

Windows Presentation Foundation – WPF

Cuprins

Windows Presentation Foundation – WPF	1
Cuprins	1
Bibliografie:	3
Introducere in modelul de programare XAML	4
Ordinea de procesare a proprietatii si evenimentului	5
Spatii de nume	5
Elemente proprietate	6
Converteri de tip	10
Extensii pentru marcare - Markup Extensions	11
Descendenti ai elementelor object	13
Proprietatea Content	14
Colectie de articole	14
Valoare ce poate fi convertita la elementul obiect	16
Spatii de nume proprii – declaratie	16
WPF – Fundamente	19
Arbori logici si arbori vizuali	19
Proprietati dependente. Proprietati atasate.	21
Proprietati dependente	22
Clasa DependencyProperty	22
Exemplu complet cu proprietate dependenta	31
Proprietati atasate	36
Proprietati atasate in cod	37
Metadata proprietatii atasate	38
Crearea unei proprietati atasate	38
Atributele proprietatii atasate	39
Construirea unei aplicatii WPF	42
Controlul marimii controalelor	42
Controlarea pozitiei	44
Aliniere continut	45
FlowDirection	46
Layout WPF	47
Structura unei aplicatii Window standard	55
Clasa Window	55
Clasa Application	56

Controale	58
Controale ce contin un singur articol	59
Controale ce contin o colectie de articole	60
Clasa EventManager	64
Handler-i ai clasei si handler-i ai instantei	66
Handler-i virtuali	67
Strategii de rutare si metode pentru evenimente	67
Evenimente in XAML si cod	68
Evenimente atasate	71
Evenimente de la tastatura.....	72
Comenzi	73
ComboBox	80
ComboBoxItem	81
ListBox	81
Listview	82
DataGrid.....	83
Coloane auto generate	84
Selectare randuri si/sau celule	85
Meniuri	88
ContextMenu	89
TreeView	89
TreeViewItem.....	89
Accesare resurse binare	91
Accesare resurse din cod	91
Resurse Statice versus Resurse Dinamice	92
Resurse fara partajare	93
Data Binding	94
Concepte de baza in asocierea de date - Data Binding	94
Proprietatea Mode din Binding	Error! Bookmark not defined.
Actualizarea sursei - UpdateSourceTrigger	96
Creare binding	97
DataTemplate	113

Bibliografie:

- **Adam Nathan – WPF 4 Unleashed**
- **MSDN**

WPF – Windows Presentation Foundation

Windows Presentation Foundation (WPF) este compus din o multime de assembly pentru a crea aplicatii GUI.

Un principal avantaj al acestui model este separarea completa dintre designeri si dezvoltatori. Designerii folosesc un limbaj numit *eXtensible Application Markup Language* - (XAML). Spatiul de nume principal este **System.Windows**.

Observatie

Spatiul de nume **System.Windows.Forms** este folosit pentru dezvoltarea aplicatiilor Windows clasice.

Introducere in modelul de programare XAML

Specificatia XAML defineste reguli ce mapeaza spatiile de nume din .NET, tipuri, proprietati si evenimente in spatii de nume XML, elemente si attribute.

XAML este bazat pe XML. Exista o mapare a tipurilor din CLR la tag-urile din XML, de la attributele din XML la proprietatile si evenimentele din CLR.

// XAML:

```
<MyObject SomeProperty='1' />
```

Observatie

MyObject este **element** in descrierea XML.

SomeProperty este **atribut** in descrierea XML de mai sus.

// C#

```
MyObject obj = new MyObject();  
obj.SomeProperty = 1;
```

Observatie

MyObject este un **tip** in .NET.

SomeProperty este o **proprietate** in sensul C# .NET.

// XAML:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
        Content="OK"/>
```

// C#:

```
System.Windows.Controls.Button b = new System.Windows.Controls.Button();  
b.Content = "OK";
```

Tag-urile XML sunt definite in contextul unui namespace si acel namespace determina ce tag-uri sunt valide. In XAML se mapaeza namespaces-uri XML la colectii de assemblies si namespaces-uri CLR.

Declararea unui *element* XML in XAML – cunoscut ca *element obiect* (object element) – este echivalent cu a instantia obiectul corespunzator din .NET folosind constructorul implicit.

Setarea unui **atribut** (in XML) pe un element obiect este echivalent cu setarea proprietatii cu acelasi nume sau atribuirea unui handler pentru un eveniment cu acelasi nume (*event attribute*).

Exemplu. Un buton are setata proprietatea **Content** si ataseaza o metoda (event handler) pentru evenimentul **Click**:

```
// XAML:
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        Content="OK" Click="button_Click"/>

// C#:
System.Windows.Controls.Button b = new System.Windows.Controls.Button();
b.Click += new System.Windows.RoutedEventHandler(button_Click);
b.Content = "OK";
```

Ordinea de procesare a proprietatii si evenimentului

La runtime, handler-ii pentru evenimente sunt atasati inaintea oricarei proprietati pentru orice obiect declarat in XAML – exceptie proprietatea **Name** care este setata imediat dupa constructia obiectului. Acest lucru face posibila tratarea unui eveniment ca raspuns la setarea unei proprietati, si ca o consecinta a acestui fapt rezulta ca nu are importanta ordinea atributelor folosite in XAML.

Spatii de nume

Elementul obiect radacina dintr-un fisier XAML trebuie sa specifice cel putin un spatiu de nume XML, spatiu de nume ce este folosit de elementul radacina precum si de toate elementele descendente.

Se pot declara spatii de nume XML aditionale, dar fiecare trebuie sa aiba un prefix distinct, prefix ce va fi utilizat pentru a identifica elemente din acel spatiu de nume.

De exemplu, WPF XAML foloseste un al doilea spatiu de nume cu prefixul x astfel :

```
xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml/presentation
```

Urmatoarele spatii de nume din .NET sunt mapate la

```
http://schemas.microsoft.com/winfx/2006/xaml/presentation
```

- . System.Windows
- . System.Windows.Automation
- . System.Windows.Controls
- . System.Windows.Controls.Primitives
- . System.Windows.Data
- . System.Windows.Documents
- . System.Windows.Forms.Integration
- . System.Windows.Ink
- . System.Windows.Input
- . System.Windows.Media
- . System.Windows.Media.Animation
- . System.Windows.Media.Effects
- . System.Windows.Media.Imaging

- . System.Windows.Media.Media3D
- . System.Windows.Media.TextFormatting
- . System.Windows.Navigation
- . System.Windows.Shapes
- . System.Windows.Shell

În majoritatea documentațiilor spațiul de nume WPF XML :

<http://schemas.microsoft.com/winfx/2006/xaml/presentation> este declarat ca spațiu primar în timp ce spațiul de nume al limbajului XAML

<http://schemas.microsoft.com/winfx/2006/xaml>

este declarat ca spațiu secundar, prefixat cu *x*.

În XML se folosește atributul **xmlns** pentru a defini noi namespace-uri.

// XAML :

```
<MyObject xmlns='clr-namespace:Samples' SomeProperty='1' />
```

// C#

```
using Samples;
MyObject obj = new MyObject();
obj.SomeProperty = 1;
```

În XAML, putem specifica locația unui assembly pentru fiecare namespace:

// XAML:

```
<MyObject
  xmlns='clr-namespace:Samples;assembly=samples.dll'
  SomeProperty='1' />
```

// C#

```
csc /r:samples.dll test.cs // compilator apelat din linia de
                          // comanda
```

În mediul vizual se adaugă referința la *samples.dll* urmat de folosirea lui **using**.

```
using Samples;
MyObject obj = new MyObject();
obj.SomeProperty = 1;
```

Elemente proprietate

XML este împărțit în două spații: *elemente* și *atribute*. În termeni de *obiecte*, *proprietăți* și *evenimente*, modelul XAML este mai apropiat de CLR. Exemplul de mai sus poate fi rescris astfel folosind un element descendent :

```
<MyObject
  xmlns='clr-namespace:Samples;assembly=samples.dll'>
  <MyObject.SomeProperty>
    1
  </MyObject.SomeProperty>
</MyObject>
```

Fiecare element *proprietate* este calificat cu tipul ce definește acea proprietate. În exemplul de mai sus tipul este **MyObject**, iar proprietatea este **SomeProperty**.

Exemplu

Presupunem că există o proprietate *Owner* (**MyObject** are această proprietate) pentru obiectul *Persoana*. Obiectul **Persoana** are proprietățile **FirstName** și **LastName**. Codul poate arăta astfel folosind sintaxa XAML:

```
<MyObject
  xmlns='clr-namespace:Samples;assembly=samples.dll'>
  <MyObject.Owner>
    <Persoana FirstName="Chris" LastName="Anderson" />
  </MyObject.Owner>
</MyObject>
```

echivalent în C# cu:

```
MyObject mo = new MyObject();
Persoana persoana = new Persoana();
persoana.FirstName = "Chris";
persoana.LastName = "Anderson";
mo.Owner = persoana;
```

În exemplul de mai sus dacă adnotăm proprietatea **Owner** a obiectului **MyObject** cu atributul **System.Windows.Markup.ContentPropertyAttribute** atunci putem scrie;

```
<MyObject
  xmlns='clr-namespace:Samples;assembly=samples.dll'>
  <Person FirstName='Megan' LastName='Anderson' />
</MyObject>
```

Exemplu complet. Tipul *MyObject* trebuie definit astfel:

```
using System;
using System.Windows;
using System.Windows.Markup;

namespace ContentPropertySamples
{
    [ContentProperty("Owner")]
    public class MyObject:UIElement
    {
        public MyObject() { }
        private Persoana _owner;
        public Persoana Owner
        {
            get { return _owner; }
            set { _owner = value; }
        }
    }
}
```

Tipul *Persoana* este definit astfel:

```
using System;
using System.Windows;
using System.Windows.Markup;
```

```
namespace ContentPropertySamples
{
    public class Persoana:UIElement
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }
}
```

iar in XAML

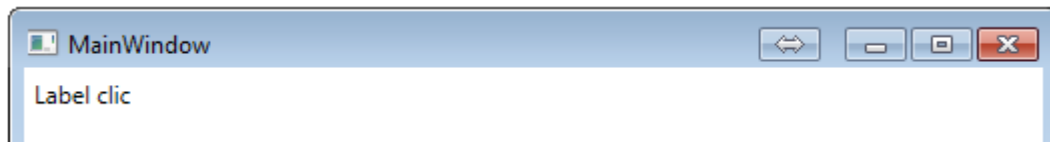
```
<Window x:Class="ContentPropertySamples.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:lc='clr-namespace:ContentPropertySamples'
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <StackPanel>
            <lc:MyObject x:Name="mo">
                <lc:MyObject.Owner>
                    <lc:Persoana x:Name="persoana" FirstName='Megan' LastName='Anderson' />
                </lc:MyObject.Owner>
            </lc:MyObject>
            <Label Name="label1" Content="Label clic"
                PreviewMouseDown="label1_PreviewMouseDown">
            </Label>
        </StackPanel>
    </Grid>
</Window>
```

si in *code behind* tratare eveniment **PreviewMouseDown** (am omis spatiile de nume):

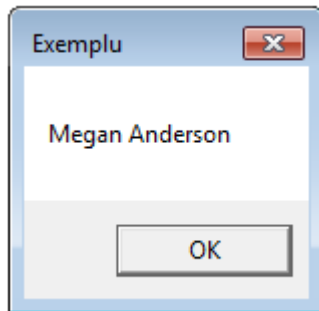
```
namespace ContentPropertySamples
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void label1_PreviewMouseDown(object sender, MouseButtonEventArgs e)
        {
            MessageBox.Show(mo.Owner.FirstName + " " + mo.Owner.LastName, "Exemplu");
        }
    }
}
```


La clic pe *label*



se va afisa:



Exemplu

In C#:

```
System.Windows.Controls.Button b = new System.Windows.Controls.Button();  
System.Windows.Shapes.Rectangle r = new System.Windows.Shapes.Rectangle();  
r.Width = 40;  
r.Height = 40;  
r.Fill = System.Windows.Media.Brushes.Black;  
b.Content = r; // Continutul butonului este un patrat
```

In XAML acest lucru poate fi scris astfel:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">  
  <Button.Content>  
    <Rectangle Height="40" Width="40" Fill="Black"/>  
  </Button.Content>  
</Button>
```

S-a folosit proprietatea **Content** pentru **Button**.

Se foloseste un element XML in locul unui atribut.

Folosind un atribut XML (**Content** este pe post de atribut, la fel **Background**) putem scrie :

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
  Content="OK" Background="White"/>
```

sau cu element proprietate :

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">  
  <Button.Content> <!-- sintaxa cu element proprietate -->  
    OK  
  </Button.Content>  
<Button.Background>
```

```
        White
    </Button.Background>
</Button>
```

Continutul unui control poate fi orice si-l putem stabili in momentul instantierii controlului.

Converteri de tip

Pentru exemplul anterior varianta C# este :

```
System.Windows.Controls.Button b = new System.Windows.Controls.Button();
b.Content = "OK";
b.Background = System.Windows.Media.Brushes.White;
```

Observam proprietatea **White** in C# si stringul **"White"** in XAML.

Un “*converter*” face conversia unui tip la alt tip (dupa cum sugereaza si numele).

Fara un “*converter*” pentru **Brush**, ar fi trebuit sa folosim sintaxa pentru elementul proprietate astfel :

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        Content="OK">
    <Button.Background>
        // aici e un converter pentru "White"
        <SolidColorBrush Color="White"/>
    </Button.Background>
</Button>
```

Daca nu exista un converter in acest caz, am putea scrie:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        Content="OK">
    <Button.Background>
        <SolidColorBrush>
            <SolidColorBrush.Color>
                <Color A="255" R="255" G="255" B="255"/>
            </SolidColorBrush.Color>
        </SolidColorBrush>
    </Button.Background>
</Button>
```

Si aici avem de a face cu un converter. "255" va fi convertit intr-un byte.

Observatie

Pentru ca un tip sau o proprietate sa accepte un converter, acesta/aceasta trebuie adnotata cu atributul **TypeConverter**.

```
[TypeConverter(typeof(BrushConverter)), ...]
public abstract class Brush : ...
{ ... }
```

sau

```
[TypeConverter(typeof(FontSizeConverter)), ...]  
public double FontSize  
{  
    get { ... }  
    set { ... }  
}
```

Extensii pentru marcare - Markup Extensions

Extensiile markup in XAML – incluse intre **{ }** - constituie o modalitate de a extinde parser-ul markup pentru a produce markup-uri mai simple. Extensiile markup sunt implementate ca tipuri CLR si lucreaza asemanator ca definitiile atributelor CLR.

Oricand valoarea unui atribut este inclusa intre {}, XAML trateaza acest lucru ca o extensie si nu ca o expresie.

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
        Background="{x:Null}"  
        Height="{x:Static SystemParameters.IconHeight}"  
        Content="{Binding Path=Height, RelativeSource={RelativeSource Self}}"  
/>
```

Ce e cu rosu reperzinta extensii, in rest parametri cu nume si valori.

Primul identificator dupa { este numele clasei extensie, clasa ce trebuie sa fie derivata din clasa **MarkupExtension**. Prin conventie numele acestor clase se termina cu *Extension*, dar care poate fi omis in XAML.

Clase :

NullExtension = Null ; StaticExtension = Static, etc.

Spatiul de nume pentru clasele extensie ale limbajului de marcare este **System.Windows.Markup**, deci prefixul **x** trebuie utilizat pentru a-l localiza.

Parametrii *pozitionali* (**SystemParameters.IconHeight** de ex.) sunt tratati ca argumente string pentru constructorul clasei extensie.

Parametrii cu *nume* (**Path, RelativeSource**) permit setarea unor *proprietati* pe obiectul extensie construit. Valorile acestor proprietati pot fi valori date cu limbajul de marcare extins – se folosesc din nou {} sau valori literale.

StaticExtension permite folosirea proprietatilor statice, campurilor, constantelor si enumerarilor in locul literalilor hard-coded. In acest caz, proprietatea **Height** a butonului este setata la inaltimea curenta a icoanelor din sistemul de operare, expusa de proprietatea statica **IconHeight** din clasa **System.Windows.SystemParameters**.

Observatie

“{} {text}” are ca efect tratarea { si } drept caractere normale care nu fac parte din sintaxa.

Din cauza ca extensiile markup sunt clase cu constructori impliciti, acestea pot fi utilizate cu sintaxa elementului proprietate.

Urmatorul cod este identic cu cel anterior:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Button.Background>
        <x:Null/>
    </Button.Background>
    <Button.Height>
        <x:Static Member="SystemParameters.IconHeight"/>
    </Button.Height>
    <Button.Content>
        <Binding Path="Height">
            <Binding.RelativeSource>
                <RelativeSource Mode="Self"/>
            </Binding.RelativeSource>
        </Binding>
    </Button.Content>
</Button>
```

Exemple

```
<Canvas Name="Parent0">
    <Border Name="Parent1"
        Width="{Binding RelativeSource={RelativeSource Self},
        Path=Parent.ActualWidth}"
        Height="{Binding RelativeSource={RelativeSource Self},
        Path=Parent.ActualHeight}">
        <Canvas Name="Parent2">
            <Border Name="Parent3"
                Width="{Binding RelativeSource={RelativeSource Self},
                Path=Parent.ActualWidth}"
                Height="{Binding RelativeSource={RelativeSource Self},
                Path=Parent.ActualHeight}">
                <Canvas Name="Parent4">
                    <TextBlock FontSize="16"
                        Margin="5" Text="Display the name of the ancestor"/>
                    <TextBlock FontSize="16"
                        Margin="50"
                        Text="{Binding RelativeSource={RelativeSource
                            FindAncestor,
                            AncestorType={x:Type Border},
                            AncestorLevel=2},Path=Name}"
                        Width="200"/>
                </Canvas>
            </Border>
        </Canvas>
    </Border>
</Canvas>
```

Rezultatul este:

Display the name of the ancestor

Parent1

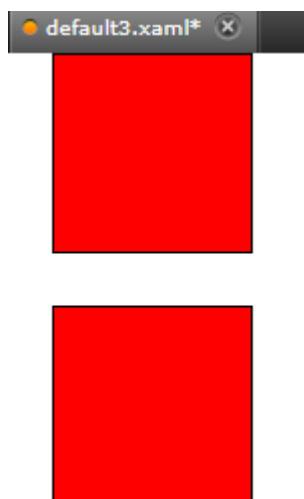
Urmatoarele doua exemple sunt echivalente.

```
<Rectangle Fill="Red" Name="rectangle"
           Height="100" Stroke="Black"
           Canvas.Top="100" Canvas.Left="100"
           Width="{Binding ElementName=rectangle,
                           Path=Height}"/>
```

sau

```
<Rectangle Fill="Red" Height="100"
           Stroke="Black"
           Width="{Binding RelativeSource={RelativeSource Self},
                           Path=Height}"/>
```

Rezultatul este



Descendenti ai elementelor object

Un fisier XAML la fel ca fisierele XML trebuie sa aiba un singur element radacina. Elementele obiect pot suporta elemente obiect descendente (elementele proprietate nu sunt elemente descendente).

Un element obiect poate avea urmatoarele tipuri de descendenti:

- valoare pentru o proprietate continut ;
- colectie de articole ;
- valoare ce poate fi convertita la elementul obiect.

In continuare le discutam pe fiecare in parte.

Proprietatea Content

Multe clase din WPF au o proprietate ce pastreaza continutul elementului. De exemplu continutul unui **Button** poate fi text si/sau imagine si/sau dreptunghi, etc. Aceasta proprietate se numeste **Content** pentru **Button** si poate fi setata in XAML sau in cod.

Pentru alte controale proprietatea are alt nume, dar intra in categoria celor ce furnizeaza continutul controlului. Clasele **ComboBox**, **ListBox**, **TabControl** au proprietatea **Items** pentru a furniza continutul.

Exemplu: **Buton** ce contine un **TextBox**

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
    <Button.Content>
        <TextBox Name="Nume_buton" Text="OK" />
    </Button.Content>
</Button>
```

Buton ce contine un patrat:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
    <Button.Content>
        <Rectangle Height="100" Width="100" Fill="White"/>
    </Button.Content>
</Button>
```

sau echivalent :

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
    <Rectangle Height="40" Width="40" Fill="Black"/>
</Button>
```

Colectie de articole

XAML permite sa adaugam articole la cele doua tipuri de colectii ce suporta indexarea: *liste* si *dictionare*.

Liste

O lista este orice colectie ce implementeaza **System.Collections.List** (de ex **ArrayList**).

Urmatorul cod adauga doua articole la un **ListBox** a carui proprietate **Items** este un **ItemCollection** ce implementeaza **IList**.

```
<ListBox xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
    <ListBox.Items>
        <ListBoxItem Content="Item 1"/>
        <ListBoxItem Content="Item 2"/>
    </ListBox.Items>
</ListBox>
```

Acest lucru e echivalent cu codul C# :

```
System.Windows.Controls.ListBox listBox = new
System.Windows.Controls.ListBox();
System.Windows.Controls.ListBoxItem item1 =
    new System.Windows.Controls.ListBoxItem();
System.Windows.Controls.ListBoxItem item2 =
    new System.Windows.Controls.ListBoxItem();
item1.Content = "Item 1";
item2.Content = "Item 2";
listbox.Items.Add(item1);
listbox.Items.Add(item2);
```

sau in XAML echivalent putem scrie :

```
<ListBox xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
    <ListBoxItem Content="Item 1"/>
    <ListBoxItem Content="Item 2"/>
</ListBox>
```

Dictionare

System.Windows.ResourceDictionary este un tip de colectie folosita de obicei in WPF. Aceasta implementeaza **IDictionary**, deci suporta adaugare, stergere si enumerare de perechi (cheie, valoare) in codul procedural.

In XAML puetm adauga perechi (cheie, valoare) la orice colectie ce implementeaza **IDictionary**.

Exemplu : adauga doua culori la **ResourceDictionary**.

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Color x:Key="1" A="255" R="255" G="255" B="255"/>
    <Color x:Key="2" A="0" R="0" G="0" B="0"/>
</ResourceDictionary>
```

Echivalentul C# este:

```
System.Windows.ResourceDictionary d = new System.Windows.ResourceDictionary();
System.Windows.Media.Color color1 = new System.Windows.Media.Color();
System.Windows.Media.Color color2 = new System.Windows.Media.Color();
color1.A = 255; color1.R = 255; color1.G = 255; color1.B = 255;
color2.A = 0; color2.R = 0; color2.G = 0; color2.B = 0;
d.Add("1", color1);
d.Add("2", color2);
```

Valoare ce poate fi convertita la elementul obiect

Existenta *converterilor* face posibila existenta urmatorului cod.

Exemplu

```
<SolidColorBrush>White</SolidColorBrush>
```

echivalent cu

```
<SolidColorBrush Color="White"/>
```

chiar daca **Color** nu a fost proiectata ca o proprietate pentru continut.

Spatii de nume proprii – declaratie

Sintaxa este urmatoarea:

```
xmlns="clr-namespace:<nume_tip>; assembly=<nume_assembly>"
```

De exemplu, pentru a seta valoarea unei proprietati la **null** putem scrie:

```
<MyObject xmlns='clr-namespace:Samples;assembly=samples.dll'>  
  <Person FirstName='Megan' LastName='{x:Null}' />  
</MyObject>
```


Cuvinte cheie XAML :

XAML Namespace Directive	Meaning	Example
<code>x:Array</code>	Creates a CLR array.	<pre><x:Array Type='{x:Type Button}'> <Button /> <Button /> </x:Array></pre>
<code>x:Class</code>	Specifies the name of the type to define (used only in markup compilation).	<pre><Window x:Class='MyNamespace.MyClass'>... </Window></pre>
<code>x:ClassModifier</code>	Specifies the modifiers ("public," "internal," etc.) of the type to define (used only in markup compilation).	<pre><Window x:Class='...' x:ClassModifier='Public'> ... </Window></pre>
<code>x:Code</code>	Delineates a block of in-line code (used only in markup compilation).	<pre><Window x:Class='...'> <x:Code> public void DoSomething() { ... } </x:Code> ... </Window></pre>
<code>x:Key</code>	Specifies the key to use for an element (supported only on elements contained in a dictionary).	<pre><Button> <Button.Resources> <Style x:Key='Hi'>...</Style> </Button.Resources> </Button></pre>
<code>x:Name</code>	Specifies the programmatic name of an element (typically used when an element doesn't have a built-in name property).	<pre><sys:Int32 xmlns:sys='clr-namespace: System;...' x:Name='_myIntegerValue'> 5</sys:Int32></pre>
<code>x:Null</code>	Creates a null value.	<pre><Button Content='{x:Null}' /></pre>
<code>x:Static</code>	Creates a value by accessing a static field or property from a type.	<pre><Button Command='{x:Static ApplicationCommands.Close}' /></pre>

XAML Namespace Directive	Meaning	Example
<code>x:Subclass</code>	Provides a base type for markup compilation for languages that don't support partial types.	
<code>x:type</code>	Provides a CLR type (equivalent to <code>Type.GetType()</code>).	<code><ControlTemplate TargetType='{x:type Button}'> ... </ControlTemplate></code>
<code>x:typeArguments</code>	Specifies the generic type arguments for instantiating a generic type.	<code><gc:List xmlns:gc='clr-namespace:System.Collections. Generic;...' x:typeArguments='{x:type Button}' /></code>
<code>x:XData</code>	Delineates a block of in-line XML; may be used only for properties of type <code>IXmlSerializable</code> .	<code><XmlDataSource> <x:XData> <Book xmlns='' Title='...' /> </x:XData> </XmlDataSource></code>

Observatie :

Trebuie sa declaram prefixul **x** pentru ca acest tag sa fie parsat. Prefixul **x** este alias pentru namespace.

```
<MyObject xmlns:x='http://schemas.microsoft.com/winfx/2006/xaml'  
  xmlns='clr-namespace:Samples;assembly=samples.dll'  
  <Person FirstName='Megan' LastName='{x:Null}' />  
</MyObject>
```

Putem defini pentru orice assembly CLR (sau multime de assemblies) un nume bazat pe URI.

<!-- option 1: import by CLR namespace -->

```
<Window  
  xmlns:x='http://schemas.microsoft.com/winfx/2006/xaml'  
  xmlns='clr-amespace:System.Windows; assembly=presentationframework.dll'>  
</Window>
```

<!-- option 2: import by URI -->

```
<Window  
  xmlns:x='http://schemas.microsoft.com/winfx/2006/xaml'  
  xmlns='http://schemas.microsoft.com/winfx/2006/xaml/presentation'>  
</Window>
```

Cele doua definitii sunt echivalente.

WPF – Fundamente

Ierarhia de clase (cea mai des folosita) :

Object – clasa de baza pentru toate clasele din .NET.

DispatcherObject – clasa de baza folosita de orice obiect ce doreste sa fie accesat numai din firul care l-a creat.

DependencyObject – clasa de baza pentru orice obiect ce suporta *proprietati dependente*, una din caracteristicile principale ale WPF.

Freezable – clasa de baza pentru obiecte ce pot fi « inghetate » intr-o stare read-only din motive de performanta. Poate fi accesata de fire multiple. Nu-si poate schimba starea dar poate fi clonata. Exemple : primitive grafice – pensoare, penite, clase pentru geometrii si animatii.

Visual – clasa de baza pentru obiecte ce au reprezentare vizuala 2D.

UIElement – clasa de baza pentru toate obiectele vizuale 2D ce suporta *evenimente rutate*, *asociere de comenzi*, *layout* si *focus*.

Visual3D – clasa de baza pentru toate obiectele ce au reprezentare vizuala 3D.

UIElement3D – clasa de baza pentru toate obiectele vizuale 3D ce suporta *evenimente rutate*, *asociere de comenzi*, *focus*.

ContentElement – O clasa de baza similara cu **UIElement** dar pentru parti de document ale continutului ce nu au o redare proprie. **ContentElement** este gazduit intr-o clasa derivata din **Visual** pentru a fi redata pe ecran.

FrameworkElement - clasa de baza ce adauga suport pentru stiluri, data binding, resurse si un mecanism pentru controalele windows cum ar fi *tooltips* si meniul contextual.

FrameworkContentElement – analog cu **FrameworkElement** pentru continut.

Control – clasa de baza pentru controalele obisnuite **Button**, **ListBox** si **StatusBar**. Adauga proprietati precum **Foreground**, **Background** si **FontSize** precum si abilitatea de a fi restilizate.

Arbori logici si arbori vizuali

In WPF, interfata cu utilizatorul este construita dintr-o arborescenta de obiecte cunoscuta sub numele de *arbore logic* (in engleza « logical tree »).

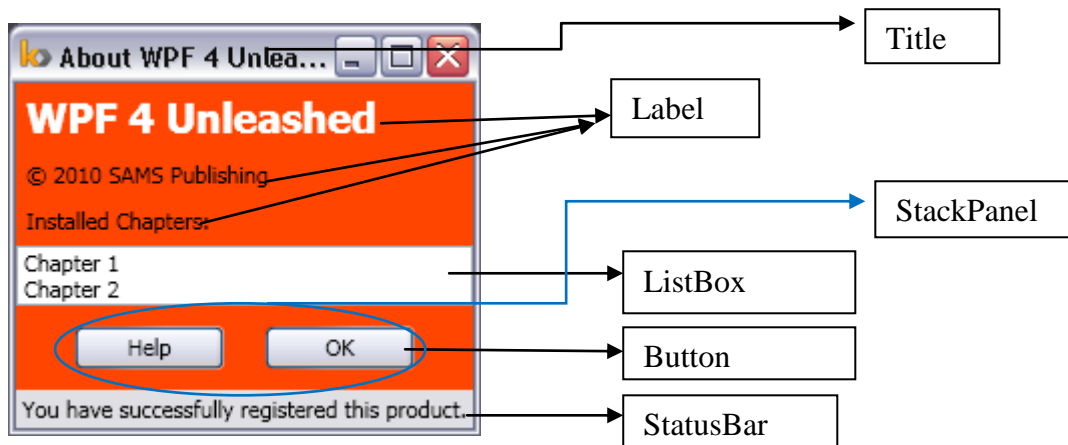
Exemplu (din Adam Nathan - WPF 4 Unleashed).

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Title="About WPF 4 Unleashed" SizeToContent="WidthAndHeight"
  Background="OrangeRed">
  <StackPanel>
    <Label FontWeight="Bold" FontSize="20" Foreground="White">
      WPF 4 Unleashed
    </Label>
    <Label>© 2019 SAMS Publishing</Label>
    <Label>Installed Chapters:</Label>
    <ListBox>
      <ListBoxItem>Chapter 1</ListBoxItem>
      <ListBoxItem>Chapter 2</ListBoxItem>
    </ListBox>
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
      <Button MinWidth="75" Margin="10">Help</Button>
```

```
        <Button MinWidth="75" Margin="10">OK</Button>
    </StackPanel>
    <StatusBar>
        You have successfully registered this product.
    </StatusBar>
</StackPanel>
</Window>
```

Window este radacina.

Contine un **StackPanel** ca element descendent care la randul lui contine ... (urmariti codul).
Codul XAML de mai sus produce urmatoarea fereastră (in Kaxaml) :



Observatie

Arborele logic exista chiar si pentru interfețele ce nu sunt create in XAML.

Multimea elementelor ce sunt redade pe ecran constituie *arborele de vizualizare* (in engleza « visual tree »).

Arborele vizual expune detaliile vizuale.

Button este in *mod logic* un singur control. Redarea lui pe ecran inseamna folosirea mai multor primitive WPF : Border, Background, etc.

Arborele logic si cel de vizualizare pot fi traversati folosind clasele

- **System.Windows.LogicalTreeHelper** si
- **System.Windows.Media.VisualTreeHelper**.

Exemplu

```
using System;
using System.Diagnostics;
using System.Windows;
using System.Windows.Media;
public partial class AboutDialog : Window
{
    public AboutDialog()
    {
        InitializeComponent();
        PrintLogicalTree(0, this);
    }
}
```

```
protected override void OnContentRendered(EventArgs e)
{
    base.OnContentRendered(e);
    PrintVisualTree(0, this);
}
void PrintLogicalTree(int depth, object obj)
{
    // Print the object with preceding spaces
    // that represent its depth
    Debug.WriteLine(new string(' ', depth) + obj);
    // Sometimes leaf nodes aren't DependencyObjects (e.g. strings)
    if (!(obj is DependencyObject)) return;
    // Recursive call for each logical child
    foreach (object child in LogicalTreeHelper.GetChildren(
        obj as DependencyObject))
        PrintLogicalTree(depth + 1, child);
}
void PrintVisualTree(int depth, DependencyObject obj)
{
    // Print the object with preceding spaces
    // that represent its depth
    Debug.WriteLine(new string(' ', depth) + obj);
    // Recursive call for each visual child
    for (int i = 0; i < VisualTreeHelper.GetChildrenCount(obj);
        i++)
        PrintVisualTree(depth + 1, VisualTreeHelper.GetChild(obj, i));
}
}
```

Proprietati dependente. Proprietati atasate.

Clasa **DependencyObject**. Clasa **DependencyProperty**.

Clasa **DependencyObject** permite servicii pe proprietatea sistem. Rolul proprietatii sistem este de a calcula valorile proprietatilor si de a furniza o notificare despre schimbarea valorilor.

O alta clasa importanta este **DependencyProperty** ce permite inregistrarea proprietatilor dependente in proprietatea sistem si identificarea acestora.

Clasa **DependencyObject**, clasa de baza pentru alte clase, permite instantelor claselor derivate din aceasta sa foloseasca proprietati dependente.

Caracteristicile acestei clase - **DependencyObject** - sunt:

- Suport pentru gazduirea proprietatii dependente. Proprietatea dependenta se inregistreaza folosind metoda statica **Register** si memorand valoarea returnata ca un camp "**public static**" in clasa noastra.
- Suport pentru gazduirea proprietatilor atasate, inregistrate cu metoda **RegisterAttached** si memorand valoarea returnata ca un camp **public static readonly** in clasa noastra. Proprietatea atasata poate fi setata pe orice clasa derivata din **DependencyObject**.
- Furnizeaza metode pentru *get*, *set* si *stergere* valori pentru orice proprietate dependenta ce exista pe **DependencyObject**.

- Clasa de baza pentru **ContentElement**, **Freezable** sau **Visual**.

Proprietati dependente

O proprietate dependenta depinde de furnizori multipli pentru a-si determina valoarea sa la orice moment de timp. Acesti furnizori pot fi elemnete ale clasei parinte ale caror valori se propaga catre copii.

Proprietatile dependente sunt asemanatoare cu proprietatile CLR, dar conceptul din spatele lor este mai complex si mai puternic.

Principala diferenta este ca valoarea unei proprietati dependente, **DependencyProperty**, este rezolvata in mod dinamic cand se apeleaza metoda **GetValue()**, metoda mostenita din **DependencyObject**, in timp ce pentru o proprietate .NET normala, valoarea este citita in mod direct din campul privat existent la nivel de clasa.

Clasa DependencyProperty

Reprezinta o proprietate ce poate fi setata prin metode utilizate in stilizare, data binding, animatie si mostenire.

Clasele derivate din **DependencyObject** pot contine proprietati dependente.

// MSDN

Metoda pentru inregistrarea unei proprietati dependente este **Register** (metoda statica) cu urmatoarele prototipuri (3 , 4 si 5 parametri – am ales cea cu 5 parametri) ce returneaza **DependencyProperty** :

```
public static DependencyProperty Register(  
    string name,  
    Type propertyType,  
    Type ownerType,  
    PropertyMetadata typeMetadata,  
    ValidateValueCallback validateValueCallback  
)
```

Inregistreaza o proprietate dependenta :

1. cu numele specificat ,
2. tipul proprietatii,
3. proprietarul proprietatii,
4. metadata proprietatii,
5. si o metoda callback, de validare pentru proprietate.

De interes este parametrul al 4 lea : **PropertyMetadata** care poate fi inlocuit si cu **FrameworkPropertyMetadata** derivata din **UIPropertyMetadata**.

Ierarhia de clase este :

```
PropertyMetadata  
    UIPropertyMetadata  
        FrameworkPropertyMetadata
```

Sintaxa cea mai complexa pentru ctor **PropertyMetadata** este :

```
//
// Summary:
//     Initializes a new instance of the System.Windows.PropertyMetadata
//     class with the specified default value and callbacks.
//
// Parameters:
//     defaultValue:
//         The default value of the dependency property, usually provided
//         as a value of some specific type.
//
//     propertyChangedCallback:
//         Reference to a handler implementation that is to be
//         called by the property system whenever the effective value
//         of the property changes.
//
//     coerceValueCallback:
//         Reference to a handler implementation that is to be called
//         whenever the property system calls
//         System.Windows.DependencyObject.CoerceValue(
//         System.Windows.DependencyProperty)
//         against this property.
//
// Exceptions:
//     System.ArgumentException:
//         defaultValue cannot be set to the value
//         System.Windows.DependencyProperty.UnsetValue;
//         see Remarks.

public PropertyMetadata(
object defaultValue,
PropertyChangedCallback propertyChangedCallback,
CoerceValueCallback coerceValueCallback);
```

Daca utilizam **FrameworkPropertyMetadata** ca parametrul 4 la **Register**, atunci avem posibilitatea de a seta si o comportare a proprietatii dependente.

Unul din ctori (cel mai complex) pentru aceasta clasa are prototipul:

```
Summary:
Initializes a new instance of the System.Windows.FrameworkPropertyMetadata
class with the provided default value and framework metadata options,
specified callbacks, a Boolean that can be used to prevent animation of the
property, and a data-binding update trigger default.

Parameters:
defaultValue:
    The default value of the dependency property,
    usually provided as a specific type.
flags:
    The metadata option flags (a combination of
    System.Windows.FrameworkPropertyMetadataOptions values).
These options specify characteristics of the dependency property
that interact with systems such as layout or data binding.

propertyChangedCallback:
A reference to a handler implementation that the property system will call
whenever the effective value of the property changes.
```

coerceValueCallback:

A reference to a handler implementation that will be called whenever the property system calls `System.Windows.DependencyObject.CoerceValue(System.Windows.DependencyProperty)` against this property.

isAnimationProhibited:

true to prevent the property system from animating the property that this metadata is applied to. Such properties will raise a run-time exception originating from the property system if animations of them are attempted. The default is false.

defaultUpdateSourceTrigger:

The `System.Windows.Data.UpdateSourceTrigger` to use when bindings for this property are applied that have their `System.Windows.Data.UpdateSourceTrigger` set to `System.Windows.Data.UpdateSourceTrigger.Default`.

```
//  
// Exceptions:  
//   System.ArgumentException:  
//     defaultValue is set to  
//   System.Windows.DependencyProperty.UnsetValue; see  
//     Remarks.
```

```
public FrameworkPropertyMetadata(  
    object defaultValue,  
    FrameworkPropertyMetadataOptions flags, PropertyChangedCallback  
    propertyChangedCallback, CoerceValueCallback coerceValueCallback,  
    bool isAnimationProhibited,  
    UpdateSourceTrigger defaultUpdateSourceTrigger);
```

Valorile pentru **flag** afecteaza comportarea proprietatii dependente. Valorile posibile sunt :

Specifies the types of framework-level property behavior that pertain to a particular dependency property in the Windows Presentation Foundation (WPF) property system.

```
[Flags]  
public enum FrameworkPropertyMetadataOptions  
{
```

No options are specified; the dependency property uses the default behavior of the Windows Presentation Foundation (WPF) property system.

```
    None = 0,
```

The measure pass of layout compositions is affected by value changes to this dependency property.

```
    AffectsMeasure = 1,
```

The arrange pass of layout composition is affected by value changes to this dependency property.

```
    AffectsArrange = 2,
```

The measure pass on the parent element is affected by value changes to this dependency property.

```
    AffectsParentMeasure = 4,
```

The arrange pass on the parent element is affected by value changes to this dependency property.

```
    AffectsParentArrange = 8,
```


Some aspect of rendering or layout composition (other than measure or arrange) is affected by value changes to this dependency property.

AffectsRender = 16,

The values of this dependency property are inherited by child elements.

Inherits = 32,

The values of this dependency property span separated trees for purposes of property value inheritance.

OverridesInheritanceBehavior = 64,

Data binding to this dependency property is not allowed.

NotDataBindable = 128,

The **System.Windows.Data.BindingMode** for data bindings on this dependency property defaults to **System.Windows.Data.BindingMode.TwoWay**.

BindsTwoWayByDefault = 256,

The values of this dependency property should be saved or restored by journaling processes, or when navigating by Uniform resource identifiers (URIs).

Journal = 1024,

The subproperties on the value of this dependency property do not affect any aspect of rendering.

SubPropertiesDoNotAffectRender = 2048,

}

Valoarea **Inherits** creaza posibilitatea ca aceasta proprietate dependenta sa poata sa-si propage valoarea catre elemnete descendente.

Observatie

Cand o anumita proprietate dependenta nu-si propaga valorile catre descendenti va trebui sa cercetam in documentatie daca acest flag are bitul corespunzator **Inherits** setat.

// end MSDN

Constructie clasa cu proprietate dependenta

Pentru clasa **Persoana** vom construi o proprietate dependenta– numita **LastName** – si un wrapper pentru aceasta proprietate.

Atentie la clasa **DependencyObject**.

```
public class Persoana : DependencyObject
{
    public static readonly DependencyProperty LastNameProperty =
        DependencyProperty.Register(
            "LastName",
            typeof(string),
            typeof(Persoana));

    public string LastName
    {
        get
        {
            return (string)GetValue(LastNameProperty);
        }
    }
}
```

```
    }  
    set  
    {  
        SetValue(LastNameProperty, value);  
    }  
}  
}
```

Numele dat proprietatii trebuie sa-l inregistram – *LastName* - , furnizam tipul mentinut de aceasta proprietate – **string** – si apoi furnizam tipul de obiect la care se ataseaza aceasta proprietate – *Persoana*.

Observatie:

In get / set nu se pune cod de validare. Cand valoarea unei proprietati dependente se schimba din XAML, se va apela direct **SetValue** si nu **set** definita mai sus.

Trasatura cea mai importanta este aceea ca aceasta proprietate furnizeaza o **notificare** la schimbarea valorii si aceasta notificare se propaga la descendentii clasei.

Proprietatile pot fi setate in XAML fara a fi necesar un cod procedural.

Vom urmari in continuare:

- **notificari** pentru o proprietate dependenta;
- **proagarea valorilor**;
- **furnizori multipli** pentru o proprietate dependenta.

Vom incepe prin a vedea cum se implementeaza o proprietate dependenta.
De aceasta data se furnizeaza valoarea implicita si metoda callback.

```
public class Button : ButtonBase  
{  
    // Proprietatea dependenta  
    public static readonly DependencyProperty IsDefaultProperty;  
    static Button()  
    {  
        // Inregistrare proprietate  
        Button.IsDefaultProperty = DependencyProperty.Register(  
            "IsDefault",  
            typeof(bool), typeof(Button),  
            new FrameworkPropertyMetadata(  
                false,  
                new PropertyChangedCallback(OnIsDefaultChanged))  
            );  
    }  
    ...  
}  
// Un wrapper pentru proprietatea .NET (optional)  
public bool IsDefault  
{  
    get { return (bool)GetValue(Button.IsDefaultProperty); }  
    set { SetValue(Button.IsDefaultProperty, value); }  
}  
// O metoda callback pentru schimbarea proprietatii (optional)  
private static void OnIsDefaultChanged(  
    DependencyObject o,  
    DependencyPropertyChangedEventArgs e)  
{ ... }
```

```
...  
}
```

Discutie

Campul static si readonly, *IsDefaultProperty*, este proprietatea dependenta. Se declara *static* si *readonly* si este de tip **DependencyProperty**. Este sufixat cu “Property”.

Se inregistreaza aceasta proprietate cu metoda statica **DependencyProperty.Register**.

Numele proprietatii va fi **IsDefault**, de tip **bool** si proprietarul proprietatii este clasa **Button**. Se inregistreaza metoda ce va fi apelata cand proprietatea isi schimba valoarea, ultimul parametru din ctor pentru **FrameworkPropertyData**.

Atentie mare la modul cum se implementeaza aceasta proprietate get/set. Obligativu **GetValue** si **SetValue**. In implementare numele proprietatii este **IsDefault**. Descendentii o vor cauta dupa numele **IsDefaultProperty**.

In get / set nu se scrie alt cod – de verificare sau altceva. XAML apeleaza direct **GetValue** si/sau **SetValue**. Codul pentru verificari poate fi scris in metoda callback.

Implementare metoda callback, **OnIsDefaultChanged**, apelata cand proprietatea isi schimba valoarea. Metoda are doi parametri: unul de tip **DependencyObject** si unul derivat din **EventArgs**.

Notificare schimbare

Cand valoarea unei proprietati dependente se schimba, WPF poate genera in mod automat actiuni ce depind de proprietatile metadatei. Aceste actiuni pot fi de redesenare a unor elemente, actualizarea layout-ului, reimprospatarea datelor asociate cu anumite controale, etc. Acest mecanism de notificare este cunoscut si sub numele de “*property trigger*” si nu necesita cod procedural.

Scenariu

Consideram un buton care sa-si schimbe culoarea cand are mouse-ul deasupra.

Metoda 1

Vom trata evenimentele **MouseEnter** si **MouseLeave**.

```
<Button MouseEnter="Button_MouseEnter" MouseLeave="Button_MouseLeave"  
        MinWidth="75" Margin="10">Help</Button>  
<Button MouseEnter="Button_MouseEnter" MouseLeave="Button_MouseLeave"  
        MinWidth="75" Margin="10">OK</Button>
```

iar metodele pentru evenimente sunt :

```
// Schimb Foreground in “blue” cand mouse-ul este pe buton  
void Button_MouseEnter(object sender, MouseEventArgs e)  
{  
    Button b = sender as Button;  
    if (b != null)  
        b.Foreground = Brushes.Blue;  
}  
  
// Restaurez Foreground la “black” (pensula originala)  
// cand mouse paraseste butonul
```

```
void Button_MouseLeave(object sender, MouseEventArgs e)
{
    Button b = sender as Button;
    if (b != null)
        b.Foreground = Brushes.Black;
}
```

Metoda 2

Folosind proprietatea **Trigger**, in XAML putem scrie :

```
<Trigger Property="IsMouseOver" Value="True">
    <Setter Property="Foreground" Value="Red"/>
</Trigger>
```

Observatie:

Cand mouse-ul este deasupra butonului, proprietatea **IsMouseOver** devine *true*, iar proprietatea **Foreground** va fi setata cu valoarea **Red**. Cand mouse-ul paraseste butonul, WPF aduce butonul la starea initiala (Foreground) iar proprietatea **IsMouseOver** devine *false*.

Ceea ce trebuie sa facem este sa atribuim **Trigger** la fiecare buton, mai precis pe stilul butonului.

```
<Button
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    MinWidth="75"
    Margin="30">
    <Button.Style>
        <Style TargetType="{x:Type Button}"> <!--Destinatia -->
            <Style.Triggers>
                <Trigger Property="IsMouseOver" Value="True">
                    <Setter Property="Foreground" Value="Red"/>
                </Trigger>
            </Style.Triggers>
        </Style>
    </Button.Style>
    OK <!-- Continut buton -->
</Button>
```

Pe langa “*property trigger*”, WPF suporta “*data trigger*” si “*event trigger*”.

Data trigger este o proprietate ce lucreaza pentru toate proprietatile .NET (nu numai pentru proprietati dependente). Modificarile unei date pot afecta alte controale din interfata.

Clasa **DataTrigger**. Este legata si de asocierea datelor la controale (data binding).

Clasa **DataTrigger** reprezinta un trigger ce aplica valorile proprietatii sau executa actiuni cand data asociata indeplineste anumite conditii.

Event trigger – permite sa specificam in mod declarativ actiunile ce vor fi executate cand apare un eveniment.

Mostenirea (propagarea) valorii unei proprietati

Nu e mostenirea clasica, ci modul de propagare a valorii proprietatii in arborele de vizualizare.

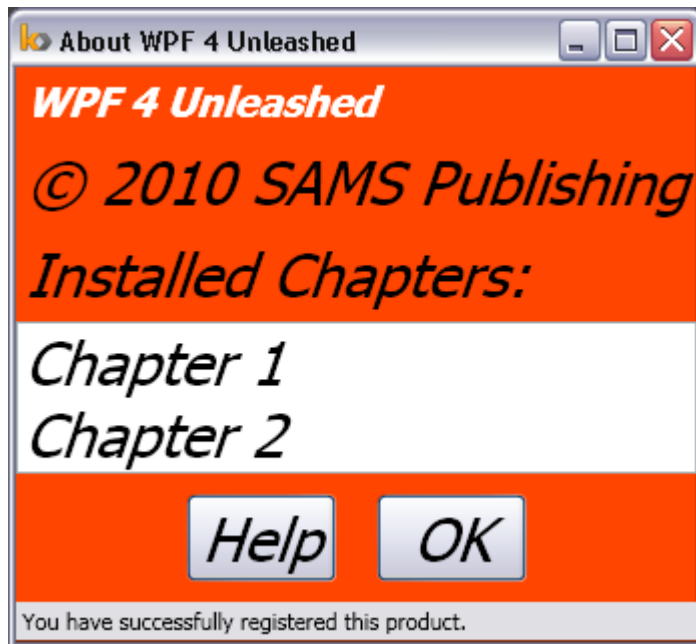
Sa urmarim exemplul ce urmeaza.

La nivel de fereastră `<Window/>` se definește proprietatea `FontSize="30"` și `FontStyle="Italic"`. Acest lucru înseamnă că toate controalele din fereastră (fereastră este un container) ce expun text vor folosi mărimea fontului ca fiind **30** și stilul fontului ***Italic***. Dacă alte controale din această fereastră au setată proprietatea `FontSize` sau `FontStyle`, vor folosi acea valoare și nu cea definită în `<Window />`.

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Title="About WPF 4 Unleashed" SizeToContent="WidthAndHeight"
  Background="OrangeRed"
  FontSize="30" FontStyle="Italic"> <!-- aici e modificarea pentru FontSize -->
  <StackPanel>
    <Label FontWeight="Bold" FontSize="20" Foreground="White">
      <!-- nu schimba FontSize -->
      WPF 4 Unleashed
    </Label>
    <Label>© 2019 SAMS Publishing</Label>
    <Label>Installed Chapters:</Label>
    <ListBox>
      <ListBoxItem>Chapter 1</ListBoxItem>
      <ListBoxItem>Chapter 2</ListBoxItem>
    </ListBox>
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
      <Button MinWidth="75" Margin="10">Help</Button>
      <Button MinWidth="75" Margin="10">OK</Button>
    </StackPanel>
    <StatusBar>You have successfully registered this product.</StatusBar>
  </StackPanel>
</Window>
```

Controalele se aliniază la mărimea ferestrei părinte deoarece fereastră părinte are setată proprietatea `SizeToContent = "WidthAndHeight"`.

Dacă un control are proprietatea `FontSize` setată atunci acel control folosește acea valoare, nu moștenește valoarea din părinte. Vezi primul **Label** de după **StackPanel**.



Observatie

StatusBar este neafectata de aceasta schimbare in controlul parinte.

Nu fiecare proprietate dependenta participa in cadrul acestei mosteniri. Intern, proprietatile dependente pot opta pentru mostenire pasand **FrameworkPropertyMetadataOption.Inherits** in metoda **Register**.

Pot exista alte prioritati – mai mari – ce seteaza valoarea proprietatii.

StatusBar, **Menu** si **ToolTip**, intern folosesc proprietatile fontului sistem, deci ceea ce setam in Control Panel. Daca punem un buton in **StatusBar** acesta va avea fontul sistem.

Suport pentru provideri multipli

Figura urmatoare arata procesul din WPF executat pentru fiecare proprietate dependenta pentru a-i calcula valoarea finala. Procesul are loc automat si se bazeaza pe notificari.

*Determina valoarea de baza => Evaluare (daca e o expresie) => Aplica animatii => Constrangeri (se apeleaza delegate **CoerceValueCallback** daca exista) => Validare (se apeleaza delegate **ValidateValueCallback** daca exista).*

Pas 1. Determina valoarea de baza.

Urmatorii furnizori pot seta valoarea (ordinea de precedenta de la cel mai prioritar catre cel mai putin prioritar !???) :

1. Valoare locala
2. Trigger template parinte
3. Template parinte
4. Setari stiluri
5. Triggeri setari teme
6. Setter-i stil

7. Trigger stil teme
8. Setter-i stil teme
9. Mostenire valoare proprietate
10. Valoare implicita.

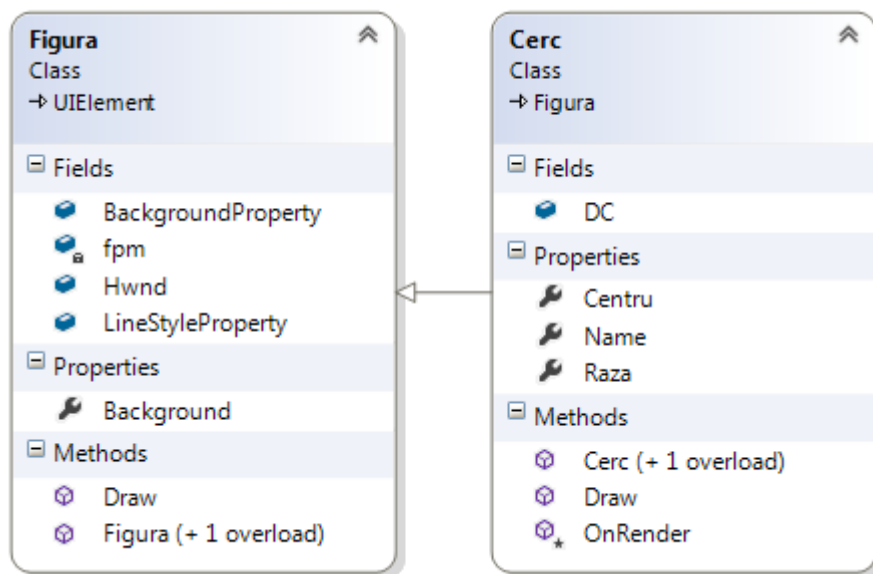
Observatie

Din strategia de determinare a valorii proprietatii dependente se observa ca valoarea implicita se aplica numai atunci cand nu exista setata o alta valoare.

Exemplu complet cu proprietate dependenta

Se creaza ierarhia de clase *Figura* <- *Cerc* (*Cerc* derivata din *Figura*).

In clasa *Figura* se defineste o proprietate dependenta numita *Background*, folosita pentru *Brush* - la umplerea unei figuri.



Clasa *Figura* este derivata din **UIElement**.

Clasa *Cerc* expune proprietatile .NET

```
public string Name { get; set; }  
public int Raza {get; set;}  
public System.Windows.Point Centru { get; set; }
```

si are definiti doi constructori.

Constructorul fara parametru este necesar in XAML, iar celalalt constructor este folosit din cod C#.

Layout-ul pentru fereastra principala este un **StackPanel**. Pe langa alte controale definite in Windows vom folosi si instante ale clasei *Cerc*, pe care le vom desena in cadrul layout-ului. Se trateaza evenimentul "clic stanga mouse" si se adauga noi instante ale clasei *Cerc* (metoda *Mld* din *MainWindow.xaml.cs*).

Tot codul este dat mai jos, inclusiv spatiile de nume.

//Fisierul Figura.cs

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

using System.Windows;
using System.ComponentModel;
using System.Globalization;
using System.Drawing;
using Media=System.Windows.Media;
using Shapes=System.Windows.Shapes;

namespace WpfProprietatiDependente
{
    public class Figura: UIElement
    {
        // Handle pentru fereastra. Va fi dat de fereastra unde
        // se afiseaza aceste obiecte.
        // In acest exemplu nu e folosit
        public IntPtr Hwnd;

        // Brush. Culoare verde : valoarea implicita
        // Proprietatea dependenta Background folosita in XAML
        static FrameworkPropertyMetadata fpm =
            new FrameworkPropertyMetadata(Media.Brushes.Green);
        public static DependencyProperty BackgroundProperty =
            DependencyProperty.Register("Background",
                typeof(Media.Brush), typeof(Figura),
                fpm);

        // Nu e folosita
        public static DependencyProperty LineStyleProperty;

        public Figura()
        {}

        public Figura(IntPtr hwnd)
        {
            Hwnd = hwnd;
        }

        // Necesar pentru apel proprietate din cod C#
        public Media.Brush Background
        {
            get { return GetValue(BackgroundProperty) as Media.Brush; }
            set { SetValue(BackgroundProperty, value); }
        }

        public virtual void Draw() { }
    }

    // Clasa Cerc

    public class Cerc : Figura
    {
        public string Name { get; set; }
        public int Raza {get; set;}
        public System.Windows.Point Centru { get; set; }
    }
}
```



```
// Nu e folosit contextul de desenare -  
// Ceva asemanator exista in MFC - Device Context  
public Media.DrawingContext DC;  
  
// Metoda pentru desenare cerc (elipsa)  
// Apelata in momentul cand se adauga elemente in StackPanel.  
protected override void OnRender(Media.DrawingContext  
    drawingContext)  
{  
    base.OnRender(drawingContext);  
    drawingContext.DrawEllipse(Background, null, Centru,  
        Raza/2, Raza/2);  
}  
  
// Constructor cerut de XAML  
public Cerc() { }  
  
// Constructor folosit in cod C#  
public Cerc(int raza, System.Windows.Point centru, IntPtr hwnd) :  
    base(hwnd)  
{  
    Raza = raza;  
    Centru = centru;  
    Background = Media.Brushes.LightCoral;  
}  
  
// Nu deseneaza ci doar schimba Background  
public override void Draw()  
{  
    Background = Media.Brushes.Yellow;  
}  
}  
}
```

Fisierul MainWindow.xaml

```
<Window x:Class="WpfProprietatiDependente.MainWindow"  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
  
    <!-- Pentru a putea folosi clasa Cerc in XAML -->  
    xmlns:my="clr-namespace:WpfProprietatiDependente"  
  
    Title="MainWindow" Height="350" Width="525" MouseLeftButtonDown="Mld"  
    Loaded="Window_Loaded_1" Name="Wpf">  
    <StackPanel Name="SP" Orientation="Vertical" Loaded="StackPanelLoaded"  
        Unloaded="PUnloaded">  
        <!-- Obiectul Cerc de culoare rosie plasat in stanga sus StackPanel.  
        Proprietatea Name are valoarea "CercXaml". Putem identifica  
        elementul dupa Name.  
        Primul element din StackPanel -->  
        <my:Cerc Raza="20" Centru="10,10" Background="Red" Name="CercXaml"/>  
  
        <!-- Valoare pentru Background este cea implicita (verde)  
        Se ia din clasa Figura.  
        -->  
        <my:Cerc Raza="20" Centru="200,200"/>  
  
        <Label Content="Label" HorizontalAlignment="Left" Height="35"  
            Margin="45,33,0,0" VerticalAlignment="Top" Width="166"  
            Name="LabelControl"/>
```

```
<Button x:Name="B1" Height="44" Margin="10,0" Background="Aquamarine">
    <Button.Style>
        <Style TargetType="{x:Type Button}">
            <!-- -Destinatia -->
            <Style.Triggers>
                <Trigger Property="IsMouseOver" Value="True">
                    <Setter Property="Background" Value="Red"/>
                </Trigger>
            </Style.Triggers>
        </Style>
    </Button.Style>
    <!-- Content pentru Button -->
    <my:Cerc Raza="30" Centru="0,0" Background="BlueViolet" />
</Button>
</StackPanel>
</Window>
```

Fisierul MainWindow.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;

using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

using System.Windows.Interop;

namespace WpfProprietatiDependente
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        // Handle la fereastra principala a aplicatiei
        public IntPtr Hwnd;

        public MainWindow()
        {
            InitializeComponent();
        }

        // Handler eveniment Mouse stanga clic
        private void Mld(object sender, MouseButtonEventArgs e)
        {
            StackPanel sp = sender as StackPanel;

            Window w = sender as Window;
            Point point = e.GetPosition(w);
            LabelControl.Content = point.X.ToString() + " , " +
                point.Y.ToString();
        }
    }
}
```

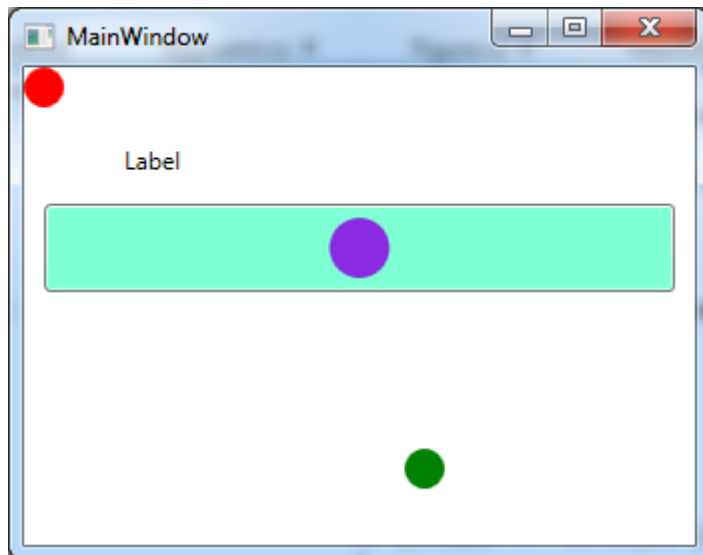
```
// Creez un obiect Cerc in cod si il adaug la StackPanel
// Are proprietatea Background mostenita din Figura
Cerc cerc = new Cerc(20, new Point(point.X, point.Y), Hwnd);
this.SP.Children.Add(cerc);

// Creez un obiect Cerc caruia ii setez proprietatea Background
// si apoi il adaug la StackPanel
Cerc cercnou = new Cerc();
cercnou.Background = System.Windows.Media.Brushes.LightPink;
cercnou.Raza = 25;
cercnou.Centru = new Point(200, 200);
this.SP.Children.Add(cercnou);

// Adaug la StackPanel un obiect Line
Line line = new Line();
line.StrokeThickness = 20;
line.Stroke = System.Windows.Media.Brushes.Green;
line.X1 = 10;
line.X2 = 100;
this.SP.Children.Add(line);
}

// Handler eveniment Loaded
private void Window_Loaded_1(object sender, RoutedEventArgs e)
{
    WindowInteropHelper helper = new WindowInteropHelper(this);
    Hwnd = helper.Handle;
}
}
```

Rezultatul executiei este:



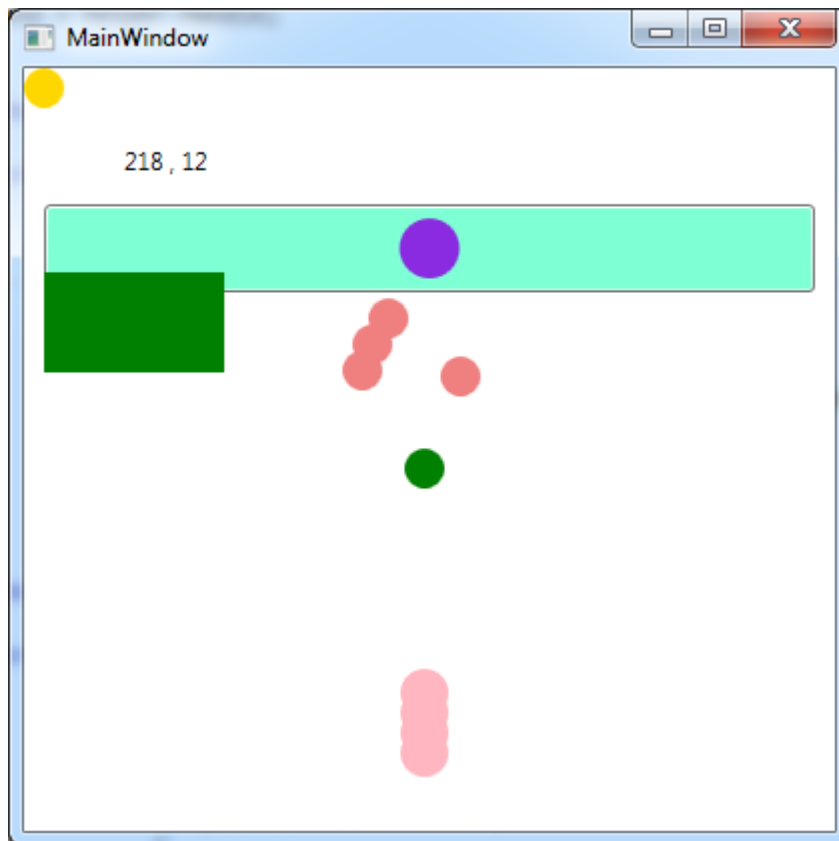
Daca tratam evenimentul **Loaded** pentru **StackPanel** si folosim urmatorul cod (determinare element Cerc cu Name="CercXaml"):

```
private void StackPanelLoaded(object sender, RoutedEventArgs e)
{
    foreach (Object o in SP.Children)
    {
```

```
Cerc c = o as Cerc;  
if (c != null)  
    if (c.Name != null)  
        if (c.Name == "CercXaml")  
        {  
            c.Background =  
                System.Windows.Media.Brushes.Gold;  
            SP.UpdateLayout();  
        }  
    }  
}
```

atunci background-ul pentru acest element se va schimba in *Gold* (la redimensionare fereastra).

La clic stanga in fereastra principala se vor adauga noi elemente la StackPanel si aplicatia poate afisa urmatoarea fereastra:



Incercati sa refaceti acest exemplu.

Proprietati atasate

XAML are abilitatea de a **extinde** tipuri cu proprietati furnizate de alte tipuri – trasatura numita *proprietati atasate*.

În versiunea WPF a lui XAML, proprietățile atașate lucrează numai dacă ambele tipuri sunt derivate din tipul **DependencyObject**.

O proprietate atașată este un concept definit de XAML și a fost construită pentru a fi utilizată ca o proprietate globală a unui tip, proprietate ce poate fi setată pe orice obiect. În WPF, proprietățile atașate sunt definite ca forme specializate de proprietăți dependente ce nu au definite getter-i și setter-i precum proprietățile .NET.

În XAML, setăm o proprietate atașată folosind sintaxa

AttachedPropertyProvider.PropertyName

Tipul ce definește proprietatea atașată urmează unul din următoarele modele:

- Tipul ce definește proprietatea atașată este proiectat astfel încât acesta poate fi element părinte al elementelor pe care va seta valorile pentru proprietatea atașată. Tipul iterează obiectele pe care le conține, obține valorile și le folosește într-un anumit mod.
- Tipul ce definește o proprietate atașată va fi utilizat ca element copil pentru alte elemente părinte.
- Tipul ce definește proprietatea atașată reprezintă un serviciu. Alte tipuri setează valorile pentru proprietatea atașată. Când elementul ce setează proprietatea este evaluat în contextul serviciului, se obțin valorile proprietății atașate.

Proprietăți atașate în cod

Proprietățile atașate în WPF nu au în mod necesar metode "wrapper" pentru accesul prin get/set.

Tipul ce implementează proprietatea atașată trebuie să implementeze accesori dedicați în forma **GetProperty** și **SetProperty**. Aceste metode pot fi folosite în cod pentru a seta sau obține valoarea proprietății.

Din punctul de vedere al implementării, proprietatea atașată este similară cu existența unui câmp ce menține valoarea și existența metodelor pentru accesori, iar acest câmp poate exista pe orice obiect fără a fi nevoie să-l definim în mod special.

Exemplu pentru setarea unei proprietăți atașate în cod.

```
DockPanel myDockPanel = new DockPanel();  
CheckBox myCheckBox = new CheckBox();  
myCheckBox.Content = "Hello";  
myDockPanel.Children.Add(myCheckBox);  
DockPanel.SetDock(myCheckBox, Dock.Top);
```

Dacă nu adăugăm controlul **CheckBox** la **DockPanel** (linia patru din cod), codul din linia cinci nu va genera excepție, nu se va întâmpla nimic.

Metadata proprietatii atasate

Cand inregistram proprietatea, **FrameworkPropertyMetadata** este setat sa specifice caracteristicile proprietatii (modul de desenare, marimea controlului, etc.). In general metadata pentru o proprietate atasata nu e diferita de cea pentru o proprietate dependenta.

In cazul cand suprascriem metadata pentru o proprietate atasata, valoarea furnizata va deveni valoare implicita pentru proprietatea atasata.

Daca dorim sa permitem mostenirea valorii pentru o proprietate atunci trebuie sa folosim proprietati atasate in detrimentul proprietatilor dependente.

Vom defini o proprietate atasata cand avem nevoie de un mecanism de setare a acesteia disponibil si din alte clase decat cea care defineste proprietatea. Proprietatea atasata trebuie inregistrata daca dorim mostenirea valorilor acesteia.

Crearea unei proprietati atasate

Proprietatea atasata se defineste ca o proprietate dependenta prin declararea unui camp **public static readonly** de tip **DependencyProperty**. Acest camp constituie valoarea returnata de metoda **RegisterAttached**. Numele campului trebuie sa fie acelasi cu numele proprietatii sufixat cu "**Property**".

Provider-ul proprietatii atasate trebuie sa furnizeze metodele statice **GetPropertyNames** si **SetPropertyName** pentru proprietatea atasata.

Accesor Get

Prototipul pentru accesorul **GetPropertyNames** trebuie sa fie:

```
public static object GetPropertyNames (object target )
```

PropertyName este numele proprietatii.

Obiectul tinta (target) poate fi un tip mai specific.
Valoarea returnata poate fi un tip mai specific.

Accesor Set

Prototipul pentru accesorul **SetPropertyName** trebuie sa fie:

```
public static void SetPropertyName (object target , object value )
```

PropertyName este numele proprietatii.

Obiectul target poate fi un tip mai specific.
Valoarea returnata poate fi un tip mai specific.

Exemplu. Proprietatea atasata este **IsBubbleSource**.

```
public static readonly DependencyProperty IsBubbleSourceProperty =
    DependencyProperty.RegisterAttached(
        "IsBubbleSource",
        typeof(Boolean),
        typeof(AquariumObject),
        new FrameworkPropertyMetadata(false,
            FrameworkPropertyMetadataOptions.AffectsRender)
    );

public static void SetIsBubbleSource(UIElement element, Boolean value)
{
    element.SetValue(IsBubbleSourceProperty, value);
}

public static Boolean GetIsBubbleSource(UIElement element)
{
    return (Boolean)element.GetValue(IsBubbleSourceProperty);
}
```

Atributele proprietatii atasate

Atributele sunt necesare in procesul de introspectie (reflection) metadata.

AttachedPropertyBrowsableAttribute
AttachedPropertyBrowsableForChildrenAttribute
AttachedPropertyBrowsableForTypeAttribute
AttachedPropertyBrowsableWhenAttributePresentAttribute

Atribut **BrowsableAttribute** : specifica daca o proprietate sau eveniment ar trebui sa fie afisat in fereastra **Properties**.

Termenul *browsable* este analog cu descrierea data de **BrowsableAttribute**, dar starea *browsable* pentru o proprietate atasata este in particular relevanta pentru XAML, pentru ca proprietatea atasata este un concept primar din XAML.

Revenim la proprietati atasate cu un exemplu clasic (MSDN, Internet).

Proprietatea **Dock** este definita de tipul **DockPanel**.

Observatie

Proprietatile atasate sunt totdeauna prefixate cu numele tipului ce furnizeaza proprietatea urmat de punct (.), chiar daca acestea apar ca atribute in XAML.

```
<Window
xmlns:x='http://schemas.microsoft.com/winfx/2006/xaml'
xmlns='http://schemas.microsoft.com/winfx/2006/xaml/presentation'>
<DockPanel>
    <Button DockPanel.Dock='Top'> Top </Button>
    <Button>
        <DockPanel.Dock> Left </DockPanel.Dock>
        Left
    </Button>
</DockPanel>
```

```
</Button>
<Button> Fill </Button>
</DockPanel>
</Window>
```

Observatie: In .NET toate definitiile tag-urilor pentru XAML sunt tipuri CLR.

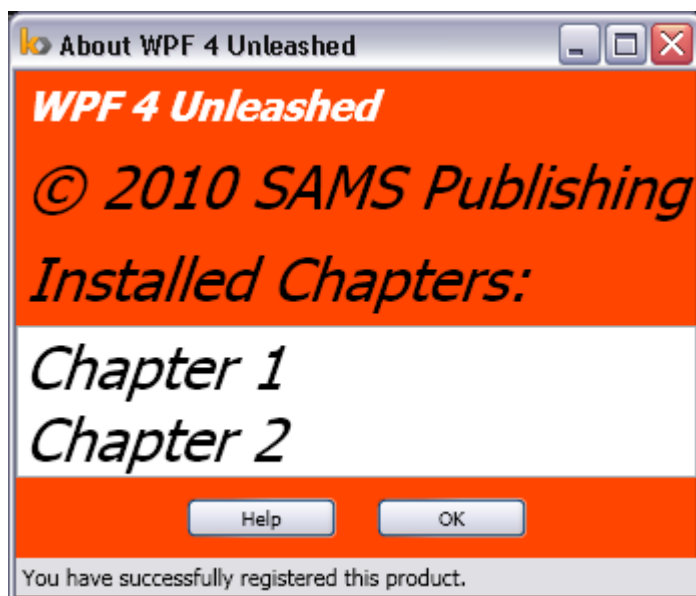
Exemplu. Vrem ca butoanele OK si Help sa apara cu font de *10* si *Normal*. Proprietatea atasata este **FontSize** din clasa **TextElement**.

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        Title="About WPF 4 Unleashed" SizeToContent="WidthAndHeight"
        Background="OrangeRed"
        FontSize="30" FontStyle="Italic">
<StackPanel>
    <Label FontWeight="Bold" FontSize="20" Foreground="White">
        WPF 4 Unleashed
    </Label>
    <Label>© 2019 SAMS Publishing</Label>
    <Label>Installed Chapters:</Label>
    <ListBox>
        <ListBoxItem>Chapter 1</ListBoxItem>
        <ListBoxItem>Chapter 2</ListBoxItem>
    </ListBox>

    <!-- aici e proprietatea atasata: TextElement.FontSize -->

    <StackPanel TextElement.FontSize="10" TextElement.FontStyle="Normal"
        Orientation="Horizontal" HorizontalAlignment="Center">
        <Button MinWidth="75" Margin="10">Help</Button>
        <Button MinWidth="75" Margin="10">OK</Button>
    </StackPanel>
    <StatusBar>You have successfully registered this product.</StatusBar>
</StackPanel>
</Window>
```

Rezultatul:



Observatie

TextElement se numeste provider pentru proprietatea atasata.

Codul C# numai pentru aceasta modificare este :

```
StackPanel panel = new StackPanel();
TextElement.SetFontSize(panel, 30);
TextElement.SetFontStyle(panel, FontStyles.Italic);
panel.Orientation = Orientation.Horizontal;
panel.HorizontalAlignment = HorizontalAlignment.Center;
Button helpButton = new Button();
helpButton.MinWidth = 75;
helpButton.Margin = new Thickness(10);
helpButton.Content = "Help";
Button okButton = new Button();
okButton.MinWidth = 75;
okButton.Margin = new Thickness(10);
okButton.Content = "OK";
panel.Children.Add(helpButton);
panel.Children.Add(okButton);
```

Observatie. O proprietate dependenta se ataseaza folosind metoda statica **DependencyProperty.RegisterAttached**.

MSDN:

```
public static DependencyProperty RegisterAttached(
    string name,
    Type propertyType,
    Type ownerType,
    PropertyMetadata defaultMetadata,
    ValidateValueCallback validateValueCallback
)
```

Parameters

name

Type: [System.String](#)

The name of the dependency property to register.

propertyType

Type: [System.Type](#)

The type of the property.

ownerType

Type: [System.Type](#)

The owner type that is registering the dependency property.

defaultMetadata

Type: [System.Windows.PropertyMetadata](#)

Property metadata for the dependency property. This can include the default value as well as other characteristics.

validateValueCallback

Type: [System.Windows.ValidateValueCallback](#)

A reference to a callback that should perform any custom validation of the dependency property value beyond typical type validation.

Return Value

Type: [System.Windows.DependencyProperty](#)

A dependency property identifier that should be used to set the value of a public static

readonly field in your class. That identifier is then used to reference the dependency property later, for operations such as setting its value programmatically or obtaining metadata.

Construirea unei aplicatii WPF

Primul lucru pe care trebuie sa-l stabilim este layout-ul aplicatiei, sau altfel spus containerul pentru controalele pe care le vom folosi in aplicatie.

Inainte de a discuta despre aceste layout-uri ne indreptam atentia asupra modalitatilor de a controla marimea si pozitionarea controalelor in cadrul ferestrei.

Controlul marimii controalelor

Height si Width

FrameworkElement are proprietatile **Height**, **Width** (tip double), **MinHeight**, **MaxHeight**, **MinWidth** si **MaxWidth** ce pot fi setate din XAML.

De asemenea exista si proprietatile: **DesiredSize**, **RenderSize**, **ActualHeight** si **ActualWidth**.

RenderSize reprezinta marimea finala a elementului dupa ce s-a calculat layout-ul si **ActualHeight** si **ActualWidth** vor avea valorile **RenderSize.Height**, respectiv **RenderSize.Width**.

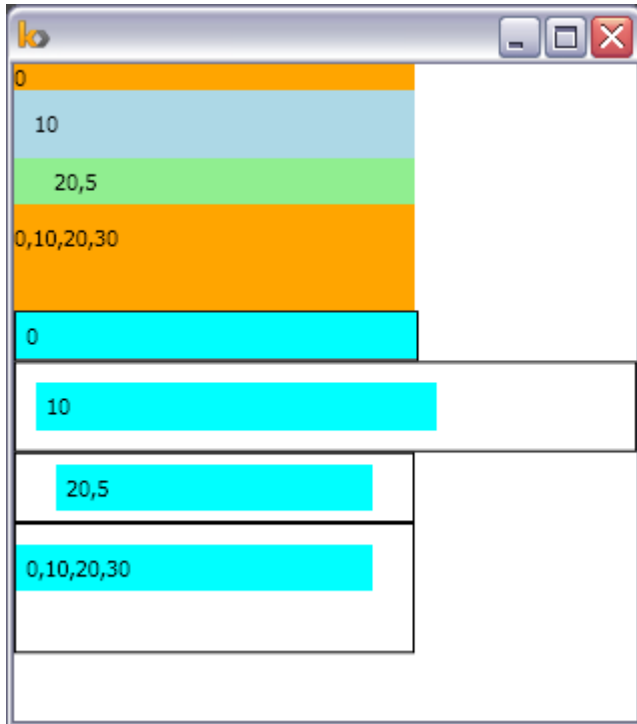
Margin si Padding

FrameworkElement au proprietatea **Margin** si toate controalele (Control) plus **Border** au proprietatea **Padding**.

Margin pozitioneaza elementul fata de marginile layout-ului, **Padding** adauga spatii in interiorul acestuia. **Margin** permite valori negative, **Padding** nu.

Margin si **Padding** sunt de tip **System.Windows.Thickness**.

Analizati exemplul cu **Margin** si **Padding** (figura este din Kaxaml).



Codul XAML este:

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <!-- PADDING: -->
  <!-- 1 value: The same padding on all four sides: -->
  <StackPanel>
    <Label Padding="0" Background="Orange" Width="200"
      HorizontalAlignment="Left">0</Label>
    <Label Padding="10" Background="LightBlue" Width="200"
      HorizontalAlignment="Left" >10</Label>
    <!-- 2 values: Left & Right get the 1st value,
      Top & Bottom get the 2nd value: -->
    <Label Padding="20,5" Background="LightGreen" Width="200"
      HorizontalAlignment="Left">20,5</Label>
    <!-- 4 values: Left,Top,Right,Bottom: -->
    <Label Padding="0,10,20,30" Background="Orange" Width="200"
      HorizontalAlignment="Left">0,10,20,30</Label>
    <!-- MARGIN: -->
    <Border BorderBrush="Black" BorderThickness="1"
      HorizontalAlignment="Left">
    <!-- No margin: -->
    <Label Background="Aqua" Width="200"
      HorizontalAlignment="Left">0</Label>
    </Border>
    <Border BorderBrush="Black" BorderThickness="1">
    <!-- 1 value: The same margin on all four sides: -->
    <Label Margin="10" Background="Aqua" Width="200"
      HorizontalAlignment="Left">10</Label>
    </Border>
    <Border BorderBrush="Black" BorderThickness="1" Width="200"
      HorizontalAlignment="Left">
    <!-- 2 values: Left & Right get the 1st value,
      Top & Bottom get the 2nd value: -->
```

```
<Label Margin="20,5" Background="Aqua" Width="200"
HorizontalAlignment="Left">20,5</Label>
</Border>
<Border BorderBrush="Black" BorderThickness="1" Width="200"
HorizontalAlignment="Left">
<!-- 4 values: Left,Top,Right,Bottom: -->
<Label Margin="0,10,20,30" Background="Aqua" Width="200"
HorizontalAlignment="Left">0,10,20,30</Label>
</Border>
</StackPanel>
</Window>
```

Sintaxa pentru Thickness

`System.Windows.ThicknessConverter` construiește un obiect **Thickness** folosind un string ca data de intrare. Există doi ctori : unul cu un parametru și altul cu patru parametri.

```
myLabel.Margin = new Thickness(10); // Same as Margin="10" in XAML
myLabel.Margin = new Thickness(20,5,20,5); // Same as Margin="20,5" in XAML
myLabel.Margin = new Thickness(0,10,20,30); // Same as Margin="0,10,20,30"
```

Unitatea de măsură folosită este pixel logic = 1/96 inch.

Vizibilitatea : proprietatea Visibility

Este o enumerare cu trei valori posibile :

- **Visible** – element vizibil și participă în layout.
- **Collapsed** – element invizibil și nu participă în layout.
- **Hidden** – element invizibil și participă în layout.

Exemplu :

```
<StackPanel Height="100" Background="Aqua">
    <Button Visibility="Collapsed">Collapsed Button</Button>
    <Button>Below a Collapsed Button</Button>
</StackPanel>
```

Efect : Al doilea buton va apărea pe locul primului buton.

Controlarea poziției

Controlarea poziției se face prin mecanismul de *aliniere*. Aliniere orizontală sau verticală.

HorizontalAlignment : Left, Center, Right, Stretch
VerticalAlignment : Left, Center, Right, Stretch.

Exemplu

```
<StackPanel>
    <Button HorizontalAlignment="Left" Background="Red">Left</Button>
    <Button HorizontalAlignment="Center"
        Background="Orange">Center</Button>
    <Button HorizontalAlignment="Right"
        Background="Yellow">Right</Button>
    <Button HorizontalAlignment="Stretch">Stretch</Button>
</StackPanel>
```

```
Background="Lime">Stretch</Button>  
</StackPanel>
```

Aliniere continut

Clasa **Control** are proprietatile

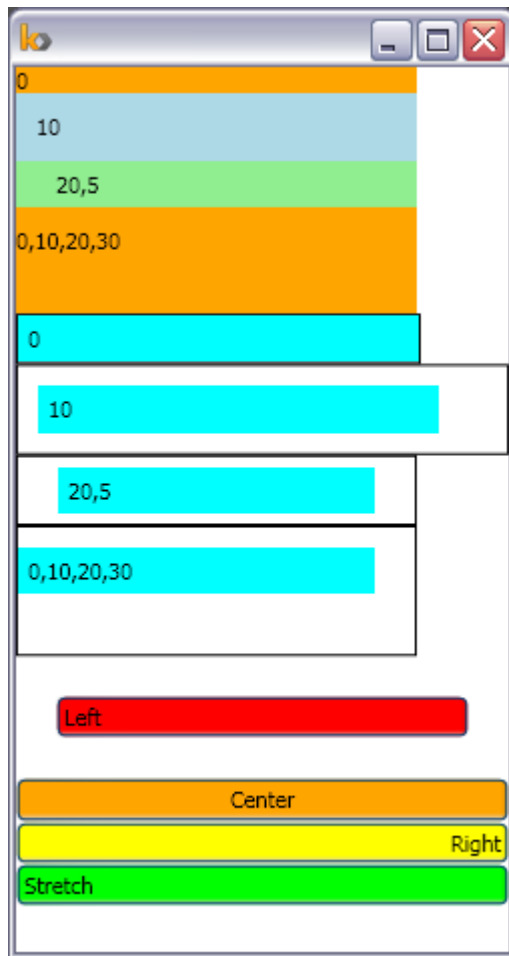
- **HorizontalAlignment** si
- **VerticalContentAlignment**.

Aceste proprietati determina modul de afisare al continutului controlului si cum se completeaza spatiile din interiorul controlului.

Valorile posibile sunt : **Left**, **Right**, **Center** si **Stretch**.

Exemplu

```
<StackPanel>  
    <Button HorizontalContentAlignment="Left" Background="Red">Left</Button>  
    <Button HorizontalContentAlignment="Center"  
        Background="Orange">Center</Button>  
    <Button HorizontalContentAlignment="Right"  
        Background="Yellow">Right</Button>  
    <Button HorizontalContentAlignment="Stretch"  
        Background="Lime">Stretch</Button>  
</StackPanel>
```



FlowDirection

Este o proprietate din **FrameworkElement** si din alte clase si poate schimba modul in care se afiseaza continutul controlului.

Se aplica la « *panel* » si la aranjamentul descendentilor precum si la alinierea continutului in interiorul controlului.

Se foloseste in special pentru culturile care citesc de la dreapta la stanga. Continutul pentru acele culturi se afiseaza de la dreapta la stanga pentru valoarea **RightToLeft**.

Valori posibile : **LeftToRight** si **RightToLeft**.

```
<StackPanel>
    <Button FlowDirection="LeftToRight"
        HorizontalContentAlignment="Left" VerticalContentAlignment="Top"
        Height="40" Background="Red">LeftToRight</Button>
    <Button FlowDirection="RightToLeft"
        HorizontalContentAlignment="Left" VerticalContentAlignment="Top"
        Height="40" Background="Orange">RightToLeft</Button>
</StackPanel>
```

Layout WPF

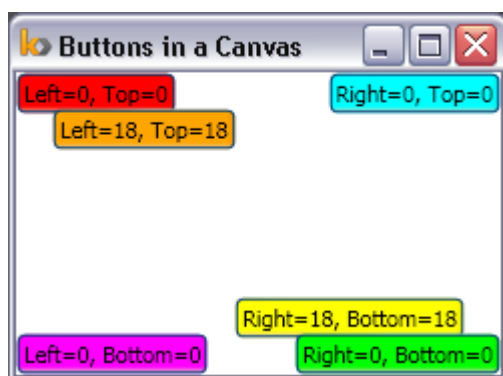
- Canvas
- StackPanel
- WrapPanel
- DockPanel
- Grid

Canvas

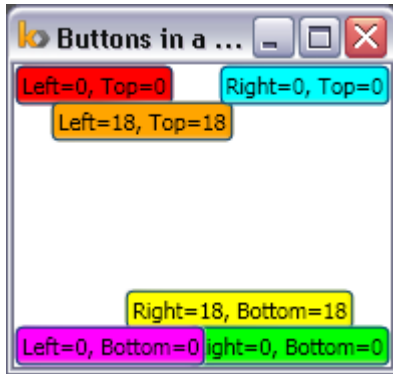
Suporta pozitionarea controalelor printr-o *pozitie fixa*. Se furnizeaza pozitia elementelor. Se pot folosi si proprietatile atasate: `Left`, `Right`, `Top`, `Bottom`. In timpul redimensionarii ferestrei acestea au pozitie fixa.

Exemplu (default7)

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
Title="Buttons in a Canvas">
    <Canvas>
        <Button Background="Red">Left=0, Top=0</Button>
        <Button Canvas.Left="18" Canvas.Top="18"
            Background="Orange">Left=18, Top=18</Button>
        <Button Canvas.Right="18" Canvas.Bottom="18"
            Background="Yellow">Right=18, Bottom=18</Button>
        <Button Canvas.Right="0" Canvas.Bottom="0"
            Background="Lime">Right=0, Bottom=0</Button>
        <Button Canvas.Right="0" Canvas.Top="0"
            Background="Aqua">Right=0, Top=0</Button>
        <Button Canvas.Left="0" Canvas.Bottom="0"
            Background="Magenta">Left=0, Bottom=0</Button>
    </Canvas>
</Window>
```



iar dupa redimensionare putem avea:



Observatie

HorizontalAlignment si **VerticalAlignment** nu au efect.

Ordinea controalelor este data de ordinea de adaugare a acestora in **Canvas**.
Ordinea poate fi schimbata cu ajutorul proprietatii atasate **zIndex**.

StackPanel

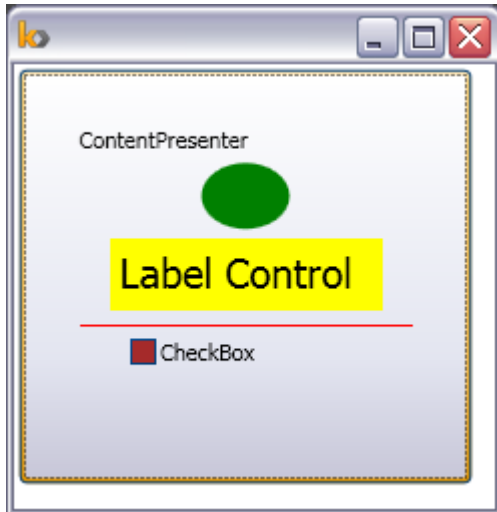
Controalele sunt aranjate pe orizontala sau verticala, in ordinea definirii.

Observatie

Un control cu continut multiplu poate contine **StackPanel** in definita sa.

```
<Button Height="208" HorizontalAlignment="Left" Margin="213,12,0,0"
        Name="button1"
        VerticalAlignment="Top" Width="227" Foreground="#FFBC0000"
        Cursor="Arrow" >
  <Button.Content>
    <StackPanel Height="150" Width="166" Orientation="Vertical" >
      <Ellipse Fill="Green" Width="44" Height="33" Margin="5"
        TextBlock.TextAlignment="Right" />
      <Label Width="136" FontSize="20" Content="Label Control"
        Height="36" Background="Yellow" />
      <Separator Background="Red" Height="10" />
      <CheckBox Content="CheckBox" Height="31" Name="checkBox1"
        Width="115" Background="Brown"
      />
    </StackPanel>
  </Button.Content>
</Button>
```

si rezultatul este : un buton ce contine o elipsa, un Label, un separator si un CheckBox.



(default4)

WrapPanel

Este asemanator cu **StackPanel**. Contine trei proprietati pentru a controla pozitia elementelor.

Orientation—cu valori posibile **Horizontal** si **Vertical** (ca la StackPanel). Horizontal este valoarea implicita. Elementele sunt pozitionate de la stanga la dreapta si apoi de sus in jos daca nu este spatiu suficient pe orizontala – cazul **Horizontal**, si de sus in jos si apoi stanga – dreapta in cazul **Vertical**.

ItemHeight— Inaltime uniforma pentru toate elementele.

ItemWidth— Latime uniforma pentru toate elementele.

DockPanel

Elementele sunt aranjate functie de margini. Proprietatile aplicate sunt: **Top**, **Left**, **Right**, **Bottom**.

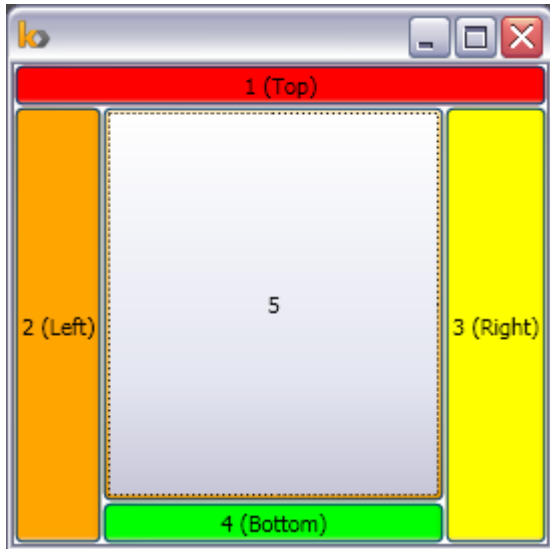
Proprietatea atasata este **Dock**.

Ultimul element poate completa spatiul ramas.

Exemplu

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <DockPanel>
    <Button DockPanel.Dock="Top" Background="Red">1 (Top)</Button>
    <Button DockPanel.Dock="Left" Background="Orange">2 (Left)</Button>
    <Button DockPanel.Dock="Right" Background="Yellow">3 (Right)</Button>
    <Button DockPanel.Dock="Bottom" Background="Lime">4 (Bottom)</Button>
    <Button Background="Aqua">5</Button>
  </DockPanel>
</Window>
```

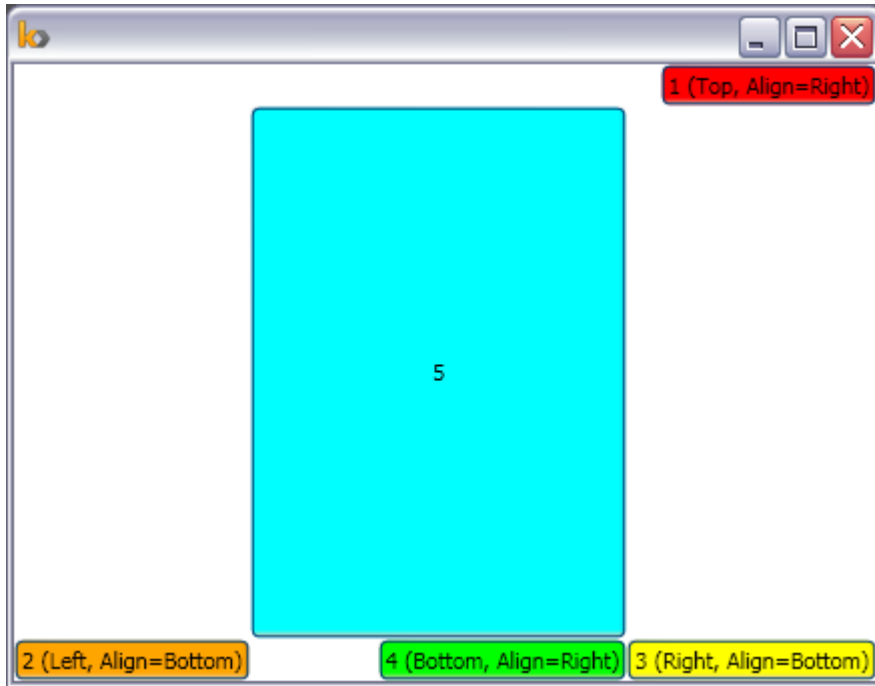
si rezultatul este: (5 ocupa spatiul ramas) (default8)



Exemplu. Atentie la alinierele orizontale.

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <DockPanel>
    <Button DockPanel.Dock="Top" HorizontalAlignment="Right"
      Background="Red">1 (Top, Align=Right)</Button>
    <Button DockPanel.Dock="Left" VerticalAlignment="Bottom"
      Background="Orange">2 (Left, Align=Bottom)</Button>
    <Button DockPanel.Dock="Right" VerticalAlignment="Bottom"
      Background="Yellow">3 (Right, Align=Bottom)</Button>
    <Button DockPanel.Dock="Bottom" HorizontalAlignment="Right"
      Background="Lime">4 (Bottom, Align=Right)</Button>
    <Button Background="Aqua">5</Button>
  </DockPanel>
</Window>
```

Rezultatul este (default9):



Grid

Este asemanator cu tabela din HTML.

Grid contine coloane si randuri ce trebuiesc definite. Ordinea de definitie da indexul.

Se poate specifica marime diferita pentru fiecare rand si / sau coloana.

Proprietatile sunt : **Height** si **Width** folosite in elementul **RowDefinition** / **ColumnDefinition**.

Exemplu

```
<!--Definim patru randuri: -->
```

```
<Grid.RowDefinitions>  
<RowDefinition/>  
<RowDefinition/>  
<RowDefinition/>  
<RowDefinition/>  
</Grid.RowDefinitions>
```

```
<!-- Definim doua coloane: -->
```

```
<Grid.ColumnDefinitions>  
<ColumnDefinition/>  
<ColumnDefinition/>  
</Grid.ColumnDefinitions>
```

Un rand si o coloana se identifica cu proprietatile **Grid.Row** si **Grid.Column**. Cand plasam elemente in **Grid** va trebui sa specificam linia si coloana.

Exemplu

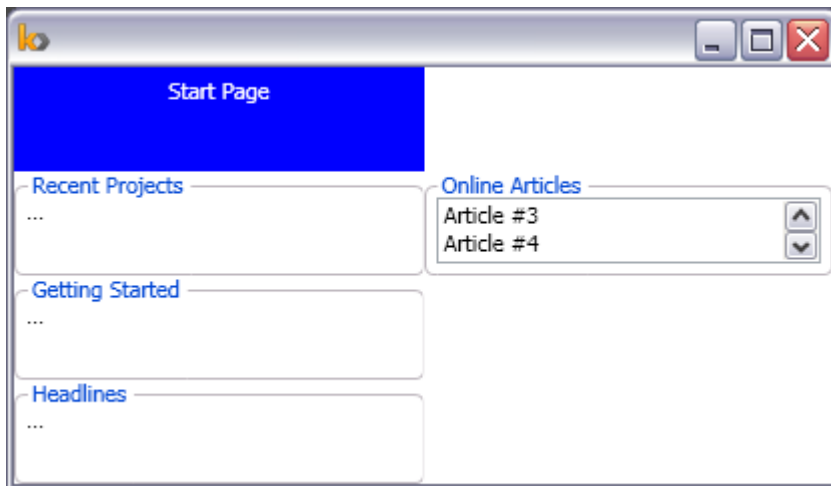
```
<Label Grid.Row="0" Grid.Column="0" Background="Blue" Foreground="White"
      HorizontalContentAlignment="Center">Start Page</Label>
<GroupBox Grid.Row="1" Grid.Column="0" Background="White"
      Header="Recent Projects">...</GroupBox>
<GroupBox Grid.Row="2" Grid.Column="0" Background="White"
      Header="Getting Started">...</GroupBox>
<GroupBox Grid.Row="3" Grid.Column="0" Background="White"
      Header="Headlines">...</GroupBox>
<GroupBox Grid.Row="1" Grid.Column="1" Background="White"
      Header="Online Articles">
  <ListBox>
    <ListBoxItem>Article #1</ListBoxItem>
    <ListBoxItem>Article #2</ListBoxItem>
    <ListBoxItem>Article #3</ListBoxItem>
    <ListBoxItem>Article #4</ListBoxItem>
  </ListBox>
</GroupBox>
</Grid>
```

Pentru acest exemplu, fereastra apare astfel:



Sa observam modul cum au plasate elementele.

Am redimensionat fereastra si...



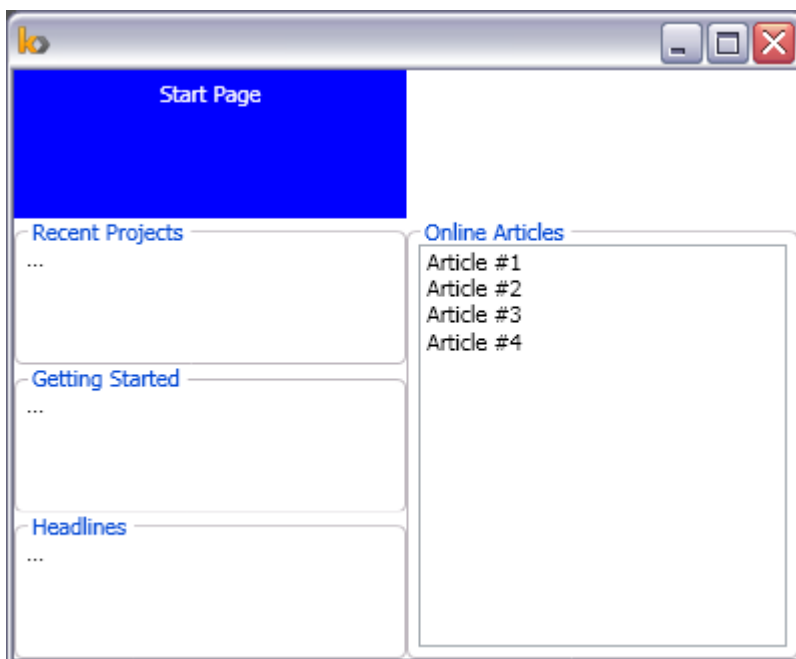
Observam ca **ListBox** are bara de scroll vertical si celelalte controale s-au redimensionat.

Exista proprietatile **Grid.RowSpan** si **Grid.ColumnSpan** ce permit ca un anumit element sa ocupe una sau mai multe linii, respectiv coloane.

Daca in codul de mai sus modificam **GroupBox** de la **ListBox** astfel :

```
<GroupBox Grid.Row="1" Grid.Column="1" Background="White"
  Header="Online Articles" Grid.RowSpan="3">
```

adica am “*spus*” ca elementul poate ocupa 3 randuri, atunci rezultatul este:



Dimensionare randuri si coloane

Grid suporta trei tipuri diferite pentru dimensionarea randurilor si coloanelor:

Absolut: **Height** si **Width** setate la o anumita valoare ce reprezinta pixeli indepent de dispozitiv. Dimensiunea ramane fixa.

Auto dimensionare: **Height= "Auto"**, **Width= "Auto"**, ceea ce da elementului descendent spatiu de cat are nevoie.

Redimensionare proprtionala (*star sizing*): **Height = ".*"**, **Width=".*"** ce are ca efect impartirea spatiului disponibil in regiuni de marimi egale sau regiuni bazate pe un raport fix. Are efect cand **Grid** se redimensioneaza.

Observatie

Daca avem un **Grid** cu patru colaone si dorim ca o coloana sa ocupe 20% din **Grid**, atunci setam **Width="5*"** pentru acea coloana si avem grija ca restul coloanelor sa faca per total 5* pentru latime. ($100:20 = 5$ deci $5* = 100$).

Manipulare continut ce depaseste marimea controlului

Stategiile sunt :

Clipping
Scrolling
Scaling
Wrapping
Trimming

Clipping

Toate elementele **UIElement** au proprietatea booleana **ClipToBounds** ce poate fi utilizata in acest scop.

Scrolling

Elementul trebuie plasat intr-un **ScrollViewer**.

```
<Window Title="Using ScrollViewer"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
<ScrollViewer>
<StackPanel>
...
</StackPanel>
</ScrollViewer>
</Window>
```

Cele mai importante proprietati pentru **ScrollViewer** sunt:

VerticalScrollBarVisibility

HorizontalScrollBarVisibility

cu urmatoarele valori posibile :

Visible – totdeauna vizibila ;

Auto – vizibila numai cand continutul nu poate fi afisat complet.

Hidden – invizibil ScrollBar, dar exista din punct de vedere logic.

Disabled – invizibila si nu exista in mod logic (nu participa la redimensionarea altor controale).

Structura unei aplicatii Window standard

O aplicatie Windows standard contine doua clase principale :

Application
Window

Folosind aceste clase, IDE VS 2010, 12,13,15,17 genereaza clasele *App.xaml* si *MainWindow.xaml* derivate din Application respectiv Window.

Codul C# se gaseste in fisierele *App.Designer.cs* si *MainWindow.Designer.cs*.

Clasa Window

WPF Window este o fereastră ca în Win32. SO nu face distincție între aceste ferestre.

Observatie

Chrome este un alt nume pentru zona « nonclient », ce contine butoanele Minimize, Maximize si Close.

Window este o abstractizare pentru o fereastră Win32. După creare se pot apela metodele Show sau Hide, etc.

Modul de afisare al ferestrei poate fi controlat cu proprietatile: Icon, Title si WindowStyle.

Pozitia este controlata cu proprietatile Left si Top sau setand WindowStartupLocation la una din valorile : CenterScreen sau CenterOwner.

Alte proprietati : Topmost, ShownInTaskbar, etc.

O fereastră poate crea alte ferestre prin instantierea unei clase derivate din Window si apoi apeland metoda Show.

O fereastră descendentă va fi închisă când fereastră părinte este închisă. O asemenea fereastră se numeste « *amodala* ».

Ferestre modale lansate cu ShowDialog().

Clasa Application

Aceasta clasa definește metoda Main (*entry point*) care trebuie să fie adnotată cu atributul [STAThread] și dispecerul de mesaje.

Codul din această metodă ar putea fi :

```
[STAThread]
public static void Main()
{
    Application app = new Application();
    MainWindow window = new MainWindow();
    window.Show();
    app.Run(window);
}
```

Observatie

Aici se creează bucla de mesaje. Toate mesajele la aplicație vor fi « citite » aici și apoi se alege metoda ce tratează mesajul. Dacă nu există nici o metodă atunci se apelează o procedură implicită din Windows (în Win32 numele acesteia este DefWindowProc).

Fiecare fereastră are asociată o procedură pentru tratarea mesajelor. Fiecare control din cadrul unei ferestre (Button, ComboBox, ListBox, etc.) are asociat o procedură fereastră. Când controalele sunt continute în cadrul unei ferestre, mesajele pentru aceste controale constituie mesaje de notificare pentru fereastră părinte, deci codul pentru tratarea mesajelor se va găsi în procedura fereastră părinte.

Observatie

Codul din bucla de mesaje arată astfel (în Win32) :

```
while (GetMessage(&msg, NULL, 0, 0))
{ // ... cod omis
    DispatchMessage(&msg);
}
```

Asemenea cod sau ceva asemănător nu vom vedea într-o aplicație Windows sub platforma .NET.

Cod echivalent de lansare aplicație :

```
[STAThread]
public static void Main()
{
    Application app = new Application();
    app.StartupUri = new Uri("MainWindow.xaml", UriKind.Relative);
    app.Run();
}
```

Observatie

În acest caz metoda Run este fără parametri.

În App.xaml vom avea ceva de genul (observați StartupUri):


```
<Application x:Class="Wpf_C2_2012.App"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

iar codul din C# (generat) este :

```
namespace Wpf_C2_2012
{
    public partial class App : Application { }
}
```

Observatie

Metoda Main se afla in fisierul App.g.cs, iar pentru exemplul de mai sus acesta contine (cod auto generat) :

```
namespace Wpf_C2_2012 {
    /// <summary>
    /// App
    /// </summary>
    [System.CodeDom.Compiler.GeneratedCodeAttribute("PresentationBuildTasks",
        "4.0.0.0")]
    public partial class App : System.Windows.Application {

        /// <summary>
        /// InitializeComponent
        /// </summary>
        [System.Diagnostics.DebuggerNonUserCodeAttribute()]
        public void InitializeComponent() {

            #line 4 "..\..\App.xaml"
            this.StartupUri = new System.Uri(
                "MainWindow.xaml", System.UriKind.Relative);

            #line default
            #line hidden

        }

        /// <summary>
        /// Application Entry Point.
        /// </summary>
        [System.STAThreadAttribute()]
        [System.Diagnostics.DebuggerNonUserCodeAttribute()]
        public static void Main()
        {
            Wpf_C2_2012.App app = new Wpf_C2_2012.App();
            app.InitializeComponent();
            app.Run();
        }
    }
}
```

Evenimentele cele mai des tratate în `Application` (prin suprascrierea metodelor `OnEventName`) sunt : `Activated`, `Deactivated`, `Exit`.

`Application` definește o proprietate `Windows` ce conține o colecție de ferestre instantiate în aplicație.

Fereastra inițială `Window` poate fi accesată cu proprietatea `MainWindow`.

O proprietate importantă a acestei clase: `Properties`.

`Properties` este un dicționar, unde putem memora informații ce vor fi accesibile tuturor ferestrelor sau altor obiecte.

Exemplu

Definire:

```
myApp.Properties["Localitate"] = "Iasi";
```

Regăsire informație (altundeva în cod):

```
string _localitate = myApp.Properties["Localitate"] as string;
```

În acest dicționar se păstrează "object" deci trebuie făcută conversia în momentul utilizării.

Instanța curentă a aplicației se obține cu proprietatea `Application.Current` și ca atare codul de mai sus poate fi rescris astfel:

```
Application.Current.Properties["Localitate"] = "Iasi" ;
```

respectiv

```
string _localitate = Application.Current.Properties["Localitate"] as  
string;
```

Controale

Controalele preconstruite în WPF *pot* fi grupate în următoarele categorii:

Controale ce conțin un singur articol.

Controale ce conțin o colecție de articole.

Controale ce conțin imagini, text și alte controale.

Altele.

Observație

Există și alte moduri de a grupa controalele din `Windows`, nu numai după conținut.

Controale ce contin un singur articol

Aceste controale sunt derivate din `System.Windows.Controls.ContentControl`, au o proprietate `Content` de tip `Object` si contin un singur articol.

Controlul poate contine o arborescenta de obiecte, arborescenta vazuta ca un singur articol. Proprietatea `HasContent` poate fi folosita pentru a determina daca proprietatea `Content` este setata sau nu, si se foloseste in special in XAML.

Controalele preconstruite sunt :

Button
Containere simple
Containere cu antet

Butoane

Clasa de baza `ButtonBase`.

Din `ButtonBase` sunt derivate :

Button
RepeatButton
ToggleButton
CheckBox
RadioButton

Containere simple

Label
ToolTip

Containere cu antete

GroupBox
Expander

Expander : contine un buton ce permite desfasurarea sau extinderea continutului intern.

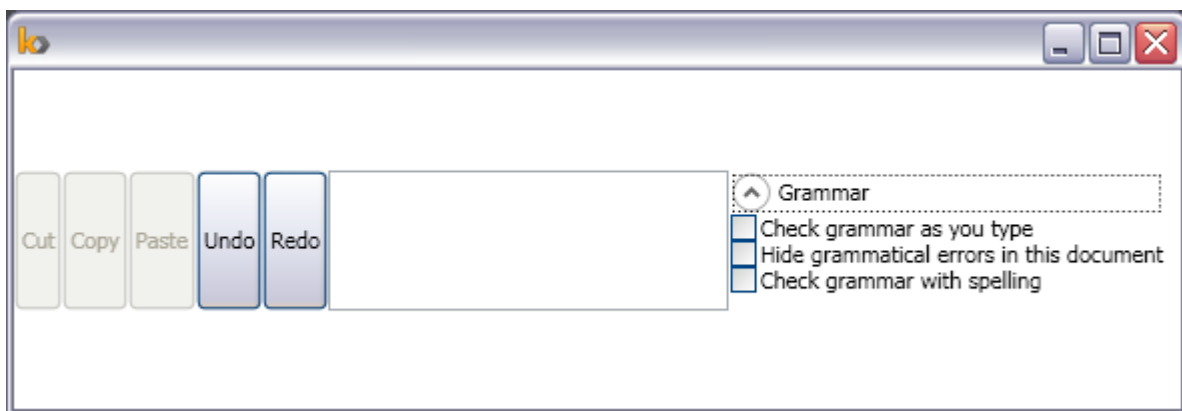
Exemplu

```
<Expander Header="Grammar">
  <StackPanel>
    <CheckBox>Check grammar as you type</CheckBox>
    <CheckBox>Hide grammatical errors in this document</CheckBox>
    <CheckBox>Check grammar with spelling</CheckBox>
  </StackPanel>
</Expander>
```

Rezultat



Dupa desfasurare



Controale ce contin o colectie de articole

Clasa de baza **ItemsControl**.

Contine proprietatea Items de tip `ItemCollection`.

Controale

ListBox
ComboBox
ListView
TabControl
DataGrid

Exemplu

```
<ListBox
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:sys="clr-namespace:System;assembly=mscorlib">
<Button>Button</Button>
<Expander Header="Expander"/>
<sys:DateTime>1/1/2012</sys:DateTime>
<sys:DateTime>1/2/2012</sys:DateTime>
```

Ioan Asiminoaei

```
<sys:DateTime>1/3/2012</sys:DateTime>  
</ListBox>
```

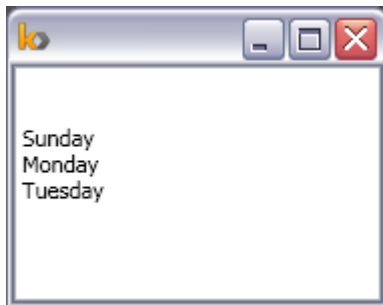
Proprietatea `ItemSource` : permite completarea colectiei de articole cu o colectie existenta.

Proprietatea `DisplayMemberPath`

Exemplu

```
<ListBox xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
xmlns:sys="clr-namespace:System;assembly=mscorlib"  
DisplayMemberPath="DayOfWeek">  
<Button>Button</Button>  
<Expander Header="Expander"/>  
<sys:DateTime>1/1/2012</sys:DateTime>  
<sys:DateTime>1/2/2012</sys:DateTime>  
<sys:DateTime>1/3/2012</sys:DateTime>  
</ListBox>
```

Rezultat



Observatie

Din cauza ca `Button` si `Expander` nu au proprietatea « `DayOfWeek` » acestea se afiseaza ca `TextBox`.

Proprietatea `Path` in WPF

`DisplayMemberPath` suporta sintaxa cunoscuta sub numele de « `property path` » folosita in asocierea de date si animatii.

Ideea de baza a acestei proprietati este de a reprezenta una sau mai multe proprietati pe care le putem folosi in cod pentru a obtine valoarea dorita.

Daca valoarea proprietatii este un obiect complex, putem invoca una din proprietatile obiectului.

Exemplu

Presupunem un obiect ce defineste proprietatea *FirstButton* a tipului `Button`, a carui proprietate `Content` este setata cu « `OK` ».

`FirstButton.Content` reprezinta stringul « OK ».
`FirstButton.Content.Length` reprezinta lungimea stringului.
`FirstButton.Content[0]` reprezinta primul caracter di string, adica “O”.

ItemsPannel

Contribuie la memorarea articolelor si modul cum articolele sunt selectate si afisate.

Aproape toate controalele Windows pot fi alterate din punct de vedere vizual aplicand un nou template la control.

De exemplu `ListBox` afiseaza articolele pe verticala, dar folosind acest template putem schimba aceasta aranjare, ca in codul de mai jos, unde articolele sunt aranjate intr-un `WrapPanel`.

```
<ListBox>
<ListBox.ItemsPanel>
<ItemsPanelTemplate>
<WrapPanel/>
</ItemsPanelTemplate>
</ListBox.ItemsPanel>
...
</ListBox>
```

Echivalent cu XAML de mai sus putem scrie in cod:

```
FrameworkElementFactory panelFactory =
new FrameworkElementFactory(typeof(WrapPanel));
myListBox.ItemsPanel = new ItemsPanelTemplate(panelFactory);
```

Aranjare orizontala a articolelor in ListBox

```
<ListBox>
<ListBox.ItemsPanel>
<ItemsPanelTemplate>
<VirtualizingStackPanel Orientation="Horizontal"/>
</ItemsPanelTemplate>
</ListBox.ItemsPanel>
...
</ListBox>
```

Selectie articole

Articolele din `ListBox`, `ComboBox`, `ListView`, `TabControl`, `DataGrid` pot fi indexate si cel mai important pot fi selectate.

Urmatoarele proprietati pot fi folosite pentru a selecta un articol din aceste controale :

SelectedIndex – returneaza un int ;

SelectedItem – returneaza *instanta* actuala a articolului selectat ;

SelectedValue – valoarea articolului selectat. Putem seta SelectedValuePath pentru a alege o proprietate sau expresie ce reprezinta valoarea articolului. SelectedValuePath lucreaza la fel ca DataMemberPath.

Pot fi folosite proprietatile IsSelected si IsSelectionActive pentru a testa sau schimba comportamentul din cod.

Important este evenimentul SelectedChanged pentru care va trebui sa furnizam o metoda.

Tratare evenimente

Evenimentele in WPF trebuiesc privite din punct de vedere *functional* si din punct de vedere al *implementarii*.

Definitie

Functional: Un eveniment rutat (routed event) este un tip de eveniment ce poate invoca metode pe mai multe obiecte (exista mai multi "listner"-i pe acelasi eveniment). In programarea windows clasica se apeleaza metoda de pe obiectul ce a generat evenimentul.

Definitie

Implementare: Un eveniment rutat este un eveniment CLR ce este memorat de o instanta a clasei RoutedEventArgs si este procesat de sistemul de evenimente WPF.

In general o aplicatie WPF contine mai multe elemente (create in cod sau declarate in XAML), elemente ce formeaza o structura arborescenta (visual tree, logical tree).

Traseul unui eveniment poate fi dat de doua directii, depinde de definitia evenimentului:

- **Bubbling** - De la obiectul care a generat evenimentul in sus in ierarhie pana la radacina (strategie numita *bubbling*).
- **Tunneling** - De la radacina catre obiectul ce a generat evenimentul (strategie numita *tunneling*);
- **Direct** - Se opreste numai la obiectul ce a generat evenimentul (strategie numita *direct*).

Observatie

Elementul *radacina* din descrierea de mai sus este in mod obisnuit o fereastră (*Window*) sau o pagina (*Page*).

Implementare

Un eveniment *rutat* este un eveniment CLR memorat intr-un camp de tip **RoutedEventArgs** si inregistrat cu sistemul de evenimente WPF. Instanta **RoutedEventArgs** obtinuta din inregistrare este retinuta ca un camp "**public static readonly**" in clasa ce inregistreaza si este in acelasi timp proprietara evenimentului. Optional pentru acest eveniment se pot suprascrie metodele *add* si *remove*.

Numele evenimentului este urmat de cuvantul "*Event*".

Evenimentele sunt inregistrate cu metodele statice:

```
EventManager.RegisterRoutedEvent  
sau  
EventManager.RegisterClassHandler
```

In continuare sunt descrise cele doua posibilitati de inregistrare a evenimentelor.

Clasa EventManager

Aceasta clasa este folosita in special pentru a inregistra un nou eveniment - "*routed event*".
Exista doua posibilitati pentru inregistrarea unor asemenea evenimente:

Metoda statica **RegisterRoutedEvent** – handler al instantei.

Metoda statica **RegisterClassHandler** – handler al clasei.

Observatie

Handler-ii inregistrati cu **RegisterClassHandler** sunt invocati inaintea handler-ilor instantei.

Metoda RegisterRoutedEvent

Prototipul pentru **RegisterRoutedEvent** este urmatorul:

```
public static RoutedEvent RegisterRoutedEvent (  
    string name,  
    RoutingStrategy routingStrategy,  
    Type handlerType,  
    Type ownerType  
)
```

Descriere parametri

name – numele evenimentului rutat. Numele trebuie sa fie unic in interiorul tipului si nu poate fi "*null reference*" sau stringul vid.

routingStrategy - strategia de rutare a evenimentului care este descrisa de enumerarea **RoutingStrategy**.

handlerType – tipul metodei pentru tratarea evenimentului. Acesta trebuie sa fie un delegate si nu poate fi "*null reference*".

ownerType – tipul clasei proprietare a evenimentului. Nu poate fi "*null reference*".

Valoare returnata : Un obiect de tip **RoutedEvent** ce poate fi memorat ca un camp static si apoi folosit ca parametru la metode ce ataseaza "handlers-i" la eveniment.

Exemplu

```
public class Button : ButtonBase
{
    // Evenimentul "rutat". Tipul este RoutedEventArgs.
    // Numele este ClickEvent
    public static readonly RoutedEventArgs ClickEvent;

    static Button()
    {
        // Inregistrare eveniment
        Button.ClickEvent =EventManager.RegisterRoutedEvent("Click",
            RoutingStrategy.Bubble,
            typeof(RoutedEventHandler),
            typeof(Button));
    }
    ...
}

// Wrapper .NET pentru Click (optional)
public event RoutedEventArgs Click
{
    add { AddHandler(Button.ClickEvent, value); }
    remove { RemoveHandler(Button.ClickEvent, value); }
}

protected override void OnMouseLeftButtonDown(MouseButtonEventArgs e)
{
    ...
    // Genereaza evenimentul
    RaiseEvent(new RoutedEventArgs(Button.ClickEvent, this));
    ...
}
...
}
```

Metodele **AddHandler**, **RemoveHandler** si **RaiseEvent** sunt mostenite din **UIElement**.

Metoda RegisterClassHandler

Prototipul metodei este (varianta 1)

```
public static void RegisterClassHandler (
    Type classType,
    RoutedEventArgs routedEvent,
    Delegate handler
)
```

classType tipul clasei ce declara tratarea evenimentului.
routedEvent identificatorul evenimentului rutat.
handler referinta la metoda de tratare a evenimentului.

Exemplu - MSDN

```
static MyEditContainer()
{
    EventManager.RegisterClassHandler(
        typeof(MyEditContainer),
        PreviewMouseRightButtonDownEvent,
        new RoutedEventHandler(LocalOnMouseRightButtonDown));
}

internal static void LocalOnMouseRightButtonDown(
    object sender,
    RoutedEventArgs e)
{
    MessageBox.Show(
        "This is invoked before the On* class handler on UIElement");
    // e.Handled = true;
    // uncommenting this would cause ONLY the subclass' class
    // handler to respond
}
```

Alt prototip pentru **RegisterClassHandler**

```
public static void RegisterClassHandler (
    Type classType,
    RoutedEvent routedEvent,
    Delegate handler,
    bool handledEventsToo
)
```

Parametri (ca mai sus)

handledEventsToo **true:** invoca acest handler chiar daca evenimentul rutat a fost marcat ca "tratata" (**Handled = true;**);
 false: retine comportarea implicita, nu se invoca handler-i pe evenimentele marcate ca tratate.

Observatie

Handler-ii inregistrati cu **RegisterClassHandler** sunt invocati inaintea handler-ilor instantei.

Handler-i ai clasei si handler-i ai instantei

Evenimentele rutate considera doua tipuri de "listeners" pentru evenimente: "*class listeners*" si "*instance listeners*".

"*Class listeners*" exista pentru ca tipurile au apelat in constructorul lor static, metoda **EventManager.RegisterClassHandler**, sau au suprascris o metoda virtuala din clasa de baza (metoda se refera la **RegisterClassHandler**).

"*Instance listeners*" sunt instanțe particulare ale clasei unde unul sau mai mulți handlers-i au fost atașați pentru acest eveniment rutat printr-un apel al metodei `AddHandler`. Elementele dintr-un arbore vizual sunt verificate dacă au implementări pentru handlers-i. Aceste metode (handlers-i) sunt apelate funcție de strategia de rutare folosită (*direct*, *tunneling*, *bubbling*).

Observație

Pentru fiecare nod dat din ruta unui eveniment, "*listeners*" din clase au oportunitatea de a răspunde la un eveniment înainte oricărui *listener* al instanței. Din acest motiv, handlers-i din clasă sunt utilizați pentru a opri propagarea unui eveniment sau pentru a furniza tratări speciale.

Handler-i virtuali

Anumite elemente, în particular elementele de bază cum ar fi **UIElement**, expun metode virtuale (fără implementări) ce corespund evenimentelor publice rutate. Aceste metode virtuale pot fi suprascrise pentru a implementa un handler la nivel de clasă pentru acel eveniment.

Strategii de rutare și metode pentru evenimente

Când se înregistrează un eveniment, se specifică una din următoarele strategii de rutare – strategie care se referă la modul cum evenimentul traversează arborele logic.

Valorile sunt date de enumerarea **RoutingStrategy**:

- **Tunneling** – evenimentul este dat elementului *radacina* și apoi se parcurge arboreșcența în *jos* până când se găsește elementul sursă sau până când un element marchează evenimentul ca tratat – proprietatea **Handled = true**;
- **Bubbling** – evenimentul este dat elementului *sursa* și apoi fiecărui element în *sus* pe ierarhie până se ajunge la radacina sau până când un element marchează evenimentul ca tratat – proprietatea **Handled = true**;
- **Direct** – evenimentul este dat numai elementului sursă.

Pattern-ul metodelor ce tratează un eveniment este :

void MetodaEvent (object sender, ClasaDerivataDinRoutedEventArgs e)

Parametrul *sender* este totdeauna elementul la care a fost atașată metoda.

Parametrul *e* – instanța a clasei **RoutedEventArgs** sau clasă derivată din **RoutedEventArgs**, expune următoarele proprietăți:

- **Source** – elementul din arborele logic care a generat evenimentul.
- **OriginalSource** – elementul din arborele vizual ce a generat evenimentul.
- **Handled** – de tip bool. Dacă se setează pe *true* înseamnă că evenimentul a fost tratat și nu mai este propagat la alte elemente (cazurile *Tunneling* și *Bubbling*).
- **RoutedEvent** – obiectul actual al evenimentului rutat (ex. `Button.Click`), ce poate fi util în a determina evenimentul generat când aceeași metodă este folosită pentru a trata evenimente multiple.

Observatie

Deosebirea dintre *Source* si *OriginalSource* se face numai pentru evenimente fizice (clic mouse), nu si pentru evenimente abstracte, caz in care *Source* = *OriginalSource*.

Evenimente in XAML si cod

Clasa **UIElement** defineste mai multe evenimente rutate pentru keyboard, mouse, multi-touch. Multe dintre acestea sunt evenimente *bubbling*, iar o parte fac pereche cu evenimente *tunneling*.

Evenimentele *tunneling* prin conventie sunt prefixate cu *Preview* si sunt trimise inaintea celor *bubbling*.

De exemplu evenimentul *tunneling* *PreviewMouseMove* este trimis inaintea evenimentului *bubbling* *MouseMove*.

Acest mod de tratare face posibila actiunea de anulare sau modificare a unui eveniment.

Prin conventie, elementele preconstruite din WPF actioneaza numai ca raspuns la un eveniment *bubbling*.

Scenariu

Dorim sa implementam un TextBox ce restrictioneaza intrari dupa un anumit pattern sau expresie regulata.

Daca tratam evenimentul *PreviewKeyDown* al TextBox-ului putem seta **Handled** pe *true* pentru a opri propagarea evenimentului dar in acelasi timp nu se va propaga evenimentul *KeyDown* si ca atare caracterul tastat nu va aparea in TextBox.

Exemplu

```
<Window x:Class="Wpf_C2_2012.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525"

        <!-- Eveniment tratat la nivel de fereastră -->
        MouseRightButtonDown="MainWindow_MouseRightButtonDown"
        SizeToContent="WidthAndHeight" Background="Orange"
    >
    <StackPanel>
        <Label FontWeight="Bold" FontSize="20" Foreground="White">
            WPF 4 Unleashed
        </Label>
        <Label>© 2010 SAMS Publishing</Label>
        <Label>Installed Chapters:</Label>
        <ListBox>
            <ListBoxItem>Chapter 1</ListBoxItem>
            <ListBoxItem>Chapter 2</ListBoxItem>
        </ListBox>
    </StackPanel>
```

```
<StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
    <Button MinWidth="75" Margin="10">Help</Button>
    <Button MinWidth="75" Margin="10">OK</Button>
</StackPanel>
<StatusBar>You have successfully registered this product.</StatusBar>
</StackPanel>
</Window>
```

iar codul sursa este (am omis directivele using):

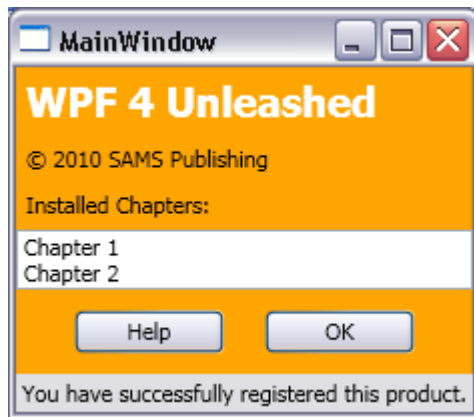
```
namespace Wpf_C2_2012
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void MainWindow_MouseRightButtonDown(object sender,
            MouseButtonEventArgs e)
        {
            // Afisare informatii despre eveniment
            this.Title = "Source = " + e.Source.GetType().Name + ",
                OriginalSource = " +
                e.OriginalSource.GetType().Name + " @ " + e.Timestamp;

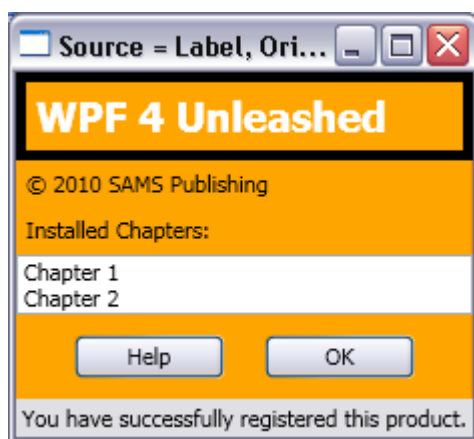
            // In acest exemplu, toate sursele sunt derivate din Control
            Control source = e.Source as Control;

            // On/Off the border on the source control
            // Pentru controalele care nu suporta proprietatea
            // BorderThickness codul de mai jos nu se executa.
            // Vezi controlul Button. Arborele de vizualizare pentru
            // Button nu are elementul Border
            if (source.BorderThickness != new Thickness(5))
            {
                source.BorderThickness = new Thickness(5);
                source.BorderBrush = Brushes.Black;
            }
            else
                source.BorderThickness = new Thickness(0);
        }
    }
}
```

Rezultatul este:



iar dupa clic dreapta mouse pe un *Label* interfata arata ca in figura de mai jos:



Observati incadrarea pentru Label.

Observatie

Daca facem clic dreapta mouse in *ListBox* nu se executa ceea ce ne-am astepta deoarece *ListBox* trateaza intern acest eveniment pentru selectia articolelor.

Oprirea rutarii unui eveniment este o iluzie (`Handled = true`).

Din cod putem face ca un mesaj sa nu mai fie rutat folosind in ctor clasei (dupa `InitializeComponent()`) metoda *AddHandler* ca in exemplul de mai jos :

```
public MainWindow()
{
    InitializeComponent();
    // apel AddHandler
    this.AddHandler(
        Window.MouseRightButtonDownEvent,
        new MouseButtonEventHandler(MainWindow_MouseRightButtonDown),
        true);
}
```

In acest caz evenimentul *MouseRightButtonDown* va fi primit de fereastra *MainWindow* chiar si cand am facut clic in *ListBox*.

Evenimente atasate

WPF suporta *tunneling* si *bubbling* pentru evenimente chiar daca elementele nu au definit acel eveniment. Acest lucru e posibil datorita evenimentelor atasate.

Evenimentele atasate sunt asemanatoare cu proprietatile atasate. Sintaxa este cea de la proprietati statice (numeClasa.proprietateStatica).

Exemplu

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="AboutDialog" ListBox.SelectionChanged="ListBox_SelectionChanged"
        Button.Click="Button_Click"
        Title="About WPF Unleashed" SizeToContent="WidthAndHeight"
        Background="OrangeRed">
    <StackPanel>
        <Label FontWeight="Bold" FontSize="20" Foreground="White">
            WPF 4 Unleashed
        </Label>
        <Label>© 2010 SAMS Publishing</Label>
        <Label>Installed Chapters:</Label>
        <ListBox>
            <ListBoxItem>Chapter 1</ListBoxItem>
            <ListBoxItem>Chapter 2</ListBoxItem>
        </ListBox>
        <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
            <Button MinWidth="75" Margin="10">Help</Button>
            <Button MinWidth="75" Margin="10">OK</Button>
        </StackPanel>
        <StatusBar>You have successfully registered this product.</StatusBar>
    </StackPanel>
</Window>
```

Ceea ce trebuie sa facem in continuare este sa scriem cod in metodele *ListBox_SelectionChanged* si *Button_Click*.

Fiecare eveniment rutat poate fi folosit ca un eveniment *atasat*.

La runtime, *AddHandler* este apelata pentru a atasa cele doua evenimente la fereastra. Acest lucru este echivalent cu plasarea urmatorului cod in ctor clasei *MainWindow* :

```
public MainWindow()
{
    InitializeComponent();
    this.AddHandler(ListBox.SelectionChangedEvent,
        new SelectionChangedEventHandler(ListBox_SelectionChanged));
    this.AddHandler(Button.ClickEvent,
        new RoutedEventHandler(Button_Click));
}
```

Evenimente de la tastatura

Aceste evenimente (*bubbling*) sunt **KeyDown** si **KeyUp** si *tunneling* **PreviewKeyDown** respectiv **PreviewKeyUp**.

Al doilea parametru al evenimentului este de tip **KeyEventArgs** ce poate fi interogat pentru a obtine informatii despre tastele apasate.

Exemplu

```
protected override void OnKeyDown(KeyEventArgs e)
{
    if ((e.KeyboardDevice.Modifiers & ModifierKeys.Alt) ==
        ModifierKeys.Alt
        && (e.Key == Key.A || e.SystemKey == Key.A))
    {
        // Alt+A a fost apasat, posibil cu CTRL, Shift si/sau Windows
    }

    base.OnKeyDown(e);
}
```

In uratorul exemplu se testeaza daca a fost apasata numai combinatia de taste Alt+A :

```
protected override void OnKeyDown(KeyEventArgs e)
{
    if (e.KeyboardDevice.Modifiers == ModifierKeys.Alt
        && (e.Key == Key.A || e.SystemKey == Key.A))
    {
        // Numai Alt+A a fost apasat
    }
    base.OnKeyDown(e);
}
```

Pentru a testa daca s-a apasat *Alt* din partea stanga a tastaturii folosim uratorul cod :

```
// Combinatia de taste este Alt+A
protected override void OnKeyDown(KeyEventArgs e)
{
    if (e.KeyboardDevice.Modifiers == ModifierKeys.Alt
        && (e.Key == Key.A || e.SystemKey == Key.A)
        && e.KeyboardDevice.IsKeyDown(Key.LeftAlt))
    {
        // LeftAlt+A a fost apasat
    }
    base.OnKeyDown(e);
}
```

Focus

UIElement defineste mai multe proprietati si evenimente legate de focus.

- **IsKeyboardFocused** : elementul curent are focusul.
- **IsKeyboardFocusedWithin** – acelasi ca mai sus dar pentru elementele descendente.

Evenimentele ce raporteaza schimbari in aceste proprietati sunt :

- `IsKeyboardFocusedChanged`,
- `IsKeyboardFocusWithinChanged`,
- `GotKeyboardFocus`,
- `LostKeyboardFocus`,
- `PreviewGotKeyboardFocus`,
- `PreviewLostKeyboardFocus`.

Evenimente asociate mouse-ului

- `MouseEnter` si `MouseLeave`
- `MouseMove` si `PreviewMouseMove`
- `MouseLeftButtonDown`,
- `MouseRightButtonDown`,
- `MouseLeftButtonUp`,
- `MouseRightButtonUp`,
- `MouseDown`
- `MouseUp`,

precum si versiunile **PreviewXXX** ale acestor evenimente.

Observatie

Metoda atasata acestor evenimente contine un argument derivat din **EventArgs** ce furnizeaza informatii cu privire la punctul unde a fost facut clic, etc.

Comenzi

Versiune abstracta si slab cuplata pentru evenimente.

O comanda este orice obiect ce implementeaza interfata **ICommand** (`System.Windows.Input`). Comenzile reprezinta actiuni independente de interfata expusa utilizatorului.

Exemplu de comenzi : Cut, Copy, Paste, etc.

Aplicatiile expun aceste actiuni prin mai multe mecanisme : articole de meniu (Menu) sau meniu contextual (ContextMenu), butoane pe Toolbar, shortcut-uri, etc.

Exista posibilitatea de a invalida anumite comenzi daca nu sunt indeplinite anumite conditii (de exemplu Paste trebuie invalidat cand nu avem nimic in clipboard).

WPF defineste un numar de comenzi preconstruite.

Comenzile au construit suport automat pentru activare (shortcut).

Anumite controale din WPF au comportari preconstruite legate de anumite comenzi.

Comenzi preconstruite

O comanda este orice obiect ce implementeaza interfata **ICommand** (`System.Windows.Input`), ce defineste trei membri :

1. **Execute**—Metoda ce executa comanda.

2. **CanExecute**— O metoda ce returneaza *true* daca comanda este *enabled* sau *false* in caz contrar (*disable*).
3. **CanExecuteChanged**—Un eveniment ce se genereaza cand se schimba valoarea lui **CanExecute**.

Daca dorim sa cream comenzile Cut, Copy si Paste, trebuie sa definim si sa implementam trei clase derivate din **ICommand**, memoram instantele acestor clase (in clasa unde avem nevoie) si apelam **Execute** daca **CanExecute** returneaza *true* si de asemenea sa implementam metoda pentru evenimentul **CanExecuteChanged**.

Anumite controale cum ar fi **Button**, **CheckBox** si **MenuItem** au construita logica necesara pentru a interactiona cu orice comanda. Acestea expun o proprietate **Command** – de tip **ICommand**.

Cand este setata aceasta proprietate se executa comanda specificata la un eveniment **Click**.

Sincronizarea intre **IsEnabled** si **CanExecute** este facuta de framework.

Proprietatea **Command** poate fi setata in XAML.

Comenzile preconstruite din WPF sunt expuse ca proprietati statice ale urmatoarelor clase :

ApplicationCommands –	Close, Copy, Cut, Delete, Find, Help, New, Open, Paste, Print, PrintPreview, Properties, Redo, Replace, Save, SaveAs, SelectAll, Stop, Undo, etc.
ComponentCommands –	MoveDown, MoveLeft, MoveRight, MoveUp, ScrollByLine, ScrollPageDown, ScrollPageLeft, ScrollPageRight, ScrollPageUp, SelectToEnd, SelectToHome, SelectToPageDown, SelectToPageUp, etc.
MediaCommands –	ChannelDown, ChannelUp, DecreaseVolume, FastForward, IncreaseVolume, MuteVolume, NextTrack, Pause, Play, PreviousTrack, Record, Rewind, Select, Stop, etc.
NavigationCommands –	BrowseBack, BrowseForward, BrowseHome, BrowseStop, Favorites, FirstPage, GoToPage, LastPage, NextPage, PreviousPage, Refresh, Search, Zoom, etc.
EditingCommands –	AlignCenter, AlignJustify, AlignLeft, AlignRight, CorrectSpellingError, DecreaseFontSize, DecreaseIndentation, EnterLineBreak, EnterParagraphBreak, IgnoreSpellingError, IncreaseFontSize, IncreaseIndentation, MoveDownByLine, MoveDownByPage, MoveDownByParagraph, MoveLeftByCharacter, MoveLeftByWord, MoveRightByCharacter, MoveRightByWord, etc.

Fiecare din aceste proprietati sunt instantele clasei **RoutedUICommand**, o clasa ce implementeaza **ICommand** si in plus suporta *bubbling*.

Toate obiectele **RoutedUICommand** definesc o proprietate **Text** ce contine un nume pentru comanda, nume ce poate fi afisat utilizatorului (un text ce descrie comanda).

Diferenta dintre clasele **RoutedUICommand** si **RoutedCommand** este proprietatea **Text**.

Toate clasele derivate din **UIElement** (si **ContentElement**) contin o colectie **CommandBindings**, colectie ce poate contine unul sau mai multe obiecte **CommandBinding**. **CommandBinding** asociaza comanda la un element.

Exemplu

Adaugam **CommandBinding** pentru Help la fereastra *MainWindow* ca mai jos scriind codul:

```
this.CommandBindings.Add(  
    new CommandBinding(ApplicationCommands.Help,  
        HelpExecuted,  
        HelpCanExecute));
```

Metodele apelate *HelpExecuted* si *HelpCanExecute* trebuie sa fie definite. Aceste metode vor fi apelate la comanda Help.

In XAML putem scrie:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    x:Class="AboutDialog"  
    Title="About WPF Unleashed" SizeToContent="WidthAndHeight"  
    Background="OrangeRed">  
    <Window.CommandBindings>  
        <CommandBinding Command="Help"  
            CanExecute="HelpCanExecute" Executed="HelpExecuted"/>  
    </Window.CommandBindings>  
    <StackPanel>  
        <Label FontWeight="Bold" FontSize="20" Foreground="White">  
            WPF 4 Unleashed  
        </Label>  
        <Label>© 2010 SAMS Publishing</Label>  
        <Label>Installed Chapters:</Label>  
        <ListBox>  
            <ListBoxItem>Chapter 1</ListBoxItem>  
            <ListBoxItem>Chapter 2</ListBoxItem>  
        </ListBox>  
        <StackPanel Orientation="Horizontal"  
            HorizontalAlignment="Center">  
            <Button MinWidth="75" Margin="10" Command="Help" Content=  
                "{Binding RelativeSource={RelativeSource Self},  
                    Path=Command.Text}"/>  
            <Button MinWidth="75" Margin="10">OK</Button>  
        </StackPanel>  
        <StatusBar>You have successfully registered this product.  
    </StatusBar>  
    </StackPanel>  
</Window>
```

si in cod avem

```
using System.Windows;  
using System.Windows.Input;  
public partial class AboutDialog : Window
```

```
{
public MainWindow()
{
    InitializeComponent();
}
void HelpCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = true;
}
void HelpExecuted(object sender, ExecutedRoutedEventArgs e)
{
    System.Diagnostics.Process.Start("http://www.adamnathan.net/wpf");
}}
```

Exemplu cu RoutedCommand - MSDN

Declararea in XAML poate fi:

```
<object property="predefinedCommandName"/>
```

sau

```
<object property="predefinedClassName.predefinedCommandName"/>
```

sau

```
<object property="{x:Static customClassName.customCommandName}"/>
```

unde:

predefinedClassName : clasa comanda predefinita.

predefinedCommandName : comanda predefinita.

customClassName : O clasa custom ce contine comanda custom.

customCommandName O comanda custom.

Pas 1. Definire comanda si instantiere.

```
public static RoutedCommand CustomRoutedCommand = new RoutedCommand();
```

Pas 2. Creare handler pentru executie comanda.

```
private void ExecutedCustomCommand(object sender,
                                   ExecutedRoutedEventArgs e)
{
    MessageBox.Show("Custom Command Executed");
}
// CanExecuteRoutedEventHandler that only returns true if
// the source is a control.
private void CanExecuteCustomCommand(object sender,
                                     CanExecuteRoutedEventArgs e)
{
    Control target = e.Source as Control;
```

```
if(target != null)
{
    e.CanExecute = true;
}
else
{
    e.CanExecute = false;
}
}
```

Pas 3. Se creaza **CommandBinding** ce asociaza comanda cu handler-ul evenimentului. CommandBinding este creat pe un obiect specific. Acest obiect definește domeniul pentru CommandBinding.

```
<Window x:Class="SDKSamples.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:custom="clr-namespace:SDKSamples"
    Height="600" Width="800"
>
<Window.CommandBindings>
    <CommandBinding Command="{x:Static custom:Window1.CustomRoutedCommand}"
        Executed="ExecutedCustomCommand"
        CanExecute="CanExecuteCustomCommand" />
</Window.CommandBindings>
```

sau echivalent cod C#:

```
CommandBinding customCommandBinding = new CommandBinding(
    CustomRoutedCommand, ExecutedCustomCommand,
    CanExecuteCustomCommand);
```

```
// attach CommandBinding to root window
this.CommandBindings.Add(customCommandBinding);
```

Pas 4. Invocare comanda.

O modalitate pentru invocare este de a asocia aceasta cu ICommandSource, cum ar fi Button.

XAML:

```
<StackPanel>
    <Button Command="{x:Static custom:Window1.CustomRoutedCommand}"
        Content="CustomRoutedCommand"/>
</StackPanel>
```

sau echivalent cod C#:

```
// create the ui
```

```
StackPanel CustomCommandStackPanel = new StackPanel();  
Button CustomCommandButton = new Button();  
CustomCommandStackPanel.Children.Add(CustomCommandButton);
```

```
CustomCommandButton.Command = CustomRoutedCommand;
```

```
// End MSDN
```

Asociere shortcut

In ctor clasei scriem urmatorul cod:

```
this.InputBindings.Add(  
new KeyBinding(ApplicationCommands.Help, new KeyGesture(Key.F2)));
```

Pentru a elimina un anumit shortcut la o comanda putem scrie (in ctor):

```
this.InputBindings.Add(  
new KeyBinding(ApplicationCommands.NotACommand, new KeyGesture(Key.F1)));
```

sau in XAML:

```
<Window.InputBindings>  
<KeyBinding Command="Help" Key="F2"/>  
<KeyBinding Command="NotACommand" Key="F1"/>  
</Window.InputBindings>
```

Sa urmarim urmatorul exemplu:

```
<StackPanel  
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
Orientation="Horizontal" Height="25">  
  
<Button Command="Cut" CommandTarget="{Binding ElementName=textBox}"  
Content="{Binding RelativeSource={RelativeSource Self},  
Path=Command.Text}"/>  
  
<Button Command="Copy" CommandTarget="{Binding ElementName=textBox}"  
Content="{Binding RelativeSource={RelativeSource Self},  
Path=Command.Text}"/>  
  
<Button Command="Paste" CommandTarget="{Binding ElementName=textBox}"  
Content="{Binding RelativeSource={RelativeSource Self},  
Path=Command.Text}"/>  
  
<Button Command="Undo" CommandTarget="{Binding ElementName=textBox}"  
Content="{Binding RelativeSource={RelativeSource Self},  
Path=Command.Text}"/>  
  
<Button Command="Redo" CommandTarget="{Binding ElementName=textBox}"  
Content="{Binding RelativeSource={RelativeSource Self},  
Path=Command.Text}"/>  
<TextBox x:Name="textBox" Width="200"/>  
</StackPanel>
```

rezultatul este



ComboBox

Defineste doua evenimente – DropDownOpened si DropDownClosed – si o proprietate IsDropDownOpen.

Proprietatile IsEditable si IsReadOnly permit sau nu permit tastarea de text direct in TextBox-ul atasat ComboBox-ului dand astfel posibilitatea de a selecta mai usor articolele dupa nume. Implicit aceste proprietati sunt setate pe *false*.

Proprietatea StaysOpenOnEdit poate fi setata pe *true* pentru a mentine ComboBox-ul deschis daca utilizatorul face clic pe caseta de selectie.

Exemplu

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <ComboBox IsEditable="True"
    TextSearch.TextPath="Children[1].Children[0].Text">

<!-- Item #1 -->

  <StackPanel Orientation="Horizontal" Margin="5">
  <Image Source="CurtainCall.bmp"/>
  <StackPanel Width="200" >
  <TextBlock Margin="5,0" FontSize="14" FontWeight="Bold"
    VerticalAlignment="center">Curtain Call</TextBlock>

  <TextBlock Margin="5" VerticalAlignment="center" TextWrapping="Wrap">
    Whimsical, with a red curtain background that represents a stage.
  </TextBlock>
  </StackPanel>
  </StackPanel>

<!-- Item #2 -->

  <StackPanel Orientation="Horizontal" Margin="5">
  <Image Source="Fireworks.bmp"/>
  <StackPanel Width="200">
  <TextBlock Margin="5,0" FontSize="14" FontWeight="Bold"
    VerticalAlignment="center">Fireworks</TextBlock>
  <TextBlock Margin="5" VerticalAlignment="center" TextWrapping="Wrap">
    Sleek, with a black sky containing fireworks. When you need to celebrate
    PowerPoint-style, this design is for you!
  </TextBlock>
  </StackPanel>
  </StackPanel>

  </ComboBox>
</Window>
```


ComboBoxItem

Putem pune proprietatea atasata `TextSearch` la nivel de `ComboBoxItem` ca in exemplul de mai jos:

```
<!-- Item #1 -->
<ComboBoxItem TextSearch.Text="Curtain Call">
<StackPanel Orientation="Horizontal" Margin="5">
...
</StackPanel>
</ComboBoxItem>
<!-- Item #2 -->
<ComboBoxItem TextSearch.Text="Fireworks">
<StackPanel Orientation="Horizontal" Margin="5">
...
</StackPanel>
</ComboBoxItem>
```

`ComboBoxItem` expune proprietatile `IsSelected` si `IsHighlighted` si de asemenea evenimentele `Selected` si `Unselected`.

Daca un articol din `ComboBox` este un control (*content control*), acesta nu va fi afisat in caseta de selectie; va fi afisat continutul intern al controlului.

`ComboBoxItem` este un *content control*, si de aceea trebuie sa avem o reprezentare sub forma de string a continutului ce-l vom afisa in caseta de selectie.

```
<ComboBox>
<ComboBoxItem>Item 1</ComboBoxItem>
<ComboBoxItem>Item 2</ComboBoxItem>
</ComboBox>
```

ListBox

Este similar cu `ComboBox`-ul si va afisa toate articolele direct in control (in partea vizibila). Daca nu se pot afisa toate articolele atunci se foloseste navigarea verticala (scrollbar).

Suporta selectie multipla, controlata de proprietatea `SelectionMode` ce accepta urmatoarele valori :

`Single` ;

`Multiple` – articolele selectate sunt adaugate la colectia `SelectedItems` ;

`Extended` – selectie simpla sau multipla (contigua : `Shift + clic mouse`) sau nu (`Ctrl+clic mouse`). Este exact controlul `Listbox` din Win32.

Articolele din `Listbox` sunt derivate din clasa `ListboxItem`.

Proprietatea `TextSearch` se aplica asemanator ca la `ComboBox`.

Cum pot sorta articolele din ListBox (sau orice alt ItemsControl)?

ItemsCollection are o proprietate SortDescriptions ce poate mentine oricate instante dorim ale clasei System.ComponentModel.SortDescription. Fiecare SortDescription descrie ce proprietate a articolelor poate fi utilizata pentru sortare si daca sortarea este crescatoare sau descrescatoare.

Exemplu:

```
// Sterg sortarea existenta
myItemsControl.Items.SortDescriptions.Clear();
// Sortare dupa proprietatea Content
myItemsControl.Items.SortDescriptions.Add(
    new SortDescription("Content", ListSortDirection.Ascending));
```

ListView

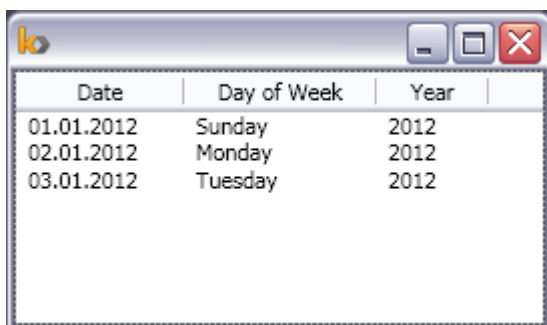
Controlul ListView este derivat din ListBox dar foloseste in plus SelectionMode = Extended.

Proprietatea View permite de a personaliza vizualizarea articolelor.

Proprietatea View este derivata din ViewBase, clasa abstracta.

WPF pune la dispozitie o clasa concreta GridView, asemanatoare ca vizualizare cu un tabel.

```
<ListView>
<ListView.View>
<GridView>
<GridViewColumn Header="Date"/>
<GridViewColumn Header="Day of Week"
    DisplayMemberBinding="{Binding DayOfWeek}"/>
<GridViewColumn Header="Year" DisplayMemberBinding="{Binding Year}"/>
</GridView>
</ListView.View>
<sys:DateTime>1/1/2012</sys:DateTime>
<sys:DateTime>1/2/2012</sys:DateTime>
<sys:DateTime>1/3/2012</sys:DateTime>
</ListView>
```



Date	Day of Week	Year
01.01.2012	Sunday	2012
02.01.2012	Monday	2012
03.01.2012	Tuesday	2012

Codul complet, pentru exemplul de mai sus, este:

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:sys="clr-namespace:System;assembly=mscorlib">
  <ListView>
  <ListView.View>
  <GridView>
  <GridViewColumn Header="Date"/>
  <GridViewColumn Header="Day of Week"
    DisplayMemberBinding="{Binding DayOfWeek}"/>
  <GridViewColumn Header="Year" DisplayMemberBinding="{Binding Year}"/>
  </GridView>
  </ListView.View>
  <sys:DateTime>1/1/2012</sys:DateTime>
  <sys:DateTime>1/2/2012</sys:DateTime>
  <sys:DateTime>1/3/2012</sys:DateTime>
  </ListView>
</Window>
```

Articolele din ListView sunt o lista simpla si ca atare afisarea de date diferite in fiecare coloana se face cu ajutorul proprietatii DisplayMemberBinding din GridViewColumn.

ListView contine un obiect complex pentru fiecare rand, si valoarea pentru fiecare coloana este o proprietate sau subproprietate a fiecarui obiect.

DisplayMemberBinding cere folosirea tehnicii de asociere de date (*data binding*).

DataGrid

Folosit pentru afisari tabelare cu posibilitati de sortare pe fiecare coloana, de redimensionare a coloanelor, etc.

Exemplu

```
public class Record
{
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public Uri Website { get; set; }
  public bool IsBillionaire { get; set; }
  public Gender Gender { get; set; }
}

public enum Gender
{
  Male,
  Female
}
```

Cele cinci coloane sunt definite in colectia Columns din DataGrid

```
<DataGrid IsReadOnly="True"
xmlns:local="clr-namespace:MyNamespace"
xmlns:sys="clr-namespace:System;assembly=mscorlib">
<!-- Support for showing all genders in the DataGridComboBoxColumn: -->
<DataGrid.Resources>
<ObjectDataProvider x:Key="genderEnum" MethodName="GetValues"
ObjectType="{x:Type sys:Enum}">
<ObjectDataProvider.MethodParameters>
<x:Type Type="local:Gender"/>
</ObjectDataProvider.MethodParameters>
</ObjectDataProvider>
</DataGrid.Resources>
<!-- The columns: -->
<DataGrid.Columns>
<DataGridTextColumn Header="First Name" Binding="{Binding FirstName}"/>
<DataGridTextColumn Header="Last Name" Binding="{Binding LastName}"/>
<DataGridHyperlinkColumn Header="Website" Binding="{Binding Website}"/>
<DataGridCheckBoxColumn Header="Billionaire?"
Binding="{Binding IsBillionaire}"/>
<DataGridComboBoxColumn Header="Gender" SelectedItemBinding="{Binding
Gender}"
ItemsSource="{Binding Source={StaticResource genderEnum}}"/>
</DataGrid.Columns>
<!-- The data: -->
<local:Record FirstName="Adam" LastName="Nathan"
Website="http://adamnathan.net" Gender="Male"/>
<local:Record FirstName="Bill" LastName="Gates"
Website="http://twitter.com/billgates" IsBillionaire="True" Gender="Male"/>
</DataGrid>
```

Tipurile de coloane suportate de DataGrid:

DataGridTextColumn — pentru stringuri, se afiseaza intr-un TextBlock pentru reprezentarea normala, si intr-un TextBox pentru editare.

DataGridHyperlinkColumn — afiseaza un link. Trebuie scris cod pentru tratarea actiunii – implicit nu exista acest cod.

DataGridCheckBoxColumn — bun pentru valori de tip bool. Afiseaza un CheckBox.

DataGridComboBoxColumn — bun pentru enumerari. TextBlock pentru afisare normala si ComboBox pentru editare (selectie valori).

DataGridTemplateColumn — Permite un template arbitrar. Se folosesc proprietatile CellTemplate si CellEditingTemplate.

Coloane auto generate

Cand articolele din DataGrid sunt setate folosind ItemsSource, se incearca generarea automata a coloanelor.

DataGridTextColumn - este folosit pentru stringuri.

DataGridHyperlinkColumn – folosit pentru URI.

DataGridComboBoxColumn - folosit pentru enumerari.

DataGridCheckBoxColumn – folosit pentru valori de tip bool.

Exemplu

```
dataGrid.ItemsSource = new Record[]
{
    new Record { FirstName="Adam", LastName="Nathan", Website=
    new Uri("http://adamnathan.net"), Gender=Gender.Male },
    new Record { FirstName="Bill", LastName="Gates", Website=
    new Uri("http://twitter.com/billgates"), Gender=Gender.Male,
    IsBillionaire=true }
};
```

DataGrid suporta editarea pe loc a articolelor din celule.

Selectare randuri si/sau celule

Selectia depinde de valorile proprietatilor SelectionMode (discutata mai sus) si SelectionUnit. SelectionUnit poate avea urmatoarele valori :

Cell—Numai celule individuale pot fi selectate.

FullRow—Numai un rand intreg poate fi selectat.

CellOrRowHeader—Selectie celule sau rand facand clic pe antet rand.

La selectie rand se genereaza evenimentul Selected si se foloseste proprietatea SelectedItems ce va contine articolul selectat.

La selectie celule individuale se genereaza evenimentul SelectedCellChanged si se foloseste proprietatea SelectedCells ce contine o lista de structuri DataGridViewCellInfo.

Celula curenta este data de proprietatea CurrentCell, iar CurrentColumn indica coloana celulei selectate si CurrentItem contine data celulei curente.

DataGrid suporta detalii extinse pentru prezentarea informatiei, acest lucru realizandu-se cu ajutorul proprietatii RowDetailsTemplate :

```
<DataGridView ...>
<DataGridView.RowDetailsTemplate>
<DataTemplate>
<TextBlock Margin="10" FontWeight="Bold">Detalii despre articolul
selectat.</TextBlock>
</DataTemplate>
</DataGridView.RowDetailsTemplate>
...
</DataGridView>
```

Cod complet aplicatie

In MainWindow.xaml

```
<Window x:Class="Wpf_C3.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Wpf_C3"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <DataGrid AutoGenerateColumns="False" Height="191"
HorizontalAlignment="Left" Margin="12,12,0,0"
                Name="dg" VerticalAlignment="Top" Width="433"
                ItemsSource="{Binding}"
                RowHeaderStyle="{Binding ElementName=dg,
                Path=CurrentItem}"
                CanUserSortColumns="False" CanUserAddRows="False"
                SelectionUnit="FullRow" SelectionMode="Single"
                MouseLeftButtonDown="dg_MouseLeftButtonDown"
                PreviewMouseUp="dg_PreviewMouseUp">

<!-- In cod va trebui sa furnizam sursa pentru ItemsSource (vezi codul)-->

                <DataGrid.Resources>
                    <ObjectDataProvider x:Key="genderEnum"
                        MethodName="GetValues"
ObjectType="{x:Type sys:Enum}"
                    >
                        <ObjectDataProvider.MethodParameters>
                            <x:Type Type="local:Gender"/>
                        </ObjectDataProvider.MethodParameters>
                    </ObjectDataProvider>
                </DataGrid.Resources>

                <!-- Detalii randuri -->
                <DataGrid.RowDetailsTemplate>
                    <DataTemplate>
                        <TextBlock Margin="10" FontWeight="Bold"
Text="{Binding Path=FirstName}"></TextBlock>
                    </DataTemplate>
                </DataGrid.RowDetailsTemplate>

                <!-- Coloanele: -->
                <DataGrid.Columns>
                    <DataGridTextColumn Header="First Name"
Binding="{Binding FirstName}"/>
                    <DataGridTextColumn Header="Last Name"
Binding="{Binding LastName}"/>
                    <DataGridHyperlinkColumn Header="Website"
Binding="{Binding Website}"/>
                    <DataGridCheckBoxColumn Header="Billionaire?"
                        Binding="{Binding IsBillionaire}"/>
                    <DataGridComboBoxColumn Header="Gender"
                        SelectedItemBinding="{Binding Gender}"
                        ItemsSource="{Binding
Source={StaticResource genderEnum}"/>
                </DataGrid.Columns>
            </DataGrid>
        </Grid>
    </Window>
```

In MainWindow.xaml.cs

```
namespace Wpf_C3
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        List<Record> lr = new List<Record>();

        public MainWindow()
        {
            InitializeComponent();

            CreareListRecord();
            // In XAML am specificat ItemsSource="{Binding}"
            // Aici ii asociem data
            this.dg.ItemsSource = lr;
        }

        private void CreareListRecord()
        {
            lr.Add(new Record() { FirstName="Iasi 1",
            LastName="Last name 1", IsBillionaire=true,
            Website=new Uri("http://www.infoasi.ro"),
            Gender=Gender.Male});
            lr.Add(new Record() { FirstName="Iasi 2",
            LastName="Last name 2", IsBillionaire=false,
            Website = new Uri("http://www.infoasi.ro"),
            Gender=Gender.Female});
        }

        private void dg_PreviewMouseUp(object sender,
        MouseButtonEventArgs e)
        {
            DataGrid grid = e.Source as DataGrid;
            if (grid == null)
            {
                MessageBox.Show("Grid null");
                return;
            }

            object o1 = grid.SelectedItem;
            Record rec = o1 as Record;
            MessageBox.Show(rec.LastName);
        }
    }
}
```

Meniuri

Menu ContextMenu

Meniurile contin o multime de controale, proiectate pentru a afisa articole sub forma arborescenta.

Menu

Menu aranjeaza articolele orizontal.

```
<Menu>
<MenuItem Header="_File">
<MenuItem Header="_New..." />
<MenuItem Header="_Open..." />
<Separator />
<MenuItem Header="Sen_d To">
<MenuItem Header="Mail Recipient" />
<MenuItem Header="My Documents" />
</MenuItem>
</MenuItem>
<MenuItem Header="_Edit">
...
</MenuItem>
<MenuItem Header="_View">
...
</MenuItem>
</Menu>
```

MenuItem derivat din HeaderedItemsControl. Header este obiectul principal, in mod obisnuit text. Articolele, daca exista, sunt elemente descendente si sunt afisate ca submeniuri.

Proprietati pentru MenuItem:

Icon

IsCheckable

InputGestureText — atasare shortcut.

Evenimentele definite MenuItem:

Checked;

Unchecked;

SubmenuOpened;

SubmenuClosed;

Click.

Se poate atribui o comanda la un MenuItem folosind proprietatea Command.

Observatie

Pentru a atasa un shortcut la un meniu se foloseste proprietatea Command.

Aranjare Menu vertical


```
<Menu>
<Menu.ItemsPanel>
<ItemsPanelTemplate>
<StackPanel/>
</ItemsPanelTemplate>
</Menu.ItemsPanel>
...
</Menu>
```

ContextMenu

Meniul contextual se ataseaza la un control folosind proprietati, cum ar fi ContextMenu definita in FrameworkElement si FrameworkContentElement.

Meniul contextual se activeaza cu clic dreapta mouse sau Shift + F10.

Exemplu

```
<ListBox>
<ListBox.ContextMenu>
    <ContextMenu>
        <MenuItem Header="_File"/>
        <MenuItem Header="_New..."/>
        <MenuItem Header="_Open..."/>
    </ContextMenu>
</ListBox.ContextMenu>
...
</ListBox>
```

TreeView

TreeViewItem

```
<TreeView>
<TreeViewItem Header="Desktop">
<TreeViewItem Header="Computer">
...
</TreeViewItem>
<TreeViewItem Header="Recycle Bin">
...
</TreeViewItem>
<TreeViewItem Header="Control Panel">
<TreeViewItem Header="Programs"/>
<TreeViewItem Header="Security"/>
<TreeViewItem Header="User Accounts"/>
</TreeViewItem>
<TreeViewItem Header="Network">
...
</TreeViewItem>
</TreeViewItem>
</TreeViewItem>
```

Proprietati importante: IsExpanded si IsSelected.

Evenimente: Expanded, Collapsed, Selected si Unselected.

Resurse

WPF suporta doua tipuri distincte de resurse:

resurse binare
resurse logice.

Resurse binare

Pot fi *impachetate* in urmatoarele moduri :

in interiorul unui assembly ;

ca fisiere ce sunt cunoscute de aplicatie in momentul compilarii;

ca fisiere ce pot sa nu fie cunoscute de aplicatie in momentul compilarii.

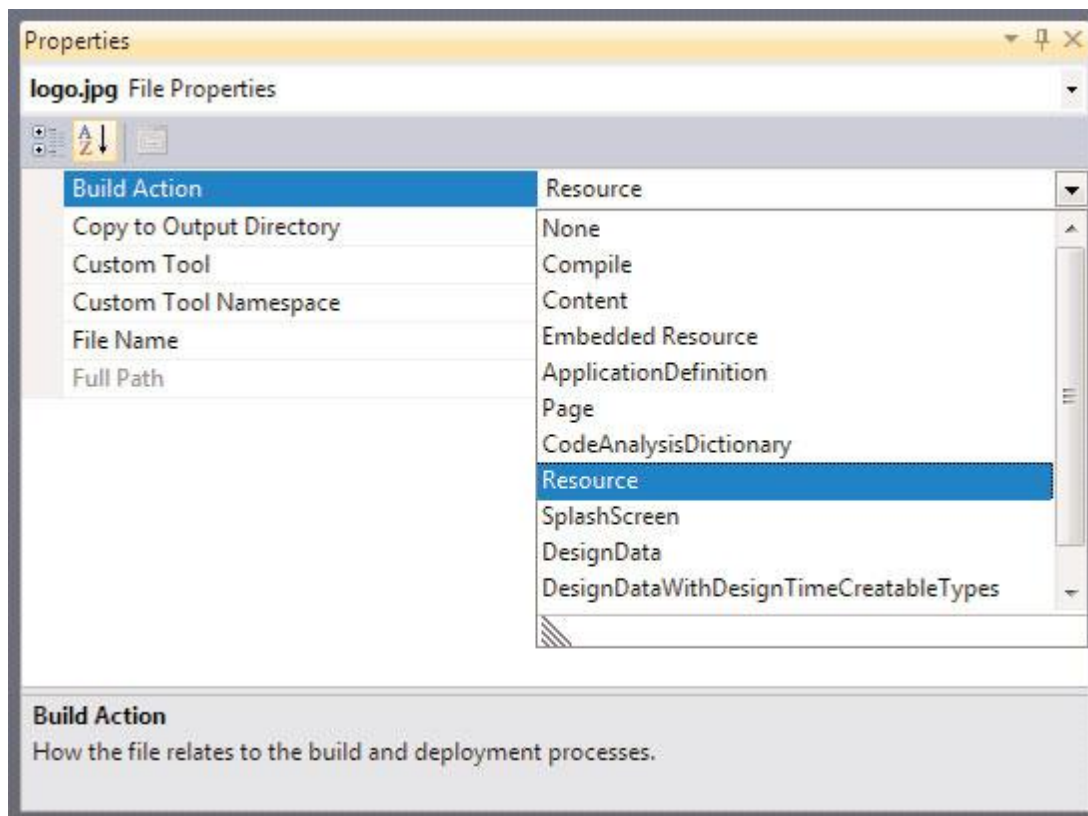
Resursele binare ale unei aplicatii se impart in:

resurse localizabile - resurse ce depind de cultura curenta si trebuie sa se schimbe pentru a fi in concordanta cu acea cultura;

resurse neutre - resurse ce nu depind de nici o cultura.

Definirea resurselor binare

Se adauga un fisier de resurse la proiectul din VS si apoi se construiesc aceasta resursa. Vezi figura.



VS suporta mai multe tipuri de constructie a resurselor binare:

Resource – resursele se regasesc in assembly.

Content – resursele sunt vazute ca un fisier dar adauga un atribut la assembly (AssemblyAssociatedContentFile) si inregistreaza existenta si locatia fisierului.

Accesare resurse binare

Se foloseste URI.

Un converter de tip permite ca acest URI sa fie specificat in XAML ca un string.

```
...  
<Image Height= "32" Source= "img.gif"/>  
...
```

In cadrul proiectului se pot crea directoare in care se pastreaza resursele (pentru o claritate a codului si a organizarii proiectului).

```
<Resource Include="images\logo.jpg"/>
```

sau

```
<Content Include="images\logo.jpg"/>
```

Accesare resurse din cod

```
Image image = new Image();  
image.Source = new BitmapImage(new Uri("pack://application:,,,/logo.jpg"));
```

Utilizarea `pack://application:,,,/` lucreaza numai cu resurse ce apartin la proiectul curent, resurse marcate ca Resource sau Content.

Resurse logice

Resursele logice sunt obiecte .NET memorate in proprietatea Resources a unui element, partajate de mai multe elemente descendente.

FrameworkElement si FrameworkContentElement au proprietatea Resources.

Aceste resurse pot avea stiluri sau furnizori de date.

Exemplu

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
Title="Simple Window" Background="Yellow">
<DockPanel>
<StackPanel DockPanel.Dock="Bottom" Orientation="Horizontal"
HorizontalAlignment="Center">
<Button Background="Yellow" BorderBrush="Red" Margin="5">
<Image Height="21" Source="zoom.gif"/>
</Button>
<Button Background="Yellow" BorderBrush="Red" Margin="5">
<Image Height="21" Source="defaultThumbnailSize.gif"/>
</Button>
<Button Background="Yellow" BorderBrush="Red" Margin="5">
<Image Height="21" Source="previous.gif"/>
```

sau plasandu-le in resursele ferestrei:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Simple Window">
<Window.Resources>
<SolidColorBrush x:Key="backgroundBrush">Yellow</SolidColorBrush>
<SolidColorBrush x:Key="borderBrush">Red</SolidColorBrush>
</Window.Resources>
<Window.Background>
<StaticResource ResourceKey="backgroundBrush"/>
</Window.Background>

<DockPanel>
<StackPanel DockPanel.Dock="Bottom" Orientation="Horizontal"
HorizontalAlignment="Center">
<Button Background="{StaticResource backgroundBrush}"
BorderBrush="{StaticResource borderBrush}" Margin="5">
<Image Height="21" Source="zoom.gif"/>
</Button>
<Button Background="{StaticResource backgroundBrush}"
BorderBrush="{StaticResource borderBrush}" Margin="5">
<Image Height="21" Source="defaultThumbnailSize.gif"/>
...

```

Resurse Statice versus Resurse Dinamice

Resurse statice – StaticResourceExtension – resursa se aplica o singura data.

Resurse dinamice – DynamicResourceExtension, resursele se reaplica ori de cate ori se schimba.

Resurse statice – timp mai mic de executie fata de resursele dinamice.

Resurse la nivel de *fereastră*, pastrate in fisiere XML

Resurse memorate la nivel de *aplicatie* pot fi pastrate intr-un fisier XML separat.

Se foloseste proprietatea MergeDictionaries din clasa ResourceDictionary.

De exemplu, o fereastră Window poate sa isi seteze colectia sa de resurse in dictionare multiple din fisiere XML separate.

```
<Window.Resources>
<ResourceDictionary>
<ResourceDictionary.MergedDictionaries>
<ResourceDictionary Source="file1.xaml"/>
<ResourceDictionary Source="file2.xaml"/>
</ResourceDictionary.MergedDictionaries>
</ResourceDictionary>
</Window.Resources>
```

Fisierele separate trebuie sa foloseasca ResourceDictionary la nivel de element radacina. De exemplu, file1.xml ar putea contine:

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Image x:Key="logo" Source="logo.jpg"/>
</ResourceDictionary>
```

Daca dictionarul are chei duplicate, se ia in considerare ultima cheie procesata.

Resurse fara partajare

Implicit, cand o resursa este aplicata in mai multe locuri, aceeaasi instanta a obiectului este folosita peste tot.

Putem marca articolele dintr-o resursa compilata ca fiind *x:Shared= "False"*, aceasta insemnand ca se va crea o noua instanta a obiectului ce poate fi modificata independent de celelalte instante.

Din acest punct de vedere urmatorul exemplu este corect :

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Window.Resources>
  <Image x:Shared="False" x:Key="zoom" Height="21"
    Source="zoom.gif"/>
  </Window.Resources>
  <StackPanel>
  <!-- Applying the resource multiple times works! -->
  <StaticResource ResourceKey="zoom"/>
  <StaticResource ResourceKey="zoom"/>
  <StaticResource ResourceKey="zoom"/>
  </StackPanel>
</Window>
```

De observat ca x:Shared poate fi folosit numai in fisier XAML compilat. Altfel, acest lucru nu mai functioneaza.

Definire resurse in cod (window este de tip Window)

```
window.Resources.Add("backgroundBrush", new
SolidColorBrush(Colors.Yellow));
window.Resources.Add("borderBrush", new SolidColorBrush(Colors.Red));
```

Pentru resurse statice in cod va trebui sa folosim metoda FindResource (mostenita din FrameworkElement sau FrameworkContentElement).

Urmatorul cod din XAML :

```
<Button Background="{StaticResource backgroundBrush}"  
BorderBrush="{StaticResource borderBrush}"/>
```

este echivalent in cod cu (stackPanel este numele unui StackPanel):

```
Button button = new Button();  
// The Button must descend from the Window before looking up resources:  
stackPanel.Children.Add(button);  
button.Background = (Brush)button.FindResource("backgroundBrush");  
button.BorderBrush = (Brush)button.FindResource("borderBrush");
```

Daca resursa nu poate fi gasita se genereaza o exceptie si ca alternativa putem utiliza metoda `TryFindResource` ce returneaza *null* daca resursa nu a fost gasita.

Pentru *resurse dinamice*, se foloseste metoda `SetResourceReference`.

Urmatorul cod din XAML:

```
<Button Background="{DynamicResource backgroundBrush}"  
BorderBrush="{DynamicResource borderBrush}"/>
```

este echivalent cu :

```
Button button = new Button();  
button.SetResourceReference(Button.BackgroundProperty, "backgroundBrush");  
button.SetResourceReference(Button.BorderBrushProperty, "borderBrush");
```

Data Binding

Data Binding – Introducere

Ce este asocierea de date (Data Binding)?

Data binding este o tehnica generala ce asociaza doua surse de date si mentine sincronizarea datelor.

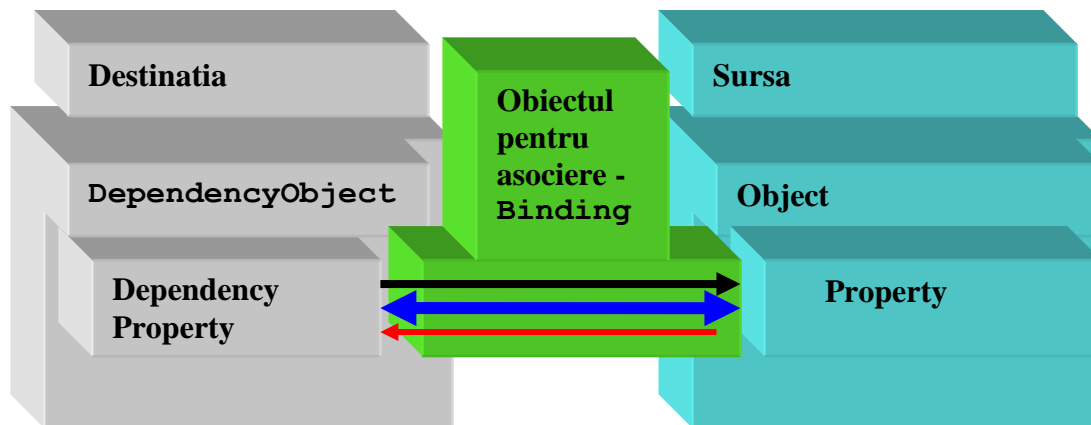
Asocierea de date este un proces ce stabileste o conexiune intre datele afisate in interfata utilizatorului si sursele de date definite in modelul aplicatiei. Asocierea de date presupune existenta sursei (surselor) de date si a elementelor destinatie unde se propaga aceste date. Vezi si directia de transfer a datelor.

In WPF proprietatile dependente ale elementelor pot fi asociate la obiecte CLR (incluzand si obiecte ADO.NET sau obiecte asociate cu servicii Web si proprietati Web) si date XML.

Concepte de baza in asocierea de date - Data Binding

- Directia de transfer a datelor.
- Actualizarea sursei folosind trigger-i.

Schematic asocierea de date se reprezinta ca in figura de mai jos.



Fiecare asociere este compusa din urmatoarele elemente:

1. Un obiect destinatie.
2. Proprietate dependenta in obiectul destinatie (proprietate ce se va actualiza).
3. Un obiect sursa.
4. Calea catre valoarea din obiectul sursa, valoare pe care dorim sa o utilizam.

Observatie

Sagetile din figura de mai sus arata directia de transfer a datelor.

Proprietatea destinatie trebuie sa fie o proprietate **dependenta**. Numai tipurile **DependencyObject** pot defini proprietati dependente.

Obiectul sursa asociat poate fi un obiect **CLR** sau **XAML**.
Stabilirea asocierii se face cu ajutorul unui obiect **Binding**.

Directia de transfer a datelor

Directia de transfer a datelor se precizeaza cu ajutorul proprietatii **Mode** din clasa **Binding** ale carei valori posibile pot fi:

- **OneTime** - o singura data. Destinatie se actualizeaza o singura data.
- **OneWayToSource** - de la destinatie la sursa.
- **OneWay** - de la sursa la destinatie.
- **TwoWay** - bidirectional.

Observatie

Pentru ca modificarile din sursa sa se reflecte in destinatie, sursa trebuie sa implementeze un mecanism de notificare, cum ar fi interfața **INotifyPropertyChanged**.

Un exemplu cu implementarea interfeței **INotifyPropertyChanged** este furnizat în continuare. **PropertyChangedEventArgs** este un tip derivat din **EventArgs**.

```
public class Customer:INotifyPropertyChanged
{
    private string _name;
    /// <summary>
    /// Initalizare instanta a clasei Customer
    /// </summary>
    /// <param name="customername"></param>
    public Customer(string customername)
    {
        _name = customername;
    }

    /// <summary>
    /// get/set name pentru Customer
    /// </summary>
    public string Name
    {
        get { return _name; }
        set
        {
            _name = value;
            OnPropertyChanged("Name");
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
    private void OnPropertyChanged(string name)
    {
        PropertyChangedEventHandler handler = PropertyChanged;
        if (handler != null)
            handler(this, new PropertyChangedEventArgs(name));
    }
}
```

Actualizarea sursei - UpdateSourceTrigger

Proprietatea **UpdateSourceTrigger** din **Binding** determina "momentul de timp" cand se actualizeaza sursa (poate fi de exemplu la parasirea controlului – eveniment **LostFocus** – sau in momentul editarii controlului – **PropertyChanged**- sau Explicit cand aplicatia apeleaza **UpdateSource**, etc.). Exemplu :

```
<Window x:Class="MVVM.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="300" Width="483">
    <StackPanel Orientation="Horizontal" VerticalAlignment="Top">
        <Label Content="Customer name"/>
        <TextBox Name="customer"
            Text="{Binding Customer.Name, UpdateSourceTrigger=PropertyChanged}"
            Width="150" />
        <ComboBox ItemsSource="{Binding Customer.Name}"
            Text="{Binding Customer.Name, UpdateSourceTrigger=PropertyChanged}"
            Width="150" />
        <Button Name="update" Content="Update" Command="{Binding UpdateCommand}"/>
    </StackPanel>
```


</Window>

In acest exemplu proprietatea **Text** din **TextBox** si din **ComboBox** este actualizata cand proprietatea **Name** din **Customer** este actualizata.

Creare binding

Crearea asocierii presupune urmatoarele actiuni:

- Specificare sursa (*Binding Source*).
- Specificare **Path** pentru valoare.
- **Binding** si **BindingExpression** (clase).

1. Specificare sursa

Se pot folosi urmatoarele proprietati:

- **DataContext** - asociem proprietati multiple la aceeasi sursa.
- **Source** – asociere directa.
- **RelativeSource** – se foloseste cand cautam valori in controalele vecine sau pe acelasi nivel in ierarhie (*siblings*) sau in parinte sau in controale descendente, controale ce se gasesc in arborele de vizualizare. Proprietatea **RelativeSource** se foloseste si cand asocierea (binding-ul) este specificata intr-un **ControlTemplate** sau **Style**.
- **ElementName** – accesare sursa prin nume si setare valoare proprietate la controlul destinatie.

Exemplu pentru **ElementName** :

```
<StackPanel>
<Button Content="{Binding ElementName=txtBlock,Path=Text}" />
    <TextBlock x:Name="txtBlock">Hello</TextBlock>
</StackPanel>
```

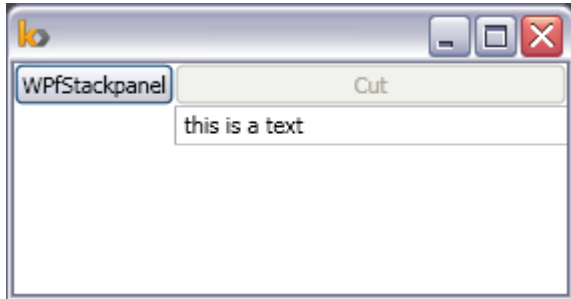
In exemplul de mai sus, continutul butonului – ceea ce vede utilizatorul - va fi dat de valoarea proprietatii **Text** a controlului **TextBlock**.

Exemplu pentru **RelativeSource**

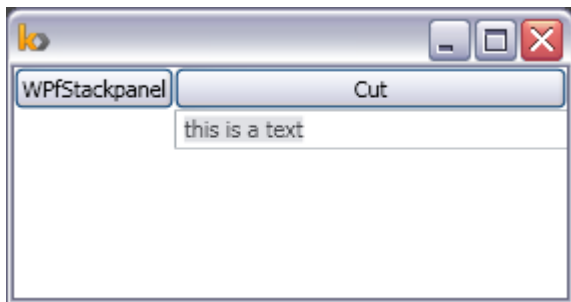
```
<Window
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
<DockPanel>
<!--Continutul butonului va fi numele StackPanel -->
    <StackPanel Name="WPfStackpanel">
        <Button Content="{Binding RelativeSource={RelativeSource
            AncestorType={x:Type StackPanel}}, Path=Name}" />
    </StackPanel>
    <StackPanel Orientation="Vertical">
        <Button Command="ApplicationCommands.Cut"
            CommandTarget="{Binding ElementName=textBox}"
            Content="{Binding RelativeSource={RelativeSource Self},
                Path=Command.Text}"
        />
        <TextBox x:Name="textBox" Width="200" Text="this is a text" />
    </StackPanel>
</Window>
```

```
</StackPanel>  
</DockPanel>  
</Window>
```

Rezultatul



iar dupa selectare text



Se observa butonul *Cut* devine activ si la evenimentul clic va executa comanda "Cut".

DataContext

Proprietatea dependentă **DataContext** este sursa implicită pentru binding și este expusă de clasa de bază **FrameworkElement**. Toți descendenții clasei **FrameworkElement** pot utiliza proprietatea **DataContext** și seta un obiect pentru valoarea acesteia.

DataContext se folosește când asociem proprietăți multiple la aceeași sursă.

DataContext se folosește cu un element UI. Dacă elementul UI este un container, atunci elementele containerului pot folosi același **DataContext** sau pot indica un alt **DataContext**, adică pot suprascrie proprietatea **DataContext** a părintelui. În cazul când un element descendent a suprascris proprietatea **DataContext**, atunci toate elementele copil ale acestuia vor folosi în mod implicit acest **DataContext** dacă nu specificăm altfel. A se vedea și exemplul complet de mai jos – Scenariu : A.InstanceB.Name.

// MSDN – Sintaxa pentru DataContext

Descrierea in XAML folosind **sintaxa elementului proprietate** este:

```
<object>  
  <object.DataContext>  
    <dataContextObject />  
  </object.DataContext>  
</object>
```

dataContextObject

Un obiect incorporat in mod direct, obiect ce serveste drept context de date pentru orice asociere din interiorul elementului parinte. In mod obisnuit, acest obiect este derivat din clasa **Binding** sau **BindingBase** sau poate fi orice tip de obiect CLR.

Descrierea in XAML folosind **atribute**:

```
<object DataContext="bindingUsage"/>  
- sau -  
<object DataContext="{resourceExtension contextResourceKey}"/>
```

bindingUsage

A binding usage that evaluates to an appropriate data context. For details, see [Binding Markup Extension](#).

resourceExtension

One of the following: [StaticResource](#) or [DynamicResource](#). This usage is used when referring to raw data defined as an object in resources. See [XAML Resources](#).

contextResourceKey

The key identifier for the object being requested from within a [ResourceDictionary](#).

// End MSDN

Scenariu –ColorName- :

Consideram ca obiectul sursa este o clasa numita *MyData* ce are o proprietate *ColorName* a carei valoare este setata pe "Red" si dorim sa cream un buton ce are background-ul dat de aceasta proprietate.

Urmatorul cod genereaza un buton cu background-ul rosu.

```
<DockPanel  
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
  xmlns:c="clr-namespace:SDKSample">  
  <DockPanel.Resources>
```

```
<c:MyData x:Key="myDataSource"/>
</DockPanel.Resources>
<DockPanel.DataContext>
    <!-- Asociere directa. Proprietatea Source -->
    <Binding Source="{StaticResource myDataSource}"/>
</DockPanel.DataContext>
<Button Background="{Binding Path=ColorName}"
        Width="150" Height="30">I am bound to be RED!
</Button>
</DockPanel>
```

In acest caz proprietatea destinatie este **Background**, iar proprietatea sursa este *ColorName*. S-a folosit asocierea directa, adica proprietatea **Source**.

Observatie

Proprietatea *ColorName* este de tip string iar proprietatea **Background** este de tip **Brush**. Exista o conversie implicita din string in Brush. Vezi conversia datelor la asocieri.

Sursa este specificata prin setarea proprietatii *DataContext* din elementul *DockPanel*. Butonul mosteneste valoarea *DataContext* din *DockPanel*, care este elementul parinte al butonului in acest caz.

Exista situatii cand specificam asocierea sursei pe destinatii individuale (Vedeti exemplul ce urmeaza).

Scenariu : A.InstanceB.Name

Presupunem ca avem doua clase A si B definite astfel :

```
public class A
{
    public A() { Name = « Clasa A » ; InstanceB = new B() ;}
    public string Name {get ; set ;}
    public B InstanceB {get ; set ;}
}
public class B
{
    public B() { Name = « Clasa B » ;}
    public string Name {get ; set ;}
}
```

Clasele A si B au o proprietate cu acelasi nume – Name - si mai mult A are o proprietate de tip B.

Cum asociem proprietatea Name din A si cum asociem proprietatea Name din B in cadrul aceleasi ferestre ? Dar proprietatea Name din B data de referinta InstanceB ?

Rezolvare.

Constructorul clasei *MainWindow* derivata din *Window* contine si urmatoarele linii de cod :

```
A a = new A() ;
DataContext = a ;
```

adica stabileste contextul obiectului *MainWindow* ca fiind instanta tipului A.

Cod din MainWindow.xaml

```
<Window x:Class="WpfDataContext.MainWindow"
        x:Name="myWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:c="clr-namespace:WpfDataContext"
        Title="MainWindow" Height="350" Width="525">
    <StackPanel>
        <!-- DataContext este setat la A -->

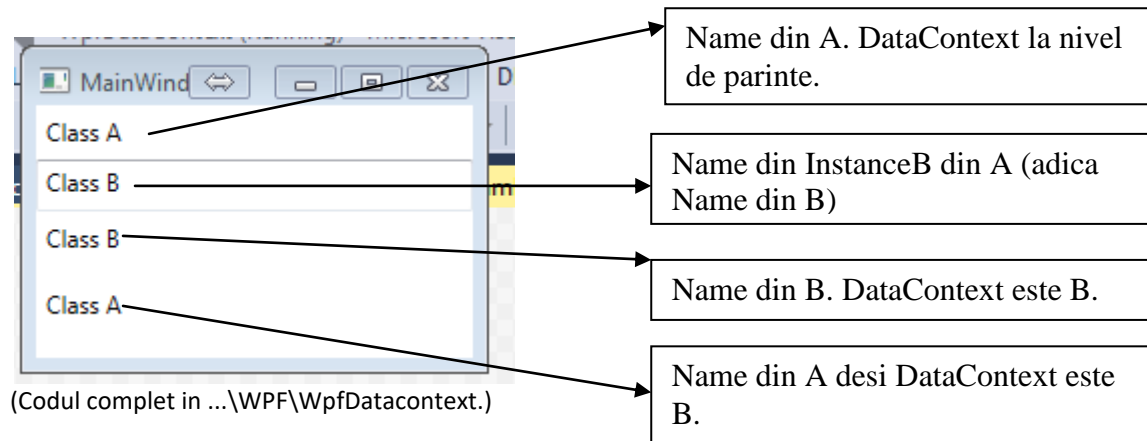
        <!-- DataContext este setat la A, deci vom afisa A.Name -->
        <Label Content="{Binding Path=Name}" Height="44" />
        <!-- Echivalent putem declara si astfel
        <Label Content="{Binding Path=Name}" Height="44" />
        -->
        <!-- TextBox are atasata proprietatea Name din B prin
            intermediul proprietatii InstanceB din clasa A -->
        <TextBox Name="tb1" Text="{Binding Path=InstanceB.Name}" />
        <!-- In aceasta parte de cod DataContext este setat la A -->
        <StackPanel >
            <StackPanel.Resources>
                <c:B x:Key="myDataSource"/>
            </StackPanel.Resources>
            <StackPanel.DataContext>
                <Binding Source="{StaticResource myDataSource}"/>
            </StackPanel.DataContext>

            <!-- DataContext pentru acest StackPanel este B -->
            <!-- DataContext este setat la B, deci vom afisa B.Name -->
            <Label Content="{Binding Name}" Height="60" />
            <!-- DataContext este setat la B, dar facem asocierea la
                conetxtul ferestrei, deci urmatorul Label va folosi A.Name -->
            <Label Content="{Binding ElementName=myWindow,
                Path=DataContext.Name}" Height="46" />
        </StackPanel>
    </StackPanel>
</Window>
```

Codul din MainWindow.xaml.cs

```
public MainWindow()
{
    InitializeComponent();
    A a = new A();
    DataContext = a;
}
```

La executie vom obtine :



2. Source

Proprietatea **Source** returneaza / seteaza obiectul folosit ca sursa la binding.

In asocierea de date (data binding), obiectul sursa binding se refera la obiectul din care obtinem datele.

Daca asociem mai multe proprietati la o sursa comuna, atunci trebuie sa folosim proprietatea **DataContext**.

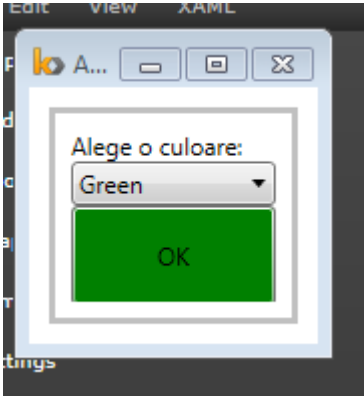
Urmatorul exemplu foloseste **Source** la nivel de control.

Functionalitatea este aceeaasi cu declaratia data mai sus.

```
<DockPanel.Resources>
  <c:MyData x:Key="myDataSource"/>
</DockPanel.Resources>
<Button Width="150" Height="30"
  Background="{Binding Source={StaticResource myDataSource},
    Path=ColorName}">
  I am bound to be RED!
</Button>
```

Daca dorim sa specificam sursa in mod explicit pe binding, avem urmatoarele optiuni.

Property	Description
Source	<p>Setam sursa la o instanta a unui obiect.</p> <p>XAML Property Element Usage</p> <pre><object> <object.Source> <Object .../> </object.Source> </object></pre> <p>XAML Attribute Usage</p> <pre><object Source="object"/></pre>

RelativeSource	<p>Returneaza sau seteaza sursa binding-ului prin specificarea locatiei sale, relativa la pozitia binding-ului destinatie (tinta). Folosita cand dorim sa asociem o proprietate a elementului la o alta proprietate a aceluiasi element sau daca definim un binding intr-un stil sau intr-un template.</p> <pre><Style x:Key="textBoxInError" TargetType="{x:Type TextBox}"> <Style.Triggers> <Trigger Property="Validation.HasError" Value="true"> <Setter Property="ToolTip" Value="{ Binding RelativeSource={x:Static RelativeSource.Self}, Path=(Validation.Errors)[0].ErrorContent }"/> </Trigger> </Style.Triggers> </Style></pre>
ElementName	<p>Specificam un string ce reprezinta un element UI, si ale carui proprietati le asociem altui element.</p> <pre><Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" Width="460" Height="200" Title="Asocierea proprietatilor intre doua controale"> <Border Margin="10" BorderBrush="Silver" BorderThickness="3" Padding="8"> <StackPanel> <TextBlock>Alege o culoare:</TextBlock> <ComboBox Name="myComboBox" SelectedIndex="0"> <ComboBoxItem>Green</ComboBoxItem> <ComboBoxItem>Blue</ComboBoxItem> <ComboBoxItem>Red</ComboBoxItem> </ComboBox> <Button Height="80"> OK <Button.Background> <Binding ElementName="myComboBox" Path="SelectedItem.Content"/> </Button.Background> </Button> </StackPanel> </Border> </Window></pre>  <p>Sursa este proprietatea SelectedItem.Content a elementului numit <i>myComboBox</i>, iar destinatia este proprietatea Background a elementului UI, Button.</p>

Specificare Path catre valoare

Daca sursa este un obiect, putem folosi proprietatea **Path** pentru a specifica valoarea ce o vom folosi in asociere. Preluat din exemplul de mai sus.

```
<Canvas.Background>  
  <Binding ElementName="myComboBox" Path="SelectedItem.Content"/>  
</Canvas.Background>
```

Explicatie

Sursa este proprietatea `SelectedItem.Content` a elementului numit `myComboBox`, iar destinatia este proprietatea `Background` a elementului UI, `Button`.

Cand nu este specificat **Path**, implicit se asociaza obiectul in intregime.

Daca asociem date din XML, trebuie sa folosim proprietatea **XPath** pentru a specifica valoarea. Daca rezultatul returnat de o cerere **XPath** este un **XmlNode** atunci trebuie folosita proprietatea **Path**.

Observatie: binding null

Consideram urmatorul exemplu in care `ListBox` este plasat intr-un `DockPanel`:

```
<ListBox ItemsSource="{Binding}"  
  IsSynchronizedWithCurrentItem="true"/>
```

Deoarece sintaxa folosita pentru binding este vida, `ListBox` mosteneste `DataContext` din `DockPanel`.

Cateva reguli (MSDN) privitoare la binding.

Binding Path Syntax

Use the [Path](#) property to specify the source value you want to bind to:

In the simplest case, the [Path](#) property value is the name of the property of the source object to use for the binding, such as `Path=PropertyName`.

Subproperties of a property can be specified by a similar syntax as in C#. For instance, the clause `Path=ShoppingCart.Order` sets the binding to the subproperty `Order` of the object or property `ShoppingCart`.

To bind to an attached property, place parentheses around the attached property. For example, to bind to the attached property `DockPanel.Dock`, the syntax is `Path=(DockPanel.Dock)`.

Indexers of a property can be specified within square brackets following the property name where the indexer is applied. For instance, the clause `Path=ShoppingCart[0]` sets the binding to the index that corresponds to how your property's internal indexing handles the literal string "0". Nested indexers are also supported.

Indexers and subproperties can be mixed in a **Path** clause; for example, `Path=ShoppingCart.ShippingInfo[MailingAddress,Street]`.

Inside indexers you can have multiple indexer parameters separated by commas (,). The type of each parameter can be specified with parentheses. For example, you can have `Path="[(sys:Int32)42, (sys:Int32)24]"`, where `sys` is mapped to the **System** namespace.

When the source is a collection view, the current item can be specified with a slash (/). For example, the clause `Path=` sets the binding to the current item in the view. When the source is a collection, this syntax specifies the current item of the default collection view.

Property names and slashes can be combined to traverse properties that are collections. For example, `Path=/Offices/ManagerName` specifies the current item of the source collection, which contains an `Offices` property that is also a collection. Its current item is an object that contains a `ManagerName` property.

Optionally, a period (.) path can be used to bind to the current source. For example, `Text="{Binding}"` is equivalent to `Text="{Binding Path=}"`.

Escaping Mechanism

Inside indexers ([]), the caret character (^) escapes the next character.

If you set `Path` in XAML, you also need to escape (using XML entities) certain characters that are special to the XML language definition:

Use `&` to escape the character "&".

Use `>` to escape the end tag ">".

Additionally, if you describe the entire binding in an attribute using the markup extension syntax, you need to escape (using backslash \) characters that are special to the WPF markup extension parser:

Backslash (\) is the escape character itself.

The equal sign (=) separates property name from property value.

Comma (,) separates properties.

The right curly brace (}) is the end of a markup extension.

Default Behaviors

The default behavior is as follows if not specified in the declaration.

A default converter is created that tries to do a type conversion between the binding source value and the binding target value. If a conversion cannot be made, the default converter returns **null**.

If you do not set `ConverterCulture`, the binding engine uses the **Language** property of the binding target object. In XAML, this defaults to "en-US" or inherits the value from the root element (or any element) of the page, if one has been explicitly set.

As long as the binding already has a data context (for instance, the inherited data context coming from a parent element), and whatever item or collection being returned by that context is appropriate for binding without requiring further path modification, a binding declaration can have no clauses at all: `{Binding}`. This is often the way a binding is specified for data styling, where the binding acts upon a collection. For more information, see the "Entire Objects Used as a Binding Source" section in the [Binding Sources Overview](#).

The default `Mode` varies between one-way and two-way depending on the dependency property that is being bound. You can always declare the binding mode explicitly to ensure that your binding has the desired behavior. In general, user-editable control properties, such as `TextBox.Text` and `RangeBase.Value`, default to two-way bindings, whereas most other properties default to one-way bindings.

The default `UpdateSourceTrigger` value varies between `PropertyChanged` and `LostFocus` depending on the bound dependency property as well. The default value for most dependency properties is `PropertyChanged`, while the `TextBox.Text` property has a default value of `LostFocus`.

Clasa Binding. Clasa BindingExpression.

Clasa **Binding** este in topul ierarhiei pentru a declara o asociere intre proprietati si a specifica caracteristicile unei asocieri. Clasa **Binding** furnizeaza acces de nivel inalt la definitia unui binding, ce conecteaza proprietatile obiectelor tinta (destinatie), in mod obisnuit elemente WPF, cu orice sursa de date (baza de date, fisiere XML, orice obiect ce contine date).

Sintaxa XAML

XAML Object Element Usage

<Binding .../>

XAML Attribute Usage

<object property="{Binding declaration}"/>

declaration

Zero or more attribute-assignment clauses separated by commas (,). For more information, see [Binding Markup Extension](#) or [Binding Declarations Overview](#).

Exemple

myDataSource este definita ca o resursa (este un tip CLR) ce are proprietatea *PersonName* (sursa) care se asociaza la proprietatea **Text** (destinatia) a elementului **TextBlock**.

```
<TextBlock Text="{Binding Source={StaticResource myDataSource},  
Path=PersonName}"/>
```

Clasa **BindingExpression** mentine conexiunea intre sursa si destinatie. Contine informatii despre o singura instanta a binding-ului. O asociere (un "binding") contine toate informatiile ce pot fi partajate peste mai multe expresii de asociere.

De exemplu, consideram *myDataObject* ce este o instanta a clasei *MyData*, *myBinding* este obiectul **Binding** sursa si clasa *MyData* defineste o proprietate string numita *MyDataProperty*.

In codul urmator se asociaza continutul unui **TextBox**, numit *myText*, la proprietatea *MyDataProperty*. Proprietatea destinatie este **TextBlock.TextProperty**.

```
// construiesc o noua sursa  
MyData myDataObject = new MyData(DateTime.Now);  
Binding myBinding = new Binding("MyDataProperty");  
myBinding.Source = myDataObject;  
myText.SetBinding(TextBlock.TextProperty, myBinding);
```

Putem folosi obiectul *myBinding* pentru a crea alte asocieri.

Un obiect **BindingExpression** poate fi obtinut ca valoare de retur a apelului metodei **GetBindingExpression** pe obiectul binding.

Exemplu

```
BindingExpression be =  
    itemNameTextBox.GetBindingExpression(TextBox.TextProperty);  
// se actualizeaza sursa  
be.UpdateSource();
```

Folosirea corecta a asocierilor presupune existenta unor metode de conversie dintr-un tip in alt tip. Clasa respectiva trebuie sa fie derivata din interfata **IValueConverter**, interfata ce defineste doua metode : **Convert** si **ConvertBack**.

Vezi exemplul de mai jos in care se asociaza proprietatii **Background** din **Button** proprietatea *ColorName* din tipul *MyData* (am mai discutat acest exemplu).

```
[ValueConversion(typeof(Color), typeof(SolidColorBrush))]  
public class ColorBrushConverter : IValueConverter  
{  
    public object Convert(object value,  
                           Type targetType,  
                           object parameter,  
                           System.Globalization.CultureInfo culture)  
    {  
        Color color = (Color)value;  
        return new SolidColorBrush(color);  
    }  
  
    public object ConvertBack(  
        object value,  
        Type targetType,  
        object parameter,  
        System.Globalization.CultureInfo culture)  
    {  
        return null;  
    }  
}
```

Asocierea colectiilor

Observatie

Controalele ce contin colectii (**ListBox**, **ComboBox**, **TreeView**, **ListView**) folosesc **ItemsControl** pentru a afisa datele din colectie.

Daca asociem un **ItemsControl** la o colectie, se foloseste proprietatea **ItemsSource** din **Binding**. Putem gandi proprietatea **ItemsSource** ca fiind continutul lui **ItemsControl**.

ItemsSource suporta modul **OneWay**.

Data Binding - continuare

Putem seta un **Binding** o data si apoi sincronizarea intre aceste proprietati este facuta automat.

Exemplu. Presupunem ca dorim sa adaugam un TextBlock la o aplicatie ce mentine informatie despre un articol selectat dintr-un alt control, TreeView. La schimbarea selectie in TreeView trebuie sa actualizam continutul TextBlock-ului.

Acest lucru poate fi facut manual la tratarea evenimentului de schimbare a selectiei in TreeView sau poate fi facut automat de aplicatie daca folosim asocierea datei.

TextBlock are numele "currentFolder".

```
<TextBlock x:Name="currentFolder" DockPanel.Dock="Top"
           Background="AliceBlue" FontSize="16" />
```

Manual (cod C#) acest lucru se poate face astfel:

```
void treeView_SelectedItemChanged(object sender,
    RoutedPropertyChangedEventArgs<object> e)
{
    currentFolder.Text = (treeView.SelectedItem as
        TreeViewItem).Header.ToString();
    Refresh();
}
```

Folosind **Binding** putem scrie acest cod in ctor clasei MainWindow:

```
public MainWindow()
{
    InitializeComponent();

    // codul pentru binding
    Binding binding = new Binding();
    // Setare obiect sursa
    binding.Source = treeView;
    // Setare proprietate sursa
    binding.Path = new PropertyPath("SelectedItem.Header");
    // Atasare la proprietatea destinatie
    currentFolder.SetBinding(TextBlock.TextProperty, binding);
}
```

In acest din urma caz nu mai trebuie sa tratam evenimentul **SelectedItemChanged** pentru **TreeView**.

Binding are notiunea de *proprietate sursa* si *proprietate destinatie*. Proprietatea sursa este setata in doi pasi :

1. Atribuire obiect sursa la **Source** (`binding.Source = treeView;`).
2. Atribuire proprietate **Path** via o instanta a **PropertyPath** (`binding.Path = new PropertyPath("SelectedItem.Header");`)

Asocierea la proprietatea destinatie se face prin apelul metodei **SetBinding**, mostenita din FrameworkElements si FrameworkContentElements, ce are ca al doilea parametru instanta clasei PropertyPath.

Observatie

Exista si metoda statica **SetBinding** pentru a realiza acelasi lucru.

```
BindingOperations.SetBinding( currentFolder,  
                             TextBlock.TextProperty,  
                             binding);
```

Avantajul metodei statice este acela ca primul parametru este definit ca **DependencyObject**, deci permite data binding pe obiecte ce nu deriva din FrameworkElement sau FrameworkContentElement.

Eliminare binding

```
BindingOperations.ClearBinding(currentFolder,TextBlock.TextProperty)  
;
```

sau

```
BindingOperations.ClearAllBindings(currentFolder);
```

Folosire binding in XAML

```
<TextBlock x:Name="currentFolder" DockPanel.Dock="Top"  
           Text="{Binding ElementName=treeView, Path=SelectedItem.Header}"  
           Background="AliceBlue" FontSize="16" />
```

Asa arata codul (descrie si in exemplul anterior):

```
Binding binding = new Binding();  
// Setare obiect sursa  
binding.Source = treeView;  
// Setare proprietate sursa  
binding.Path = new PropertyPath("SelectedItem.Header");  
// Atasare la proprietatea destinatie  
currentFolder.SetBinding(TextBlock.TextProperty, binding);
```

Observatie

Binding are si un constructor ce accepta ca parametru Path.

si ca atare putem scrie codul de mai sus (in XAML) astfel:

```
<TextBlock x:Name="currentFolder" DockPanel.Dock="Top"  
           Text="{Binding SelectedItem.Header, ElementName=treeView}"  
           Background="AliceBlue" FontSize="16" />
```

Ambele declaratii din XAML fac acelasi lucru, prima declaratie fiind mai lizibila.

In declaratia

```
<TextBlock x:Name="currentFolder" DockPanel.Dock="Top"  
           Text="{Binding ElementName=treeView, Path=SelectedItem.Header}"  
           Background="AliceBlue" FontSize="16" />
```

se foloseste proprietatea ElementName si nu Source.

Pentru a seta Source in WPF 4 putem scrie :

```
<TextBlock x:Name="currentFolder" DockPanel.Dock="Top"
Text="{Binding Source={x:Reference TreeView}, Path=SelectedItem.Header}"
Background="AliceBlue" FontSize="16" />
```

De observat claritatea acestei declaratii.

RelativeSource - asociere

RelativeSource se refera la un element prin relatia sa cu elementul destinatie.

Atributul implicit pentru RelativeSource este proprietatea Mode. Cateva valori importante:
PreviousData, TemplatedParent, Self, FindAncestor.

Sintaxe diverse pentru sursa

Sursa si destinatia apartin aceluiasi element.

Setare valoare proprietate destinatie la valoarea unei proprietati din cadrul aceluiasi element.

```
{Binding RelativeSource={RelativeSource Self}}
```

Exemplu. Definim un Rectangle pentru care Height sa fie egal cu Width (adica patrat).

```
<Rectangle Fill="Red" Name="rectangle"
Height="100" Stroke="Black"
Canvas.Top="100" Canvas.Left="100"
Width="{Binding ElementName=rectangle,
Path=Height}"/>
```

echivalent

```
<Rectangle Fill="Red" Height="100"
Stroke="Black"
Width="{Binding RelativeSource=
{RelativeSource Self},
Path=Height}"/>
```

Element sursa egal cu elementul destinatie din TemplatedParent
(proprietate) :

```
{Binding RelativeSource={RelativeSource TemplatedParent}}
```

Element sursa egal cu parintele cel mai apropiat al unui tip dat :

```
{Binding RelativeSource={RelativeSource FindAncestor,
AncestorType={x:Type desiredType}}}
```

Element sursa egal al n-lea parinte cel mai apropiat al unui tip
dat :

```
{Binding RelativeSource={RelativeSource FindAncestor,
AncestorType={x:Type desiredType}}}
```

Element sursa egal cu articolul anterior dintr-o colectie de date asociata:

```
{Binding RelativeSource={RelativeSource PreviousData}}
```

RelativeSource este folositor pentru template-urile controalelor.

Asocierea unei proprietati la un alt element fara a furniza numele elementului :

```
<Slider ToolTip="{Binding RelativeSource={RelativeSource Self},  
    Path=Value}"/>
```

Asocierea la proprietati clasice .NET

Presupunem ca avem o colectie de obiecte *Customer*, numita *ListCustomers* si dorim sa asociem la un *Label* numarul elementelor din aceasta colectie. In acest caz putem scrie :

```
<Label x:Name="numItemsLabel"  
Content="{Binding Source={StaticResource ListCustomers}, Path=Count}"  
DockPanel.Dock="Bottom"/>
```

ListCustomers poate fi definita astfel :

```
List<Customers> _listCustomers = new List<Customer>() ;
```

O metoda ce completeaza initial *_listCustomers*, numita *FillListCustomers(List<Customer> param)* si o proprietate publica :

```
public List<Customer> ListCustomers  
{  
    get {return _listCustomers ;}  
    set { _listCustomers.Add(value) ;  
        // S-a omis codul pentru verificarea existentei datei inainte de  
        // actualizare colectie.  
    }  
}
```

Deoarece proprietatile clasice din .NET nu au posibilitatea de notificare a schimbarii valorii, destinatia nu va reflecta intotdeauna realitatea.

Pentru a depasi acest inconvenient obiectul sursa trebuie :
sa implementeze interfata System.ComponentModel.INotifyPropertyChanged ce are un singur eveniment PropertyChanged
sau
sa implementeze evenimentul XXXChanged, unde XXX este numele proprietatii a carei valoare s-a schimbat.

Prima varianta este cea corecta (WPF) si are in vedere dezvoltarile ulterioare ale .NET –ului.

Pentru exemplul de mai sus clasa *Customers* ar trebui sa arate astfel:

```
public class Customers : ObservableCollection<Customer>
{ ... }
```

Asocierea la un intreg obiect

Putem asocia o proprietate destinatie la un intreg obiect, nu numai la o proprietate a acestuia.

```
<Label x:Name="numItemsLabel"
Content="{Binding Source={StaticResource customers}}"
DockPanel.Dock="Bottom"/>
```

unde *customers* este un obiect de tip *Customers*.

Asocierea la o colectie

Proprietatea folosita este *ItemsSource*.

```
<ListBox x:Name="pictureBox"
ItemsSource="{Binding Source={StaticResource customers}}" ...>
...
</ListBox>
```

Colectia trebuie sa implementeze interfata *INotifyCollectionChanged*.
ObservableCollection implementeaza interfetele *INotifyCollectionChanged* si *INotifyPropertyChanged*.

Proprietatea *DisplayMemberPath* folosita pentru a atasa o proprietate la ceea ce afiseaza controlul.

```
<ListBox x:Name="pictureBox" DisplayMemberPath="Name"
ItemsSource="{Binding Source={StaticResource customers}}" ...>
...
</ListBox>
```

Gestionare articole selectate - sincronizare

Proprietatea folosita este *IsSynchronizedWithCurrentItem*.

```
<ListBox IsSynchronizedWithCurrentItem="True" DisplayMemberPath="Name"
ItemsSource="{Binding Source={StaticResource customers}}"></ListBox>

<ListBox IsSynchronizedWithCurrentItem="True" DisplayMemberPath="DateTime"
ItemsSource="{Binding Source={StaticResource customers }}"></ListBox>

<ListBox IsSynchronizedWithCurrentItem="True" DisplayMemberPath="Size"
ItemsSource="{Binding Source={StaticResource customers }}"></ListBox>
```

Cele trei *ListBox*-uri sunt sincronizate.
Atentie: puncteaza la aceeasi sursa de date.

Partajare Source cu DataContext

Situatia cand avem proprietati diferite pe obiectul sursa si acelasi obiect sursa.

In acest caz nu folosim Source sau RelativeSource sau ElementName ci DataContext.

Pentru a proiecta un obiect sursa ce poate folosi DataContext, trebuie sa identificam un element parinte comun si sa setam proprietatea DataContext a acestui element la obiectul sursa.

FrameworkElement si FrameworkContentElement au aceasta proprietate.
Exemplu :

```
<StackPanel DataContext="{StaticResource customers}">
<Label x:Name="numItemsLabel"
Content="{Binding Path=Count}" .../>
...
<ListBox x:Name="pictureBox" DisplayMemberPath="Name"
ItemsSource="{Binding}" ...>
...
</ListBox>
...
</StackPanel>
```

In cod putem scrie (si nu mai scriem in XAML):

```
parent.DataContext = customers;
```

DataTemplate

Data Templating – Introducere

<http://msdn.microsoft.com/en-us/library/ms742521.aspx>

Pentru a intelege ce inseamna "machetarea" datelor (data templating) vom incepe cu un exemplu.

Presupunem ca am definit o clasa *Task* astfel:

```
class Task
{
    public string TaskName;
    public string Description;
    public int Priority;
    public TaskType Homework {get; set;}
}
```

unde TaskType este

```
enum TaskType
{
    Home,
    Work }
```

Construim o aplicatie Windows ce are pe interfata un TextBox in care se afiseaza un string si apoi un ListBox in care se afiseaza obiecte de tip Tasks.

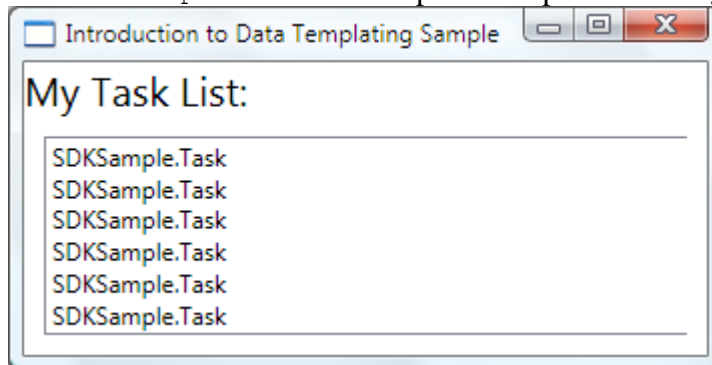
Pentru inceput in XAML scriem (cod partial):

```
<Window x:Class="SDKSample.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:SDKSample"
    Title="Introduction to Data Templating Sample">
    <Window.Resources>
        <local:Tasks x:Key="myToDoList"/>
    ...

    </Window.Resources>
    <StackPanel>
        <TextBlock Name="myTaskText" FontSize="20" Text="My Task List:"/>
        <ListBox Width="400" Margin="10"
            ItemsSource="{Binding Source={StaticResource myToDoList}}"/>
    ...

    </StackPanel>
</Window>
```

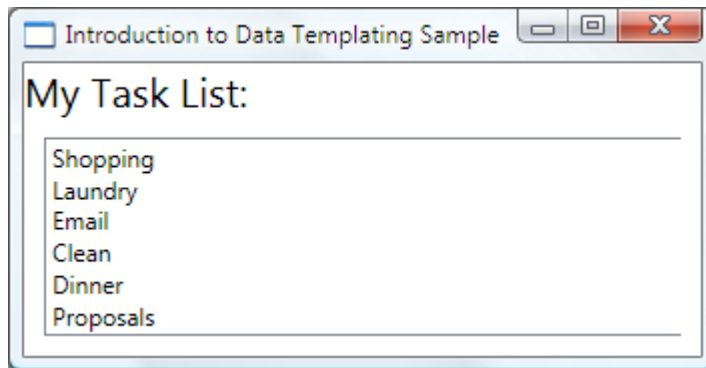
Fara DataTemplate in interfata poate sa apara ceva de genul:



Daca suprascriem metoda ToString() pentru clasa Tasks astfel:

```
public override string ToString()
{
    return name.ToString();
}
```

atunci rezultatul este:



Observatie

Daca asociem date din XML metoda ToString nu mai este disponibila. Ce facem?

Definire DataTemplate

Pentru a defini un DataTemplate (in cazul ListBox), trebuie sa folosim proprietatea ItemTemplate din ListBox.

Ceea ce specificam in DataTemplate devine structura vizuala a obiectului nostru de date. In urmatorul exemplu, dorim ca fiecare articol din ListBox sa apara ca fiind compus din trei TextBox-uri in cadrul unui StackPanel.

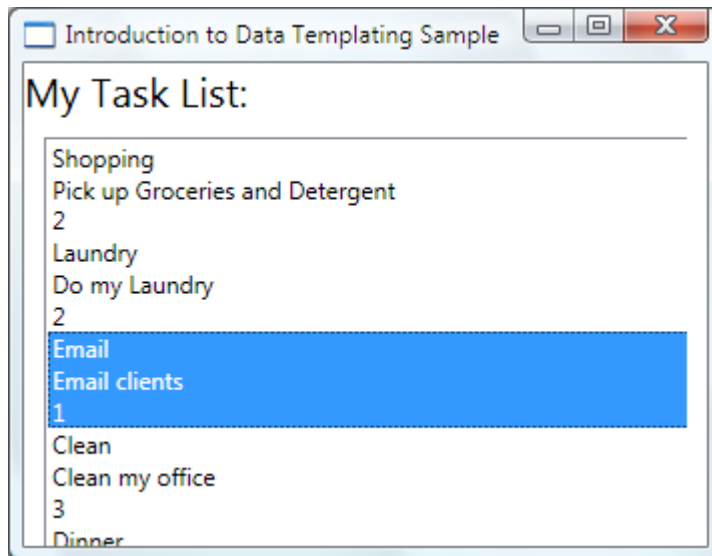
Definim un DataTemplate inline – adica in interiorul definitiei ListBox-ului.

```
<ListBox Width="400" Margin="10"
    ItemsSource="{Binding Source={StaticResource myToDoList}}">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel>
        <TextBlock Text="{Binding Path=TaskName}" />
        <TextBlock Text="{Binding Path=Description}" />
        <TextBlock Text="{Binding Path=Priority}" />
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Observatie

Daca folosim date din XML atunci trebuie sa folosim XPath in loc de Path.

Rezultatul va fi (un articol din ListBox contine trei TextBox-uri):



Crearea unui DataTemplate ca o resursa

Definim DataTemplate in resursele ferestrei:

```
<Window.Resources>
```

...

```
<DataTemplate x:Key="myTaskTemplate">
  <StackPanel>
    <TextBlock Text="{Binding Path=TaskName}" />
    <TextBlock Text="{Binding Path=Description}" />
    <TextBlock Text="{Binding Path=Priority}" />
  </StackPanel>
</DataTemplate>
```

...

```
</Window.Resources>
```

In acest moment putem utiliza *myTaskTemplate* ca o resursa, ca in exemplul urmator:

```
<ListBox Width="400" Margin="10"
  ItemsSource="{Binding Source={StaticResource myToDoList}}"
  ItemTemplate="{StaticResource myTaskTemplate}" />
```

Observatie

Deoarece am definit DataTemplate ca o resursa putem folosi aceasta pe un alt control ce are o proprietate de tip DataTemplate.

Pentru ItemsControl este ItemTemplate.

Pentru ContentControl este ContentTemplate.

Proprietatea DataType

DataType specifica tipul folosit in DataTemplate, asemanator ca la clasa Style. In cazul nostru este tipul *Task*.

```
<DataTemplate DataType="{x:Type local:Task}">
  <StackPanel>
    <TextBlock Text="{Binding Path=TaskName}" />
    <TextBlock Text="{Binding Path=Description}" />
    <TextBlock Text="{Binding Path=Priority}" />
  </StackPanel>
</DataTemplate>
```

Acest template se aplica automat la toate obiectele de tip *Tasks*. In acest caz x :Key este implicita.

Daca suprascriem x :Key, DataTemplate nu se va mai aplica automat la toate obiectele de tip *Tasks*.

Observatie

ContentControl nu foloseste DataTemplate in mod automat, pentru ca are nevoie de informatii privitoare la ceea ce vrem sa asociem din colectie la continutul controlului.

Daca ContentControl foloseste selectia unui tip ItemsControl, atunci putem seta Path la "/" pentru a indica ca suntem interesati de *articolul curent*.

Observatie

DataTemplate poate contine si alte elemente de interfata (Border, Grid, TextBox, etc).

```
<DataTemplate x:Key="myTaskTemplate">
  <Border Name="border" BorderBrush="Aqua" BorderThickness="1"
    Padding="5" Margin="5">
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
      </Grid.ColumnDefinitions>
      <TextBlock Grid.Row="0" Grid.Column="0" Text="Task Name:" />
      <TextBlock Grid.Row="0" Grid.Column="1" Text="{Binding Path=TaskName}" />
      <TextBlock Grid.Row="1" Grid.Column="0" Text="Description:" />
      <TextBlock Grid.Row="1" Grid.Column="1" Text="{Binding Path=Description}" />
      <TextBlock Grid.Row="2" Grid.Column="0" Text="Priority:" />
      <TextBlock Grid.Row="2" Grid.Column="1" Text="{Binding Path=Priority}" />
    </Grid>
  </Border>
</DataTemplate>
```

```
</Border>
```

...

```
</DataTemplate>
```

iar pentru ListBox in XAML putem scrie:

```
<ListBox Width="400" Margin="10"
  ItemsSource="{ Binding Source={ StaticResource myTodoList } }"
  ItemTemplate="{ StaticResource myTaskTemplate }"
  HorizontalContentAlignment="Stretch"/>
```

Folosire DataTriggers pentru a aplica valorile proprietatilor

DataTrigger: reprezinta un trigger ce aplica valoarea proprietatii sau executa actiuni cand data asociata indeplineste anumite conditii specificate.

DataTemplate: descrie structura vizuala a unui obiect.

Proprietatea DataTemplate.Triggers : returneaza o colectie de trigger-i ce aplica valorile proprietatilor sau executa actiuni bazate pe una sau mai multe conditii.

Proprietatea poate fi setata numai in XAML folosind sintaxa pentru colectie sau accesand obiectul din colectie si folosind in continuare una din metodele Add. Proprietatea ce acceseaza obiectul colectie este read-only, iar colectia este read-write.

```
<DataTemplate x:Key="myTaskTemplate">
```

...

```
<DataTemplate.Triggers>
  <DataTrigger Binding="{ Binding Path=TaskType }">
    <DataTrigger.Value>
      <local:TaskType>Home</local:TaskType>
    </DataTrigger.Value>
    <Setter TargetName="border" Property="BorderBrush" Value="Yellow"/>
  </DataTrigger>
</DataTemplate.Triggers>
```

...

```
</DataTemplate>
```

Clasele trigger au proprietatile EnterActions si ExitAction ce permit de a starta o multime de actiuni.

Exista si clasa MultiDataTrigger ce permite sa aplicam modificari ale interfetei bazate pe valori multiple ale proprietatilor si a relatiilor logice dintre acestea (AND , OR).

Alegerea unui DataTemplate bazat pe valoarea unei proprietati

In exemplul cu clasa *Tasks* presupunem ca dorim ca obiectele ce au prioritatea egala cu 1 sa fie afisate altfel fata de celelalte obiecte.

Pentru acest lucru cream un DataTemplate separat pe care-l vom aplica articolelor ce indeplinesc acea conditie.

Cand putem evalua prioritatea unui obiect?
La selectarea articolului din ListBox.

Idee : Va trebui sa gasim o cale de a intercepta acea selectie si apoi sa aplicam template-ul dorit articolului.

Implementare :

Cream o subclasa a clasei DataTemplateSelector si suprascriem metoda SelectTemplate.

Codul din XAML si C# este dat in continuare.

Se creaza un template separat pentru articolele ce au prioritatea 1.
Numele template-ului este *importantTaskTemplate*.

```
<DataTemplate x:Key="importantTaskTemplate">
  <DataTemplate.Resources>
    <Style TargetType="TextBlock">
      <Setter Property="FontSize" Value="20"/>
    </Style>
  </DataTemplate.Resources>
  <Border Name="border" BorderBrush="Red" BorderThickness="1"
    Padding="5" Margin="5">
    <DockPanel HorizontalAlignment="Center">
      <TextBlock Text="{Binding Path=Description}" />
      <TextBlock>!</TextBlock>
    </DockPanel>
  </Border>
</DataTemplate>
```

```
using System.Windows;
using System.Windows.Controls;
```

```
namespace SDKSample
{
  public class TaskListDataTemplateSelector : DataTemplateSelector
  {
    public override DataTemplate
      SelectTemplate(object item, DependencyObject container)
```

```
{
    FrameworkElement element = container as FrameworkElement;

    if (element != null && item != null && item is Task)
    {
        Task taskitem = item as Task;

        if (taskitem.Priority == 1)
            return
                element.FindResource("importantTaskTemplate")
                    as DataTemplate;

        else
            return
                element.FindResource("myTaskTemplate")
                    as DataTemplate;
    }

    return null;
}
}
```

Putem declara *TaskListDataTemplateSelector* ca o resursa:

```
<Window.Resources>

...

<local:TaskListDataTemplateSelector x:Key="myDataTemplateSelector"/>

...

</Window.Resources>
```

si o folosim atribuind-o proprietatii *ItemTemplateSelector* din *ListBox*.
In acest fel *ListBox* apeleaza metoda *SelectTemplate* din clasa *TaskListDataTemplateSelector* pentru fiecare articol ce a fost selectat.

```
<ListBox Width="400" Margin="10"
    ItemsSource="{Binding Source={StaticResource myToDoList}}"
    ItemTemplateSelector="{StaticResource myDataTemplateSelector}"
    HorizontalContentAlignment="Stretch"/>
```

Folosire template-uri de date

Reamintim ca un template de date este o parte a interfeței utilizatorului pe care o aplicăm la un obiect .NET arbitrar, când acesta este afișat.

Multe din controalele din WPF au proprietăți de tipul `DataTemplate` pentru a atașa un template de date.

De exemplu `ContentControl` are proprietatea `ContentTemplate` pentru a controla redarea obiectului sau `Content`, iar `ItemsControl` are proprietatea `ItemTemplate` ce se aplică la fiecare articol.

O listă cu aceste template-uri :

Proprietatea	Clasa
<code>ContentTemplate</code>	<code>ContentControl</code> , <code>ContentPresenter</code> , <code>TabControl</code>
<code>ItemTemplate</code>	<code>ItemsControl</code> , <code>HierarchicalDataTemplate</code>
<code>HeaderTemplate</code>	<code>HeaderedContentControl</code> , <code>HeaderedItemsControl</code> , <code>DataGridRow</code> , <code>DataGridColumn</code> , <code>GridViewColumn</code> ,
<code>GroupStyle</code>	
<code>SelectedContentTemplate</code>	<code>TabControl</code>
<code>DetailsTemplate</code>	<code>DataGridRow</code>
<code>RowDetailsTemplate</code>	<code>DataGrid</code>
<code>RowHeaderTemplate</code>	<code>DataGrid</code>
<code>ColumnHeaderTemplate</code>	<code>GridView</code> , <code>GridViewHeaderRowPresenter</code>
<code>CellTemplate</code>	<code>DataGridTemplateColumn</code> , <code>GridViewColumn</code>
<code>CellEditingTemplate</code>	<code>DataGridTemplateColumn</code>

Setând una din aceste proprietăți la o instanță a `DataTemplate`, se creează un nou arbore de vizualizare.

`DataTemplate` derivă din `FrameworkTemplate`, și ca atare are proprietatea `VisualTree` ce poate fi setată la un arbore arbitrar de elemente `FrameworkElement`.

Este ușor de folosit în XAML și dificil în cod.

Să încercăm să folosim un `DataTemplate` cu o colecție *customers*.

```
<ListBox x:Name="pictureBox"
ItemsSource="{Binding Source={StaticResource customers}}" ...>
<ListBox.ItemTemplate>
  <DataTemplate>
    <Image Source="photo.jpg" Height="35"/>
  </DataTemplate>
</ListBox.ItemTemplate>
...
</ListBox>
```

Când aplicăm acest template la un `Item`, contextul este dat de articolul curent din `ItemsSource`.

Se va afișa aceeași imagine pentru toate articolele.

Putem actualiza acest template astfel pentru a afișa imagine diferită pentru fiecare articol :

```
<ListBox x:Name="pictureBox"
ItemsSource="{Binding Source={StaticResource customers}}" ...>
<ListBox.ItemTemplate>
<DataTemplate>
<Image Source="{Binding Path=FullPath}" Height="35"/>
</DataTemplate>
</ListBox.ItemTemplate>
...
</ListBox>
```

Selectori Template

Uneori este de dorit sa personalizam un template pe baza datelor de intrare.
Acest lucru poate fi facut in cod, unde putem selecta orice template dorim si chiar sa cream unul nou inainte ca datele sa fie afisate (interfata actualizata).

Pentru acest lucru cream o clasa ce va fi derivata din `DataTemplateSelector` si va suprascrie metoda virtuala `SelectTemplate`.

Se poate asocia o instanta cu elementul dorit prin setarea proprietatii `XXXTemplateSelector` a acelui element.

Fiecarei proprietati `XXXTemplate` (vezi tabelul) ii corespunde o proprietate `XXXTemplateSelector`.

Proprietate

Clase

<code>ContentTemplateSelector</code>	<code>ContentControl</code> , <code>ContentPresenter</code> , <code>TabControl</code>
<code>ItemTemplateSelector</code>	<code>ItemsControl</code> , <code>HierarchicalDataTemplate</code>
<code>HeaderTemplateSelector</code>	<code>HeaderedContentControl</code> , <code>HeaderedItemsControl</code> , <code>DataGridView</code> ,
<code>DataGridColumn</code> , <code>GridViewColumn</code> , <code>GroupStyle</code>	
<code>SelectedContentTemplateSelector</code>	<code>TabControl</code>
<code>DetailsTemplateSelector</code>	<code>DataGridView</code>
<code>RowDetailsTemplateSelector</code>	<code>DataGrid</code>
<code>RowHeaderTemplateSelector</code>	<code>DataGrid</code>
<code>ColumnHeaderTemplateSelector</code>	<code>GridView</code> , <code>GridViewHeaderRowPresenter</code>
<code>CellTemplateSelector</code>	<code>DataGridTemplateColumn</code> , <code>GridViewColumn</code>
<code>CellEditingTemplateSelector</code>	<code>DataGridTemplateColumn</code>

Furnizori de date

WPF contine doua clase ce furnizeaza asociieri de date:

`XmlDataProvider` si
`ObjectDataProvider`.

Observatie

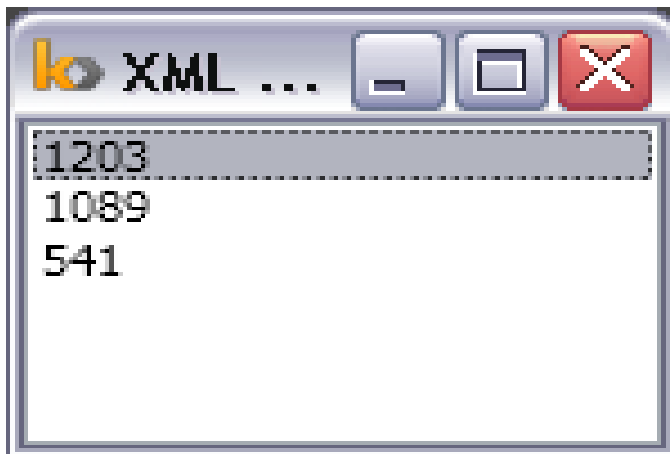
În cadrul furnizorilor de date se încadrează și LINQ cu variantele sale.

XmlDataProvider

Această clasă oferă o posibilitate de a asocia date cu un XML (fișier sau în memorie).

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="XML Data Binding">
<Window.Resources>
<XmlDataProvider x:Key="dataProvider" XPath="GameStats">
<x:XData>
<GameStats xmlns="">
<!-- One stat per game type -->
<GameStat Type="Beginner">
<HighScore>1203</HighScore>
</GameStat>
<GameStat Type="Intermediate">
<HighScore>1089</HighScore>
</GameStat>
<GameStat Type="Advanced">
<HighScore>541</HighScore>
</GameStat>
</GameStats>
</x:XData>
</XmlDataProvider>
</Window.Resources>
<Grid>
<ListBox ItemsSource="{Binding Source={StaticResource dataProvider},
XPath=GameStat/HighScore}" />
</Grid>
</Window>
```

Rezultatul este



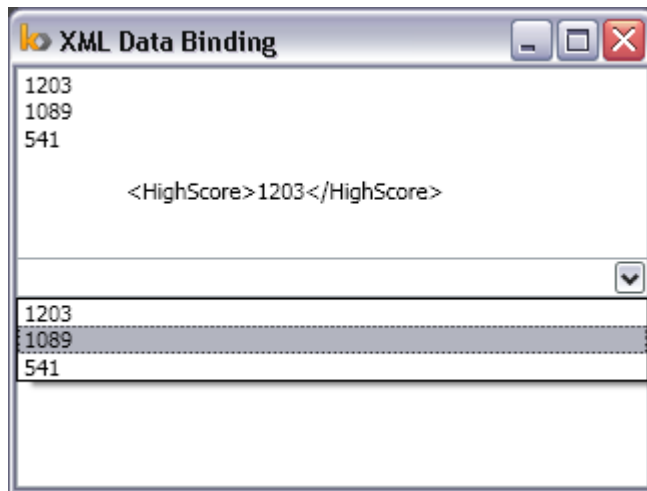


Figura de mai sus are in spate urmatorul cod (s-a adaugat un Label si un ComboBox):

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="XML Data Binding">
<Window.Resources>
<XmlDataProvider x:Key="dataProvider" XPath="GameStats">
<x:XData>
<GameStats xmlns="">
<!-- One stat per game type -->
<GameStat Type="Beginner">
<HighScore>1203</HighScore>
</GameStat>
<GameStat Type="Intermediate">
<HighScore>1089</HighScore>
</GameStat>
<GameStat Type="Advanced">
<HighScore>541</HighScore>
</GameStat>
</GameStats>
</x:XData>
</XmlDataProvider>
</Window.Resources>
<Grid>
<ListBox ItemsSource="{Binding Source={StaticResource dataProvider},
XPath=GameStat/HighScore}" />

<ComboBox ItemsSource="{Binding Source={StaticResource
dataProvider},XPath=GameStat/HighScore}" Height="20"/>

<Label Margin="50" Content="{Binding Source={StaticResource dataProvider},
XPath=GameStat/HighScore, Path=OuterXml}" />

</Grid>
</Window>
```

Observatie

In acest caz se foloseste proprietatea XPath si nu Path.

Daca XML este in fisier atunci trebuie sa indicam acel fisier astfel:

```
<XmlDataProvider x:Key="dataProvider" XPath="GameStats"
Source="GameStats.xml" />
```

ObjectDataProvider

ObjectDataProvider expune obiecte .NET ca o sursa de date.

Acesta permite urmatoarele :
de a instantia obiectul sursa cu un ctor cu parametri, obiect declarat in XAML ;
de a asocia o metoda pe obiectul sursa ;
optiuni pentru asociere data in mod asincron.

Folosire ctor parametrizat in XAML

```
<Window.Resources>  
<local:Customers x:Key="customers"/>  
<ObjectDataProvider x:Key="dataProvider"  
ObjectInstance="{StaticResource customers}"/>  
</Window.Resources>
```

In acest caz, cand asociem la *customers* sau la *dataProvider* obtinem acelasi rezultat.

Cazul cand ctor cere un int:

Colectia este instantiata intern de catre ObjectDataProvider.

```
<ObjectDataProvider x:Key="dataProvider"  
ObjectType="{x:Type local:Customers}">  
<ObjectDataProvider.ConstructorParameters>  
<sys:Int32>23</sys:Int32>  
</ObjectDataProvider.ConstructorParameters>  
</ObjectDataProvider>
```

Asociere metoda

```
<ObjectDataProvider x:Key="dataProvider"  
ObjectType="{x:Type local:Customers}"  
MethodName="GetFolderName"/>
```

Daca e nevoie de a pasa parametri la metoda, se foloseste proprietatea MethodParameters.

Pentru a asocia metoda la un control putem scrie :

```
<TextBlock Text="{Binding Source={StaticResource dataProvider} }"/>
```

Observatie: *dataProvider* este definit mai sus.

Stiluri

Un stil este reprezentat de clasa System.Windows.Style.
Asemantor ca functionalitate cu CSS si HTML.

Permite partajarea proprietatilor, resurselor si a metodelor atasate evenimentelor intre instante ale unui tip.

Putem seta un Style pe orice element ce deriva din FrameworkElement sau FrameworkContentElement. Cel mai adesea un stil este declarat in interiorul unei sectiuni Resources.

Declararea unui stil consta dintr-un obiect Style ce contine o colectie de unul sau mai multe obiecte Setter. Fiecare Setter consta dintr-o proprietate – Property - si o valoare - Value.

Proprietatea este numele proprietatii elementului pentru care se aplica stilul.

Exemple (fara stiluri) :

```
<StackPanel Orientation="Horizontal">
<Button FontSize="22" Background="Purple" Foreground="White"
Height="50" Width="50" RenderTransformOrigin=".5,.5">
<Button.RenderTransform>
<RotateTransform Angle="10"/>
</Button.RenderTransform>
1
</Button>
<Button FontSize="22" Background="Purple" Foreground="White"
Height="50" Width="50" RenderTransformOrigin=".5,.5">
<Button.RenderTransform>
<RotateTransform Angle="10"/>
</Button.RenderTransform>
2
</Button>
<Button FontSize="22" Background="Purple" Foreground="White"
Height="50" Width="50" RenderTransformOrigin=".5,.5">
<Button.RenderTransform>
<RotateTransform Angle="10"/>
</Button.RenderTransform>
3
</Button>
</StackPanel>
```

Folosind stiluri acest lucru poate fi rescris astfel:

```
<StackPanel Orientation="Horizontal">
<StackPanel.Resources>
<Style x:Key="buttonStyle">
<Setter Property="Button.FontSize" Value="22"/>
<Setter Property="Button.Background" Value="Purple"/>
<Setter Property="Button.Foreground" Value="White"/>
<Setter Property="Button.Height" Value="50"/>
<Setter Property="Button.Width" Value="50"/>
<Setter Property="Button.RenderTransformOrigin" Value=".5,.5"/>
<Setter Property="Button.RenderTransform">
<Setter.Value>
<RotateTransform Angle="10"/>
</Setter.Value>
</Setter>
</Style>
</StackPanel.Resources>
<Button Style="{StaticResource buttonStyle}">1</Button>
<Button Style="{StaticResource buttonStyle}">2</Button>
```

```
<Button Style="{StaticResource buttonStyle}">3</Button>  
</StackPanel>
```

Trigger-i

Contin o colectie de Setter-i care se aplica conditional.

Exista trei tipuri de triggeri :

Property triggers— invocat cand valoarea proprietatii dependente se schimba.

Data triggers— invocat cand valoarea unei proprietati .NET se schimba.

Event triggers— invocat cand un eveniment este generat de FrameworkElement, Style, DataTemplate si ControlTemplate.

Clasa Trigger.

O proprietate trigger executa o colectie de Setter-i cand o proprietate specificata are o anumita valoare. Cand proprietatea nu mai are acea valoare se face "rollback", se revine la starea anterioara.

```
<Style x:Key="buttonStyle" TargetType="{x:Type Button}">  
  <Style.Triggers>  
    <Trigger Property="IsMouseOver" Value="True">  
      <Setter Property="RenderTransform">  
        <Setter.Value>  
          <RotateTransform Angle="10"/>  
        </Setter.Value>  
      </Setter>  
      <Setter Property="Foreground" Value="Black"/>  
    </Trigger>  
  </Style.Triggers>  
  <Setter Property="FontSize" Value="22"/>  
  <Setter Property="Background" Value="Purple"/>  
  <Setter Property="Foreground" Value="White"/>  
  <Setter Property="Height" Value="50"/>  
  <Setter Property="Width" Value="50"/>  
  <Setter Property="RenderTransformOrigin" Value=".5,.5"/>  
</Style>
```

Data Triggers

Pot fi generate de orice proprietate .NET.

Trebuie adaugat un obiect DataTrigger la colectia de Triggers si de specificat perechea cheie/valoare.

Proprietatea trebuie specificata cu un Binding.

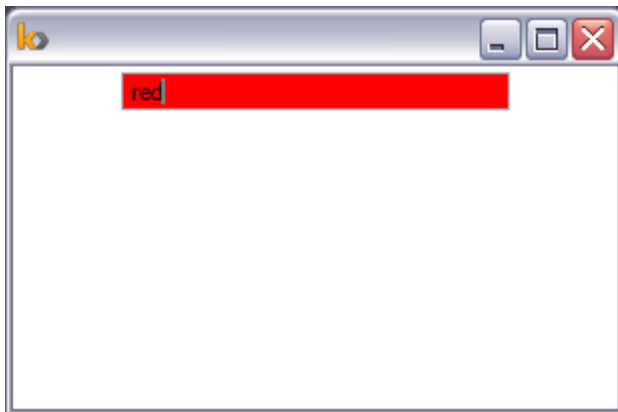
```
<StackPanel Width="200">  
  <StackPanel.Resources>  
    <Style TargetType="{x:Type TextBox}">  
      <Style.Triggers>  
        <DataTrigger  
          Binding="{Binding RelativeSource={RelativeSource Self}, Path=Text}"
```

```
Value="disabled">
<Setter Property="IsEnabled" Value="False"/>
</DataTrigger>
</Style.Triggers>
<Setter Property="Background"
Value="{Binding RelativeSource={RelativeSource Self}, Path=Text}"/>
</Style>
</StackPanel.Resources>
<TextBox Margin="3"/>
</StackPanel>
```

Acest cod produce:



Daca tastam red in TextBox obtinem



Mai multe triggere

OR

```
<Style.Triggers>
<Trigger Property="IsMouseOver" Value="True">
<Setter Property="RenderTransform">
<Setter.Value>
<RotateTransform Angle="10"/>
</Setter.Value>
</Setter>
<Setter Property="Foreground" Value="Black"/>
</Trigger>
<Trigger Property="IsFocused" Value="True">
<Setter Property="RenderTransform">
<Setter.Value>
<RotateTransform Angle="10"/>
</Setter.Value>
```



```
</Setter>  
<Setter Property="Foreground" Value="Black"/>  
</Trigger>  
</Style.Triggers>
```

AND

Se foloseste MultiTrigger sau MultiDataTrigger.

```
<Style.Triggers>  
  <MultiTrigger>  
    <MultiTrigger.Conditions>  
      <Condition Property="IsMouseOver" Value="True"/>  
      <Condition Property="IsFocused" Value="True"/>  
    </MultiTrigger.Conditions>  
    <Setter Property="RenderTransform">  
      <Setter.Value>  
        <RotateTransform Angle="10"/>  
      </Setter.Value>  
    </Setter>  
    <Setter Property="Foreground" Value="Black"/>  
  </MultiTrigger>  
</Style.Triggers>
```

Template

Un template permite de a inlocui un element din arborele de vizualizare cu orice altceva.

Vizualizarile implicite pentru fiecare control din WPF sunt definite in template-uri.

Codul sursa este complet separat, pentru fiecare control, de reprezentarea arborelui vizual.

Template-urile pentru Control sunt reprezentate de clasa ControlTemplate, derivata din FrameworkTemplate.

Alte clase DataTemplate pentru obiecte .NET si ItemsPanelTemplate atribuita la ItemsControl din ItemPanel.

ControlTemplate contine proprietatea VisualTree ce contine arborele de elemente ce defineste modul de afisare al controalelor.

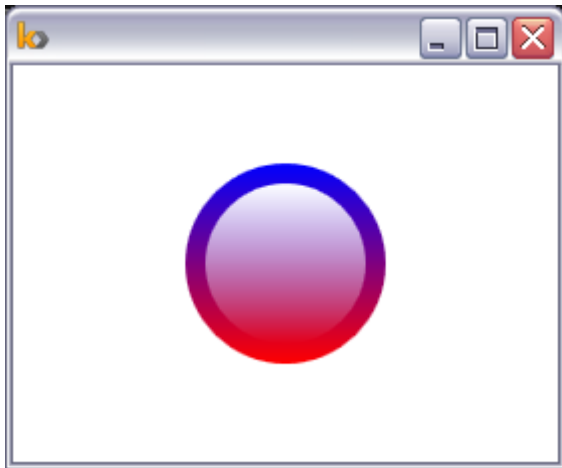
Dupa ce definim un ControlTemplate (in XAML), il putem atasa la orice Control sau Page prin setarea proprietatii Template.

Exemplul urmatoar defineste un template ce il aplica la un Button.

```
<Grid>  
<Grid.Resources>  
  <ControlTemplate x:Key="buttonTemplate">  
    <Grid>  
      <Ellipse Width="100" Height="100">  
        <Ellipse.Fill>
```

```
<LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
<GradientStop Offset="0" Color="Blue"/>
<GradientStop Offset="1" Color="Red"/>
</LinearGradientBrush>
</Ellipse.Fill>
</Ellipse>
<Ellipse Width="80" Height="80">
<Ellipse.Fill>
<LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
<GradientStop Offset="0" Color="White"/>
<GradientStop Offset="1" Color="Transparent"/>
</LinearGradientBrush>
</Ellipse.Fill>
</Ellipse>
</Grid>
</ControlTemplate>
</Grid.Resources>
<Button Template="{StaticResource buttonTemplate}">OK</Button>
</Grid>
```

si rezultatul este, iar butonul are evenimentul Click (este o instanta a clasei Button):



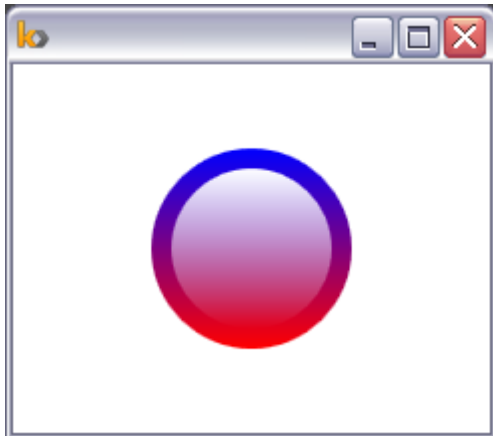
Interactiunea cu trigger-i

Templates poate contine toate tipurile de triggeri intr-o colectie de Triggers.
In exemplul urmator se adauga o colectie de triggeri pentru exemplul prezentat mai sus.

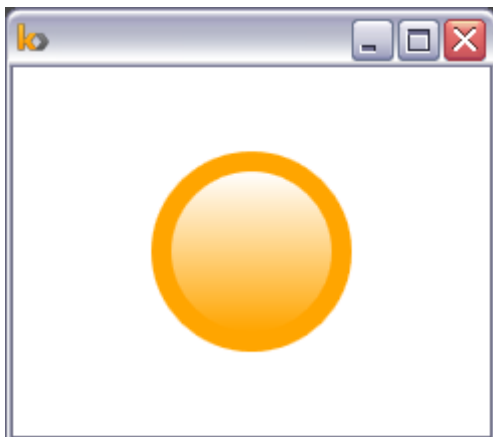
```
<Grid>
<Grid.Resources>
<ControlTemplate x:Key="buttonTemplate">
<Grid>
<Ellipse x:Name="outerCircle" Width="100" Height="100">
<Ellipse.Fill>
<LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
<GradientStop Offset="0" Color="Blue"/>
<GradientStop Offset="1" Color="Red"/>
</LinearGradientBrush>
</Ellipse.Fill>
</Ellipse>
<Ellipse Width="80" Height="80">
```

```
<Ellipse.Fill>
<LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
<GradientStop Offset="0" Color="White"/>
<GradientStop Offset="1" Color="Transparent"/>
</LinearGradientBrush>
</Ellipse.Fill>
</Ellipse>
</Grid>
<ControlTemplate.Triggers>
<Trigger Property="Button.IsMouseOver" Value="True">
<Setter TargetName="outerCircle" Property="Fill" Value="Orange"/>
</Trigger>
<Trigger Property="Button.IsPressed" Value="True">
<Setter Property="RenderTransform">
<Setter.Value>
<ScaleTransform ScaleX=".9" ScaleY=".9"/>
</Setter.Value>
</Setter>
<Setter Property="RenderTransformOrigin" Value=".5,.5"/>
</Trigger>
</ControlTemplate.Triggers>
</ControlTemplate>
</Grid.Resources>
<Button Template="{StaticResource buttonTemplate}">OK</Button>
</Grid>
```

si rezultatul este:



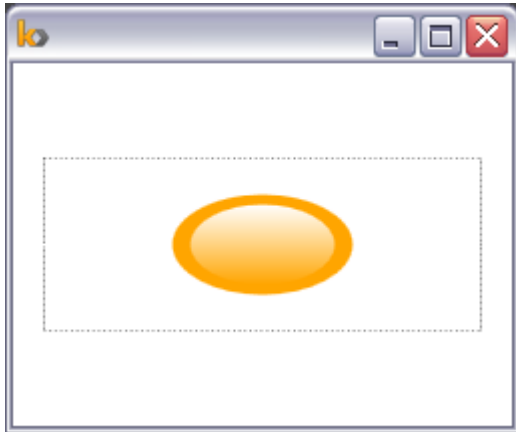
iar cu mouse-ul deasupra butonului, devine:



Analizati codul.

Cand se face clic pe acest buton, se aplica o transformare ce-l face mai mic folosind factorul de scalare 0.9 pe ambele axe.

Daca folosim factorul de scalare de 0.5 pe axa Y atunci la clic pe buton acesta arata astfel :



Proprietatea TargetType din ControlTemplate restrictioneaza actiunea template-ului; furnizeaza elementul destinatie pe care se aplica acesl template.

