Indexari, proprietati, structuri, enumerari

Indexari

O indexare permite unui *obiect sa fie indexat la fel ca un tablou*. In principal, indexarile se utilizeaza la crearea *de tablouri specializate care sunt obiectul unor restrictii*. Indexarile pot avea una sau mai multe dimensiuni.

In cazul indexarilor unidimensionale, forma generala este:

```
tip-element this[int index]
{
get {  //se intoarce valoarea precizata de index }
set  {  //se modifica valoarea precizata de index}
}
```

unde: - tip-element reprezinta tipul de baza al indexarii. Fiecare element al indexarii este de tipul tip-element;

-paramentrul index primeste valoarea indexului elementului care va fi accesat; Acest parametru poate fi si de alt tip decat int, insa intrucata indexarile se utilizeaza pentru a implementa un mecanism similar tablourilor, de regula se recurge la un tip intreg.

-in corpul unei indexari sunt definiti doi accesori, denumiti get si set. Un accesor este similar cu o metoda, cu diferenta ca nu contine parametrii si nu specifica tipul rezultatului. La utilizarea indexarii, accesorii sunt automat apelati, ambii accesori primind index ca parametru: Daca indexarea apare in partea stanga a unei instructiuni de atribuire, se apeleaza accesorul set. Acesta primeste, in plus fata de parametrul index, o valoare numita value, care este atribuita elementului indexarii cu indexul precizat de parametrul index. Daca indexarea apare in partea dreapta a unei expresii atunci se apeleaza accesorul get, care intoarce valoarea asociata cu index.

Unul dintre avantajele utilizarii unei indexari este acela ca se *poate controla cu exactitate modul de acces la un tablou*, eliminand accesarile incorecte (utilizarea unor indici negativi sau indici mai mari decat dimensiunea tabloului).

Exemplul 1. Utilizarea indexarilor

```
using System;
class Tablou
{
    int[] a;
    public int dimensiune;
    public bool coderr;
    public Tablou(int lungime)
    {
        a = new int[lungime];
        dimensiune= lungime;
    }
    public int this[int index]
    {
        get
        {
            if (ok(index))
            {
                coderr = false;
                return a[index];
            }
            else
            { coderr = true;  return -1;  }
        }
        set
        {
            if (ok(index))
            {
                a[index] = value;
                coderr = false;
            }
            else {  coderr = true;}
        }
    }
    private bool ok(int index)
    {
        if((index>=0) & (index<dimensiune))
            return true;
        else
            return false;
    }
}
class TablouDemo
{
    public static void Main()
    {
        int x;
        Tablou tb = new Tablou(5);
        for (int i = 0; i < tb.dimensiune*2; i++)
            tb[i] = 10 * i;
        for (int i = 0; i < tb.dimensiune*2; i++)
        {
            x = tb[i];
            if (tb.coderr)
                Console.Write("\ntb[{0}] Depaseste
      marginile", i);
            else
                Console.Write("{0} \t",x);
        }
    }
}
```

Rezultat:

0 10 20 30 40

tb[5] Depaseste marginile
tb[6] Depaseste marginile
tb[7] Depaseste marginile
tb[8] Depaseste marginile
tb[9] Depaseste marginile

Exemplul 2. Utilizarea indexarilor

Nu este neaparat ca o indexare sa lucreze cu un tablou. Indexarile ofera functionalitate care se apropie de cea a tablourilor. Spre exemplu, programul alaturat contine o indexare care se comporta ca un tablou accesibil la citire si care contine puterile lui 3.

De asemenea, nu este neaparat ca prametrul index sa fie de tip int. In acest sens, realizati in programul alaturat urmatoarele modificari pentru a obtinte puterile lui 3 la o putere de tip double: schimbati in tot corpul programului int cu double, stergeti castul (int) din accesorul get si initializati variabila i din instructiunea for cu o valoare rationala, spre exemplu 0.1.

In cazul tablourilor bidimensionale, o indexare are forma:

```
tip-element this[int index1, int index2]
{
get {  //se intoarce valoarea precizata de index1 si index2 }
set  {  //se modifica valoarea precizata de index1 si index2}
}
```

Exercitiu: Modificati exemplul 1 in asa fel incat sa implementati o indexare bidimensionala.

```
using System;
class Putere3
{
   public int this[int index]
   {
      get
      {
         if (index >= 0)
            return (int)Math.Pow(3, index);
         else
            return -1;
      }
   }
}
class PutereDemo
{
   public static void Main()
   {
      Putere3 p = new Putere3();
      for (int i = 0; i < 10; i++)
      {
         Console.WriteLine("3^{0}={1}", i, p[i]);
      }
   }
}
```

Rezultat:

```
3^0=1
3^1=3
3^2=9
3^3=27
3^4=81
3^5=243
3^6=729
3^7=2187
3^8=6561
3^9=19683
```

Proprietati

Unul din beneficiile programelor orientate obiect il reprezinta capacitatea de a *controla reprezentarea interna si accesul la date*. In acest sens, C# furnizeaza un concept (implementeaza o categorie de membrii ai claselor) numit *proprietati*. O *proprietate* gestioneaza accesul la un camp (o variabila privata) prin intermediul accesorilor get si set.

Forma generala a unei proprietati este:

```
 tip nume
{
get {  //codul accesorului get }
set  {  //codul accesorului set}
}
```

unde tip precizeaza tipul proprietatii, ca de exemplu int, iar nume este numele proprietatii. La fel ca in cazul indexarilor, accesorul set primeste un parametru numit value, care contine valoarea atribuita propretatii.

Observatii:-este esential sa intelegem ca proprietatile nu definesc locatii de memorie. Gestionand accesul la un camp, acesta din urma trebuie specificat separat de proprietate. Proprietatea nu contine campul respectiv;

-se pot defini proprietati accesibile la citire si scriere (contin ambii accesori get si set) sau accesibile doar la citire (contin accesorul get) sau la scriere (contin accesorul set);

-deoarece proprietatile nu definesc o zona de memorie, acestea nu pot fi transmise ca parametri ref sau out unei metode;

-proprietatile nu pot fi supraincarcate;

-o proprietate nu trebuie sa altereze starea variabilei asociate la apelul accesorului get.

In exemplul urmator se considera utilizeaza doua proprietati pentru a accesa la citire si scriere doua campuri private my_x si my_y.

Exemplul 3. Proprietati

```
using System;
class Point
{
   double my_x;
   double my_y;
   public double x
   {
      get { return my_x; }
      set { if (value<=0) my_x = value;      else my_x=-value;  }
   }
   public double y
   {
      get { return my_y; }
      set {  if (value <= 0) my_y = value;     else my_y = -value; }
   }
}
class Segmdr
{
   public static void Main()
   {
      Point punct1 = new Point();  Point punct2 = new Point();
      double dist;
      punct1.x = 3;      punct1.y = 4;
      punct2.x = 5;      punct2.y = 3;
      dist = Math.Sqrt((punct1.x - punct2.x) * (punct1.x - punct2.x) + (punct1.y - punct2.y) * (punct1.y - punct2.y));
      Console.WriteLine("Distanta dintre punctele ({0},{1}) si ({2},{3}) este: {4:#.##}", punct1.x, punct1.y, punct2.x,
     punct2.y, dist);
   }
}
```

Rezultat:

Distanta dintre punctele (-3,-4) si (-5,-3) este: 2,24

Structuri

Clasele sunt tipuri de referinta. In consecinta obiectele claselor sunt accesate prin intermediul referintelor, spre deosebire de tipurile valorice care sunt accesate direct. Exista insa cazuri cand este util sa accesati un obiect in mod direct. Unul din motive o reprezinta eficienta.

Pentru a eficientiza programele, C# pune la dispozitie structurile. O structura este similara din punct de vedere sintactic unei clase, fiind insa de tip valoric si nu de referinta.

Forma generala a unei structuri este:

```
struct nume
{ //declaratii de membrii}
```

unde nume reprezinta numele structurii, iar struct reprezinta cuvantul cheie care declara structura.

Observatii: -ca si in cazul claselor, membrii structurilor pot fi *metode, variabile, indexari, proprietati, metode operator supraincarcate, evenimente*. De asemenea, se pot defini *constructori insa nu destructori*. In ceea ce priveste constructorii, *nu se pot defini constructori impliciti (fara parametri)* pentru structuri. Aceasta datorita faptului ca un constructor implicit este definit automat pentru toate structurile si acesta nu poate fi modificat;

-spre deosebire de clase, *structurile nu pot mosteni alte structuri sau alte clase* si nu pot fi utilizate ca baze pentru alte structuri sau clase. Ele pot implementa insa interfete;

-un obiect structura poate fi creat utilizand operatorul new in acelasi mod ca si pentru o instanta a unei clase, insa nu este obligatoriu. Daca nu se utilizeaza operatorul new atunci obiectul este creat fara a fi initializat. Initializarea trebuie facuta manual;

-structurile din C# difera din punct de vedere conceptual fata de structurile din C++. In C++ struct si class sunt aproape echivalente. Diferenta consta in accesul implicit la membrii, care este privat pentru class si public pentru struct. In C#, struct defineste un tip valoric, in timp ce class un tip de referinta.

Exemplul 4. Structuri

```
using System;
struct Cont {
    public string nume;
    public decimal sold;
    public Cont(string n, decimal s)   {   nume = n;   sold = s;  }
}
 class StructDemo{
    public static void Main()  {
        Cont c1 = new Cont("Gheorghe", 1244.926m);
        Cont c2 = new Cont();
        Cont c3;
        Console.WriteLine("{0} are un sold de {1:#.##}", c1.nume,c1.sold);
        Console.WriteLine();
        if (c2.nume == null)
        {
            Console.WriteLine("c2.nume este null");
            Console.WriteLine("c2.sold este {0}", c2.sold);
            Console.WriteLine();
        }
        c3.nume = "Ana";
        c3.sold = 100000m;
        Console.WriteLine("{0} are un sold de {1:C}", c3.nume, c3.sold);
        Console.WriteLine();
    }
}
```

Rezultat:
Gheorghe are un sold de 1.244,93 lei

c2.nume este null
c2.sold este 0

Ana are un sold de 100.000,00 lei

Enumerari

O *enumerare* este o multime de constante intregi cu nume, care specifica toate valorile posibile pe care le pot lua variabilele care au ca tip enumerarea.

Enumerarile sunt des intalnite in viata de zi cu zi. Spre exemplu, o enumerare a bancnotelor utilizate in Romania (1 leu, 5 lei, 10 lei, 50 lei, 100 lei, 200 lei, 500 lei) sau o enumerare a zilelor saptamanii (Luni, Marti etc.)

Un tip enumerare se declara utilizand cuvantul cheie enum. Forma generala a unei enumerari este:

```
acces enum nume {lista-constante};
```

unde numele tipului enumerare este specificat prin nume, iar lista-constante este o lista de identificatori separati prin virgule. Spre exemplu, secventa urmatoare de cod defineste o enumerare numita bancnote:

```
enum bancnote {unLeu, cinciLei, zeceLei, cincizeciLei, osutaLei, douasuteLei, cincisuteLei};
```

Observatii:-aspectul esential care trebuie accentuat este ca fiecare constanta dintr-o enumerare reprezinta o valoare intreaga. Fiecare constanta simbolica primeste o valoare mai mare cu o unitate fata de constanta precedenta din enumerare. In mod implicit, valoarea primei constante este 0. In cazul exemplului de mai sus constanta unLeu are valoarea 0, constanta cinciLei are valoarea 1, etc

-membrii unei enumerari pot fi accesati prin intermediul numelui tipului enumerarii cu ajutorul operatorului punct (.). Astfel, intructiunile: bancnote bancnotaMea=bancnote.zeceLei;

Console.WriteLine("{0}:{1}", bancnotaMea, (int)bancnotaMea); afiseaza zeceLei: 2.

-se poate preciza in mod explicit valoarea uneia sau mai multor constante simbolice, ca in exemplul de mai jos. In acest caz constantele simbolice care urmeaza vor primi valori mai mari decat cele situate inaintea lor:

```
enum bancnote {unLeu, cinciLei, zeceLei, cincizeciLei=50, osutaLei, douasuteLei, cincisuteLei};
```

Valorile constantelor sunt acum: unLeu 0, cinciLei 1, zeceLei 2, cincizeciLei 50, osutaLei 51, douasuteLei 52, cincisuteLei 53

-in mod implicit, enumerarile au tipul int. Puteti insa sa creati o enumerare de orice tip intreg, cu exceptia lui char. Pentru a specifica un alt tip de baza decat int, tipul trebuie precizat dupa numele enumerarii, fiind separat de acesta prin doua puncte (ca in exemplul de mai jos).

```
enum bacnote: byte {unLeu, cinciLei, zeceLei, cincizeciLei, osutaLei, douasuteLei, cincisuteLei};
```

Exemplul 5. Enumerari

```
using System;
public class Bday
{
enum Month
{   January = 1, February,  March,  April, May, June, July , August, September, October, November,
      December }

 struct birthday
 {
    public Month bmonth;
    public int bday;
    public int byear;
 }

 public static void Main()
 {
     birthday MyBirthday;

     MyBirthday.bmonth = Month.August;
     MyBirthday.bday = 11;
     MyBirthday.byear = 1955;

    System.Console.WriteLine("My birthday is {0} {1}, {2}",
    MyBirthday.bmonth, MyBirthday.bday, MyBirthday.byear);
 }
 }
```

Rezultat:

My birthday is August 11, 1955

Notiuni despre mostenire

Introducere

Mostenirea, alaturi de polimorfism, incapsulare si reutilizare a codului reprezinta principiile fundamentale ale programarii orientate obiect.

Utilizand mostenirea, se poate crea o clasa generala, care defineste caracteristici comune unei multimi si care poate fi mostenita de alte clase, mai specifice, fiecare adaugand numai caracteristicile care o identifica in mod unic.

O clasa care este mostenita se numeste *clasa de baza*, iar clasa care mosteneste anumite caracteristici se numeste *clasa derivata*.

Forma generala a unei declaratii class care mosteneste o clasa de baza este:

```
class nume-clasa-derivata : nume-clasa-baza
{
   //corpul clasei derivate
}
```

Spre deosebire de C++, in C# orice clasa derivata pe care o creati poate mosteni doar o singura clasa de baza. Se poate insa sa creati o ierarhie de clase, in care o clasa derivata sa devina clasa de baza din care sunt derivate alte clase.

In exemplul urmator este creata o clasa de baza numita Persoana care memoreaza numele, locul nasterii, data nasterii, dupa care creeaza o clasa derivata numita Student care adauga campul facultatea si metoda showStud().

Exemplul 1. Mostenire

```
using System;
class Persoana
{
   public string nume;   public string locnas;    public string datanas;
   public void showPers()
   {   Console.WriteLine("{0} s-a nascut in {1} la data de {2}",nume, locnas, datanas);    }
}
class Student:Persoana
{
   public string facultatea;
   public void showStud()
   {    Console.WriteLine(“Este student(a) la Facultatea de {0}",facultatea);   }
}

class PersoanaDemo
{
   public static void Main()
   {
      Student s1 = new Student();
      Student s2 = new Student();
      s1.nume = "Popescu Vasile";   s1.locnas = "Suceava";  s1.datanas = "23 iunie 1985";
      s1.facultatea = "Matematica";   s1.showPers();   s1.showStud();
      Console.WriteLine();
      s1.nume = "Ionescu Ioana";      s1.locnas = "Vaslui";     s1.datanas = "1 martie 1986";
      s1.facultatea = "Fizica";      s1.showPers();      s1.showStud();
   }
}
```

Rezultat:

Popescu Vasile s-a nascut in Suceava la data de 23 iunie 1985

Este student(a) la Facultatea de Matematica

Ionescu Ioana s-a nascut in Vaslui la data de 1 martie 1986

Este student(a) la Facultatea de Fizica

Accesul la membrii

Adeseori membrii unei clase sunt declarati ca privati pentru a evita utilizarea neautorizata a acestora. Mostenirea unei clase nu anuleaza restrictiile impuse de accesul *privat* (membrii privati ai unei clase nu pot fi accesati de clasele derivate).

Pe de alta parte, interzicerea accesului la membrii privati este de multe ori prea severa. Pentru a depasi acest inconvenient, se pot utiliza proprietatile publice care sa asigure accesul la membrii privati sau in cazul mostenirii utilizarea membrilor *protected*. Un membru protected este acesibil din clasa data si din clasele derivate.

Prezentam o varianta a exemplului anterior in care o parte din membrii sunt implementati cu ajutorul proprietatilor, iar altii sunt protejati. *Observati ca au fost introdusi constructori, aceasta deoarece membrii protected nu pot fi initializati manual din afara clasei de baza si clasei derivate.*

Exemplul 2. Membrii protejati

```
using System;
class Persoana
{
    protected string nume;    protected string locnas;
    private string my_datanas;
    public string datanas    {
        get{return my_datanas;}
        set { my_datanas = value; }
    }
public Persoana(string n, string l, string d)    {
        nume = n; locnas = l; datanas = d;
    }
    public void showPers()
    { Console.WriteLine("{0} s-a nascut in {1} la data de
      {2}", nume, locnas, datanas); }
}
```

```
class Student : Persoana
{
    string my_facultatea;
    public string facultatea
    {
        get{return my_facultatea;}
        set { my_facultatea = value; }
    }
    public Student(string n, string l, string d, string f)
        : base(n, l, d)
    {
        facultatea=f;
    }

    public void showStud()
    { Console.WriteLine("Student(a) la Facultatea de {0}",
      facultatea); }
}

class PersoanaDemo
{
    public static void Main()
    {
        Student s1 = new Student("Popescu Ion", "Bacau",
      "22 mai 1984", "Matematica");
         s1.showPers(); s1.showStud();
    }
}
```

Rezultat:

Popescu Ion s-a nascut in Bacau la data de 22 mai 1984
Este student(a) la Facultatea de Matematica

Constructorii

Intr-o ierarhie, atat clasele de baza cat si cele derivate pot dispune de constructori proprii. Pentru construirea unui obiect apartinand unei clase derivate se utilizeaza ambii constructori (atat cel din clasa de baza cat si cel din clasa derivata). Constructorul clasei de baza construieste portiunea obiectului care apartine de clasa de baza, iar constructorul clasei derivate construieste portiunea care apartine de clasa derivata.

In practica se procedeaza astfel:

- atunci cand ambele clase nu dispun de constructori se utilizeaza constructorii impliciti;

-atunci cand doar clasa derivata defineste un constructor, obiectul clasei derivate se construieste utilizand pentru partea care corespunde clasei de baza constructorul implicit;

-atunci cand cand ambele clase dispun de constructori, se poate proceda in doua moduri. Fie se utilizeaza o clauza base (ca in exemplul anterior), apelandu-se constructorul clasei de baza pentru construirea partii obiectului care tine de clasa de baza. Fie nu este utilizata o clauza base, iar atunci partea care apartine clasei de baza se construieste cu ajutorul constructorului implicit.

In exemplul anterior a fost utilizata o clauza base pentru a utiliza facilitatile oferite de constructorul clasei de baza.

Ascunderea numelor

Este posibil ca o clasa derivata sa defineasca un membru al carui nume sa coincida cu numele unui membru din clasa de baza. In acest caz, membrul clasei de baza nu va fi vizibil in clasa derivata.

Din punct de vedere tehnic aceasta nu este o eroare, insa compilatorul va genera un mesaj de avertisment care informeaza utilizatorul ca un nume a fost ascuns. Daca intradevar se doreste ascunderea unui membru al clasei de baza atunci se poate utiliza cuvantul cheie new (ca in exemplul de mai jos) pentru ca mesajul de avertizare sa nu mai apara.

Exemplul 3. Ascunderea numelor

```
using System;
class A     {public int i = 1;}
class B : A
{ public new int i;
public B(int b)  {i = b;}
}
class DemoAB
{
 public static void Main()
{ B ob = new B(10);
Console.WriteLine("i={0}", ob.i);}
}
```

Rezultat:

i=10

Cuvantul cheie base poate fi utilizat si in alt scop decat cel utilizat in paragraful anterior. Se comporta oarecum similar cu this, referindu-se la clasa de baza a clasei derivate pentru care se utilizeaza. Aceasta versiune are forma generala: base.membru; unde membru poate desemna atat o metoda cat si o variabila. Cuvatul cheie base permite accesul la variabila sau metoda ascunsa.

Exemplul 4. Ascunderea numelor si utilizarea clauzei base

```
using System;
class A
{public int i = 1;
public void show()
{ Console.WriteLine("In clasa de baza i={0}", i); }
}
class B : A
{ new public int i;
public B(int a, int b)
{ base.i = a; i = b; }
new public void show()
{   base.show();
Console.WriteLine("In clasa derivata i={0}", i); }
}
class DemoAB
{
public static void Main()
{  B ob = new B(5, 10);
ob.show(); }
}
```

Rezultat:

In clasa de baza i=5
In clasa derivata i=10

Ierarhii pe mai multe nivele

Pana acum, am creat ierarhii simple care contin numai o clasa de baza si o clasa derivata. Se pot insa construi ierarhii care contin oricate niveluri de mostenire doriti. Utilizarea unei clase derivate pe post de clasa de baza pentru o alta clasa este corecta.

Mai jos, este utilizata clasa derivata Student pe post de clasa de baza pentru a creia clasa derivata StudentMath.

Un alt aspect important: base refera intodeauna constructorul din clasa de baza cea mai apropiata.

De asemenea, intr-o ierarhie de clase, constructorii sunt apelati in ordinea derivarii: de la clasa de baza la clasele derivate.

Exemplul 5. Ierarhii pe doua nivele

```
using System;
class Persoana {
    protected string nume;
    protected string locnas;
    private string my_datanas;
    public string datanas    {
        get{return my_datanas;}
        set { my_datanas = value; }
    }
    public Persoana(string n, string l, string d)    {
        nume = n; locnas = l; datanas = d;
    }
    public void showPers()    {
    Console.WriteLine("{0} s-a nascut in {1} la data de
      {2}", nume, locnas, datanas); }
}
```

```
class Student : Persoana
{   string my_facultatea;
    public string facultatea
    {   get { return my_facultatea; }
        set { my_facultatea = value; }
    }
    public Student(string n, string l, string d, string f) : base(n, l,
        d)
    {  facultatea = f; }
 }
class StudentMath : Student
{    string my_specializarea;
    public string specializarea
    {   get { return my_specializarea; }
        set { my_specializarea = value; }
    }
    public StudentMath(string n, string l, string d, string f, string
        s)
        : base(n, l, d,f)
    {    specializarea = s;   }
     public void showStudMath()
    { Console.WriteLine("Student(a) la Facultatea de {0},
        specializarea {1}", facultatea, specializarea); }
}
class PersoanaDemo{
    public static void Main()  {
        StudentMath s1 = new StudentMath("Popescu Ion",
        "Bacau", "22 august 1984", "Matematica", "Matematica-
        Informatica");
        s1.showPers(); s1.showStudMath();
    }
}
```

Rezultat:

Popescu Ion s-a nascut in Bacau la data de 22 august 1984
Student(a) la Facultatea de Matematica, specializarea Matematica-Informatica

Variabilelor de tipul clasei de baza le pot fi atribuite instante ale claselor derivate

Limbajul C# este puternic tipizat. In afara conversiilor standard si a promovarilor automate care se aplica tipurilor simple, *compatibilitatea tipurilor este strict controlata*. Aceasta inseamna ca o variabila referinta de tipul unei clase nu poate, in majoritatea cazurilor, sa refere un obiect apatinand unei alte clase.

Exista insa o exceptie. O variabila referinta de tipul unei clase de baza poate sa refere un obiect apartinand oricarei clase derivate din clasa de baza.

In acest *caz tipul variabilei referinta si nu obiectul la care acesta se refera determina membrii care pot fi accesati*. In consecinta, cand se atribuie o referinta catre o instanta a clasei derivate unei variabile referinta de tipul clasei de baza, veti avea acces numai la partile obiectului definite in clasa de baza. In programul urmator (Exemplul 6), x2 nu are acces la variabila b.

Din punct de vedere practic, exista doua cazuri importante cand se atribuie referinte catre clase derivate unor variabile de tipul clasei de baza. Un prim caz este la apelul constructorilor intr-o ierarhie de clase (vezi exemplul 7, unde constructorii accepta ca parametrii instante ale claselor lor). Celalalt caz se refera la metode virtuale.

Exemplul 6. Atribuirea unor instante ale claselor derivate variabilelor de tipul clasei de baza

```
using System;
class A
{
    public int a;
    public A(int i) { a = i; }
}
class B : A
{
    public int b;
    public B(int i, int j)   : base(i)
    {
        b = j;
    }
}
class RefBaza
{
    public static void Main()
    {
        A x = new A(10);
        A x2;
        B y = new B(3, 4);
        x2 = x;                  //corect, fiind de acelasi tip
        Console.WriteLine("x2.a={0}", x2.a);
        x2 = y;                 //corect deoarece B deriva din A
        Console.WriteLine("x2.a={0}",x2.a);
        //Console.WriteLine("x2.b={0}", x2.b); //eroare clasa A nu contine membrul b
     }
}
```

Rezultat:

x2.a=10

x2.a=3

Exemplul 7. In programul de mai jos constructorul clasei Persoana (referit prin clauza base) asteapta un obiect Persoana, insa Student() ii transmite un obiect de tipul Student. Motivul pentru care se poate proceda astfel este ca o referinta de tipul clasei de baza poate referi o instanta a unei clase derivate.

```
using System;
class Persoana
{
   protected string nume;
   protected string locnas;
   private string my_datanas;
   public string datanas
   {
      get{return my_datanas;}
      set { my_datanas = value; }
   }
   public Persoana(string n, string l, string d)
   {
      nume = n; locnas = l; datanas = d;
   }
   public Persoana(Persoana ob)
   {
      nume = ob.nume; locnas = ob.locnas; datanas =
      ob.datanas;
   }
   public void showPers()
   { Console.WriteLine("{0} s-a nascut in {1} la data de
      {2}", nume, locnas, datanas); }
}
```

```
class Student : Persoana
{   string my_facultatea;
   public string facultatea
   {   get { return my_facultatea; }
      set { my_facultatea = value; }
   }
   public Student(string n, string l, string d, string f)
      : base(n, l, d)
   {  facultatea = f;  }
   public Student(Student ob)
      : base(ob)
   { facultatea = ob.facultatea;  }
   public void showStud()
   { Console.WriteLine("Student(a) la Facultatea de {0}",
      facultatea); }
}
class PersoanaDemo
{  public static void Main()
   {   Student s1 = new Student("Popescu Ion", "Bacau",
      "22 august 1984", "Matematica");
      Student s2 = new Student(s1);
      Console.WriteLine("Informatii despre s1:");
      s1.showPers(); s1.showStud();
      Console.WriteLine();
      Console.WriteLine("Informatii despre s2:");
      s1.showPers(); s1.showStud();
   }
}
```

Rezultat:

Informatii despre s1:
Popescu Ion s-a nascut in Bacau la data de 22 august 1984
Student(a) la Facultatea de Matematica

Informatii despre s2:
Popescu Ion s-a nascut in Bacau la data de 22 august 1984
Student(a) la Facultatea de Matematica

Metode virtuale

O metoda virtuala este o metoda declarata ca virtual intr-o clasa de baza si redefinita apoi in una din clasele derivate. Fiecare din clasele derivate pot dispune de o versiune proprie a unei metode virtuale.

Metodele virtuale sunt interesante atunci cand sunt apelate prin intermediul unei referinte de tipul clasei de baza. In acest caz compilatorul determina versiunea care va fi apelata tinand cont de tipul obiectului referit de referinta si nu de tipul referintei. Determinarea se efectueaza la momentul executiei. Cu alte cuvinte tipul obiectului (si nu tipul referintei) determina care versiune a metodei virtuale se va executa. Astfel daca o clasa de baza contine o metoda virtuala si exista clase derivate din clasa de baza atunci cand se refera tipuri diferite de obiecte prin intermediul unei referinte de tipul clasei de baza, se vor executa versiuni diferite ale metodei virtuale.

Declararea unei metode virtuale in clasa de baza se realizeaza cu cuvantul cheie virtual. La redefinirea metodei in cadrul claselor derivate se utilizeaza modificatorul override. Acest proces de redefinire a unei metode se numeste extindere a metodei. La extinderea unei metode numele si signatura metodei extinse trebuie sa coincida cu numele si signatura metodei virtuale care se extinde. Metodele virtuale nu pot fi declarate ca static sau abstract.

Extinderea metodelor reprezinta baza pentru unul dintre cele mai puternice concepte implementate in C#: apelarea dinamica a metodelor. Apelarea dinamica a metodelor este mecanismul prin care rezolvarea apelului unei metode extinse este amanata de la momentul compilarii pana la momentul executiei programului. Apelarea dinamica a metodelor este modalitatea prin care limbajul C# implementeaza polimorfismul la executie.

Exemplul 8. Tipul obiectului referit (si nu tipul referintei) este cel care stabileste ce versiune a metodei virtuale se va executa.

```
using System;
class Baza
{    public virtual void Care()
   {  Console.WriteLine("Metoda Care() din clasa de baza"); }
}
class Derivata1:Baza
{    public override void Care()
  {  Console.WriteLine("Metoda Care() din clasa Derivata1"); }
}
class Derivata2 : Derivata1
{   public override void Care()
   {  Console.WriteLine("Metoda Care() din clasa Derivata2");
      }
}
class ExtindereDemo
{
   public static void Main()
   {
      Baza obB = new Baza();
      Derivata1 obD1 = new Derivata1();
      Derivata2 obD2 = new Derivata2();
      Baza a;
      a = obB;   a.Care();
      a = obD1;    a.Care();
      a = obD2;     a.Care();
   }
}
```

Rezultat:

Metoda Care() din clasa de baza
Metoda Care() din clasa Derivata1
Metoda Care() din clasa Derivata2

Exemplul 9. Nu este obligatoriu ca o metoda virtuala sa fie extinsa. Daca o clasa derivata nu contine o versiune proprie a unei metode virtuale atunci se utilizeaza versiunea din clasa de baza.

```
using System;
class Baza
{    public virtual void Care()
   { Console.WriteLine("Metoda Care() din clasa de
      baza"); } }
class Derivata1 : Baza  {   }
class Derivata2 : Derivata1
{    public override void Care()
   { Console.WriteLine("Metoda Care() din clasa
      Derivata2"); } }
class ExtindereDemo
{
   public static void Main()
   {
      Baza obB = new Baza();
      Derivata1 obD1 = new Derivata1();
      Derivata2 obD2 = new Derivata2();
      Baza a;
      a = obB; a.Care();
      a = obD1; a.Care();
      a = obD2; a.Care();
   }
}
```

Rezultat:

Metoda Care() din clasa de baza
Metoda Care() din clasa de baza
Metoda Care() din clasa Derivata2

Exemplul 10. Extinerea metodei show() din cadrul exemplelor anterioare

```
using System;
class Persoana
{
   protected string nume;
   protected string locnas;
   private string my_datanas;
   public string datanas
   {
      get{return my_datanas;}
      set { my_datanas = value; }
   }
   public Persoana(string n, string l, string d)
   {
      nume = n; locnas = l; datanas = d;
   }
   public virtual void show()
   { Console.WriteLine("{0} s-a nascut in {1} la data de {2}", nume, locnas,
        datanas); }
}
class Student : Persoana
{
   string my_facultatea;
   public string facultatea
   {
      get { return my_facultatea; }
      set { my_facultatea = value; }
   }
   public Student(string n, string l, string d, string f)
      : base(n, l, d)
   {
      facultatea = f;
   }

   public override void show()
   { Console.WriteLine("{0} s-a nascut in {1} la data de {2}. Este student(a)
        la Facultatea de {3}", nume, locnas, datanas, facultatea); }
}
```

```
class StudentMath : Student
{
   public string specializarea;
   public StudentMath(string n, string l, string d, string f, string s)
      : base(n, l, d, f)
   {
      specializarea = s;
   }
   public override void show()
   { Console.WriteLine("{0} s-a nascut in {1} la data de {2}. Este student(a)
        la Facultatea de {3}, specializarea {4}", nume, locnas, datanas,
        facultatea, specializarea); }
}

class PersoanaDemo
{
   public static void Main()
   {
      Persoana p1 = new Persoana("Vasilescu Vasile", "Iasi", "11 februarie
        1966");
      Student s1 = new Student("Popescu Gina", "Bacau", "22 august
        1984", "Chimie");
      StudentMath sm1 = new StudentMath("Ionescu Ion", "Botosani", "30
        martie 1986", "Matematica", "Matematica-Informatica");
      p1.show();
      Console.WriteLine();
      s1.show();
      Console.WriteLine();
      sm1.show();
   }
}
```

Rezultat:

Vasilescu Vasile s-a nascut in Iasi la data de 11 februarie 1966

Popescu Gina s-a nascut in Bacau la data de 22 august 1984. Este student(a) la Facultatea de Chimie

Ionescu Ion s-a nascut in Botosani la data de 30 martie 1986. Este student(a) la Facultatea de Matematica, specializarea Matematica-Informatica

Utilizarea metodelor abstracte

In unele situatii este necesar sa creati o clasa de baza care stabileste un sablon general pentru toate clasele derivate. O astfel de clasa fixeaza un set de metode, numite metode abstracte, pe care clasele derivate trebuie sa le implementeze.

O metoda abstracta se poate crea specificand modificatorul de tip abstract. Metodele abstracte nu au corp si nu sunt implementate in clasa de baza. Clasele derivate trebuie sa le extinda—nu se admite sa utilizeze versiunea definita de clasa de baza. Metodele abstracte sunt in mod implicit virtuale. In fapt, utilizarea ambilor modificatori virtual si abstract genereaza eroare.

Pentru declararea unei metode abstracte se utilizeaza sintaxa:

public abstract tip nume(lista-parametri);

Metoda nu are corp, este publica si nu este permisa utilizarea sa in cazul metodelor statice.

O clasa care contine o metoda abstracta trebuie declarata ea insasi abstracta, precedand declaratia class cu specificatorul abstract. Deoarece clasele abstracte nu definesc implementari complete, nu se pot instantia. Incercarea de a crea un obiect cu ajutorul operatorului new, genereaza eroare.

Atunci cand o clasa derivata mosteneste o clasa abstracta trebuie sa implementeze toate metodele abstracte din clasa de baza.

Exemplul 11. Utilizarea metodelor abstracte

```
using System;
abstract class Forma2D
{
    public int lungime;
    public int inaltime;
    public string nume;

    public Forma2D(int l, int i, string n)
    { lungime = l; inaltime = i; nume = n; }

    public abstract double aria();
}
class Triunghi : Forma2D
{
    public string style;
    public Triunghi(int l, int i, string n, string s)
        : base(l, i, n)
    {
        style = s;
    }
    public override double aria()
    {
        return lungime * inaltime / 2;
    }

}
```

```
class Dreptunghi : Forma2D
{
    public Dreptunghi(int l, int i, string n)
        : base(i, i, n)
    {  }
    public override double aria()
    {
        return lungime * inaltime;
    }
}
class DemoAbs
{
    public static void Main()
    {
        Triunghi t1 = new Triunghi(8, 4, "triunghi", "dreptunghic
        isoscel");
        Triunghi t2 = new Triunghi(5, 7, "triunghi", "oarecare");
        Dreptunghi d1 = new Dreptunghi(3, 4, "dreptunghi");
        Console.WriteLine("Obiectul t1 este un  {0} {1} avand
        aria={2}", t1.nume, t1.style, t1.aria());
        Console.WriteLine("Obiectul t2 este un  {0} {1} avand
        aria={2}", t2.nume, t2.style, t2.aria());
        Console.WriteLine("Obiectul d1 este un  {0} avand aria={1}",
        d1.nume, d1.aria());
//Forma2D f2d=new Forma2D(); Eroare!, Forma2D este abstracta
    }
}
```

Rezultat:

Obiectul t1 este un triunghi dreptunghic isoscel avand aria=16
Obiectul t2 este un triunghi oarecare avand aria=17
Obiectul d1 este un dreptunghi avand aria=16

Utilizarea cuvantului cheie sealed

In cazul in care doriti ca o clasa sa nu fie mostenita, utilizati cuvantul cheie sealed. Declaratia clasei trebuie precedata de sealed. Spre exemplu: sealed class A { } nu permite mostenirea clasei A. Este incorect sa declarati o clasa abstract si sealed.

Clasa object

Limbajul C# defineste o clasa speciala, numita object, care este clasa de baza implicita pentru toate celelalte clase si pentru toate celelalte tipuri, inclusiv tipurile valorice. Astfel, toate celelalte tipuri deriva din clasa object. Din punct de vedere tehnic, numele C# object este sinonim pentru System.Object, din arhitectura .NET.

Clasa object defineste mai multe metode, disponibile pentru orice obiect. Spre exemplu:

public virtual bool Equals (object ob)	Determina daca obiectul apelant este acelasi ca si cel referit de ob
public static bool Equals(object ob1, object ob2)	Determina daca ob1 si ob2 refera acelasi obiect.
public virtual string ToString()	Intoarce un string care descrie obiectul. Aceasta metoda se apeleaza automat atunci cand obiectul este afisat utilizand WriteLine().
protected Finalize()	Executa actiunile de distrugere a obiectului, anterioare colectarii spatiului ocupat de acesta. Finalize() se acceseaza prin intermediul destructorilor.
protected object MemberwiseClone()	Efectueaza o copie superficiala a obiectului. Acesta contine toti membrii obiectului, dar nu si obiectele pe care acestia le refera.

Exemplele 12 si 13. Metode ale clasei object in actiune

```
using System;
class Persoana
{
   public string nume;     public string datanas;
   public Persoana(string n, string d)
   {       nume = n; datanas = d;    }
}
class Impdesp
{    public static void Main()
   {   string n = "Ana";    string d = "1975";     bool a;
       Persoana p1 = new Persoana(n, d);
       Persoana p2 = new Persoana(n, d);
       Persoana p3;
       p3 = p1;
       a = Equals(p1,p2);
       Console.WriteLine("a={0}",a);

       a=p1.Equals(p2);
       Console.WriteLine(“a={0}”,a);

       a = Equals(p1, p3);
       Console.WriteLine("a={0}", a);

       int i = 10, j=10;
       a = Equals(i, j);
       Console.WriteLine("a={0}",a);
   }
}
```

Rezultat:

a=False
a=False
a=True
a=True

```
using System;
class MyClass
{
   static int count = 1;
   int id;
   public MyClass()
   {
      id = count;
      count++;
   }
   public override string ToString()
   {
     return "Instanta MyClass #"+id;
   }
}
class Test
{
   public static void Main()
   {
      MyClass ob1 = new MyClass();
      MyClass ob2 = new MyClass();
      MyClass ob3 = new MyClass();
      Console.WriteLine(ob1);
      Console.WriteLine(ob2);
      Console.WriteLine(ob3);
   }
}
```

Rezultat:

Instanta MyClass #1
Instanta MyClass #2
Instanta MyClass #3

Metode de extindere

Functionalitatea unor clase poate fi extinsa fara a include alte campuri sau membrii clasei respective.

Acest lucru este posibil daca este implementata o metoda de extinedere in orice alta clasa.

Se utilizeaza cuvantul cheie this (ca in exemplul alaturat).

Daca metoda primeste si alti parametrii atunci acestia sunt pozitionati dupa this (vezi exemplul).

In exemplul alaturat functionalitatea stringurilor este extinsa de metoda MetodaDeExtindere.

Observatie. Pentru compilare, este necesar referirea asamblajului System.Core.dll

```
using System;
namespace SpatiuDeNume
{
   public static class ExtindereClass
   {
      public static void MetodaDeExtindere( this
     string obiect, int a)
      {
         Console.WriteLine("Aceasta metoda
     extinde clasa string");
         Console.WriteLine(obiect);
         Console.WriteLine(a);
      }
   }
   class Demo
   {
      public static void Main()
      {
         string s = "Un string.";
         s.MetodaDeExtindere(5);
      }
   }
}
```

Impachetare si despachetare

Toate tipurile valorice sunt derivate din object. Asadar o referinta de tip object se poate utiliza pentru a referi orice alt tip, inclusiv un tip valoric. *Atunci cand o referinta de tip object refera un tip valoric, are loc un fenomen numit impachetare*. Impachetarea determina memorarea valorii apartinand unui tip valoric intr-un obiect. Astfel, tipul valoric este impachetat intr-un obiect. Obiectul poate fi folosit ca oricare altul. *Impachetarea se realizeaza prin atribuirea valorii dorite unei referinte de tip obiect.*

Despachetarea este operatia de determinare a valorii continute de un obiect. Aceasta se utilizeaza folosind un cast al referintei obiectului la tipul dorit.

Programele alaturate demonstreaza impachetarea si despachetarea. In cel de-al doilea program se observa cum valoarea 10 este automat impachetata intr-un obiect.

Exemplele 14 si 15. Impachetare si despachetare

```
using System;
class BoxDemo
{
   public static void Main()
   {
      int x;
      object ob;
      x = 10;
      ob = x; //impachetam x intr-un obiect

      int y = (int)ob; //despachetam obiectul ob intr-o
      //valoare de tip int
      Console.WriteLine(y);
   }
}
```

Rezultat:

10

```
using System;
class BoxDemo
{
   public static void Main()
   {
      string l;
      l = 10.ToString();
      Console.WriteLine(l);
   }
}
```

Rezultat:

10