Interfete, Delegari, Evenimente

Definirea si implementarea interfetelor

O interfata *defineste un set de metode care vor fi implementate de una sau mai multe clase*. O *interfata nu implementeaza metode* ci doar precizeaza ce anume va contine o clasa care implementeaza interfata. Din punct de vedere sintactic, interfetele sunt similare claselor abstracte. Insa, exista mai multe aspecte care le diferentiaza. Spre exemplu: daca in clasele abstracte unele metode erau implementate iar altele nu*, in cazul interfetelor toate metodele nu pot avea corp (nu sunt implementate)*.

O interfata poate contine doar: *metode, proprietati, evenimente si indexari*. Interfetele nu pot contine variabile, constructori sau destructori.

Daca o clasa nu poate mosteni decat o clasa, in schimb o *clasa poate implementa oricate interfete*. De asemenea, o interfata poate fi implementata de oricate clase. Asadar, este posibil ca doua clase sa implementeze aceeasi interfata in moduri diferite.

Daca o clasa implementeaza o interfata atunci *aceasta trebuie sa implementeze toti membrii interfetei.* Prin intermediul interfetelor, limbajul C# permite beneficierea la maximum de *aspectul “o singura interfata mai multe metode” al polimorfismului.*

Interfetele se declara utilizand cuvantul cheie interface. O forma simplificata a unei interfete (contine doar metode) este:

```
acces interface nume
{ tip-rez nume-metoda1 (lista-param);
  tip-rez nume-metoda2 (lista-param);
  //….
  tip-rez nume-metodaN (lista-param); }
```

unde nume reprezinta numele intefetei. De remarcat faptul ca metodele sunt declarate utilizand numai tipul lor si signatura. *Metodele unei interfete sunt in mod implicit publice, nefiind permisa prezenta modificatorilor de acces*. De asemenea, *metodele interfetelor nu pot fi declarate ca statice*.

Dupa ce o interfata a fost declarata, una sau mai multe clase (sau structuri) o pot implementa. Forma generala a unei clase care implementeaza o interfata este:

```
class nume-clasa: nume-interfata
{//corpul clasei}
```

Daca o clasa (sau o structura) implementeaza o interfata atunci ea trebuie sa implementeze toti membrii interfetei. Daca o clasa implementeaza mai multe interfete, atunci numele interfetelor sunt separate prin virgula (class nume-clasa: nume-interfata1, nume-interfata2,…,nume-interfataN).

O clasa poate sa mosteneasca o clasa de baza si sa implementeze mai multe interfete. In acest caz, *numele clasei trebuie sa fie primul in lista separata prin virgule. Metodele care implementeaza o interfata trebuie declarate publice*, deoarece metodele sunt in mod implicit publice in cadrul interfetei. De asemenea, tipul si signatura metodei din cadrul clasei trebuie sa se potriveasca cu tipul si signatura metodei interfetei.

Exemplul 1. Utilizarea unei interfete

```
using System;
public interface Forma2D
{
   double Aria();      double LungFrontiera();
}
public class Cerc: Forma2D
{
   public double raza;     private const float PI = 3.14159f;
   public double Aria()    { return (PI * raza * raza);}
   public double LungFrontiera()    {return (2*PI*raza);}
   public Cerc(double r)    {raza=r;}
 }
public class Patrat : Forma2D
{
   public double latura;
   public double Aria()    {return (latura * latura);}
   public double LungFrontiera()    { return (4 * latura);}
   public Patrat(double l)    {latura = l;}
}
class IterfDemo
{   public static void Main()
   {   Cerc c = new Cerc(3);       Patrat p = new Patrat(3);
     Console.WriteLine("Afiseaza informatii despre cerc:\naria={0:#.##} \t lungimea frontierei={1:#.##}", c.Aria(),
      c.LungFrontiera());
       Console.WriteLine("\nAfiseaza informatii despre patrat:\naria={0:#.##} \t lungimea frontierei={1:#.##}",
     p.Aria(), p.LungFrontiera());
   }
}
```

Rezultat:

Afiseaza informatii despre cerc:
aria=28,27 lungimea frontierei=18,85
Afiseaza informatii despre patrat:
aria=9 lungimea frontierei=12

Referinte avand ca tip o interfata

Desi pare surprinzator, *puteti declara o variabila referinta avand ca tip o interfata*. O astfel de variabila poate referi orice obiect care implementeaza interfata.

La apelul unei metode prin intermediul referintei catre interfata, se va executa versiunea metodei care este implementata de obiectul apelant.

Exemplul 2

```
using System;
public interface Forma2D
{
    double Aria();
    double LungFrontiera();
}
public class Cerc : Forma2D
{
    public double raza; private const float PI = 3.14159f;

    public double Aria() { return (PI * raza * raza); }

    public double LungFrontiera() { return (2 * PI * raza); }

    public Cerc(double r) { raza = r; }
}
public class Patrat : Forma2D
{
    public double latura;

    public double Aria() { return (latura * latura); }

    public double LungFrontiera() { return (4 * latura); }

    public Patrat(double l) { latura = l; }
}
class IterfDemo
{
    public static void Main()
    {
        Cerc c = new Cerc(3);    Patrat p = new Patrat(3);

        Console.WriteLine("Afiseaza informatii despre cerc:");
        DisplayInfo(c);

        Console.WriteLine("\nAfiseaza informatii despre patrat:");
        DisplayInfo(p);
    }
    static void DisplayInfo(Forma2D f)
    {Console.WriteLine("aria={0:#.##} \t lungimea
        frontierei={1:#.##}", f.Aria(), f.LungFrontiera());}
}
```

Rezultat:

Afiseaza informatii despre cerc:
aria=28,27 lungimea frontierei=18,85
Afiseaza informatii despre patrat:
aria=9 lungimea frontierei=12

Utilizarea proprietatilor in interfete

Ca si metodele, *proprietatile se pot specifica in cadrul unei interfete fara a include corpul.*

In cazul proprietatilor accesibile atat la scriere cat si la citire vor aparea ambii accesori (get si set), in tip ce pentru proprietatile accesibile doar la citire (scriere) va aparea numai accesorul get (set).

Exemplul 3. Utilizarea unei proprietati accesibila la citire in cadrul unei interfete

```
using System;
public interface Forma2D
{
   double Aria();
   double LungFrontiera();
   string denumire { get; }
}
public class Cerc : Forma2D
{
   string s = "cerc";
   public double raza; private const float PI = 3.14159f;

   public double Aria() { return (PI * raza * raza); }
   public double LungFrontiera() { return (2 * PI * raza); }

   public string denumire{ get{return s;}}

   public Cerc(double r) { raza = r; }
}
public class Patrat : Forma2D
{
   string s="patrat";
   public double latura;

   public double Aria() { return (latura * latura); }
   public double LungFrontiera() { return (4 * latura); }
   public string denumire { get {return s; } }

   public Patrat(double l) { latura = l; }
}
class IterfDemo
{
   public static void Main()
   {
      Cerc c = new Cerc(3); Patrat p = new Patrat(3);
      Console.WriteLine("Afiseaza informatii despre {0}:",
      c.denumire);
      DisplayInfo(c);
      Console.WriteLine("\nAfiseaza informatii despre
      {0}:", p.denumire);
      DisplayInfo(p);
   }
   static void DisplayInfo(Forma2D f)
   {Console.WriteLine("aria={0:#.##} \t lungimea
      frontierei={1:#.##}", f.Aria(), f.LungFrontiera());}
}
```

Rezultat:

Afiseaza informatii despre cerc:
aria=28,27 lungimea frontierei=18,85
Afiseaza informatii despre patrat:
aria=9 lungimea frontierei=12

Implementari explicite

In exemplele anterioare totul a decurs firesc in ceea ce priveste implementarea interfetelor. Pot aparea insa o serie de probleme atunci cand se doreste implementarea mai *multor interfete care contin metode cu aceeasi denumire.*

Spre exemplu, daca o clasa implementeaza doua interfete care contin o metoda avand acelasi nume (sa zicem Display()) atunci o singura implementare a metodei Display() satisface ambele interfete.

Exista totusi situati cand doriti sa implementati in mod independent metoda Display() pentru ambele interfete. In acest caz, trebuie sa implementati interfetele in mod explicit*. O implementare explicita este realizata prin includerea numelui interfetei impreuna cu numele metodei*.

La realizarea unei implementari explicite *nu trebuie apelat modificatorul de acces public*. De fapt*, prin implementarea explicita, metoda devine practic private si nu poate fi accesata din afara clasei decat de o referinta de tipul interfetei sau utilizand un cast* (a se vedea exemplul 4). Practic, am implementat o metoda si in acelasi timp am ascuns-o.

Exemplul 4 prezinta aceste aspecte.

Exemplul 4. Implementari explicite

```
using System;
public interface Forma2D
{    double Aria(); double LungFrontiera();
     void Display();  string denumire { get; }  }

public interface Forma2DDisplay
{    void Display(); }

public class Cerc : Forma2D, Forma2DDisplay
{
    string s = "cerc";
    public double raza; private const float PI = 3.14159f;

    public double Aria() { return (PI * raza * raza); }
    public double LungFrontiera() { return (2 * PI * raza); }

    void Forma2D.Display()
    {
        Console.WriteLine("Afiseaza informatii despre {0}:",
        denumire);
        Console.WriteLine("aria={0:#.##}", Aria());
        Console.WriteLine("lungimea frontierei={0:#.##}",
        LungFrontiera());
    }
    void Forma2DDisplay.Display()
    {
        Console.WriteLine("Aceasta metoda ar putea furniza
        informatii despre cerc");
    }
    public string denumire { get { return s; } }

    public Cerc(double r) { raza = r; }
}
```

```
public class Patrat : Forma2D, Forma2DDisplay
{    string s="patrat";    public double latura;
     public double Aria() { return (latura * latura); }
     public double LungFrontiera() { return (4 * latura); }
     void Forma2D.Display()
     {   Console.WriteLine("Afiseaza informatii despre {0}:",
         denumire);
         Console.WriteLine("aria={0:#.##}", Aria());
         Console.WriteLine("lungimea frontierei={0:#.##}",
         LungFrontiera());   }
     void Forma2DDisplay.Display()
     {     Console.WriteLine("Aceasta metoda ar putea
         furniza informatii despre patrat");    }
     public string denumire { get {return s; } }
     public Patrat(double l) { latura = l; }
}
class IterfDemo
{    public static void Main()
     {  Cerc c = new Cerc(3); Patrat p = new Patrat(3);
         Forma2D f1 = (Forma2D)c;        f1.Display();
         Forma2DDisplay f2 = (Forma2DDisplay)c;
         f2.Display();
         Console.WriteLine();
         Forma2D f3 = (Forma2D)p;        f3.Display();
         Forma2DDisplay f4 = (Forma2DDisplay)p;
         f4.Display();
 } }
```

Rezultat:

Afiseaza informatii despre cerc:
aria=28,27
lungimea frontierei=18,85
Aceasta metoda ar putea furniza informatii despre cerc

Afiseaza informatii despre patrat:
aria=9
lungimea frontierei=12
Aceasta metoda ar putea furniza informatii despre patrat

Interfetele pot fi mostenite

O interfata poate fi mostenita de o alta interfata. Sintaxa este comuna cu cea care se utilizeaza la mostenirea claselor.

Daca o clasa implementeaza o interfata care mosteneste o alta interfata, clasa trebuie sa contina implementari pentru toti membrii definiti pe lantul de mostenire.

Exemplul 5 ilustreaza aceste aspecte.

Daca incercati sa eliminati implemetarea metodei Metoda1() din clasa Myclass, se va produce o eroare la compilare.

Exemplul 5. Implementarea unui lant de mostenire

```
using System;
public interface A
{    void Metoda1();  }
public interface B : A
{    void Metoda2();    void Metoda3(); }
class Myclass : B
{
    public void Metoda1()
    { Console.WriteLine("Implementarea metodei 1"); }
    public void Metoda2()
    { Console.WriteLine("Implementarea metodei 2");  }
    public void Metoda3()
    { Console.WriteLine("Implementarea metodei 3"); }
}
class MostenireDemo
{
    public static void Main()
    {
        Myclass ob = new Myclass();
        ob.Metoda1();
        ob.Metoda2();
        ob.Metoda3();
    }
}
```

Rezultat:

Implementarea metodei 1
Implementarea metodei 2
Implementarea metodei 3

Delegari

O delegare reprezinta un tip referinta care *executa metode avand acelasi format (acelasi tip rezultat si acelasi numar si tip de parametrii)*.

Intr-un limbaj mai detaliat, chiar daca o metoda nu este un obiect, ea ocupa o locatie in memoria fizica. La invocarea metodei, controlul se transfera la adresa punctului de intrare in metoda. Aceasta adresa poate fi atribuita unei delegari si astfel metoda poate fi apelata prin intermediul delegarii. Mai mult, acceasi delegare poate fi utilizata pentru a apela si alte metode, modificand pur si simplu metoda referita.

Delegarile sunt similare pointerilor catre functii in C si C++.

Delegarile se declara utilizand cuvantul cheie delegate. Forma generala a unei delegari este:

acces delegate tip-rez nume (lista-parametrii);

unde tip-rez este tipul valorii intoarse de metodele pe care delegarea le va apela, numele delegarii este specificat prin nume, iar parametrii necesari metodelor care vor fi apelate prin intermediul delegarii sunt specificati prin lista-parametrii.

Delegarile sunt importante din *doua motive.* In primul rand, *delegarile permit implementarea evenimentelor*, iar in al doilea rand, *delegarile amana determinarea metodei invocate pana la momentul executiei*. Aceasta din urma capacitate se dovedeste utila atunci cand creati o arhitectura care permite adaugarea componentelor pe parcurs.

Exemplele 6 si 7. Delegarile pot apela atat metode statice (exemplul 6) cat si metode ale obiectelor (exemplul 7)

```
using System;
public delegate double Mydelegate(double a);
class DelegateDemo
{
        const float PI = 3.14159f;

     static double Aria(double r)
    { return (PI * r * r); }

    static double LungFrontiera(double r)
    { return (2 * PI * r); }

    public static void Main()
    {
        double raza=3;

        Mydelegate del = new Mydelegate(Aria);

        Console.WriteLine("Aria= {0:#.##}", del(raza));

        del = new Mydelegate(LungFrontiera);

        Console.WriteLine("Lungimea frontierei={0:#.##}",
        del(raza));
    }
}
```

Rezultat:
Aria=28,27
Lungimea frontierei=18,85

```
using System;
public delegate double Mydelegate();
public class Cerc
{
    public double raza; private const float PI = 3.14159f;

    public double Aria()
    { return (PI * raza * raza); }

    public double LungFrontiera()
    { return (2 * PI * raza); }

    public Cerc(double r) { raza = r; }
}

class DelegateDemo
{
    public static void Main()
    {
        Cerc c = new Cerc(3);
        Mydelegate del = new Mydelegate(c.Aria);
        Console.WriteLine("Aria= {0:#.##}", del());
        del = new Mydelegate(c.LungFrontiera);
        Console.WriteLine("Lungimea frontierei={0:#.##}",
        del());
    }
}
```

Rezultat:
Aria=28,27
Lungimea frontierei=18,85

Multicasting

Una dintre cele mai interesante facilitati oferite de delegari o reprezinta *capacitatea de multicasting. Multicastingul reprezinta capacitatea de a crea un lant de metode care vor fi automat apelate la invocarea unei delegari.*

Un astfel de lant este usor de creat. Se instantiaza mai intai o delegare, iar apoi utilizand operatorul += se adauga metode in lant sau utilizand operatorul -= se elimina metode din lant.

Exista o restrictie importanta, si anume: *delegarile multicast trebuie sa intoarca un rezultat de tip void*.

Exemplul alaturat rescrie exemplul 6 modificand tipul intors si utilizand parametrul out pentru a intoarce aria si lungimea frontierei in modulul apelant.

Exemplul 8. Multicasting

```
using System;
public delegate void Mydelegate(double r, out double a);
class DelegateDemo
{   const float PI = 3.14159f;
    static void Aria(double r,out double a)
    { a=PI * r * r; }
    static void LungFrontiera(double r, out double lf)
    { lf=2 * PI * r; }
    static void Cerc(double r, out double b)
    { Console.WriteLine("Cercul de raza r={0} are", r);
    b = 0;  }

    public static void Main()
    {
        double raza=3, a,lf;
        Mydelegate del = new Mydelegate(Cerc);
        del += new Mydelegate(Aria);
        del(raza, out  a);
        Console.WriteLine("Aria= {0:#.##}", a);

        del += new Mydelegate(LungFrontiera);
        del(raza, out lf);
        Console.WriteLine("Lungimea frontierei={0:#.##}",
        lf);
    }
}
```

Rezultat:

Cercul de raza r=3 are
Aria= 28,27
Cercul de raza r=3 are
Lungimea frontierei=18,85

Evenimentele

Pe fundamentul reprezentat de delegari, limbajul C# a construit o alta facilitate importanta: evenimentele.

Un *eveniment* reprezinta, in esenta, *notificarea automata din partea unei clase ca s-a produs o actiune in program*. Pe baza acestei instiintari puteti raspunde cu o rutina pentru tratarea acestei actiuni.

Evenimentele functioneaza dupa urmatorul mecanism: *un obiect care este intresat de un eveniment isi inregistreaza o rutina de tratare a acelui eveniment. Atunci cand evenimentul se produce, se apeleaza toate rutinele inregistrate*. Rutinele de tratare a evenimentelor sunt reprezentate prin delegari.

Cele mai frecvente exemple de procesare a evenimentelor sunt intalnite atunci o fereastra de dialog este afisata pe ecran si utilizatorul poate realiza diverse actiuni: *click pe un buton, selectare a unui meniu, tastare a unui text* etc. Atunci cand utilizatorul realizeaza una dintre aceste actiuni, se produce un eveniment. Rutinele de tratare a evenimentelor reactioneaza in functie de evenimentul produs.

Evenimentele sunt entitati membre ale unei clase si se declara utilizand cuvantul cheie event. Forma generala a declaratiei este:

public event delegare-eveniment nume-eveniment;

unde delegare-eveniment reprezinta *numele delegarii utilizate pentru tratarea evenimentului*, iar nume-eveniment *este numele instantei eveniment create*.

Pentru crearea unui eveniment sunt necesare parcurgerea urmatoarelor etape:

(a) setarea delegarii corespunzatoare evenimentului (*the delegate*);

(b) crearea unei clase pentru a transmite argumente (parametrii) rutinei de tratare a evenimentului;

(c) declararea codului corespunzator evenimentului (*the event*);

(d) crearea rutinei de tratare a evenimentului (codul care este executat ca raspuns pentru eveniment, *the handler);*

(e) lansarea evenimentului (*firing the event*).

In exemplul urmator sunt prezentate aceste etape, cu exceptia etapei (b) care nu este necesata deoarece rutina care trateaza evenimentul nu are parametrii. Observati ca pe langa aceste etape, sunt necesare si o serie de alte actiuni care sunt explicate in textul programului.

Exemplul 9. Evenimente

```
using System;
public delegate void MyEventHandlerDelegate(); //(a)
class MyEvent
{
    public event MyEventHandlerDelegate activare; //(c)

    public void Fire()          //Crearea unei metode care genereaza evenimentul
    {
            activare();
    }
}
class EventDemo
{
    static void HandlerEv()        //(d)
    {
        Console.WriteLine("Evenimentul s-a produs");
    }
    public static void Main()
    {
        MyEvent ev = new MyEvent();  //Crearea instantei eveniment
        ev.activare += new MyEventHandlerDelegate(HandlerEv);  //Adaugarea rutinei de tratare in lant

        ev.Fire();                  //(e)
    }
}
```

Rezultat:

Evenimentul s-a produs

Exemplul 10. Evenimentele pot fi multicast. Aceasta permite ca mai multe obiecte sa poata raspunde la instiintarea aparitiei unui eveniment. In exemplul de mai jos este prezentat un exemplu de eveniment multicast.

```
using System;
public delegate void MyEventHandlerDelegate();
class MyEvent
{   public event MyEventHandlerDelegate activare;
    public void Fire()
    {       if (activare != null)  activare();    } }
class A {   public void Ahandler()
    {Console.WriteLine("Eveniment tratat si de metoda Ahandler");}}
class B {   public void Bhandler()
    {Console.WriteLine("Eveniment tratat si de metoda Bhandler");} }
class EventDemo {
    static void HandlerEv()
    {Console.WriteLine("Evenimentul s-a produs");}
    public static void Main()
    {   MyEvent ev = new MyEvent();
        A obA = new A();     B obB = new B();
        ev.activare += new MyEventHandlerDelegate(HandlerEv);
        ev.activare += new MyEventHandlerDelegate(obA.Ahandler); //adaugarea metodei Ahandler la lant
        ev.activare += new MyEventHandlerDelegate(obB.Bhandler); //adaugarea metodei Bhandler la lant
        ev.Fire();
        Console.WriteLine();
              ev.activare -= new MyEventHandlerDelegate(obA.Ahandler); //eliminarea metodei Bhandler din lant
        ev.Fire();      }
}
```

Rezultat:

Evenimentul s-a produs
Eveniment tratat si de metoda Ahandler
Eveniment tratat si de metoda Bhandler

Evenimentul s-a produs
Eveniment tratat si de metoda Bhandler

Observatii:

Evenimentele se utilizeaza, de regula, la crearea aplicatiilor Windows. Pentru a usura munca utilizatorului, C# permite utilizarea unor clase care pun la dispozitie evenimente si delegari standard. Astfel:

1) spatiul de nume System contine o delegare standard, intitulata EventHandler [mai precis forma sa generala este public delegate void EventHandler(object sender, EventsArgs e)]. Aceasta primeste doi parametrii. Primul parametru, object sender, contine sursa (obiectul) care genereaza evenimentul iar cel de-al doilea argument este un obiect dintr-o clasa standard, intitulata EventArgs din spatiul de nume System. Aceasta clasa este utilizata pentru a transmite argumente (parametrii) rutinei de tratare a evenimentului (clasa amintita la punctul (b) din al doilea slide referitor la evenimente);

2) intrucat EventHandler are doi parametrii, urmeaza ca metodele care trateaza evenimentul contin aceiasi parametri ca si delegarea;

3) spatiul de nume System.Windows.Forms pune la dispozitie un numar mare de evenimente standard. Acestea sunt asociate diverselor controlere (button, label, radio button, etc.). In Visual C#, lista evenimentelor se gaseste in meniul Properties.

Exemplul urmator ilustreaza utilizarea evenimentelor si rutinelor de tratare ale acestora in cazul unei aplicatii windows.

```
using System;
using System.Windows.Forms;
using System.Drawing;

public class MyForm : Form
{
private Label myDateLabel;
private Button btnUpdate;

public MyForm()
{ InitializeComponent();}

//INSEREAZA METODA InitializeComponent() AICI

protected void btnUpdate_Click( object sender, System.EventArgs e)
{ DateTime currentDate =DateTime.Now ;
this.myDateLabel.Text = currentDate.ToString(); }

protected void btnUpdate_MouseEnter(object sender, System.EventArgs e)
{ this.BackColor = Color.HotPink;}

protected void btnUpdate_MouseLeave(object sender, System.EventArgs e)
{ this.BackColor = Color.Blue;}

protected void myDataLabel_MouseEnter(object sender, System.EventArgs e)
{ this.BackColor = Color.Yellow;}

protected void myDataLabel_MouseLeave(object sender, System.EventArgs e)
{ this.BackColor = Color.Green; }

public static void Main( string[] args )
{ Application.Run( new MyForm() );   /* creaza fereastra*/      }
}
```

```
private void InitializeComponent()
 { this.Text = Environment.CommandLine;
 this.StartPosition = FormStartPosition.CenterScreen;
 this.FormBorderStyle = FormBorderStyle.Fixed3D;

 myDateLabel = new Label(); // Creaza label

 DateTime currentDate = new DateTime();
 currentDate = DateTime.Now;
 myDateLabel.Text = currentDate.ToString();

 myDateLabel.AutoSize = true;
 myDateLabel.Location = new Point( 50, 20);
 myDateLabel.BackColor = this.BackColor;

 this.Controls.Add(myDateLabel); // Adauga label-ul ferestrei

 this.Width = (myDateLabel.PreferredWidth + 100); // Seteaza latimea ferestrei pe baza latimii labelui

 btnUpdate = new Button(); // Creaza button

 btnUpdate.Text = "Update";
 btnUpdate.BackColor = Color.LightGray;
 btnUpdate.Location = new Point(((this.Width/2) -
(btnUpdate.Width / 2)), (this.Height - 75));

 this.Controls.Add(btnUpdate); // Adauga button-ul ferestrei

 btnUpdate.Click += new System.EventHandler(this.btnUpdate_Click);
 btnUpdate.MouseEnter += new System.EventHandler(this.btnUpdate_MouseEnter);
 btnUpdate.MouseLeave += new System.EventHandler(this.btnUpdate_MouseLeave);
 myDateLabel.MouseEnter += new System.EventHandler(this.myDataLabel_MouseEnter);
 myDateLabel.MouseLeave += new System.EventHandler(this.myDataLabel_MouseLeave);
 }
```

Spatii de nume

Spatii de nume

Un spatiu de nume defineste un domeniu de valabilitate, separand astfel un set de clase, variabile, metode, etc. de un alt set.

Spatiul de nume pe care l-am utilizat frecvent pana in momentul de fata este System. Din acest motiv, progamele utilizate pana in prezent au inclus linia using System; Arhitectura .NET pune la dispozitie o serie larga de spatii de nume (spre exemplu System.IO, System.Windows.Forms; etc.)

Spatiile de nume sunt importante deoarece in aplicatii sunt utilizate o gama mare de variabile, metode, proprietati, clase avand diverse nume. Acestea se refera la functiile de biblioteca, la codul dezvoltat de terti sau la codul dezvoltat de utilizator. Fara spatiile de nume, aceste nume de variabile, metode, etc. ar putea intra in conflict daca sunt aceleasi.

Spatiile de nume se declara utilizand cuvantul cheie namespace. Forma generala a declaratiei namespace este:

```
namespace nume
{//membrii}
```

unde nume reprezinta numele spatiului.

In cadrul unui spatiu de nume se pot declara *clase, structuri, delegari, enumerari, interfete sau chiar un alt spatiu de nume.*

Exemplul alaturat demonstreaza utilizarea spatiului de nume. Deoarece clasa Point este declarata in cadrul spatiului de nume Puncte, pentru crearea unei instante a clasei Point trebuie utilizat spatiul de nume si operatorul punct (.) Dupa ce s-a creat obiectul de tipul Point, nu mai este necesar sa calificam obiectul sau oricare din membrii sai cu numele spatiului.

Exemplul 12. Spatii de nume

```
using System;
namespace Puncte
{
class Point
{
   public double x;
   public double y;
   public Point(double xx, double yy)
   {
      x = xx; y = yy;
   }

}
}
class Segmdr
{
   public static void Main()
   {
      Puncte.Point punct1 = new Puncte.Point(3,4);
      Puncte.Point punct2 = new Puncte.Point(5,6);
      double dist;

      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 intre punctele (3,4) si (5,6) este: 2.83

Daca programul utilizatorului include referinte frecvente catre membrii unui spatiu de nume, specificarea spatiului de nume ori de cate ori trebuie sa referiti un membru al sau devine greoaie. Directiva using rezolva aceasta problema. Directiva using are doua forme: using nume; si respectiv using alias=nume; Referitor la cea de-a doua forma, alias devine un alt nume pentru clasa sau spatiul de nume specificat prin nume. Ambele forme ale directivei using sunt specificate in exemplele de mai jos, pentru care rezultatele sunt acelasi ca in exemplul 12.

Exemplul 13

```
using System;
using Puncte;
namespace Puncte
{
   class Point
   {
      public double x;
      public double y;
      public Point(double xx, double yy)
      { x = xx; y = yy; }
   }
}
class Segmdr
{
   public static void Main()
   {
      Point punct1 = new Point(3, 4); Point punct2 = new
      Point(5, 6);
      double dist;

      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);
   }
}
```

Exemplul 14

```
using System;
using p=Puncte.Point;
namespace Puncte
{
   class Point
   {
      public double x;
      public double y;
      public Point(double xx, double yy)
      { x = xx; y = yy; }
   }
}
class Segmdr
{
   public static void Main()
   {
      p punct1 = new p(3, 4); p punct2 = new p(5, 6);
      double dist;

      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);
   }
}
```

Spatiile de nume sunt aditive. Astfel pot exista declaratii namespace cu acelasi nume. Aceasta permite distribuirea unui spatiu de nume in mai multe fisiere sau chiar separarea sa in cadrul aceluiasi fisier. In exemplul de mai jos sunt definite doua spatii de nume cu aceeasi denumire. La compilare, continutul ambelor spatii de nume este adaugat laolalta.

Exemplul 15

```
using System;
namespace Puncte
{    class Point
     {        public double x;        public double y;    }
}
namespace Puncte
{
     class Line
     {
          public Point punct1 = new Point();        public Point punct2 = new Point();
          public double Lung()
          {    double l;
               l = Math.Sqrt((punct1.x - punct2.x) * (punct1.x - punct2.x) + (punct1.y - punct2.y) * (punct1.y - punct2.y));
               return l;
          }
     }
}
class Segmdr
{
     public static void Main()
     {
          Puncte.Line seg = new Puncte.Line();
          double dist;
          seg.punct1.x = 3;        seg.punct1.y = 4;
          seg.punct2.x = 5;        seg.punct2.y = 3;
          dist = seg.Lung();
          Console.WriteLine("Distanta dintre punctele ({0},{1}) si ({2},{3}) este: {4:#.##}", seg.punct1.x, seg.punct1.y,
          seg.punct2.x, seg.punct2.y, dist);
     }
}
```