely tartalmazza a tárgykör megvalósításához szükséges adatokat, valamint műveleteket
  - az objektum képes önálló működésre, amely során saját adatait manipulálja, saját műveleteit futtatja és kommunikál a programban jelen levő többi objektummal
  - pl.: egy téglalapot kezelhetünk objektumként
    - adatai: szélessége és magassága
    - műveletei: területkiszámítás, méretváltoztatás
  - pl.: egy verem adatszerkezetet kezelhetünk objektumként
    - adatai: elemek tartalmazó tömb és a felhasznált méret
    - műveletei: Push, Pop, Top

Objektumorientált programozás

Objektum-orientált program

- *Objektum-orientált*nak nevezzük azt a programot, amely egymással kommunikáló objektumok összessége alkot
  - minden adat egy objektumhoz tartozik, és minden algoritmus egy objektumhoz rendelt tevékenység, nincsenek globális adatok, vagy globális algoritmusok
  - a program így kellő tagoltságot kap az objektumok mentén
  - az adatok élettartama így összekapcsolható az objektum élettartamával
  - a módosítások általában az objektum belsejében véghezvihetők, ami nem befolyásolja a többi objektumot, így nem szükséges jelentősen átalakítani a programot

Objektumorientált programozás

Alaptulajdonságok

- Az objektum-orientáltság öt alaptényezője:
  - *absztrakció*: az objektum reprezentációs szintjének megválasztása
  - *enkapszuláció*: az adatok és alprogramok egységbe zárása, a belső működés elrejtése
  - *nyílt rekurzió*: az objektum mindig látja saját magát, eléri műveleteit és adatait
  - *öröklődés*: az objektum tulajdonságainak átruházása más objektumokra
  - *polimorfizmus és dinamikus kötés*: a műveletek futási időben történő működéshez kötése, és a viselkedés átdefiniálása

Objektumorientált programozás

Objektumok állapotai

- Az objektumok *állapot*tal (state) rendelkeznek, ahol az állapot értékeinek összességét jelenti
  - az objektum mindig tisztában van az állapotával
  - két objektum állapota ugyanaz, ha értékeik megegyeznek
  - az állapot *esemény* (műveletvégzés, kommunikáció) hatására változhat meg
- A program teljes állapotát a benne lévő objektumok összállapota adja meg, így az továbbra is imperatív marad
- Minden objektum egyértelműen azonosítható, és ez független a tárolt értékektől
  - két objektum akkor sem azonos, ha egy állapotban vannak

Objektumok

- Az objektumoknak *életciklusa* van
  - megszületik: létrejön a memóriában, benne lévő adatok inicializálódnak
    - a *konstruktor* művelet végzi, feladata a változók kezdőértékeinek beállítása, és a működéshez szükséges tevékenységek elvégzése
  - él: kommunikálhat (üzeneteket fogadhat és küldhet), állapotot válthat
  - meghal: befejezi a működését
    - a *destruktor* művelet végzi, amely lezárja a működését eltávolítja a memóriából (általában nem kell külön implementálnunk, szintén van alapértelmezett)

Objektumorientált programozás

Osztályok

- Az objektumok viselkedési mintáját – azaz a tárolható adatokat, és azokkal végezhető műveletek halmazát – az *osztály* tartalmazza
  - tehát az osztály az objektum típusa
  - az osztályból *példányosít*hatjuk az objektumokat
  - pl. háromszögek és vermek osztálya
- Az osztályban tárolt adatokat *attribútum*oknak, vagy *mező*knek (field), az általa elvégezhető műveleteket *metódus*oknak (method) nevezzük, együtt ezek az osztály *tag*jainak (member)
  - a tagokat elrejthetjük a többi osztály elől a láthatóság szabályozásával, így az osztály megvalósítása, belső működése később könnyen módosítható lesz

Osztályok

- Az osztály kívülről látható részét *interfész*nek, a kívülről nem látható részét *implementáció*nak nevezzük
  - a metódusok megvalósítása mindig az implementáció része, tehát más osztályok számára a működés maga mindig ismeretlen
- Az osztályokat úgy kell megtervezni, hogy ne háruljon
  - egyikre se túl nagy felelősség, azaz ne legyenek túl bonyolultak, mert áttekinthetetlenné válik az objektumok működése
  - mindegyikére túl kicsi felelősség, mert ekkor túl sok objektum jelenik meg a programban, és áttekinthetetlenné válik a program működése

Objektumorientált programozás

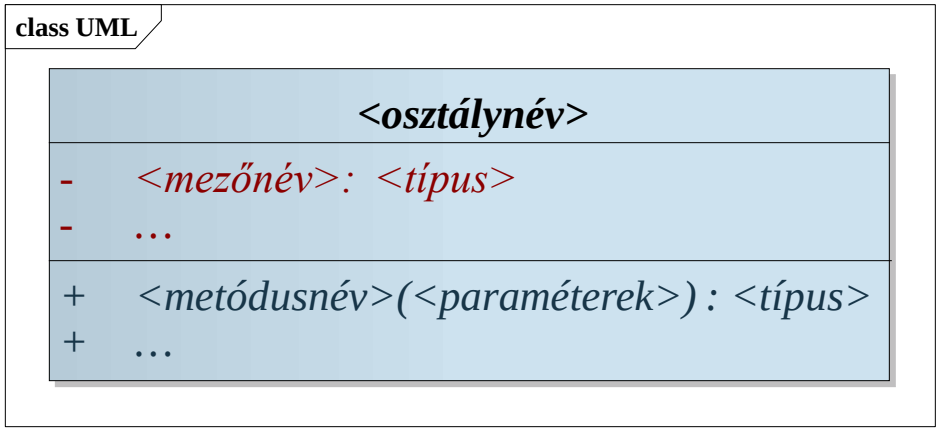
Osztályok tervezése

- A program szerkezetének és működésének megtervezések az osztályok és objektumok szempontjából történik
  - a szerkezeti tervezésnél az osztályok tagjait, azok kapcsolatait, illetve elhelyezkedését a programban, ez alkotja a *statikus terv*et
  - a programfutási tervezésnél az osztályok időbeli működését, az objektumok állapotainak változását modellezzük, ez a *dinamikus terv*ezés
- Az objektumorientált tervezés eszköze a *Unified Modelling Language* (UML), amelyben 13 diagramtípus segítségével tervezhető meg a program szerkezete és működése

Objektumorientált programozás

Az osztálydiagram

- Az UML diagramok körében az osztályok szerkezetét, a program felépítését az osztálydiagram reprezentálja
  - ábrázolja a rendszerben részt vevő osztályokat, azok felépítését (mezők és metódusok bontásában, típusok és láthatóság megjelölésével):



  - az osztályok közötti kapcsolatot relációk formájában, név és multiplicitás megjelölésével

Osztályok implementációja

- Az osztályokat minden nyelv más formában implementálja, de az általános jellemzőket megtartja
- A C# programozási nyelv tisztán objektum-orientált, ezért minden érték benne objektum, és minden típus egy osztály
  - az osztály lehet érték szerint kezelt (struct), vagy referencia szerint kezelt (class), előbbi élettartama szabályozott az őt tartalmazó blokk által, utóbbié független tőle
  - az osztály tagjai lehetnek mezők, metódusok, illetve tulajdonságok (property), utóbbi lényegében a lekérdező (*get*) és beállító műveletek (*set*) absztrakciója
  - minden tagot jelöl a láthatósággal, a látható tagokat a *public*, a rejtett tagokat a *private* kulcsszóval

Objektumorientált programozás

Osztályok implementációja

- A C# osztály szerkezete:

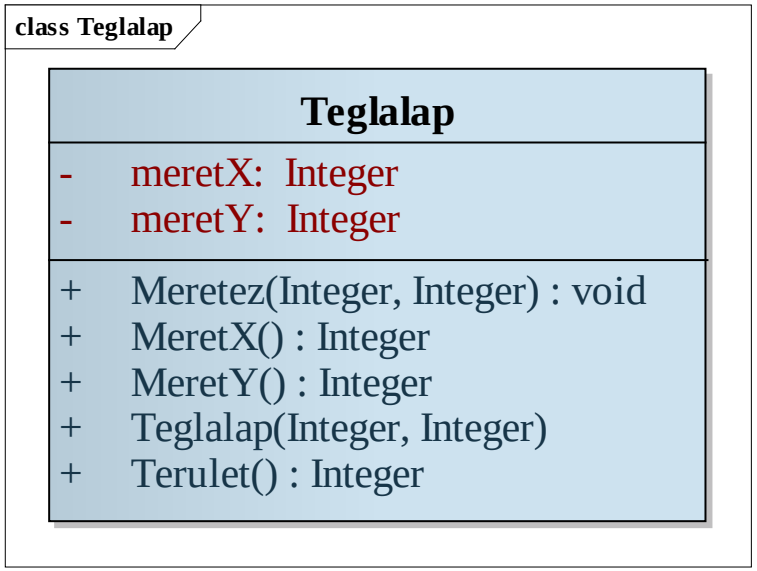
```
class/struct <osztálynév> {

    public/private <típus> <mezőnév>;
    …
    public/private <típus> <metódusnév>
      ([ <paraméterek> ]) { <működés> }
    …
    public/private <típus> <tulajdonságnév> {
       [ get { <működés> } ]
       [ set { <működés> } ]
    }
    …
}
```

Objektumorientált programozás

Példák

- A téglalap osztály megvalósítása:
  - mezői a két oldal mérete, ezeket elrejtjük
  - műveletei az átméretezés, illetve a méretek és a terület lekérdezése, ezeket láthatóvá tesszük
- A téglalap osztály terve (UML osztálydiagramja):



- A téglalap osztály megvalósítása C# nyelven:

```
class Teglalap{
   // a láthatóságokat tagonként adjuk meg
   private Int32 meretX;
   private Int32 meretY;

   public Teglalap(Int32 x, Int32 y)
   { // konstruktor
      meretX = x; meretY = y;
   }
   // az alapértelmezett destruktort használjuk
   public void Meretez(Int32 x, Int32 y){
      meretX = x; meretY = y;
   }
```

Példák

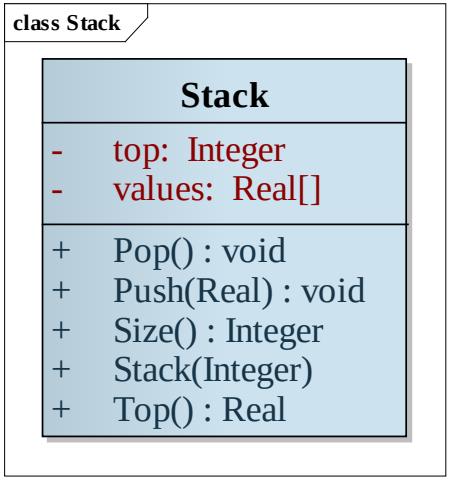
```
    // lekérdező műveletek, mint tulajdonságok:
    public Int32 MeretX { get { return meretX; }}
    public Int32 MeretY { get { return meretY; }}
    public Int32 Terulet {
       get { return meretX * meretY; }
    }
} // osztály vége

Teglalap t = new Teglalap(10,20);
// az objektumpéldány létrehozása
t.Meretez(20,20);
// módosító művelet meghívása az objektumra
Console.WriteLine(t.Terulet);
// lekérdező tulajdonság meghívása az objektumra
```

Objektumorientált programozás

Példák

- A verem (stack) osztály megvalósítása:
  - a verem egy tömb és elemszám segítségével írható le, amiket elrejtünk
  - műveletei a behelyezés (push), kivétel (pop), tetőelem (top), illetve méret (size) lekérdezés, ezeket láthatóvá tesszük
- A verem osztály terve (UML osztálydiagramja):



Objektumorientált programozás

Példák

- A verem osztály megvalósítása C# nyelven:

```
class Stack {
   private Single[] values;
   private Int32 top;

   public Stack(Int32 maxSize){ // konstruktor
      values = new Single[maxSize];
      top = 0;
   }
   void Push(Single v) {
      if (top < values.Length) {
         values[top] = v; top++; // méret növelés
      }
   }
```

Objektumorientált programozás

Példák

```
    public void Pop() {
       if (top > 0) top--; // méret csökkentése
    }
    public Single Top { // lekérdező tulajdonság
       get {
          if (top > 0) return values[top];
          else throw new Exception(); // hibajelzés
       }
    }
    public Int32 Size { get { return top; } }
}

Stack s = new Stack(10); // verem létrehozása
s.Push(5.5f); // elem behelyezése
…
```

Objektumorientált programozás

Objektumok példányosítása

- A program során objektumokat használunk a kommunikációhoz, ehhez példányosítanunk kell az osztályokat
- A példányosítás két lépésben történik:
  - *deklaráció*: megadjuk a változó nevet, valamint a típust (osztályt), ekkor még nem jön létre a változó, pl.:

```
Int32 x; // egész szám típusú változó
Teglalap t; // téglalap típusú változó
```

  - *definíció*: létrehozzuk a változót a **new** operátorral, lefoglaljuk a memóriában értékadás segítségével, ekkor futtatjuk a konstruktor műveletet, pl.:

```
x = new Int32(); // a 0 értéket kapja meg
t = new Teglalap(1,1);
```

Objektumok példányosítása

- a deklarációhoz nem csak új értéket állíthatunk be, hanem (primitív típusok esetén) azonos típusú konstanssal is létrehozhatjuk a változót, illetve más típusú változóból konvertálhatjuk, pl.:

```
x = 1;
x = Convert.ToInt32(Console.ReadLine());
```

- sok esetben a két lépést összevonjuk, és egyszerre deklaráljuk és definiáljuk a változót, pl.:

```
Int32 x = 1;
Teglalap t = new Teglalap(1, 1);
```

- Minden deklaráció élettartammal rendelkezik, lokális változó esetén a blokk végéig, mező esetén az objektum megsemmisüléséig használható

A memória és a mutatók

- A programok indításuk után bekerülnek a memóriába, és egy számukra fenntartott területen, úgynevezett *szegmens*ben futnak
  - a szegmensben tárolják a programkódot, valamint a változók értékeit
- A szegmens tekinthető egy vektornak, amelynek bájtjai sorszámozva vannak, és a sorszám alapján lekérhető bármely változó memóriabeli címe
  - mivel a sorszám (32, vagy 64 bites) egész szám, azért az értéke eltárolható, így lehetőségünk van olyan változókat létrehozni, amelyek direkt memóriacímeket tárolnak, őket nevezzük *mutató*knak (*pointer*)

Objektumorientált programozás

Memóriahely foglalás

- A C# programozási nyelvben a deklaráció során igazából egy mutató keletkezik, amit az értékadáskor egy tényleges objektum memóriacímére állítunk be
  - az ilyen módú változó létrehozást nevezzük manuális, vagy dinamikus memóriafoglalásnak
- A módszer előnyei:
  - az objektum élettartama függetleníthető a mutató élettartamától, tehát az objektum tovább létezhet, mint a rá hivatkozó mutató
  - az objektumok átadásakor (pl. paraméterként) nem kötelező a teljes objektum átmásolása, csupán a mutató értékének másolása, így megvalósítható a *cím szerinti átadás*

Érték és referencia típusok

- A létrehozás és kezelés módjától függően ezért kétféle osztálytípust különböztetünk meg:
  - *érték* (*struct*): érték szerint kezelendő típusok, azaz mindig másolódnak a memóriában átadáskor
    - az osztályt a **`struct`** kulcsszóval kell megadni
    - élettartamukat befolyásolja a blokk, vagy az őket tartalmazó objektum élettartama
    - többnyire primitív és egyszerű típusokra, amelyek kevés memóriaterületet foglalnak, ezért gyorsan másolhatóak
    - lehetőséget adnak paraméterként a cím szerinti kezelésre a **`ref`** kulcsszóval
    - pl.: **`Boolean`**, **`Int32`**, **`Char`**, **`Decimal`**, **`Single`**, ...

Érték és referencia típusok

- *referencia* (*class*): mutatókon keresztül kezelt típusok
  - az osztályt a **`class`** kulcsszóval kell megadni
  - blokktól és objektumtól független élettartammal, amíg legalább egy hivatkozás van az objektumra, addig a memóriában van
  - csak a memóriacím másolódik, amennyiben az egész objektumot másolni akarjuk a memóriában, a **`Clone()`** metódus meghívásával, vagy paraméterként a **`value`** kulcsszóval tehetjük, ez létrehozza a másodpéldányt
  - többnyire összetett típusokra, ahol a másolás művelete költséges lenne
  - pl.: **`String`**, (**`Teglalap`**, **`Stack`**,) …

Objektumorientált programozás

Példák

- Pl.:

```
struct Stack{ /*…*/ } // érték szerint kezelt

void PrintSize(Stack s){
   Console.WriteLine("Verem mérete: " + s.Size);
}
void MultiPop(Stack s, Int32 n){
   for (Int32 i = 0; i < n; i++) s.Pop();
}

Stack st = new Stack(100);
st.Push(1); // … elemek behelyezése
PrintSize(st); // a verem átmásolódik a memóriában
MultiPop(st,10); // az st nem fog módosulni
```

Objektumorientált programozás

Példák

- Pl.:

```
class Stack{ /*…*/ } // referencia szerint kezelt

void PrintSize(Stack s){
   Console.WriteLine("Verem mérete: " + s.Size);
}
void MultiPop(Stack s, Int32 n){
   for (Int32 i = 0; i < n; i++) s.Pop();
}

Stack st = new Stack(100);
st.Push(1); // … elemek behelyezése
PrintSize(st); // a verem nem másolódik
MultiPop(st,10); // az st módosulni fog
```

Biztonságos mutatók

- A C# nyelv elsődlegesen az úgynevezett *biztonságos mutató*kat, vagy *referenciá*kat támogatja a referencia típusok megvalósításánál
  - olyan mutatók, amelyek mindig rendelkeznek értékkel, és a működésüket felügyeli a virtuális gép
  - amennyiben nem adunk nekik objektum memóriacímét értékül, az üres (`null`) hivatkozást kell adnunk
  - mivel az objektum nem semmisül meg a blokk végén, amennyiben egy objektumra nem mutat egyetlen mutató sem, akkor már nem elérhető a program számára, ezért a virtuális gép gondoskodik a megsemmisítéséről (ezt hívjuk *szemétgyűjtés*nek), hogy ne foglalja feleslegesen a memóriát

Nyílt rekurzió

- Az objektum mindig tisztában van saját állapotával, vagyis elérheti mezőit, és azok aktuális értékét, és saját magából is meghívhatja metódusait
- Lényegében azt mondhatjuk, hogy az objektum rendelkezik saját maga felett, eléri saját magát, ezt *nyílt rekurzió*nak (*open recursion*) nevezzük
  - a nyílt rekurzióban az objektumon belül mindig elérhetjük az objektum hivatkozását (mutatón keresztül), a C#-ban erre a **`this`** kulcsszó használható
  - a nyílt rekurzió amellett, hogy lehetővé teszi a hivatkozáson keresztül is a tagok elérését (**`this.<`*`tagnév`*`>`**), lehetőséget ad saját magára való hivatkozás átadására (pl. metódus paraméterében)

Nyílt rekurzió

- Pl.:

```
class FirstClass {
    public SecondClass secClass;
    public FirstClass(SecondClass sc){
        this.secClass = sc;
    } // a mező eléréséhez nyílt rekurzió
}

class SecondClass {
    public SecondClass(){
        FirstClass fc = new FirstClass(this);
    } // objektumhivatkozás átadása nyílt
      // rekurzióval
}
```