OOP alapjai: osztályok szerkezete

Összefoglaló

Az objektumorientált programozás (OOP) egy olyan módszertan (paradigma), amely a logikailag összetartozó adatokat és a rajtuk végezhető műveleteket úgynevezett *objektumokba* csomagolja, ez az *egységbezárás* alapelve. Gondoljunk például egy egyszerű bankalkalmazásra: az ügyfeleket, a bankszámlákat és a tranzakciókat modellezhetjük objektumokként. Az ügyfelek egy osztályt alkothatnak, amely tartalmazza az ügyfél nevét, címét és számlaszámát (ezek a logikailag az ügyfélhez tartozó adatok). A bankszámlák szintén osztályként definiálhatók, és tartalmazhatják az aktuális egyenleget és tranzakciókat, a tranzakciók pedig további objektumok lehetnek, amelyek tartalmazzák az összeget és a dátumot. Minden objektumnak van egy meghatározott szerepe és felelőssége: az ügyfél például pénzt helyezhet be vagy veszhet le a bankszámlájáról (ez az objektum adatain végzett műveleteket jelenti), a bankszámla számítja az egyenleget és rögzíti a tranzakciókat, míg a tranzakció egyszerűen rögzíti az összeget és a dátumot.

Nézzük meg az előbbi példában szereplő *ügyfél* egy lehetséges modelljét, amely csak az adatait tárolja (tehát műveleteket egyelőre nem valósít meg).

```
namespace BankApp
{
    class BankClient
    {
        public string name;
        string userID;
        private DateTime dateOfBirth;
        public double balance;
    }

    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Látható, hogy a `BankClient` osztály elkülönül a **Main** metódust tartalmazó osztálytól. A `BankClient` ebben a példában négy *tagváltozót* vagy *adatmezőt* tartalmaz, amelyek lényegében közönséges változók, az egyetlen speciális jellemzőjük, hogy mind ugyannak az osztálynak a változói, vagyis egyetlen egység részei. Az osztály tehát a banki ügyfelek egyféle sablonja: nem azt definiálja, hogy pontosan mi kell legyen az ügyfél neve, azonosítója, születési dátuma vagy egyenlege, hiszen ez minden ügyfélnél különböző lehet, de előírja azt, hogy minden ügyfélnek rendelkeznie kell ezekkel a jellemzőkkel.

Példányosítás

Az alábbi kódrészletben két *példányt* (vagy más néven objektumot) készítünk el, a példányosításhoz a `new` operátort kell használnunk (ahogy tettük azt például a `Random` osztály esetében is). A `clientOne` és `clientTwo` nevű változókon keresztül egy-egy olyan példányt érünk el ezt követően, amelyeknek saját neve, azonosítója, születési dátuma és egyenlege van. Ami közös bennük, hogy mindketten ugyanabból az osztályból (sablonból) lettek példányosítva.

```
static void Main(string[] args)
{
    BankClient clientOne = new BankClient();
    BankClient clientTwo = new BankClient();
}
```

Taghozzáférés

Lehetőségünk van a példányok tagjainak (mezőinek és metódusainak) külső elérésére, amennyiben azokat az osztályban elláttuk a `public` módosítóval. Amely tag előtt nem szerepel ez, azok belső (privát) tagok, és kívülről nem hozzáférhetők. (Ezt az oszályban külön is jelezhetjük a `private` kulcsszó kiírásával.)

```
BankClient clientOne = new BankClient();
BankClient clientTwo = new BankClient();

clientOne.name = "Mike Ehrmantraut";
clientTwo.name = "Gustavo Fring";
clientTwo.balance = 123456789.0;
clientOne.userID = "MIKE123"; // !!! INACCESSIBLE !!!
Console.WriteLine(clientTwo.balance);
```

Tagfüggvények (metódusok)

Az osztályban az adatmezőkön felül úgynevezett *tagfüggvényeket* vagy metódusokat is definiálhatunk, amelyek utasításokat tartalmaznak. A metódus *meghívása* ezen utasítások lefuttatását jelenti, és ennek segítségével tudunk például az objektum tagváltozóival dolgozni (beállítani, lekérdezni, módosítani), amely megfeleltethető a bevezetőben említett műveletvégzésnek. Az alábbi osztályban két metódust definiálunk.

A **SetName** metódus egy karakterláncot kér paraméterként, és annak megfelelően állítja be a példány adattagjának értékét. Ettől a metódustól nem várunk *visszatérési értéket*, ezt a neve elé írt `void` kulcsszóval kell jeleznünk. Mivel azt szeretnénk, hogy a metódust külső programrészek is elérjék, így a láthatóságát publikusra állítjuk.

A **GetFirstnameAndLastname** metódus paraméter nélküli (vagyis meghíváskor nem kell argumentumként valamilyen adatot átadnunk neki). A metódus egy karakterlánc tömböt ad vissza (ezt jelezünk kell a neve előtt a típus megadásával), majd a metódus *törzsében* (tipikusan utolsó utasításként) a `return` kulcsszóval egy előírtnak megfelelő típusú értéket kell visszaadnunk.

Az osztályban ilyen módon definiált metódusokat az osztályból készített példányon keresztül tudjuk meghívni.

```
class BankClient
{
    private string name;

    public void SetName(string clientName)
    {
        name = clientName;
    }

    public string[] GetFirstnameAndLastname()
    {
        string[] firstAndLast = name.Split(' ');
        return firstAndLast;
    }
}

class Program
{
    static void Main(string[] args)
    {
        BankClient clientOne = new BankClient();
        BankClient clientTwo = new BankClient();

        clientOne.SetName("Eduardo Salamanca");
        clientTwo.SetName("Nacho Varga");
        string[] clientOneFullName = clientOne.GetFirstnameAndLastname();
        // { "Eduardo", "Salamanca" }
    }
}
```

Konstruktor

Egy osztály konstruktora egy speciális metódus az osztályban a példányok elkészítésére. A konstruktor feladata az objektum mezőinek inicializálása, vagyis azok kezdeti értékekkel történő feltöltése. A konstruktor felépítése hasonló az előbb említett metódusokéhoz, vagyis rendelkezhet paraméterekkel, azonban neve kötelezően az osztály nevével kell megegyezzen, és visszatérési értéket nem írhatunk elő. Az alábbi kódrészletben az oszály kétparaméteres konstruktora az átadott értékeknek megfelelően állítja be az ügyfél nevét és azonosítóját, a számlaegyenleg azonban minden esetben 1.0 kezdőértéket kap.

Gyakori, hogy egy metódus valamely paraméterének ugyanaz a neve, mint az osztály egyik tagváltozójának. Ilyen esetben különbséget tehetünk a két szimbólum között azzal, hogy a mező neve elé írjuk a `this` kulcsszót. A kulcsszó a program futtatásakor azt a példányt reprezentálja, amelyen keresztül a taghozzáférést kezdeményeztük. Az alábbi példában ez azt jelenti, hogy az első konstruktorhíváskor az átadott azonosító a `clientOne` változóban tárolt példány `userID` mezőjének értéke lesz, a második konstruktorhíváskor az átadott érték viszont értelemszerűen a `clientTwo` példány mezőjében kerül eltárolásra. A `this` kulcsszó egyértelműsítés céljából egyébként máskor is kiírható a tagok elé.

```
class BankClient
{
    private string name;
    private string userID;
    private double balance;

    public BankClient(string clientName, string userID)
    {
        name = clientName;
        this.userID = userID;
        this.balance = 1.0;
    }
}

class Program
{
    static void Main(string[] args)
    {
        BankClient clientOne = new BankClient("Eduardo Salamanca", "LAL042");
        BankClient clientTwo = new BankClient("Hector Salamanca", "TIO1939");
    }
}
```

Láthatósági szabályok

Az egységbezárás alapelvéhez szorosan kapcsolódik az *adatrejtés* alapelve, amely azt a célt szolgálja, hogy az objektumok állapotát a „külvilágból" csak erre definiált metódusokon keresztül lehessen módosítani, vagyis csak közvetett módon lehessen elérni őket. Az elv alkalmazásával megakadályozhatjuk az érvénytelen állapotok létrehozását az objektumokban és kivédhetjük az olyan hibákat, amelyek akkor történhetnek, amikor az objektumok adatait közvetlenül módosítják az osztályon kívülről.

Ennek megfelelően az osztály semelyik mezőjének láthatóságát sem javasolt a `public` kulcsszóval nyilvánossá tenni, továbbá az osztály tagfüggvényei közül is csak azokat javasolt publikusnak jelölni, amelyek külső elérése mindenképp szükséges.

Az alábbi példában a `BankClient` osztály minden mezője privát láthatóságú (mivel ez az alapértelmezett), vagyis kívülről nem érhetőek el. Ettől függetlenül, mivel az osztály részei, így az osztály minden tagfüggvénye eléri őket. A belső használatra létrehozott **HasPositiveBalance** segédmetódust az osztályon kívülről nem tudjuk meghívni, azonban az osztály bármely metódusa meghívhatja, például a **GetClientStatus** metódus is. Figyeljük meg, hogy az osztály metódusai hozzáférnek az osztály tagjaihoz (tagváltozókhoz és tagfüggvényekhez is), ezen felül pedig szükség esetén paramétereket is előírhatunk, így meghívásukkor az oszályon kívüli adatokkal is el tudjuk látni őket.

```
class BankClient
{
    string userID;
    double balance;

    public BankClient(string userID)
    {
        this.userID = userID;
        this.balance = 1.0;
    }

    private bool HasPositiveBalance()
    {
        bool positiveBalance = this.balance > 0;
        return positiveBalance;
    }

    public string GetClientStatus(DateTime currentDate)
    {
        if (this.HasPositiveBalance())
        {
            return $"UserID {userID} has positive balance at {currentDate}";
        }
        else
        {
            return $"UserID {userID} does not have a positive balance at
                {currentDate}";
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        BankClient tio = new BankClient("TIO1939");
        string status = tio.GetClientStatus(DateTime.Now);
        // UserID TIO1939 has positive balance at 2023. 10. 23. 20:23:10
    }
}
```

Feladatok

1 Készítsünk egy `Book` osztályt, amely eltárolja egy könyv szerzőjét, címét, a kiadás évét és a könyv oldalszámát. Az adattagok értékét a konstruktoron keresztül lehessen beállítani. Hozzunk létre egy publikus **AllData** nevű metódust, amely a könyv adatait egyetlen formázott karakterláncban adja vissza.

Példa

```
Book b = new Book("The Hobbit - or There and Back Again", "J.R.R. Tolkien", 1937, 312);
Console.WriteLine(b.AllData());
// J.R.R. Tolkien: The Hobbit - or There and Back Again, 1937 (312 pages)
```

2 Készítsünk osztályt egy téglalap reprezentálására. A téglalapról tároljuk el, hogy hány egység széles, hány egység magas, továbbá hogy milyen színű. Legyen lehetőség az adattagok beállítására a konstruktoron keresztül. Az osztálynak legyen egy privát **Area** és egy publikus **IsValid** nevű metódusa. Az előbbi a példány területét határozza meg, az utóbbi pedig egy logikai változóban visszaadja, hogy a példány szerkeszthető-e (nullánál nagyobb-e a területe). Hozzunk létre egy kétparaméteres **Draw** nevű metódust is, amely nem rendelkezik visszatérési értékkel. A metódus az osztályban tárolt színnel és méretekkel kirajzolja a képernyőre a téglalapot, a bal felső sarkát a metódus két paraméterének (x és y) megfelelően igazítva. A főprogramban hozzunk létre néhány téglalap példányt, ellenőrizzük, hogy szerkeszthetőek-e, majd rajzoljuk ki, amelyek azok.

3 Futóverseny szimulációjához készítünk alkalmazást. Hozzunk létre egy `Runner` nevű osztályt, amely tárolja a futó nevét, sorszámát, sebességét (m/s), és a startvonaltól számított aktuális távolságát. Az adattagoknak a konstruktoron keresztül lehessen kezdőértéket adni, a távolság minden esetben a 0 kezdőértéket kapja. Az osztály rendelkezzen egy **RefreshDistance** nevű metódussal, amely egy másodpercben megadott időtartamot jelentő egész számot vár paraméterként, és a példány távolságát minden hívásnál a sebességéből és a kapott időtartamból számított értékkel növeli meg. A **Show** metódus a futó sorszámának megfelelő sorba, a képernyő bal szélétől a távolságának megfelelő oszlopba írja a futó nevének kezdőbetűjét. A **GetDistance** metódus a futó aktuális távolságát adja vissza. A főprogramban hozzunk létre két futó példányt, majd léptessük és jelenítsük meg őket annyi alkalommal a képernyőn, amíg egyikük távolsága elér egy előre megadott értéket.

4 Készítsünk egy szöveges üzenet titkosítására és visszafejtésére alkalmas osztályt. Az üzenet titkosítása történjen a korábban megismert Caesar-rejtjelezéssel. A példány tárolja el a kódoláshoz használt kulcsot (az eltolás mértékét), amelyet példányosításkor lehessen megadni. Definiáljunk egy privát **TransformMessage** nevű metódust, amelynek két paramétere van: egy karakterlánc (az üzenet) és egy egész érték (a kulcs). A metódus visszatérési értéke a karakterek megfelelő eltolásával kapott üzenet. Definiáljunk továbbá egy **Encode** és egy **Decode** nevű metódust is, amelyek egyetlen paramétere egy üzenet, és a példányban tárolt kulcsot felhasználva meghívja a **TransformMessage** metódust, majd visszaadja az ez által kódolt vagy visszafejtett üzenetet.

5 Készítsük el a korábban már megoldott NHANES adatbázisból származó adatok feldolgozására képes programunkat objektumorientált megközelítésben is. Hozzunk létre egy osztályt egy személy adatainak tárolására. A mezők inicializálása történjen a konstruktornak átadott paraméterek alapján. Írjunk tagfüggvényeket a szükséges adatmezők érékének lekérdezésére. Dolgozzuk fel a bemeneti fájlt, majd a tábla soraiban lévő adatok alapján előálló példányokat tároljuk egy tömbben vagy listában. Írjuk meg a korábbi lekérdezéseket a példányok metódusain keresztüli adathozzáféréssel is.