Laborator 9

Structura unei aplicaţii orientată pe obiecte în C#

Limbajul C# permite utilizarea programării orientate pe obiecte respectând toate principiile enunţate anterior.

Toate componentele limbajului sunt într-un fel sau altul, asociate noţiunii de clasă. Programul însuşi este o clasă având metoda statică **Main()** ca punct de intrare, clasă ce nu se instanţiază.

Chiar şi tipurile predefinite **byte, int** sau **bool** sunt clase sigilate derivate din clasa **ValueType** din spaţiul **System**. Tot din ierarhia de clase oferită de limbaj se obţin şi tipuri speciale cum ar fi: interfeţe, delegări şi atribute. Începând cu versiunea 2.0 a limbajului i s-a adăugat un nou tip: clasele generice, echivalentul claselor **template** din C++.

În cele ce urmează vom analiza, fără a intra în detalii o aplicaţie POO simplă în C#.

Clasă de bază şi clase derivate

Să definim o clasă numită Copil:

```
public class Copil { }
```

unde:

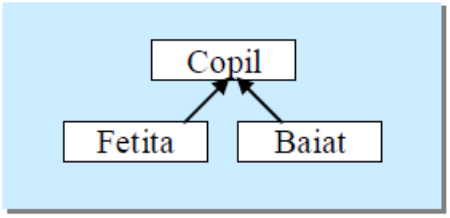
public – sunt modificatori de acces.

class – cuvânt rezervat pentru noţiunea de clasă

Copil – numele clasei

{ } – corpul clasei

Dacă considerăm clasa Copil ca şi clasă de bază, putem deriva două clase Fetiţa şi Băiat




```
public class Fetita: Copil { }
public sealed class Baiat: Copil { }
```

unde:

modificatorul **sealed** a fost folosit pentru a desemna faptul că nu se mai pot ob_ine clase derivate din clasa Baiat

Constructori

Înainte de a continua amintim câteva noţiuni legate de constructorii unei clase:

Constructorul este o funcţie care face parte din corpul unei clase. Corpul constructorului este format din instrucţiuni care se execută la crearea unui nou obiect al clasei respective (sau la crearea clasei, în cazul constructorilor cu modificatorul static).

- pot exista mai mulţi constructori care se pot diferenţia prin lista lor de parametri
- constructorii nu pot fi moşteniţi

- dacă o clasă nu are definit niciun constructor, se va asigna automat constructorul fără parametri al clasei de bază (clasa object, dacă nu este precizată clasa de bază)

Instanţierea presupune declararea unei variabile de tipul clasei respective şi iniţializarea acesteia prin apelul constructorului clasei (unul dintre ei, dacă sunt definiţi mai mulţi) precedat de operatorul new.

Reluăm exemplu de mai sus în care vom prezenta un constructor fără parametri şi constructorul implicit din clasa derivată. Vom adăuga un constructor fără parametri. La iniţializarea obiectului se va citi de la tastatură un şir de caractere care va reprezenta numele copilului.

Exemplul 60:

```
public class Copil
{
     protected string nume;  //data accesibila numai in interiorul
                             //clasei si a claselor derivate
     public Copil ( )        //constructorul fara parametrii ai clasei
     {
          nume = Console.ReadLine( );
     }
}

class Fetita: Copil
{ }
...
Fetita f = new Fetita ( );
Copil c = new Copil ( );
```

Supraîncărcarea constructorilor şi definirea constructorilor în clasele derivate

Reluăm exemplul anterior şi îl dezvoltăm:

```
public class Copil
{
     protected string nume;  //data accesibila numai in interiorul
                             //clasei si a claselor derivate
     public Copil ( )        //constructorul fara parametrii ai clasei
     {nume = Console.ReadLine( );}
     public Copil (string s)  //constructor cu parametru
     {nume = s;}
}
class Fetita: Copil
{
     public Fetita (string s): base(s) //base semnifica faptul ca
{                              //se face apel la
          nume = "Fetita "+ nume;      //constructorul
  //din clasa de baza
} }
...
Copil c1 = new Copil ( );    //numele copilului se citeste de la
                             //tastatura
Copil c2 = new Copil ("Gigel"); //numele lui c2 va fi Gigel
Fetita f1 = new Fetita ( );
Fetita f2 = new Fetita ("Maria");
```

Destructor

Corpul destructorului este format din instrucţiuni care se execută la distrugerea unui obiect al clasei respective. Pentru orice clasă poate fi definit un singur constructor. Destructorii nu pot fi moşteniţi. În mod normal,

destructorul nu este apelat în mod explicit, deoarece procesul de distrugere a unui obiect este invocat şi gestionat automat de **Garbage Collector**

Metode

Din corpul unei clase pot face parte şi alte funcţii: metodele. Exemplificarea o vom face tot pe exemplul anterior.

Exemplul 61:

```
public class Copil
{
    protected string nume;  //data accesibila numai in interiorul
    //clasei si a claselor derivate
    public const int nr_max = 10;     //constanta
    public static int nr_copii = 0;  //camp simplu (variabila)
    static Copil[] copii = new Copil[nr_max]; //camp de tip
    //tablou (variabila)
    public static void adaug_copil(Copil c) //metodă
    {
        copii[nr_copii++] = c;
        if (nr_copii == nr_max)
            throw new Exception("Prea multi copii");
    }
    public static void afisare()           //metodă
    {
        Console.WriteLine("Sunt {0} copii:", nr_copii);
        for (int i = 0; i < nr_copii; i++)
            Console.WriteLine("Nr.{0}. {1}", i + 1, copii[i].nume);
    }
    public Copil()        //constructorul fara parametrii ai clasei
    {
        nume = Console.ReadLine();
    }

    public Copil(string s)       //constructor cu parametru
    {
        nume = s;
    }
}
class Fetita : Copil
{
    public Fetita(string s)
        : base(s)                  //base semnifica faptul ca
    {                              //se face apel la
        nume = "Fetita " + nume;  //constructorul
        //din clasa de baza
    }
}
```

```
Fetita c = new Fetita();
Copil.adaug_copil(c);
//referința noului obiect se memorează în tabloul static copii
//(caracteristic clasei) şi se incrementează data statică nr_copii
Baiat c = new Baiat();
Copil.adaug_copil(c);
Copil c = new Copil();
Copil.adaug_copil(c);
Copil.afisare();    //se afişează o listă cu numele celor 3 copii

...
```

Definirea datelor şi metodelor **nestatice** corespunzătoare clasei Copil şi claselor derivate

Exemplul 62:

```
public class Copil
{
     protected string nume;
     ...

     public virtual void se_joaca( )      //virtual - functia se poate
     {                                    //suprascrie la derivare
          Console.WriteLine("{0} se joaca.", this.nume);
     }
     public void se_joaca(string jucaria)  //supradefinirea metodei
     {                                      //se_joaca
          Console.WriteLine("{0} se joaca cu {1}.",this.nume,jucaria);
     }

     ...
}
class Fetita: Copil
{
     public override void se_joaca( )     //redefinire
     {
          Console.WriteLine("{0} chinuie pisica.", this.nume);
}
}
...
//polimorfism
Fetita f = new Fetita( );
f.se_joaca("pisica");
f.se_joaca( );
Baiat b = new Baiat ( );
b.se_joaca("calculatorul");
b.se_joaca( );
```

Proprietăţi

Proprietăţile sunt asemănătoare cu metodele în ceea ce priveşte modificatorii şi numele metodelor. Metodele de acces sunt două: set şi get. Dacă proprietatea nu este abstractă sau externă, poate să apară una singură dintre cele două metode de acces sau amândouă, în orice ordine.

Este o manieră de lucru recomandabilă aceea de a proteja datele membru (câmpuri) ale clasei, definind instrumente de acces la acestea: pentru a obţine valoarea câmpului respectiv (**get**) sau de a memora o anumită valoare în câmpul respectiv (**set**). Dacă metoda de acces get este perfect asimilabilă cu o metodă ce returnează o valoare (valoarea datei pe care vrem s-o obţinem sau valoarea ei modificată conform unei prelucrări suplimentare specifice problemei în cauză), metoda **set** este asimilabilă cu o metodă care un parametru de tip valoare (de intrare) şi care atribuie (sau nu, în funcţie de context) valoarea respectivă câmpului. Cum parametrul corespunzător valorii transmise nu apare în structura sintactică a metodei, este de ştiut că el este implicit identificat prin cuvântul value. Dacă se supune unor condiţii specifice problemei, se face o atribuire de felul câmp=**value**.

Definirea în clasa Copil a proprietăţii Nume, corespunzătoare câmpului protejat ce reţine, sub forma unui şir de caractere, numele copilului respectiv. Se va observă că proprietatea este moştenită şi de clasele derivate Fetiţa şi Băiat.

Exemplul 63:

```
public class Copil
{...
    string nume; // este implicit protected
    public string Nume //proprietatea Nume
    {  get
       { if(char.IsUpper(nume[0]))
             return nume;
         else
             return nume.ToUpper();
       }
       set
       {
         nume = value;
       }
    }
    public Copil()                    //metoda set
    {
       Nume = Console.ReadLine( );
    }
}

class Fetita:Copil
{
    public override void se_joaca()  //metoda get
    {
       Console.WriteLine("{0} leagana papusa.",this.Nume);
    }
}
```

Concluzie

Scrierea unui program orientat obiect implică determinarea obiectelor necesare; acestea vor realiza prelucrările care definesc comportarea sistemului. Obiectele sunt responsabile pentru modificarea datelor proprii.

În proiectarea unei aplicaţii POO parcurgem următoarele etape:

1. identificarea entităţilor, adică a obiectelor care apar în domeniul aplicaţiei, prin evidenţierea substantivelor din enunţul problemei
2. pentru fiecare obiect se identifică datele şi operaţiile, prin evidenţierea verbelor şi adjectivelor care caracterizează subiectul respectiv
3. identificarea relaţiilor dintre entităţi
4. crearea unei ierarhii de clase, pornind de la aceste entităţi
5. implementarea claselor şi a sistemului
6. testarea şi punerea la punct.