

# **Documentatie de laborator**

# **Capitolul I**

## **NOTIUNI DE WPF**

**(notiuni teoretice si exemple  
de aplicatii)**

Sursa: <http://www.wpf-tutorial.com/>

## **INTRODUCERE**

WPF este un framework grafic pentru realizarea de aplicatii folosind .NET Framework.

Exista mai multe framework-uri pentru realizarea de GUI (Graphic User Interface), dar sub .NET Framework, cele mai importante sunt Windows Forms si WPF.

WPF a aparut sub .NET Framework 3.0 si pune la dispozitie o serie de clase si assembly-uri care permit scrierea de cod mult mai usor si mult mai flexibil. Spre exemplu, foloseste tehnologia de randare grafica Direct3D care permite redarea de grafica 3D in aplicatii in care performanta este un element cheie, asa cum e cazul jocurilor video. (Pentru detalii suplimentare vezi <https://en.wikipedia.org/wiki/Direct3D>) Astfel, componentele reprezentate in ferestrele aplicatiei vor avea contururi mult mai line, putand solicita astfel si resursele hardware ale calculatorului. In cazul in care se foloseste traditionalul Windows Forms Application nu vor fi solicitate performante grafice avansate, acesta fiind mult mai inefficient decat un WPF Application.

WPF introduce un concept nou, si anume acela de „template” care permite redefinirea controalelor, dar despre care vom vorbi mai tarziu.

WPF este mai nou decat Windows Form si prezinta cateva avantaje fata de acesta:

- Este mai nou si mai in ton cu noile standarde de programare
- Este mai flexibil, deci se pot realiza mai multe lucruri cu el fara a crea sau cumpara controale noi
- Partea de cod XAML pe care o foloseste WPF-ul face editarea interfetei grafice mult mai usoara si permite separarea codului pe care il scrie programatorul (C# in cazul nostru) de codul pe care il scrie designer-ul (XAML)
- Legarea datelor de controale (data binding) permite o separare mai buna a datelor de interfata grafica (de layout-ul aplicatiei)
- Foloseste si performantele resurselor hardware pentru desenarea de GUI, pentru a avea performante mai ridicate (are inclus un motor care identifica ce nivel de accelerare hardware este suportat de catre sistem si isi ajusteaza singur performantele in functie de acesta). Astfel, impactul cel mai mare il au Video RAM-ul, calcularea efectelor per pixeli, aplicarea texturilor multiple, etc
- Permite realizarea de interfete grafice atat pentru aplicatii desktop cat si pentru aplicatii web

Dupa cum se poate subantelege din cele precizate anterior, WPF este o combinatie de cod XAML cu cod C# sau VB.NET sau orice alt cod al unui limbaj suportat de catre .NET.

Pentru scrierea codului vom folosi IDE-ul (Integrated Development Environment) Visual Studio, cel folosit de majoritatea programatorilor .NET. Acesta se poate descarca de pe Microsoft Imagine (in varianta Profesional, Premium sau Ultimate) sau se poate descarca gratuit de pe site-ul Microsoft varianta Express, care contine mai putine functionalitati decat celelalte variante, dar este suficient pentru dezvoltarea de aplicatii simple pentru laborator.

Puteti descarca versiunea de Visual Studio 2015 sau o versiune mai noua (spre exemplu, versiunea Visual Studio 2019 aparuta in iunie 2018).

XAML vine de la **eXtensible Application Markup Language** si reprezinta varianta Microsoft, cu ajutorul careia se realizeaza interfete grafice, a XML-ului clasic. Acest limbaj este folosit pentru specificarea si setarea de caracteristici pentru clase. Cu alte cuvinte, se vor putea seta variabile, se pot defini proprietati pentru clase si folosi ulterior in aplicatie (in codul .cs) Parser-ul XAML va converti codul in mod automat si va crea obiecte in momentul rularii aplicatiei. Cu XAML se pot realiza doar aspectele vizuale ale aplicatiei, nu si partea de logica a aplicatiei. Pentru aceasta este nevoie de un limbaj de programare (cum ar fi C# sau VB.NET)

In varianta de Windows Form, GUI-ul se crea neaparat folosind limbajul de programare in care se dezvolta aplicatia (C#, VB.NET, etc); dar folosind XAML, realizarea interfetei grafice seamana mai mult cu realizarea unei interfete grafice cu HTML. Intr-o aplicatie realizata folosind WPF, pagina sau fereastră este compusa din doua parti: partea de interfata, cu toate elementele sale, in fisierul XAML si partea de Code Behind (fisierul cu extensia .cs sau .vb) care trateaza evenimentele si care poate manipula codul XAML.

## **DISPATCHER THREAD SI THREAD AFFINITY**

In momentul in care se ruleaza o aplicatie WPF se creaza in mod automat doua fire de executie unul este „Rendering Thread” care ii este ascuns programatorului si nu poate fi accesat sau modificat de catre acesta din aplicatie, iar celalalt este „Dispatcher Thread” care cuprinde toate elementele interfetei grafice. Se poate spune astfel ca dispatcher-ul este firul de executie grafic, cel care leaga toate elementele create folosind WPF. Faptul ca WPF solicita legarea tuturor componentelor grafice in dispatcher mai este numita si „Thread Affinity”. Exista astfel clasa **Dispatcher** care se ocupa de aceasta afinitate catre firul de executie grafic. Aceasta contine o bucla prin care sunt canalizate toate elementele de interfata

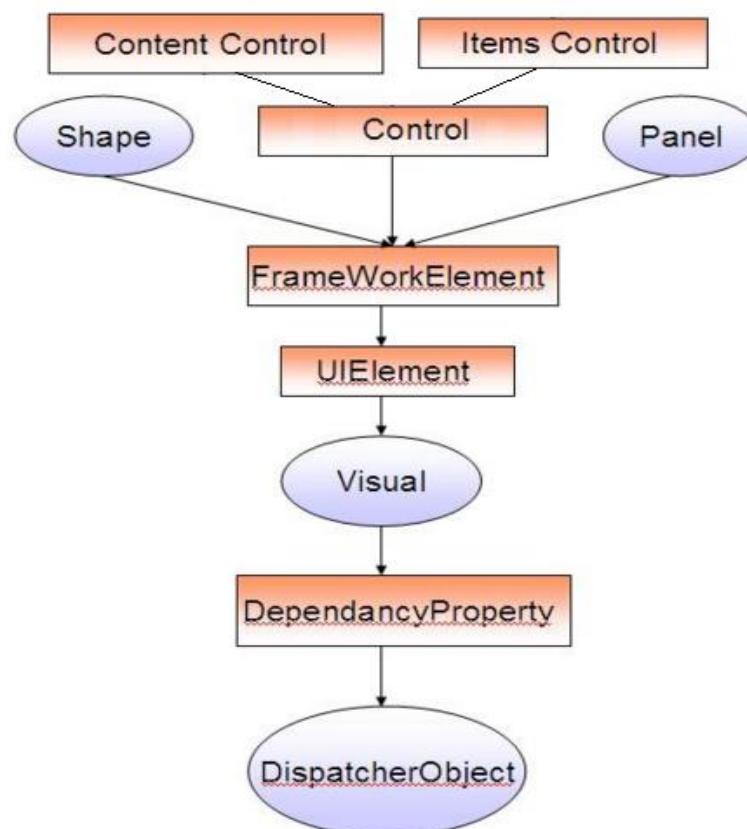
grafica. Fiecare element grafic este derivat din **DispatcherObject** care definește o proprietate numită Dispatcher, care indică spre firul de execuție grafic. Astfel, dacă dintr-un alt fir de execuție se dorește accesarea vreunei componente grafice, se va apela firul Dispatcher. Principala atribuție a lui DispatcherObject este aceea de a verifica dacă firul de execuție are acces la obiect.

### **FOLOSIREA LUI „DependencyObject”**

Orice control WPF este derivat din **DependencyObject**. Aceasta este o clasă care suportă **DependencyProperty** (o proprietate sistem oferită de către WPF). Cum toate componentele grafice sunt derivate din clasa DependencyObject, prin urmare acestea moștenesc comportamente și proprietăți ale acestei clase, cum ar fi asocierea cu un trigger de evenimente, proprietăți de asociere, animații, etc.

### **IERARHIA DE OBIECTE**

Există câteva obiecte care definesc un control WPF și sunt prezentate în desenul de mai jos. Clasele abstracte sunt desenate într-un oval, iar cele concrete în dreptunghi.

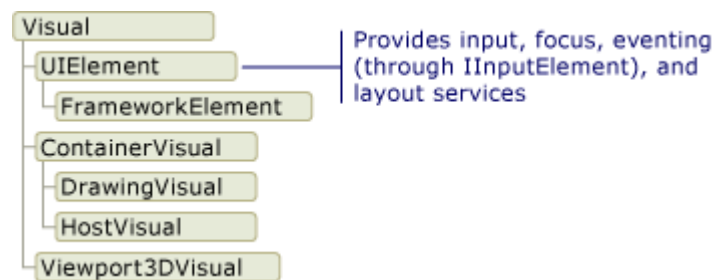


De jos in sus, avem urmatoarea ierarhie:

1. **DispatcherObject** - este parintele tuturor controalelor WPF; este clasa care are grija de firul de executie grafic

2. **DependencyObject** - este clasa care construiește “observatorul” pentru DependencyProperty

3. **Visual** - este legatura dintre DependencyObject si UIElements. Acesta clasa reprezinta abstractizarea de baza pentru toate obiectele de tip “interfata grafica”. Imaginea de mai jos prezinta ierarhia obiectelor vizuale oferite de arhitectura WPF:



Pentru detalii vezi

[https://msdn.microsoft.com/en-us/library/system.windows.media.visual\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.windows.media.visual(v=vs.110).aspx)

4. **UIElement** - creaza suportul pentru comportamente WPF ca: layout-uri, evenimente, inputuri

5. **FrameworkElement** - implementeaza interfata UIElement

6. **Shape** - clasa de baza pentru toate formele de baza

7. **Panel** – clasa de baza pentru toate tipurile de paneluri din WPF

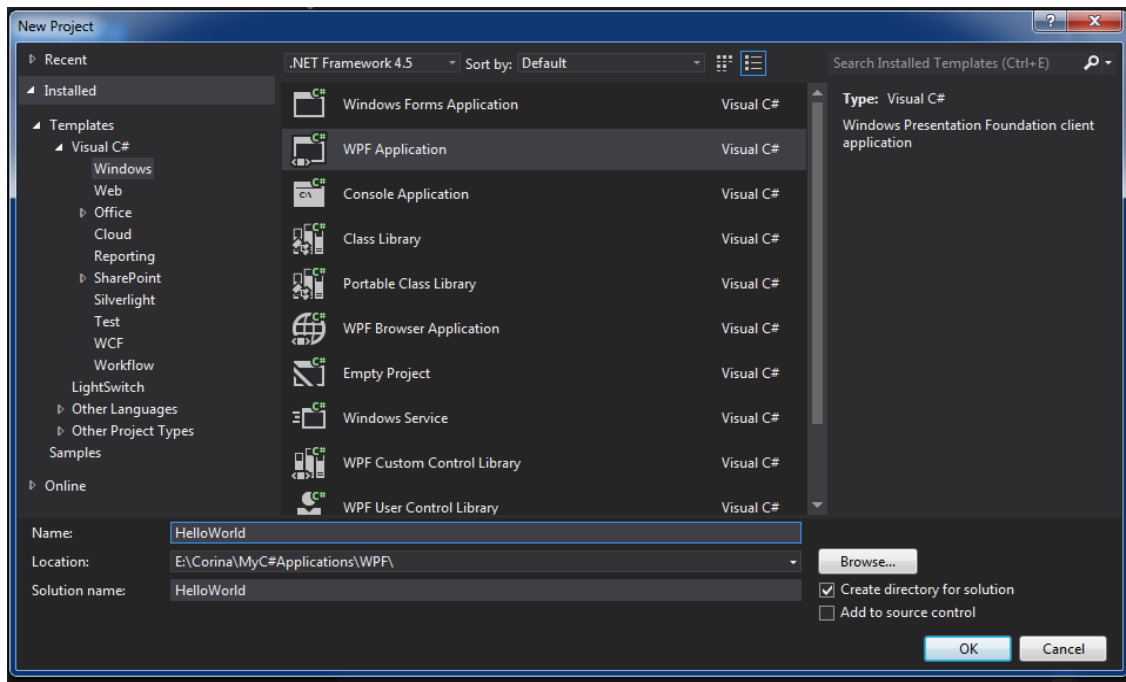
8. **Control** - elementul grafic care interactioneaza cu utilizatorul. Controalelor li se pot aplica template-uri pentru schimbarea infatisarii

9. **ContentControl** - clasa de baza pentru toate controalele care au continut singular: TextBlock, Label, etc...

10. **ItemsControl** - clasa de baza pentru toate controalele care permit afisarea unei colectii: ListView, ComboBox, etc...

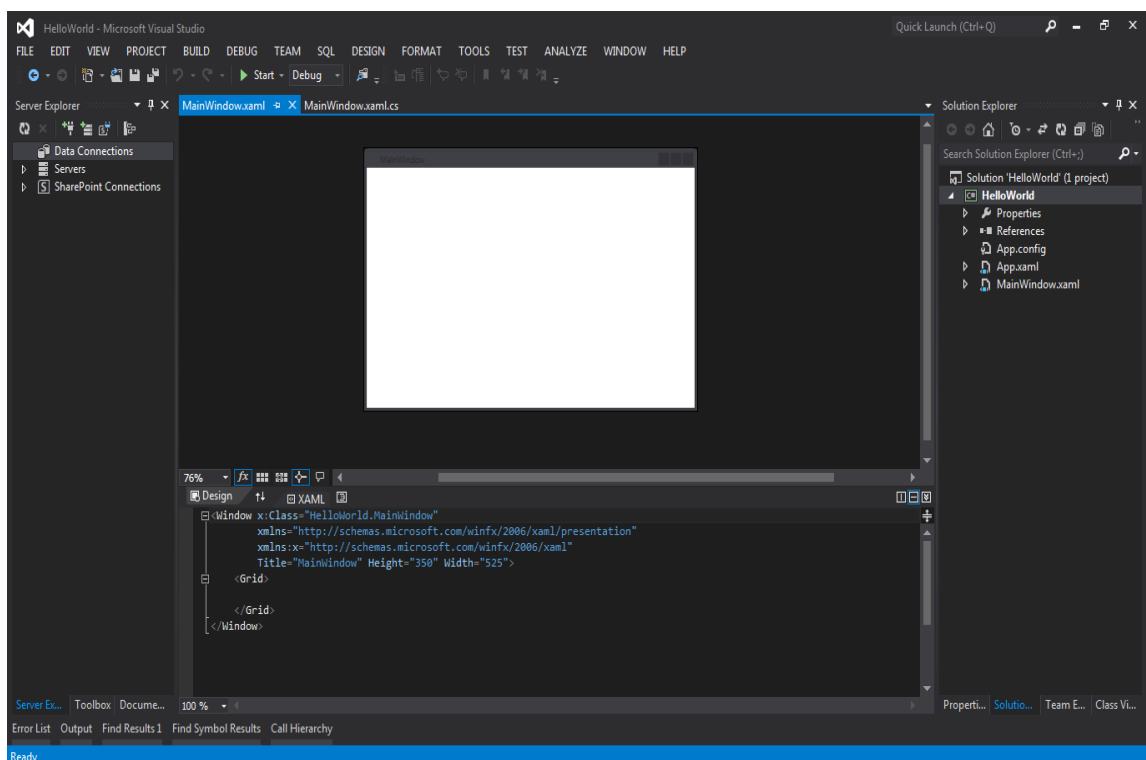
## **PRIMA APLICATIE WPF**

In fereastra de Visual Studio selectam din meniul de sus **File -> New Project**. Va apare o fereastra ca cea din imaginea de mai jos.



Alegem din partea stanga **Templates -> Visual C# -> Windows**

Selectam din partea centrala a ferestrei tipul proiectului – **WPF Application** – si apoi alegem calea unde vom salva proiectul si ii dam un nume (**Ex: HelloWorld**) in casetele de text din partea de jos a ferestrei, unde se cer aceste informatii.



Va aparea o fereastră ca în imaginea de mai sus, unde putem observa fișierul **MainWindow.xaml**. Aceasta este fereastra principală care se va încărca la lansarea în

executie a aplicatiei (Acest lucru se poate modifica prin schimbarea valorii atributului **StartupUri** din fisierul **App.xaml**, pe care il putem observa in fereastra **Solution Explorer**).

In continuare vom afisa un mesaj in fereastra prin adaugarea unei componente **TextBlock**, careia ii vom seta 3 attribute: **VerticalAlignment**, **HorizontalAlignment** si **FontSize**. Aceasta componenta se va introduce in interiorul grid-ului din fereastra **MainWindow.xaml**, iar codul va arata asa:

```
<Window x:Class="HelloWorld.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">
    <Grid>
        <TextBlock VerticalAlignment="Center" HorizontalAlignment="Center" FontSize="50">
            Hello world!!!
        </TextBlock>
    </Grid>
</Window>
```

Dupa executia programului (butonul F5) va aparea o fereastra pe care va scrie „Hello world!!!”

## INTRODUCERE IN XAML

Dupa cum am vazut in exemplul anterior, crearea unui control in XAML se face prin scrierea numelui acestuia in limba engleza intre paranteze ascutite. De exemplu, pentru a crea un buton vom scrie `<Button>`. Acesta este un marcator (tag) XAML. Marcatorii trebuiesc intotdeauna inchisi. Inchiderea unui marcator se face fie prin adaugarea unui slash(/) la finalul marcatorului de inceput, inaintea parantezei ascutite (**Exemplu:** `<Button />`), fie prin adaugarea unui marcator final, diferit de marcatorul de inceput (**Exemplu:** `<Button> </Button>`).

Unele controale permit scrierea de text intre marcatorul de inceput si cel final (**Exemplu:** `<Button> Click me</Button>`)

Controalelor li se pot seta si attribute. Acestea corespund proprietatilor controlului (spre exemplu ale obiectului de tip Button)

Pentru a seta textul de pe un buton si grosimea scrisului vom scrie:

```
<Button FontWeight="Bold" Content="Click me" />
```

Proprietatea **FontWeight** indica grosimea scrisului, iar proprietatea **Content** indica textul scris pe buton.



Atributele unui control mai pot fi scrise si sub forma de marcatori „fiu” pentru marcatorul cu numele controlului. Diferenta este doar de sintaxa, nu apar si functionalitati diferite in cele doua cazuri de scriere.

```
<Button>
    <Button.FontWeight>Bold</Button.FontWeight>
    <Button.Content>Click me</Button.Content>
</Button>
```

Daca dorim sa avem text de diferite culori pe un buton, atunci vom scrie urmatorul cod:

```
<Button>
    <Button.FontWeight>Bold</Button.FontWeight>
    <Button.Content>
        <WrapPanel>
            <TextBlock Foreground="Red">R</TextBlock>
            <TextBlock Foreground="Green">G</TextBlock>
            <TextBlock Foreground="Blue">B</TextBlock>
        </WrapPanel>
    </Button.Content>
</Button>
```

In exemplul de mai sus am folosit un element de tip `<WrapPanel>` deoarece `<Button.Content>` nu permite decat un singur marcator fiu, si noi aveam nevoie de 3 `TextBlock`-uri, fiecare cu cate o culoare. Asa ca le-am inclus pe toate 3 pe un singur astfel de panel. Prescurtat, am fi putut scrie astfel:

```
<Button FontWeight="Bold">
    <WrapPanel>
        <TextBlock Foreground="Red">R</TextBlock>
        <TextBlock Foreground="Green">G</TextBlock>
        <TextBlock Foreground="Blue">B</TextBlock>
    </WrapPanel>
</Button>
```

Exact acelasi lucru il putem realiza si folosind cod C#, dar scriind mai mult cod. Acest lucru se poate face si in cazul unei aplicatii WPF, dar mai cu seama s-ar fi facut daca doream o aplicatie de tip `WindowsForm`.

```
Button btn = new Button();
btn.FontWeight = FontWeight.Bold;
```

```
WrapPanel pnl = new WrapPanel();
TextBlock txt = new TextBlock();
txt.Text = "R";
txt.Foreground = Brushes.Red;
pnl.Children.Add(txt);
txt = new TextBlock();
txt.Text = "G";
txt.Foreground = Brushes.Green;
pnl.Children.Add(txt);
txt = new TextBlock();
txt.Text = "B";
txt.Foreground = Brushes.Blue;
pnl.Children.Add(txt);
btn.Content = pnl;
mainPanel.Children.Add(btn);
```

## **EVENIMENTE IN XAML**

Ca majoritatea framework-urilor pentru GUI, si WPF-ul permite tratarea de evenimente. Aceasta inseamna ca toate controalele, inclusiv Window (care este derivat din clasa Control) dispun de o serie de evenimente la care pot subscrie. Subscrierea la evenimente inseamna ca aplicatia va fi notificata atunci cand se petrec aceste evenimente, iar controlul asupra caruia se produc, trebuie sa raspunda.

Sunt mai multe tipuri de evenimente care se pot produce, dar cele mai frecvent intalnite sunt acelea care raspund la interactiunea utilizatorului cu mouse-ul sau cu tastatura. La majoritatea controalelor vom intalni evenimente ca: **KeyDown** (apasarea unei taste), **KeyUp** (eliberarea unei taste), **MouseDown** (apasarea butonului stang al mouse-ului), **MouseUp** (eliberarea butonului stang al mouse-ului), **MouseEnter** (cursorul mouse-ului intra peste un control), **MouseLeave** (cursorul mouse-ului paraseste un control), **Click**, etc.

Pentru a lega evenimentul unui control de o metoda din Code Behind (codul din spatele ferestrei) va trebui sa setam marcatorului cu numele controlului un atribut cu numele evenimentului, atribut a carei valoare va fi data de numele metodei care se va executa.

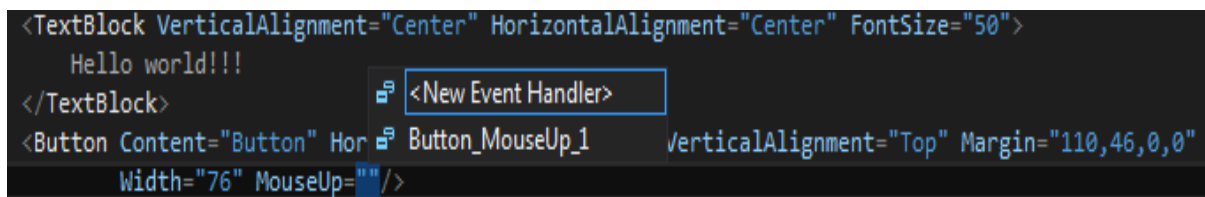
Spre exemplu, pentru a face sa apara o fereastră cu un mesaj la click pe un buton vom scrie urmatorul cod in XAML: `<Button Content="Click me" Click="btnClick" />`, unde `btnClick` este numele metodei din clasa C# din spatele ferestrei curente.

**Atentie!** Metoda trebuie sa respecte o anumita semnatura.

```
private void btnClick(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Hello World!!!");
}
```

Evenimentul **Click** foloseste un delegat numit **RoutedEventHandler** la care va subscrie butonul. Acesta are 2 parametri: „**sender**”, care este de tip **object** si reprezinta controlul asupra caruia s-a produs evenimentul si „**e**” care este de tip **RoutedEventArgs** si care contine informatii utile cu privire la evenimentul produs.

Atunci cand scriem o metoda care trateaza un eveniment, trebuie sa avem grija ce delegat vom folosi. Spre exemplu evenimentul **Click** foloseste delegatul **RoutedEventHandler**, evenimentele **MouseUp** si **MouseDown** folosesc delegatul **MouseButtonEventHandler**, iar evenimentul **MouseMove** foloseste delegatul **MouseEventHandler**. VisualStudio ofera sprijin in acest sens, intrucat daca scriem numele evenimentului asociat controlului in XAML, IntelliSense-ul pus la dispozitie va face sa apara o fereastră de unde putem alege optiunea `<New Event Handler>`, iar VisualStudio va genera automat metoda cu parametrii potriviti in fisierul cu codul C#. Metoda generata automat va avea un nume implicit de forma **NumeControl\_NumeEveniment(lista parametri)**. Acest lucru poate fi vizualizat in imaginea de mai jos.



Daca dorim sa facem subscrierea la un eveniment folosind doar codul C#, atunci va trebui sa folosim operatorul „+=” pentru a adauga *handlerul de evenimente* direct obiectului.

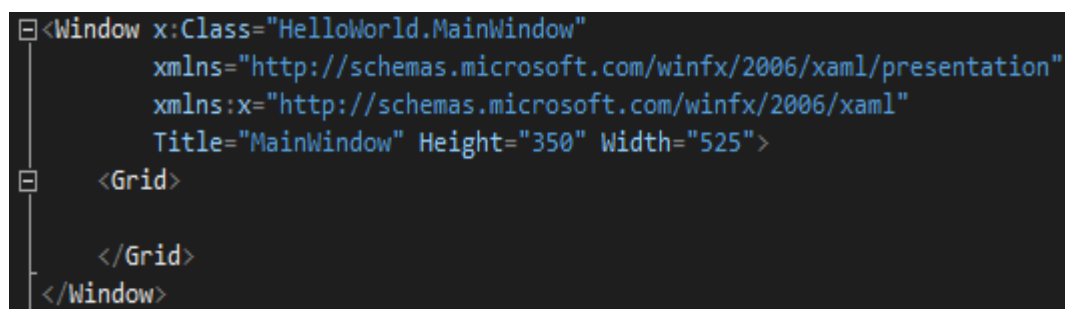
Dupa ce am scris metoda `btnClick` vom scrie, tot in clasa C#,

**`btn.MouseUp += new MouseButtonEventHandler(btnClick);`**

## APLICATII IN WPF

Atunci cand creem o aplicatie folosind WPF, primul lucru pe care il putem constata pe pagina este fereastra (Window), insotita in codul XAML de un marcator cu acest nume, `<Window>`, care are un border standard, o bara de titlu si butoane pentru minimizare, maximizare si inchidere. Acest marcator este marcatorul radacina pentru toti marcatorii pe care ii vom crea in XAML.

Codul XAML arata astfel:



```
<Window x:Class="HelloWorld.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">
    <Grid>
    </Grid>
</Window>
```

Atributul **x:Class** al marcatorului Window indica numele clasei folosite de catre aceasta fereastra. În cazul nostru, clasa este MainWindow. Atributul **xmlns** reprezinta namespace-ul de unde se pot prelua numele elementelor XAML. Daca nu am defini acest atribut, cu valoarea de mai sus, nume ca *Window*, *Button*, *TextBox*, *TextBlock*, etc... nu ar mai fi recunoscute. Atributul **xmlns:x** defineste namespace-uri care nu sunt implicite. Daca dorim definirea unui element sau a unui atribut in acest namespace, atunci il vom califica cu prefixul **x:**. Astfel se poate extinde vocabularul XAML.

Spre exemplu, daca vom considera definitia `<StackPanel x:Name="panelName">`, va aparea urmatoarea situatie:

1. Deoarece **StackPanel** nu este calificat cu **x:**, va fi rezolvat de catre spatiul de nume standard dat ca valoare atributului **xmlns**
2. Atributul **x:Name** este calificat cu **x:**, ceea ce inseamna ca va fi rezolvat de catre spatiul de nume dat ca valoare atributului **xmlns:x**.

Deasemenea, se poate folosi atributul **xmlns** si pentru a face trimitere din XAML la propriile spatii de nume. Pentru aceasta va trebui sa facem trimitere la propriul spatiu de nume definind atributul **xmlns:local="clr-namespace:NumeNamespace"** pentru marcatorul `<Window>` (se putea pune orice nume dupa „:” in loc de *local*, dar acesta este un nume sugestiv pentru spatiul local de nume)

Pentru a folosi clase din spatiul de nume local vom putea scrie, spre exemplu, astfel:

```
<local:NumeClasa x:Key="identificator" />
```

Codul C# pentru clasa **Main Window** arata ca in imaginea de mai jos:

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

namespace HelloWorld
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

Se poate observa faptul ca aceasta clasa este declarata ca partiala (are marcatorul *partial*) deoarece este combinata cu fisierul XAML la runtime pentru a obtine fereastra completa. Acest lucru il face apelul metodei **InitializeComponent** din constructorul clasei **MainWindow**.

Revenind la fisierul XAML, vom putea observa ca a fost creat si un marcator <Grid> in interiorul lui <**Window**>. Acesta este un panel WPF si este singurul fiu al lui Window (care oricum nu poate avea decat un singur control ca si fiu). Acest **Grid** insa poate avea mai multi fii.

Putem observa ca Window mai are si alte attribute in afara de **x:Class**. Acestea ar fi:

**Title** – care semnifica titlu scris pe bara de titlu a ferestrei

**Width** – reprezinta latimea ferestrei

**Height** – reprezinta inaltimea ferestrei

Fereastra mai poate avea si alte attribute, cum ar fi:

**Icon** – Permite definirea unei iconite pentru coltul stanga sus al ferestrei, si care va aparea deasemenea si in bara de task-uri, atunci cand ruleaza aplicatia si fereastra este deschisa.

**ResizeMode** – defineste modul in care utilizatorul poate redimensiona fereastra. Valoarea implicita a acestui atribut este *CanResize*, care permite redimensionarea ferestrei in mod obisnuit (tragand cu mouse-ul de margini sau folosind butoanele de minimizare si maximizare). Valoarea *CanMinimize* a acestui atribut permite doar minimizarea ferestrei (fara maximizare si fara a putea trage cu mouse-ul de marginile ei pentru redimensionare). Valoarea *NoResize* a atributului **ResizeMode** face sa dispara de pe bara de titlu a ferestrei butoanele de maximizare si minimizare, iar fereastra nu poate fi redimensionata nici folosind mouse-ul.

**ShowInTaskbar** – are valoarea implicita *True*. Daca ii setam valoarea *False*, fereastra nu va mai aparea in bara de task-uri. Aceasta proprietate este util sa fie setata pentru ferestrele care nu sunt principale sau atunci cand dorim ascunderea in System Tray a ferestrei la minimizarea acesteia.

**SizeToContent** – decide daca fereastra se va auto-redimensiona in functie de continutul ei. Valoarea implicita este *Manual*, care inseamna ca fereastra nu se redimensioneaza automat. Alte valori pe care le poate lua sunt: *Width*, *Height* sau *WidthAndHeight*, care vor redimensiona in mod automat fereastra pe latime, pe inaltime sau pe ambele dimensiuni.

**Topmost** – valoarea implicita a acestui atribut este *False*, dar daca este *True*, atunci fereastra va fi tot timpul in fata oricarei ferestre, mai putin in cazul in care este minimizata.

**WindowStartupLocation** – controleaza pozitia de start a ferestrei. Valoarea implicita este *Manual*, ceea ce inseamna ca fereastra va fi pozitionata implicit in functie de coltul stanga sus al ferestrei. Alte valori pe care le poate lua acest atribut sunt: *CenterOwner* sau *CenterScreen*. *CenterOwner* permite pozitionarea implicita in centrul ferestrei detinatoare, iar *CenterScreen* va pozitiona fereastra in centrul ecranului.

**WindowState** – controleaza starea initiala a ferestrei. Aceasta poate fi *Normal*, *Maximized* sau *Minimized*. Valoarea implicita este *Normal*, care inseamna ca fereastra este deschisa in mod normal (nici maximizata, nici minimizata).

Mai sunt si alte attribute, dar acestea sunt cele mai importante dintre ele.

## FISIERUL App.xaml

Atunci cand realizam o aplicatie folosind WPF se va crea in mod automat si un fisier cu acest nume, putandu-l vedea in fereastra **SolutionExplorer** din VisualStudio. **App.xaml** este punctul de start la lansarea in executie a aplicatiei, si este compus din doua fisiere (**App.xaml** si **App.xaml.cs** – unul va folosi marcatori, iar celalalt va folosi cod C#) Clasa **App** extinde clasa **Application**, care este o clasa centrala pentru aplicatiile create cu WPF. Aceasta este prima clasa care se executa la rularea aplicatiei, si de aici se va indica ce fereastra se va deschide prima data.

O alta trasatura importanta a acestei clase este aceea ca aici se pot defini resursele globale ale aplicatiei, care pot fi accesate de oriunde din aplicatie.

Structura initiala a fisierului App.xaml este urmatoarea:

```
<Application x:Class="HelloWorld.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>
```

Se poate observa atributul **StartupUri** care indica fereastra principala a aplicatiei.

Daca dorim sa specificam din cod C# fereastra care sa se deschida la lansarea in executie a aplicatiei, atunci vom inlocui atributul **StartupUri** cu atributul **Startup** care va reprezenta evenimentul de Startup si va avea ca si valoare numele unei metode ce va trata acest eveniment.

Metoda se va scrie in clasa App si arata astfel:

```
public partial class App : Application
{
    private void ApplicationStart(object sender, StartupEventArgs e)
    {
        // Crearea ferestrei de start
        MainWindow window = new MainWindow();
        // Show the window
        window.Show();
    }
}
```

Numele acestei metode va reprezenta valoarea atributului **Startup** al marcatorului Application din **App.xaml**.

## UTILIZAREA RESURSELOR IN APLICATIILE WPF

In WPF, datele se pot stoca sub forma de resurse. Resursele pot fi locale, pentru un anumit control, locale pentru intreaga fereastră sau globale pentru intreaga aplicatie.

Astfel, datele se vor plasa intr-un anumit loc, din care vor fi preluate pentru a fi utilizate in unul sau mai multe locuri.

In cele mai multe situatii, resursele se folosesc pentru stilizarea paginii sau pentru crearea de template-uri.

Exemple de utilizare a resurselor:

```
<Window x:Class="HelloWorld.MainWindow"
    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"
    Title="MainWindow" Height="350" Width="525">
  <Window.Resources>
    <sys:String x:Key="strHello">Hello, world!</sys:String>
  </Window.Resources>
  <Grid>
    <StackPanel Margin="10">
      <TextBlock Text="{StaticResource strHello}" FontSize="30" />
      <TextBlock>
        Just another "<TextBlock Text="{StaticResource strHello}" />" example, but with resources!
      </TextBlock>
    </StackPanel>
  </Grid>
</Window>
```

In exemplul de mai sus am declarat String-ul „Hello, World!” ca resursa pentru fereastra curenta. Ca sa putem folosi resursa de tip `<sys:String>`, va trebui sa adaugam la window urmatorul atribut – `xmlns:sys="clr-namespace:System;assembly=mscorlib"`. Resursele vor avea un atribut numit **x:Key** care ne va ajuta sa ne referim ulterior la acea resursa. (Il putem vedea pe acest atribut ca pe un nume dat resursei, pe care il folosim pentru a o identifica). Stringul retinut sub forma de resursa a fost folosit ulterior pe fereastra in doua locuri, sub forma de resursa statica – s-a folosit sintagma **StaticResource**.

Exista alternativa folosirii unei resurse dinamice, folosind sintagma **DynamicResource**.

Diferenta principala dintre aceste tipuri de resurse este aceea ca resursa statica este incarcata o singura data, la incarcarea fisierului XAML in care este definita. Daca resursa este



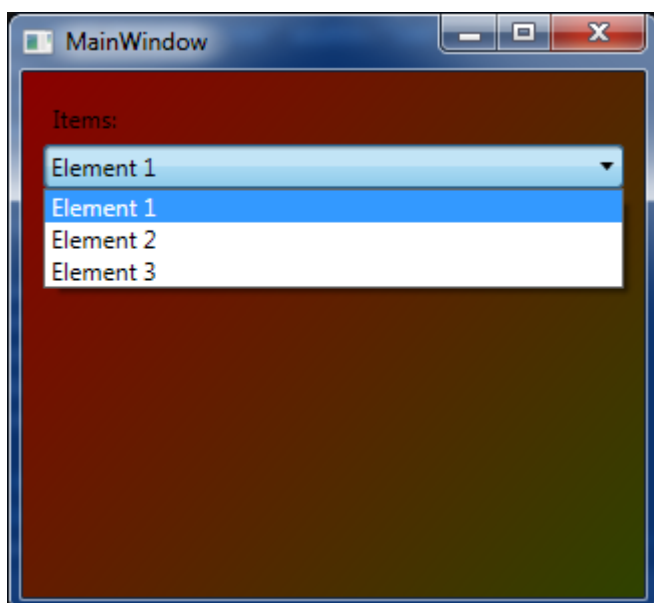
modificata ulterior in program, aceasta modificare nu se va reflecta asupra locurilor in care apare apelata resursa folosind **StaticResource**.

Resursa apelata cu **DynamicResource** este apelata doar atunci cand este nevoie de ea, si de asemenea se reapeleaza de fiecare data cand se modifica continutul ei.

Exemplu de aplicatie care foloseste atat resursa dinamica cat si resursa statica:

```
<Window x:Class="HelloWorld.MainWindow"
        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"
        Background="{DynamicResource WindowBackgroundBrush}"
        Title="MainWindow" Height="300" Width="500">
    <Window.Resources>
        <sys:String x:Key="ComboBoxTitle">Items:</sys:String>
        <x:Array x:Key="ComboBoxItems" Type="sys:String">
            <sys:String>Element 1</sys:String>
            <sys:String>Element 2</sys:String>
            <sys:String>Element 3</sys:String>
        </x:Array>
        <LinearGradientBrush x:Key="WindowBackgroundBrush">
            <GradientStop Offset="0" Color="DarkRed"/>
            <GradientStop Offset="1.5" Color="DarkGreen"/>
        </LinearGradientBrush>
    </Window.Resources>
    <Grid>
        <StackPanel Margin="10">
            <Label Content="{StaticResource ComboBoxTitle}" />
            <ComboBox ItemsSource="{StaticResource ComboBoxItems}" />
        </StackPanel>
    </Grid>
</Window>
```

Iar rezultatul este:



Textul de deasupra Combobox-ului este o resursa statica, identificata prin cheia **ComboBoxTitle**. Elementele din Combobox sunt reprezentate de o resursa de tip Array, identificata prin cheia **ComboBoxItems**, si apelata tot static. Fundalul ferestrei s-a realizat prin setarea proprietatii **Background** a lui Window. Acesta a primit ca valoare o resursa apelata dinamic, identificata prin cheia **WindowBackgroundBrush**. Aceasta resursa este de tipul LinearGradientBrush.

Resursele declarate la <Window.Resources> pot fi utilizate oriunde in acea fereastră.

Putem defini resurse si numai pentru un singur control. Spre exemplu am putea muta string-ul care defineste textul de deasupra Combobox-ului ca resursa locala a StackPanel-ului, astfel:

```
<StackPanel Margin="10">
    <StackPanel.Resources>
        <sys:String x:Key="ComboBoxTitle">Items:</sys:String>
    </StackPanel.Resources>
    <Label Content="{StaticResource ComboBoxTitle}" />
    <ComboBox ItemsSource="{StaticResource ComboBoxItems}" />
</StackPanel>
```

Acum vom putea folosi aceasta resursa doar pentru fiii acestui StackPanel.

Daca dorim sa folosim o anumita resursa pe mai multe ferestre ale aplicatiei, o vom putea defini global in **App.xaml**. Astfel vom crea un marcator <Application.Resources> in care vom defini resursele in acelasi mod in care le-am definit si pentru fereastră sau pentru un control.

Am putea sa adaugam resursele si din cod C#. Spre exemplu am putea adauga un buton pe pagina, iar la click pe acel buton am putea incarca Label-ul de deasupra Combobox-ului cu textul din resursa. Vom adauga astfel metoda:

```
private void btnClickMe_Click(object sender, RoutedEventArgs e)
{
    String labelText = panel.FindResource("ComboBoxTitle").ToString();
    lbl.Content = labelText;
}
```

Aceasta metoda o vom apela la evenimentul Click al butonului. La Label, in XAML, in loc de Content="{StaticResource ComboBoxTitle}" vom adauga proprietatea Name="lbl". Astfel vom putea identifica Label-ul in codul C#. Pentru a identifica StackPanel-ul ii vom pune si acestuia proprietatea **Name** cu valoarea *panel*. Din obiectul de tip StackPanel am apelat metoda **FindResource(*nume\_resursa*)** care va returna resursa sub forma de obiect (**object**), in cazul in care o gaseste.

Daca apelam metoda **FindResource** dintr-un control al ferestrei, iar resursa cautata nu se gaseste, atunci aceasta este cautata mai sus in ierarhie. Mai exact, este cautata printre resursele controlului parinte, printre resursele lui Window si in ultima instanta printre resursele globale din Application.

## CONTROALELE DE BAZA IN WPF

Printre cele mai uzuale controale din WPF sunt: TextBlock, Label, TextBox, CheckBox, RadioButton, PasswordBox.

Acestea vor fi detaliate in cateva randuri si vor fi prezentate cateva exemple pentru fiecare in parte.

### 1. TextBlock

Acest control grafic permite afisarea unui text pe ecran. Acelasi lucru il mai permite si Label-ul, doar ca, in linii mari, **Label-ul este pentru o linie de text, pe cand TextBlock-ul permite formatarea textului pe mai multe linii**. Daca Label-ul permite includerea si a altor elemente, precum imaginile, TextBlock-ul este exclusiv pentru redarea textului, si consuma si mai putine resurse fizice.

Daca textul din TextBlock este mai mare decat dimensiunea parintelui TextBlock-ului, atunci textul nu se va vedea in intregime (nu trece pe linia urmatoare in mod automat). Pentru a formata un text de dimensiuni mai mari, se poate recurge la cateva solutii, cum ar fi: **introducerea de break-uri explicite in text**, setarea atributului **TextWrapping** cu valoare **“Wrap”** sau marcarea trunchierii textului folosind atributul **TextTrimming**.

```
<TextBlock Margin="10" Foreground="Red">
    This is a TextBlock control <LineBreak /> with multiple lines of text.
</TextBlock>
<TextBlock Margin="10" TextTrimming="CharacterEllipsis" Foreground="Green">
    This is a TextBlock control with text that may not be rendered completely,
    which will be indicated with an ellipsis.
</TextBlock>
<TextBlock Margin="10" TextWrapping="Wrap" Foreground="Blue">
    This is a TextBlock control with automatically wrapped text, using the TextWrapping property.
</TextBlock>
```

În interiorul tag-ului TextBlock putem introduce alți marcatori de formatare a textului, cum ar fi: <Italic>...</Italic>, <Bold>...</Bold>, <Underline>...</Underline>, <Span>...</Span>, <Hyperlink>...</Hyperlink>

Exemplu cu Hyperlink:

```
<TextBlock Margin="10" TextWrapping="Wrap">
    This text has a
    <Hyperlink RequestNavigate="Hyperlink_RequestNavigate"
        NavigateUri="https://www.google.com">link</Hyperlink>
    in it.
</TextBlock>
```

Evenimentul care permite deschiderea unei pagini din browser cu url-ul specificat:

```
private void Hyperlink_RequestNavigate(object sender, RequestNavigateEventArgs e)
{
    System.Diagnostics.Process.Start(e.Uri.AbsoluteUri);
}
```

## 2. Label

Pe lângă cele prezentate anterior despre Label, acesta mai permite setarea unui border, definirea unui template pentru conținutul său, definirea de scurtături.

Label-ul nu are definită proprietatea Text, ci Content, deoarece nu este un control dedicat pentru text, ci permite definirea unor controale grafice în interiorul său.

Exemplu de scurtături:

```
<Label Content="_Name:" Target="{Binding ElementName=txtName}" />
<TextBox Name="txtName" />
<Label Content="_Mail:" Target="{Binding ElementName=txtMail}" />
<TextBox Name="txtMail" />
```

Am definit cheia de acces prin plasarea unui “\_” înaintea primului caracter al conținutului Label-ului (nu este obligatoriu înaintea primului) și astfel am setat cheile pentru scurtături: Alt+N, respectiv Alt+M. Prin setarea atributului Target am conectat Label-ul cu TextBox-ul specificat prin nume, ca valoare pentru Target. Astfel, apăsând cele 2 combinații de taste, vom muta focus-ul de la un TextBox la celălalt.

În cele ce urmează am detaliat exemplul de mai sus pentru a defini un control grafic ca și conținut pentru Label:

```

<Label Target="{Binding ElementName=txtName}">
    <StackPanel Orientation="Horizontal">
        <Image Source="Images/bullet_green.png" />
        <AccessText Text="_Name:" />
    </StackPanel>
</Label>
<TextBox Name="txtName" />
<Label Target="{Binding ElementName=txtMail}">
    <StackPanel Orientation="Horizontal">
        <Image Source="Images/bullet_blue.png" />
        <AccessText Text="_Mail:" />
    </StackPanel>
</Label>
<TextBox Name="txtMail" />

```

In exemplul de mai sus, am folosit tag-ul `<AccessText>` pentru a putea defini o cheie de acces pentru textul din Label.

### 3. TextBox

Aceasta componenta permite introducerea de text de la tastatura. In mod implicit, textul este scris pe un singur rand, dar daca adaugam atributul **AcceptsReturn** cu valoarea **“true”**, atunci va permite redimensionarea TextBox-ului si **treccerea la o noua linie in momentul in care dam click pe Enter**. Atributul **TextWrapping** cu valoare **“Wrap”** va trece automat la o linie noua in momentul in care textul din TextBox este mai mare decat latimea TextBox-ului:

```
<TextBox AcceptsReturn="True" TextWrapping="Wrap" />
```

Exemplu de **eveniment SelectionChanged** pentru TextBox:

```

<TextBox SelectionChanged="TextBox_SelectionChanged" />
<Label Name="lblResult"/>

```

```

private void TextBox_SelectionChanged(object sender, RoutedEventArgs e)
{
    TextBox textBox = sender as TextBox;
    lblResult.Content = "Selection starts at " + textBox.SelectionStart + Environment.NewLine;
    lblResult.Content += "Selection is " + textBox.SelectionLength + " chars.";
    lblResult.Content += Environment.NewLine;
    lblResult.Content += "Selected text: " + textBox.SelectedText;
}

```

Exemplu de **eveniment TextChanged** pentru TextBox:

```
<TextBox Name="txt" SelectionChanged="TextBox_SelectionChanged" TextChanged="TextBox_TextChanged" />
<Label Name="lblResult"/>
<Label Name="lblResult2"/>
```

```
private void TextBox_TextChanged(object sender, TextChangedEventArgs e)
{
    lblResult2.Content = txt.Text;
}
```

#### 4. CheckBox

```
<Label FontWeight="Bold">Options:</Label>
<CheckBox>First option</CheckBox>
<CheckBox IsChecked="True">Second option</CheckBox>
<CheckBox>Third option</CheckBox>
```

Si CheckBox-urile permit adaugarea de componente grafice ca fii ai acestora:

```
<StackPanel Margin="10">
    <Label FontWeight="Bold">Options:</Label>
    <CheckBox>
        <TextBlock>
            Option <Run Foreground="Green" FontWeight="Bold">ABC</Run>
        </TextBlock>
    </CheckBox>
    <CheckBox IsChecked="True">
        <WrapPanel>
            <TextBlock>
                Option <Span FontWeight="Bold">DEF</Span>
            </TextBlock>
            <Image Source="/WpfSimpleControls;component/Images/bullet_blue.png"
                Width="16" Height="16" Margin="5,0" />
        </WrapPanel>
    </CheckBox>
    <CheckBox>
        <TextBlock>
            Option <Span Foreground="Blue" TextDecorations="Underline" FontWeight="Bold">GHI</Span>
        </TextBlock>
    </CheckBox>
</StackPanel>
```

Principalul eveniment la care subscrie un CheckBox este **CheckedChanged**.

## 5. RadioButton

Componentele de tip RadioButton se definesc de obicei intr-un “radio group”. Adica se va seta proprietatea **GroupName** cu aceeasi valoare pentru elementele din acelasi “radio group”. In caz contrar, se vor putea selecta mai multe elemente simultan.

```
<Label FontWeight="Bold">Are you ready?</Label>
<RadioButton GroupName="ready">Yes</RadioButton>
<RadioButton GroupName="ready">No</RadioButton>
<RadioButton GroupName="ready" IsChecked="True">Maybe</RadioButton>
```

Principalele evenimente pentru RadioButtons sunt **Checked** si **Unchecked**. Exemplu:

```
private void RadioButton_Checked(object sender, RoutedEventArgs e)
{
    MessageBox.Show((sender as RadioButton).Content.ToString());
}

private void RadioButton_Unchecked(object sender, RoutedEventArgs e)
{
    MessageBox.Show("unchecked");
}
```

## 6. PasswordBox

Acest control este asemanator cu TextBox-ul, numai ca nu afiseaza caracterele scrise in interior. Acesta permite modificarea caracterelor care mascheaza textul prin proprietatea **PasswordChar**. Permite deasemenea si definirea unei dimensiuni maxime a textului din interior prin setarea unei valori pentru proprietatea **MaxLength**. Textul scris intr-un PasswordBox se gestioneaza folosind **proprietatea Password**. **Aceasta proprietate nu poate fi asociata cu o valoare prin Binding!**

### Adaugarea de Tooltip-uri componentelor grafice

Majoritatea componentelor vizuale permit adaugarea de Tooltip-uri (acele Help-uri care apar la Mouse Over pe componenta). Un exemplu simplu de adaugare de Tooltip ar fi urmatorul:

```
<TextBox Tooltip="Enter Username" />
```

Un exemplu de ToolTip mai complex pentru un buton, care contine text formatat in diverse moduri si o imagine:

```
<Button Margin="80">
  <Button.Content>
    <Image Source="/WpfSimpleControls;component/Images/bullet_green.png" Height="25" />
  </Button.Content>
  <Button.ToolTip>
    <StackPanel>
      <TextBlock FontWeight="Bold" FontSize="14" Margin="0,0,0,5">My Text</TextBlock>
      <TextBlock>
        Line 1 <LineBreak /> Line 2.
      </TextBlock>
      <Border BorderBrush="Silver" BorderThickness="0,1,0,0" Margin="0,8" />
      <WrapPanel>
        <Image Source="/WpfSimpleControls;component/Images/bullet_blue.png"
          Margin="0,0,5,0" />
        <TextBlock FontStyle="Italic">Another text</TextBlock>
      </WrapPanel>
    </StackPanel>
  </Button.ToolTip>
</Button>
```

Pentru a seta durata unui ToolTip vom seta proprietatea **ToolTipService.ShowDuration** cu o valoare in milisecunde (de ex daca vom seta la valoarea 3000, ToolTip-ul se va afisa 3 secunde). Pentru alte proprietati ale ToolTip-ului puteti testa ce va mai pune la dispozitie ToolTipService.

## PANEL-URI IN WPF

Panel-urile sunt unele dintre cele mai importante componente grafice oferite de WPF. Si panel-urile pot fi apelate cu denumirea de “controale WPF” Acestea reprezinta containerele pentru componentele grafice, cele care se afiseaza pe fereastra aplicatiei. Daca fereastra (elementul Window) poate contine un singur fiu, panel-urile pot contine mai multi, impartind spatiul intre acesti fii ai lor in diverse moduri, in functie de tipul panel-ului.

Panel-urile sunt urmatoarele:

1. **Canvas** - este un panel simplu, care nu ofera o pozitionare specifica, ci afiseaza elementele fiu **in functie de coordonatele la care au fost pozitionate** acestea. In acest caz, programatorul trebuie sa specifice pozitiile elementelor de pe canvas si sa se asigure ca acestea sunt aliniate.



2. **WrapPanel** - este un panel care permite fiilor sa se alinieze unul dupa celalalt, **orizontal in mod implicit**, si vertical doar daca i se specifica la proprietatea **Orientation**. Daca dimensiunea WrapPanel-ului nu este suficient de mare incat sa incapa toti fiii acestuia aliniati orizontal (sau vertical, daca s-a facut o astfel de specificare), atunci elementele care nu incap vor trece pe o linie (sau coloana) noua....si tot asa pana se afiseaza toate elementele.

3. **StackPanel** - este un panel asemanator cu WrapPanel-ul, doar ca aici, daca elementele nu incap, nu se creaza o noua linie(sau coloana) pentru a fi afisate, ci se afiseaza o singura linie(coloana), deoarece dimensiunile fiilor sunt date de dimensiunea StackPanel-ului, nu de dimensiunea celui mai mare element din sirul de fii care se afiseaza. **Orientarea implicita la StackPanel este cea verticala**, dar se poate face si afisare orizontala in mod explicit, prin setarea aceleiasi proprietati, **Orientation**. De exemplu, daca avem 10 TextBox-uri pe care dorim sa le pozitionam intr-un astfel de panel, acestea vor avea latimea cat este StackPanel-ul de lat, iar inaltimea va fi cea implicita pentru un TextBox. Daca Inaltimea StackPanel-ului este cat suma inaltimilor a 7 textBox-uri, atunci ultimele 3 TextBox-uri nu se vor afisa. Ele vor aparea doar in cazul in care marim inaltimea StackPanel-ului.

4. **DockPanel** - este un panel care permite alinierea fiilor incepand cu partea de sus, jos, stanga sau dreapta, prin proprietatea **Dock**. Daca ultimului fiu nu i se specifica o pozitionare pentru proprietatea Dock, atunci el va umple tot spatiul ramas disponibil in panel.

5. **Grid** - este unul dintre cele mai complexe panel-uri. Acesta permite dispunerea tabelara a fiilor sai, pe linii si coloane. Liniile si coloanele unui grid pot avea latimea si inaltimea specificate in pod explicit, iar in caz contrar vor lua dimensiunile celui mai mare dintre elementele fiu. Pozitionarea elementelor dintr-o celula a grid-ului se face dupa coordonate, ca in canvas. Daca nu ii specificam `<RowDefinition />` si `<ColumnDefinition />`, atunci grid-ul va avea o singura linie si o singura coloana.

6. **UniformGrid** - este un panel asemenator cu grid-ul, cu diferenta ca are celulele de dimensiuni egale...se poate folosi spre exemplu daca dorim sa reprezentam o tabla de sah.

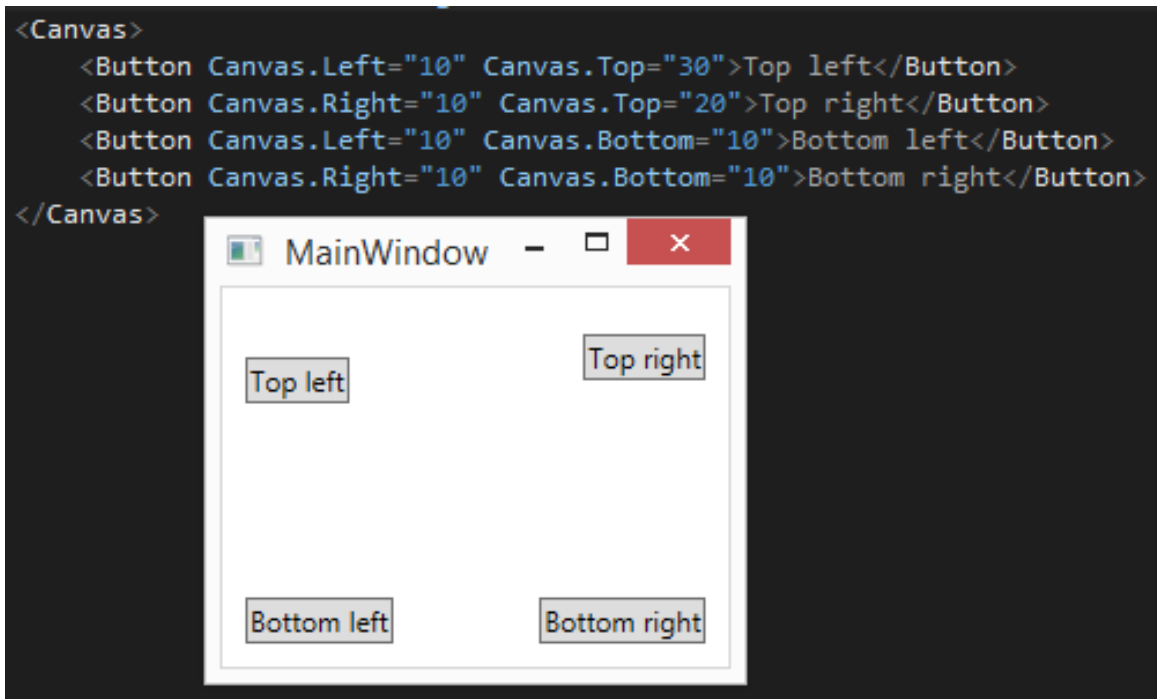
In cele ce urmeaza vom detalia componentele de tip panel si vom urmari cateva exemple pentru fiecare tip in parte:

### Canvas

**In mod implicit, daca nu specificam vreo pozitionare a elementelor de pe un canvas, acestea vor fi pozitionate din coltul stanga sus al canvasului, unul peste altul.** Din acest

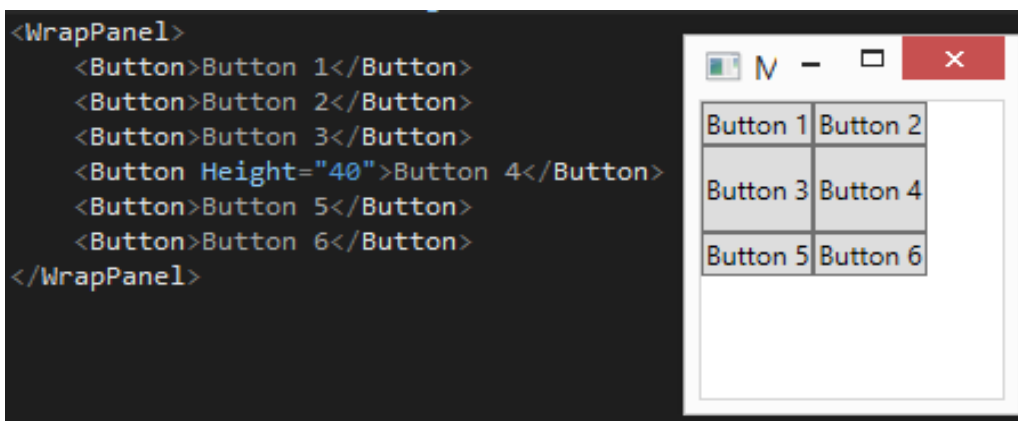
motiv, elementele fiu ale unui astfel de panel vor trebui pozitionate fie cu ajutorul atributului **Margin**, care seteaza pixelul din stanga sus al elementului (valoarea pentru coordonata X si pentru coordonata Y) si dimensiunile elementului (latime si inaltime), fie cu ajutorul atributelor **Canvas.Left**, **Canvas.Right**, **Canvas.Top**, **Canvas.Bottom**, care, prin valorile lor numerice intregi, specifica numarul de pixeli de la marginile din stanga, dreapta, sus sau jos pana la conturul elementului.

### Exemplu:



Daca dorim ca anumite elemente de pe canvas sa se suprapuna intr-un anumit mod, pentru a stabili ordinea de suprapunere vom seta atributul **Panel.ZIndex** cu o valoare intreaga. Cu cat valoarea este mai mare, cu atat elementul care o are setata va sta mai la suprafata. In mod implicit, elementele se suprapun in ordinea in care au fost create.

### **WrapPanel**



Se poate observa in exemplul de mai sus ca toate elementele de pe linia pe care se gaseste elementul cu inaltimea 40 (cea mai mare dimensiune) au luat dimensiunea elementului respectiv.

Un lucru similar s-ar fi petrecut daca setam o latime mai mare acelui buton; toate elementele de pe coloana cu butonul 4 ar fi avut latimea setata.

## StackPanel

Este asemanator cu WrapPanel, doar ca elementele nu fac “wrapping” in momentul in care nu incap pe o linie sau pe o coloana in panel (in functie de orientarea acestuia), ci se intind pe toata dimensiunea data de orientarea panelului. Daca are orientare implicita, verticala, atunci elementele se vor lăți cat toata latimea panelului si se vor aseza unul sub celalalt. Daca orientarea este pe orizontala, atunci toate elementele vor avea aceiasi inaltime, egala cu inaltimea panelului, si se vor pozitiona unul dupa celalalt. Aceasta se datoreaza proprietatii **VerticalAlignment/HorizontalAlignment** care au valoarea setata pe **Stretch**.

Pentru **VerticalAlignment** se poate alege dintre **Stretch, Bottom, Center si Top**, iar pentru **HorizontalAlignment** se poate alege dintre **Stretch, Left, Center si Right**.

Elementele care nu mai incap in panel nu se vor mai afisa.

### Exemplu:

```
<StackPanel Orientation="Horizontal">
  <Button VerticalAlignment="Top">Button 1</Button>
  <Button VerticalAlignment="Center">Button 2</Button>
  <Button VerticalAlignment="Bottom">Button 3</Button>
  <Button VerticalAlignment="Bottom">Button 4</Button>
  <Button VerticalAlignment="Center">Button 5</Button>
  <Button VerticalAlignment="Top">Button 6</Button>
</StackPanel>
```

## DockPanel

Dupa cum mentionam anterior, elementele fiu ale unui astfel de panel vor putea avea setata o proprietate numita **DockPanel.Dock**, care decide in ce parte a panel-ului se vor ancora aceste elemente. Daca componentele unui DockPanel nu au setata aceasta proprietate, ele vor fi ancorate implicit la stanga. Valorile pe care le poate lua aceasta proprietate sunt **Left, Right, Top sau Bottom**. Ultimul fiu al panel-ului va umple toata portiunea ramasa.

### Exemplu:

```
<DockPanel>
  <Button DockPanel.Dock="Top" Height="50">Top</Button>
  <Button DockPanel.Dock="Bottom" Height="50">Bottom</Button>
  <Button DockPanel.Dock="Left" Width="50">Left</Button>
  <Button DockPanel.Dock="Right" Width="50">Right</Button>
  <Button>Center</Button>
</DockPanel>
```

Pentru a dezactiva comportamentul DockPanel-ului de a umple spatiul ramas liber cu ultimul element din interiorul sau, i se poate seta proprietatea LastChildFill="false".

### **Grid**

Este un panel caruia i se pot seta randuri si coloane. Elementelor din grid li se va specifica randul si coloana pe in care se vor pozitiona. Daca nu se specifica nimic, toate se vor suprapune in prima celula din grid (Grid.Row="0" Grid.Column="0"....acestea sunt valorile implicite).

### Exemplu:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="3*" />
    <ColumnDefinition Width="1*" />
    <ColumnDefinition Width="1*" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="2*" />
    <RowDefinition Height="1*" />
    <RowDefinition Height="1*" />
  </Grid.RowDefinitions>
  <Button Grid.Row="0" Grid.Column="0">Button 1</Button>
  <Button Grid.Row="0" Grid.Column="1">Button 2</Button>
  <Button Grid.Row="0" Grid.Column="2">Button 3</Button>
  <Button Grid.Row="1" Grid.Column="0">Button 4</Button>
  <Button Grid.Row="1" Grid.Column="1">Button 5</Button>
  <Button Grid.Row="1" Grid.Column="2">Button 6</Button>
  <Button Grid.Row="2" Grid.Column="0">Button 7</Button>
  <Button Grid.Row="2" Grid.Column="1">Button 8</Button>
  <Button Grid.Row="2" Grid.Column="2">Button 9</Button>
</Grid>
```

În exemplul de mai sus puteți observa că la definirea rândurilor și a coloanelor s-a specificat și înălțimea și respectiv lățimea. Spre exemplu la coloane, prima va avea lățimea cât 3 părți din lățimea totală a gridului, iar celelalte două coloane vor avea câte o parte. Dacă aceste dimensiuni nu sunt specificate, rândurile și coloanele din grid vor avea dimensiuni egale. **Latimea rândurilor și înălțimea coloanelor pot fi specificate și prin număr exact de pixeli sau cu valoarea "Auto"**, care semnifică faptul că elementul va avea dimensiunea exact cât este nevoie pentru a încadra cel mai mare conținut al său.

Puteți testa codul de mai jos, redimensionând în diverse moduri fereastra, după rulare.

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="1*" />
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="100" />
  </Grid.ColumnDefinitions>
  <Button>Button 1</Button>
  <Button Grid.Column="1">Button 2 with long text</Button>
  <Button Grid.Column="2">Button 3</Button>
</Grid>
```

Dacă dorim să unim două sau mai multe celule, pe rând sau pe coloană, atunci va trebui să specificăm acest lucru elementului din celulă modificată, prin setarea proprietății `Grid.RowSpan` sau `Grid.ColumnSpan`. Acestea vor specifica numărul de linii și respectiv de coloane care se vor uni.

Exemplu de utilizare:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>
  <Button Grid.ColumnSpan="2">Button 1</Button>
  <Button Grid.Column="3">Button 2</Button>
  <Button Grid.Row="1">Button 3</Button>
  <Button Grid.Column="1" Grid.Row="1" Grid.RowSpan="2" Grid.ColumnSpan="2">Button 4</Button>
  <Button Grid.Row="2">Button 5</Button>
</Grid>
```

Se poate folosi spre exemplu un **Splitter** pe grid ca in exemplul de mai jos (splitter vertical). Se imparte grid-ul in 3 celule, iar cea din mijloc va avea o latime foarte mica (5 px). In celulele din stanga si din dreapta se va pozitiona continutul dorit, iar in celula din centru va fi pozitionat splitter-ul, cu proprietatea **HorizontalAlignment="Stretch"**.

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="5" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <TextBlock Grid.Column="0" FontSize="55" HorizontalAlignment="Center"
    VerticalAlignment="Center" TextWrapping="Wrap">Left side</TextBlock>
  <GridSplitter Grid.Column="1" Width="5" HorizontalAlignment="Stretch" />
  <TextBlock Grid.Column="2" FontSize="55" HorizontalAlignment="Center"
    VerticalAlignment="Center" TextWrapping="Wrap">Right side</TextBlock>
</Grid>
```

Similar se va putea defini si un Splitter orizontal.

## **INTRODUCERE IN WPF DATA BINDING**

**Data Binding** este tehnica generala care permite asocierea sursei datelor cu elementul care utilizeaza datele respective, sincronizand totodata sursa cu utilizatorul. In procesul de “Data Binding”, **daca se modifica sursa datelor, atunci aceasta modificare se va reflecta si la utilizatorul datelor**, iar utilizatorul, la randul sau, poate modifica si el sursa de date.

In WPF, Data Binding este cheia majoritatii lucrurilor pe care le efectuam. Spre deosebire de WindowsForms, unde evenimentele apareau oriunde se interactiona cu interfata grafica, in WPF se poate realiza doar Data Binding intre o sursa de date si o componenta de pe interfata grafica. Insa WPF permite si abordarea clasica, cea cu evenimente.

Spre exemplu, daca dorim sa populam un ComboBox cu date dintr-o lista, putem face un Data Binding intre cele doua (varianta eleganta) sau putem atribui fiecarui Item al Combobox-ului cate un element din lista, parcurgand-o cu un “for” la evenimentul de incarcare a paginii (varianta clasica).

**Exemplu de utilizare:** Dorim sa avem pe pagina un TextBox in care sa introducem date si un TextBlock in care sa afisam ceea ce am introdus, chiar in momentul in care am facut introducerea textului. Daca folosim metoda clasica, va trebui sa manipulam un eveniment al

TextBox-ului (**TextChangedEventHandler** mai exact), eveniment care ar trebui sa schimbe textul din TextBlock cu textul din TextBox.

Daca folosim Data Binding, nu avem nevoie de “code behind”, ci vom face acest lucru folosindu-ne doar de marcatori si de proprietatile acestora.

```
<StackPanel Margin="10">
    <TextBox Name="txtValue" />

    <WrapPanel Margin="0,10">
        <TextBlock Text="Value: " FontWeight="Bold" />
        <TextBlock Text="{Binding Path=Text, ElementName=txtValue}" />
    </WrapPanel>

    <Label Name="lblHello" FontWeight="Bold" ContentStringFormat="Text: {0}"
        Content="{Binding Path=Text, ElementName=txtValue}" />
</StackPanel>
```

Conform codului de mai sus, TextBlock-ul are proprietatea Text asociata cu proprietatea Text a TextBox-ului (txtValue). Acest lucru se intampla datorita prezentei cuvintului Binding intre acolade, care descrie relatia de asociere. Cel mai simplu mod in care se poate folosi Data Binding este: **{Binding}**. Acest apel fara alti parametri returneaza “Data Context-ul” curent. Vom vedea acest lucru intr-un exemplu viitor. **De cele mai multe ori insa, dorim sa facem asocierea unei proprietati cu o alta proprietate a DataContext-ului, nu cu intreg DataContext-ul.** In acest caz vom scrie {Binding **Path**=NumeProprietate}, sau prescurtat, {Binding NumeProprietate}. Binding mai are si alte proprietati, asa cum se poate observa si in codul de mai sus. Proprietatile sale se despart prin virgula. Una dintre acestea este **ElementName**, care ne permite sa facem asocierea direct cu o componenta grafica. In acest caz, DataContext-ul nu va fi reprezentat de vre-un obiect din logica aplicatiei, ci de un alt element grafic (un TextBox). Astfel, **proprietatea Text a TextBlock-ului a fost asociata cu proprietatea Text a TextBox-ului.** Vedeti in exemplul de mai sus si afisarea textului din TextBox intr-un Label (o varianta si mai eleganta).

**DataContext**-ul reprezinta sursa implicita de date pentru Binding in WPF. Daca nu se specifica nici o valoare pentru DataContext, aceasta va fi implicit “null”. Se poate declara in mod explicit o alta sursa de date, prin atributul ElementName al lui Binding, asa cum am vazut in exemplul anterior. **DataContext este o proprietate a clasei FrameworkElement, clasa din care sunt derivate majoritatea elementelor grafice, inclusiv elementul Window.** Pe langa „ElementName” se mai poate atribui o sursa de date DataContext-

ului prin proprietatea „RelativeSource”, care va primi valoarea de la o resursa declarata anterior.

### Exemplu:

Am creat o clasa User in namespace-ul WpfSimpleControls. In constructorul clasei am atribuit valori proprietatilor Username si Password.

In XAML am specificat proprietatea DataContext pentru fereastra curenta ca fiind User. Prin acea specificare, se creaza un obiect de tip User si se atribuie ca valoare lui DataContext.

**Observatie!** Cuvantul “context” nu este predefinit, ci l-am definit ca atribut al lui Window astfel: `xmlns:context="clr-namespace:WpfSimpleControls"`

Am anuntat astfel namespace-ul din care sa apeleze clasa User.

**Dupa ce a fost setat DataContext-ul, in interiorul ferestrei avem acces la proprietatile DataContext-ului (adica ale obiectului de tip User in cazul nostru).**

Astfel am asociat Proprietatea Text a primului TextBox cu proprietatea Username a obiectului de tip User si proprietatea Text a celui de-al doile TextBox cu proprietatea Password a obiectului de tip User. Datorita acestor asocieri, la rularea aplicatiei, textul din primul TextBox va fi valoarea proprietatii Username, adica “ion”, iar textul din al doilea TextBox va fi valoarea proprietatii Password, adica “123”.

Exemplul de mai jos a oferit valoare proprietatii DataContext a intregii pagini, iar toate elementele de pe pagina au mostenit acea valoare.

```
namespace WpfSimpleControls
{
    class User
    {
        public User()
        {
            Username = "ana";
            Password = "123";
        }
        public string Username { get; set; }
        public string Password { get; set; }
    }
}
```



```

<Window.DataContext>
    <context:User />
</Window.DataContext>

<StackPanel>
    <WrapPanel Margin="0,10,10,10">
        <TextBlock Text="Username: " />
        <TextBox Text="{Binding Username}" Width="100" />
    </WrapPanel>
    <WrapPanel>
        <TextBlock Text=" Password: " />
        <TextBox Text="{Binding Password}" Width="100" />
    </WrapPanel>
</StackPanel>

```

DataContext-ul se poate insa suprascrie pentru anumite componente grafice, avand in vedere ca majoritatea au definita aceasta proprietate. Adica se poate ca obiectul Window sa aiba un DataContext, iar un TextBox de pe pagina sa aiba definit un alt DataContext.

Am mentionat anterior, ca modificarea sursei de date implica modificarea valorilor proprietatilor componentelor grafice. Cand si cum se reflecta modificarile elementelor grafice asupra sursei de date ne indica o proprietate a lui Binding, numita **UpdateSourceTrigger**. **Implicit, aceasta proprietate are valoarea *Default*, dar mai poate lua valorile *Explicit*, *LostFocus* sau *PropertyChanged*.** Valoarea "Default" a lui UpdateSource Trigger inseamna ca sursa se va modifica in functie de proprietatea cu care este asociata. De exemplu pentru scriere, toate proprietatile cu exceptia proprietatii Text se modifica imediat ce s-a schimbat valoarea pe interfata. Proprietatea Text se actualizeaza dupa ce pierde focus-ul elementul grafic.

In cele ce urmeaza puteti observa un exemplu care sa evidentieze valorile pe care le poate lua UpdateSourceTrigger.

Vrem ca fereastra insasi reprezinta sursa de date pentru ea. Acest lucru a fost semnalat in cod, in constructorul ferestrei, prin apelul: `this.DataContext = this;`

Acest lucru se mai poate semnala si prin adaugarea urmatoarei proprietati la Window, in codul XAML: `DataContext="{Binding RelativeSource={RelativeSource Self}}"`.

Se doreste afisarea in 3 TextBox-uri, 3 dintre proprietatile ferestrei curente, si nume: numele ferestrei, latimea si inaltimea (Title, Width si Height). Modificarile valorilor din TextBox vor modifica proprietatile ferestrei **Explicit** (la click pe un buton se va apela metoda UpdateSource()), la evenimentul de **LostFocus** (cand TextBox-ul in care se scrie latimea ferestrei pierde focus-ul, atunci se reflecta modificarile asupra ferestrei) si la

**PropertyChanged** (pe masura ce modificam textul din TextBox-ul in care scriem inaltimea ferestrei, se modifica si fereastra).

```
<StackPanel Margin="15">
    <WrapPanel>
        <TextBlock Text="Window title: " />
        <TextBox Name="txtWindowTitle" Text="{Binding Title, UpdateSourceTrigger=Explicit}"
            Width="150" />
        <Button Name="btnUpdateSource" Click="btnUpdateSource_Click" Margin="5,0" Padding="5,0">
            *
        </Button>
    </WrapPanel>
    <WrapPanel Margin="0,10,0,0">
        <TextBlock Text="Window dimensions: " />
        <TextBox Text="{Binding Width, UpdateSourceTrigger=LostFocus}" Width="50" />
        <TextBlock Text=" x " />
        <TextBox Text="{Binding Height, UpdateSourceTrigger=PropertyChanged}" Width="50" />
    </WrapPanel>
</StackPanel>
```

Codul de la evenimentul de click pe buton este urmatorul:

```
private void btnUpdateSource_Click(object sender, RoutedEventArgs e)
{
    BindingExpression binding = txtWindowTitle.GetBindingExpression(TextBox.TextProperty);
    binding.UpdateSource();
}
```

**Valoarea Explicit a lui UpdateSourceTrigger mascheaza insa adevarata valoare a lui Binding.**

Am vorbit pana acum de modul in care influenteaza modificarile facute pe interfata grafica sursa de date. *Se pune insa problema si invers. Cum influenteaza modificarile facute asupra sursei de date interfata grafica.*

Reconsideram exemplul cu utilizatorii de mai sus. Vom adauga insa un buton pe interfata si dorim ca la click pe acel buton sa se modifice username-ul si parola:

Codul C# pentru evenimentul de Click pe buton este urmatorul:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    User user = this.DataContext as User;
    user.Username = "maria";
    user.Password = "12345";
}
```

Putem observa ca se preia utilizatorul care reprezinta DataContext-ul clasei curente (adica al clasei Window) si i se modifica cele 2 proprietati: Username si Password.

```
<Window.DataContext>
    <context:User />
</Window.DataContext>
<StackPanel>
    <WrapPanel Margin="10">
        <TextBlock>Username:</TextBlock>
        <TextBox Width="100" Text="{Binding Username}" />
    </WrapPanel>
    <WrapPanel Margin="10">
        <TextBlock>Password:</TextBlock>
        <TextBox Width="100" Text="{Binding Password}" />
    </WrapPanel>
    <Button Margin="10,10,300,10" Content="Change" Click="Button_Click" />
</StackPanel>
```

Daca rulam aplicatia si dam click pe buton, vom astepta sa vedem ca in cele 2 TextBox-uri s-a modificat textul cu noile valori ale proprietatilor Username si Password. Dar nu se va intampla asa. Motivul: La rulara aplicatiei, atunci cand s-a stabilit DataContext-ul ferestrei curente, s-a apelat constructorul clasei User, cel in care se stabilesc valorile initiale pentru Username si Password, si cum aceste proprietati sunt asociate cu proprietatile Text ale celor doua TextBox-uri din fereastra, valorile initiale vor aparea pe interfata. **In momentul in care dam click pe buton, vom schimba valorile pentru Username si Password - vom schimba valorile sursei de date pentru pagina curenta - dar nu a cerut nimeni ca interfata sa verifice daca se modifica cumva sursa de date, asa ca interfata nu va fi notificata in mod automat de modificarile intervenite asupra DataContext-ului.** Problema se poate rezolva prin implementarea interfetei **INotifyPropertyChanged** de catre clasa care constituie sursa de date a ferestrei (clasa User in cazul nostru). Daca am fi reprezentat si liste de elemente pe interfata grafica, pentru a fi notificata interfata ca s-a facut o schimbare in lista de elemente, aceasta lista (List<T>) ar fi trebuit inlocuita cu tipul **ObservableCollection<T>**.

Interfata **INotifyPropertyChanged** contine un eveniment de tipul **PropertyChangedEventHandler**. Acest eveniment trebuie tratat in clasa User, pentru fiecare proprietate care va trebui sa notifice interfata grafica cum ca i s-a schimbat valoarea.

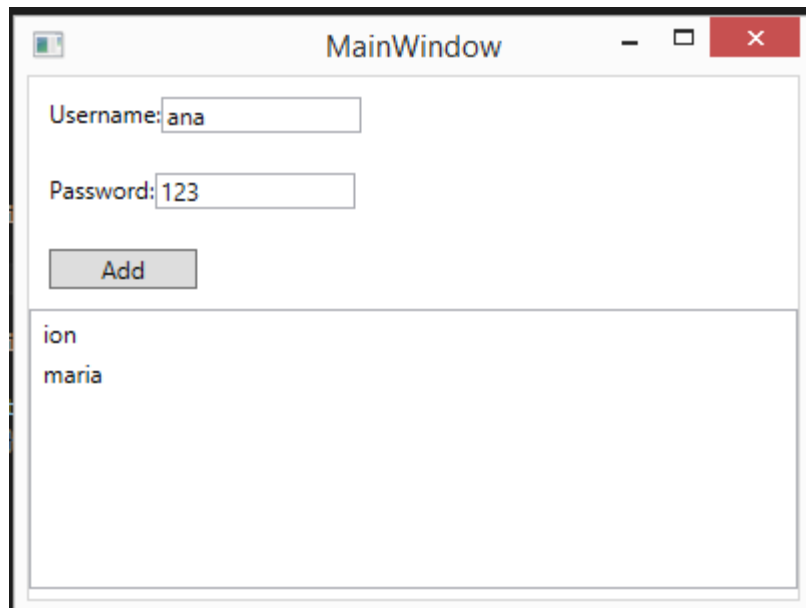
Codul modificat al clasei User este urmatorul:

```
class User : INotifyPropertyChanged
{
    public User()
    {
        Username = "ana";
        Password = "123";
    }
    private string username;
    public string Username
    {
        get
        {
            return username;
        }
        set
        {
            username = value;
            NotifyPropertyChanged("Username");
        }
    }
    private string password;
    public string Password
    {
        get
        {
            return password;
        }
        set
        {
            password = value;
            NotifyPropertyChanged("Password");
        }
    }
    public event PropertyChangedEventHandler PropertyChanged;
    public void NotifyPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
            PropertyChanged(this,
                new PropertyChangedEventArgs(propertyName));
    }
}
```

Daca rulam acum aplicatia, vom vedea ca in urma apasarii pe buton, textele din cele doua TextBox-uri se vor schimba.

Sa presupunem ca dorim ca pe fereastra curenta sa se afiseze si o lista de utilizatori, vom putea utiliza ca si componenta grafica un ListBox. Aceasta presupune ca vom avea nevoie de

o lista de utilizatori care sa reprezinte sursa de date pentru ListBox. Fereastra va trebui sa arate astfel:



Pentru aceasta, **am mai creat o clasa UserVM** care are 2 atribute private **User myUser** si **ObservableCollection<User> Users**. Am pus **ObservableCollection** si nu **List**, pentru a se face notificarea modificarilor colectiei de utilizatori. **UserVM implementeaza si ea interfata INotifyPropertyChanged** deoarece se doreste notificarea modificarilor proprietatii de tip User, numita MyUser.

Codul XAML al ferestrei va arata astfel:

```
<Window.DataContext>
    <context:UserVM />
</Window.DataContext>
<StackPanel>
    <WrapPanel Margin="10">
        <TextBlock>Username:</TextBlock>
        <TextBox Width="100" Text="{Binding MyUser.Username}" />
    </WrapPanel>
    <WrapPanel Margin="10">
        <TextBlock>Password:</TextBlock>
        <TextBox Width="100" Text="{Binding MyUser.Password}" />
    </WrapPanel>
    <Button Margin="10,10,300,10" Content="Add" Click="Button_Click" />
    <ListBox ItemsSource="{Binding Users}" DisplayMemberPath="Username" Height="140" />
</StackPanel>
```

Se poate observa ca DataContext-ul s-a modificat. Proprietatile pentru Binding s-au modificat si ele (deoarece UserVM nu are proprietati string Username si string Password, ci User MyUser). Componenta ListBox (ca si celelalte componente grafice care pot afisa o

colectie de elemente) are o proprietate numita **ItemSource**, care semnifica sursa de date pentru elementele sale. In cazul nostru, sursa de date se va lega de proprietatea Users a lui UserVM, care este un ObservableCollection<User>. De aici rezulta ca fiecare item din ListBox este un obiect de tip User. WPF nu va sti sa afiseze obiectul de tip User in ListBox. Pentru a rezolva problema va trebui fie sa **suprascrim metoda ToString() in clasa User**, fie sa **adaugam la ListBox proprietatea DisplayMemberPath** care sa ia valoarea proprietatii din User care se va afisa in ListBox. In exemplul de mai sus s-a mers pe varianta a doua si s-a afisat proprietatea Username.

Codul C# al clasei UserVM va arata stfel:

```
class UserVM : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    public void NotifyPropertyChanged(string propertyName)
    {
        if (this.PropertyChanged != null)
            this.PropertyChanged(this,
                                new PropertyChangedEventArgs(propertyName));
    }
    private User myUser;
    public User MyUser
    {
        get
        {
            return myUser;
        }
        set
        {
            myUser = value;
            NotifyPropertyChanged("MyUser");
        }
    }
    public ObservableCollection<User> Users { get; set; }

    public UserVM()
    {
        MyUser = new User();
        MyUser.Username = "ana";
        MyUser.Password = "123";
        Users = new ObservableCollection<User>()
        {
            new User()
            {
                Username = "ion",
                Password = "123"
            },
            new User()
            {
                Username = "maria",
                Password = "123"
            }
        }
    }
}
```

```
        };  
    }  
}
```

Codul de la click pe buton se va modifica si va arata astfel:

```
private void Button_Click(object sender, RoutedEventArgs e)  
{  
    UserVM userVM = this.DataContext as UserVM;  
    userVM.Users.Add(new User() { Username = "test", Password = "123" });  
}
```

Se poate observa ca la click pe buton se va adauga un nou utilizator in colectie, lucru care va fi actualizat si pe interfata grafica.