Laborator 11

Funcţii

Funcţiile pot fi:

- Evenimente
- Indexatori
- Operatori

Evenimente şi delegări

Evenimentele sunt membri ai unei clase ce permit clasei sau obiectelor clasei să facă notificări, adică să anunţe celelalte obiecte asupra unor schimbări petrecute la nivelul stării lor.

Clasa furnizoare a unui eveniment **publică** (pune la dispoziţia altor clase) acest lucru printr-o declarare **event** care asociază evenimentului un **delegat**, adică o referinţă către o funcţie necunoscută căreia i se precizează doar antetul, funcţia urmând a fi implementată la nivelul claselor interesate de evenimentul respectiv. Este modul prin care se realizează comunicarea între obiecte.

Tehnica prin care clasele implementează metode (**handler-e**) ce răspund la evenimente generate de alte clase poartă numele de **tratare a evenimentelor**.

Sintaxa:

```
[atribut][modificatorAcces]even tipDelegat nume
```

unde:

modificatorAcces - este la fel ca în cazul metodelor

tipDelegat – este un tip de date, derivat din clasa sigilată **Delegate** din spaţiul **System**.

Definirea unui **tipDelegat** se realizează astfel:

```
[atribut][modificatorAcces] delegate tipRezultat nume[listaParametri])
```

Un delegat se poate defini şi în afara clasei generatoare de evenimente şi poate servi şi altor scopuri în afara tratării evenimentelor

Exemplul 75: dorim să definim o metodă asociată unui vector de numere întregi, metodă ce verifică dacă vectorul este o succesiune crescătoare sau descrescătoare. O implementare „generică” se poate realiza folosind delegări:

```
using System.Linq;
using System.Text;
namespace Delegari
{
    public delegate bool pereche_ok(object t1, object t2);
    public class Vector
    {
        public const int nmax = 4;
        public int[] v = new int[nmax];
        public Vector()
        {
            Random rand = new Random();
            for (int i = 0; i < nmax; i++)
                v[i] = rand.Next(0, 5);
        }
        public void scrie()
        {
            for (int i = 0; i < nmax; i++)
                Console.Write("{0}, ", v[i]);
            Console.WriteLine();
        }
public bool aranj(pereche_ok ok)//ok e o delegare către o
        //funcţie necunoscută
        {
            for (int i = 0; i < nmax - 1; i++)
                if (!ok(v[i], v[i + 1])) return false;
            return true;
        }
    }
    class Program
    {
        public static bool f1(object t1, object t2)
        {
            if ((int)t1 >= (int)t2) return true;
            else return false;
        }
        public static bool f2(object t1, object t2)
        {
            if ((int)t1 <= (int)t2) return true;
            else return false;
        }
        static void Main(string[] args)
        {
            Vector x;
            do
            {
                x = new Vector(); x.scrie();
                if (x.aranj(f1)) Console.WriteLine("Monoton descrescator");
                if (x.aranj(f2)) Console.WriteLine("Monoton crescator");
            } while (Console.ReadKey(true).KeyChar != '\x001B'); //Escape
        }
    }
}
```

Revenind la evenimente, descriem pe scurt un exemplu teoretic de declarare şi tratare a unui eveniment. În clasa Vector se consideră că interschimbarea valorilor a două componente ale unui vector e un eveniment de interes pentru alte obiecte sau clase ale aplicaţiei. Se defineşte un tip delegat TD (să zicem) cu nişte parametri de interes(de exemplu indicii componentelor interschimbate) şi un eveniment care are ca asociat un delegat E (de tip TD). Orice obiect x din clasa Vector are un membru E (iniţial **null**). O clasă C interesată să fie înştiinţată când se face vreo interschimbare într-un vector pentru a genera o animaţie (de exemplu), va implementa o metodă M ce realizează animaţia şi va adăuga pe M (prin intermediul unui delegat) la **x.E+=new [tip_delegat](M).** Cumulând mai multe astfel de referinţe, x.E ajunge un fel de listă de metode (**handlere**). În clasa Vector, în metoda sort, la interschimbarea valorilor a două componente se invocă delegatul E. Invocarea lui E realizează de fapt activarea tuturor metodelor adăugate la E.

Indexatori

Sunt cazuri în care are sens să tratăm o clasă ca un array. Cei care au studiat C++ vor observa că este o generalizare a supraîncărcării operatorului [ ] din respectivul limbaj.

Sintaxa:

```
[atribut][modificatorIndexator] declaratorDeIndexator
{
    declaratiiDeAccesor
}
```

unde:

modificatorIndexator – poate fi **new, public, protected, internal, private, virtual, sealed, override, abstract, extern**.

declaratorDeIndexator – are forma:

```
tipReturnat this [listaParametrilorFormali]
```

unde:

listaParametrilorFormali – trebuie să conţină cel puţin un parametru, parametru care nu trebuie să fie de tipul **ref** sau **out**.

declaratiiDeAccesor – asemănătoare cu cele de la proprietăţi, trebuie să conţină accesorul **get** sau accesorul **set.**

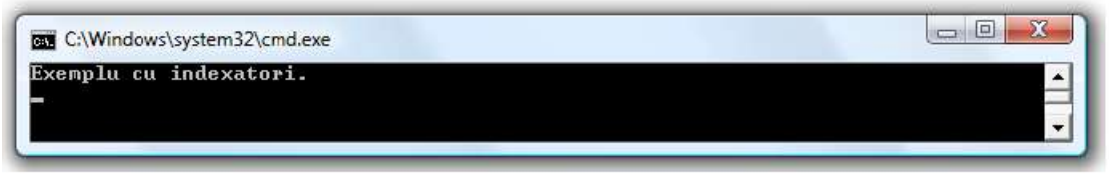
Observaţie: Indexatorii şi proprietăţile sunt asemănătoare în ceea ce priveşte utilizarea accesorilor **get** şi **set**. Un indexator poate fi privit ca o proprietate cu mai multe valori. Pe când o proprietate poate fi declarată statică, acest lucru este interzis în cazul indexatorilor.

Când folosim un indexator, sintaxa este asemănătoare cu cea de la vectori. Totuşi există deosebiri:

- indexatorii pot folosi indici nenumerici, pe când un vector trebuie să aibă indicii de tip întreg
- indexatorii pot fi supradefiniţi, la fel ca metodele, pe când vectorii nu
- indexatorii nu pot fi folosiţi ca parametrii **ref** sau **out**, pe când vectorii da

Exemplul 76:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Exemplul_76
{
    class ClasaMea
    {
        private string[] data = new string[6];
        public string this[int index]
        {
            get
            {
                return data[index];
            }
            set
            {
                data[index] = value;
            }
        }
    }
    class Rezultat
    {
        public static void Main()
        {
            ClasaMea v = new ClasaMea();
            v[0] = "Exemplu";
            v[1] = "cu";
            v[2] = "indexatori";
            Console.WriteLine("{0} {1} {2}.", v[0], v[1], v[2]);
            Console.ReadLine();
        }
    }
}
```



Operatori

Definiţie: operatorul este un membru care defineşte semnificaţia unei expresii operator care poate fi aplicată unei instanţe a unei clase. Pentru cei care cunosc C++, operatorul corespunde supraîncărcării din respectivul limbaj.

Sintaxa:

```
[atribut] modificatorOperator declaratieDeOprator corpOperator
```

Observaţia 1: Operatorii trebuiesc declaraţi publici sau statici.

Observaţia 2: Parametrii operatorilor trebuie să fie de tip valoare. Nu se admit parametri de tip **ref** sau **out**.

Observaţia 3: În antetul unui operator nu poate apărea, de mai multe ori, acelaşi modificator.

Se pot declara operatori: unari, binari şi de conversie.

Operatori unari

Supraîncărcarea operatorilor unari are următoarea sintaxă:

```
tip operatorUnarSupraîncărcabil (tip identificator) corp
```

Operatorii unari supraîncărcabili sunt: **+ - ! ~ ++ – true false**.

Reguli pentru supraîncărcarea operatorilor unari:

Fie T clasa care conţine definiţia operatorului

1. Un operator + - ! ˜ poate returna orice tip şi preia un singur parametru de tip T

2. Un operator ++ sau –- trebuie să returneze un rezultat de tip T şi preia un singur parametru de tip T

3. Un operator unar **true** sau **false** returnează **bool** şi trebuie să preia un singur parametru de tip T. Operatorii **true** şi **false** trebuie să fie ambii definiţi pentru a prevenii o eroare de compilare.

Exemplul 77:

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

namespace Exemplul_77
{
    class Complex
    {
        private int x;
        private int y;
        public Complex()
        {
        }
        public Complex(int i, int j)
        {
            x = i;
            y = j;
        }

        public void Afis()
        {
            Console.WriteLine("{0} {1}i", x, y);
        }

        public static Complex operator -(Complex c)
        {
            Complex temp = new Complex();
            temp.x = -c.x;
            temp.y = -c.y;
            return temp;
        }
    }
```

```
    class Program
    {
        public static void Main()
        {
            Complex c1 = new Complex(10, 13);
            c1.Afis();
            Complex c2 = new Complex();
            c2.Afis();
            c2 = -c1;
            c2.Afis();
            Console.ReadLine();
        }
    }

}
```

Operatori binari

Supraîncărcarea operatorilor binari are următoarea sintaxă:

```
tip operator operatorBinarSupraîncărcabil (tip identificator,
                                           tip identificator) corp
```

Operatorii binari supraîncărcabili sunt: + - * / □ & | ^ << >> == != > < >= <=

Reguli pentru supraîncărcarea operatorilor binari:

1. Cel puţin unul din cei doi parametri trebuie să fie de tipul clasei în care respectivul operator a fost declarat

2. Operatorii de shift-are trebuie să aibă primul parametru de tipul clasei în care se declară, iar al doilea parametru de tip **int**

3. Un operator binar poate returna orice tip

4. Următorii operatori trebuie să se declare în pereche:

a. operatorii == şi !=

b. operatorii > şi <

c. operatorii >= şi <=

Exemplul 78:

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

namespace ExempluOperatori
{
    class Complex
    {
        private int x;
        private int y;
        public Complex()
        {
        }
        public Complex(int i, int j)
        {
            x = i;
            y = j;
        }
        public void Afis()
        {
            Console.WriteLine("{0} {1}", x, y);
         }
```

```
        public static Complex operator +(Complex c1, Complex c2)
        {
            Complex temp = new Complex();
            temp.x = c1.x + c2.x;
            temp.y = c1.y + c2.y;
            return temp;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Complex c1 = new Complex(1, 2);
            Console.Write("c1: ");
            c1.Afis();
            Complex c2 = new Complex(3, 4);
            Console.Write("c2: ");
            c2.Afis();
            Complex c3 = new Complex();
            c3 = c1 + c2;
            Console.WriteLine("\nc3 = c1 + c2\n");
            Console.Write("c3: ");
            c3.Afis();
            Console.ReadLine();
        }
    }
}
```

Operatori de conversie

Operatorul de conversie introduce o conversie definită de utilizator. Această conversie nu va suprascrie conversiile predefinite. Operatorii de conversie pot fi:

- **impliciţi** – se efectuează de la un tip ”mai mic” la un tip „mai mare” şi reuşesc întotdeauna, nepierzându-se date
- **expliciţi** – se efectuează prin intermediul expresiilor de conversie, putându-se pierde date

Sintaxa:

```
implicit operator tip(tip parametru) corp
explicit operator tip(tip parametru) corp
```

Un operator de acest tip va face conversia de la tipul sursa (S) (tipul parametrului din antet) în tipul destinaţie (D) (tipul returnat).

O clasă poate să declare un operator de conversie de la un tip S la un tip D dacă:

1. S şi D au tipuri diferite
2. S sau D este clasa în care se face definirea
3. S şi D nu sunt object sau tip interfaţă
4. S şi D nu sunt baze una pentru cealaltă

Exemplu 79: conversii dintr-un tip de bază într-o clasă şi un tip clasă într-un tip de bază folosind conversia operator:

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

namespace Exemplul_79
{
    class MyDigit
    {
        private int x;
        public MyDigit()
        {
        }
        public MyDigit(int i)
        {
            x = i;
        }
        public void ShowDigit()
        {
            Console.WriteLine("{0}", x);
        }
        public static implicit operator int(MyDigit md)
        {
            return md.x;
        }
        public static explicit operator MyDigit(int val)
        {
            return new MyDigit(val);
        }
    }
```

```
    class Program
    {
        public static void Main(string[] args)
        {
            MyDigit md1 = new MyDigit(10);
            int x = md1;              //Implicit
            Console.WriteLine(x);
            int y = 25;
            MyDigit md2 = (MyDigit)y; //Explicit
            md2.ShowDigit();
            Console.ReadLine();
        }
    }
}
```

Exemplul 80: Conversia dintr-un tip clasă în altul folosind conversia operator:

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

namespace OperatoriImplicitiExpliciti
{
    class Clasa1
    {
        public int x;
        public Clasa1(int a)
        {
            x = a;
        }
        public void Afis1()
        {
            Console.WriteLine(x);
        }
        public static explicit operator Clasa2(Clasa1 mc1)
        {
            Clasa2 mc2 = new Clasa2(mc1.x * 10, mc1.x * 20);
            return mc2;
        }
    }
    class Clasa2
    {
        public float x, y;
        public Clasa2(float a, float b)
        {
            x = a;
            y = b;
        }
        public void Afis2()
        {
            Console.WriteLine(x);
            Console.WriteLine(y);
        }
    }
    class Program
    {
        public static void Main(string[] args)
        {
            Clasa1 mc1 = new Clasa1(100);
            mc1.Afis1();
            Clasa2 mc2 = (Clasa2)mc1;
            mc2.Afis2();
            Console.ReadLine();
        }
    }
}
```

Clase şi funcţii generice

Definiţie: genericele sunt şabloane (templates) sau modele care ajută la reutilizarea codului. Ele descriu clase şi metode care pot lucra într-o manieră uniformă cu tipuri de valori diferite.

Ele permit definirea de funcţionalităţi şi metode care se adaptează la tipurile parametrilor pe care îi primesc, ceea ce permite construirea unui şablon.

Singura diferenţă faţă de declararea în mod obişnuit a unei clase, este prezenţa caracterelor < şi >, care permit definirea tipului pe care stiva îl va avea, ca şi cum ar fi un parametru al clasei.

La instanţierea clasei trebuie să declarăm tipul datelor utilizate.

Tipurile generice (parametrizate) permit construirea de clase, structuri, interfeţe, delegaţi sau metode care sunt parametrizate printr-un tip pe care îl pot stoca sau manipula.

Exemplul 81: Să considerăm clasa Stiva care permite stocarea de elemente. Această clasă are două metode **Push()** care permite introducerea de elemente şi **Pop()** care permite extragerea de elemente din stivă.

```
public class Stiva<TipElement>  //clasa generica
  {
  private TipElement[] element;

  public void Push(TipElement data)
    {
    // code corespunzator introducerii de elemente
    }

  public TipElement Pop()
    {
    // code corespunzator extragerii de elemente
    }
  }

Stiva<char> StivaMea = new Stiva<char>();
StivaMea.Push("a");
char x = StivaMea.Pop();
```

Exemplul 82: tipurile parametrizate pot fi aplicate claselor şi interfeţelor

```
interface IGeneric1<T>
{ }
class ClassGeneric1<UnTip, Altul>
{ }
class ClassInt1 : ClassGeneric1<int, int>
{ }
class ClassInt2<T> : ClassGeneric1<int, T>
{ }
class ClassInt3<T, U> : ClassGeneric1<int, U>
{ }
```

Exemplul 83: tipurile parametrizate se pot aplica metodelor

```
class clA
{
    public void methode1<T>()
    {
    }
    public T[] methode2<T>()
    {
        return new T[10];
    }
}
```

Exemplul 84: Dorim să implementăm o clasă Stiva care să permită adăugarea şi extragerea de elemente. Pentru a simplifica problema, vom considera că stiva nu poate conţine decât un anumit număr de elemente, ceea ce ne va permite să utilizăm tablouri în C#.

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

namespace Exemplul_84
{
    class Stiva
    {
        private object[] m_ItemsArray;
        private int m_Index = 0;
        public const int MAX_SIZE = 100;
        public Stiva() { m_ItemsArray = new object[MAX_SIZE]; }
        public Object Pop()
        {
            if (m_Index == 0)
                throw new InvalidOperationException("Nu putem extrage un
element dintr-o stiva vida.");
            return m_ItemsArray[--m_Index];
             }
```

```
        public void Push(Object item)
        {
            if (m_Index == MAX_SIZE)
                throw new StackOverflowException("Nu se poate adauga un
elemet: stiva este plina.");
            m_ItemsArray[m_Index++] = item;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Stiva stiva = new Stiva();
            stiva.Push(1234);
            int numar = (int)stiva.Pop();
        }
    }
}
```

Implementarea suferă de câteva probleme:

- elementele clasei Stiva trebuie să fie convertite explicit

- atunci când se foloseşte clasa Stiva cu elemente de tip valoare, se realizează implicit o operaţie de boxing cu inserarea unui element şi o operaţie de tip unboxing cu recuperarea unui element

- dorim să introducem în stivă elemente de tipuri diferite în aceea_i instanţă a clasei Stiva. Acest lucru va duce la probleme de convertire care vor fi descoperite la execuţie

Deoarece problema conversiei nu este detectată la compilare, va produce o excepţie la execuţie. Din acest motiv spunem: codul nu este type-safe.

Pentru a rezolva aceste neajunsuri s-ar putea implementa un cod pentru stive cu elemente de tip int, alt cod pentru elemente de tip sir de caractere. Acest lucru duce la dublarea unor porţiuni din cod. Acest lucru se va rezolva cu ajutorul tipurilor generice.

C# ne permite rezolvarea unor astfel de probleme introducând tipurile generice. Concret putem implementa o listă de elemente de tip T, lăsând libertatea utilizatorului să specifice tipul T la instanţierea clasei.

Exemplul 85:

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

namespace Exemplul_85
{
    class Stiva<T>
    {
        private T[] m_ItemsArray;
        private int m_Index = 0;
        public const int MAX_SIZE = 100;
        public Stiva() { m_ItemsArray = new T[MAX_SIZE]; }
        public T Pop()
        {
            if (m_Index == 0)
                throw new InvalidOperationException("Nu putem extrage un
element dintr-o stiva vida.");
            return m_ItemsArray[--m_Index];
        }
        public void Push(Object item)
        {
            if (m_Index == MAX_SIZE)
                throw new StackOverflowException("Nu se poate adauga un
elemet: stiva este plina.");
            m_ItemsArray[m_Index++] = item;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Stiva<int> stiva = new Stiva<T>();
            stiva.Push(1234);
            int numar = stiva.Pop(); //nu mai este necesar cast
            Stiva<string> sstiva = new Stiva<string>();
            sstiva.Push("4321");
            string sNumar = sstiva.Pop();
        }
    }
}
```