

Laborator 14

Polimorfism

Introducere

În Capitolul 3 defineam noțiunea de **polimorfism**, folosind o extensie a sensului etimologic:

un obiect polimorfic este cel capabil să ia diferite forme, să se afle în diferite stări, să aibă comportamente diferite. **Polimorfismul obiectual**, care trebuie să fie abstract, se manifestă în lucrul cu obiecte din clase aparținând unei ierarhii de clase, unde, prin redefinirea unor date sau metode, se obțin membri diferiți având însă același nume.

Pentru a permite acest mecanism, metodele care necesită o decizie contextuală (în momentul apelului), se declară ca metode virtuale (cu modificatorul **virtual**). În mod curent, în C# modificatorului **virtual** al funcției din clasa de bază, îi corespunde un specificator **override** al funcției din clasa derivată ce redefinește funcția din clasa de bază.

O metodă ne-virtuală nu este polimorfică și, indiferent de clasa căreia îi aparține obiectul, va fi invocată metoda din clasa de bază.

Limbajul C# admite trei tipuri de polimorfism:

- polimorfism parametric
- polimorfism ad-hoc
- polimorfism de moștenire

Polimorfismul parametric

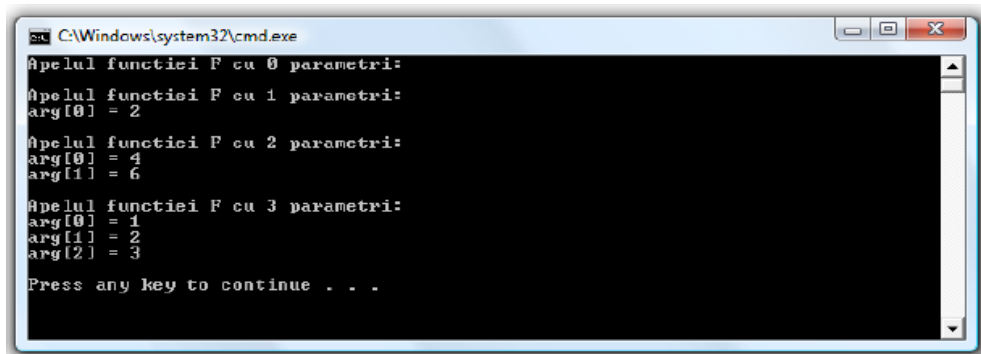
Această formă de polimorfism este preluată de la limbajele neobiectuale: Pascal, C. Prin această formă de polimorfism, o funcție va prelucra orice număr de parametri. Pentru aceasta se va folosi un parametru de tip **params**.

Exemplul 93: Să considerăm o funcție **F** cu un parametru formal, de tip vector, declarat folosind modificatorul **params**. Acest lucru va permite folosirea mai multor parametri actuali, la apelul funcției, prin intermediul acelui singur parametru formal.

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

namespace Exemplul_93
{
    class Program
    {
        static void F(params int[] arg)
        {
            Console.WriteLine("Apelul functiei F cu {0} parametri:",
                              arg.Length);

            for (int i = 0; i < arg.Length; i++)
            {
                Console.WriteLine("arg[{0}] = {1}", i, arg[i]);
            }
            Console.WriteLine("");
        }
        static void Main(string[] args)
        {
            F();
            F(2);
            F(4, 6);
            F(new int[] { 1, 2, 3 });
        }
    }
}
```



```
C:\Windows\system32\cmd.exe
Apelul functiei F cu 0 parametri:
Apelul functiei F cu 1 parametri:
arg[0] = 2
Apelul functiei F cu 2 parametri:
arg[0] = 4
arg[1] = 6
Apelul functiei F cu 3 parametri:
arg[0] = 1
arg[1] = 2
arg[2] = 3
Press any key to continue . . .
```

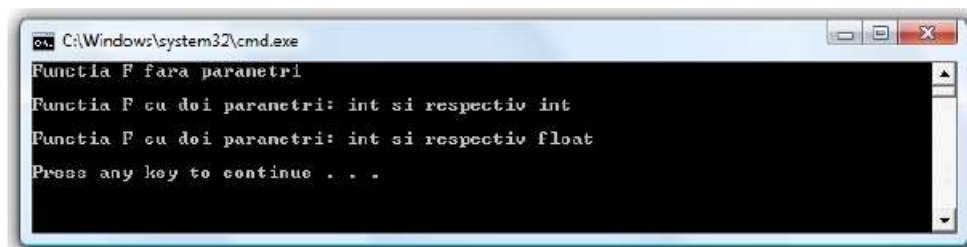
Polimorfismul ad-hoc

Acest tip de polimorfism se mai numește și supraîncărcarea metodelor. Prin acest mecanism se pot defini în cadrul unei clase mai multe metode, toate având același nume, dar cu tipul și numărul de parametri diferiți. La compilare, în funcție de parametri folosiți la apel, se va apela o funcție sau alta.

Exemplul 94:

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

namespace PolimorfismAdHoc
{
    class Program
    {
        static void F()
        {
            Console.WriteLine("Functia F fara parametri\n");
        }
        static void F(int a, int b)
        {
            Console.WriteLine("Functia F cu doi parametri: int si respectiv
int\n");
        }
        static void F(int a, double b)
        {
            Console.WriteLine("Functia F cu doi parametri: int si respectiv
float\n");
        }
        static void Main(string[] args)
        {
            F();
            F(2, 3);
            F(4, 6.3);
        }
    }
}
```



```
C:\Windows\system32\cmd.exe
Functia F fara parametri
Functia F cu doi parametri: int si respectiv int
Functia F cu doi parametri: int si respectiv float
Press any key to continue . . .
```

Polimorfismul de moștenire

În cazul acestui tip de moștenire vom discuta într-o ierarhie de clase. În acest caz ne punem problema apelării metodelor, având aceeași listă de parametri formali, metode ce fac parte din clase diferite.

Exemplul 95:

```

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

namespace Exemplul_95
{
    class Baza
    {
        public void Afis()
        {
            Console.WriteLine("Apelul functiei Afis din clasa de baza\n");
        }
    }
    class Derivata : Baza
    {
        public void Afis()
        {
            Console.WriteLine("Apelul functiei Afis din clasa derivata\n");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Derivata obiect2 = new Derivata();
            Baza obiect1 = obiect2;
            obiect1.Afis(); // (1)
            obiect2.Afis(); // (2)
        }
    }
}

```

```

C:\Windows\system32\cmd.exe
Apelul functiei Afis din clasa de baza
Apelul functiei Afis din clasa derivata
Press any key to continue . . . _

```

Să discutăm despre prima linie afișată (cea de-a doua este evidentă). Apelul lui **Afis()** se rezolvă în momentul compilării pe baza tipului declarat al obiectelor. Deci linia (1) din program va duce la apelul lui **Afis()** din clasa **Baza**, chiar dacă obiect1 a fost instanțiat pe baza unui obiect din clasa **Derivata**.

Modificatorii virtual și override

În cazul în care se dorește ca apelul metodelor să se facă la rulare și nu la compilare vom reconsidera exemplul anterior în care funcția **Afis()** din clasa de bază o declarăm virtuală, iar funcția **Afis()** din clasa derivată o considerăm ca suprascriere a lui **Afis()** din clasa de bază:

Exemplul 96:

```

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

namespace Exemplul_96
{
    class Baza
    {
        public virtual void Afis()
        {
            Console.WriteLine("Apelul functiei Afis din clasa de baza\n");
        }
    }
    class Derivata : Baza
    {
        public override void Afis()
        {
            Console.WriteLine("Apelul functiei Afis din clasa derivata\n");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Derivata obiect2 = new Derivata();
            Baza obiect1 = obiect2;
            obiect1.Afis(); // (1)
            obiect2.Afis(); // (2)
        }
    }
}

```



Modificatorul new

În cazul în care se dorește ca o metodă dintr-o clasă derivată să aibă aceeași semnătură cu o metodă dintr-o clasă de bază, dar să nu fie considerată o suprascriere a ei, vom folosi modificatorul **new**.

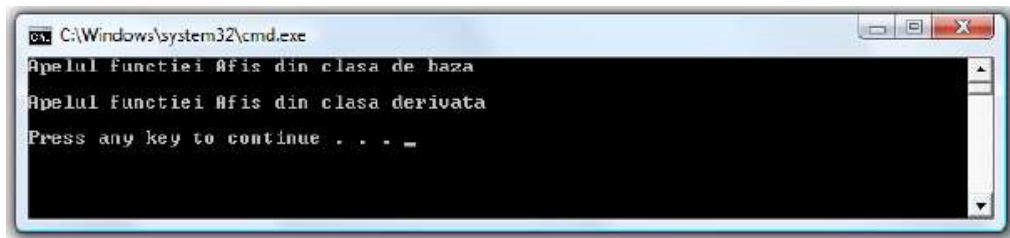
Exemplul 97:

```

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

namespace PolimorfismDeMostenire
{
    class Baza
    {
        public virtual void Afis()
        {
            Console.WriteLine("Apelul functiei Afis din clasa de baza\n");
        }
    }
    class Derivata : Baza
    {
        public new void Afis() // !!! new
        {
            Console.WriteLine("Apelul functiei Afis din clasa derivata\n");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Derivata obiect2 = new Derivata();
            Baza obiect1 = obiect2;
            obiect1.Afis(); // (1)
            obiect2.Afis(); // (2)
        }
    }
}

```



Metoda sealed

O metodă având tipul **override** poate fi declarată **sealed**. În acest fel ea nu mai poate fi suprascrisă într-o clasă derivată

Exemplul 98:

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

namespace Exemplul_98
{
    class Baza
    {
        public virtual void Afis()
        {
            Console.WriteLine("Apelul functiei Afis din clasa de baza\n");
        }
    }
    class Derivata : Baza
    {
        sealed override public void Afis()
        {
            Console.WriteLine("Apelul functiei Afis din clasa derivata\n");
        }
    }
    class Derivata2 : Derivata
    {
        override public void Afis()          //!!! EROARE !!!
        {
            Console.WriteLine("Apelul functiei Afis din clasa Derivata2\n");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Derivata obiect2 = new Derivata();
            Baza obiect1 = new Derivata();
            Derivata2 obiect3 = new Derivata2();
            obiect1.Afis();                  //(1)
            obiect2.Afis();                  //(2)
            obiect3.Afis();
        }
    }
}
```

Va genera eroare, deoarece modificatorul **sealed** al metodei **Afis()**, din clasa **Derivata**, va împiedică suprascrierea acestei metode în clasa **Derivata2**.