Tutorial 3.

Evenimente în C#

În tutorialul precedent, în care am construit o aplicație de tip Windows Forms, am setat cu designerul unele evenimente ale formei și ale butoanelor, alegându-le din tabelul pus la dispoziție de fereastra Properties și apoi am completat șabloanele handlerelor de eveniment scrise de designer. Am vazut că funcționează, butoanele răspund la comenzi, Packman poate fi pornit și oprit din rotație, acum vrem să explicăm codul C# care asigură această funcționalitate.

Incepem prin a fixa un vocabular comun privind organizarea claselor .NET folosite într-o aplicație cu forme Windows.

1. Ferestre, componente, controale. Sistemul de operare Windows se bazează, asa cum este și denumit, pe *ferestre*. Ferestrele sunt obiecte (instanţe ale unor clase C++) dotate cu un identificator numeric pe 32 de biţi (*window handle)* folosit de sistemul de operare drept cheie de intrare în tabelul obiectelor aflate în lucru. Ferestrele au un titlu, un stil, o anumită poziţie pe ecran, au lăţime si înălţime, au o fereastră părinte și, în sfârsit, au o procedură de fereastră (*WindowProc*) pentru tratarea mesajelor.

Pentru gestionarea eficientă a alocării și dealocării resurselor utilizate într-o aplicație Windows, a fost definită *clasa componentelor*: un obiect din această clasă este dotat cu o listă de alte componente, peste care el este proprietar (*Owner*), aceasta însemnă doar că destructorul său apelează mai întâi destructorii obiectelor aflate în proprietate și apoi eliberează memoria ocupată de el.

In programarea Windows, se numesc controale (*Windows Controls*) o serie de componente specializate în interacţiunea aplicaţiei cu utilizatorul. Între controalele utilizate într-o aplicație există o relație de găzduire: fiecare control (mai puțin cel *top-level*) are un părinte (*Parent*) față de care își precizează poziția ocupată.

Formele Windows sunt controale destinate de a fi top-level, fără un control părinte. Ele pot fi minimizate, maximizate, pot fi mutate cu mausul ca niște platforme solide pe care stau fixate controalele copil.

Toată această organizare a claselor Windows a fost păstrată și adaptată corespunzător în .NET, formele având aici următoarea linie ereditară: `Object -> MarshalByRefObject -> Component -> Control-> ScrollableControl -> ContainerControl -> Form`

2. Mesaje. Sistemul de operare Windows comunică cu aplicaţiile pe care le gestionează prin *mesaje*. Mesajele Windows sunt impementate sub forma unor structuri C/C++ dotate cu câmpuri numerice în care sunt codificate informaţii referitoare la mesajul transmis, începând cu numărul de indentificare al evenimentului consemnat, și continuând, de exemplu, cu coordonatele mausului în momentul clicului, modul de transmitere, și așa mai departe. În .NET este definit un

format propriu pentru mesaje, implementat tot ca o structură, când o aplicație C# primește un mesaj Windows acesta este tradus în formatul .NET înainte a fi trimis mai departe.

Aplicațiile de tip Windows Forms Applications sunt gestionate, după cum am văzut, prin intermediul clasei `Application`. Apelul

```
Application.Run(new MainForm());
```

crează un obiect anonim de tip `MainForm` și îl pune în funcțiune. Aceasta însemnă că formei i se atașează o listă de tip coadă de așteptare în care sunt depuse mesajele primite de aplicație de la sistemul de operare, iar aplicația intră într-o buclă de procesare continuă a acestor mesaje, până când apare mesajul de închidere a formei.

Când procesează un mesaj, aplicația determină, dacă este cazul, care este controlul vizat de mesaj și îi cere să acționeze în consecință. De exemplu, dacă sistemul de operare a trimis mesajul "s-a tastat Enter", mesajul ajunge la butonul care deține focusul și acest control trebuie să răspundă prin apelul unei metode proprii.

Apare următoarea problemă: să presupunem că amplasăm pe o formă un buton, altfel spus dotăm clasa formei cu un câmp de tip `System.Windows.Forms.Button`. Proiectantul acestei clase .NET nu avea cum să prevadă răspunsul dorit de noi, iar noi nu avem avem acces la codul clasei butonului, ca să implementăm răspunsul nostru. Forma are un câmp de tip `Button`, dar nu moștenește clasa `Button` ca să putem, eventual, suprascrie o metodă de răspuns. În acest caz se spune doar că `MainForm` este o *clasă utilizator* a clasei `Button`, iar noi, cei care proiectăm clasa `MainForm` suntem *utilizatorii* controlului `Button`.

Soluția clasică, C++, la această problemă este următoarea: clasa Button este prevăzută de proiectant cu un câmp public de tip pointer către funcție, câmp dedicat mesajului "Enter" și pe care noi, în calitate de utilizatori, îl putem seta să ținească o funcție de răspus definită de noi în acest scop, numită funcție *callback*.

Acest mecanism de răspuns la mesajele primite a fost preluat în .NET într-o formă mult mai sigură și mai ușor de folosit: în loc de pointeri către funcții, obiectele din clasa Control au fost dotate cu câmpuri de tip eveniment (*event*), o noțiune introdusă în limbajul C# după modelul din Visual Basic. Printr-un astfel de câmp controlul poate invoca la rulare răspunsul dorit de utilizator.

În C# o clasă poate fi dotată cu câmpuri, metode, proprietăți și evenimente. După cum știm, o proprietate este o pereche de metode *get/set* care se comportă sintactic ca un câmp. În mod analog, un eveniment este format dintr-o pereche de metode *add*/*remove* care se comportă ca un câmp de tip pointer către metodă, mai precis de tip metodă delegată.

3. Delegarea metodelor. Începem cu exemplul următor, în care am implementat o metodă de aproximare a valorii maxime a unei funcții reale definite pe un interval [*a*,*b*], prin discretizarea funcției cu pasul *h*=0.001:

```
using System;
```

```
namespace Delegari
{
    class Exemplul1
    {
        delegate double Functie(double x);

        double F1(double x)
        {
            return x * x;
        }
        double F2(double t)
        {
            return t * t * -t - 1;
        }
        double Maxim(double a, double b, Functie f)
        {
            double max = f(a);
            for (double x = a; x <= b; x += 0.001)
            {
                if (max < f(x)) max = f(x);
            }
            return max;
        }
        public void Print()
        {
            Console.WriteLine(Maxim(1, 2, F1));
            Functie df = F2;
            Console.WriteLine(Maxim(1, 2, df));
            df = new Functie(F2);
            Console.WriteLine(df(10));
            Console.WriteLine(df.Invoke(10));
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Exemplul1 ex1 = new Exemplul1();
            ex1.Print();
        }
    }
}
```

Observăm că metoda `double Maxim(double a, double b, Functie f)` are ca parametru de intrare chiar funcția al cărui maxim îl determină, aceasta se datorează declarației

`delegate double Functie(double x);`

care introduce tipul `Functie` ca tip de metodă delegată care primește un `double` și returnează un `double`.

În C++ `Functie` ar fi tipul "pointer către funcții de la `double` la `double`" declarat astfel

```
typedef double (*Functie)(double);
```

și, după cum se vede la atribuirea `Functie df = F2;` în C# tipul `Functie` se comportă exact la fel, ca un pointer către metode care duc `double` în `double`. Aceasta este doar o facilitate sintactică, compilatorul transformând atribuirea `df = F2` în `df = new Functie(F2)` apelând aici constructorul clasei `Functie`.

Declarația

```
delegate double Functie(double x);
```

este interpretată de compilator ca declarația unei clase

```
class Functie : System.MulticastDelegate
{    }
```

iar apelul `df(10)` este tradus în `df.Invoke(10)`, unde `Invoke()` este o metodă moștenită de la clasa de bază.

Reținem: când definim un tip de metode delegate precizăm numele tipului și semnătura metodelor delegate, tipul definit este de fapt o clasă iar prin instanțele acestei clase putem invoca orice metodă cu semnătura dată.

Clasele de metode delegate au fost introduse mai ales pentru a preciza unui control cum să răspundă unui anumit mesaj și, de regulă, răspunsul constă într-o serie de acțiuni. Din acest motiv, un obiect din clasa `System.MulticastDelegate` este dotat cu o listă de metode delegate, listă care poate fi gestionată cu operatorii aritmetici aditivi.

Exemplu:

```
using System;
namespace Delegari
{
    delegate void Afisare(int n);

    class Exemplul2
    {
        void ScrieA(int n)
        {
            for (int i = 0; i < n; i++)
            {
                Console.Write("A");
            }
        }
        void ScrieB(int n)
        {
            for (int i = 0; i < n; i++)
            {
                Console.Write("B");
            }
        }
```

```
        void ScrieC(int n)
        {
            for (int i = 0; i < n; i++)
            {
                Console.Write("C");
            }
        }

        public void Executa()
        {
            Afisare afisA = ScrieA, afisB = ScrieB;
            Afisare afisare = afisA + afisB;
            afisare += ScrieC;
            afisare(3);
            Console.WriteLine();
            afisare -= ScrieB;
            afisare(5);
            Console.WriteLine();
        }
        //AAABBBCCC
        //AAAAACCCCC
        //Press any key to continue . . .
    }
    class Program
    {
        static void Main(string[] args)
        {
            Exemplul2 ex2 = new Exemplul2();
            ex2.Executa();
        }
    }
}
```

Aceste operații de concatenare a metodelor delegate sunt des utilizate mai ales cu metode cu rezultat de tip void și sunt esențiale în definirea evenimentelor.

4. Evenimente C#. Să rezolvăm acum problema răspunsului unui buton la mesajul "ai fost selectat cu tasta Enter".

Prima încercare:

```
using System;
namespace Delegari
{
    public delegate void RaspusEnter();
    public class MyButton
    {
        public  RaspusEnter Enter;

    }
```

```
    public class MyForm
    {
        public MyButton btnOk = new MyButton();
        void btn_enter()
        {
            Console.WriteLine("sunt utilizat de MyForm\nsi am fost apasat");
        }

        public MyForm()
        {
            btnOk.Enter = btn_enter;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            MyForm forma = new MyForm();
            forma.btnOk.Enter();
        }
    }
}
//sunt utilizat de MyForm
//si am fost apasat
//Press any key to continue . . .
```

Funcționează, dar iată ce se întâmplă dacă în clasa utilizator `MyForm` eliminăm atribuirea `btnOk.Enter = btn_enter;` Obținem o excepția `NullReferenceException` care apare deoarece am încercat să utilizăm o referință neinițializată:

```
using System;
namespace Delegari
{
    public delegate void RaspusEnter();
    public class MyButton
    {
        public  RaspusEnter Enter;

    }

    public class MyForm
    {
        public MyButton btnOk = new MyButton();
        void btn_enter()
        {
            Console.WriteLine("sunt utilizat de MyForm\nsi am fost apasat");
        }
        public MyForm()
        {
            //btnOk.Enter += btn_enter;
        }
    }
```

```
    class Program
    {
        static void Main(string[] args)
        {
            MyForm forma = new MyForm();
            forma.btnOk.Enter();
        }
    }
}

//Unhandled Exception: System.NullReferenceException: Object reference not set to
//an instance of an object.
```

Rezolvarea standard a problemei: răspunsul butonului îl formulăm indirect, prin metoda `OnEnter()` de mai jos

```
using System;
namespace Delegari
{
    public delegate void RaspusEnter();
    public class MyButton
    {
        public  RaspusEnter Enter;
        public void OnEnter()
        {
            if (Enter != null) Enter();
        }

    }
    public class MyForm
    {
        public MyButton btnOk = new MyButton();
        void btn_enter()
        {
            Console.WriteLine("sunt utilizat de MyForm\nsi am fost apasat");
        }

        public MyForm()
        {
            btnOk.Enter = btn_enter;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            MyForm forma = new MyForm();
            forma.btnOk.OnEnter();
        }
```

```
    }
}
//sunt utilizat de MyForm
//si am fost apasat
//Press any key to continue . . .
```

De regulă răspunsul unui control constă din mai multe acțiuni, o parte dintre ele fiind precizate chiar de proiectantul controlului. În acest caz utilizatorul trebuie să utilizeze atribuirea cu += pentru a-și adăuga metoda sa de răspuns:

```
using System;

namespace Delegari
{
    public delegate void RaspusEnter();
    public class MyButton
    {
        public RaspusEnter Enter;
        public void OnEnter()
        {
            if (Enter != null) Enter();
        }
        void antet()
        {
            Console.WriteLine("Eu, un buton");
        }
        public MyButton()
        {
            Enter = antet;
        }
    }

    public class MyForm
    {
        public MyButton btnOk = new MyButton();
        void btn_enter()
        {
            Console.WriteLine("sunt utilizat de MyForm\nsi am fost apasat");
        }

        public MyForm()
        {
            //btnOk.Enter = btn_enter;
            btnOk.Enter += btn_enter;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
```

```
            MyForm forma = new MyForm();
            forma.btnOk.OnEnter();
        }
    }
}
//Eu, un buton
//sunt utilizat de MyForm
//si am fost apasat
//Press any key to continue . . .
```

Apare următoarea problemă: deoarece o atribuirea directă șterge răspunsul pre-format al butonului, cum poate fi obligat utilzatorul clasei `MyButton` să folosească numai atribuirile compuse `+=` și `-=` pentru a-și preciza contribuția?

Raspunsul e simplu și constă într-un sigur cuvânt cheie, cuvântul `event`:

```
using System;
namespace Delegari
{
    public delegate void RaspusEnter();
    public class MyButton
    {
        public event RaspusEnter Enter;
        //public RaspusEnter Enter;
        public void OnEnter()
        {
            if (Enter != null) Enter();
        }
        void antet()
        {
            Console.WriteLine("Eu, un buton");
        }
        public MyButton()
        {
            Enter = antet;
        }
    }

    public class MyForm
    {
        public MyButton btnOk = new MyButton();
        void btn_enter()
        {
            Console.WriteLine("sunt utilizat de MyForm\nsi am fost apasat");
        }

        public MyForm()
        {
            //btnOk.Enter = btn_enter;
            btnOk.Enter += btn_enter;
        }
    }
```

```
    class Program
    {
        static void Main(string[] args)
        {
            MyForm forma = new MyForm();
            forma.btnOk.OnEnter();
        }
    }
}
//Eu, un buton
//sunt utilizat de MyForm
//si am fost apasat
//Press any key to continue . . .
```

Mai sus, în clasa `MyButton`, am transformat câmpul `Enter` de tip `RaspusEnter` în *evenimentul* `Enter`, tot de tip `RaspusEnter`. Dacă scoatem acum din comentariu atribuirea

```
//btnOk.Enter = btn_enter;
```

avem următoarea eroare la compilare:

```
The event 'MyButton.Enter' can only appear on the left hand side of += or -=
(except when used from within the type 'MyButton')
```

Declarația

```
public event RaspusEnter Enter;
```

spune că `Enter` este un eveniment care trebuie rezolvat cu o metodă cu semnătura dată de tipul `RaspusEnter`. Metoda invocată prin intermediul lui `Enter` se numește *handler*-ul evenimentului.

Declarația de mai sus este doar forma prescurtată a declarației explicite

```
private RaspusEnter enter;
public event RaspusEnter Enter
{
     add
     {
         enter += value;
     }
     remove
     {
         enter -= value;
     }
}
```

care apare în următorul program:

```
using System;
namespace Delegari
{
    public delegate void RaspusEnter();
```

```
    public class MyButton
    {
        private RaspusEnter enter;
        public event RaspusEnter Enter
        {
            add
            {
                enter += value;
            }
            remove
            {
                enter -= value;
            }
        }

        public void OnEnter()
        {
            if (enter != null) enter();
        }
        void antet()
        {
            Console.WriteLine("Eu, un buton");
        }
        public MyButton()
        {
            enter = antet;
        }
    }
    public class MyForm
    {
        public MyButton btnOk = new MyButton();
        void btn_enter()
        {
            Console.WriteLine("sunt utilizat de MyForm\nsi am fost apasat");
        }

        public MyForm()
        {
            btnOk.Enter += btn_enter;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            MyForm forma = new MyForm();
            forma.btnOk.OnEnter();
        }
    }
}
```

Forma explicită este utilă, de exemplu, dacă dorim ca utilizatorul butonului nostru să nu poată scoate nimic din lista de acțiuni:

```
private RaspusEnter enter;
public event RaspusEnter Enter
{
    add
    {
        enter += value;
    }
    remove
    {
        //enter -= value;
    }
}
```

5. Evenimente în .NET. Clasa `System.Windows.Forms.Control` a fost prevăzută cu o serie întreagă de evenimente, cum ar fi `Click`, `Enter`, `GotFocus`, etc[1], iar clasele derivate pot adăuga altele proprii. Fiecare eveniment este prevăzut cu metoda corespunzătoare de răspus `OnClick`, `OnEnter`, `OnGotFocus`, etc.

Toate evenimentele din biblioteca .NET au tipul

```
public delegate void EventHandler(object sender, EventArgs e);
```

sau un tip derivat din acesta. Parametrul de tip `EventArgs` poartă informații extrase din mesajul primit de la sistemul de operare iar parametrul `sender` (expeditorul) este de regulă chiar controlul vizat de mesaj.

Următorul exemplu de cod în clasa unui control este tipic

```
public event EventHandler Enter;
protected virtual void OnEnter(object sender, EventArgs e)
{
    EventHandler handler = Enter;
    if (handler != null)
    {
        handler(this, e);
    }
}
```

Deoarece metoda de răspuns `OnEnter()` este cea care apelează handler-ul evenimentului `Enter`, se spune că ea declanșează evenimentului `Enter`, sau că provoacă apariția (*raise*) lui. Controlul trimite către clasa utilizator informații despre evenimentul declanșat și primește de la aceasta metoda de gestionare a evenimentului, handlerul de eveniment.

1 vezi https://docs.microsoft.com/en-us/dotnet/api/system.windows.forms.control?view=windowsdesktop-6.0

Când implementează un handler de eveniment, utilizatorul controlului poate să țină cont sau nu de faptul că, la rulare, `sender`-ul va fi controlul care a produs evenimentul.

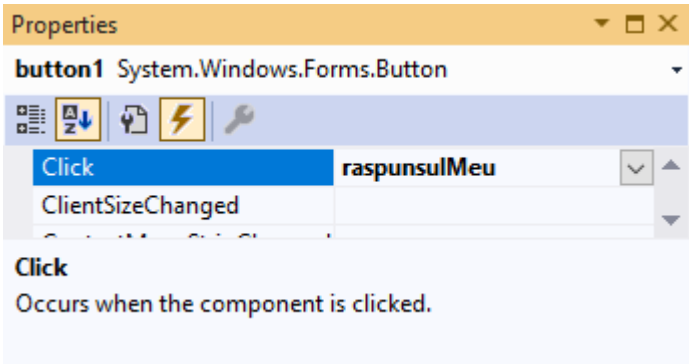
Dăm în continuare exemplul tipic de utilizare a parametrului `sender`.

Incepem un proiect `Windows Forms App.` și, în tab-ul designer, tragem din Toolbox pe șablonul formei `Form1` butoanele `button1`, `button2`, `button3` și eticheta `label1`. Atât.

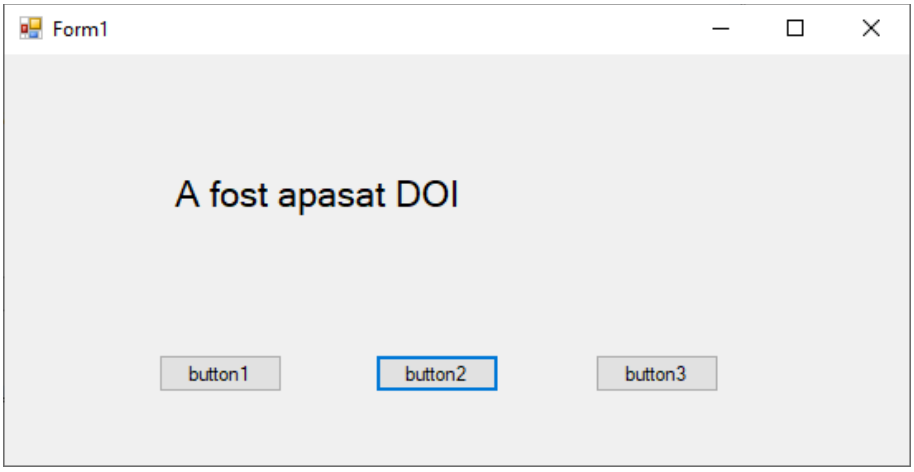
Înainte de a folosi ferestra Propierties pentru a seta evenimentele, scriem în `Form1.cs` co-dul metodei `raspunsulMeu()`:

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

namespace CsharpEvents
{
    public partial class Form1 : Form
    {

        public Form1()
        {
            InitializeComponent();
        }

        void raspunsulMeu(object sender, EventArgs e)
        {
            if (((Button)sender).Name == "button1") label1.Text= "A fost apasat UNU";
            else if (((Button)sender).Name == "button2") label1.Text = "A fost apasat
DOI";
            else  label1.Text = "A fost apasat TREI";
        }
    }
}
```

Ne întoarcem la design și din fereastra Properties, pentru fiecare buton, selectăm pentru evenimetul `Click` același handler de eveniment: `raspunsulMeu`



Verificăm dacă în `Form1.Designer.cs` găsim în corpul metodei `InitializeComponent()` instrucțiunile

```
this.button1.Click += new System.EventHandler(this.raspunsulMeu);
this.button2.Click += new System.EventHandler(this.raspunsulMeu);
this.button3.Click += new System.EventHandler(this.raspunsulMeu);
```

și dacă da, la rulare trebuie să obținem: