Laborator 10

Clase şi obiecte

Clase

Clasele reprezintă tipuri referinţă definite de utilizator.

O aplicaţie C# este formată din una sau mai multe clase, grupate în spaţii de nume - **namespaces**. În mod obligatoriu, doar una dintre aceste clase conţine un punct de intrare - **entry point**, şi anume metoda **Main**.

Sintaxa:

```
[atribut][modificatorAcces] class
[identificator][:clasaBaza]
{
     corpul_clasei
}
```

unde:

atribut – este opţional, reprezentând informaţii declarative cu privire la entitatea definită

modificatorAcces - este opţional, iar în cazul în care lipseşte se consideră **public**

modificatorAcces	Explicații
public	acces nelimitat, clasa este vizibilă peste tot
internal	acces permis doar în clasa sau spațiul de nume care o cuprinde
protected	acces în clasa curentă sau în cele derivate
private	modificator implicit. Acces permis doar pentru clase interioare
protected internal	folosit pentru clase interioare semnificând accesul în clasa care-l conține sau în tipurile derivate din clasa care-l conține
new	permis claselor interioare. Clasa cu acest modificator ascunde un membru cu acelaşi nume care este moştenit
sealed	clasa nu poate fi moştenită
abstract	clasa nu poate fi decât clasă de bază, neputând fi instanțiată. Se foloseşte pentru clase interioare sau spații de nume

identificator - este numele clasei

clasaBaza - este opţional, fiind numele clasei de bază, din care derivă clasa actuală.

Exemplul 64: Se consideră clasa **IncludeClase** care include şase clase având modificatori de acces diferiţi. Se pune problema „vizibilităţii” lor din exterior

```
using System;
using System.Collections.Generic;
using System.Text;
namespace AplicatiiClase
{
    public class IncludeClase
    {
        public class Clasa1
        { }
        abstract class Clasa2
        { }
        protected class Clasa3
        { }
        internal class Clasa4
        { }
        private class Clasa5
        { }
        class Clasa6
        { }
    }
class Program
    {
        static void Main(string[] args)
        {
            IncludeClase.Clasa1 a;
            IncludeClase.Clasa2 b; //Eroare,
            //Clasa2 este inaccesibila
            IncludeClase.Clasa3 c; //Eroare,
            //Clasa3 este inaccesibila
            IncludeClase.Clasa4 d;
            IncludeClase.Clasa5 e; //Eroare,
            //Clasa5 este inaccesibila
            IncludeClase.Clasa6 f; //Eroare,
            //Clasa6 este inaccesibila
        }
    }
}
```

Corpul clasei - este alcătuit din:

- **date**
- **funcţii**

Atât datele cât şi funcţiile pot avea ca modificatori de acces:

modificatorAcces	Explicații
public	Membrul este accesibil de oriunde
internal	Membrul este accesibil doar în assembly-ul curent (bloc funcțional al unei aplicații .NET)
protected	Membrul este accesibil oricărui membru al clasei care-l conține şi a claselor derivate
private	Modificator implicit. Accesibil permis doar pentru clasa care-l conține
protected internal	Membrul este accesibil oricărui membru al clasei care-l conține şi a claselor derivate, precum şi în assembly-ul curent

Date

Datele situate într-o clasă sunt desemnate sub numele de **variabile** sau **atribute**. Datele pot fi de orice tip, inclusiv alte clase.

Declararea datelor se face:

```
[modificatorAcces] tipData nume;
```

unde:

modificatorAcces - este opţional. Implicit este **private**.

tipData - reprezintă tipul datei obiectului pe care vrem să-l atribuim.

nume - se referă la numele dat de utilizator obiectului respectiv.

Datele pot fi:

- constante,
- câmpuri.

Constantele - descriu valori fixe, putând fi valori calculate sau dependente de alte constante. În mod obligatoriu valoarea unei astfel de constante trebuie să fie calculată în momentul compilării. Valoarea unei constante se declară prin cuvântul **const**. Sintaxa este:

```
[modificator] const tip identificator = expresieConstanta
```

unde **tip** poate fi: **bool, decimal, sbyte, byte, short, ushort, int, uint, long,**
ulong, char, float, double, enum, string

Constanta mai poate avea ca **modificator de acces: new, public, protected, internal, protected internal, private.**

Exemplul 65:

```
class Constante
{    public const int MAX = 100;
     const string SALUT = "Buna ziua!";
     public const double MIN = MAX / 3.2;
}
```

Câmpul - reprezintă o dată variabilă a unei clase. În afară de modificatorii menţionaţi mai sus, se mai adaugă: **new**, **readonly**, **volatile**, **static**. Opţional, câmpurile pot fi iniţializate cu valori compatibile. Un astfel de câmp se poate folosi fie prin specificarea numelui său, fie printr-o calificare bazată pe numele clasei sau al unui obiect. Sintaxa este:

```
     tip identificator [=valoare]
```

Exemplul 66:

```
class Camp
{
    public int varsta;
    protected string nume;
    private int id = 13;
    int a;      //implicit private
    static void Main(string[] args)
    {
        Camp obiect = new Camp();
        obiect.a = 1;
    }
}
```

Câmpuri de instanţă

În cazul în care într-o declaraţie de câmp nu este inclus modificatorul static, atunci respectivul câmp se va regăsi în orice obiect de tipul clasei curente care va fi instanţiat. Deoarece un astfel de câmp are o valoare specifică fiecărui obiect, accesarea lui se va face folosind numele obiectului:

obiect.a = 1;

Un câmp special este **this** care reprezintă o referinţă la obiectul curent

Câmpuri statice

Dacă într-o declaraţie de câmp apare specificatorul static, câmpul respectiv va aparţine clasei. Accesarea unui astfel de câmp din exteriorul clasei se poate face doar prin intermediul numelui de clasă:

Exemplul 67:

```
class Camp
{
    public static int a = 13;
    static void Main(string[] args)
    {
        Camp.a++;
    }
}
```

Câmpuri readonly

Pentru a declara un câmp **readonly** se va folosi cuvântul **readonly** în declaraţia sa.

Atribuirea se face doar la declararea sa, sau prin intermediul unui constructor:

Exemplul 68:

```
class Camp
{
    public readonly string a = "Exemplu"; //camp readonly initializat
     public readonly string b;
     public class Camp(string b)          //constructor
     {this.b = b;}                        //camp readonly initializat
}
```

În momentul compilării valoarea câmpului readonly nu se presupune a fi cunoscută.

Câmpuri volatile

Câmpurile volatile se declară cu ajutorul cuvântului **volatile**, care poate fi ataşat doar următoarelor tipuri:

-byte, sbyte, short, ushort, int, uint, char, float, bool

- un tip enumerare care are tipul: **byte, sbyte, short, ushort, int, uint**

- un tip referinţă

Iniţializarea câmpurilor

Valorile implicite pe care le iau câmpurile la declararea lor sunt:

tip	valoare
numeric	**0**
bool	**false**
char	**\0**
enum	**0**
referință	**null**

Funcţii

Funcţiile pot fi:

- Constructori
- Destructori
- Metode
- Proprietăţi

Constructori

Definiţie: Constructorii sunt funcţii care folosesc la iniţializarea unei instanţe a clasei.

Constructorii au acelaşi nume cu al clasei. Constructorul poate avea un modificator de acces şi nu returnează nimic. Sintaxa este:

```
modificatorAcces numeConstructor([parametri])[:initializator]
[{
     corp_constructor
}]
```

unde:

initializator – permite invocarea unui constructor anume, înainte de executarea instrucţiunilor care formează corpul constructorului curent. Iniţializatorul poate lua două forme: **base([parametri])** sau **this([parametri])**. Dacă nu se precizează niciun iniţializator, implicit se va asocia **base( )**.

În cazul în care nu definim nici un constructor, C# va crea unul implicit având corpul vid.

Exemplul 69:

```
class Elev
{
     public Elev()    //constructor
     {
     }
}
```

O clasă poate conţine mai mulţi constructori, diferenţiaţi după numărul şi tipul de parametri.

Exemplul 70:

```
class Elev
{
    public string nume;
    public Elev()             //constructor
    {
        nume = "";
    }
    public Elev(string Nume) //constructor
    {
        nume = Nume;
    }
}
```

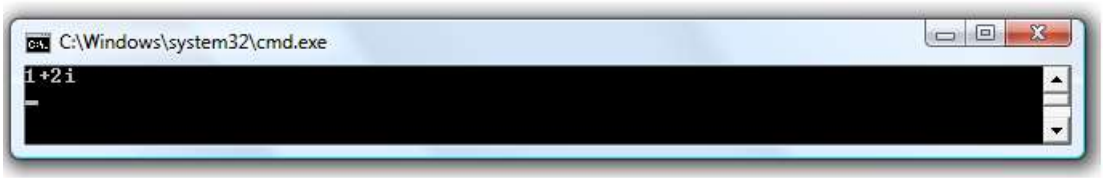
Apelul unui constructor se face automat la instanţierea clasei prin operatorul **new**.

Exemplul 71:

```
class Exemplu_71
{
    Elev elev = new Elev();
}
```

Exemplul 72: Constructor cu doi parametri

```
using System;
namespace Complex
{
    class Complex
    {
        private int re;
        private int im;
        //constructor cu doi parametri
        public Complex(int i, int j)
        {
            re = i;
            im = j;
        }
        public void Afis()
        {
            Console.WriteLine(re + "+" + im + "i");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Complex c = new Complex(1, 2);
            c.Afis();
            Console.ReadLine();
        }
    }
}
```



Observaţie: Constructorii nu pot fi moşteniţi.

Destructori

Destructorul clasei implementează acţiunile necesare distrugerii unei instanţe a clasei.

Numele destructorului coincide cu numele clasei, fiind precedat de caracterul „~”. Destructorul nu are parametri şi nici modificator de acces. Destructorul este apelat automat. Într-o clasă există un singur destructor. Destructorul nu poate fi moştenit.

Exemplul 73:

```
using System;
using System.Collections.Generic;
using System.Text;
namespace Mesaj
{
    class Program
    {
        static void Main(string[] args)
        {
            Mesaj a = new Mesaj();
            Console.ReadLine();
        }
        class Mesaj
        {
            public Mesaj()
            {
                Console.WriteLine("Apel constructor");
            }
            ~Mesaj()
            {
                Console.WriteLine("Apel destructor");
            }
        }
    }
}
```

Metode

Metoda este un membru al unei clase care implementează o acţiune. Metoda poate admite parametri şi returna valori. Tipul returnat de către o metodă poate fi unul predefinit (**int, bool** etc.) sau de tip obiect (**class**). În cazul în care metoda nu returnează nimic, tipul este **void**.

Metodele pot fi supradefinite (supraîncărcate), adică se pot defini mai multe metode, care să poarte acelaşi nume, dar să difere prin numărul şi tipul de parametri. Valoarea returnată de către o metodă nu poate să fie luată în considerare în cazul supradefinirii.

Sintaxa este:

```
modificatorAcces tipReturnat numeMetoda([parametri])
[{
     corp_Metoda
}]
```

unde:

modificatorAcces - este opţional. În cazul în care lipseşte se consideră implicit **private. modificatorAcces** poate fi orice **modificatorAcces** amintit, precum şi **new, static, virtual, sealed, override, abstract, extern**.

tipReturnat – poate fi un tip definit sau **void**.

numeMetoda - poate fi un simplu identificator sau, în cazul în care defineşte în mod explicit un membru al unei interfeţe, numele este de forma:

```
[numeInterfata].[numeMetoda]
```

parametri - lista de parametri formali este o succesiune de declarări despărţite prin virgule, declararea unui parametru având sintaxa:

```
[atribut][modificator] tip nume
```

Modificatorul unui parametru poate fi **ref** (parametru de intrare şi ieşire) sau **out** (parametru care este numai de ieşire). Parametrii care nu au niciun modificator sunt parametri de intrare.

Un parametru formal special este parametrul tablou cu sintaxa:

```
[atribut] params tip [ ] nume
```

Pentru metodele abstracte şi externe, corpul metodei se poate reduce la un semn ;

Semnătura fiecărei metode este formată din numele metodei, modificatorii acesteia, numărul şi tipul parametrilor. Din semnătură (amprentă) nu fac parte tipul returnat, numele parametrilor formali şi nici specificatorii ref şi out.

Numele metodei trebuie să difere de numele oricărui alt membru care nu este metodă.

La apelul metodei, orice parametru trebuie să aibă acelaşi modificator ca la definire

Invocarea unei metode se realizează prin:

[nume_obiect].[nume_metoda] pentru metodele nestatice

[nume_clasă].[nume_metoda] pentru metodele statice

Proprietăţi

Proprietatea este un membru al clasei care ne permite să accedem sau să modificăm caracteristicile unui obiect sau al clasei.

Sintaxa este:

```
[atribut]modificatorAcces tipReturnat numeProprietate
{
     get
     {
     }
     set
     {
     }
}
```

unde:

modificatorAcces - poate fi orice **modificatorAcces** amintit, precum şi **new, static, virtual**, **sealed**, **override**, **abstract**, **extern**.

tipReturnat - poate fi orice tip valid în C#, el specificând tipul folosit de accesorii **get** (tipul valorii returnate) şi **set** (tipul valorii atribuite).

Accesorul **get** corespunde unei metode fără parametri, care returnează o valoare de tipul proprietăţii.

Accesorul **set** corespunde unei metode cu un singur parametru, de tipul proprietăţii şi tip de retur **void**.

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**.

Exemplul 74:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace GetSet
{
    class ClasaMea
    {
        private int x;
        public int P
        {
            get
            {
                Console.WriteLine("get");
                return x;
            }
            set
            {
                Console.WriteLine("set");
                x = value;
            }
        }
     }

     class Program
    {
        public static void Main(string[] args)
        {
            ClasaMea obiect = new ClasaMea();

            //linia urmatoare apeleaza accesorul
            //'set' din proprietatea P si ii
            //paseaza 10 lui 'value'

            obiect.P = 10;

            int xVal = obiect.P;

            // linia urmatoare apeleaza accesorul
            //'get' din proprietatea P

            Console.WriteLine(xVal);
            Console.ReadLine();
        }
    }
}
```