Összefoglaló

Osztályok közötti kapcsolatok

A programunkban általában több osztályt is deklarálunk, amelyek között különféle kapcsolatokat hozunk létre. Egy ilyen kapcsolat az osztályok *függősége*, amikor egy osztály valamely példánya használja egy másik osztály példányát. Azt mondjuk, hogy egy osztály függ egy másiktól, ha a független osztályunk paraméterként vagy lokális változóként megjelenik a függő osztály valamely metódusánál. Erre mutat példát az alábbi programkód, ahol a `Player` osztály valamely példányán keresztül meghívott **UseItem** metódusnak paraméterként egy `Item` típusú objektumot kell átadnunk.

```
class Item
{
    public void DoSomething() {   }
}
class Player
{
    string name;
    int score;

    public Player(string name) {   }

    public void UseItem(Item item) {
        item.DoSomething();
    }
}
```

Asszociáció típusú kapcsolatról beszélünk, amikor a független osztály példánya a függő osztály attribútumaként (mezőjeként) jelenik meg. Az alábbi példában a `Game` osztály az előbbi `Player` osztályból példányosított objektumokat tárol egy listában (a játék résztvevőit), tehát a `Player` típus a `Game` osztály attribútumaként van jelen.

```
class Game
{
    List<Player> players;

    public Game() {
        players = new List<Player>();
    }
}
```

Megemlítjük még az asszociációs kapcsolat speciális esetét, az *aggregációt*, amikor az egyik osztály objektumai részét képezik egy másik osztály objektumainak. *Kompozíció* jellegű aggregációról beszélünk, amikor a függő és független objektum életciklusa megegyezik (vagyis egyszerre jönnek létre, és egyszerre is szűnnek meg). További

speciális kapcsolat az osztályok között a *származtatás*, amelyet ezen tárgy keretein belül nem tárgyalunk.

Metódusok túlterhelése

Egy metódust a neve és a paramétereinek listája egyértelműen azonosít, ezeket együtt a metódus *szignatúrájának* nevezzük[1]. Lehetőségünk van egy már létező metódushoz eltérő implementációt (új működést) adnunk, amennyiben a két metódus paraméterlistáját különbözőnek definiáljuk. Ilyen esetben a metódus *túlterheléséről* (overloading) beszélünk, és meghíváskor az átadott paraméterek típusa és sorrendje alapján dől el, melyik megvalósítás fog lefutni. Az alábbi példában a `Point` osztály **Show** metódusa három különböző módon hívható meg: paraméter nélkül, egyetlen karakterrel, illetve egy karakterrel és egy színnel paraméterezve.

Természetesen lehetőségünk van valamely metódusban egy másik metódus, illetve egy túlterhelt változat hívására is, amely hasznos lehet olyan esetekben, amikor el szeretnénk kerülni bizonyos kódsorok ismételt leírását, ha azt egy másik metódus már megvalósította. Erre mutat egyszerű példát a **Show** metódus kétparaméteres változata.

```
class Point
{
    int positionX;
    int positionY;

    public void Show()
    {
        Console.SetCursorPosition(positionX, positionY);
        Console.Write('*');
    }
    public void Show(char symbol)
    {
        Console.SetCursorPosition(positionX, positionY);
        Console.Write(symbol);
    }
    public void Show(char symbol, ConsoleColor color)
    {
        Console.ForegroundColor = color;
        Show(symbol);
    }
}
```

A metódusok túlterhelésének speciális esete a konstruktorok túlterhelése. Amennyiben egy konstruktor már megvalósít bizonyos működést, amelyet szeretnénk kihasználni egy másik konstruktorban, akkor a metódusoknál megismertektől eltérően *konstruktor hívási láncot* kell létrehoznunk az alábbi példához hasonló módon a `this` kulcsszó használatával. Ilyen esetekben az elsőként hívott konstruktor törzsében felsorolt utasítások végrehajtása előtt a másodikként megjelölt konstruktor fut le előbb, ezt követi csak az elsőként hívottban felsorolt utasítások végrehajtása.

Az alábbi példában egy személyt létrehozhatunk kizárólag a neve megadásával (ilyenkor minden más mező értékét a típusnak megfelelő alapértékre állítja a konstruktor). Létrehozhatjuk ugyanakkor a személyt a neve, magassága, testtömege és testtömeg-indexe megadásával is, a név beállítását azonban elvégeztethetjük a korábban már megírt

[1] Kontextustól függően a szignatúrának része a visszatérési érték típusa is, lásd bővebben itt: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/methods#method-signatures

egyparaméteres konstruktorral is. Hasonlóan, a testtömeg-index kiszámítása a magasság és testtömeg alapján könnyen megoldható (vagyis ezt nem feltétlenül kell külön paraméterként elkérnünk), tehát a három paraméterrel rendelkező konstruktor meghívhatja a négyparaméteres változatot a mezők inicializálásához.

```
class Person
{
    string name;
    double height;
    double weight;
    double BMI;

    public Person(string name)
    {
        this.name = name;
    }

    public Person(string name, double height, double weight, double BMI) : this(name)
    {
        this.height = height;
        this.weight = weight;
        this.BMI = BMI;
    }

    public Person(string name, double height, double weight) : this(name, height,
       weight, weight/(height*height))
    {
    }
}
```

Tulajdonságok

Az egységbezárás elvének megfelelően egy objektum mezői privát elérésűek, vagyis külső osztályok számára közvetlenül nem hozzáférhetőek. Az objektum adatainak lekérdezése és módosítása épp ezért általában metódusokon keresztül történik. A C# nyelv ennek a folyamatnak a megkönnyítésére úgynevezett *tulajdonságok* (properties) használatát kínálja fel. A tulajdonságok speciális tagok az osztályokban, amelyek lehetővé teszik a mezők kontrollált elérését és beállítását: olyan metódusokról van szó, amelyek használat során mezőként viselkednek.

Egy tulajdonság általában két részből áll: a **get** rész az olvasásért (adatlekérésért), míg a **set** rész az írásért (adatbeállításért) felel. Az alábbi példában a `Circle` osztály egyetlen privát adatmezőt deklarál, amelyhez a tulajdonságon keresztül férhetünk hozzá más osztályokból. Szokás a tulajdonság nevét a kapcsolódó mező nevével egyezőnek választani, és nagy kezdőbetűvel ellátni. A **get** részben lévő kódrészlet ebben az esetben közvetlenül a háttérben lévő mező értékét szolgáltatja vissza. A **set** részben a beállítani kívánt értékre (ez a **Main** metódusban 2.5 lesz) a `value` kulcsszóval hivatkozhatunk. A példában a kör sugarának beállítása csak pozitív érték esetén történik meg, így megakadályozhatjuk a téves értékadásból származó hibás működést.

Lehetséges csak **get** vagy csak **set** részek deklarálása is, illetve ezekben a részekben akár komplexebb kódrészletet is megadhatunk. Erre mutat példát az osztály **Area** tulajdonsága, amelyen keresztül a kör területét kérdezhetjük le (számítás eredményeként, nem közvetlenül egy mező értékét visszaadva), azonban mivel **set** részt nem hoztunk

létre, így a terület beállítása a tulajdonságon keresztül nem lehetséges. A **get** és **set** részek hozzáférhetősége szükséges esetben szigorítható az eléjük írt módosító kulcsszó használatával, így elérhető például, hogy egy deklarált **set** rész csak az osztályon belül legyen hozzáférhető, míg a **get** rész publikus más osztályok számára is.

```
class Circle
{
    double radius;
    public double Radius
    {
        get { return radius; }
        set {
            if (0 < value)
            {
                radius = value;
            }
        }
    }
    public double Area
    {
        get { return Math.Pow(radius, 2) * Math.PI; }
    }
}

class Program
{
    static void Main(string[] args)
    {
        Circle circle = new Circle();
        circle.Radius = 2.5;
        Console.WriteLine($"Area of the circle is {circle.Area}.");
        circle.Area = 10; // !!! READ ONLY !!!
    }
}
```

Az úgynevezett *automatikusan megvalósított tulajdonságok* (Auto-Implemented Properties, AIP) egyszerű és rövid tulajdonságdeklarációt tesznek lehetővé, és olyan esetekben előnyösek, amikor a mezők hozzáférésekor nincs szükség komplexebb logika megvalósítására. AIP használatakor a fordító egy privát, névtelen mezőt hoz létre a háttérben, amelyet csak a tulajdonság **get** és **set** tagjain keresztül lehet elérni.

```
class ComplexNumber
{
    public double RealPart { get; set; }
    public double ImaginaryPart { get; set; }
}
```

Felsorolt típus (enum)

Felsorolt típus alatt előre definiált értékek (címkék) halmazát értjük. Ilyen típus használata célszerű lehet, amikor olyan adatokat akarunk tárolni, amelyeknek csak egy meghatározott számú értékük lehet (például napok nevei,

vagy az iskolai értékelésnél használt osztályzatok megnevezései). Saját felsorolt típust az enum kulcsszóval, a típus nevének, valamint lehetséges értékeinek megadásával definiálhatunk az alábbihoz hasonló módon.

```
enum Days { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday }
// Monday=0, Tuesday=1, Wednesday=2, ...
```

A címkéket egész értékek azonosítják, amelyek egyénileg beállíthatók. Ha nem rendelkezünk másképp, a felsorolt típus első címkéje a 0, a következő az 1, stb. értékeket kapja.

```
enum Days { Monday=3, Tuesday, Wednesday, Thursday, Friday=420, Saturday, Sunday }
// Monday=3, Tuesday=4, ... Friday=420, Saturday=421, ...
```

Egy felsorolt típusú változó értékei ezt követően csak az előre definiált értékek közül kerülhetnek ki.

```
Days myDay = Days.Friday;
```

A felsorolt értékek egész értékké konvertálhatók, illetve megfelelő egész értékeket felsorolt értékekké konvertálhatunk az alábbihoz hasonló módon.

```
Days myDay = Days.Friday;
int fridayAsInt = (int)myDay;   // 420
int sundayAsInt = 422;
Days anotherDay = (Days)sundayAsInt;    // Days.Sunday
```

Lehetőségünk van a felsorolt típus értékét karakterlánccá alakítani a megszokott módon, a **ToString** metódus használatával.

```
Days myDay = Days.Friday;
string myDayString = myDay.ToString();   // "Friday"
```

Leteséges karakterláncokból is kinyerni a címkék értékét az Enum osztály **Parse** nevű metódusával. A metódus első paramétere az a *típus*, amelynek az értékeit kívánjuk beolvasni, második paramétere a feldolgozandó karakterlánc. A metódus által visszaadott értéket utolsó lépésként át kell alakítanunk a kívánt felsorolt típusba, az alább látható módon.

```
Days myDay = (Days)Enum.Parse(typeof(Days), "Friday");   // Days.Friday
```

Feladatok

1a Hozzunk létre egy `ExamResult` osztályt, amely egy ZH dolgozat eredményét reprezentálja. Az osztály rendelkezze egy Neptun-kódot valamint egy 0-100 közötti pontszámot tároló mezővel. Készítsünk publikus tulajdonságokat a két mezőhöz, amelyek csak formailag helyes értékek beállítását végzik el (tehát például nem állíthatunk be vele 5 karakterből álló Neptun kódot, vagy negatív pontszámot). Készítsünk konstruktort, amely a Neptun-kódot és a pontszámot kéri el, és elvégzi a mezők beállítását a tulajdonságokon keresztül. Hozzunk létre egy olyan konstruktort is, amely segítségével véletlenszerű Neptun-kóddal és pontszámmal hozhatunk létre egy példányt. Készítsünk egy **Passed** nevű tulajdonságot, amely `true` vagy `false` értékkel tér vissza attól függően, hogy a dolgozat sikeres (legalább 50 pontos) volt-e.

1b Készítsünk egy **Grade** metódust is, amelynek paramétere egy ötelemű, egészekből álló tömb, az osztályzatok ponthatárai (például `{0, 50, 62, 74, 86 }`). A metódus a pontszám alapján kalkulált érdemjegy megnevezését (pl. `Jeles`, `Jó`, stb.) adja vissza egy saját felsorolt típusként. Hozzuk létre a szükséges felsorolt típust is az osztályzatok megnevezésével.

1c Kérjünk el a felhasználótól egy N egész értéket, majd hozzunk létre egy N elemű gyűjteményt `ExamResult` típusú objektumok tárolására. Töltsük fel a gyűjteményt a felhasználó által megadott vagy véletlenszerűen generált értékeket tartalmazó példányokkal. Soroljuk fel a sikeres dolgozatokhoz tartozó Neptun-kódokat. Írjuk ki a pontszámok átlagát, illetve a legmagasabb pontszámhoz tartozó Neptun-kódot.

2 Készítsük el a *Whac-A-Mole* játék haladó változatát, ahol azt kell megtippelnünk, hol fog felbukkanni a vakond. Hozzunk létre egy `Mole` osztályt, amely tárolja a vakond pozícióját (egész érték). A pozíció legyen lekérhető (de nem módosítható) egy publikus tulajdonságon keresztül. Hozzunk létre egy **TurnUp** nevű metódust, amely meghíváskor egy `M` karaktert ír ki a képernyőre a vakond pozíciójának megfelelő helyre. Készítsünk még egy **Hide** nevű kétparaméteres metódust is, amely a vakond pozícióját véletlen értékre állítja úgy, hogy az az első és második paraméterként megadott egészek közé essen.

Készítsünk egy példányt, és hívjuk meg a **Hide** metódusát. Kérjük el a felhasználótól egy tippet, rajzoljuk ki a vakondot, majd döntsünk róla, hogy a felhasználó eltalálta-e a pozícióját. A játék akkor ér véget, ha a felhasználó elkapta a vakondot.