Laborator 12

Derivarea claselor (moştenire)

Principiile moştenirii

Prin utilizarea moştenirii se poate defini o clasă generală care defineşte trăsături comune la un ansamblu de obiecte. Această clasă poate fi moştenită de către alte clase specifice, fiecare dintre acestea adăugând elemente care-i sunt unice ei.

O clasă care este moştenită se numeşte **clasă de bază** sau **superclasă**, iar o clasă care o moşteneşte pe aceasta se numeşte **clasă derivată**, sau **subclasă**, sau **clasă descendentă**.

- Pe baza a ceea ce am amintit, putem spune că o clasă derivată este o versiune specializată sau extinsă a clasei de bază.
- Clasa derivată moşteneşte toate elementele clasei de bază şi-şi adaugă altele proprii.
- Clasa derivată nu poate să şteargă nici un membru al clasei de bază.

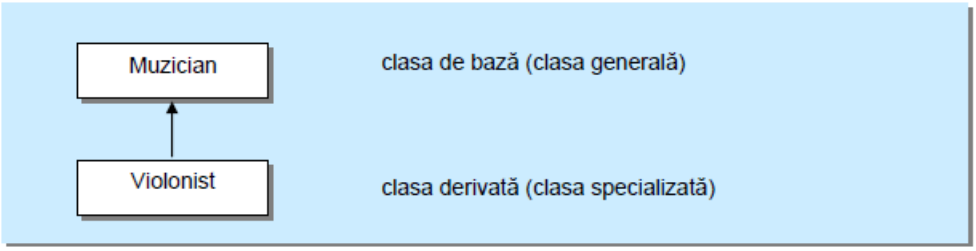
Definirea unei clase derivate se face folosind sintaxa:

```
class ClasaDerivata : ClasaDeBaza
{
    ...
}
```

O clasă derivată poate la rândul ei să fie clasă de bază pentru o altă clasă. În acest fel se poate defini noţiunea de **ierarhie de clase**.

Limbajul C#, spre deosebire de C++, admite doar **moştenirea simplă**, în sensul că derivarea se admite doar dintr-o clasă de bază, fiind permisă doar derivarea publică

În contextul mecanismelor de moştenire trebuie amintiţi modificatorii **abstract** şi **sealed** aplicaţi unei clase, modificatori ce obligă la şi respectiv se opun procesului de derivare. Astfel, o clasă abstractă trebuie obligatoriu derivată, deoarece direct din ea nu se pot obţine obiecte prin operaţia de instanţiere, în timp ce o clasă sigilată (**sealed**) nu mai poate fi derivată (e un fel de terminal în ierarhia claselor). O metodă abstractă este o metodă pentru care nu este definită o implementare, aceasta urmând a fi realizată în clasele derivate din clasa curentă. O metodă sigilată nu mai poate fi redefinită în clasele derivate din clasa curentă.



Exemplul 86:

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

namespace Exemplul_86
{
    class Muzician
    {
        public void Canta(string nume)
        {
            Console.WriteLine("{0} canta", nume);
        }
    }
    class Violonist : Muzician
    {
        public void CantaLaVioara(string nume)
        {
            Console.WriteLine("{0} canta la vioara", nume);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Muzician m = new Muzician();
            m.Canta("Ilie");
            Violonist n = new Violonist();
            n.Canta("Andrei");
            n.CantaLaVioara("Andrei");
            Console.ReadLine();
        }
    }
}
```

Accesibilitatea membrilor moşteniţi

Deseori, în procesul derivării, avem nevoie de acces la membrii moşteniţi ai clasei de bază.

Pentru aceasta se va folosi o expresie de tip **base access**.

De exemplu, dacă **MembruB** este un membru al clasei de bază, pentru a-l folosi într-o clasa derivată vom folosi, în aceasta, o expresie de forma:

```
base.MembruB
```

Exemplul 84: apelul din clasa derivată a unui membru al clasei de bază

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

namespace Exemplul_87
{
    class Program
    {
        class ClasaDeBaza
        {
            public string sir = "Sir din clasa de baza";
        }

        class ClasaDerivata : ClasaDeBaza
        {
            public string sir = "Sir din clasa derivata";
            public void afis()
            {
                Console.WriteLine("{0}", sir);
                Console.WriteLine("{0}", base.sir);
            }
        }

        static void Main(string[] args)
        {
            ClasaDerivata cd = new ClasaDerivata();
            cd.afis();
            Console.ReadLine();
        }
    }
}
```

Utilizarea cuvântului cheie protected

Cuvântul cheie **protected** permite restrângerea accesului unui membru al clasei de bază doar la clasele sale derivate. Membrii protejaţi moşteniţi devin în mod automat protejaţi.

Apelul constructorilor clasei de bază

Exemplul 88:

```
class ClasaDeBaza
{
    protected string var;
    public ClasaDeBaza(string var)    //constructor
    {
        this.var = var;
    }
}

clasa Derivata : ClasaDeBaza
{
    public ClasaDeBaza(string var) : base(var)
    {
        ...
    }
}
```

Metode

Prin mecanismul de moştenire avem posibilitatea reutilizării codului şi redefinirii (prin polimorfism) a metodelor.

Virtual şi override

O clasă declarată virtuală implică faptul că o metodă implementată în ea poate fi redefinită în clasele derivate.

Doar metodele virtuale ne statice şi/sau private pot fi redefinite într-o clasă derivată. Aceste metode trebuie să aibă aceeaşi signatură (nume, modificator de acces, tip returnat şi parametri).

Pentru declararea unei metode ca fiind virtuală se foloseşte cuvântul cheie **virtual**. În clasele derivate se va folosi cuvântul cheie **override** pentru redefinirea metodei virtuale din clasa de bază.

Exemplul 89:

```
class ClasaDeBaza
{
    public virtual void Metoda()
     {
          ...
     }
}
class Derivata : ClasaDeBaza
{
    public override void Metoda()
     {
          ...
     }
}
```

new

Există cazuri în care în loc să redefinim o metodă avem nevoie să specificăm că metoda clasei derivate este o implementare nouă a respectivei metode. Pentru aceasta vom folosi new cu semnificaţia că metoda are aceeaşi signatură cu a celei din clasa de bază, dar dorim să mascăm definirea ei în clasa de bază.

Exemplul 90:

```
class ClasaDeBaza
{
    public virtual void Metoda()
     {
          ...
     }
}
class Derivata : ClasaDeBaza
{
    public new void Metoda()
     {
          ...
     }
}
```

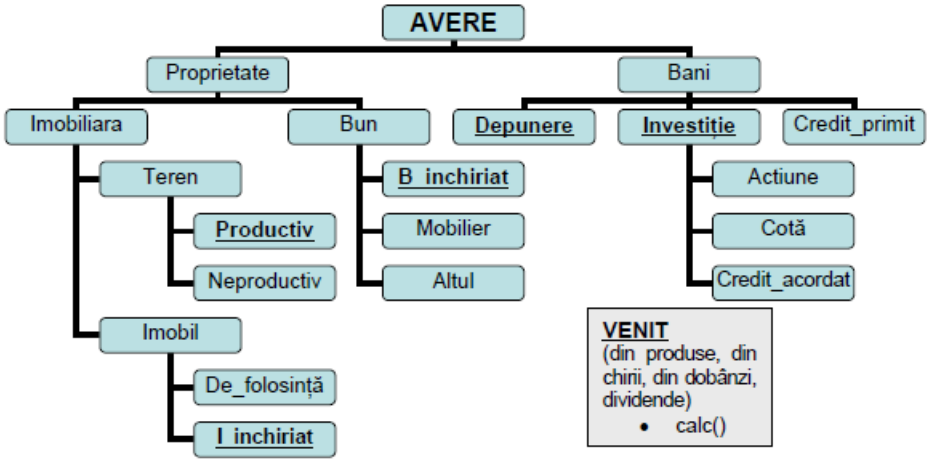
Interfeţe

Interfeţele sunt foarte importante în programarea orientată pe obiecte, deoarece permit utilizarea polimorfismului într-un sens mai extins.

Definiţie: O **interfaţă** este o componentă a aplicaţiei, asemănătoare unei clase, care declară prin membrii săi (metode, proprietăţi, evenimente şi indexatori) un „comportament” unitar aplicabil mai multor clase, comportament care nu se poate defini prin ierarhia de clase a aplicaţiei.

De exemplu, dacă vom considera arborele din figura următoare, în care AVERE este o clasă abstractă, iar derivarea claselor a fost concepută urmărind proprietăţile comune ale componentelor unei averi, atunci o clasă

VENIT nu este posibilă, deoarece ea ar moşteni de la toate clasele evidenţiate, iar moştenirea multiplă nu este admisă în C#.



Pentru metodele din cadrul unei interfeţe nu se dă nici o implementare, ci sunt pur şi simplu specificate, implementarea lor fiind furnizată de unele dintre clasele aplicaţiei. Acele clase care „aderă" la o interfaţă spunem că „implementează" interfaţa respectivă. Nu există instanţiere în cazul interfeţelor, dar se admit derivări, inclusiv moşteniri multiple.

În exemplul nostru, se poate defini o interfaţă **VENIT** care să conţină antetul unei metode calc (să zicem) pentru calculul venitului obţinut, fiecare dintre clasele care implementează interfaţa **VENIT** fiind obligată să furnizeze o implementare (după o formulă de calcul specifică) pentru metoda **calc** din interfaţă. Orice clasă care doreşte să adere la interfaţă trebuie să implementeze toate metodele din interfaţă. Toate clasele care moştenesc dintr-o clasă care implementează o interfaţă moştenesc, evident, metodele respective, dar le pot şi redefini (de exemplu, clasa **Credit_acordat** redefineşte metoda **calc** din clasa **Investiţie**, deoarece formula de calcul implementată acolo nu i se „potriveşte" şi ei. Dacă în sens polimorfic spunem că Investiţie este şi de tip **Bani** şi de tip **Avere**, tot aşa putem spune că o clasă care implementează interfaţa **VENIT** şi clasele derivate din ea sunt şi de tip **VENIT**).

De exemplu, dacă presupunem că toate clasele subliniate implementează interfaţa **VENIT**, atunci pentru o avere cu acţiuni la două firme, un imobil închiriat şi o depunere la bancă, putem determina venitul total:

Exemplul 91:

```
Actiune act1 = new Actiune();
Actiune act2 = new Actiune();
I_inchiriat casa = new I_inchiriat();
Depunere dep=new Depunere();
Venit[] venituri = new Venit()[4];
venituri[0] = act1; venituri[1] = act2;
venituri[2] = casa; venituri[3] = dep;
...
int t=0;
for(i=0;i<4;i++)
   t+=v[i].calc();
```