Alma (tutoriál)

Készíts egy Alma osztályt! Az osztályt külön osztály könyvtárba készítsd el!

- Létrehozáskor az alma gyümölcskezdemény állapotú, 5 mm átmérőjű. Ebben az állapotban 2 másodpercenként 1-3 mm-t nő egészen addig, amíg el nem éri vagy meg nem haladja a 8 cm-t.
- A 8 cm-t elérő alma éretlen állapotú lesz, 5 másodpercenként érik 5-10%-ot.
- Amikor a gyümölcs eléri a 100%-os érettséget, akkor véletlenszerűen 2-5 perc elteltével elkezd rohadni.
- A rohadt gyümölcs 5 perc után megszűnik.

A programkódokban nem szeretjük a „mágikus konstans” értékeket látni. Például az 5-ös szám a feladat leírásában többször is előfordul. Ha a későbbiekben valamelyik 5-ös értéket meg szeretnénk változtatni, akkor nem lehet egyszerűen csere művelettel módosítani az értéket, hiszen ez az érték több mindenhez is tartozik. Ahhoz, hogy az alma ne 5, hanem 10 másodpercenként érjen, és ne 5 perc után, hanem 15 perc után pusztuljon el, végig kell böngészni a teljes kódot, és ki kell deríteni, hogy melyik 5-ös értékek látnak el ilyen funkciókat. Ez hosszadalmas művelet, ráadásul sok hiba lehetőséget rejt.

Ehelyett az osztályban konstans vagy readonly értékként felvesszük ezeket a számokat, és megfelelő módon elnevezzük őket.

A konstans mezők nevét végig nagybetűvel írjuk, amennyiben több szóból állnak, az olvashatóság kedvéért a szavak közé aláhúzás jelet teszünk.

Mivel a feladat leírás többféle mértékegységet is használ, ezért vagy az elnevezésben, vagy az érték mögé megjegyzésben odatesszük a mértékegységet is.

```
public class Alma
{
    const int KEZDO_MERET = 5; //mm
    const int NOVEKEDES_LEPES_IDO = 2; //sec
    readonly (int, int) NOVEKEDES_MERET = (1, 3); //mm
    const int ERESHATAR_MERET = 80; //mm
    const int ERES_LEPES_IDO = 5; //sec
    readonly (int, int) ERES_SZAZALEK = (5, 10); //%
    const int PERC = 60;
    readonly (int, int) ROHADAS_IDO = (2 * PERC, 5 * PERC);
    const int HALAL_IDO = 5 * PERC;

}
```

Mivel ezeket a mezőket még sehol nem használtuk, a program zöld hullámos vonallal aláhúzza őket, és a mezők neve halvány szürke. Ez az aláhúzás majd eltűnik, ha a mezőket használni kezdjük, és a mezőnevek betűszíne is erősebb lesz.

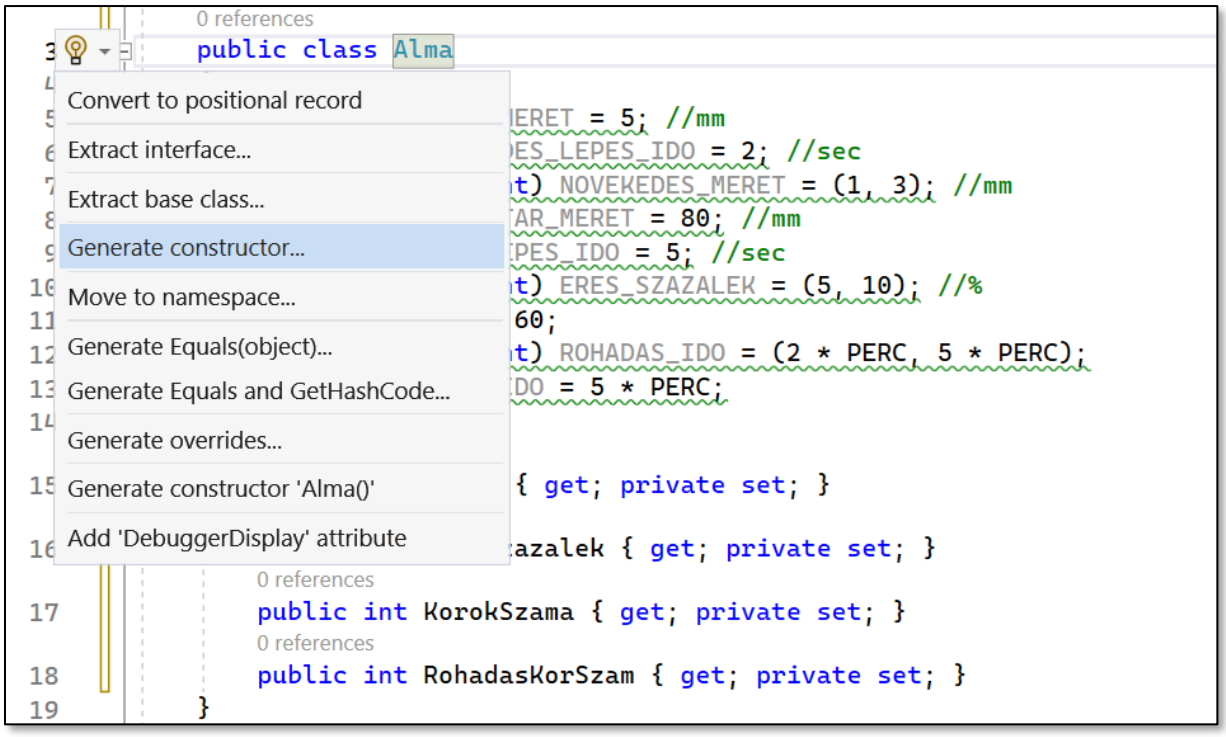
A növekedési méret, az érés százalék és a rohadás idő 2-2 egymáshoz tartozó érték. Felvehetnénk őket külön-külön is, de mivel összetartoznak, ezért célszerűbb inkább Tuple típusként tárolni őket. Tuple típus viszont nem lehet konstans, így az elérhetőségüket readonly-ra állítjuk a maximális védelem érdekében.

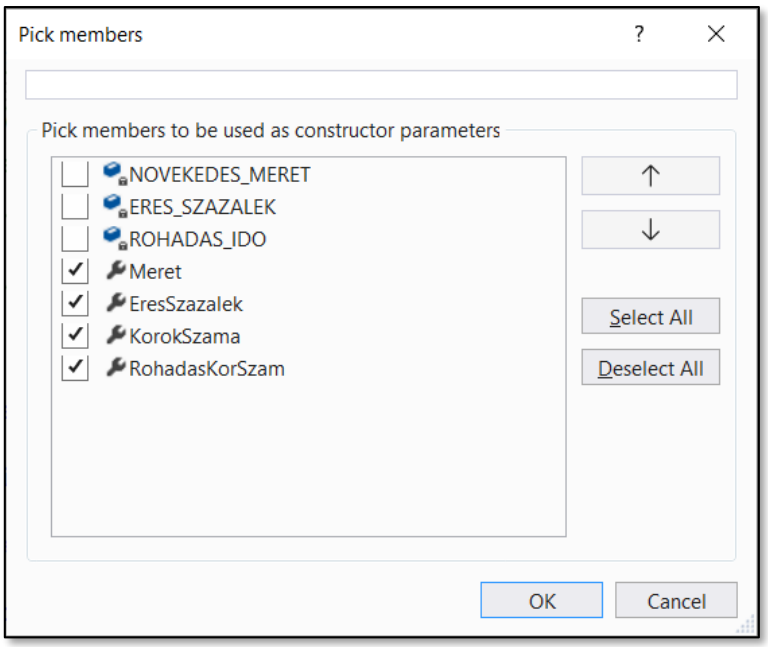
Az osztálynak 4 adattagja lesz, tárolni fogjuk az alma méretét, hogy hány százalékosan érett, hogy az alma létrejöttétől mennyi kör telt el, valamint a rohadás kezdetétől eltelt körök számát. Ezek az adatok kívülről lekérdezhetőek, belülről viszont nem csak a konstruktor állítja be őket, hanem az osztályon belül más metódusok is megváltoztatják majd az értéküket. A példányokból direkt módon természetesen nem engedjük módosítani az adattagok értékét. A tároláshoz tehát get; private set; részekből álló automatikus tulajdonságokat hozunk létre.

```
0 references
public int Meret { get; private set; }
0 references
public int EresSzazalek { get; private set; }
0 references
public int KorokSzama { get; private set; }
0 references
public int RohadasKorSzam { get; private set; }
```

Az osztály adattagjainak kezdőértékét egy paraméter nélküli konstruktorral állítjuk be. Erre a konstruktorra akkor lesz szükség, amikor létrehozunk egy még nem létező almát. Lesz egy paraméteres konstruktor is, amit majd akkor használunk, ha az alma már létezik, és az állapotát úgy szeretnénk visszaállítani a JSON fájlból beolvasott értékekből.

A paraméteres konstruktort úgy generáltatjuk a programmal, az osztály nevére kattintva, a Quick Actions and Refactorings… menüpontból kiválasztjuk a Generate constructor… almenüt.





Csak az automatikus tulajdonságok előtt hagyjuk ott a pipát, a Tuple típusú readonly mezőknél nem.
Az elkészült konstruktor:

```
0 references
public Alma(int meret, int eresSzazalek, int korokSzama, int rohadasKorSzam)
{
    Meret = meret;
    EresSzazalek = eresSzazalek;
    KorokSzama = korokSzama;
    RohadasKorSzam = rohadasKorSzam;
}
```

Mivel 2 konstruktor is lesz, és deszerializálásnál ezt a konstruktort szeretnénk használni, a konstruktor elé beírjuk a `JsonConstructor` attribútumot.

```
[JsonConstructor]
0 references
public Alma(int meret, int eresSzazalek, int korokSzama, int rohadasKorSzam)
{
    Meret = meret;
    EresSzazalek = eresSzazalek;
    KorokSzama = korokSzama;
    RohadasKorSzam = rohadasKorSzam;
}
```

A paraméter nélküli konstruktorral ugyanazoknak a tulajdonságoknak a kezdőértékét állítjuk be, mint a paraméteres konstruktornál. A méret tulajdonság a KEZDO_MERET néven elmentett konstans értéket kapja, az érés százalék és a körök száma 0 kezdőértéket, a rohadás körszám pedig a -1 értéket.

```
0 references
public Alma()
{
    Meret = KEZDO_MERET;
    EresSzazalek = 0;
    KorokSzama = 0;
    RohadasKorSzam = -1;
}
```

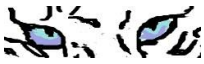
Készítsük el azokat a számított tulajdonságokat, amik segíteni fognak az alma állapotának megvizsgálásában. Ehhez 3 logikai tulajdonságot hozunk létre. Az alma akkor nőtt meg, ha a mérete elérte vagy meghaladta az éréshatár méretét. Az alma akkor fog rohadni, ha a körök száma nagyobb, vagy egyenlő a rohadás körszámnál. Az alma akkor van életben, ha vagy a rohadás körszám negatív, azaz az alma még nem kezdett el rohadni, vagy a körök száma még kisebb a rohadás körszám és az elhalálozáshoz tartozó idő összegénél. Az utóbbi tulajdonságot nem szeretnénk szerializálni, mivel halott almának nem tároljuk az adatait, így a tulajdonság elé beírjuk a JsonIgnore attribútumot.

```
0 references
bool Megnott => Meret >= ERESHATAR_MERET;
0 references
bool Rohad => KorokSzama >= RohadasKorSzam;

[JsonIgnore]
0 references
public bool EletbenVan =>
    RohadasKorSzam < 0 || KorokSzama < RohadasKorSzam + HALAL_IDO;
```

Mivel többször is kell 2 értékhatár között véletlen egész számot generálnunk, és a határokat readonly Tuple típusú mezőkbe mentettük, ezért készítsünk egy olyan private elérésű, statikus metódust, ami egy Tuple által megadott határok között generál egy véletlen értéket. A véletlen értékek előállításához a Random osztály Shared statikus tulajdonságát használjuk, amire a Next() metódust hívjuk meg megfelelő módon paraméterezve.

```
0 references
static int RandomIntervallum((int, int) intervallum)
    => Random.Shared.Next(intervallum.Item1, intervallum.Item2 + 1);
```



Készítsük el a növekedésért felelős metódust! Ha az alma megnőtt, akkor már nem növekszik tovább. Az alma a növekedés lépés idő eltelésével növekszik csak, így azt vizsgáljuk, hogy az eltelt körök száma osztható-e a növekedés lépési idővel. Amennyiben igen, akkor egy megfelelő véletlen értékkel megnöveljük a méretét, amihez felhasználjuk az előbb megírt RandomIntervallum() metódust.

```
0 references
void Novekedes()
{
    if (Megnott) return;
    if (KorokSzama % NOVEKEDES_LEPES_IDO == 0)
    {
        Meret += RandomIntervallum(NOVEKEDES_MERET);
    }
}
```

Készítsük el az érésért felelő metódust! Amíg az alma nem nőtt meg, addig nem érik. Ha már megnőtt, akkor az érés lépés idő eltelésével érik valahány százalékot, de csak addig, amíg el nem éri a 100%-os érést. Mivel véletlen százalékot érik, ezért előfordulhat, hogy az érési százalék meghaladja a 100%-ot. Ilyenkor az érési százalékot beállítjuk 100-ra, és elkezdjük számolni az időt a rohadásig. Ehhez a rohadás kör számát beállítjuk úgy, hogy annak a körnek a számához, amikor elérte az alma a teljes érést, hozzáadjuk a véletlenszerűen generált rohadási időt.

```
0 references
void Eres()
{
    if (!Megnott) return;
    if (KorokSzama % ERES_LEPES_IDO == 0 && EresSzazalek < 100)
    {
        EresSzazalek += RandomIntervallum(ERES_SZAZALEK);
        if (EresSzazalek >= 100)
        {
            EresSzazalek = 100;
            RohadasKorSzam = KorokSzama + RandomIntervallum(ROHADAS_IDO);
        }
    }
}
```

Készítsük el kör metódust! Minden körben nő a körök száma, valamint meghívjuk a növekedésért felelő és az érésért felelő metódust. Mivel a metódusok elején kezeltük, hogy csak akkor növekedjen és csak akkor érjen az alma, ha megfelelő állapotban van, így ebben a metódusban ezeket nem kell vizsgálnunk.

```
0 references
public void Kor()
{
    ++KorokSzama;
    Novekedes();
    Eres();
}
```

Most már csak egy ToString() metódust kell elkészítenünk, amely az aktuális alma állapotot kiírja. Ehhez egy switch kifejezésben megvizsgáljuk (amennyiben szükséges) az érés százalék értékét, és ennek megfelelő szöveges értékkel térünk vissza.

```
0 references
public override string ToString()
{
    return EresSzazalek switch
    {
        _ when !Megnott => $"Gyümölcskezdemény: {Meret} mm",
        < 100 => $"Gyümölcs: {EresSzazalek}% érett",
        _ when !Rohad => "Érett gyümölcs",
        _ when EletbenVan => "Ez az alma megrohadt",
        _ => "Ez az alma meghalt"
    };
}
```

Készen is van az Alma osztályunk.

Készíts egy futtatható projektet, amely a következő módon működik:

- Induláskor, ha nincs mentett alma, akkor hozzon létre egyet. Ha van mentett alma, akkor töltse be a JSON állományt, és jelenítse meg az alma állapotát!
- Kilépéskor mentse el az alma állapotát JSON állományba!
- A program futtatása alatt tizedmásodpercenként jelenítse meg az alma állapotát!

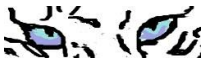
Készítsünk egy konzol alkalmazást, majd adjuk hozzá projekt referenciaként az előző részben létrehozott osztály könyvtárat!
Hozzunk létre egy Alma típusú, alma nevű változót. Ahhoz, hogy használhassuk az Alma osztályt, jobb egér gomb segítségével illesszük be a fájl elejére az osztály névterének használatát (nálam GyumolcsosKert_Lib)!

```
using GyumolcsosKert_Lib;

Alma alma;
```

Vegyük fel változóként azt a fájlnevet, amit szeretnénk használni az alma szerializálásához, deszerializálásához. Próbáljuk meg egy ilyen nevű fájlból deszerializálni az almánkat. Ha nem sikerül, akkor hozzunk létre egy új almát.

```
string fajlNev = "alma.json";
try
{
    alma = JsonSerializer.Deserialize<Alma>(File.ReadAllText(fajlNev))!;
}
catch
{
    alma = new Alma();
}
```



Többféle módon is megoldhatjuk a szimulációt. Először nézzük azt a verziót, hogy a körök léptetését billentyűleütéssel szimuláljuk. Egy hátul tesztelős ciklus segítségével lefuttatjuk a szimulációs kört, minden kör futása után kiírjuk az alma állapotát, majd a Thread.Sleep() metódus segítségével várunk egy tizedmásodpercet, végül beolvasunk egy billentyűleütést. A szimuláció addig tart, amíg a felhasználó Escape-et nem üt, vagy az alma elpusztul.

Ha kiléptünk a szimulációs ciklusból, akkor megvizsgáljuk, hogy az alma életben van-e. Amennyiben igen, akkor szerializáljuk az alma állapotát, és kiírjuk a megadott fájlba. Ha az alma már nincs életben, akkor töröljük az állapot tárolására szolgáló fájlt, hogy elpusztult alma már ne kerüljön visszatöltésre.

```
ConsoleKey keyChar;
do
{
    alma.Kor();
    Console.Clear();
    Console.WriteLine(alma.ToString());
    Thread.Sleep(100);
    keyChar = Console.ReadKey().Key;
}
while (keyChar != ConsoleKey.Escape && alma.EletbenVan);
if (alma.EletbenVan)
{
    File.WriteAllText(fajlNev, JsonSerializer.Serialize(alma!));
}
else
{
    if (File.Exists(fajlNev))
    {
        File.Delete(fajlNev);
    }
}
```

Készítsük el azt a verziót, amikor automatikusan fut a szimuláció addig, amíg az alma életben van, vagy nem ütünk Entert!

Vegyünk fel egy logikai változót, amiben tároljuk, hogy kaptunk-e utasítást a programból való kilépésre! A Parallel osztály Invoke() metódusával párhuzamos taszkokat tudunk indítani viszonylag egyszerűen, lambda kifejezések használatával. A lambda kifejezés meghívhat egy elnevezett metódust, vagy megadhatjuk a kódot beágyazottan is. Mi most az utóbbit fogjuk tenni.

Az első szálnál a while ciklus addig fut, amíg a kilépés változó hamis, és az alma életben van. Minden ciklus lépésben lefut egy kör, kiírásra kerül az alma állapota, majd egy tizedmásodpercig várakozik a program. Ha kilépünk a ciklusból, akkor megvizsgálja, hogy életben van-e az alma, amennyiben igen, akkor szerializálja és kiírja a fájlba. Ha már nincs életben az alma, akkor törli a JSON fájlt, amennyiben létezik.

A másik taszk azt figyeli, hogy a felhasználó Entert üt-e, és amennyiben igen, akkor a kilépés változót igazra állítja.

```
bool kilepes = false;
Parallel.Invoke(
    () =>
    {
        while (!kilepes && alma.EletbenVan)
        {
            alma.Kor();
            Console.Clear();
            Console.WriteLine(alma.ToString());
            Thread.Sleep(100);
        }
        if (alma.EletbenVan)
        {
            File.WriteAllText(fajlNev, JsonSerializer.Serialize(alma!));
        }
        else
        {
            if (File.Exists(fajlNev))
            {
                File.Delete(fajlNev);
            }
        }
    },
    () =>
    {
        Console.ReadLine();
        kilepes = true;
    });
```

Az elkészített programot módosítsd úgy, hogy a későbbiekben más gyümölcs működésének szimulálására is alkalmas legyen!

Bármit is szeretnénk körökre osztva szimulálni, biztos, hogy el kell készítenünk egy olyan metódust, ami azt adja meg, hogy mi történik az objektummal a körben. Egy másik közös pont, hogy vizsgálnunk kell, hogy az objektum életben van-e. Utóbbit egy lekérdezhető tulajdonságként érdemes megírni.

Készítsünk egy interfészt ISzimulacio néven az osztály könyvtárban, amelyben meghatározzuk, hogy azok az osztályok, amelyek szimulációt futtatnak, rendelkezzenek egy ilyen metódussal és tulajdonsággal!

Az interfésznek publikus láthatósággal kell rendelkeznie, hogy a futtatható projektből is elérjük.

```
public interface ISzimulacio
{
    0 references
    void Kor();
    0 references
    bool EletbenVan { get; }
}
```

Jelöljük az Alma osztályon, hogy ő szimulálható, azaz megvalósítja az ISzimulacio interfészt!

```
public class Alma : ISzimulacio
```

Módosítsuk a futtatható programot úgy, hogy Alma osztály helyett az ISzimulacio interfész legyen annak a változónak a típusa, amelyiket szimulálunk. Természetesen akkor már a változó név sem lesz megfelelő, neveztessük át az alma változó összes előfordulását szimulációra.

A program így is fut, gondolhatnánk azt, hogy ennyi változtatás elég is. Ha viszont leállítjuk majd elindítjuk újra a programot, akkor kiderül, hogy a kódunk nem jól szerializál. Mi lehet a probléma?

Szerializáláskor nem adtuk meg a szerializálandó típus paraméterét. Próbáljuk ki, hogy megoldja-e, ha a szerializálásnál megadjuk az Alma osztályt!

```
File.WriteAllText(fajlNev, JsonSerializer.Serialize<Alma>(szimulacio!));
```

Jól látható, hogy fordítási hibához jutunk. Nem lehet automatikusan konvertálni ISzimulacio típusról Alma típusra.

Módosítsuk a kódot úgy, hogy elvégezzük a kódban a típusátalakítást!

```
File.WriteAllText(fajlNev,
    JsonSerializer.Serialize<Alma>((szimulacio as Alma)!));
```

A programkód így már jól működik.

Most már megnyílt a lehetőség arra, hogy egy újabb szimulálható osztályt hozzunk létre az alkalmazásunkban, és a futtatható program megfelelő módosítása után a felhasználó választhasson, hogy milyen ISzimulacio interfészt megvalósító osztály példányának működését szeretné szimulálni és szerializálni.