Magasszintű Programozási Nyelvek II.

(2018-as tételsor)

# OOP története, alapfogalmai és alapelvei

## Mi az az OOP?

Alan Kay fektette le az OOP alapelveit a diplomamunkájában 1969-ben. A Xerox Palo Alto-i kutatóközpontban forradalmi fejlesztések voltak. GUI, egér, hálózat és a SmallTalk OOP nyelv kifejlesztése.

* **Adatmodell**: rekordok = osztályok
* **Algoritmus**: eljárások, függvények = metódusok

**Az OOP célja:** az adatmodell és az algoritmus szorosabb integrációja.

A meglévő procedurális nyelveket bővítették ki az OOP lehetőségeivel, például:

* Pascal -> Object Pascal, Delphi
* C -> C++

Új, tisztább OOP nyelveket is terveztek, például: **Java**, **C#**

## Fogalmak:

* **Objektum (osztály):** típusdefiníció, mely a mezőket és metődusokat tartalmazza.
* **Objektum:** adott osztály egy példánya.
* **Mező:** az osztályban adattárolási feladata van, a változóknak felel meg.
* **Metódus:** az osztályban lévő függvények és eljárások.
* **Példányosítás:** adott osztály típusú adathoz memóriafoglalás és inicializálás.
* **Konstruktor:** egy osztály speciális metódusa, mely egy új példány inicializálását végzi.
* **Destruktor:** egy osztály speciális metódusa, mely az objektum memóriából való felszabadítását végzi.
* **Adatrejtés:** annak szabályozása, hogy az osztály mely mezői hogyan lehetnek a külvilág számára elérhetőek. (irható/olvasható)

## Alapelvek:

* **Egységbezárás (encapsulation)**
  + Az osztály egyetlen egységként tartalmazza az adatokat és az őket kezelő metódusokat.
  + A mezőket csak a metódusokon keresztül változtathatjuk meg, ellenőrzött módon.
  + Az osztály felelős a saját állapota konzisztenciájának fenntartásában.
* **Öröklődés (inheritance)**
  + Egy már definiált osztályból új, hasonló osztályok kifejlesztése.
  + A gyerekosztály örökli az ősosztály mezőit és metódusait.
  + Ezeketa a programpzó felüldefiniálhatja.
* **Sokoldalúság**
  + Ha a gyerekosztály felülírja az ősosztály metódusét, ennek használata automatikusan történjen.

# Adatrejtés. Property. Konténerosztályok

## Mező

Az osztályon belül adattárolási feladatot lát el, változónak felel meg.

## Metódus

Az osztályon belüli funkciók, melyek az eljárásoknak és függvényeknek felelnek meg.

## Védelmi szintek, adatrejtés

Az egységbezárás alproblémája. Az osztály kritikus mezőit elrejtjük a külső programrészek elől. Célja, hogy a külvilág csak ellenőrzött mődon férjen hozzá mezőkhöz.

* **Private:** csak az adott osztályban érhető el.
* **Protected:** csak az adott osztályban vagy annak gyerekosztályában érhető el.
* **Public:** bárki számára elérhető.

Ha egy osztály DLL-ben van definiálva, akkor:

* **Internal:** csak az adott DLL-en belül érhető el.
* **Internal protected:** csak az adott DLL-en belül vagy a gyerekosztályaiból érhető el.

Az alapértelmezett védelmi szint a **private**. A public mező kontroll nélkül írható, ezért kockázatos ezt hasznáni, nem ajánlott. A private/protected mező javasolt, amelyekhez a hozzáférést public metóduson keresztül biztosítjuk.

## Property

Egy virtuális „mező”. Nem igazi, külön adatmező, így a memóriában nem foglalódik neki hely. Van, illetve lehet hozzá „getter”(azaz lekérő) és „setter”(azaz módosító) is. Ezekkel lehet megadni, hogy mi történjen olvasáskor és íráskor. Kulcsszavai: **get** és **set**.A set égban a *value* kulcsszóval lehet hivatkozni a beállítandó értékre.

* **Előnye**
  + Hibakezelés (mező közvetlen eléréséhez képest)
  + Szebb, elegánsabb a kód (a metódushíváshoz képest)
* **Hátránya**
  + Sebessége megtévesztő lehet: a programozó gyors elérésű (fizikai) mezőnek hiszi, holott hosszú, bonyolult kód állhat mögötte.

## Konténerosztályok

Olyan osztály mely gyűjtőként szolgál egy másik osztály példányai számára és alkalmazza az adatrejtés elvét. Pl. Privát lista egy propertyvel, ami egy másik listaként adja vissza a mező elemeit.

# 3. Példány- és osztályszint. Névterek

## Példányszintű mezők

Olyan adatokat tárolnak, amelyek az objektum példányai esetén eltérő értékeket tartalmazhatnak. Például egy kor objektum esetén a kör példányok sugarai eltérőek lehetnek, akár a körök X és Y koordinátái. A példányszintű mező lényegében a rekord adatszerkezetbeli “mező” fogalommal egyenértékű.

**Szintaktika**: [védelmi szint] <típus> <mezőnév> [= kezdőérték]

A kezdőérték elhanyagolható, alapértelmezetten null értéket kap a típusnak megfelelően.

Ugyanazon osztálybeli példányszintű metódusok, konstruktorok, property törzsében közvetlenül hivatkozhatunk a mezőre. Ha a hatókör minimum protected a gyerekosztálybeli metódusok is hivatkozhatnak rá. A public mezőkre a külvilági kódok is hivatkozhatnak, számukra a szintaktika kötelező, azonosítani kell a példányt is melynek a mezőjére hivatkoznának. “példánynév.mezőnév”

**Szintaktikák:**

* Mezőnév
* This.mezőnév
* Példánynév.mezőnév

A példányszintű mezőkből a memóriában kezdetben nulla darab van, példányosításkor a ’new’ operátor foglalja le a helyet az új példénynak.

Hatókör: A védelmi szinttől függ.

Életttartam: A példány élettartamával jellemezhető. A mező addig létezik, míg a példány is. (Ez automatikus folyamat a C#-ban, a Garbage Collector végzi. Észleli, hogy a példány memóriacímét elvesztette.)

## Osztályszintű mezők

Olyan adatok, amlyeket a program szempontjából elég egyetlen egyszer eltárolni. Iyen például a KRESZ szempontjából a lakott területen belüli legnagyobb sebesség 50 km/h. Ezek olyan értékek, amelyeket akár konstansként is felvehetnénk, mert bármikor változhatnak.

Az osztályszintű mezők nem egy példány jellemeznek, hanem az adott osztály minden példányára jellemző.

**Szintaktika:** [védelmi szint] static <típus> <mezőnév> [= kezdőérték]

A static és a védelmi szint kulcsszavak helye felcserélhető.

Az osztályszintű mezők már a program futáskor léteznek, ezért külső kódból “osztálynév.mezőnév” alakban hivatkozhatunk. Az osztályon belüli konstruktorok, metódusok, propertyk belsejében az osztályszintű mezőkre közvetlenül hivatkozhatunk “osztálynév.mezőnév” alakban, amennyiben az “osztálynév.mezőnév” az alapalak. A “this” szó használata szintaktikai hiba, mert az osztályszintű mezők nem köthetőek példányokhoz.

**Szintaktikák:**

* Mezőnév
* Osztálynév.mezőnév

**Hatókör:** Védelmi szinttől függ.

**Élettartam:** Statikus, tehát a mezők a program indulásakor kerülnek a memóriába és a program futásának végéig ott is maradnak. Az osztályszintű mezőkből csak egy van a memóriában, akkor is, ha egyetlen példány sem készül az osztályról.

## Konstansok

Olyan adatokat tartalmaznak, melyeknek értéke a program írásakor ismert és várhatóan nem fog változni. Például a Pi értéke. Szintén osztályba helyezendők. Érdemes a konstansokat olyan osztályba tenni, amelynek a neve köthető a konstanshoz.

**Szintaktika:** const [védelmi szint] <típus> <mezőnév> = kezdőérték;

A ’const’ és a védelmi szint kulcsszavak felcserélhetőek.

Osztályon belüli kód közvetlenül a névvel hivatkozhat. Gyerekosztály szintén hozzáfér. Külvilág esetén ’osztálynév.konstansnév’.

**Hatókör:** Védelmi szinttől függ. Általában public, mert úgysem módosítható. De lehet private vagy protected is.

**Élettartami:** Statikusnak tekinthető az élettartama.

## Névterek

A névterek azért jöttek létre, hogy az osztálynevek egyediségét erősítse valamit, hogy csoportosítsa az összetartozó programrészeket. Például A matematikai metódusoka a Math névtérbe tartoznak.

**Tartalmazhat:** class, struct, enum, interface, delegate és másik névtéret is.

**Nem tartalmazhat:** metódus, mező, konstans, property.

A névterek használata redundáns és kényelmetlen lehet, ezért a **using** kulcsszóval kiemelhetőek a gyakran használt névterek.

Ha egy osztályt nem helyezünk névtérbe, akkor az a globális névtérbe fog tartozni.

# 4. Öröklődés

## Az öröklődés szabályai

Ha már megírtunk egy A osztályt, akkor egy másik hasonló B osztályt ne kelljen nulláról megírni, B osztályt az A-ból származtatjuk (örököltetjük). B örökli az A szülőosztály összes mezőjét, konstansát, property-jét, metódusát. B-ben új mezőket, metődusokat stb. Is tudunk bevezetni.

* **Simula (1967, Norvégia):** Az öröklődés első megjelenése.
* **Smalltalk, Java, C#:** egyszeres öröklődés = 1 ős.
* **C++, Python:** többszörös öröklődés = több ős.

## Védelmi szintek

Mezők property-k, metódusok 3 védelmi szinttel rendelkeznek.

* **Public**: öröklődik és a gyerekosztályban közvetlenül elérhető + mindenki számára elérhető.
* **Protected**: öröklődik és a gyerekosztályban is közvetlenül elérhető.
* **Private**: öröklődik, de a gyerekosztályban nem érhető el közvetlenül, csak indirekt módon public/protected metódust írva.

## Mező újradefiniálása

**Privat****e:** a gyerekosztály nem látja a mezőt, ezért bevezethetünk ugyanolzan nevű mezőt.

**Protected/public:** a gyerekosztály látja a mezőt, ezért ugyanolyan nevű mező bevezetésénél használni kell a **new** kulcsszót. A new-val jelzi a programozó a fordítónak, hogy „tudja mit csinál”.

## Metódusok és propery-k újradefiniálása

* **Overloading:** Más paraméterezéssel ugyanolyan nevű metódust hozhatnuk létre, de az overloading szabályai vonatkoznak rá, azaz nem fedi el az eredeti metódust.
* **Korai kötés:** Ugyanolyan nevű és paraméterezésű metódus „elfedi” az ősosztály metódusát. A **new** kulcsszóval a fordító „összeköti” a metódus hivatkozását a fizikai metódussal. A metódushívások által hivatkozott metódusok **fordítási időben** lesznek kiválasztva.
* **Késői kötés:** Felüldefiniáláskor kötelező használni az **override** kulcsszót. (Ilyenkor érdemes az ősosztályban a **virtual** kulcsszót használni a felüldefiniálandó metódusnál.) Nem szabad megváltoztatni a felüldefiniált metódus egyik adatát sem, **csak a metódus törzsét szabad átírni**! Virtuális metódus nem lehet private!

Property esetén, ha nincs kidolgozva a get/set rész az ősosztályban, akkor a gyerekosztály sem vezetheti be a hiányzó részeket.

# 5. Konstruktorok

## Fogalma, feladata

Sepciális metódus, melynek feladata példányosításkor alap állapotba helyezni (inicializálni) az objektumot.

* Érték típusú mezőknek kezdőértéket ad.
* Referencia típusú mezőkhöz példányt rendel.

## Alapértelmezett konstruktor

Ha nem írunk konstruktort egy osztályba, akkor is tudjuk példányosítani. Ilyenkor a fordító automatikusan egy alapértelmezett konstruktort hoz létre az osztályban. **Public, nincs paramétere és üres a törzse.** (Ha írtunk konstruktort, akkor ez nem jön létre.)

## Konstruktorhívási lánc

Konstruktor fejléce mögé **kettőspont**, majd a **this** vagy **base** kulcsszó segítségével másik konstruktorra hivatkozik. A konstruktorok lefutási sorrendje fentről-lefelé történik, tehát az első az ősosztály konstruktora és az utolsó az objecté.

## Osztályszintű konstruktor

Az osztályszintű (static) mezőknek is szüksége van inicializálásra. Ezt végzi az osztályszintű konstruktor. Ezt a konstruktort nem lehet lefuttatni, a futtatórendszer hívja meg automatikusan.

* Neve kötött: ugyanaz, mint az osztály neve.
* Nincs visszatérési típisa (még void sem!)
* **Nincs paramétere**
* **Static**
* Kötelezően **private**, ezért nem is kell láthatóságot megadni.
* Csak 1 db. Lehet belőle egy osztályban.

## Object Factory

Nem tesszük lehetővé, hogy bárki ellenőrzetlen módon példányokat készítsen az osztálból, ezért **private konstruktort** adunk meg. Készétünk egy **public static metódust**, mellyel ellenőrzött módon készíthetjük el a példányunkat. Ezt nevezzük **object factory**-nak. Az object factory ellenőrzéseket végezhet el, mielőtt új példányt hozunk éátre. Meg is tagadhatja a példány létrehozását.

# 6. Típuskompatibilitás

## OOP területén a típuskompatibilitás fogalma

Ha az Y típus kompatibilis az X típussal, akkor mindenhol, ahol X típusú kifejezés szerepelhet, ott szerepelhet Y típusú is. **Y osztály kompatibilis X osztállyal, ha Y-nak őse X**. (A típuskompatibilitás mindig **egyirányú**)

**Miért?:**

* Y az X minden mezőjét, property-jét, metódusát örökli.
* Ezért: Mindent, amit el tud végezni X-szel, el tudja végezni Y-nal is.
* Ezért: Y egy példánya képes helyettesíteni X bármely példányát.

## as operátor

**Formája:** **<példány> as <osztály>**

A példány úgy viselkedik, mintha az adott osztály példánya lenne (ha kompatibilis)

Késői kötés működéséhez ez a típuskényszerítés nem szükséges. Helyette ez is lehet:

(**<osztály>**)**<példány>**

Ha a típuskénszerítés nem vérgehatható, akkor **InvalidCastException** kivétel dobódik.

## is operátor

**Formája:** **<példány> is <osztály>**

Logikai értéked ad vissza:

* True, ha a példány típusa kompatibilis az adoott osztállyal.
* Mindig felesleges x is Object, x as Object

## Típuskompatibilitás haszna kollekciók használatakor

Kollekciók esetén a típuskompatibilitás haszna különböző példányok eltárolásánál, majd visszakeresésénél válhat hasznunkra. Például, ha van egy **Élőlény osztályunk**, aminek a gyerekosztálya az **Ember** – és **Madár osztály**, akkor ezeknek a példányait eltárolhatjuk egy közös kollekcióban, ahonnan az is és as operátorok segítségével kikereshetjük, hogy melyek az Ember – és melyek a Madár osztály példányai.

## Object típus és annak virtuális metódusai

C#-ban minden osztálynak (max.) 1 szülője lehet. Ha nem deklarálunk egy osztálynak őst, akkor is van neki: **System.Object** (röviden **object**). Mivel minden osztálynak az Object az őse, így **minden osztály kompatibilis az Object**-tel.

# 7. Sealed, static és abstract osztályok

## Sealed osztályok

Az osztály „véglegesnek” tekintjük, **zároljuk**.

A fodító bizonyos optimalizálásokat végezhet el pl. virtuális metódushívásokat feloldhat korai kötéssel.

Nem csak osztályok, hanem metódusok is lehetnek seald-ek, de csak **virtuális metódusokra alkalmazható**. **Sealed** csak **override**-dal együtt szerepelhet.

## Static osztályok. Sineleton

Csak **osztályszintű (static) mezőket, property-ket, metódusokat tartalmazhat.**

Nem lehet benne példányszintű elem, így konstruktor sem.

* Úgynevezett **single osztály** = csak 1 db. létezik a a memóriában.
* A fordító sem generál alapértelmezett konstruktort.

Ezzel a belőle öröklést is letiltjuk, a konstruktorhívási lánc miatt. **(Erősebb mint a sealed.)**

## Absztrakt osztályok

**Olyan esetben, mikor az osztályban egy metódust még nem tudunk megírni, mert a metódus implementációja csak a gyerekosztályban nyer értelmet**.

Meg lehet oldani virtuális metódussal:

* Üres/fiktív törzzsel: Nem jó, mert a programizó elfelejtheti override-olni.
* A törzsben kivétel dobásával: Csak futási időben figyelmeztet.

**Abstract** kulcsszóval kell megjelölni a nem kidolgozott metódusokat. **(Ne adj neki törzset!)**

A gyerekosztálbyan kötelező override-olni az abstract metódusokat, kivéve, ha az is abstract osztály.

## Absztrakt metódusok és virtuális metódusok összehasonlítása

**Virtuális metódus**: Olyan metódus, melyet a **gyerekosztályban valószínűleg felüldefiniálunk** és **késői kötéssel kezelünk**. Az ősosztályban **virtual** kulcsszóval kell őket megjelölni. A gyerekosztályban **override** kulcsszóval kell felüldefiniálni.

**Absztrakt metódus:** Olyan metódus, mely még nincs definiálva, ezért a gyerekosztályra bízzuk a definiálását.

# 8. Indexelők. Operator overloading

## A this kulcsszó fogalma és használati lehetőségei

Néha szükséges az objektum metódusaiban magára az aktuális példányra hivatkozni. Ezt a **this** kulcsszóval hetetjük meg.

Ha formális paraméter neve megegyezik egy mező nevével, akkor

* **This nélkül:** a paramétert jelöli
* **This-szel:** hangsúlyozza, hogy ez példánymező

Property esetén a mező neve **value**

* **This nélkül:** a kulcsszót jelöli
* **This-szel:** a példánymezőt jelöli

## Indexelők és azok jelentősége

Family f = **new** Family();

Person p = f[3];

…

**public** Person **this**[int i]

{

**get** {

**if**(0 <= i && i < members.Count)

**return** members[i];

**throw** **new** Exception("Indexing error");

}

**set** {

**if**(0 <= i && i < members.Count)

members[i] = **value**;

**else** **throw** **new** Exception("Indexing error");

}

}

Speciális indexelésre való propert:

* Neve kötelezően **this**
* Paraméterlistája szögletes [] zárójelek között.

## Operator overloading

**class** **Date** {

**public** **static** Date **operator**++ (Date d) {

d.numOfDays++;

**return** d;

}

}

Az operátor metódusa kötelezően statis és public!

Csak ezeket a C#-ban meglévő operátorokat lehet overloadolni:

* Unáris: pl. + - ! ++ --
* Bináris: pl. + - \* / % == != < > <= >=
* Kasztolás: (T)x

Szabadon pareméterezhető és nem lehet az operátorok precedenciáját megváltoztatni.

# 9. Interface

## Interface szintaktikai szabályai

Új nyelvi elem, amit arra használunk, hogy egymással nem rokon osztályok funkcionalitását írja le

Interface = „funkcionalitás definíció”

Nem számít osztálynak (még absztraktnak sem, de arra hasonlít)

Csak metódusok és property-k szignatúráját tartalmazza => mindegyikük absztrakt.

Nem tartal mezőket.

Mindegyikük publikus.

## Interface és absztrakt osztály összehasonlítása

* „abstract class” helyett **„interface class”.**
* **Csak metódus/property** signatúrák.
* Metódusok/property-k előtt **nem kell public.**
* Metódusok/property-k előtt **nem kell abstract.**
* Interfészből öröklődés esetén **nem kell override.**
* Nincsenek mezők.
* Nincsenek konstruktorok/destruktorok.
* Nincsenek osztályszintű elemek.

## Interface implementálása, öröklődéssel való kapcsolata

Egy osztálynak csak 1 db. ősosztálya lehet, de **akármennyi interfész őse** **lehet**!

Interfész is származhat interfész(ek)ből.

Implementálás:

* **„Az osztály implementálja az interfészt”**
* Az osztály lehet absztrakt, ha nem tudja az interfész valamely metódusát implementálni.

## Ismert interface-ek a .NET keretrendszerben

1. IComparable: int CompareTo (Object x) )
2. IComparer: int Compare (Object x, Object y)
3. IEnumerable: Ienumerator GetEnumerator
4. IEnumrator: object Current (prop), bool MoveNext(), void Reset()
5. IDisposable: erőforrások felszabadítása
6. IClonable: objektum klónok előállítása
7. ISerializable: objektumok szerializálása, azaz bifolyammá alakítása

# 10. Kivételkezelés

## Hagyományos hibajelzési technikák, azok hátrányai

A metódus hívásakor if utasításokkal észleljük a metódus által „dobott” hibát.

* A hiba kezelése túlbonyolítja a kódot.
* Nem emlékeznek a programozók a speciális visszatérési értékek jelentésére.

Állítsuk meg a programot? Írjuk ki a hibaüzenentet?

## Kivételkezelés filozófiája

A program alapértelmezett „normál” módban fut. (Nincs a kódban if-es hibakezelő kód.)

Ha valamilyen hiba merül fel a program „hiba” módba lép, melyben:

* A normál mód utasítása nem kerülnek végrehatjásra.
* A return-höz hasonlóa visszaugrálunk a metódusok láncán
* Ha közben a hibát nem kezeljük le, a program véget ér.

## Kivételosztályok

Exception

SystemException

ArgumentException

ArgumentNullException

ArgumentOutOfRangeException

FormatException

IOException

DriveNotFoundException

FileNotFoundException

PathTooLongException

PipeException

NotImplementedException

NullReferenceException

OutOfMemoryException

StackOverflowException

Exception

SystemException

ArithmeticException

DivideByZeroException

OverflowException

ArrayTypeMismatchException

IndexOutOfRangeException

InvalidCastException

PrintSystemException

PrintCommitAttributesException

PrintingNotSupportedException

PrintJobException

PrintQueueException

PrintServerException

## Kivételek dobása és elkapása

„Hiba” állapotba egy **kivétel** dobásával tudunk lépni. A **throw** utasítást kell hasznáni. Az **Exception** osztály egy példányát kell dobni.

A „hiba” mód hívási láncon való visszalépegetését egy **try-catch** utasításpár segítságável állíthatjuk meg.

* **Try:** Utasításblokkjában kivétel dobódhat fel és így a program „hiba” módba kerülhet.
* **Catch:** Elkapja a try-ban feldobott kivételt és visszaállítja a programot „normál” módba.

# 11. Generikusok

## Boxing és unboxing

**Boxing:** Az a művelet, mikor egy **értéktípusú** értéket egy **referenciatípusú** *változóba helyezünk*.

* Az értékről másolat készül a memóriában.
* A másolat memóriacíme bekerül a változóba.

**Unboxing:** Az a művelet, mikor *kinyerjük* a **refernciatípusú** változóba helyezett **értéktípusú** értéket.

A boxing és unboxing műveletek automatikusan hajtódnak végre.

## Általános célú programozás Object használatával

Általános célú metódusokhoz bármilyen típusú aktuális paramétert átadhatunk. Pl. Display(object x).

Az Object metódusai: ToString(), Equals(), GetHashCode()

## Generikusok írása

**class** **KeyNode**<K, T> {

K key;

T data;

…

**public** T Find(K key) {…}

}

* Előre meghatározzuk pl. a kollekció elemeinek típusát.
* Típusvizsgálat fitási időben.

Paraméterben bármennyi típusparamétert fogadhat egy generikus.

KeyNode<int, bool> n1 = **new** KeyNode<int, bool>();

KeyNode<string, double> n2 = **new** KeyNode<string, double>();

double x = n2.Find("wazzup");

## Ismert generikusok a .NET keretrendszerben.

|  |  |
| --- | --- |
| **using System.Collections;** | **using System.Collections.Generic;** |
| ArrayList | List<T> |
| Stack | Stack<T> |
| Queue | Queue<T> |
| Hashtable | Dictionary<K, V> |

# 12. Delegate

## Delegate fogalma és használata

Memóriában kétfajta tartalom:

* Adat => erre mutatnak a referencia típusú változóink.
* Kód => erre mutatnak a delegate-ek

**Delegate = metódus referencia**

* Eljárás/függvény belépési pontjára mutat.
* Továbbá tárolja a visszatérési típusát és a paraméterezését.

Használata:

1. Delegate típus bevezetése
   * Delegate int Operation(int a, int b);
2. Megfelelő szignatúrájú metódusok készítése
   * Static int Addition(int x, int y) { return x + y; }
3. Delegate típusú változók használata
   * Operation op = Addition; // Metódus neve zárójel nélkül: **REFERENCIA**
4. Delegate hívása
   * Op(89, -5); // Metódus neve zárójellel: **HÍVÁS**

## Callback metódusok

Az alkalmazáslogika NE írjon a felhasználói interfészre direkt módon, inkább használjon callback metódust!

## Események

Több delegate hívása olyan gyakori feladat, hogy a C# külön konstrukciót biztosít ehhez.

**event = esemény(kezelő);**

* Nem más mint a delegate-ek listája.
* Az eventhez akármennyi metódust hozzá tudunk adni.
* Az event hívásakor az összes metódus meghívódik.

## Anonim delegate-ek

Sok esetben egy metódust csak egyszer használunk fel delegate-ként, soha többet nem lesz szükségünk erre a metódusra. Ilyenkor Anonim delegate-et használunk.

**Anonim delegate = egyszer használatos név nélküli delegate**

Pl.: log += deleagate(string message) { Console.WriteLine(”{0}: {1}”, DateTime.Today, message); }

# 13. Lambda kifejezések és LINQ

## Lambda kifejezés fogalma, szintaxisa és jelentősége

Olyan kifejezés, mely névtelenül hajt végre adott metódusokat. Jele C#-ban: **=>**

* 1.db paraméterrel: x => x >= 0 ? x : -x;
* Több paraméterrel: (dx, dy) => Math.Sqrt(dx \* dx + dy \* dy);
* Paraméter nélkül: () => 42

## Ismert generikus delegate-ek a .NET keretrendszerben és ezek lambda kifejezésekkel való kapcsolata

**Predictate<T>**

* 1db. paraméter: T típusú
* Visszarétési érték: **bool** típusú
  + T = int, long double… -> x => x >=0 && x <= 100
  + T = string -> s => s.StartsWith("S")
  + T = point -> p => p.X \* p.Y >= 0

**Action, Action<T1>, Action <T1, T2>, …, Action <T1, …, T16>**

* 0, …, 16 db. paraméter: T1, …, T16 típusúak.
* Visszatérési érték nincs (**void**)
  + T1 = string, int, … -> s => Console.WriteLine("Az üzenet: {0}", s)
  + T1… = int, int string -> (x,y,t) => { int sum = x + y; t.Text = sum.ToString(); }

**Func<TR>, Func <T1, TR>, …, Func <T1, …, T16, TR>**

* 0, …, 16 db. paraméter: T1, …, T16 típusúak.
* Visszatérési érték **TR típusú**
  + T1, T2 = string, string -> s => s.ToUpper()
  + T1, T2, TR = string, uint, char -> (s,i) => i < s.Length ? s[i] : ' '

## LINQ bővítő metódusok és használatuk

LINQ = Language Integrated Queries = Nyelve integrált lekérdezések

A System.Linq névtér „kibővíti” az IEnumerable<T> interfészt.

* Ezt az interfészt implementálja a tömb a List<T>, stb.

„Bővítő metódus” (Extension methods) jelenne meg.

A metódusoka össze lehet fűzni. ( list.Where(…).OrederBy(…).Take(…).Select(…) )

Tipikusan készői kiértékeléssel dolgoznak:

* Csak akkor adják vissza a kötelező elemeket, amikor kérjük.
* Erre utal az IEnumerable interfész
* Ha azonnali kiértékelés kell: tömbbé/listává/stb. szükséges az eredményt konvertálni. (list.Where(…).OrederBy(…).Take(…).Select(…).ToList())

|  |  |
| --- | --- |
| **LINQ bővítő metódusok** | |
| Szűrés | **Where** |
| Projekció | **Select**, *SelectMany* |
| Rendezés | **OrderBy, ThenBy** |
| Csoportosítás | *GroupBy* |
| Joinok | *Join, GroupJoin* |
| Quantifiers | **Any, All** |
| Particionálás | **Take, Skip, TakeWhile, SkipWhile** |
| Halmazműveletek | *Distinct, Union, Intersect, Except* |
| Elemek | **First, Last,** *Single, ElementAt* |
| Aggregáció | **Count, Sum, Min, Max, Average** |
| Konverzió | **ToArray, ToList**, *ToDictionary* |
| Kasztolás | *OfType<T>, Cast<T>* |