.NET Programming Technologies

Dr. Kovásznai, Gergely

Biró, Csaba

.NET Programming Technologies

Dr. Kovásznai, Gergely
Biró, Csaba

Publication date 2013

Tartalom

1. Prologue 1
2. Introduction (written by Csaba Biró) 2
    1. Device-independent units 2
    2. WPF multi-layer architecture 4
    3. WPF class hierarchy 4
        3.1. System.Object 5
        3.2. System.Threading.DispatcherObject 5
        3.3. System.Windows.DependencyObject 5
        3.4. System.Windows.Media.Visual 6
        3.5. System.Windows.UIElement 6
        3.6. System.Windows.FrameworkElement 6
        3.7. System.Windows.Shapes.Shape 6
        3.8. System.Windows.Controls.Control 6
        3.9. System.Windows.Controls.ContentControl 7
3. XAML (written by Csaba Biró) 8
    1. The fundamental files of WPF project 10
    2. Code-behind class 10
    3. XAML namespaces 12
    4. Properties 12
    5. Advanced properties 13
    6. Content property 14
    7. Markup Language Extensions 14
    8. More x:prefixes 15
    9. Special characters and whitespaces 15
        9.1. Explanation 15
        9.2. Character entities 15
4. Layouts (written by Csaba Biró) 16
    1. Alignment, margins 16
        1.1. Inner and outer margins 16
        1.2. Adjustments 17
    2. StackPanel 17
    3. WrapPanel 18
    4. DockPanel 18
    5. Grid 21
    6. GridSplitter 23
    7. Canvas 25
5. Controls (written by Csaba Biró) 27
    1. Content controls 27
        1.1. Button 27
        1.2. ToggleButton 27
        1.3. Label 28
        1.4. CheckBox and RadioButton 28
        1.5. RadioButton 28
        1.6. CheckBox 29
    2. Other controls 30
        2.1. TextBox 30
        2.2. TextBlock 30
        2.3. Image 30
        2.4. MediaElement 31
        2.5. Slider 32
        2.6. Progressbar 33
    3. List-based controls 33
        3.1. ListBox 33
        3.2. ComboBox 34
        3.3. TreeWiew 36
        3.4. Menu 38

3.5. ToolBar 41
3.6. StatusBar 43
6. Colours and Brushes (written by Biró Csaba) 47
1. Color management 47
2. Brushes 47
2.1. SolidColorBrush 47
2.2. LinearGradientBrush 48
2.3. RadialGradientBrush 49
2.4. ImageBrush 50
2.5. DrawingBrush 50
2.6. VisualBrush 51
7. Shapes (written by Csaba Biró) 54
1. Built-in shapes 54
1.1. Line 54
1.2. Polyline 55
1.3. Polygon 56
1.4. Ellipse és Rectangle 57
2. Geometry class 59
2.1. Path 59
2.2. PathGeometry 59
2.3. GeometryGroup 63
2.4. StreamGeometry 63
2.5. CombinedGeometry 65
8. Transformations (written by Csaba Biró) 68
1. TranslateTransform 68
2. SkaleTransform 69
3. RotateTransform 70
4. SkewTransform 71
5. MatrixTransform 72
6. TransformGroup 73
9. Effects (written by Csaba Biró) 77
1. Effects 77
1.1. DropShadowEffect 77
1.2. BlurEffect 78
2. BitmapEffects 79
2.1. DropShadowBitmapEffect 79
2.2. OuterGlowBitmapEffect 79
2.3. BlurBitmapEffect 79
2.4. EmbossBitmapEffect 79
2.5. BevelBitmapEffect 80
2.6. BitmapEffectGroup 80
10. Triggers (written by Csaba Biró) 83
1. Trigger 83
2. DataTrigger 84
3. MultiTrigger and MultiDataTrigger 85
4. EventTrigger 88
11. Animations (written by Biró Csaba) 90
1. Animations core classes 90
2. Other important properties 91
3. Key frame-based animation 94
4. Path-based animation 99
12. Resources and Styles (written by Gergely Kovásznai) 102
1. Resources 102
2. Styles 103
13. Data Binding (written by Gergely Kovásznai) 107
1. The Binding Class 107
2. Converters 110
3. Validation 113
14. Templates (written by Gergely Kovásznai) 117
1. Control Templates 117

2. Data Templates 120
15. LINQ (written by Gergely Kovásznai) 122
1. Lambda Expressions 123
2. Extension Methods 124
3. Comprehension Syntax 125
4. Query Operators 126
4.1. Filtering 126
4.2. Ordering 128
4.3. Projection 129
4.4. Grouping 131
4.5. Join 133
4.6. Nondeferred Operators 135
16. LINQ to XML (written by Gergely Kovásznai) 139
1. Loading XML files 141
2. Queries 142
3. XML Serialization 145
17. LINQ to Entities (written by Gergely Kovásznai) 150
1. MS-SQL and Server Explorer 150
2. Linq to SQL and Linq to Entities 153
3. Data Manipulations 157
18. Development Environments (written by Gergely Kovásznai) 159
1. Visual Studio 159
1.1. Designer 161
1.2. Toolbox and Document Outline 162
1.3. Properties 162
1.4. Transformations 163
1.5. Effects 164
1.6. Brushes 164
2. Blend 165
2.1. Triggers and Animations 165
3. Expression Design 167
19. Epilogue 170
Bibliográfia 171

1. fejezet - Prologue

At the Faculty of Sciences at the Eszterházy Károly College, the C# programming language has been being proved itself for years as a good base for teaching programming; Software Information BSc. and Teacher of Informatics MA. are prominent examples. C# is a good choice in the educational point of view, since it is a modern and clear object-oriented language, offering numerous automated solutions, due to the .NET framework behind the scenes. There exist numerous .NET-based technologies, which are quite widely used, also by major industrial giants; therefore, students can acquire competitive knowledge by moving along this line. A student can pick technologies from a wide range of state-of-the-art .NET-based ones, for developing either for desktop, web, or mobile devices.

In this lecture note, we move along the direction of developing desktop applications, and we are going to introduce a technology called the Windows Presentation Foundation (WPF). WPF was first released as a part of .NET 3.0 in 2006. Since then, of course, it gets extended and updated from one .NET release to another. In the Sections II-II, we are going to give an introduction into the world of WPF, and then, between the Sections III and XIV, this knowledge will be deepened step by step. If the reader possibly drew to developing for web or for mobile devices, then it would be really worth to get familiar with WPF, since Silverlight is a fundamental technology on either web or on Windows Phone platform; Silverlight, which came out as a subset of WPF (and was called WPF/E=„WPF/Everywhere”), is based on the same foundation and solutions.

In the lecture note, we are going to touch upon the topic of database programming as well, since each graduate programmer must possess at least basic familiarity with this topic. We are going to introduce the corresponding .NET technology, Language Integrate Query (LINQ) in the Sections XV-XVII, and we will combine it with WPF in our examples.

Finally, in Section XVIII, we will be acquainted with the development environments; nevertheless, the reader will have unwittingly become familiar with Visual Studio until reaching this section, by practicing and running through the given examples. With an eye to designers, we will give a short introduction to Expression Studio as well.

2. fejezet - Introduction (written by Csaba Biró)

Windows Presentation Foundation (WPF) is regarded as the successor of Windows Forms in desktop application development. Although WPF can be considered different than traditional Windows Forms, it is based on a number of principles which form the basis of existing desktop frameworks. The largest and the most important difference is that the code responsible for the appearance of application distinct from the code describes the functionality of the application. It is only one of the several technological innovation.

Breaking the WinForm traditions, WPF based on graphics technology now is DirectX instead of GDI/GDI+. DirectX is due to direct access to any type of user interface can be created. Complex three-dimensional graphics can be planned, but you can use the rich graphical effects for business applications (anti-aliasing, transparency, animation). Due to the hardware acceleration, the DirectX relieves the processor as possible during the graphics rendering burdens the video card (GPU) instead. They become much faster the intensive graphics tasks such as playing animations.

In the traditional Windows Forms applications the cancellation/revolution tied the developers who usually designed it for a standard definition monitor (e.g: 1024 x 768). One of the biggest problem in the case of traditional Windows applications, that the user interface was not scalable. The previous problem can be eliminated due to the WPF, because the graphic elements are not rasterized, but vector-based. Consequently, each items can be arbitrarily resized. Another great advantage is that the vector-based images occupy less space than the raster elements. However, it shouldbe noted that the WPF will continue to support raster graphics.

1. Device-independent units

The WPF generated for the dimension-treatment of the window and all the elements called device independent units (device-independent unit – DIU), which is one-ninety-sixth part of an inch. In the case of a standard Windows DPI setting (96 dpi) corresponds exactly to a phisically real pixel.

$$[Physical\ Unit\ Size\ ] = [DUI\ Size] \times [System\ DPI] \tag{2.1}$$

$$\frac{1}{96} inch \cdot 96\ dpi = 1\ pixel \tag{2.2}$$

A 19-inch LCD display for a maximum resolution of 1600 x 1200, the real pixel density can be calculated as follows:

$$[Display\ DPI] = \frac{\sqrt{1600^2 + 1200^2}\ pixel}{19\ inch} \tag{2.3}$$

It follows that the actual size of this display, a 96-pixel wide object is less than 1 inch. While a lower resolution (85 dpi) in the case of 15-inch LCD display, the preceding object is slightly larger than 1 inch.

The WPF introduced the following method of calculation to overcome this problem. Let's consider an already said to be ordinary resolution (120 dpi). In this case, 120 pixels are necessary to fill 1 inch. In WPF, the logical units can be transformed into pixels with the following calculation model:

$$= 100\ dpi \tag{2.4}$$

$$[Physical\ Unit] = [DUI\ Size] \times [System\ DPI] \tag{2.5}$$

$$= \frac{1}{96} \cdot 120 dpi$$
$$= 1.25\ pixel \tag{2.6}$$

So a 120 dpi resolution corresponds to a DIU 1.25 pixel. Thus, the previously examined 96 DIU wide button, the physical size: 120 pixels (96x 1.25=120).

Of course, other units may be used, such as „in" (inch), „pt" (pixels), „cm" (centimetre). The default unit is „px" (pixel)

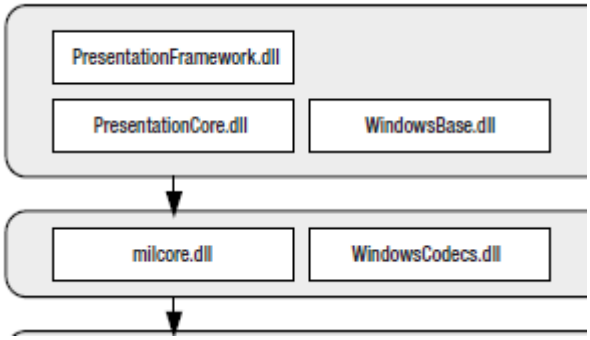
II.1 Example Units

<StackPanel>

```
    <Button Width="200" Height="30">
        <Label Content="Hello World!" FontSize="12"/>
    </Button>
    <Button Width="200pt" Height="30pt">
        <Label Content="Hello World!" FontSize="12pt"/>
    </Button>
</StackPanel>
```

2. WPF multi-layer architecture

On the top-level of WPF multi-layer architecture (Picture II. 1) you can find the PresentationFramework.dll. This is used during development, variety of drivers are implemented here. (Button, Border…), styles, etc…



II.1. WPF multi-layer architecture

The PresentationCore.dll ensures the classes for the PresentationFramework (e.g UIElement, Visual, etc.). Derived from the class, including the shape and the controls. The WindowsBase includes object classes for the basic operation of WPF (e.g. DispatcherObject, DependencyObject)

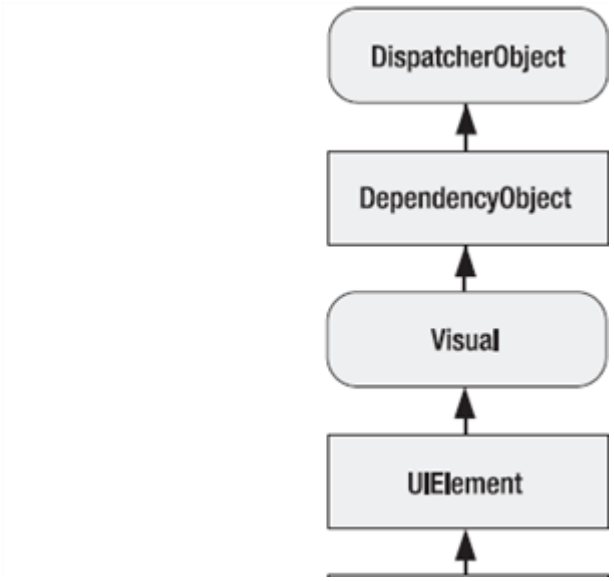
Media Integration Layer includes milcore.dll, which is the core of the WPF. Its task is to translates the higher level of graphical elements (controls and other visual elements) to DirectX elements (triangles, texture). Another component of the layer is the WindowsCodecs.dll, it is for processing, manipulatig a low-level API, primarily images (bmp, jpg, …)

The lowest layer is the Direct3D and the User32. The function of the former is to graph the defined graphical elements by the milcore on the screen, the task of the other is the treatment and management of user input.

3. WPF class hierarchy

The WPF namespaces are located in the System.Windows namespace (eg. System.Windows, System.Windows.Controls and the System.Windows.Media). An exception is System.Windows.Forms lower-namespace,which still includes the traditional GDI/GDI+ based controls.

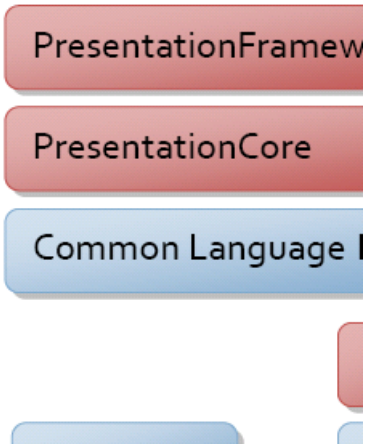
The most important WPF subsystems, their functionality and interactions between them will be displayed in the following lines.



II.2. WPF classes

3.1. System.Object

All of the classes of the WPF from the System.Object. The major components of WPF: (picture II.3) The red marked parts PresentationFramework, PresentationCore and milcore including the main WPF code details. Milcore the only one is unmanaged code written component. The main reason is to permit of close integration with DirectX. In the case of WPF displaying happens through the DirectX motor/engine. It enables efficient software and hardware rendering. The milcore engine has extremely sharpened performance, that is why it losts a number of CLR (Common Language Runtime) advantage.



3.2. System.Threading.DispatcherObject

Most of the WPF objects are from the DispatcherObject. The WPF applications use the well known single-stranded (single-thread affinity, STA) model. It means that one thread controls and monitors the entire user interface. Some objects can not be achieved safely connected directly to other fibers. It means that only the owner thread access an object created with affinity.

3.3. System.Windows.DependencyObject

Its primary task is tocompute the property's value and notify about the changes of properties to the system.

Some methods:

public void SetValue(DependencyProperty dp, object value);

public object GetValue(DependencyProperty dp);

public void ClearValue(DependencyProperty dp);

3.4. System.Windows.Media.Visual

It is the basic of the visualizable elements on the windows. The Visual class provides a link between the managed WPF libraries and the milcore.dll. Actually, the Visual class is a basic abstraction, from which all the FrameworkElement objects come from, whose primary function is to support the rendering. The UI controls, such as Button, Slider, etc… all from the Visual class.

Some methods:

protected DependencyObject VisualParent { get; }

protected void AddVisualChild(Visual child);

protected void RemoveVisualChild(Visual child);

3.5. System.Windows.UIElement

The UIElement contains the main elements of WPF (e.g StackPanel, Grid, etc..), and supports input, focus and manage events.

Some methods:

public event MouseButtonEventHandler PreviewMouseLeftButtonDown;

public event MouseButtonEventHandler MouseLeftButtonDown;

fSystem.O public static readonly DependencyProperty IsEnabledProperty;

public bool IsMouseOver { get; }

3.6. System.Windows.FrameworkElement

This is the last stage of the inheritance chain. While in the UIElement class you can define an array, its layout properties can be specified here, such as HorizontalAlignment, Width, Margin, etc.

Some methods:

public double MinHeight { get; set; }

public Style Style { get; set; }

public ResourceDictionary Resources { get; set; }

public object FindResource(object resourceKey);

public object ToolTip { get; set; }

public void BeginStoryboard(Storyboard storyboard);

3.7. System.Windows.Shapes.Shape

The basic shapes such as Rectangle, Poligon, Line derived from this class.

3.8. System.Windows.Controls.Control

The basic controls are from the Control class TextBox, Button, ListBox, etc. It also provides options for the additional setup of the controllers. (e.g Font, Background, etc.)

Some methods:

public ControlTemplate Template { get; set; }

public Brush Background { get; set; }

public FontFamily FontFamily { get; set; }

3.9. System.Windows.Controls.ContentControl

The System.Windows.Controls.ContentControl class allows to give rich content to the controllers.

II.2 Example ContentControl

```
<Button>
  <StackPanel>
    <Ellipse Height="40" Width="40" Fill="Blue"/>
    <TextBlock TextAlignment="Center">WPF</TextBlock>
  </StackPanel>
</Button>
```

3. fejezet - XAML (written by Csaba Biró)

XAML (eXtensible Application Markup Language) is an XML-based declarative markup language, which simplifies the creation of the graphical user interface (GUI) in the .NET model.

The grammatical system of rules of the declarative language rules is very easy. Its general design principle is that every element of the XAML document – unless it defines an attribute – is a copy of the .NET class. The implementation of the XAML files can be carried out with interpretation or translation. Here is an example of the interpretation performing.

After starting a simple editor (e.g. Notepad), type the following code:

```
<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

   <TextBlock Text="Hello World!" />

</Page>
```

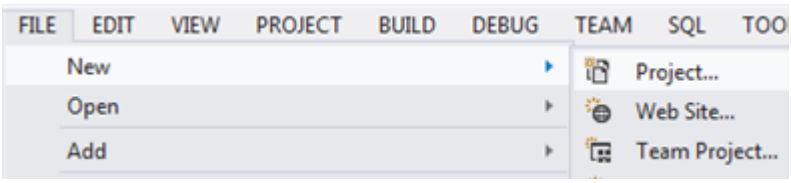
Save it as HelloWorld.xaml, and open this file in a browser (IE, Firefox). We created our first application in XAML language running in XAML browser.

The implementation with another translation is more common. So, if you want to embed codes written in C# or Visual Basic language, our passcode have to be translated.

Let's look an example of it, start the Visual Studio. We are going to work in this developing environment further, so the chapter: 'Error: The source of the reference is not found.' recommend to read.

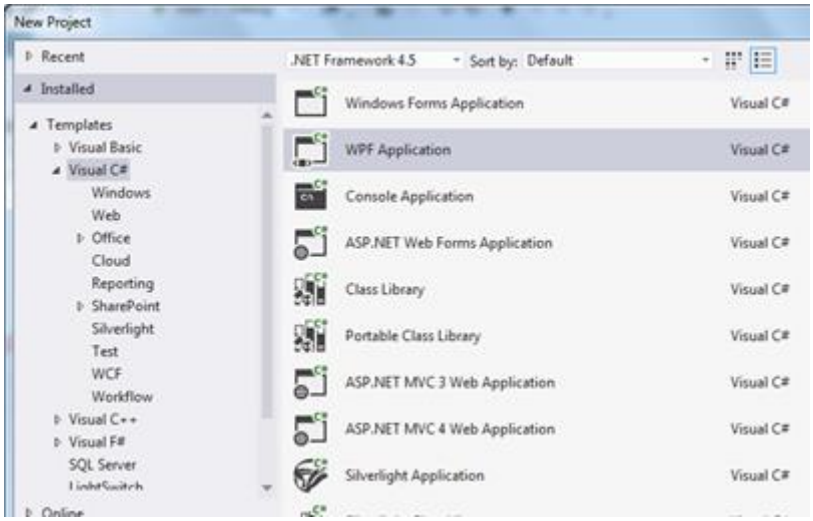
Now the following steps are performed in the Visual Studio:

1. File / New Project

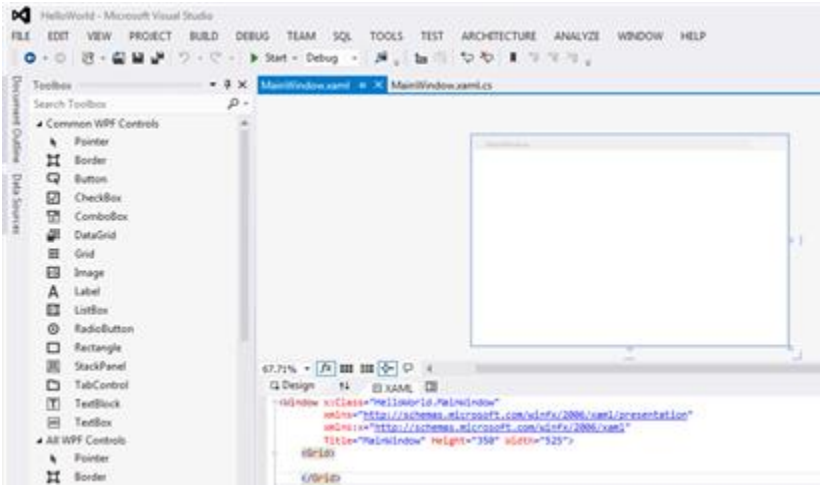


III.1. New project

1. New WPF Application / Name: HelloWorld



III.2. New WPF application



III.3. Visual Studio IDE

1. Type the following line between Grid controls.

<TextBlock Text="Hello World!" />

Application code is:

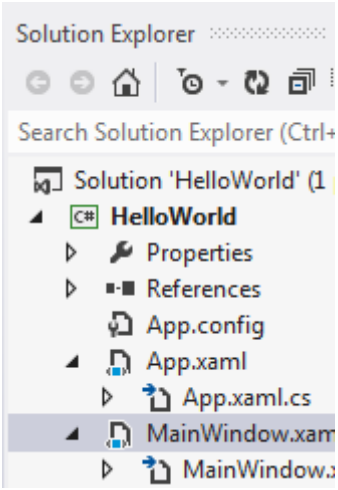
```
<Window x:Class="HelloWorld.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <TextBlock Text="Hello World!" />
    </Grid>
</Window>
```

Run the project. In this case we translate the XAML code (in the example) into the resulting executable file as well.

Note that in the first case the root is the Page (for web pages) element, whereas in the second one the Window element. We will make only desktop applications longer, which has Window root element.

1. The fundamental files of WPF project

Look through the fundamental files of a WPF project with returning to the HelloWorld examples.



III.4. Solution Explorer

In the Solution Explorer (Chapter XVIII.1.1) an App.xaml file can be found with the MainWindow.xaml file has been used above, its content is:

<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>

The application resources and startup settings can be defined in the App.xaml file, which begins with the Application root element.

The StartupUri property point to the first visualized window.

StartupUri="MainWindow.xaml"

We will deal with the management styles and resources more in Chapter XII.

2. Code-behind class

It can be observed that when we create our project from the basis of the WPF application template, a .cs or a .vb extension file created with the same name for both of the xaml files.

The aim of the foreground code files to separate the application appearance from the application functionality through the development, as you could read in the introduction.

It becomes available with using the x:Class attribute.

x:Class="HelloWorld.MainWindow"

Actually, what happens is that the x:Class attribute tells the XAML parser has to create a new lass with the specified name. In other words the former one creates a Window class called MainWindow from the Window class.

Then the content of the MainWindow.xaml.cs file is the following:

```
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();

      }

   }

}
```

3. XAML namespaces

The previous examples shows that the Page and the Window root elements – in any case – define two namespaces:

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

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

1. The default WPF namespace including the WPF classes (controls) need for building of the user interface.

2. XAML –namespace. It includes general definitions necessary for the interpretation of the XAML documents. It is interesting that the namespace with a prefix x has a wider view.

The role of the xmlns – a special attribute – is to give a local name (pseudonym) to the URI (Uniform Resource Locator) form namespace.

4. Properties

As it has already been mentioned above, a class properties (attributes) definied in a XAML file, the same with the features of an object element. Of course, it takes a number of ways because of the characteristic of this particular property.

Look at the following example for the interpretation displaying a button.

```
<Window x:Class="Tulajdonságok.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>
  <Button x:Name="Gomb"
   Content="Gomb"
   Width="150" Height="30"
   HorizontalAlignment="Center" VerticalAlignment="Top"
   Background="Azure" Foreground="Blue"
   FontFamily="Times New Roman" FontSize="20"
   FontStyle="Italic" FontWeight="Heavy" Opacity="0.5"/>
</Grid>
</Window>
```

The Button element in the example is a „member” of the System.Windows.Controls. The features of the Button element represent the object’s properties, that is why we can assign values to the following characteristics (Content, Width, Height, HorizontalAlignment, VerticalAlignment, Background, foreground, FontFamily, FontSize, FontStyle, FontWeight, Opacity)

However, it is important to note, that the x:Name is not the feature of the Button object, but a feature which assigns a unique identifier to the object.

If an object is associated with only simple type values, can be defined with an abbreviated form shown in the example below.

<Button X:Name ="Gomb" Background = "Blue" />

The previous button looks like this in C# language:

```
Button button = new Button();
    button.Name = "Gomb";
    button.Content = "Gomb";
    button.Width = 150;
    button.Height = 30;
    button.HorizontalAlignment = HorizontalAlignment.Center;
    button.VerticalAlignment = VerticalAlignment.Top;
    button.Foreground = Brushes.Blue;
    button.Background = new SolidColorBrush(Colors.Azure);
    button.FontFamily = new FontFamily("Times New Roman");
    button.FontSize = 20;
    button.FontStyle = FontStyles.Italic;
    button.FontWeight = FontWeights.Heavy;
    button.Opacity = 0.5;
```

5. Advanced properties

If we would like to assign a complex valued property (e.g background with gradient fill), the use of the abbreviated forms is not enough like in the former example. The complex features can be given with children-items called feature-elements.

Feature elements –syntax is as follows:

<classname.propertyname>

Complementing the previous example code:

```
<Button x:Name="Gomb"
    Content="Gomb"
    Foreground="Blue"
    Width="150" Height="30"
    HorizontalAlignment="Center" VerticalAlignment="Top"
    FontFamily="Times New Roman" FontSize="20"
    FontStyle="Italic" FontWeight="Heavy" Opacity="0.5">
    <Button.Background>
```

```
            <LinearGradientBrush EndPoint="1,0.5" StartPoint="0,0.5">
               <GradientStop Color="Black" Offset="0" />
               <GradientStop Color="White" Offset="1" />
            </LinearGradientBrush>
         </Button.Background>
   </Button
```

6. Content property

It is possible to enter the hidden Content property, which we cannot see in the previous examples.

```
<Button>
Button
</Button>
```

The Content property has an object type, that is why (as the example shows) a gradient filled ellipse can be placed on the button instead of a simple text.

```
   <Button Width="150" Height="30" Background="Yellow">
      <Button.Content>
         <Ellipse Width="20" Height="20">
            <Ellipse.Fill>
               <RadialGradientBrush>
                  <GradientStop Color="Red" Offset="0" />
                  <GradientStop Color="#FFD3FF42" Offset="1" />
               </RadialGradientBrush>
            </Ellipse.Fill>
         </Ellipse>
      </Button.Content>
   </Button>
```

7. Markup Language Extensions

Actually, most of the features are described conveniently by the XAML syntax. Howewer, it cannot be satisfactory. (For example: an object you want to set property value that already exists, or would like to seet a value with a dynamic data binding, etc.) In these cases, markup extensions will be required. Markup language extensions have to be placed between curly {} braces.

```
{MarkupLangueExtensionName value}
```

The name of the Markup Language Extension defines for the WPF which extension is it, for example: StaticResource, DynamicResource, etc.

```
<Application.Resources>
```

```
    <SolidColorBrush x:Key="MyBrush" Color="Gold"/>
</Application.Resources>
```

A unique key can be assigned to the resources created in the ResourceDictionary. More about the resources in the Chapter 0.

```
<Button Background="{StaticResource MyBrush}"/>
<Ellipse Fill="{StaticResource MyBrush}"/>
```

If more than one parameter is required to specify the following notation:

{MarkupLanguageExtentionName Parameter1=value1, parameter2=value2, parameter3=value3}

8. More x:prefixes

The x:Name, x:Class, x:Key members may have already been mentioned above, if the task requires other prefix may also be used.

x:Type: Type reference can be created

x:Static: a reference to a static value allowed

9. Special characters and whitespaces

The XAML follows the XML rule system. So small and capital letter- sensitive, which should particularly pay attention to objects, properties and attributes specification.

Depending on the currently used converter is not always true for the values. The Boolean converter completely waived from this convention. The XAML parser ignores the irrelevant whitespace, normalizes the importants.

9.1. Explanation

The following complex symbols (four characters) are necessary to open <!— , and these are need to be closed --> the explanations. There is only one restriction for the explanatory test: it must not contain two consecutive hyphens characters except there are spaces between them.

9.2. Character entities

Of course, as in the case of XML, the <>;'& symbols define structure definitions. If these signs in our documents will not be used as structure descriptive, the following entities correspond to the special characters.

Speciális karakter	Karakter entitás
Less than (<)	<
Greater than (>)	>
And (&)	&amp;
Quotation marks (")	"

If you want to create a button with Margin&Padding subtitles, it can be done in the following way

```
<Border>
   <Button Content="Margin &amp;amp; Padding"/>
</Border>
```

4. fejezet - Layouts (written by Csaba Biró)

The planning and execution of the application user interface has to be attractive and practical, and it also has to adapt to different window sizes, not an easy task.

A great advantage of the WPF to support the solving of these situations. The majority of elements used for the user interface creation are from the System.Windows.FrameworkElement – as has been previously were involved.

1. Alignment, margins

You can find the properties with which the position of the child elements can be set precisely. We will get to know only the four most important among them (Margin, Padding, HorizontalAlignment, VerticalAlignment).

1.1. Inner and outer margins

With the help of inner and outer margins we can set the distance between the child elements. While with the Margin property you can specify the distance that can be measured on the outside of the element, and the Padding determines the distance in an element which must be free. However, it is important to note that the Margin property is inherited by all the classes from the FrameworkElement class, but the Padding property can be set only for the elements from the Control Class.

The inner and the outer margins has to be set with 1, 2 and 4 values.

If you want to set the same margin settings on all the sides:

Margin="10"

In the case of two numbers the left and the right side, and the second indicates the top and bottom margins.

Margin="10 20"

Négy szám esetében a számok a bal, felső, jobb és alsó margókat jelentik.

Margin="10 20 30 40"

For four numbers means the left, top, right and bottom margins. At accurate values a decimal point is used, the elements may be separated from each other by commas.

Margin="10.25, 2.5, 4.09, 3"

IV.1 Example Inner and outer margins

IV.1. Inner and outer margins

<Grid Height="200" Width="400" Background="Green">

```
    <Border Margin="10 20 30 40" Background="Yellow">
        <Border Padding="40 30 20 10">
            <Button Content="Margin &amp;amp; Padding" Background="Brown" Foreground="White"/>
        </Border>
    </Border>
</Grid>
```

1.2. Adjustments

The individual child elements can be aligned vertically and horizontally, of course.

Possible values for horizontal alignment: Left, Center, Right, Strech; for vertical alignment: Top, Bottom, Center, Stretch.

2. StackPanel

The StackPanel is one of the simplest among layout controls, in many cases the most useful layout control. It places its inside elements list-like arrangement (one under the other). It is enought to give the height of the elements, because their width is adapted to the StackPanel width.

To demonstrate the operation of the StackPanel, look at the following two examples:

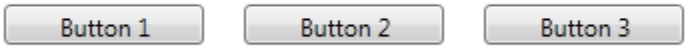
Example IV.2 StackPanel

IV.2. StackPanel

```
<StackPanel Width="100">
    <Button Height="20" Content="Button 1" Margin="10"/>
    <Button Height="20" Content="Button 2" Margin="10"/>
    <Button Height="20" Content="Button 3" Margin="10"/>
</StackPanel>
```

If you would like to visualize the elements arranged next to each other (Orientation =”Horizontal”), it is enough to give the width of the elements.

Example IV.3 StackPanel



IV.3. StackPanel

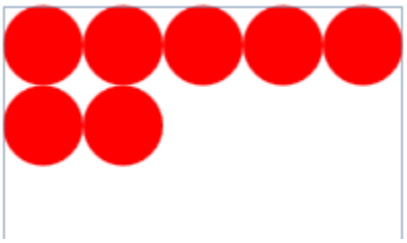
```
<StackPanel Height="20" Orientation="Horizontal">
```

```
    <Button Width="100" Content="Button 1" Margin="10 0 10 0"/>
    <Button Width="100" Content="Button 2" Margin="10 0 10 0"/>
    <Button Width="100" Content="Button 3" Margin="10 0 10 0"/>
</StackPanel>
```

3. WrapPanel

It is suitable for displaying items alongside or next to each other. If an item does not fit on the line, it is automatically placed on the next one. The width and the height of the elements stored in this panel is optional.

Example IV.4 WrapPanel



IV.4. WrapPanel

```
<Grid Width="200" Height="200">
    <WrapPanel>
        <Ellipse Fill="Red" Height="40" Width="40"/>
        <Ellipse Fill="Red" Height="40" Width="40"/>
        <Ellipse Fill="Red" Height="40" Width="40"/>
        <Ellipse Fill="Red" Height="40" Width="40"/>
        <Ellipse Fill="Red" Height="40" Width="40"/>
        <Ellipse Fill="Red" Height="40" Width="40"/>
        <Ellipse Fill="Red" Height="40" Width="40"/>
    </WrapPanel>
</Grid>
```

If Vertical Orientation attribute is set, the stored items will be under each other.

```
<WrapPanel Orientation="Vertical">
```

4. DockPanel

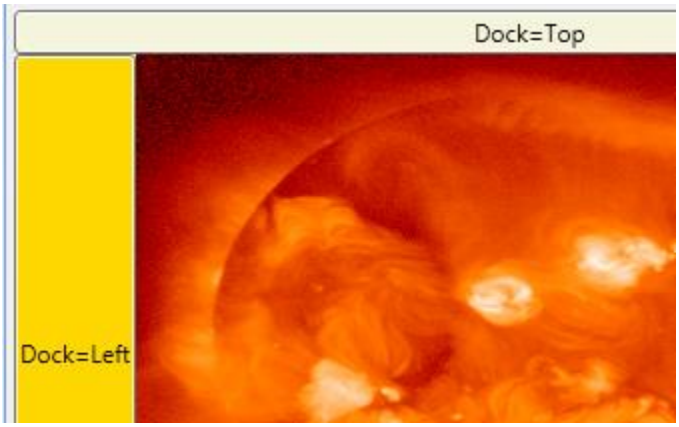
The DockPanel compared to the StackPanel and the WrapPanel can be used to design more complex layouts, and as a root element replaced the DataGrid. With the help of DockPanel.Dock's features, the location of each child elements can be set inside the DockPanel.

```
    <Grid>
        <DockPanel>
            <StackPanel DockP
        </DockPanel>
```

IV.5. DockPanel.Dock

Create the two applications shown below to understand the DockPanel.

Example IV.5 DockPanel



IV.6. DockPanel

```
<DockPanel LastChildFill="True">
    <Button Content="Dock=Top" DockPanel.Dock="Top" Background="Beige"/>
    <Button Content="Dock=Right" DockPanel.Dock="Right" Background="Gold"/>
    <Button Content="Dock=Left" Background="Gold"/>
    <Button Content="Dock=Bottom" DockPanel.Dock="Bottom" Background="Beige"/>
    <Image Source="Nap.gif" Stretch="Fill"/>
</DockPanel>
```

LastChildFill (True or False) property specifies that the last element fills or not the available space.

Example IV.6 Saturn - DockPanel

IV.7. DockPanel

```
<Window x:Class="Bolygok.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Saturn" Height="400" Width="600" Background="Yellow">
    <DockPanel LastChildFill="True">
      <Border DockPanel.Dock="Top" Height="80"
        CornerRadius="10" Margin="10">
        <TextBlock HorizontalAlignment="Center"
          VerticalAlignment="Center" FontSize="36"
          FontFamily="Book Antiqua" Foreground="White">
        Saturn
        </TextBlock>
        <Border.Background>
          <ImageBrush ImageSource="/Planets;component/Images/Galaxy1.jpg"/>
        </Border.Background>
      </Border>
      <Border DockPanel.Dock="Bottom">
      </Border>
          <Border Height="50" Background="Black" BorderBrush="Black" BorderThickness="1"
DockPanel.Dock="Bottom">
        <StackPanel Orientation="Horizontal">
```

```
            <TextBlock Foreground="White" Width="580" TextWrapping="Wrap" Text="
```

Saturn is the sixth planet from the Sun and the second largest planet in the Solar System, after Jupiter. Named after the Roman god of agriculture, Saturn, its astronomical symbol (♄) represents the god's sickle. Saturn is a gas giant with an average radius about nine times that of Earth.” />

```
            </TextBlock>

          </StackPanel>

        </Border>

                                <Border Background="Black" CornerRadius="10" Margin="10" Padding="10" DockPanel.Dock="Left">

          <StackPanel Background="Black">

                                                <TextBlock Text="Data" Foreground="White" HorizontalAlignment="Center" FontSize="16" />

            <Separator/>

            <TextBlock Text="Mass" Foreground="White"/>

            <TextBlock Text="95,184 (earthmass)" Foreground="White"/>

            <Separator/>

            <TextBlock Text="Semi-major axis" Foreground="White"/>

            <TextBlock Text="1427,0 million km" Foreground="White"/>

            <Separator/>

            <TextBlock Text="Orbital period" Foreground="White"/>

            <TextBlock Text="10759,2 days" Foreground="White"/>

            <Separator/>

            <TextBlock Text="Orbital velocity" Foreground="White"/>

            <TextBlock Text="9,64 km/s" Foreground="White"/>

          </StackPanel>

        </Border>

         <Border Background="Black" BorderBrush="Black" BorderThickness="1" CornerRadius="10" Margin="10">

          <Image Source="/Bolygok;component/Images/Szaturnusz.jpg" />

        </Border>

      </DockPanel>

  </Window>
```

5. Grid

With the help of the Grid control the above mentioned layouts can be real. One of the most commonly used controls.

The following example includes the definition of a grid with two columns and three rows.

```
<Grid ShowGridLines="True">
<Grid.RowDefinitions>
  <RowDefinition />
  <RowDefinition />
  <RowDefinition />
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
  <ColumnDefinition />
  <ColumnDefinition />
</Grid.ColumnDefinitions>
</Grid>
```

The RowDefinitions and the ColumnDefinitions are elements to define the rows or columns.

During designing and testing the value of ShowGridLines should be set True. In this case, symbolic lines are drawn in the grid when it runs.

Paste three elements into this structure.

It is important to note that the rows and colums numbering starts from zero.

```
<Button Content="0/0" Width="30"/>
<Label Grid.Row="1"
       Content=" In fact, I'm in the second row of the first column!" />
<Calendar Grid.Row="3" Grid.Column="2" />
```

The key will be located in the cell [0,0], because of the undefined location. In the case of Label the line is defined, so it will be located the cell [0,1], while the calendar can be found in the second line of the third column. That is why the rows and columns in proportion to share the width and height of the form. Of course, the size of each rows and columns can be adjusted precisely.

```
<Grid.RowDefinitions>
  <RowDefinition Height="20"/>
  <RowDefinition  Height="1*"/>
 <RowDefinition Height="2*"/>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
   <ColumnDefinition Width="Auto"/>
   <ColumnDefinition />
</Grid.ColumnDefinitions>
```

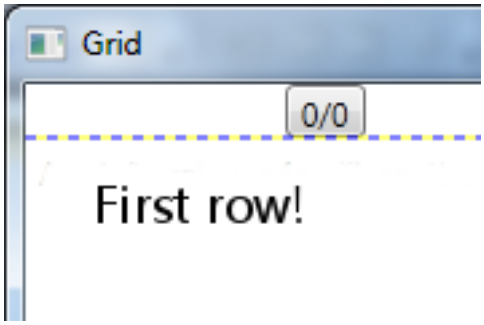
In our example the first line –height will be 20 pixels, while the first (second) and the second (third) one shares the remaining place at a ratio of 1:2.

<RowDefinition Height="1*"/>

<RowDefinition Height="2*"/>

In this case with the "auto" value the width of the zero-column takes the biggest width driver value among the driver with the content of the column.

<ColumnDefinition Width="auto"/>



IV.1. Grid

With the help of RowSpan and ColumnSpan instructions it is possible to combine rows and columns. These will be discussed in the following example.

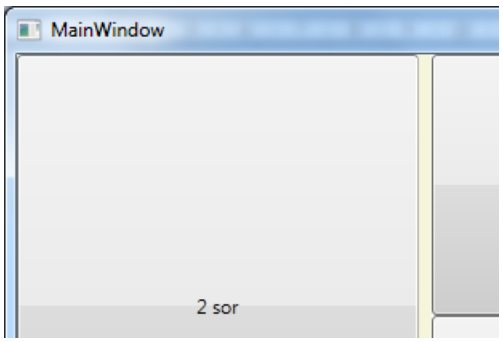
6. GridSplitter

Using GridSlitter control is possible during the program running to resize the rows and columns of the grid. It has to be placed between the rows and columns which we would like to resize. The ResizeDirection property can be used when we want to resize the rows or columns, the function of the ResizeBehaviour is to set the exact operation.

ResizeBehavior property:

1. BasedOnAlignment
2. CurrentAndNext
3. PreviousAndCurrent
4. PreviousAndNext

Example IV.7 Grid és GridSlitter



IV.8. Grid and GridSplitter

<Window x:Class="grid.MainWindow"

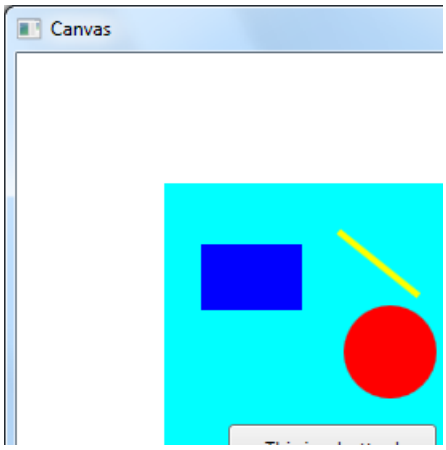
```
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Grid" Height="300" Width="500">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition Height="150"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="2*"/>
      <ColumnDefinition Width="auto"/>
      <ColumnDefinition Width="*"/>
      <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <Button Grid.RowSpan="2" Content="2 sor"/>
    <GridSplitter Grid.Row="0"
                  Grid.RowSpan="2"
                  Grid.Column="1"
                  Width="8"
                  Background="Beige"
                  ResizeBehavior="PreviousAndNext"
                  ResizeDirection="Columns" />
    <Button Grid.Column="2"
            Grid.ColumnSpan="2"
            Content="2 oszlop"/>
    <Button Grid.Row="1"
            Grid.Column="2"
            Content="1,2" />
    <Button Grid.Row="1"
            Grid.Column="3"
            Content="1,3" />
  </Grid>
```

</Window>

7. Canvas

The Canvas pixel delivers precise layout for ideal fixed-size applications. The elements' position on Canvas can be done by setting features Top-Left, and the Bottom-Right. It is important to note that Canvas is designed to accommodate drawings - controls have to be avoided here.

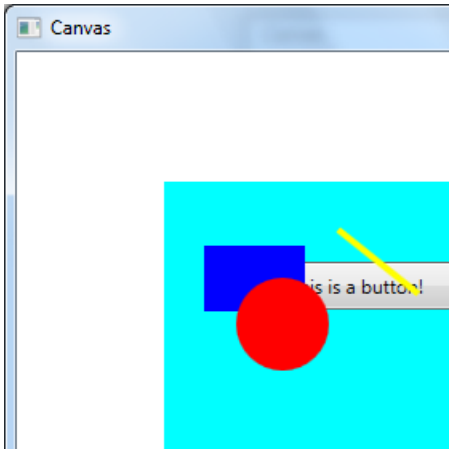
Example IV.8 Canvas



IV.9. Canvas

```
<Window x:Class="Canvas.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Canvas" Height="400" Width="400">
    <Grid>
        <Canvas Height="200" Width="200" Background="Aqua">
                                        <Rectangle Canvas.Left="23" Canvas.Top="38" Width="63" Height="41" Fill="Blue"/>
            <Ellipse Canvas.Left="112" Canvas.Top="76" Width="58" Height="58" Fill="Red" />
            <Line Canvas.Right="40" Canvas.Top="30" X2="50" Y2="40"
                Stroke="Yellow" StrokeThickness="4"/>
            <Button Canvas.Bottom="20" Canvas.Left="40" Width="130"
                 Height="30" Content="This is a button!"></Button>
        </Canvas>
    </Grid>
</Window>
```

We have an opportunity a Z coordinate assignment for each item with the Zindex feature. A higher index elements appear above the indices are lower.



IV.10. ZIndex

```
<Canvas Height="200" Width="200" Background="Aqua">
  <Rectangle Canvas.Left="25" Canvas.Top="40" Width="63" Height="41" Fill="Blue"
        Canvas.ZIndex="1"/>
  <Ellipse Canvas.Left="45" Canvas.Top="60" Width="58" Height="58" Fill="Red"
      Canvas.ZIndex="2"/>
  <Line Canvas.Right="40" Canvas.Top="30" X2="50" Y2="40"
    Stroke="Yellow" StrokeThickness="4"
    Canvas.ZIndex="3"/>
  <Button Canvas.Bottom="120" Canvas.Left="55" Width="130"
      Height="30" Content=" This is a button!"/>
</Canvas>
```

5. fejezet - Controls (written by Csaba Biró)

In this chapter the using of the basic controls have to be discussed including the advantages of user controls. A lot of WPF tools provide elegant and dynamic design of user interfaces.

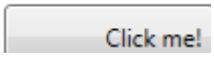
1. Content controls

Controls in this group (Button, Label, Checkbox, RadioButton) are from the ContentControl class. They contain a special embedded element (content attribute) which is actually an object type, so any content can be placed in it.

1.1. Button

Push button have already been mentioned on the previous chapters. Only the Content feature will be described here.

Exemple V.1 Click me!



V.1. Click me

```
<Button x:Name="button" Content="Click me!" Margin="180 80 180 200" />
```

V.2 Példa Solar Sytem

V.2. Solar Sytem

```
<Button x:Name="buttonStackPanel"  Margin="180,80,180,135" Background="Black">
    <StackPanel>
        <Image Source="solarsystem.jpg" Stretch="Fill"/>
        <Label Content="Solar System" HorizontalAlignment="Center"
            FontWeight="Bold"
            FontStyle="Italic"
            Foreground="White"/>
    </StackPanel>
</Button>
```

1.2. ToggleButton

A special feature-button, which is used for the designation of some options (on-off), typical well-used toolbars – it will be discussed later.

<ToggleButton Width="35" Height="35" Content="B" FontWeight= "Bold" />

1.3. Label

The Label control is one of the simplest controls inWPF – as it occured in our examples many times. What is important to note, that Label includes - like many controls – built-in support for mnemonic keys placement

<Label Content="Press Alt+_L"/>

In the following example, through the Target feature, at runtime the textBox1 will be on focus by pressing the Alt+N key combination.

Example V.3 Target property

V.3. Target property

```
<StackPanel Orientation="Horizontal"
        HorizontalAlignment="Center"
        VerticalAlignment="Center">
    <Label Content="_Név:" Target="{Binding ElementName=textBox1}" />
    <TextBox x:Name="textBox1" Width="300" Height="30" />
</StackPanel>
```

Using the Binding element we can create a data binding, see Chapter XIII.

1.4. CheckBox and RadioButton

The input data can include not only text data input, but there is a possibility for the simple input of select values.

The selection of the options can be done by the following two controls:

CheckBox,

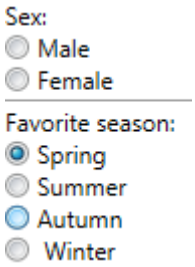
RadioButton.

Both of them are the descendants of ButtonBase class.

1.5. RadioButton

The radio buttons allow you to choose options in a way that we can choose only one (exactly) from the mutually exclusive options with the help of them.

Example V.4 RadioButton



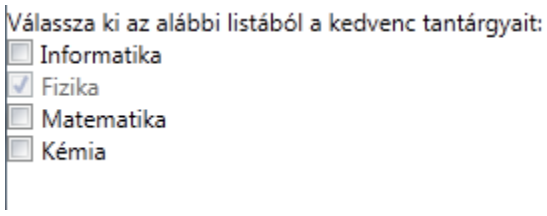
V.4. RadioButton

```
<StackPanel>
    <TextBlock Text="Sex:" />
    <RadioButton GroupName="Sex" Content="Male" />
    <RadioButton GroupName="Sex" Content="Female"/>
    <Separator/>
    <TextBlock Text="Favorite season:" />
    <RadioButton GroupName="season " Content="Spring" IsEnabled="True"/>
    <RadioButton GroupName="season " Content="Summer"/>
    <RadioButton GroupName="season " Content="Authum"/>
    <RadioButton GroupName="season " Content="Winter"/>
</StackPanel>
```

1.6. CheckBox

Checkboxes allow you to choose between one or more independent modes different from radio buttons, in which only one setting can be chosen. With the help of check boxes a variety of settings is possible at the same time. Key features are IsChecked and IsEnabled. The first assign it to default status, the second one does not allow you to edit.

Example V.5 CheckBox



V.5. CheckBox

```
<StackPanel>
    <TextBlock Text="Válassza ki az alábbi listából a kedvenc tantárgyait:"/>
    <CheckBox Content="Informatika"/>
    <CheckBox Content="Fizika" IsChecked="True" IsEnabled="False"/>
    <CheckBox Content="Matematika"/>
```

```
    <CheckBox Content="Kémia"/>
</StackPanel>
```

2. Other controls

Controls in this group do not have any Content features. Specially suited for a specific task, such as Image control for displaying images, or the TextBlock control for texts.

2.1. TextBox

Control for entering and displaying texts. TextBox can be created by the following syntaxes:

```
<TextBox />
<TextBox Text="TextBox!"/>
<TextBox>TextBox</TextBox>
```

If you only want to use it for displaying texts, the value of the IsReadOnly feature has to be ”True”.

```
<TextBox IsReadOnly="True" Text="Read only text!"/>
```

If you would like to the text clicking automatically, set for Wrap the TextWrapping feature.

```
<TextBox TextWrapping="Wrap" Text=" Wrapping text " Height="40"/>
```

The ScrollBar can be visible with using Visible value of the VerticalScrollBarVisibility feature.

```
<TextBox VerticalScrollBarVisibility="Visible" TextWrapping="Wrap" Text="Wrapping text" Height="100"/>
```

2.2. TextBlock

The TextBox control is used for displaying a relatively small amount of texts or even formatted content can be done by the following syntaxes, which does not support hot keys.

```
<TextBlock>
    TextBlock
</TextBlock>
<TextBlock Text="TextBlock" TextWrapping="Wrap" Width="40"/>
```

2.3. Image

Image control is used for displaying images. The most important feature is the Source. Using this feature it is possible to attribute the file location with Uniform Resource Identifier (URI) and relative reference.

```
<Image Source="C:\Kepek\Föld.jpg" />
<Image Source="Föld.jpg" Width="100" Height="100" Stretch="Fill"/>
```

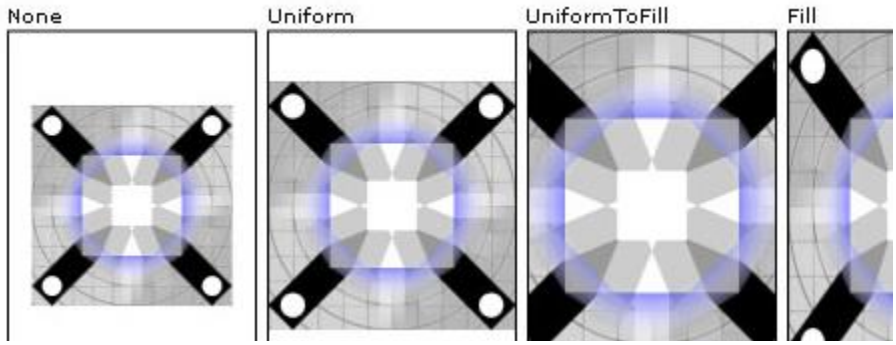
Strech property:

None – The image is displayed in its original size and cut the part of the image, which does not fit within the designated area.

Fill – Fills the selected area of the image passing over the original aspect ratios, so the picture may be distorted.

Uniform – Completes the selected area with the image, retention the aspect ratios (proportion)

UniformToFill – Fills completely the selected area with the image, retention the proportion.

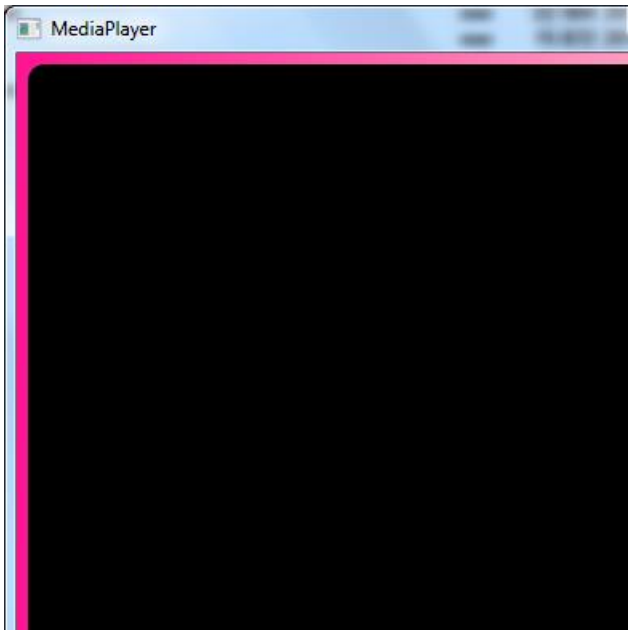


V.6. Strech property

2.4. MediaElement

The MediaElement allows you to play various multimedia files supporting any types like Windows Media Player 10 do.

Example V.6. MediaPlayer



V.7. Media Player

```
<Grid x:Name="Player">
    <Border Margin="7" Background="Black" CornerRadius="10">
        <MediaElement x:Name="Media" Margin="10"
                Volume="{Binding ElementName=slidVolume, Path=Value}"
                Balance="{Binding ElementName=slidBalance, Path=Value}"
                MediaOpened="Media_MediaOpened"
                MediaEnded="Media_MediaEnded"
                LoadedBehavior="Manual"
```

```
            MouseLeftButtonUp="Media_MouseLeftButtonUp"/>
    </Border>
</Grid>
```

Open media:

```
    private void btnBrowse_Click(object sender, RoutedEventArgs e)
    {
        OpenFileDialog dlg = new OpenFileDialog();
        Nullable<bool> result = dlg.ShowDialog();
        if (result == true)
            Media.Source = new Uri(dlg.FileName);
    }
```

Paly media:

```
    private void btnPlay_Click(object sender, RoutedEventArgs e)
    {
        Media.Play();
        dispTimer.Start();
}
```

Pause media:

```
    private void btnPause_Click(object sender, RoutedEventArgs e)
    {
       Media.Pause();
    }
```

2.5. Slider

The sliders allow you to enter a setting within a specified range of values.

Properties:

IsDirectionReserved – the minimum value is assigned to the left side, the maximum one to the right side of the slider by default. If this feature is set to „True”, the two sides will be swapped.

IsEnabled – allows to enable or disable the slider

LargeChange – you can adjust the step size for PageUp and PageDown keys

Maximum – the maximum value of the slider

Minimum – the minimum value of the slider

Orientation – the slider orientation can be adjusted

SmallChange – the increment can be adjusted for cursor keys

Value – Current value, which is always between the minimum and the maximum

Example V.7 Slider

```
<StackPanel>
   <Slider x:Name="slider1" Width="100" Value="50" Minimum="10" Maximum="100"/>
   <Image Source="C:\Pictures\Earth.jpg" Height="{Binding ElementName=slider1, Path=Value}"
        Width="{Binding ElementName=slider1, Path=Value}" />
</StackPanel>
```

2.6. Progressbar

We can find it as the element of the StatusBar. We will look at some examples for it later.

Properties:

IsEnabled – allows you to enable or disable of the progress bar,

LargeChange – allows you to set large step,

Maximum – the maximum value of the progress bar,

Minimum – the minimum value of the progress bar ,

Orientation – used to help the orientation of the progress bar ,

SmallChange- allows you to set small step ,

Value - Current value, which is always between the minimum and the maximum.

Example V.8 ProgressBar

```
<ProgressBar x:Name="ProgressBar1" Width="200" Height="30"  Value="40"/>
```

3. List-based controls

The list-based controls provide the well-known conventional services for us. The most important list-based controls (ListBox, ComboBox, TreeView, Menu, StatusBar, ToolBar).

3.1. ListBox

The ListBox control allows you to choose only one element by default.

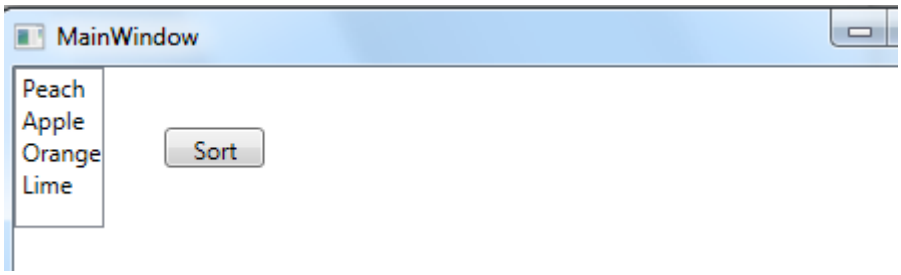
V.9 Példa ListBox

```
<ListBox x:Name="Lista" SelectionMode="Extended">
        <ListBoxItem>Peach</ListBoxItem>
        <ListBoxItem>Apple</ListBoxItem>
        <ListBoxItem>Orange</ListBoxItem>
        <ListBoxItem>Lime</ListBoxItem>
</ListBox>
```

Key features:

1. SelectedIndex – returns the index of the selected element (position in list)
2. SelectedItem – the name of the selected item is returned
3. IsSelected – it has a positive value if the current element is in an assigned state.
4. Single – allows you to select an item
5. Multiple – allows you to select multiple items
6. Extended – it is also allows you to select multiple items, but it provides option to choose listings which are not under each other with pressing the Ctrl button.

Example V.10 ListBox



V.8. ListBox

```
<StackPanel Orientation="Horizontal" VerticalAlignment="Top">
      <ListBox x:Name="List1" SelectionMode="Extended">
         <ListBoxItem>Peach</ListBoxItem>
         <ListBoxItem>Apple</ListBoxItem>
         <ListBoxItem>Orange</ListBoxItem>
         <ListBoxItem>Lime</ListBoxItem>
      </ListBox>
         <Button x:Name="sortButton" Width="50" Height="20" Margin="30"
                  Content="Sort" Click="sortButton _Click">
         </Button>
 </StackPanel>
```

Code-behind:

```
private void sortButton_Click(object sender, RoutedEventArgs e)
      {
         List1.Items.SortDescriptions.Add(new System.ComponentModel.SortDescription("Content",
      System.ComponentModel.ListSortDirection.Ascending ));
      }
```

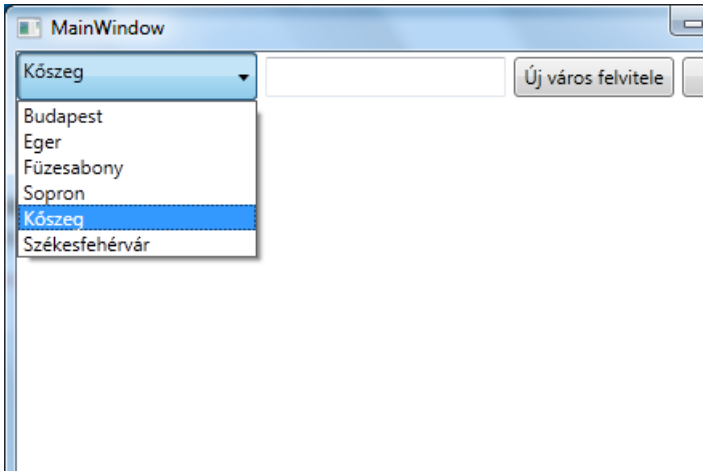
3.2. ComboBox

The ComboBox is like LisBox, a drop-down list.

Example V.11 Példa ComboBox

```
<ComboBox x:Name="comboBox1" IsDropDownOpen="True">
        <ComboBoxItem>This</ComboBoxItem>
        <ComboBoxItem>is</ComboBoxItem>
        <ComboBoxItem>a</ComboBoxItem>
        <TextBlock>ComboBox!</TextBlock>
</ComboBox>
```

Example V.12 ComboBox



V.9. ComboBox

```
<StackPanel Orientation="Horizontal" VerticalAlignment="Top">
      <ComboBox Name="list1" Width="150" Height="30" Margin="0 0 5 0">
         <ComboBoxItem Content="Budapest"></ComboBoxItem>
         <ComboBoxItem Content="Eger"></ComboBoxItem>
         <ComboBoxItem Content="Füzesabony"></ComboBoxItem>
         <ComboBoxItem Content="Sopron"></ComboBoxItem>
         <ComboBoxItem Content="Kőszeg"></ComboBoxItem>
         <ComboBoxItem Content="Székesfehérvár"></ComboBoxItem>
      </ComboBox>
      <TextBox Name="tb1" Height="25" Width="150"/>
      <Button Name="addButton" Width="100" Height="25"
       Content="Add new city" Click="addButton_Click" Margin="5 0 5 0"/>
      <Button Name="deleteButton" Content="Delete city" Height="25"
       Width="100" Click="deleteButton_Click"/>
</StackPanel>
```

Code-behind:

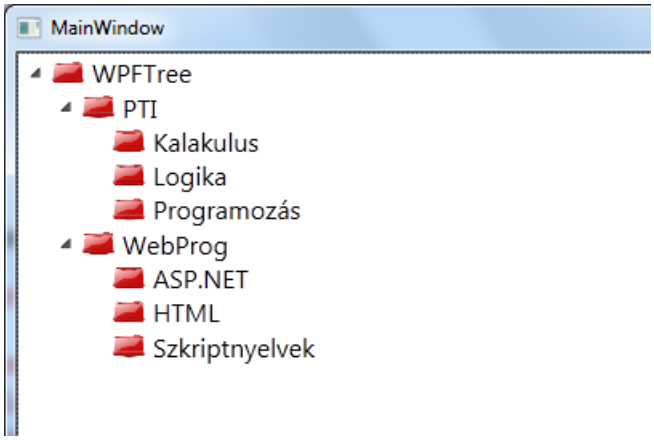
```
    private void addButton_Click(object sender, RoutedEventArgs e)
     {
        list1.Items.Add(tb1.Text);
        tb1.Text = "";
     }
    private void deleteButton_Click(object sender, RoutedEventArgs e)
     {
        list1.Items.RemoveAt
        (list1.Items.IndexOf(list1.SelectedItem));
}
```

3.3. TreeWiew

The TreeView control elements can be arranged hierarchically. It is ItemsControl too, nodes may contain not the text only. Type of the elements: TreeViewItem from the HeaderedItemsControl class. Each item has a Header feature, could be set the label of each elements with the help of it.

Example V.13 TreeView

```
<TreeView>
     <TreeViewItem Header="Female names:">
        <TreeViewItem Header="Éva"/>
        <TreeViewItem Header="Krisztina"/>
        <TreeViewItem Header="Mária"/>
     </TreeViewItem>
     <TreeViewItem Header="Male names">
        <TreeViewItem Header="Gábor"/>
        <TreeViewItem Header="Áron"/>
        <TreeViewItem Header="Csaba"/>
    </TreeViewItem>
</TreeView>
```



V.10. TreeView

```
<TreeView Name="tv" ItemsSource="{Binding}">
        <TreeView.ItemTemplate>
            <HierarchicalDataTemplate ItemsSource="{Binding Path=Folders}">
                <StackPanel Orientation="Horizontal">
                    <Image Source="Icons/advanced_directory.ico" Width="20"
                     Height="20" Margin="0 0 5 0"/>
                    <TextBlock Text="{Binding Path=Name}" FontSize="16" />
                </StackPanel>
            </HierarchicalDataTemplate>
        </TreeView.ItemTemplate>
</TreeView>
```

Code-behind:

```
public class Folder
{
   public string Name
   {
      get
      {
         if (!String.IsNullOrEmpty(Path))
         {
            return System.IO.Path.GetFileName(Path);
         }
         return null;
```

```
        }
    }
    public string Path
    { get; set; }
    public List<Folder> Folders
    { get; set; }
    public Folder()
    {
        Folders = new List<Folder>();
    }
    public static Folder CreateFolderTree(string rootFolder)
    {
        Folder fld = new Folder { Path = rootFolder };
        foreach (var item in Directory.GetDirectories(rootFolder))
        {
            fld.Folders.Add(CreateFolderTree(item));
        }
        return fld;
    }
}
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        List<Folder> folders = new List<Folder>();
        folders.Add(Folder.CreateFolderTree(@"C:\WPFTree"));
        tv.DataContext = folders;
    }
}
```

3.4. Menu

In WPF applications you can create menus easily. This is very important, as the Menu control is necessary in most of the applications. The menu allows the hierarchical arrangement of the most commonly used commands.

Key features:

Command – you can set commands for the individual menu items

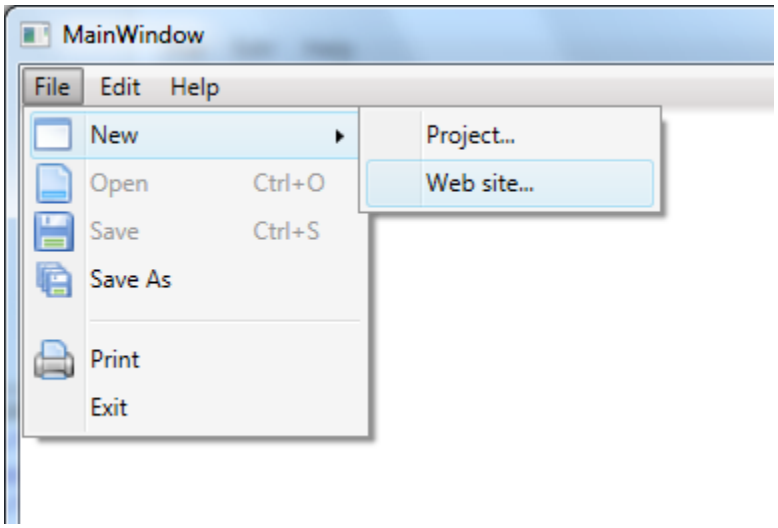
Header – allows you to adjust the text on the menu items

Icon – it can be assigned to individual menu items

IsChecked - set or queried whether each item is checked

IsEnabled – set or queried whether each option is enabled

Example V.14 Menu



V.11. Menu

```
<Menu Height="25" VerticalAlignment="Top">
    <MenuItem Header="_File">
      <MenuItem Header="_New" InputGestureText="Ctrl+N">
        <MenuItem.ToolTip>
          <ToolTip>
            New document
          </ToolTip>
        </MenuItem.ToolTip>
        <MenuItem.Icon>
          <Image Width="20" Height="20" Source="Icons/application.png" />
        </MenuItem.Icon>
        <MenuItem Header="_Project..."/>
        <MenuItem Header="_Web Site..."/>
        </MenuItem>
        <MenuItem Header="_Open"
```

```
            Command="ApplicationCommands.Open">
          <MenuItem.Icon>
            <Image Width="20" Height="20" Source="Icons/page.png" />
          </MenuItem.Icon>
        </MenuItem>
        <MenuItem Header="_Save"
              Command="ApplicationCommands.Save">
          <MenuItem.Icon>
            <Image Width="20" Height="20" Source="Icons/disk.png" />
          </MenuItem.Icon>
        </MenuItem>
        <MenuItem Header="Save _As">
          <MenuItem.Icon>
            <Image Width="20" Height="20" Source="Icons/disk_multiple.png" />
          </MenuItem.Icon>
        </MenuItem>
        <Separator/>
        <MenuItem Header="_Print">
          <MenuItem.Icon>
            <Image Width="20" Height="20" Source="Icons/printer.png" />
          </MenuItem.Icon>
        </MenuItem>
        <MenuItem Header="Exit" Click="exit_Click"/>
      </MenuItem>
      <MenuItem Header="_Edit">
        <MenuItem Header="_Undo"/>
        <MenuItem Header="_Cut"/>
        <MenuItem Header="_Copy"
              Command="ApplicationCommands.Copy"/>
        <MenuItem Header="_Paste"/>
      </MenuItem>
    <MenuItem Header="Help">
```

```
        <MenuItem Header="Program"  Click="program_Click"/>
    </MenuItem>
</Menu>
```

We can assign Click events for the individual members of the Menu class, like in the Button one.

Code-behind:

```
private void exit_Click(object sender, RoutedEventArgs e)
{
    Application.Current.Shutdown();
}
private void program_Click(object sender, RoutedEventArgs e)
{
     MessageBox.Show(".Net Programming Technologies");
}
```

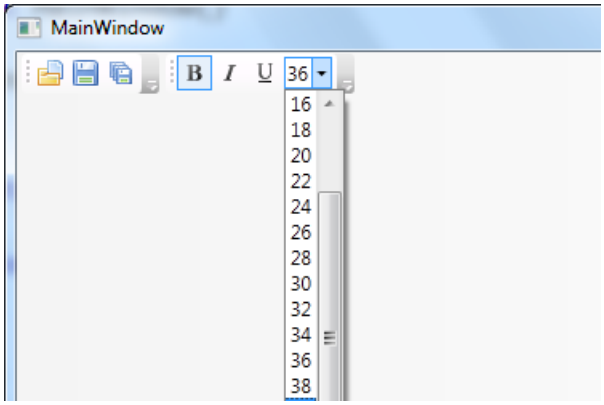
3.5. ToolBar

With the help of toolbars we can take the interface of our application userfriendly.

Example V.15 ToolBar 1

```
<ToolBarTray>
        <ToolBar>
          <Button ToolTip="Open" Command="ApplicationCommands.Open">
            <Image Source="Icons/folder_page.png" />
          </Button>
          <Button ToolTip="Save" Command="ApplicationCommands.Save">
            <Image Source="Icons/disk.png" />
          </Button>
          <Button ToolTip="Save As" Command="ApplicationCommands.SaveAs">
            <Image Source="Icons/disk_multiple.png" />
          </Button>
        </ToolBar>
</ToolBarTray>
```

Example V.16 ToolBar 2



V.12. ToolBar

```
<ToolBar>
    <ToggleButton x:Name="bold" ToolTip="Bold"
                  Command="EditingCommands.ToggleBold">
                  <Image Source="Icons/text_bold.png" />
    </ToggleButton>
    <ToggleButton x:Name="italic" ToolTip="Italic"
                  Command="EditingCommands.ToggleItalic">
                 <Image Source="Icons/text_italic.png" />
    </ToggleButton>
    <ToggleButton ToolTip="Underline" x:Name="underline"
                  Command="EditingCommands.ToggleUnderline">
                <Image Source="Icons/text_underline.png" />
    </ToggleButton>
    <ComboBox x:Name="sizeT" Width="30" />
</ToolBar>
```

Code-behind:

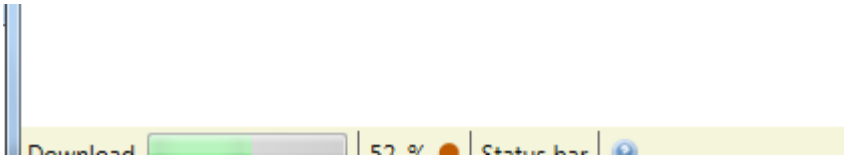
```
    private void Window_Loaded(object sender, RoutedEventArgs e)
    {
      for (double i = 8; i < 90; i += 2)
       {
         sizeT.Items.Add(i);
       }
    }
    private void SetFontSize(double size)
```

```
{
  sizeT.SelectedValue = size;
}
private void SetFontWeight(FontWeight weight
{
  bold.IsChecked = weight == FontWeights.Bold;
}
private void SetFontStyle(FontStyle style)
{
  italic.IsChecked = style == FontStyles.Italic;
}
private void SetTextDecoration(TextDecorationCollection decoration)
{
  underline.IsChecked = decoration == TextDecorations.Underline;
}
```

3.6. StatusBar

StatusBar is very similar to the previously known toolbar. Variety of information can be written on the bottom of the application window for users. For example, in a graphics editor we can write the mouse coordinates, the coordinates of the selected part, size, line width, the current font, etc.

Example V.17 StatusBar



V.13. StatusBar

```
<StatusBar VerticalAlignment="Bottom" Background="Beige" >
        <StatusBarItem>
          <TextBlock>Letöltés</TextBlock>
        </StatusBarItem>
        <StatusBarItem>
          <ProgressBar Width="100" Height="20" Name="pb"  >
            <ProgressBar.Triggers>
              <EventTrigger RoutedEvent="Window.Loaded">
                <BeginStoryboard>
```

```
                <Storyboard>
                  <DoubleAnimation
                    Storyboard.TargetName="pb"
                    Storyboard.TargetProperty="Value"
                    From="0" To="100" Duration="0:0:5" />
                  <ColorAnimation
                    Storyboard.TargetName="dlE"
                    Storyboard.TargetProperty="Fill.Color"
                    From="Green" To="Red"
                    Duration="0:0:5" />
                </Storyboard>
              </BeginStoryboard>
            </EventTrigger>
          </ProgressBar.Triggers>
        </ProgressBar>
      </StatusBarItem>
      <Separator/>
      <StatusBarItem>
        <TextBlock Text="{Binding ElementName=pb, Path=Value, StringFormat=0}"/>
      </StatusBarItem>
      <StatusBarItem>
        <TextBlock Text="%"/>
      </StatusBarItem>
      <StatusBarItem>
        <Ellipse x:Name="dlE" Fill="Green" Height="10" Width="10"/>
      </StatusBarItem>
      <Separator/>
      <StatusBarItem>
        <TextBlock x:Name="tb">Állapotsor</TextBlock>
      </StatusBarItem>
      <Separator/>
      <StatusBarItem >
```

```
            <Image Source="Images\help.png" Width="16" Height="16"
                ToolTip="Segítség"/>
        </StatusBarItem>
        <StatusBarItem HorizontalAlignment="Right">
            <TextBlock x:Name="idoTb" />
        </StatusBarItem>
</StatusBar>
```

Code-behind:

```
namespace StatusBar
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        DispatcherTimer ido;
        StringBuilder stb;
        void Initialize()
        {
            ido = new DispatcherTimer();
            ido.Interval = TimeSpan.FromSeconds(1.0);
            ido.Start();
            stb = new StringBuilder();
            ido.Tick += new EventHandler(delegate(object s, EventArgs a)
            {
                stb.Clear();
                if (DateTime.Now.Hour < 10) stb.Append(0);
                stb.Append(DateTime.Now.Hour);
                stb.Append(':');
                if (DateTime.Now.Minute < 10) stb.Append(0);
                stb.Append(DateTime.Now.Minute);
                stb.Append(':');
```

```
            if (DateTime.Now.Second < 10) stb.Append(0);
            stb.Append(DateTime.Now.Second);
            idoTb.Text = stb.ToString();
        });
    }
    public MainWindow()
    {
        Initialize();
        InitializeComponent();
    }
  }
}
```

6. fejezet - Colours and Brushes (written by Biró Csaba)

1. Color management

First let's see something about the color management. If you want to create or use your own colors in the applications, you can define them in the following ways.

<Color x:Key="narancssargaSzin" A="255" R="255" G="176" B="59" ></Color>

<Color x:Key="sotetpirosSzin">#FFAA2C27</Color>

In the first case: ARGB - means A(Alpha), R(red), G(Green), and B (Blue) values. The first (alpha) parameter defines the degree of opacity. It means 100 % in our case (value of 255). The other, hexadecimal way beginning with # well known from the web world.

The first two digits: Opacity value (255=100%)

the second two digits: quantity of red colour

the third two digits: quantity of green

the fourth two digits: the amount of blue

2. Brushes

Colors can be produced by brushes in WPF.

2.1. SolidColorBrush

The simpliest class of Brush class for painting a given area with only a specified colour by the Color attribute.

Example VI.1 SolidColorBrush

VI.1. SolidColorBrush

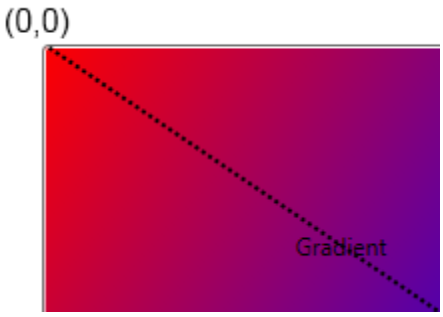
```
<Grid>
    <Grid.Resources>
      <Color x:Key="ornageColor" A="255" R="255" G="176" B="59" ></Color>
      <Color x:Key="redColor">#FFAA2C27</Color>
      <SolidColorBrush x:Key="ornageFill"
            Color="{StaticResource ornageColor}"></SolidColorBrush>
      <SolidColorBrush x:Key="redFill"
            Color="{StaticResource redColor}"></SolidColorBrush>
    </Grid.Resources>
```

```
    <Button Width="150" Height="50" Content="Színek kezelése" FontSize="18"
        Background="{StaticResource ResourceKey=ornageFill}"
        Foreground="{StaticResource ResourceKey=redFill}">
    </Button>
</Grid>
```

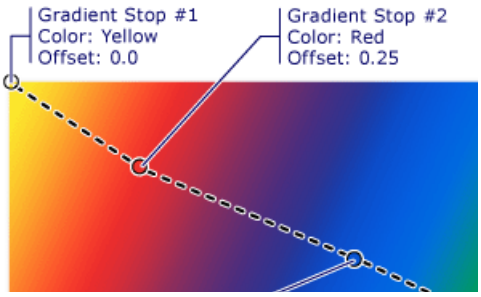
2.2. LinearGradientBrush

LinearGadientBrush permits gradient fills.

```
<LinearGradientBrush
    StartPoint="0.5,0"
    EndPoint="0.5,1">
  <GradientStop Color="color1" Offset="0.4"/>
      (0 – 1 )
  <GradientStop Color=”color2” Offset=”1”/>
</LinearGradientBrush>
```



VI.2. Gradient



VI.3. GradientStop

Example VI.2 LinearGradientBrush

VI.4. LinearGradientBrush

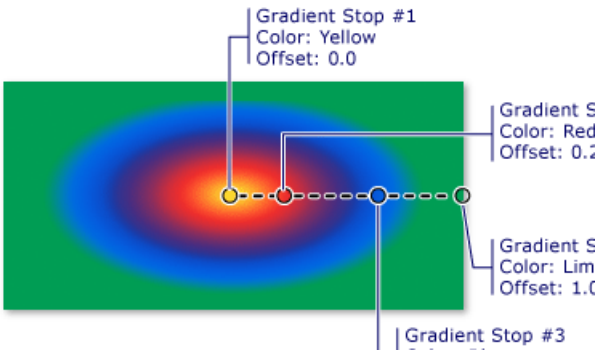
```
<LinearGradientBrush x:Key="orangetoyellowFill"
                     StartPoint="0.5,0"
                     EndPoint="0.5,1">
    <GradientStop Color="{StaticResource ResourceKey=orangeColor}"
                  Offset="0.4"/>
    <GradientStop Color="Red"
                  Offset="1"/>
</LinearGradientBrush>
<LinearGradientBrush x:Key="yellowtoblueFill"
                     StartPoint="0,0.5"
                     EndPoint="1,0.5">
    <GradientStop Color="Yellow" Offset="1"/>
    <GradientStop Color="Blue" Offset="0.4"/>
</LinearGradientBrush>
```

2.3. RadialGradientBrush

RadialGradientBrush makes the radial gradient fill possible.

```
<RadialGradientBrush GradientOrigin=”0.45, 0.30”>
    <GradientStop Color=””
    Offset=”0”/>
    <GradientStop Color=””
    Offset=”1”/>
</RadialGradientBrush>
```

The transition is a stretching line from the center of the transition (GradientOrigin) to the circumference of the ellipse.



VI.5. RadialGradientBrush

Example VI.3 RadialGradienBrush

VI.6. RadialGradientBrush

```
<Grid.Resources>
    <RadialGradientBrush x:Key="yellowtoorangeRadialFill"  GradientOrigin="0.75,0.25">
            <GradientStop Color="Yellow" Offset="0.0" />
            <GradientStop Color="Orange" Offset="0.2" />
            <GradientStop Color="Red" Offset="1.0" />
    </RadialGradientBrush>
</Grid.Resources>
<Button Width="150" Height="50" Content="Színek kezelése" FontSize="18"
        Background="{StaticResource ResourceKey= yellowtoorangeRadialFill}"
        Foreground="{StaticResource ResourceKey=yellowtoblueFill}"/>
```

2.4. ImageBrush

We can draw on the ImageSource with the help of ImageBrush.

VI.4 Példa ImageBrush

VI.7. ImageBrush

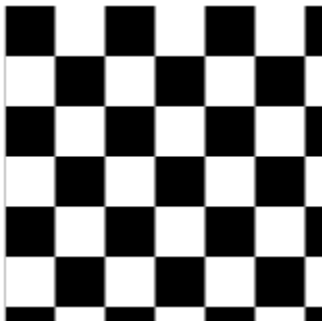
```
<Rectangle Width="75" Height="75">
        <Rectangle.Fill>
           <ImageBrush ImageSource="images/apple.jpg"/>
        </Rectangle.Fill>
</Rectangle>
```

2.5. DrawingBrush

The previously known brushes have plenty of options for formatting elements. When more complex operations needs, we can do it by the BrawingBrush drawing brush suitable for coloring images' backgrounds working with geometric data instead of DrawingBrush shapes.

First, here an example for preparing chessboard (Rectangle object).

VI.5 Példa Chess board



VI.8. Chess board

```
<Rectangle Width="200" Height="200">
        <Rectangle.Fill>
          <DrawingBrush Viewport="0,0,0.25,0.25" TileMode="Tile">
            <DrawingBrush.Drawing>
              <DrawingGroup>
                <GeometryDrawing Brush="White">
                  <GeometryDrawing.Geometry>
                    <RectangleGeometry Rect="0,0,100,100" />
                  </GeometryDrawing.Geometry>
                </GeometryDrawing>
                <GeometryDrawing Brush="Black">
                  <GeometryDrawing.Geometry>
                    <GeometryGroup>
                      <RectangleGeometry Rect="0,0,50,50" />
                      <RectangleGeometry Rect="50,50,50,50"/>
                    </GeometryGroup>
                  </GeometryDrawing.Geometry>
                </GeometryDrawing>
              </DrawingGroup>
            </DrawingBrush.Drawing>
          </DrawingBrush>
        </Rectangle.Fill>
</Rectangle>
```

2.6. VisualBrush

The VisualBrush has various functions, which can paint on any surfaces as a descendant of the Visual class.

VI.7 Példa BMW emblem

VI.9. BMW emblem

```
<Rectangle Width="75" Height="75">
    <Rectangle.Fill>
      <VisualBrush TileMode="Tile">
        <VisualBrush.Visual>
          <StackPanel>
            <StackPanel.Background>
              <DrawingBrush>
                <DrawingBrush.Drawing>
                  <GeometryDrawing>
                    <GeometryDrawing.Brush>
                      <RadialGradientBrush>
                  <GradientStop Color="MediumBlue" Offset="0.0" />
                  <GradientStop Color="White" Offset="1.0" />
                      </RadialGradientBrush>
                    </GeometryDrawing.Brush>
                    <GeometryDrawing.Geometry>
                      <GeometryGroup>
                        <RectangleGeometry Rect="0,0,50,50" />
                        <RectangleGeometry Rect="50,50,50,50" />
                      </GeometryGroup>
                    </GeometryDrawing.Geometry>
                  </GeometryDrawing>
                </DrawingBrush.Drawing>
              </DrawingBrush>
            </StackPanel.Background>
            <TextBlock FontSize="10pt" Margin="10">BMW</TextBlock>
```

```
            </StackPanel>
        </VisualBrush.Visual>
    </VisualBrush>
  </Rectangle.Fill>
</Rectangle>
```

7. fejezet - Shapes (written by Csaba Biró)

1. Built-in shapes

WPF also provides the familiar basic shapes are found in the System.Windows.Shapes class.

Built-in shapes are as follows:

1. Line,
2. Polyline ,
3. Polygon,
4. Rectangle,
5. Ellipse,
6. Path.

1.1. Line

A Line draw a line between two points.

Key features

X1 Y1 initial coordinates,

X2 Y2 ending coordinates,

Stroke line color,

StrokeThickness stroke thickness,

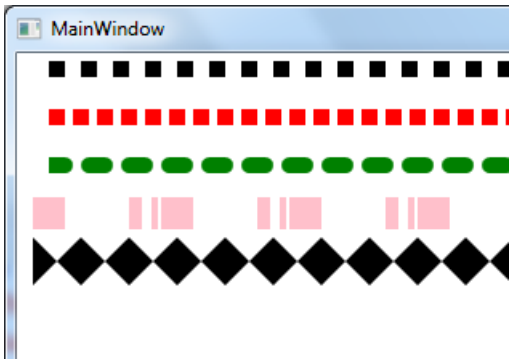
StrokeStartLineCap line endings,

StrokeEndLineCap line endings,

StrokeDashCap line endings,

StrokeDashArray sections and length of gaps.

Example VII.1 Line



VII.1. Line

```
<Line X1="20" Y1="10" X2="480" Y2="10"
      StrokeThickness="10"
      Stroke="Black"
      StrokeDashArray="1 1">
</Line>
<Line X1="20" Y1="40" X2="480" Y2="40"
      StrokeThickness="10"
      Stroke="Red"
      StrokeDashArray="1 0.5">
</Line>
<Line X1="20" Y1="70" X2="480" Y2="70"
      StrokeThickness="10"
      Stroke="Green"
      StrokeDashArray="1 1.5"
      StrokeDashCap="Round"
      StrokeStartLineCap="Flat"
      StrokeEndLineCap="Triangle">
</Line>
<Line X1="10" Y1="100" X2="490" Y2="100"
      Stroke="Pink"
      StrokeThickness="20"
      StrokeDashArray="1 2 0.4 0.3 0.2 0.1">
</Line>
<Line X1="10" Y1="130" X2="490" Y2="130" Stroke="Black"
      StrokeThickness="30"
      StrokeDashArray="0 1"
      StrokeDashCap="Triangle">
</Line>
```

1.2. Polyline

Polyline is suited to draw lines consist of several sections. Most features of line shapes are already known.

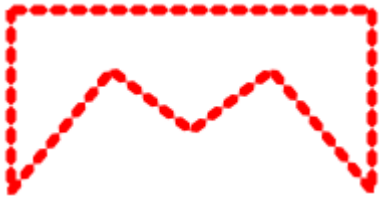
Key features:

Points X and Y coordinates,

StrokeLineJoin rounding of the peaks,

StrokeMiterLimit thinning of the peaks.

Example VII.2 PolyLine



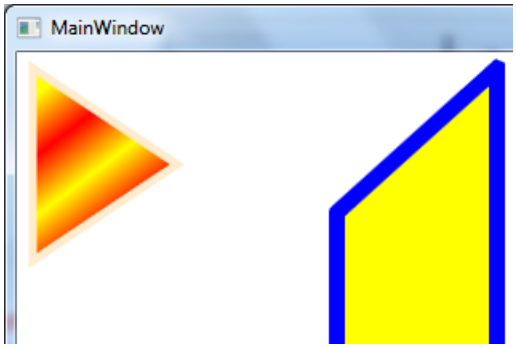
VII.2. PolyLine

```
<Polyline
    Points="10 10 10 100 60 40 100 70 140 40 190 100 190 10 10 10"
    Stroke="Red"
    StrokeThickness="5"
    StrokeDashArray="1 1"
    StrokeDashCap="Triangle">
</Polyline>
```

1.3. Polygon

Polygon is for drawing polygons. It has a new feature (Fill) has been described in previous figures.

Example VII.3 Polygon



VII.3. Polygon

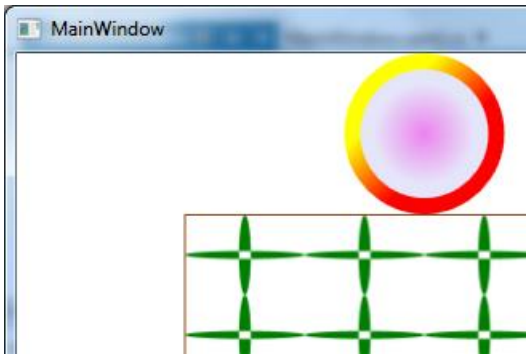
```
<Polygon Points="10 10 100 70 10 130"
        Stroke="BlanchedAlmond"
        StrokeThickness="5">
      <Polygon.Fill>
        <LinearGradientBrush>
          <GradientStop Color="Yellow" Offset="0.1"/>
```

```
            <GradientStop Color="Red" Offset="0.3"/>
            <GradientStop Color="Yellow" Offset="0.5"/>
            <GradientStop Color="Red" Offset="0.7"/>
            <GradientStop Color="Yellow" Offset="0.9"/>
        </LinearGradientBrush>
    </Polygon.Fill>
</Polygon>
<Polygon Points="200 100 200 200 300 200 300 10"
        Stroke="Blue" StrokeThickness="10"
        StrokeLineJoin="Miter"
        StrokeMiterLimit="1"
        Fill="Yellow"/>
<Polygon Points="400 10 380 60 340 75 380 100 400 140 420 100 460 75 420 60 ">
    <Polygon.Fill>
        <ImageBrush ImageSource="/Pictures/Nap.gif"/>
    </Polygon.Fill>
</Polygon>
```

1.4. Ellipse és Rectangle

They have a drawing ability (rectangle and ellipse) completed with Width and Height features.

Example VII.4 Ellipse and Rectangle



VII.4. Ellipse and Rectangle

```
<StackPanel>
    <Ellipse Width="100" Height="100" StrokeThickness="10">
        <Ellipse.Stroke>
            <LinearGradientBrush>
                <GradientStop Offset="0.3" Color="Yellow"/>
```

```
            <GradientStop Offset="0.6" Color="Red"/>
        </LinearGradientBrush>
    </Ellipse.Stroke>
        <Ellipse.Fill>
            <RadialGradientBrush>
                <GradientStop Offset="0" Color="Violet"/>
                <GradientStop Offset="0.7" Color="Lavender"/>
            </RadialGradientBrush>
        </Ellipse.Fill>
    </Ellipse>
    <Rectangle Width="300" Height="200"
               Stroke="SaddleBrown">
        <Rectangle.Fill>
        <DrawingBrush Viewport="1,1,0.25,0.25" TileMode="Tile">
            <DrawingBrush.Drawing>
                <DrawingGroup>
                    <GeometryDrawing Brush="White">
                    </GeometryDrawing>
                    <GeometryDrawing Brush="Green">
                        <GeometryDrawing.Geometry>
                            <GeometryGroup>
                            <EllipseGeometry RadiusX="10" RadiusY="10"/>
                            <EllipseGeometry RadiusX="100" RadiusY="1"/>
                            </GeometryGroup>
                        </GeometryDrawing.Geometry>
                    </GeometryDrawing>
                </DrawingGroup>
            </DrawingBrush.Drawing>
        </DrawingBrush>
        </Rectangle.Fill>
    </Rectangle>
</StackPanel>
```

2. Geometry class

The Geometry class contains various geometric shapes. Not directly, but can be accessed through its descendants.

Geometries:

1. LineGeometry

<LineGeometry StartPoint="20,50" EndPoint="200,50" />

1. RectangleGeometry

<RectangleGeometry Rect="80,167 150 30"/>

1. EllipseGeometry

<EllipseGeometry Center="80,150" RadiusX="50" RadiusY="50" />

1. GroupGeometry

2. PathGeometry

3. CombinedGeometry

4. StreamGeometry

The first three have been mentioned before, it is clear on the basis of foregoing studies.

2.1. Path

WPF has a class called Path to product composite drawings and curves. Curves construction cannot be compared to the former one, actually it can be created by a series of commands. The producing of these commands is not an easy way by hand, so it can be generated with the Microsoft software called Expression Design. (Chapter XVIII.3)

2.2. PathGeometry

The PathGeometry class is considered one of the strongest classes in geometry. You can build up one or more PathFigure object within a PathGeometry object. A PathFigure (curve element) is actually a set of continuous lines and curves can be closed (IsClosed – completion of items) or open depending on that the final and the starting point of the shape are connected with each other or not.

PathGeometry syntax:

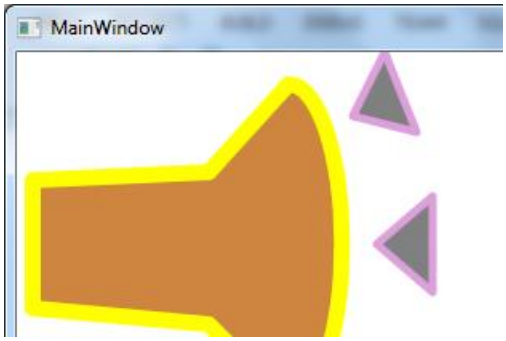
```
<Path>
  <Path.Data>
    <PathGeometry>
      <PathFigure>
        <ArcSegment />
        <BezierSegment />
        <LineSegment />
        <PolyBezierSegment />
        <PolyLineSegment />
```

```
            <PolyQuadraticBezierSegment />
            <QuadraticBezierSegment />
          </PathFigure>
        </PathGeometry>
      </Path.Data>
    </Path>
```

Line segments:

1. ArcSegment,
2. BezierSegment,
3. LineSegment,
4. PolyBezierSegment,
5. PolyLineSegment,
6. PolyQuadraticBezierSegment,
7. QuadraticBezierSegment.

Example VII.5 Speaker



VII.5. Speaker

```
    <Path Fill="Peru"
        Stroke="Yellow" StrokeThickness="10"
        StrokeLineJoin="Round">
      <Path.Data>
        <PathGeometry>
          <PathFigure StartPoint="120,75"
                IsClosed="True">
            <LineSegment
              Point="10,80" />
            <PolyLineSegment
```

```
                Points="10,160 120,170 170,220" />
            <ArcSegment
                Point="170,20"
                Size="10,30" />
        </PathFigure>
      </PathGeometry>
    </Path.Data>
  </Path>
  <Path Fill="Gray"
      Stroke="Plum" StrokeThickness="5"
      StrokeLineJoin="Round">
        <Path.Data>
            <PathGeometry>
        <PathFigure StartPoint="210,200" IsClosed="True">
            <LineSegment Point="230,240"/>
            <LineSegment Point="250,200"/>
        </PathFigure>
        <PathFigure StartPoint="225,120" IsClosed="True">
            <LineSegment Point="260,150"/>
            <LineSegment Point="260,90"/>
        </PathFigure>
        <PathFigure StartPoint="210,40" IsClosed="True">
            <LineSegment Point="230,0"/>
            <LineSegment Point="250,50"/>
        </PathFigure>
      </PathGeometry>
    </Path.Data>
  </Path>
```

VII.6 Példa M letter

VII.6. M letter

```
<Path StrokeThickness="10">
  <Path.Stroke>
    <LinearGradientBrush>
      <GradientStop Offset="0.3" Color="Yellow"/>
      <GradientStop Offset="1" Color="Red"/>
      </LinearGradientBrush>
  </Path.Stroke>
  <Path.Data>
     <PathGeometry>
       <PathGeometry.Figures>
         <PathFigureCollection>
           <PathFigure StartPoint="10,280">
             <PathFigure.Segments>
               <PathSegmentCollection>
                <QuadraticBezierSegment Point1="120,380" Point2="80,100" />
                <QuadraticBezierSegment Point1="170,190" Point2="240,100"/>
                <QuadraticBezierSegment Point1="200,380" Point2="320,280"/>
               </PathSegmentCollection>
             </PathFigure.Segments>
           </PathFigure>
         </PathFigureCollection>
       </PathGeometry.Figures>
     </PathGeometry>
  </Path.Data>
```

</Path>

2.3. GeometryGroup

We can create objects consist of a number of geometric shapes.

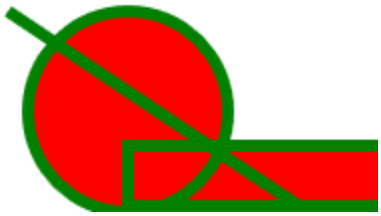
Example VII.7 Complex shapes 1

VII.7. Complex shapes 1

```
<Path Stroke="Green" StrokeThickness="6" Fill="Red">
    <Path.Data>
        <GeometryGroup>
            <LineGeometry StartPoint="20,100" EndPoint="200,220" />
            <EllipseGeometry Center="80,150" RadiusX="50" RadiusY="50" />
            <RectangleGeometry Rect="80,167 150 30"/>
        </GeometryGroup>
    </Path.Data>
</Path>
```

The FillRule feature of the GeometryGroup is attended to combine the parts of the shapes have been cut from each other, we can set it with Norzero value, defaults to EvenOdd.

VII.8 Példa Complex shapes2



VII.8. Complex shapes 2

```
<GeometryGroup FillRule="Nonzero">
```

2.4. StreamGeometry

Our shapes can be set in much more compact form in contrast with the above mentioned ones. The StreamGeometry is basically a mini-language.

Elements of Language:

Move

M or m initial coordinates (X,Y). e.g. M 0,0

Draw line

L or l ending coordinates (X,Y) e.g. L 10,10

Horizontlal line

H or h X coordinate e.g. H 90.

Vertical line

V or v Y coordinate. e.g. V 90

Cubic Bezier Curve

C or c controlpoint1, controlpoint 2, endpoint e.g. C 10,10 20,20 30,10

Quadratic Bezier Curve

Q or q controlpoint, endpoint. e.g. Q 100,100 130,240

Smooth cubic Bezier curve

S or s controlpoint, endpoint. e.g. s 100,100 130,240

Smooth quadratic Bezier curve

T or t controlpoint, endpoint. e.g. T 100,100 130,240

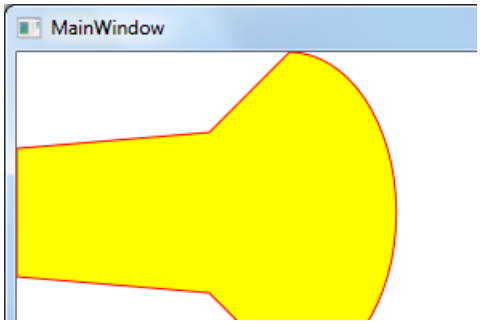
Arc

A or a size, rotation angle, heavy drinkers, curve direction, endpoint.

Closed shape

Z or z the current point to the starting point draw a line.

Example VII.8 Speaker- StreamGeometry



VII.9. Speaker - StreamGeometry

```
Path Fill="Yellow"
      Stroke="Red"
      Data="M 120,50 L 0,60 0,140 120,150 170,200 A 20,30,0,0, 0, 170, 0 Z"/>
```

Example VII.9 StreamGeometry

VII.10. StreamGeometry

<Path Stroke="Black" Fill="Gray" Data="M 100,100 C 110,300 400,-100 400,100" />

2.5. CombinedGeometry

CombinedGeometry is used for the combination of two geometric object.

<CombinedGeometry>

<CombinedGeometry.Geometry1>

Elem

</CombinedGeometry.Geometry1>

<CombinedGeometry.Geometry2>

Elem

</CombinedGeometry.Geometry2>

</CombinedGeometry>

GeometryCombineMode properties:

1. Union
2. Intersect
3. Exclude
4. Xor

Example VII.10 Union

VII.11. Union

<Path Fill="Gainsboro">

<Path.Data>

<CombinedGeometry GeometryCombineMode="Union">

<CombinedGeometry.Geometry1>

```
            <EllipseGeometry RadiusX="50" RadiusY="50" Center="75,75" />
          </CombinedGeometry.Geometry1>
          <CombinedGeometry.Geometry2>
            <EllipseGeometry RadiusX="50" RadiusY="50" Center="125,75" />
          </CombinedGeometry.Geometry2>
        </CombinedGeometry>
      </Path.Data>
</Path>
```

Example VII.11 Intersect

VII.12. Intersect

```
<Path Fill="Gainsboro">
        <Path.Data>
          <CombinedGeometry GeometryCombineMode="Intersect">
            <CombinedGeometry.Geometry1>
              <EllipseGeometry RadiusX="50" RadiusY="50" Center="75,75" />
            </CombinedGeometry.Geometry1>
            <CombinedGeometry.Geometry2>
              <EllipseGeometry RadiusX="50" RadiusY="50" Center="125,75" />
            </CombinedGeometry.Geometry2>
          </CombinedGeometry>
        </Path.Data>
</Path>
```

Example VII.12 Exclude

VII.13. Exclude

```
<Path Fill="Gainsboro">
    <Path.Data>
        <CombinedGeometry GeometryCombineMode="Exclude">
          <CombinedGeometry.Geometry1>
            <EllipseGeometry RadiusX="50" RadiusY="50" Center="75,75" />
          </CombinedGeometry.Geometry1>
          <CombinedGeometry.Geometry2>
            <EllipseGeometry RadiusX="50" RadiusY="50" Center="125,75" />
          </CombinedGeometry.Geometry2>
        </CombinedGeometry>
    </Path.Data>
</Path>
```

Example VII.13 Xor

VII.14. Xor

```
<Path Fill="Gainsboro">
    <Path.Data>
        <CombinedGeometry GeometryCombineMode="Xor">
          <CombinedGeometry.Geometry1>
            <EllipseGeometry RadiusX="50" RadiusY="50" Center="75,75" />
          </CombinedGeometry.Geometry1>
          <CombinedGeometry.Geometry2>
            <EllipseGeometry RadiusX="50" RadiusY="50" Center="125,75" />
          </CombinedGeometry.Geometry2>
        </CombinedGeometry>
    </Path.Data>
</Path>
```

8. fejezet - Transformations (written by Csaba Biró)

We will discuss about the transformation in this chapter. All Ulelement objects are transformable through the features of RenderTransform and LayoutTransform. With the help of transformations individual elements'mapping can be modified.

Transform class inhertited:

1. TranslateTransform,
2. SkaleTransform,
3. RotateTransform,
4. SkewTransform,
5. MatrixTransform,
6. TransformGroup.

Each transformation class derived from the System.Windows.Media.Transform one.

1. TranslateTransform

TranslateTransform can help you to change the location of an item.

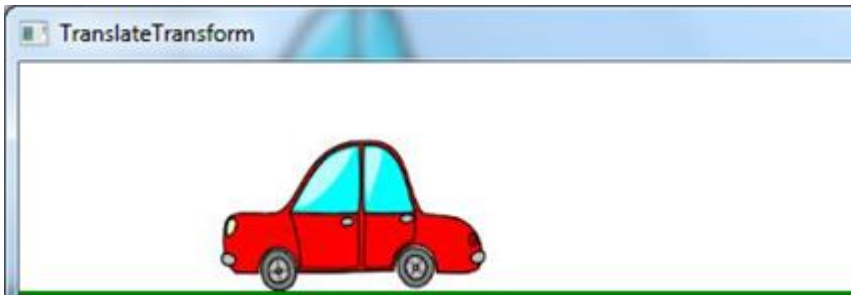
Properties:

1. X the extent of the x-axis offset of the item,
2. Y the scale on the y-axis offset of the item.

1 0 0 0 1 0 dx dy 1 left [matrix {1 # 0 # 0 ## 0 # 1 # 0 ## dx # dy # 1} right]

3 x 3 matrix

Example VIII.1 TranslateTransform



VIII.1. TranslateTransform

```
<Window x:Class="Transzformaciok.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="TranslateTransform" Height="205" Width="600">
```

```
<Grid>
  <StackPanel>
    <Rectangle Height="120" Width="150" HorizontalAlignment="Left">
      <Rectangle.Fill>
        <ImageBrush ImageSource="images/car.jpg"/>
      </Rectangle.Fill>
      <Rectangle.RenderTransform>
        <TranslateTransform X="{Binding ElementName=slider1, Path=Value}"/>
      </Rectangle.RenderTransform>
    </Rectangle>
    <Line X1="0" X2="600" Stroke="Green" StrokeThickness="40"/>
    <Slider x:Name="slider1" Minimum="0" Maximum="600" Background="Green"/>
  </StackPanel>
</Grid>
</Window>
```

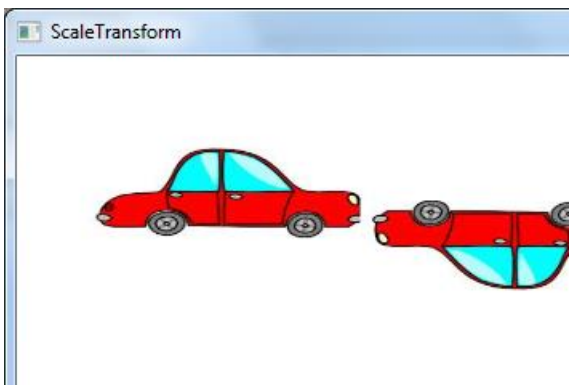
2. SkaleTransform

You can change the size of an item with the help of the SkaleTransform.

Properties:

1. ScaleX elongation in X direction,
2. ScaleY elongation in Y direction,
3. CenterX center X coordinate,
4. CenterY center Y coordinate,
5. Default value is (0,0).

Example VIII.2 ScaleTransform



VIII.2. ScaleTransform

```
<StackPanel>
  <Image Height="60" Width="60" Source="Images/car.jpg" Margin="20">
    <Image.RenderTransform>
     <ScaleTransform CenterX="0" CenterY="0" ScaleX="-3" ScaleY="2"/>
      </Image.RenderTransform>
    </Image>
  <Image Height="60" Width="60" Source="Images/car.jpg" Margin="20">
    <Image.RenderTransform>
     <ScaleTransform CenterX="5" CenterY="15" ScaleX="3" ScaleY="-2"/>
    </Image.RenderTransform>
  </Image>
</StackPanel>
```

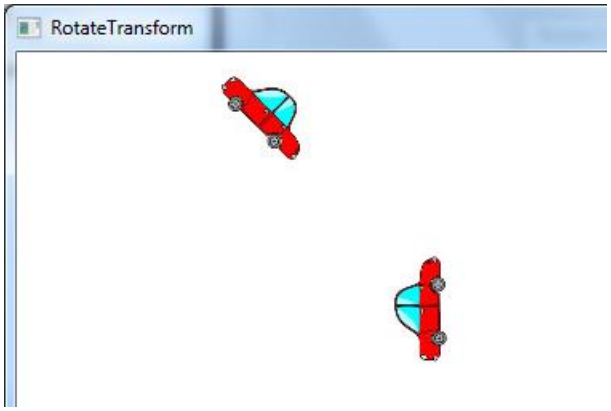
3. RotateTransform

You can rotate the item in a specified angle.

Properties:

1. Angle angle of rotation,
2. CenterX center of rotation (x axis)
3. CenterY center of rotation (y axis)

Example VIII.3 RotateTransform



VIII.3. RotateTransform

```
<StackPanel>
      <Image Source="Images/car.jpg" Height="60" Width="70" Margin="30">
         <Image.RenderTransform>
            <RotateTransform Angle="45" CenterX="25" CenterY="-100"/>
         </Image.RenderTransform>
```

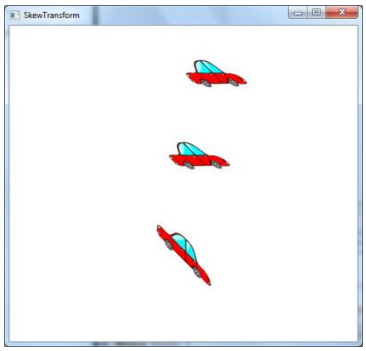
```
    </Image>
      <Image Source="Images/car.jpg" Height="60" Width="70" Margin="70">
        <Image.RenderTransform>
          <RotateTransform Angle="-90"/>
        </Image.RenderTransform>
      </Image>
</StackPanel>
```

4. SkewTransform

The SkewTransform can strain the original element.

1. AngleX tilt the item on the x axis
2. AngleY tilt the item on the y axis

Example VIII.4 SkewTransform



VIII.4. SkewTransform

```
<StackPanel>
   <Image Width="90" Height="60" Source="images/car.jpg" Margin="30">
     <Image.RenderTransform>
      <SkewTransform AngleX="45" AngleY="0" CenterX="0" CenterY="0"/>
     </Image.RenderTransform>
   </Image>
   <Image Width="90" Height="60" Source="images/car.jpg" Margin="30">
     <Image.RenderTransform>
      <SkewTransform AngleX="45" AngleY="0" CenterX="25" CenterY="25"/>
     </Image.RenderTransform>
   </Image>
   <Image Width="90" Height="60" Source="images/car.jpg" Margin="30">
     <Image.RenderTransform>
```

```
      <SkewTransform AngleX="0" AngleY="45" CenterX="25" CenterY="25"/>
    </Image.RenderTransform>
  </Image>
</StackPanel>
```

5. MatrixTransform

You can create unique transformation with using MatrixTransform, for which the previously known classes (RotateTransform, SkewTransform, ScaleTransform, TranslateTransform) are not suitable. The connected chapters of 'Kovács, Hernyák, Radvány & Király 2005' is offered to get acquainted with the theoretical background of matrix transformations.

A Transzformációs mátrix elemei:

M11 transformation matrix (1,1) element value,

M12 transformation matrix (1,2) element value,

M21 transformation matrix (2,1) element value,

M22 transformation matrix (2,2) element value,

OffsetX transformation matrix (3,1) element value,

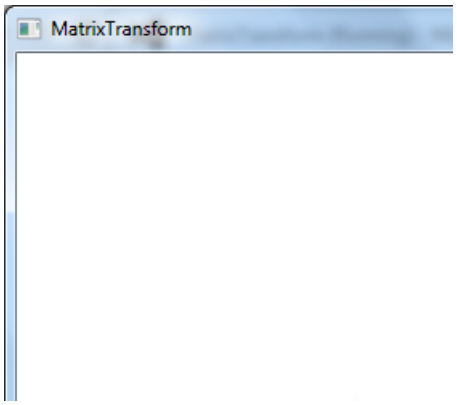
OffsetY transformation matrix (3,2) element value.

The matrix used by WPF has the following structure:

M11	M12	0
M21	M22	0
OffsetX	OffsetY	1

As in the case of an affine transformation matrix, the last column value is (0,0,1), so we only need to define the first two column elements.

Example VIII.4 MatrixTransform



VIII.5. MatrixTransform

```
<Image Width="90" Height="60" Source="car.jpg">
```

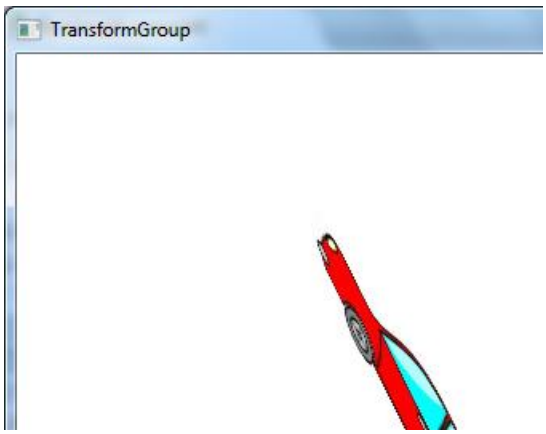
```
    <Image.RenderTransform>
      <MatrixTransform>
        <MatrixTransform.Matrix>
            <Matrix OffsetX="10" OffsetY="1" M11="3" M12="2"/>
        </MatrixTransform.Matrix>
      </MatrixTransform>
    </Image.RenderTransform>
</Image>
```

6. TransformGroup

TransformGroup has to be used if the wanted effect cannot be achieved.

```
<TransformGroup>
   Gyerekelemek
</TransformGroup>
```

Example VIII.5 TransformGroup 1

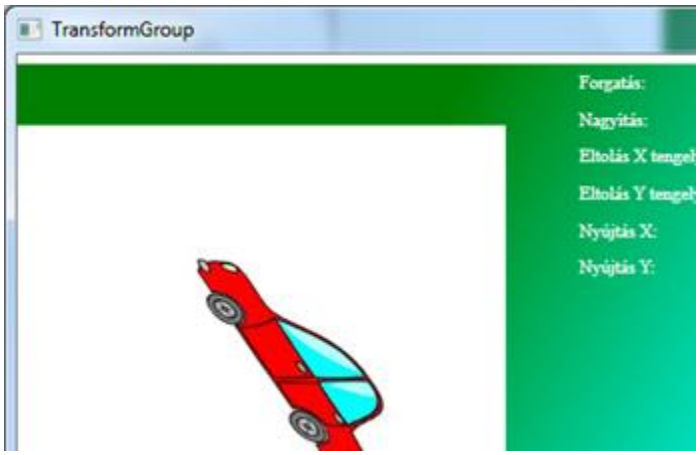


VIII.6. TransformGroup

```
<Image Width="90" Height="60" Source="car.jpg">
    <Image.RenderTransform>
      <TransformGroup>
        <MatrixTransform>
            <MatrixTransform.Matrix >
              <Matrix OffsetX="10" OffsetY="1" M11="3" M12="2"/>
            </MatrixTransform.Matrix>
        </MatrixTransform>
        <RotateTransform Angle="30"/>
```

```
            <TranslateTransform X="-20" Y="-110"/>
        </TransformGroup>
    </Image.RenderTransform>
</Image>
```

Example VIII.7 TransformGroup 2



VIII.7. TransformGroup

Code-behind:

```
namespace TranszformGroup
{
/// <summary>
/// Interaction logic for Window1.xaml
/// </summary>
public partial class Window1 : Window
{
    TransformGroup tg = new TransformGroup();
    RotateTransform rt = new RotateTransform();
    ScaleTransform st = new ScaleTransform();
    TranslateTransform ttr = new TranslateTransform();
    SkewTransform skt = new SkewTransform();
public Window1()
{
   InitializeComponent();
}
private void Window_Loaded(object sender, RoutedEventArgs e)
```

```
{
  try
   {
     BitmapImage bi = new BitmapImage();
     bi.BeginInit();
bi.UriSource = new Uri(System.Windows.Forms.Application.StartupPath + @"\kutya.jpg", UriKind.RelativeOrAbsolute);
bi.EndInit();
image1.Source = bi;
}
catch (FileNotFoundException)
{
MessageBox.Show("Hiba! A bemutatóhoz szükséges kép nem nyitható meg. Kérem, válassza ki manuálisan a Transzformáció fül Megnyitás gombjával!");
}
}
private void Megnyit_Click(object sender, RoutedEventArgs e)
{
System.Windows.Forms.OpenFileDialog OpenFile = new System.Windows.Forms.OpenFileDialog();
OpenFile.CheckFileExists = true;
OpenFile.FileName = String.Empty;
OpenFile.Filter = "Képfájlok(*.BMP;*.JPG;*.GIF)|*.BMP;*.JPG;*.GIF|Összes fájl (*.*)|*.*";
  if (OpenFile.ShowDialog() == System.Windows.Forms.DialogResult.OK)
    {
      BitmapImage bi = new BitmapImage();
      bi.BeginInit();
      bi.UriSource = new Uri(OpenFile.FileName, UriKind.RelativeOrAbsolute);
      bi.EndInit();
      image1.Source = bi;
     }
}
private void forgatasSlider_ValueChanged(object sender,
RoutedPropertyChangedEventArgs<double> e)
```

```
{
rt.Angle = forgatasSlider.Value;
rt.CenterX = image1.ActualWidth / 2;
rt.CenterY = image1.ActualHeight / 2;
tg.Children.Add(rt);
image1.RenderTransform = tg;
}
```

9. fejezet - Effects (written by Csaba Biró)

One of the great power of WPF is that almost all GUI elements can easily be associated with different effect. Various effects can be created with using effects (shading, outer glow, embossing, etc.). Every effect is the descendant of System.Windows.Media.Effects.Effect class.

1. Effects

First, let's look through several tasks how can we create different shades for individual controls.

1.1. DropShadowEffect

The easiest way is the adaptation of DropShadowEffect, which place a shadow behind the element.

DropShadowEffect properties

BlurRadius shadow blur radius,

Color shadow color,

Direction shadow direction,

Opacity shadow opacity,

RenderingBias shadow rendering bias,

ShadowDepth shadow depth.

Example IX.1 Effekt hozzárendelése egy Border-hez

```
<Border>
    <Border.Effect >
        <DropShadowEffect
        ShadowDepth=""
        Opacity=""
        Color=""
        Direction=""/>
    </Border.Effect>
</Border>
```

Examlpe IX.2 Shadow1

IX.1. Shadow 1

```
<TextBlock Text="Shadow1" Foreground="Teal" FontSize="22">
        <TextBlock.Effect>
```

```
        <DropShadowEffect
                ShadowDepth="4"
                Direction="330"
                Color="Red"
                Opacity="0.5"
                BlurRadius="4"/>
    </TextBlock.Effect>
</TextBlock>
```

Example IX.3 Shadow2

IX.2. Shadow 2

```
<TextBlock Foreground="Black" Text="Shadow2"
  Grid.Column="0" Grid.Row="0">
    <TextBlock.RenderTransform>
        <TranslateTransform X="3" Y="3" />
    </TextBlock.RenderTransform>
</TextBlock>
<TextBlock Foreground="Coral" Text="Shadow2" Grid.Column="0" Grid.Row="0">
</TextBlock>
```

1.2. BlurEffect

The GUI elements seem to be blurred with the help of it.

BlurEffect properties:

Radius blur radius,

KernelType kernel type

RenderingBias rendering bias.

Example IX.4 BlurEffect

IX.3. BlurEffect

```
<TextBlock Text="What’s this?" Foreground="Green" FontSize="40"
```

```
 Grid.Column="0" Grid.Row="0" >
<TextBlock.Effect>
    <BlurEffect
      Radius="16"
      KernelType="Gaussian"/>
</TextBlock.Effect>
</TextBlock>
```

2. BitmapEffects

BitmapEffects are from the System.Windows.Media.Effects.Effects.BitmapEffect class. They can be used ubiquitously, more parameterized than the previously known effects.

2.1. DropShadowBitmapEffect

A shadow can be placed behind an item with the help of this application.

Properties:

Color shadow color,

Direction shadowdirection,

Opacity shadow opacity,

Softness shadow softness.

2.2. OuterGlowBitmapEffect

OuterGlowBitmapEffect can be used for adding a light circle to the element.

Properties:

GlowColor light color

GlowSize light size,

Opacity light opacity.

2.3. BlurBitmapEffect

The GUI elements seem to be blurred with the help of it.

Properties:

Radius blur radius,

KernelType kernel type,

2.4. EmbossBitmapEffect

Patterns, depth can be assigned to an objects.

Properties:

LightAngle light angle,

Relief embossing rate.

2.5. BevelBitmapEffect

The previous EmbossBitmapEffect is supplemented with setting the width of the projection, the edge profile and smoothness.

LightAngle light angle,

Relief embossing rate,

BevelWidth bevel width,

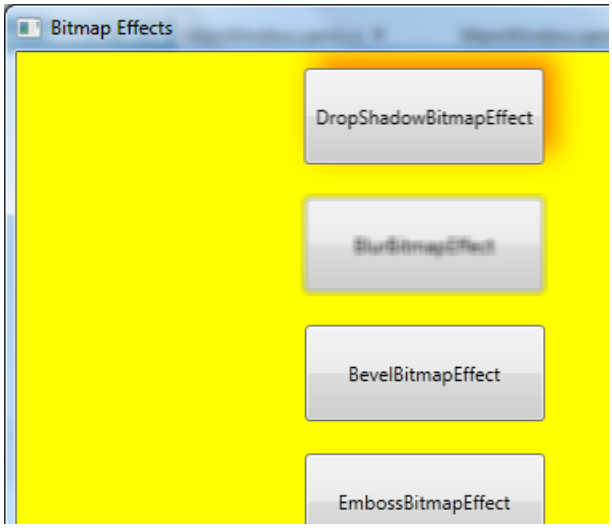
EdgeProfile edge profile,

Smoothness smoothness.

2.6. BitmapEffectGroup

BitmapEffectGroup has to be used if you would like to add even more effect to an item. Any number of effects can be assigned in the Children collection.

Example IX.5 BitmapEffectGroup 1



IX.4. BitmapEffectGroup

```
<StackPanel Background="yellow">
    <Button Width="150" Height="60" Content="DropShadowBitmapEffect" Margin="10">
        <Button.BitmapEffect>
            <DropShadowBitmapEffect Color="Red"
                                    Direction="45"
                                    ShadowDepth="10"
                                    Opacity="0.5"
                                    Softness="1" />
        </Button.BitmapEffect>
```

```
    </Button>
    <Button Width="150" Height="60" Content="BlurBitmapEffect" Margin="10">
        <Button.BitmapEffect>
            <BlurBitmapEffect KernelType="Box" Radius="2" />
        </Button.BitmapEffect>
    </Button>
    <Button Width="150" Height="60" Content="BevelBitmapEffect" Margin="10">
        <Button.BitmapEffect>
            <BevelBitmapEffect BevelWidth="10"
                               EdgeProfile="BulgedUp"
                               LightAngle="270"
                               Relief="0.7"
                               Smoothness="0.3"/>
            </Button.BitmapEffect>
    </Button>
    <Button Width="150" Height="60" Content="EmbossBitmapEffect" Margin="10">
        <Button.BitmapEffect>
            <EmbossBitmapEffect LightAngle="270"
                               Relief="2"/>
    </Button.BitmapEffect>
    </Button>
    <Button Width="150" Height="60" Content="OuterGlowBitmapEffect" Margin="10">
        <Button.BitmapEffect>
            <OuterGlowBitmapEffect GlowColor="Red"
                                   GlowSize="10"
                                   Noise="0.7"
                                   Opacity="0.5"/>
          </Button.BitmapEffect>
     </Button>
</StackPanel>
```

Example IX.6 BitmapEffectGroup 2

```
<Button Content="BitmapEffectGroup" Height="50" Width="150">
```

```
    <Button.BitmapEffect>
        <BitmapEffectGroup>
            <DropShadowBitmapEffect Color="Beige"/>
            <BlurBitmapEffect KernelType="Box" Radius="2"/>
            <BevelBitmapEffect LightAngle="20"/>
        </BitmapEffectGroup>
    </Button.BitmapEffect>
</Button>
```

10. fejezet - Triggers (written by Csaba Biró)

Triggers often be assigned to styles. We can set how a control reacts to the occurring of a particular event, or to the change of a feature. They provide supports for Style, ControlTemplate, DataTemplate (Chapter XIV) and FrameworkElement classes.

There are five different trigger, they are the following:

1. Trigger
2. DataTrigger
3. EventTrigger
4. MultiTrigger
5. MultiDataTrigger

1. Trigger

This is the simplest and the most frequently used 'trigger' is activated (validates the changes assigned to the controller) when the feature of the trigger fills any requirements.

Example X.1 Property Trigger 1

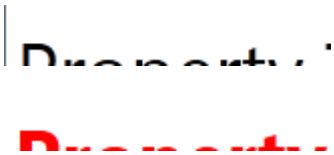
In the following example, we produced a trigger to the style can be assigned for buttons, which monitors the IsPressed property, and when the condition becomes true, a bond font style will be set on the text of the pushbutton.

X.1. Property trigger 1

```
<Grid>
   <Grid.Resources>
         <Style x:Key="Trigger" TargetType="Button">
            <Style.Triggers>
               <Trigger Property="IsPressed" Value="True">
                  <Setter Property="FontWeight" Value="Bold"/>
               </Trigger>
            </Style.Triggers>
         </Style>
   </Grid.Resources>
<Button  FontSize="30" Height="50" Width="250"
```

```
    Style="{StaticResource  ResourceKey=Trigger}">
    Property trigger
</Button>
</Grid>
```

Example X.2 Property Trigger 2



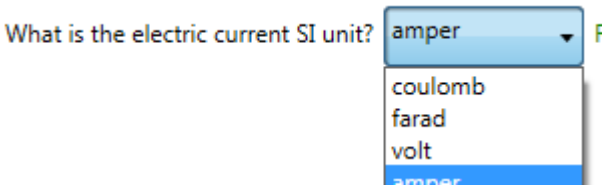
X.2. Property trigger 2

```
<Grid.Resources>
        <Style x:Key="Trigger2">
          <Style.Triggers>
            <Trigger Property="Label.IsMouseOver" Value="True">
              <Setter Property="Label.FontWeight" Value="Bold"/>
              <Setter Property="Label.Foreground" Value="Red" />
            </Trigger>
          </Style.Triggers>
        </Style>
    </Grid.Resources>
<Label FontSize="38" Style="{StaticResource ResourceKey=Trigger2}">Property Trigger 2</Label>
</Grid>
```

2. DataTrigger

DataTrigger operation is similar to the previously known PropertyTigger, except that in this case any of data control expression can be used as the source of the Trigger.

Example X.3 DataTrigger



X.3. DataTrigger

```
<Grid>
    <Grid.Resources>
```

```
        <Style x:Key="DataTrigger" TargetType="Label">
            <Setter Property="Foreground" Value="White" />
            <Style.Triggers>
              <DataTrigger Binding="{Binding ElementName=cbElectric, Path=SelectedIndex}" Value="3">
                  <Setter Property="Foreground" Value="Green" />
               </DataTrigger>
            </Style.Triggers>
        </Style>
    </Grid.Resources>
    <StackPanel Orientation="Horizontal">
     <Label Content="What is the electric current SI unit?" Height="30" />
     <ComboBox x:Name="cbElectric" Width="100" Height="30">
          <ComboBoxItem Content="coulomb"/>
          <ComboBoxItem Content="farad"/>
          <ComboBoxItem Content="volt"/>
          <ComboBoxItem Content="amper"/>
     </ComboBox>
     <Label Content="Right!" Height="30"
         Style="{StaticResource ResourceKey=DataTrigger}"/>
    </StackPanel>
</Grid>
```

3. MultiTrigger and MultiDataTrigger

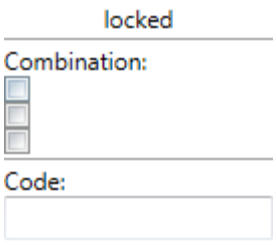
We have an opportunity to specify only one condition in a case of a simple Trigger or DataTrigger. If you want to specify multiple conditions, the MultiTrigger and MultiDataTrigger as complex triggers should be used.

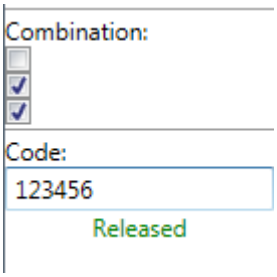
The conditions can be placed among the Conditions tags through these triggers.

If more than one copy of Condition defined, the trigger effect will be adaptive when all conditions are satisfied.

„Or” relationship is possible between each conditions.

Example X.4 MultiTrigger





X.4. MultiTrigger

```
<Window x:Name="window" x:Class="MultiTrigger.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="200">
    <Grid>
        <Grid.Resources>
            <DataTemplate DataType="{x:Type sys:String}">
                <StackPanel>
                    <TextBlock x:Name="tBlock" Text="{Binding}" HorizontalAlignment="Center"></TextBlock>
                    <StackPanel>
                        <Separator />
                        <TextBlock Text="Combination: " />
                        <CheckBox x:Name="cBox1"/>
                        <CheckBox x:Name="cBox2" />
                        <CheckBox x:Name="cBox3" />
                    </StackPanel>
                    <Separator />
                    <StackPanel>
                        <TextBlock Text="Code:" />
                        <TextBox x:Name="tBox"  />
                    </StackPanel>
                    <TextBlock x:Name="tBlock2" Text="Released" Foreground="Green"
                               HorizontalAlignment="Center"
                               Visibility="Hidden" />
                </StackPanel>
```

```
        <DataTemplate.Triggers>
          <MultiTrigger>
            <MultiTrigger.Conditions>
              <Condition
                                SourceName="cBox1"
                                Property="IsChecked"
                                Value="False" />
              <Condition
                                SourceName="cBox2"
                                Property="IsChecked"
                                Value="True" />
              <Condition
                                SourceName="cBox3"
                                Property="IsChecked"
                                Value="True" />
              <Condition
                                SourceName="tBox"
                                Property="Text"
                                Value="123456" />
            </MultiTrigger.Conditions>
            <MultiTrigger.Setters>
              <Setter
                                TargetName="tBlock2"
                               Property="Visibility" Value="Visible" />
              <Setter TargetName="tBlock" Property="Visibility" Value="Collapsed" />
            </MultiTrigger.Setters>
          </MultiTrigger>
        </DataTemplate.Triggers>
      </DataTemplate>
    </Grid.Resources>
    <ContentControl>
      <sys:String>locked</sys:String>
```

```
      </ContentControl>
   </Grid>
</Window>
```

4. EventTrigger

The event trigger is slightly different from the previously known studies. These triggers allows to control animations started by managed events.

Example X.5 EventTrigger 1

```
<Grid>
    <ToggleButton Width="60" Height="40" Content="Speech">
       <ToggleButton.Triggers>
          <EventTrigger RoutedEvent="ToggleButton.Checked">
             <SoundPlayerAction Source="c:\windows\media\Speech On.wav" />
          </EventTrigger>
          <EventTrigger RoutedEvent="ToggleButton.Unchecked">
             <SoundPlayerAction Source="c:\windows\media\Speech Off.wav" />
          </EventTrigger>
       </ToggleButton.Triggers>
   </ToggleButton>
</Grid>
```

Example X.6 EventTrigger 2

```
<Image Source="Neptunusz.jpg" Width="100" >
    <Image.Triggers>
      <EventTrigger RoutedEvent="Image.MouseEnter">
        <BeginStoryboard>
          <Storyboard Storyboard.TargetProperty="Width">
            <DoubleAnimation  Duration="0:0:3" To="200"  />
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger>
      <EventTrigger RoutedEvent="Image.MouseLeave">
        <BeginStoryboard>
          <Storyboard Storyboard.TargetProperty="Width">
```

```
            <DoubleAnimation  To="100" Duration="0:0:3" />
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger>
    </Image.Triggers>
</Image>
```

11. fejezet - Animations (written by Biró Csaba)

The animation is actually rapid projected succession of images. We need to revise our knowledge about dependency properties, managed events, and triggers. WPF provides 42 animation classes located in the System.Windows.Media.Animation namespace.

These classes can be classified into the following three groups:

1. Linear animations:

Animations in this group can be used most commonly because of their simplicity.

Actually, the value of a dependency property changes gradually from a starting and the final point. The linear animation format <Type>Animation, where <Type> is the name of the animation type, for example Color, Double, etc.

1. Key frame-based animations:

An important feature of these animations that the beginning and the final value is not fixed (they are different from the previous one), so the value of the dependency property is changed arbitraryly at any given moment. The key frame-based animation format: <Type> AnimationUsingKeyFrames, where <Type> is the name of the animation type, for example: String, Double, etc.

1. Path-based animations:

Path-based animations we can reach the movement of the object with tracking the specified path. The path-based animation format: <Type>AnimationUsingPath, where <Type> is the name of the animation type, for example: Point, Double, Matrix

1. Animations core classes

Familiarise two important classes before the widely investigation. The first is the Timeline as a centre element of every animations and the „father" of all the animation- types (e.g DoubleAnimation, MatrixAnimationUsingPath). The timeline is actually nothing more than a stretch of time, which has a starting point and a period of time can be assigned to it.

Its role is to oversee the set time frame, determines the length of time, the repetition, etc. Another important class is Storyboard actually perceived as a special timeline will be used for animation creation. This class will be responsible for controlling the animation of the child class (defines which object and property has to be targetted by animations).

```
<Storyboard>

        <DoubleAnimation Duration="0:0:10" From "1" To="200" />

</Storyboard>
```

The Duration attribute can specify the animation duration (in hours, minutes, second format). From signs the initial value of the related property, while To indicates the target one. You can use the By property aswell which is the difference between the initial and the final value. If To and By properties are given, the compiler ignores the value of the By property.

The linked property (TargetProperty) and the related object name (TargetName) can be indicated in two ways (Example XI.1, Example XI.2)

Example XI.1

```
<Storyboard TargetName="rect" TargetProperty="Height">
```

```
        <DoubleAnimation Duration="0:0:10" From "1" To="200" />
</Storyboard>
```

Example XI.2

```
<Storyboard>
    <DoubleAnimation Duration="0:0:10" From "1" To="200"
                Storyboard.TargetName="rect"
                Storyboard TargetProperty="Height">/>
</Storyboard>
```

2. Other important properties

The AccelerationRatio property accepts a value between 0.0 and 1.0 is a percentage of the animation duration, while the animation accelerates from the beginning.

The DecelerationRatio also receive values between 0.0 and 1.0 is actually a percentage of the animation duration, while the animation slows down.

The value of the AutoReverse property can be True or False. If the animation reaches the end, vice playback occurs when you choose the first value.

With using BeginTime property it is possible to delay the start of the animation. Format: hours:minutes:secons.

FillBehavior is the property to determine what happens when the animation is over. There are two possible values: HoldEnd and Stop.

The RepeatBehavior property specifies the number of times the animation is repeated.

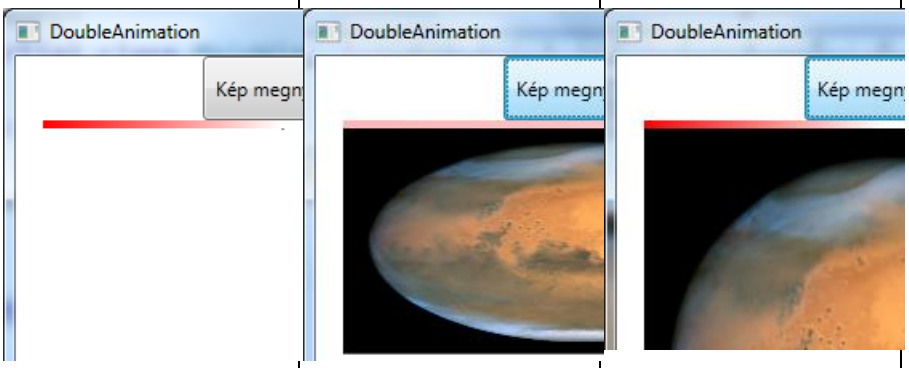
The SpeedRatio property can change the playback speed of the animation relative to the parent timeline.

Example XI.3 ColorAnimation

```
<Grid>
    <Rectangle Width="100" Height="100" Fill="Black">
      <Rectangle.Triggers>
        <EventTrigger RoutedEvent="Rectangle.MouseDown">
          <EventTrigger.Actions>
            <BeginStoryboard>
             <Storyboard>
               <ColorAnimation To="Red" Duration="0:0:10"
                          Storyboard.TargetProperty="Fill.Color"
                          AutoReverse="True"
                          AccelerationRatio="0.1"
                          DecelerationRatio="0.6"/>
             </Storyboard>
```

```
            </BeginStoryboard>
            </EventTrigger.Actions>
          </EventTrigger>
        </Rectangle.Triggers>
      </Rectangle>
</Grid>
```

Example XI.4 DoubleAnimation



XI.1. DoubleAnimation

```
<Window x:Class="DoubleAnimation.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="DoubleAnimation" Height="400" Width="350">
  <Window.Resources>
    <Storyboard x:Key="sbpicture"
            FillBehavior="HoldEnd">
      <DoubleAnimation Storyboard.TargetName="rectmars"
            Storyboard.TargetProperty="Width" From="1" To="300"
            Duration="0:0:1" BeginTime="0:0:0"/>
      <DoubleAnimation Storyboard.TargetName="rectmars"
            Storyboard.TargetProperty="Height" From="1" To="300"
            Duration="0:0:1" BeginTime="0:0:1"/>
    </Storyboard>
    <Storyboard x:Key="sbheader"
            FillBehavior="Stop">
      <ColorAnimation Storyboard.TargetName="stop1" Storyboard.TargetProperty="Color"
```

```
                From="Red" To="White" Duration="0:0:1" BeginTime="0:0:0"/>
        <ColorAnimation Storyboard.TargetName="stop1" Storyboard.TargetProperty="Color"
                From="White" To="Red" Duration="0:0:1" BeginTime="0:0:0.5"/>
        <ColorAnimation Storyboard.TargetName="stop1" Storyboard.TargetProperty="Color"
                From="Red" To="White" Duration="0:0:1" BeginTime="0:0:1"/>
        <ColorAnimation Storyboard.TargetName="stop2" Storyboard.TargetProperty="Color"
                From="White" To="Red" Duration="0:0:1" BeginTime="0:0:0"/>
        <ColorAnimation Storyboard.TargetName="stop2" Storyboard.TargetProperty="Color"
                From="Red" To="White" Duration="0:0:1" BeginTime="0:0:0.5"/>
        <ColorAnimation Storyboard.TargetName="stop2" Storyboard.TargetProperty="Color"
                From="White" To="Red" Duration="0:0:1" BeginTime="0:0:1"/>
    </Storyboard>
</Window.Resources>
<Grid>
    <StackPanel>
    <Button x:Name="button1" Click="button1_Click"
          Content="Open image"
          Width="100" Height="40"></Button>
        <Rectangle Height="5" x:Name="rect1" Width="300">
         <Rectangle.Fill>
          <LinearGradientBrush x:Name="brush" StartPoint="0,0"  EndPoint="1,1">
                <GradientStop x:Name="stop1" Offset="0" Color="Red"/>
                <GradientStop x:Name="stop2" Offset="0.5" Color="White"/>
          </LinearGradientBrush>
         </Rectangle.Fill>
        </Rectangle>
          <Rectangle Name="rectmars" Width="1" Height="1" >
        <Rectangle.Fill>
          <ImageBrush ImageSource="Mars.jpg"/>
        </Rectangle.Fill>
    </Rectangle>
    </StackPanel>
```

```
    </Grid>
</Window>
```

Code-behind:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
Storyboard sp = (Storyboard)TryFindResource("sbpicture");
Storyboard sh = (Storyboard)TryFindResource("sbheader");
sp.Begin();
sh.Begin();
}
```

3. Key frame-based animation

In the case of Key frame-based animations the control can be carried out with using keyframes different from the previously known types. Due to the key frames we can create not only linear curved, or constant rate-changing animations. The Key frame-based animation format: <Type>AnimationUsingKeyFrames as we have already discussed above.

Before we go on, here is a new feature appear to be acquainted KeyTime. The key dates format: hours:minutes:seconds. But that is not the only way to enter, it can be expressed in a percentage and in a written form aswell. You must have the animation duration too if the KeyTime was given in a percentage.

The duration of the animation is divided evenly between the keyframes when the KeyTime property is adjusted to Uniform, but in the case of scheduled (Paced) signing the WPF tries to cope with keyframes achieving a constant rate.

Key frame-based animation syntax:

```
    <<Type>AnimationUsingKeyFrames>
            <Interpoláció><Type>KeyFrame
    </<Type>AnimationUsingKeyFrames>
```

In the following child elements interpolations are available:

Linear interpolation

```
<LinearDoubleKeyFrame Value="" KeyTime="" />
```

Discrete interpolation

```
<DiscreteDoubleKeyFrame Value="" KeyTime="" />
```

Spline interpolation

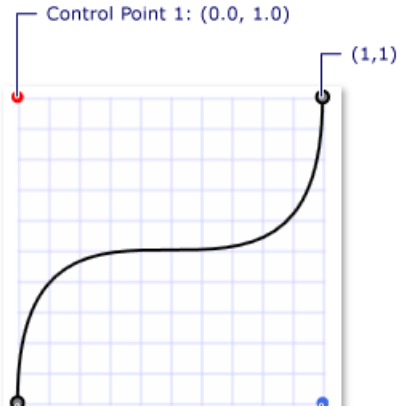
```
<SplineDoubleKeyFrame Value="" KeyTime="" KeySpline="" />
```

All in all, LinearDoubleKeyFrame is a double-valued keyframe using linear interpolation method for insertion.

In the case of the KeySpline property, a cubic Bezier curve has to be parameterized. The starting (0,0) and the final point (1.1) is given, we need to select two conrol point. The first control point effect to the first segment of the curve, the second to the second one. The resulting curve can describe the rate of change can be used to describe very different physical movements (e.g falls, bouncing balls, etc.).

Example XI.4KeySpline 1

Control points (0.0, 1.0) and (1.0, 0.0).

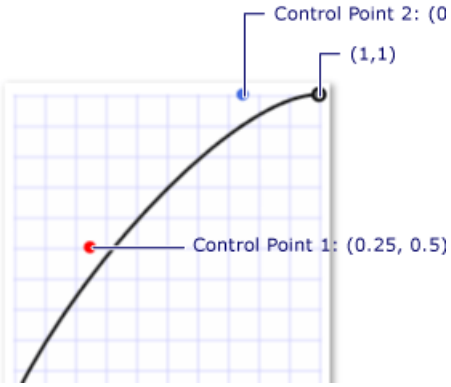


XI.2. KeySpline 1

<SplineDoubleKeyFrame Value="500" KeyTime="0:0:7" KeySpline="0.0,1.0 1.0,0.0" />

Example XI.1 KeySpline 2

Control points (0.25, 0.5) and (0.75, 1).



XI.3. KeySpline 2

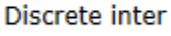
<SplineDoubleKeyFrame Value="350" KeyTime="0:0:15" KeySpline="0.25,0.5 0.75,1" />

Example XI.5 Linear interpolation

```
<Canvas Width="300" Height="300">
    <Ellipse x:Name="Ball" Width="30" Height="30" Fill="Red" >
        <Ellipse.Triggers>
            <EventTrigger RoutedEvent="MouseDown">
                <BeginStoryboard>
                    <Storyboard>
                        <DoubleAnimationUsingKeyFrames
                            Storyboard.TargetName="Ball"
                            Storyboard.TargetProperty="(Canvas.Top)">
```

```
                <LinearDoubleKeyFrame KeyTime="0:0:2" />
                <LinearDoubleKeyFrame Value="100"  KeyTime="0:0:8" />
                <LinearDoubleKeyFrame Value="200"  KeyTime="0:0:10" />
               </DoubleAnimationUsingKeyFrames>
              </Storyboard>
             </BeginStoryboard>
            </EventTrigger>
          </Ellipse.Triggers>
    </Ellipse>
</Canvas>
```

Example XI.6 Discrete interpolation

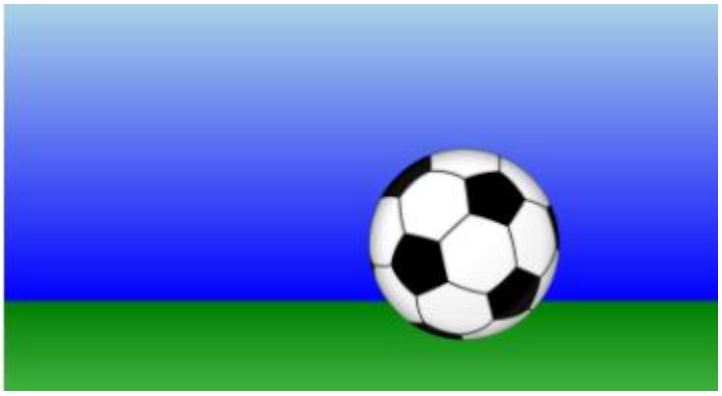


XI.4. Discrete interpolation

```
<StackPanel HorizontalAlignment="Center">
<Label Name="animLabel" Margin="200"
   FontFamily="Verdana">Kattints ide!
   <Label.Triggers>
     <EventTrigger RoutedEvent="Label.MouseDown">
       <BeginStoryboard>
        <Storyboard>
         <StringAnimationUsingKeyFrames
           Storyboard.TargetName="animLabel"
           Storyboard.TargetProperty="(Label.Content)"
           Duration="0:0:9" FillBehavior="HoldEnd">
            <DiscreteStringKeyFrame Value="Discrete " KeyTime="0:0:0" />
            <DiscreteStringKeyFrame Value="Discrete i" KeyTime="0:0:1" />
            <DiscreteStringKeyFrame Value="Discrete in" KeyTime="0:0:1.5" />
            <DiscreteStringKeyFrame Value="Discrete int" KeyTime="0:0:2" />
            <DiscreteStringKeyFrame Value="Discrete inte" KeyTime="0:0:2.5" />
            <DiscreteStringKeyFrame Value="Discrete inter" KeyTime="0:0:3" />
            <DiscreteStringKeyFrame Value="Discrete interp" KeyTime="0:0:3.5" />
```

```
            <DiscreteStringKeyFrame Value="Discrete interpo" KeyTime="0:0:4" />
            <DiscreteStringKeyFrame Value="Discrete interpol" KeyTime="0:0:4.5" />
            <DiscreteStringKeyFrame Value="Discrete interpola" KeyTime="0:0:5" />
            <DiscreteStringKeyFrame Value="Discrete interpolat" KeyTime="0:0:5.5" />
            <DiscreteStringKeyFrame Value="Discrete interpolati" KeyTime="0:0:6" />
            <DiscreteStringKeyFrame Value="Discrete interolatio" KeyTime="0:0:6.5" />
            <DiscreteStringKeyFrame Value="Discrete interpolation" KeyTime="0:0:7" />
            <DiscreteStringKeyFrame Value=" Discrete interpolation!" KeyTime="0:0:7" />" KeyTime="0:0:7.5"
/>
          </StringAnimationUsingKeyFrames>
        </Storyboard>
       </BeginStoryboard>
      </EventTrigger>
   </Label.Triggers>
 </Label>
</StackPanel>
```

Example XI.7 Spline interpolation



XI.5. Spline interpolation

```
<Window.Resources>
      <Storyboard x:Key="sbBounce" RepeatBehavior="Forever">
          <ParallelTimeline BeginTime="0:0:0" AutoReverse="True">
             <DoubleAnimationUsingKeyFrames
              Storyboard.TargetName="ellBall"
              Storyboard.TargetProperty="(Canvas.Top)">
                <SplineDoubleKeyFrame KeyTime="0:0:1"
                          KeySpline="0.5,0 1,1"
```

```
                    Value="120"/>
            </DoubleAnimationUsingKeyFrames>
            <DoubleAnimationUsingKeyFrames
             Storyboard.TargetName="ellShadow"
             Storyboard.TargetProperty="Opacity">
               <SplineDoubleKeyFrame KeyTime="0:0:1"
                    KeySpline="0.5,0 1,1"
                    Value="1"/>
            </DoubleAnimationUsingKeyFrames>
          </ParallelTimeline>
       </Storyboard>
   </Window.Resources>
```

You can create more complex animations with using ParalellTimeline – more than one child animation is embed into the Storyboard.

Example XI.8 The control of animations with Triggers

Animations can also be controlled by triggers (BeginStoryBoard – starting Storyboard, PauseStoryBoard – pausing StoryBoard, ResumeStoryBoard – continuing Storyboard, StopStoryBoard – stopping StoryBoard).

XI.6. The control of animations with Triggers

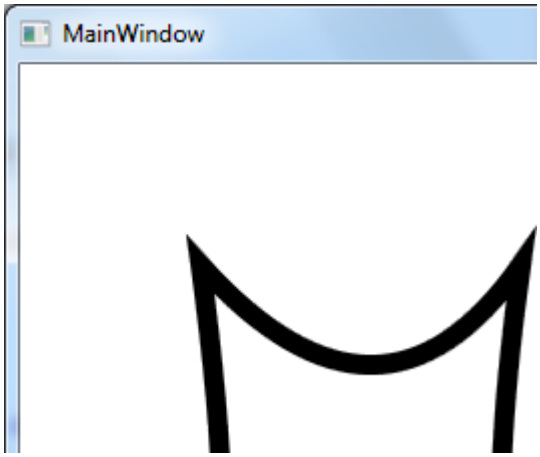
```
<EventTrigger RoutedEvent="Button.Click" SourceName="btnStart">
   <EventTrigger.Actions>
      <BeginStoryboard Name="begSbBounce" Storyboard="{StaticResource sbBounce}"/>
   </EventTrigger.Actions>
</EventTrigger>
<EventTrigger RoutedEvent="Button.Click" SourceName="btnPause">
   <EventTrigger.Actions>
      <PauseStoryboard BeginStoryboardName="begSbBounce"/>
```

```
    </EventTrigger.Actions>
</EventTrigger>
<EventTrigger RoutedEvent="Button.Click" SourceName="btnResume">
    <EventTrigger.Actions>
      <ResumeStoryboard BeginStoryboardName="begSbBounce"/>
    </EventTrigger.Actions>
</EventTrigger>
<EventTrigger RoutedEvent="Button.Click" SourceName="btnStop">
    <EventTrigger.Actions>
       <StopStoryboard BeginStoryboardName="begSbBounce"/>
    </EventTrigger.Actions>
</EventTrigger>
<EventTrigger RoutedEvent="Button.Click" SourceName="btnRemove">
    <EventTrigger.Actions>
      <RemoveStoryboard BeginStoryboardName="begSbBounce"/>
    </EventTrigger.Actions>
</EventTrigger>
```

4. Path-based animation

The values of target properties can be described by curves with the help of Path-based animations. It is worth to place PathGeometry between the resources.

Example XI.9 Path-based animation



XI.7. Path-based animation

```
<Window.Resources>
            <PathGeometry  x:Key="path1">
```

```
            <PathGeometry.Figures>
               <PathFigureCollection>
                  <PathFigure StartPoint="10,280">
                     <PathFigure.Segments>
                        <PathSegmentCollection>
                           <QuadraticBezierSegment Point1="120,380" Point2="80,100" />
                           <QuadraticBezierSegment Point1="170,190" Point2="240,100" />
                           <QuadraticBezierSegment Point1="200,380" Point2="320,280" />
                        </PathSegmentCollection>
                     </PathFigure.Segments>
                  </PathFigure>
               </PathFigureCollection>
            </PathGeometry.Figures>
         </PathGeometry>
      <PathGeometry  x:Key="path2">
         <PathGeometry.Figures>
            <PathFigureCollection>
               <PathFigure StartPoint="20,290">
                  <PathFigure.Segments>
                     <PathSegmentCollection>
                        <QuadraticBezierSegment Point1="130,390" Point2="90,100" />
                        <QuadraticBezierSegment Point1="180,200" Point2="250,100" />
                        <QuadraticBezierSegment Point1="210,390" Point2="330,290" />
                     </PathSegmentCollection>
                  </PathFigure.Segments>
               </PathFigure>
            </PathFigureCollection>
         </PathGeometry.Figures>
      </PathGeometry>
   </Window.Resources>
   <Window.Triggers>
      <EventTrigger RoutedEvent="Window.Loaded">
```

```
        <BeginStoryboard>
          <Storyboard Storyboard.TargetName="feny">
            <DoubleAnimationUsingPath Duration="0:0:10"
                                      Storyboard.TargetProperty="(Canvas.Left)"
          Source="X"
                                PathGeometry="{StaticResource ResourceKey=path1}"/>
            <DoubleAnimationUsingPath Duration="0:0:10"
                                      Storyboard.TargetProperty="(Canvas.Top)"
                                      Source="Y"
                                      PathGeometry="{StaticResource ResourceKey=path1}"/>
          </Storyboard>
        </BeginStoryboard>
    </EventTrigger>
</Window.Triggers>
<Canvas>
    <Path Stroke="Black" StrokeThickness="10" Data="{StaticResource ResourceKey=path2}" />
    <Ellipse x:Name="feny" Width="20" Height="20">
        <Ellipse.Fill>
            <RadialGradientBrush>
                <GradientStop Offset="0.1" Color="Orange"/>
                <GradientStop Offset="0.5" Color="Yellow"/>
                <GradientStop Offset="1" Color="White"/>
            </RadialGradientBrush>
        </Ellipse.Fill>
    </Ellipse>
</Canvas>
```

12. fejezet - Resources and Styles (written by Gergely Kovásznai)

It is tiring and tedious to set certain attributes of certain XAML elements, especially in practice, since GUI elements usually share their design with each other. For example, the buttons on a stylish GUI have the same height, textboxes have the same font, the margins of images are set to the same value, etc. That is to say, WPF must support uniform formatting. Such unification has several advantages. First, it supports source code recycling, i.e. common settings have to be defined only at a single location in the XAML source code; at other locations in the code we just reference to that single definition. Furthermore, it becomes easy to modify the source code, since any modification in the definition immediately appears at other locations in the code.

Styles in WPF serve the above-mentioned purposes. They have been inspired by the Cascading Style Sheets (CSS) used in web development. A style is actually a bunch of formatting settings, as detailed in Section XII.2. But first, in Section Hiba! A hivatkozási forrás nem található., we are introducing the so-called resources, which can be used for similar purposes.

1. Resources

For programmers, a resource usually means an external object (e.g. audio or video file) that can be accessed from any location of the application and used by any module. Resources in WPF have a slightly more general meaning; any object can be used to make a resource from.

As we have already mentioned, each XAML element results in constructing a completely new object. What to do if we intended to use the same object from several GUI locations; e.g. the same image or brush or shape? Let us consider the case when we would like to paint the buttons on our GUI uniformly by using the same brush. For the sake of example, we are going to use an ImageBrush. First, let us add an image file to our project (c.f. Section XVIII.1.1), which is called panda.png and referenced in the source code below, and then add a Resource to our Window, as follows:

```
<Window.Resources>
    <ImageBrush x:Key="pandaBrush" ImageSource="panda.png"/>
    <LinearGradientBrush x:Key="gradientBrush">
        <GradientStop Offset="0.2" Color="DarkBlue"/>
        <GradientStop Offset="0.5" Color="#FF1CB433"/>
        <GradientStop Offset="0.8" Color="DarkBlue"/>
    </LinearGradientBrush>
</Window.Resources>
```

As can be seen, for the sake of example, we have also defined another brush, a LinearGradientBrush, as another resource. It is very important to assign a unique identifier to each of our resources, by using the x:Key attribute, since later we will be able to reference to each resource by its identifier, at any location of Window. Referencing can be done by using a StaticResource element[1], which provides the ResourceKey default property for specifying a resource identifier.

For instance, let us create a few controls (two buttons and one ListBox) in our window, and use the resources defined above.

```
<Button Content="Panda button!"
    Background="{StaticResource ResourceKey=pandaBrush}"/>
```

[1] There exists a DynamicResource as well, however we are not giving details here.

```
<Button Content="Color button!"
        Background="{StaticResource gradientBrush}"/>
<ListBox Background="{StaticResource pandaBrush}">
    <StackPanel Orientation="Horizontal">
        <Image Source="giant_panda.jpg"/>
        <TextBlock Text="Giant panda"></TextBlock>
    </StackPanel>
    <StackPanel Orientation="Horizontal">
        <Image Source="red_panda.jpg"/>
        <TextBlock Text="Red panda"/>
    </StackPanel>
</ListBox>
```

The GUI can be seen in Figure 1. It is important to emphasize that whenever we modify our resources, e.g. we replace panda.png or make the ImageBrush transparent (by setting its Opacity property), the modification gets applied to every GUI elements that use the given resource.

XII.1. Using brushes as resources

2. Styles

There is much room for improving the design of the application above. For instance, it would be nice to apply bigger font size or, perhaps, some coloring to the TextBlocks, and, furthermore, the increase their left margin. Therefore, we are defining a style, which will be referenced from each TextBlock. As can been seen in the source code below, a style (Style) can be defined as a resource. One of its important attributes is called TargetType, which specifies a class (TextBlock in the example) to which the given style can be applied. In the body of a Style one is allowed to give a list of Setters, each of which can be used to specify a concrete value for one of the properties of the class referenced by TargetType.

```
<Window.Resources>
    ...
    <Style TargetType="TextBlock" x:Key="fancyStyle">
```

```
            <Setter Property="FontSize" Value="28"/>
            <Setter Property="FontWeight" Value="Bold"/>
            <Setter Property="VerticalAlignment" Value="Center"/>
            <Setter Property="Margin" Value="40,0,0,0"/>
            <Setter Property="Foreground">
                <Setter.Value>
                    <LinearGradientBrush>
                        <GradientStop Offset="0.2" Color="#FFA4002D"/>
                        <GradientStop Offset="0.5" Color="DarkBlue"/>
                        <GradientStop Offset="0.8" Color="#FFA4002D"/>
                    </LinearGradientBrush>
                </Setter.Value>
            </Setter>
        </Style>
    </Window.Resources>
    <ListBox Background="{StaticResource ResourceKey=pandaBrush}">
        <StackPanel Orientation="Horizontal">
            <Image Source="giant_panda.jpg"/>
            <TextBlock Text="Giant panda" Style="{StaticResource fancyStyle}/>
        </StackPanel>
        <StackPanel Orientation="Horizontal">
            <Image Source="red_panda.jpg"/>
            <TextBlock Text="Red panda" Style="{StaticResource fancyStyle}/>
        </StackPanel>
    </ListBox>
```

As can be seen, Setter.Value can be either some simple value (e.g., number, text) or some compound one, i.e. an instance of a class, just like the LinearGradientBrush in the example.

If a style does not have an x:Key specified, then it becomes the default style for the type given in TargetType. We could modify the source code above for the sake of example. Let us assign a default style (using a DropShadowBitmapEffect, c.f. Section IX.) to the StackPanel type:

```
<Style TargetType="TextBlock">
    ...
</Style>
<Style TargetType="StackPanel">
```

```
        <Setter Property="VerticalAlignment" Value="Center"/>
        <Setter Property="Margin" Value="40,20,0,0"/>
        <Setter Property="BitmapEffect">
            <Setter.Value>
                <DropShadowBitmapEffect/>
            </Setter.Value>
        </Setter>
</Style>
...
<StackPanel Orientation="Horizontal">
        <Image Source="giant_panda.jpg"/>
        <TextBlock Text="Giant panda"/>
</StackPanel>
<StackPanel Orientation="Horizontal">
        <Image Source="red_panda.jpg"/>
        <TextBlock Text="Red panda"/>
</StackPanel>
```

As a result, we get a form similar to the following one:

XII.2. Using styles

Regarding styles, two more facts are still worth to mention. One of them is that styles can be inherited from each other by the use of the Style.BasedOn property. By doing so, one can construct an inheritance tree of styles, in which each style inherits all the Setters of all its ascendants, and it is even able to overwrite them.

The other important thing is the possibility of assigning event handlers to styles. By doing so, styles are able to specify not only outlook, but also behavior. Style.EventSetter is a special setter, which assigns an event handler specified by its Handler property to an event given by its Event property. Let us modify the above example in order to execute event handlers whenever the mouse pointer is dragged over a StackPanel:

```
<Style TargetType="StackPanel">
    ...
    <EventSetter Event="MouseEnter" Handler="StackPanel_MouseEnter"/>
    <EventSetter Event="MouseLeave" Handler="StackPanel_MouseLeave"/>
</Style>
```

Let the StackPanel_MouseEnter event handler rotate the given StackPanel by 5 degrees (by using transformations introduced in Section Hiba! A hivatkozási forrás nem található.), and StackPanel_MouseLeave set them back to their original position (by removing the transformation)! All of these can be implemented in C# as follows:

```
private void StackPanel_MouseEnter(object sender, MouseEventArgs e)
{
    StackPanel s = sender as StackPanel;
    s.RenderTransform = new RotateTransform(5);
    }
private void StackPanel_MouseLeave(object sender, MouseEventArgs e)
{
    StackPanel s = sender as StackPanel;
    s.RenderTransform = null;
}
```

13. fejezet - Data Binding (written by Gergely Kovásznai)

WPF is especially capable to implement dynamic user interfaces. This must involve the ability of updating the data, which are stored in the data model and change dynamically, on the GUI immediately, and also in the other direction: the data that get modified on the GUI (e.g., a phone number in a textbox) have to be immediately updated in the model.

By using the instrumentation we have introduced so far, this can be done the easiest by the help of event handlers. In order to update data in the data model, after being changed on the GUI, we need to handle events of the GUI. Let us assume, for instance, that we overwrite the phone number in the above-mentioned textbox, and then we move the focus out of the textbox. In this very moment, i.e. when the textbox has just lost the focus, we would like to update the corresponding data of the person stored in the model to the one in the textbox. For this, we need to implement an event handler for the LostFocus event of the textbox:

```
private void textbox_LostFocus(object sender, RoutedEventArgs e)

{

    TextBox textbox = sender as TextBox;

      Person.vipPerson.TelNumber = textbox.Text;

}
```

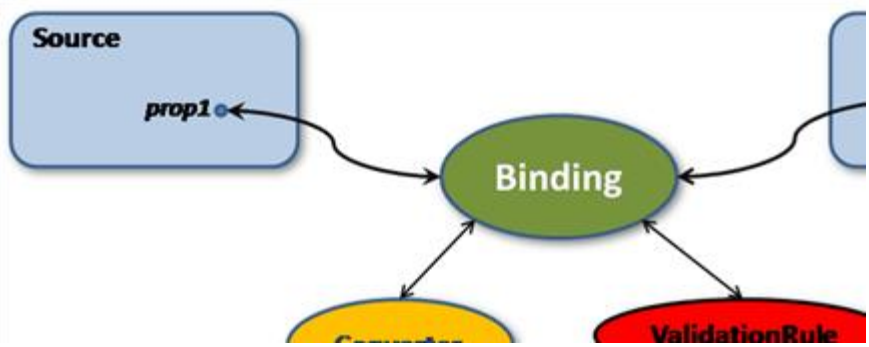
It is even more complicated to realize a flow of data from the model toward the GUI (e.g., the phone number changes in the model and has to be updated on the GUI). We need to define events for certain properties of the classes in the model; such an event is going to be fired whenever the value of the given property changes. Then, for those events, we need to implement event handlers, which are to update all(!) such GUI elements whose values depend on the given property.

Such a development process is inconvenient, in several senses. First, we would need to implement a dedicated event handler for almost each GUI element, and also for each property of each class in the model (e.g. for the static vipPerson property of the Person class). This would be an extremely boring and tedious work, since those event handlers look almost the same. Is it not possible to automate this process? Could we just “bind” the GUI elements to the objects “behind” them, instead of dedicating variables for all of them?

Furthermore, the above-mentioned solution is rather complicated and opens the door for numerous errors. It does not separate the model and the GUI as much as we would expect. In the case of a robust project that involves a lot of developers, the above-mentioned solution would result in total chaos and hell of bugs.

1. The Binding Class

In WPF, data binding frees us from the above-mentioned troubles. The classes that implement data binding are located in the System.Windows.Data namespace. Among them, the Binding class is the most important. In order to bind a source property with a target property, in each case we are going to instantiate Binding. The resulting Binding object can be visualized (c.f. Figure 1) as “taking place” between the source and target properties, in order somehow to mediate between them, by doing synchronization.



XIII.1. The architecture of data binding

The Binding object continuously monitors the properties, marked by us, of the source and target classes. If the value of one of those properties changes, the Binding object does the synchronization between the two, based on the settings we have specified. We have the opportunity to interfere in the process of synchronization by using an optional converter object; e.g., if the data being synchronized are of different types (e.g., one is a string, the other is an integer). One might also optionally apply validation rules, which can be used to check whether the data to set fulfill certain criteria (e.g., a phone number has the correct format).

The following table summarizes the most important properties of the Binding class:

Binding		
Source		The source object.
ElementName		For a special case, when the source object is a GUI element.
Path		The “path” to the property of the source object that we are about to bind.
Mode	OneWay	The target is updated whenever the source changes.
	TwoWay	The target is updated whenever the source changes, and the source is updated whenever the target

		changes.
	OneTime	The target gets a value only at initialization (based on the source).
Converter		The converter object.
StringFormat		Special format for how to display the value of the property.
ValidationRules		Validation rules.

In the table, one might miss the possibility for specifying a target object and a target property. Well, the reason is that in XAML we simply need to assign the Binding instance (containing a reference to the source) to the target property, as we will see soon.

Binding in XAML. It is very intuitive in XAML to use Binding; however, as we have already discussed above, rather complex processes run in the background behind each data binding. One uses Binding in the form of a markup extension most frequently; therefore, we show this form first. The above-mentioned binding between a textbox and the Person.vipPerson.TelNumber property can be written in XAML as follows:

<TextBox Text="{Binding Source={x:Static Member=my:Person.vipPerson},

Path=TelNumber}"/>

In the above example, it is very important that the source is now referenced via a static property; this is why we can reference it in the form classname.propname and use the x:Static tag. (In the case of a non-static source, this could be done in a more complicated way.) By doing so, the compiler can uniquely identify the source; after that, it only has to find the Person class, requiring us to specify an alias (my) to the namespace that contains this class (Section III.3).

Path. As mentioned before, Path can be used for specifying the “path” to a source property. This means that, within the source specification, one can even reference a property of a property of a property etc., i.e., properties of embedded objects (separated by dots). For example, if the Person class has an Address property, which is of such a class which has City, Street etc. properties, then one can specify such a data binding, as follows:

<TextBox Text="{Binding Source={x:Static Member=my:Person.vipPerson},

Path=Address.City}"/>

Mode. The meaning of Mode is self-explanatory. It can have several other values, however the above-mentioned ones are worth to mention. The OneWay mode performs, in fact, only the initialization of a GUI element, later there is no more synchronization between the source and the target. Of course, it is more frequent to use the OneWay and TwoWay modes. In the case of the first one, the value of the source is prevented from any change by data binding, however the target is always updated whenever the value of the source has been modified. In the case of the latter one, the values are synchronized in both directions.

It is optional to specify the Mode, since every GUI element has its own default data binding mode, depending on which mode fits the best to the given GUI element. E.g., in the case of a Label or TextBlock, OneWay is the default mode, while in the case of a TextBox or CheckBox, it is TwoWay.

ElementName. Sometimes it might happen that not only the target of a data binding is a GUI element, but so is the source. Let us imagine a form that contains a TextBox and a Slider! We would like to make the TextBox always display the current value of the Slider, i.e., whenever we drag the Slider, the content of the TextBox should be updated, and, furthermore, whenever we overwrite the content of the TextBox, the Slider should reposition itself. All of these can be easily achieved in XAML, as follows:

```
<Slider Name="sliderToBind" Minimum="0" Maximum="100"/>

<TextBox Text="{Binding ElementName=sliderToBind, Path=Value, Mode=TwoWay}"/>
```

Note that it is necessary to assign a name to the source GUI element since this is the only way for referencing it. Note furthermore that, in the above example, it is actually unnecessary to specify the TwoWay mode, since this is default for a TextBox.

2. Converters

In the above example, it is not easy to notice a hidden WPF service, namely the automatic conversion between data of certain types. As we all know, the TextBox.Text property is a string, while Slider.Value is a double. It is very convenient that programmers do not have to implement such obvious conversions.

However, there are cases when it is not so obvious to realize an appropriate conversion, and, furthermore, it would be desirable if WPF provided opportunities to customize data conversion. Consider the following simple example: let us assume that we are implementing a management software for a German company, and the price of a product (Product.Price) is represented as a double, but should be displayed in the German form of currencies (i.e., truncated to two decimal points and divided into groups of thousands). Therefore we need to create an own converter, a new class called, for instance, PriceConverter, which must implement the IValueConverter interface (in the System.Windows.Data namespace).

```
class PriceConverter : IValueConverter

{

    public object Convert(object value, Type targetType, object parameter,

                                                System.Globalization.CultureInfo culture)

    {

        double price = (double)value;

        return price.ToString("c", culture);

    }

    public object ConvertBack(object value, Type targetType, object parameter,

                                                System.Globalization.CultureInfo culture)

    {

        string price = value as string;

        return double.Parse(price);

    }

}
```

As it can be infered from the above source code, the IValueConverter interface has two methods: Convert converts the source data, while ConvertBack converts the target data. They receive the data they should convert in their value parameter; since both methods must be universal to use, the type of value is object, and, hence, we always need to cast it to the required type. Since, in the current example, the source data (Product.Price) is double, at the beginning of Convert the value parameter is converted to double; and to string in ConvertBack, since the target data (TextBox.Text) is a string.

At the end of Convert, the double data is formatted with respect to the currency format. For this, it is the simplest to use an appropriate .NET service, namely the opportunity for passing a format string as a parameter to the ToString method. The c format string[1] stands for „currency”, and makes the data formatted with respect to system settings and formatting rules used in a given language or region. For instance, if we use Windows with Hungarian language settings, then data is formatted w.r.t. the rules in Hungary. How to force our application, for example, to format prices w.r.t. the German rules? For this, one can use the Language attribute of each GUI element (as the target object of data binding); this attribute is to specify a language/region, and we are now setting it to de[2]. The data binding in XAML is going to look like as follows:

```
<TextBox Language="de">
    <TextBox.Text>
        <Binding Source="{x:Static Member=my:Product.prodToSale}" Path="Price">
            <Binding.Converter>
                <my:PriceConverter/>
            </Binding.Converter>
        </Binding>
    </TextBox.Text>
</TextBox>
```

In the above example, Binding is specified by not using markup extension, on purpose. The reason is that we need to instantiate the PriceConverter class, and this is automatically done when using an XAML tag. Nevertheless, the above source code can demonstrate how to use data binding in the traditional way. If you would rather like to use markup extension, then it is worth to apply the following solution, which is the most popular in literature: to add a PriceConverter instance as a resource to our form or application, and later to reference it at every location where we need a PriceConverter.

```
<Window.Resources>
    <my:PriceConverter x:Key="priceConverterObject"/>
</Window.Resources>
    ...
<TextBox Language="en-US" Text="{Binding
        Source={x:Static my:Product.prodToSale},
        Path=Salary,
        Converter={StaticResource ResourceKey=priceConverterObject}
}"/>
```

[1] There is a bunch of different format strings. One can easily find references and tutorials on the internet, e.g., http://msdn.microsoft.com/en-us/library/26etazsy.

[2] One can find the language/region codes on the internet, e.g., http://msdn.microsoft.com/en-us/goglobal/bb896001.aspx. The code of the Hungarian language is hu-HU, the one of the American English is en-US.

A Complex Example. Let us realize the form in Figure 2. We want to display an image (in the Image at the bottom) which belongs to the dog breed selected in the ListBox. It is a nice solution to create a Dog class in order to represent dog breeds; let this class have two properties called Name and ImageName. For the sake of example, let us introduce the Dog.dogs static list, into which we push a few dog breeds.

```
class Dog
{
    public string Name { set; get; }
    public string ImageName { set; get; }
    public static List<Dog> dogs = new List<Dog> {
        new Dog { Name="Bernese Mountain Dog", ImageName="bernese.jpg" },
        new Dog { Name="Chihuahua", ImageName="chihuahua.jpg" },
        new Dog { Name="Great Dane", ImageName="dane.jpg" },
        new Dog { Name="Komondor", ImageName="komondor.jpg" }
    };
}
```

The corresponding image files will be put next to the compiled program, into the images directory.

XIII.2. Data binding between a ListBox and an Image

First, we bind the Dog.dogs list to our ListBox, as can be seen in the 5th line in the XAML code below. The ItemSource and DisplayMemberPath attributes of list controls will be detailed in the next section; currently the point is that the ListBox receives its items from the Dog.dogs list, and displays their Name property.

```
<Window.Resources>
    <my:DogToImageConverter x:Key="dogConverterObject"/>
</Window.Resources>
```

```
    ...
<ListBox Name="dogListBox"
          ItemsSource="{x:Static my:Dog.dogs}" DisplayMemberPath="Name"/>
<Image Source="{Binding
          ElementName=dogListBox,
          Path=SelectedItem,
          Converter={StaticResource dogConverterObject}
}"/>
```

The Image will show the picture of the currently selected item (SelectedItem) of the ListBox. At this point, it is advantageous to use a converter, since the source of the data binding in the 7th line is a Dog object, while its target is of type ImageSource. Our converter can be implemented as follows:

```
class DogToImageConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
                                                    System.Globalization.CultureInfo culture)
    {
        Dog dog = value as Dog;
        if (dog == null) return null;
        string imagepath = Path.Combine(Directory.GetCurrentDirectory(),
                                                        "images", dog.ImageName);
        return new BitmapImage(new Uri(imagepath));
    }
    public object ConvertBack(object value, Type targetType, object parameter,
                                                    System.Globalization.CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}
```

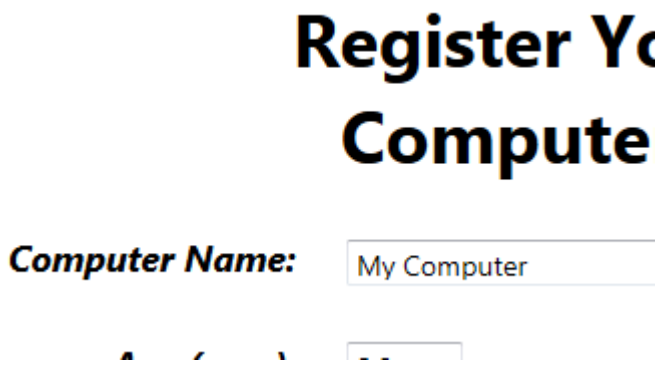
As can be seen, the Dog object passed in the value parameter is “converted” to a BitmapImage object by concatenating the current directory, the images directory, and the name of the image file that belongs to the Dog object. Note that backward conversion is disabled; consequently, even Mode=OneWay could have been used in the data binding for the Image.

3. Validation

As already mentioned in Section XIII.1, in the Binding.ValidationRules property one can specify rules for validating the source data of data binding. It is a usual validation task to check whether the user has filled all the

mandatory textboxes in a form, i.e., none of the mandatory source data is empty. Of course, there also exist more complex validation tasks, e.g., checking whether a phone number is well-formed. Such tasks are supported by different subclasses of the ValidationRule class. We are going to introduce only one of them, called DataErrorValidationRule, and, additionally, the IDataErrorInfo interface. In the next example, we are going to show how to use them, but keep in mind that there are several different ways for realizing validation combined with data binding, c.f. corresponding literature!

Our (slightly unnatural) example: let us create a form on which users can register their own computers. They have to specify three pieces of data: the name of a computer (string), its age in years (double), and its IP address (string). The form should look like the one in Figure 3: around a textbox that contains incorrect data a red frame should appear, and a tooltip with an error message as well if the user drags the mouse pointer above the textbox.



XIII.3. Validation

First, we need to add a DataErrorValidationRule to each textboxes, as the next source code shows:

```
<TextBox>
    <Binding Path="IpAddress">
        <Binding.ValidationRules>
            <DataErrorValidationRule/>
        </Binding.ValidationRules>
    </Binding>
</TextBox>
```

IpAddress referenced by the Binding is, obviously, a property of one of our own classes in the code behind. In order to make this own class “compatible” with DataErrorValidationRule, it must implement the IDataErrorInfo[3] interface. This interface declares an indexer, which receives as parameter the name of the property being validated and checks the value of this property. If the value turns out to be badly formed, then the indexer may return an error message; otherwise, it simply returns null.

```
public class Computer : IDataErrorInfo
{
    public string Name { get; set; }
    public double Age { get; set; }
    public string IpAddress { set; get; }
    public string this[string columnName]
```

[3] The IDataErrorInfo interface can be found in the System.ComponentModel namespace.

```
    {
        get
        {
            switch (columnName)
            {
                case "Name":
                    if (string.IsNullOrEmpty(this.Name))
                        return "Name must not be empty";
                    break;
                case "IpAddress":
                    if (string.IsNullOrEmpty(this.IpAddress))
                        return "IP address must not be empty";
                    var parts = this.IpAddress.Split('.');
                    if (parts.Length != 4)
                        return "IP address must contain 4 numbers, separated by dots";
                    foreach (string part in parts)
                    {
                        int intPart;
                        if (!int.TryParse(part, out intPart)
                                                || intPart < 0 || intPart > 255)
                    return "IP address must contain integers, each between 0 and 255";
                    }
                    break;
            }
            return null;
        }
    }
}
```

As can be seen, validating the Name property only requires to check whether it is empty or it is not. The same check is needed to apply to IpAddress, but this is not sufficient only. IpAddress has to be splitted into several pieces by using dots as delimiters (string.Split), and then one has to check whether the number of the resulting pieces is four. Finally, one has to check whether each piece can be converted to an integer (int.tryParse), and whether its value is between 0 and 255.

Note that we have not specified any validation code for the Age property! In such cases, the default WPF validator is used, depending on the type of the given property (double): it checks whether the textbox is not empty and its content can be converted to double.

When running our current application, one can notice that while red frames appear around the badly formed textboxes, no error message pops up. The reason is that we need to make our textboxes capable to notice validation errors. A relatively easy way is to define a default style (Section XII.2) for textboxes, in which a trigger (Section X) is used to observe the Validation.HasError property of a textbox, whose bool value shows whether such an error has occurred. If it has, the trigger sets the ToolTip property of the textbox to the error message, and all of this can be realized in a slightly complicated way, as follows:

```
<Style TargetType="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>
```

14. fejezet - Templates (written by Gergely Kovásznai)

Styles, as introduced in Section 0, provide uniform format setting for controls and code reuse. In this section, we are going to introduce templates, which serve the same purposes in an even higher level. Furthermore, the use of templates gives us even more power, since they are primarily suitable for customizing controls, besides preserving their functionality. This can be done in a direct way by applying control templates, as introduced in Section Hiba! A hivatkozási forrás nem található., or in an indirect way, via bound data objects, by using data templates, as shown in Section Hiba! A hivatkozási forrás nem található..[1]

1. Control Templates

A control template can be realized by using the ControlTemplate XAML element. It has an important attribute called TargetType, which is for specifying the control type to which the given template can be applied. Since in the next example we are going to create a button that has special outlook and behavior, one has to assign TargetType to Button. In the body of the ControlTemplate, one has to give the XAML code that defines the control. For the sake of example, let the shape of our button be rather irregular; therefore, it is worth to use a Path (Section VII.2).[2] Besides defining all the visual elements (in our example there is only one such element), it is an important task to place the content assigned to the button (Content). For this reason we need to use the ContentPresenter XAML element, and place it to where Content should be displayed, based on the design we apply.

```
<Window.Resources>
    <ControlTemplate TargetType="Button" x:Key="buttonTemplate">
      <Canvas x:Name="canvas">
          <Path Data="M 11,23C 11,11 23,11 23,11L 116,11L 125,58L 80,59L
                    75,45L 67,59L 14,60L 11,23" Stretch="Fill" x:Name="path">
            ...
          </Path>
          <ContentPresenter Canvas.Left="15" Canvas.Top="8"
                            TextBlock.FontFamily="Vivaldi"/>
      </Canvas>
    </ControlTemplate>
</Window.Resources>
...
<Button Template="{StaticResource buttonTemplate}" FontSize="20">
    Get Started
</Button>
```

[1] We do not give details about two other WPF templates, HierarchicalDataTemplate and ItemsPanelTemplate.
[2] This is a rather hard manual task. It is advantageous to use Microsoft's Expression Design (Section XVIII.3), which can even export XAML code (and this is exactly how the example was done).

As can be seen, the template can be specified as a resource, therefore a key (called buttonTemplate now) has to be assigned to it. If one would like to apply the template to a certain GUI control (to the Button now), then the key has to be used in the Template property of the control. In the above example, the Content of the button is set to „Get Started" (it could be anything else), therefore this text is going to be displayed at the location of the ContentPresenter.

As can be seen, we have assigned fixed values to certain properties of the template, e.g., Vivaldi to the text font, which makes all the buttons that apply this template use this font to display texts.[3] Nevertheless, other attributes of a Button can be customized on demand (e.g. FontSize); however, sometimes it is complicated to specify which property of a template is "bound" to which property of a control. E.g., one should rather bind the Width of the Button to the Width of the Path (instead of the one of the default ContentPresenter). This can be achieved by a special variant of data binding (Section Hiba! A hivatkozási forrás nem található.) called TemplateBinding. The properties of a template can be bound to certain properties of a control by the use of TemplateBinding. Below such an example is shown, which binds the contour brush (Stroke) of the Path to the contour brush (BorderBrush) of the Button. The result can be seen in Figure 1.

```
<ControlTemplate TargetType="Button" x:Key="buttonTemplate">
    ...
    <Path ... Width="{TemplateBinding Width}" Stroke="{TemplateBinding BorderBrush}">
        ...
    </Path>
    ...
</ControlTemplate>
...
<Button Template="{StaticResource buttonTemplate}" FontSize="20" Width="150">
    Get Started
</Button>
<Button Template="{StaticResource buttonTemplate}" FontSize="16" BorderBrush="Red">
    Help
</Button>
```

XIV.1. The usage of a control template

The result is however not completely satisfactory since these buttons are static, do not move, and do not react to mouse operations, as buttons are expected to do. It would be so nice if one were able to specify in our template the behavior of buttons as well! How fortunate that this is also possible, by adding triggers (Section Hiba! A hivatkozási forrás nem található.) to a template (ControlTemplate.Triggers). For the sake of example, let us show two such triggers. The first one is going to be a simple Trigger, which starts to react whenever the mouse pointer enters the area of the control. It sets, for instance, the contour of the Path thicker and colored:

```
<ControlTemplate TargetType="Button" x:Key="buttonTemplate">
    ...
```

[3] Templates can be used very well for unifying our GUI design.

```
    <ControlTemplate.Triggers>
        <Trigger Property="IsMouseOver" Value="True">
            <Setter TargetName="path" Property="Stroke" Value="Orange"/>
            <Setter TargetName="path" Property="StrokeThickness" Value="3"/>
        </Trigger>
    </ControlTemplate.Triggers>
</ControlTemplate>
```

Let the other trigger be an EventTrigger, which observes the Click event of the button, and starts to play an animation (Storyboard) (Section Hiba! A hivatkozási forrás nem található.). The illusion of the button descending and ascending is going to be realized by using a translation (TranslateTransform), and, furthermore, we are going to add shadow effect (DropShadowBitmapEffect) to the button and decrease the shadow depth when clicking. Below it can be seen how to realize this, and the flow of animation in Figure 2.

```
<ControlTemplate TargetType="Button" x:Key="buttonTemplate">
    <Canvas x:Name="canvas">
        ...
        <Canvas.RenderTransform>
            <TranslateTransform x:Name="translate"/>
        </Canvas.RenderTransform>
        <Canvas.BitmapEffect>
            <DropShadowBitmapEffect Color="{StaticResource purplish}"
                                    ShadowDepth="20" x:Name="shadow"/>
        </Canvas.BitmapEffect>
    </Canvas>
    <ControlTemplate.Triggers>
        ...
        <EventTrigger RoutedEvent="Button.Click">
            <BeginStoryboard>
                <Storyboard AutoReverse="True">
                    <DoubleAnimation Storyboard.TargetName="shadow"
                                     Storyboard.TargetProperty="ShadowDepth"
                                     To="5" Duration="{StaticResource duration}"/>
                    <DoubleAnimation Storyboard.TargetName="translate"
                                     Storyboard.TargetProperty="X"
                                     To="15" Duration="{StaticResource duration}"/>
```

```
                    <DoubleAnimation Storyboard.TargetName="translate"
                                     Storyboard.TargetProperty="Y"
                                     To="20" Duration="{StaticResource duration}"/>
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger>
    </ControlTemplate.Triggers>
</ControlTemplate>
```

XIV.2. Control template with triggers

2. Data Templates

In Section Hiba! A hivatkozási forrás nem található. we showed an example how to display complex data on a GUI. There we used a ListBox for enumerating dog breeds (own Dog class), and displayed the photo of the breed selected in the ListBox. We implemented this by using the DisplayMemberPath property of the list control and binding controls to each other. WPF provides a more sophisticated toolkit, which improves the limited way in which one can display data on controls. This toolkit is based on the so-called data templates (DataTemplate).

Content controls (with Content property) (Section 0), such as e.g. a Button, have a ContentTemplate property, which can get such a data template as value. For example, if a Dog instance is assigned to Button.Content, then the data template given in Button.ContentTemplate makes it possible to display the name, the photo, and any other property of a given dog on the surface of the button, in any layout and design.

The ItemTemplate property of list controls (Section V.3) has a similar purpose, since it can be used to customize the (uniform) outlook of list elements. In the following example, it can be seen how to assign a DataTemplate to this property.[4] DataType is an important attribute of DataTemplate, and can be used to specify the data type (our Dog class in this case) that the template is allowed to format.

In the following example, we are going to improve the example in Section Hiba! A hivatkozási forrás nem található., by employing and modifying, in a self-explanatory way, the Dog class and the converter that we introduced there. The Path in Section Hiba! A hivatkozási forrás nem található. is going to be used again as a design element. We would like to emphasize the following elements in the source code:

1. The properties of the Dog class are bound (Binding) one by one to the desired controls. For instance, the Weight property to a TextBox (which can even edit the property), the ImageName property to an Image (by using a self-explanatory converter).

2. The controls in the template can be aligned on demand. In the current example, a Grid is used, and, furthermore, a StackPanel in one of the cells of the Grid.

```
<ListBox ItemsSource="{x:Static my:Dog.dogs}">
    <ListBox.ItemTemplate>
        <DataTemplate DataType="my:Dog">
```

[4] Another way to do the same is, of course, to define the DataTemplate as a resource, and to reference its key in the form of ItemTemplate={StaticResource ...}.

```
            <Grid Width="130">
                ...
                <Path ... Grid.RowSpan="2" Grid.ColumnSpan="2">
                    ...
                </Path>
                <TextBlock ... Grid.ColumnSpan="2" Text="{Binding Name}"/>
                <Image ... Grid.Row="1" Source="{Binding ImageName,
                        Converter={StaticResource imagenameToImageConverterObject}}"/>
                <StackPanel ... Grid.Row="1" Grid.Column="1">
                    <TextBlock ... Text="weight: "/>
                    <TextBox ... Text="{Binding Weight}"/>
                    <TextBlock ... Text=" kg"/>
                </StackPanel>
            </Grid>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

Although we are not giving details on the actual design of the template in the above XAML code, the result should be something similar to the one in Figure 3.

XIV.3. Using a data template in a ListBox

15. fejezet - LINQ (written by Gergely Kovásznai)

In software development, it is quite a frequent task to handle and display collections of data. Consequently, one often needs to apply operations to those collections, e.g., sorting the elements of a collection, filtering them, or linking two (or more) collections within a query (join).

In the previous sections, we have learned how to develop form-based applications, by using WPF. Let us image a form that displays a ListBox that contains data of employees! It would be cool to sort employees in the alphabetical order of their names, simply by clicking, or to display only those who work at the development division, or to filter out those on holiday.

One might find these kinds of tasks familiar from database management. In this section, we are introducing a .NET technology called LINQ (Language Integrated Query), which is capable to handle collections in an even more general way. In the followings, we are presenting the LINQ basics, and then the LINQ to Objects technology, which is about creating queries on collections of objects. In Section XVI we will learn about the LINQ to XML technology, and in Section XVII about LINQ to Entities.

But what is this all about? Let us suppose that we use a list of city names in our application:

```
List<string> cities = new List<string> {
    "Bologna", "Kiev", "Budapest",
    "Düsseldorf", "Madrid", "Sarajevo"
};
```

We would like to sort the city names by length. This can be done very easily, as follows:

```
IEnumerable<string> citiesToDisplay = cities.OrderBy(c => c.Length);
```

In order to increase pleasures, before sorting we would like to filter the city names that are at least 8 character long:

```
IEnumerable<string> citiesToDisplay = cities
        .Where(c => c.Length >= 8)
        .OrderBy(c => c.Length);
```

In the latter code, one can notice several interesting (and maybe novel) aspects. Probably the special methods (Where, OrderBy) are the easiest to notice; they are called extension methods. The parameters used by extension methods are given in the form of ...=>...; an expression of this kind is called a lambda expression. Lambda expressions will be detailed in Section XV.1, and extension methods in Section Hiba! A hivatkozási forrás nem található..

As a really quick example, we would like to present another form for the above query:

```
IEnumerable<string> citiesToDisplay = from c in cities
                                      where c.Length >= 8
                                      orderby c.Length
                                      select c;
```

Such a declarative (SQL-like) expression, being absolutely equivalent with the previous syntax, is completely legal above C# 3.0, and is called a query expression, as will be detailed in Section Hiba! A hivatkozási forrás nem található..

1. Lambda Expressions

Lambda expressions are very intuitive elements in C#, above 3.0. But first, before starting to use them, let us try to clarify what kind of language elements they are!

First, it is worth to recall our knowledge about delegates. As it is well known, they behave like method types, and can be used to write methods that receive "method pointers" as parameters. Consider the following example: we would like to write a FilterDates method, which filters elements in a list of dates. We do not fix the filtering rules within the method, but we will rather pass those "rules" as parameters to the method. This is why the special parameter filter is specified, in which our filtering method (or rather a pointer to that method) will be passed. The delegate is used to declare a signature (i.e., parameters and return type) for the filtering method in advance. In the following example, we declare an own delegate, which requires a DateTime parameter and bool return type.

```
delegate bool DateFilter(DateTime date);

List<DateTime> FilterDates(List<DateTime> dates, DateFilter filter)

{

    List<DateTime> filteredDates = new List<DateTime>();

    foreach (DateTime d in dates)

        if (filter(d))

            filteredDates.Add(d);

    return filteredDates;

}
```

It is very important that whatever method one would like to pass to FilterDates, its signature must be the same as the one of the delegate. For instance:

```
bool Filter21stCentury(DateTime date)

{

    return date.Year > 2000;

}

...

List<DateTime> dates = ...;

List<DateTime> datesToDisplay = FilterDates(dates, Filter21stCentury);
```

Suppose that we intend not to reference the Filter21stCentury method at any other location in the code. In the case of such "disposable" methods, it is quite inconvenient to define them in a nice and accurate way by assigning names to them. Above C# 2.0, it is also legal to pass anonymous methods as parameters. For example, the latter method call could be written as follows (without defining the filtering method separately):

```
List<DateTime> datesToDisplay = FilterDates(dates,

    delegate(DateTime d) { return d.Year > 2000; }

);
```

Lambda expressions, introduced in C# 3.0, can be considered as another, more intuitive syntax for anonymous methods. The above code can be written by using a lambda expression, as follows:

```
List<DateTime> datesToDisplay = FilterDates(dates, d => d.Year > 2000);
```

An important ingredient of a lambda expression is the => (arrow) operator. It is interesting that the compiler is able to infer the type of the parameter (d) from the context. One is also allowed to use more than one parameter, or even to write a lambda expression without any parameter. Moreover, the right-hand side of a lambda expression can even be a complete block (since it is actually the body of an anonymous method).

```
(a, b) => (a + b) / 2

() => 42

(x, y) => {
    if (x > y) return x;
    else return y;
}
```

In case of only one parameter, parentheses can be omitted. So can curly braces as well if the block on the right-hand side consists of one single „return …;", and even the return command can be omitted.

2. Extension Methods

In the previous section, we created a simple, but universal method (FilterDates) to filter given dates. As it can be expected, in .NET there are already existing solutions for such basic tasks such as filtering. When browsing the methods of the List class, one can spot the Where method, which performs exactly this kind of filtering, and can be used instead of our own FilterDates method:

```
IEnumerable<DateTime> datesToDisplay = dates.Where(d => d.Year > 2000);
```

When browsing the methods of dates (as a List), one can notice that it has a bunch of useful methods that are similar to Where, e.g., for searching, sorting, filtering, summing etc. One can also discover that those methods do not actually belong to the List class, but rather to the IEnumerable interface. However, in reality, IEnumerable defines one single method (GetEnumerator). Where do those useful methods come from, and where are they defined? Well, programmers have developed them outside of IEnumerable, inside other classes, and then – so to speak – they extended IEnumerable with those external methods.

Before getting to know existing extension methods other than Where, let us see in a nutshell how to write extension methods by ourselves! First, we need to create a public and static class, since extension methods are only allowed to be defined in such classes. The first parameter of an extension method must begin with the this keyword. The type of this first parameter specifies the type (class or interface) that is about to be extended with the given method. In the example below, the string class gets extended with a method that counts all the occurrences of a given character in a string.

```
public static class MyExtensionMethods
{
    public static int CountChar(this string s, char c)
    {
        int count = 0;
        foreach (char x in s)
            if (x == c) count++;
        return count;
    }
```

```
    ...
}
```

You can check that any string object has, from now on, a CountChar method, and you can call it! E.g., "almafa".CountChar('a')

In another example, let us extend List<string> objects with a method that converts all their elements to uppercase:

```
public static List<string> ToUpper(this List<string> stringList)
{
    List<string> newList = new List<string>();
    foreach (string s in stringList)
        newList.Add(s.ToUpper());
    return newList;
}
```

As mentioned before, we can use a large repertoire of already existing extension methods. We have already seen the Where and OrderBy methods, which extend the IEnumerable interface, but besides those two, there exist a lot of other extension methods that are useful in queries. Note that you need to import the System.Linq namespace (by using the using keyword) in order to use those extension methods in your source code! The reason for that is that those extension methods are defined in the Enumerable (static) class in the System.Linq namespace, according to exactly the same rules that we have introduced above.[1] In Section Hiba! A hivatkozási forrás nem található., we are going to give a detailed survey on the most important such extension methods, but first, let us introduce another way of using those methods, namely a special declarative syntax.

3. Comprehension Syntax

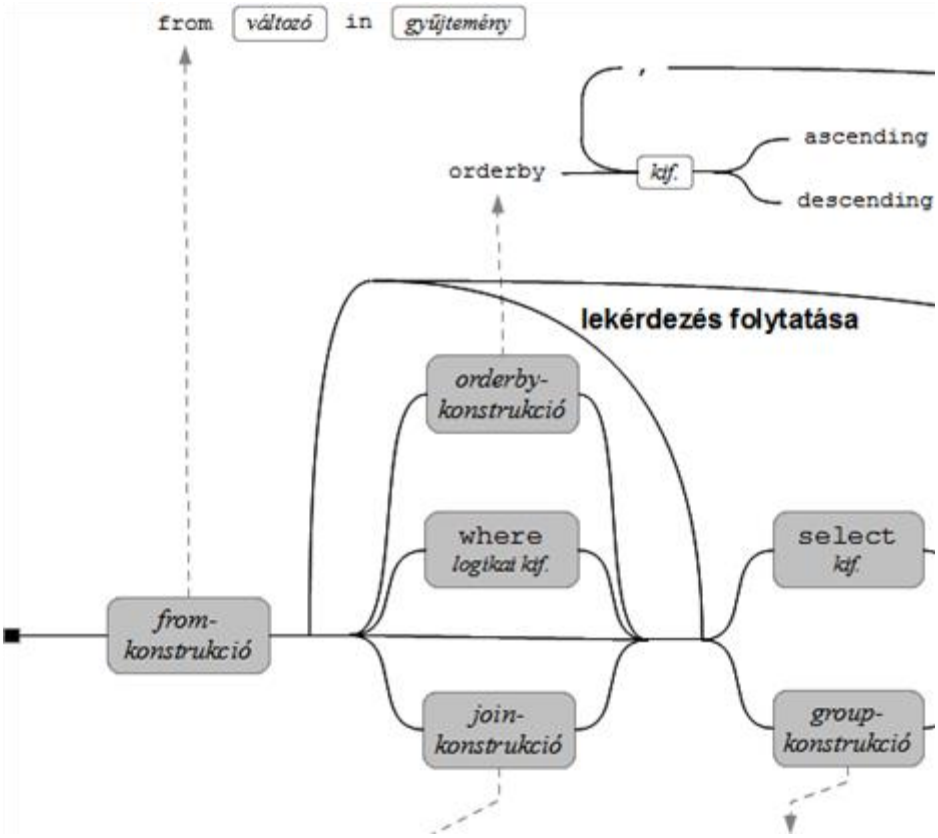
We have already met the comprehension syntax at the beginning of Section XV. This is an SQL-like syntax that realizes an easier-to-use, more intuitive access to the major extension methods in the System.Linq.Enumerable class (especially for a programmer who is familiar with SQL).

A comprehension query expression must always start with a from clause. A from clause declares an (iteration) variable, which one can use for traversing through the collection specified after the in keyword. The (almost) complete syntax (Albahari & Albahari, 2008) is depicted in Figure 1. A from clause can be followed by any number of orderby (Section Hiba! A hivatkozási forrás nem található.), where (Section Hiba! A hivatkozási forrás nem található.) or join (Section Hiba! A hivatkozási forrás nem található.) clauses. At the end of a query, one must write either a select (Section Hiba! A hivatkozási forrás nem található.) or a group clause (Section Hiba! A hivatkozási forrás nem található.). By using into (Section Hiba! A hivatkozási forrás nem található.), the result of a query can be saved into a variable, which can then be used to further continue the query with orderby/where/join clauses.

[1] For instance, the signature of the System.Linq.Enumerable.Where method:

```
public static IEnumerable<T> Where<T>(this IEnumerable<T> source, Func<T, bool> predicate);
```

As can be seen, the this keyword is standing before the first parameter, therefore this method extends the IEnumerable<T> interface. The type of the second parameter is a delegate (defined in the System namespace), which expects a parameter of arbitrary type (T) and returns bool.



XV.1. Comprehesion syntax (Albahari & Albahari, 2008)

In the subsequent sections, we will show examples how to use comprehension syntax.

4. Query Operators

In this section, we are walking through the list of the major query operators, divided into categories. But first, let us show an interesting aspect of them (and of LINQ queries), which is called deferred execution. Query operators, except for a few exceptions, are executed not at the moment when the query is constructed, but rather when the result of the query is iterated through. Let us see the following query as an example:

```
List<int> numbers = new List<int>() { 1, 2 };


IEnumerable<int> query = numbers.Select(n => n * 10);

numbers.Add(3);

foreach (int n in query)

    textBlock.Text += string.Format("{0} ", n);
```

The text in the textBlock will be „10 20 30”; i.e., the number 3 that we “sneaked” into the list is also appearing in the result of the query, since the query is only executed when being iterated through by foreach.

All the query operators we are going to introduce in the subsequent sections provide deferred execution, except for the ones in Section Hiba! A hivatkozási forrás nem található..

4.1. Filtering

Some operators are for filtering the elements of collections. In examples for comprehension syntax, we have already met the where keyword (which corresponds to the Where extension method), which is for returning

those elements of a collection that fulfill a given condition. Let us see one more example (in comprehension syntax)!

```
IEnumerable<Book> selBooks = from b in books
                             where b.ReleaseDate.Year >= 2000
                             select b;
```

Overall, one can use the following extension methods for filtering:

Filtering operators		
Where	T=>bool	Returns elements that fulfill a condition.
Distinct		Returns only distinct elements.
Take	int	Returns the first n elements.
TakeWhile	T=>bool	Returns the first elements until reaching an element that does not fulfill the condition anymore.
Skip	int	Skips the first n elements and returns the rest.
SkipWhile	T=>bool	Skips the first elements until reaching an element fulfilling the condition,

		and returns the rest.

The Take and Skip operators might be quite useful in real-world applications, since, by using them, one can split the result of a query into smaller chunks, e.g., for displaying only 20 elements at a time:

```
IEnumerable<Book> selBooks = books
    .Where(b => b.Title.Contains("world war"))
    .Take(20);
...
selBooks = books
    .Where(b => b.Title.Contains("world war"))
    .Skip(20)
    .Take(20);
```

The Distinct operator is very useful in any application that uses databases (Section XVII). Let us now show a rather unconventional example, which filters the distinct letters out of the characters of a string (as a collection), and, furthermore, sorts them in alphabetical order (c.f. the next section):

```
IEnumerable<char> letters = "Hello World"
    .Where(c => char.IsLetter(c))
    .Distinct()
    .OrderBy(c => char.ToUpper(c));
```

4.2. Ordering

Although we have already showed several examples for ordering, one can additionally specify the order of sorting and more than one levels of sorting. In comprehension syntax, one can use the orderby keyword, which is capable to realize even multi-level sorting, and the descending keyword, for sorting in a descending order. For instance, let us sort persons' data in alphabetical order (primarily by last name and secondarily by first name), and then sort further the resulting list in descending order by date of birth!

```
IEnumerable<Person> selPersons = from p in persons
                                 orderby p.FirstName, p.LastName,
                                          p.DateOfBirth descending
                                 select p;
```

Overall, the following extension methods can be used for ordering:

Ordering operators		
OrderBy, ThenBy	T=>TKey	Ascending order.
OrderByDescending, ThenByDescending	T=>TKey	Descending order.

Reverse	int	Reverse order.

As can be seen, the lambda expressions used as parameters must select a “key” (TKey) to sort by. The above example can also be written in another way:

```
IEnumerable<Person> selPersons = persons
    .OrderBy(p => p.FirstName)
    .ThenBy(p => p.LastName)
    .ThenByDescending(p => p.DateOfBirth);
```

4.3. Projection

In each example we have given for comprehension syntax so far, the select keyword stands at the end of queries. By using select, one can actually select the data to be included in the query result (as a collection). In the above examples, this kind of expression appeared only in the form of „select x", where x was a variable that occured in the query. However, right after select one can use an arbitrary expression; this expression might contain x (and, of course, it usually does). Let us see a few examples:

```
IEnumerable<string> firstNames = from p in persons
                                 select p.FirstName;
IEnumerable<string> personNames = from p in persons
                                  select p.FirstName + " " + p.LastName;
IEnumerable<int> personAges = from p in persons
                              select (DateTime.MinValue +
                                      DateTime.Now.Subtract(p.DateOfBirth)
                              ).Year;
```

In the latter example, a particularly complex expression takes place right after select (in order to compute the age of a given person).

In each of the above examples, only one data stands right after select. What to do if one would like to select more than one data and return them all together? This can only be done by wrapping the selected data in an object. The next example is about selecting the identifier and the name of a given person, therefore they are wrapped in an instance of a class (PersonSimple), defined by us elsewhere.

```
IEnumerable<PersonSimple> selPersons = from p in persons
                                       select new PersonSimple
                                       {
                                           Id = p.Id,
                                           Name = p.FirstName + " " + p.LastName
                                       };
```

Let us image that, in our application, there exist many various queries related to the Person class! In one of them persons’ identifiers and names are selected, in another one names and ages (such as in the example below), in a third one identifiers, jobs, and dates of birth, and so on. It is very complicated and tedious to define a separate

"wrapper" class for each kind of selection. Anonymous classes in C# provide convenient solution for this problem.

The compiler, with respect to an anonymous instantiation written in the source code, automatically defines an anonymous class. If two instantiations contain properties of the same type and of the same name (and in the same order), then the same anonymous class will be instantiated.

```
var selPersons = from p in persons
                 select new
                 {
                     Name = p.FirstName + " " + p.LastName,
                     Age = (DateTime.MinValue + DateTime.Now.Subtract(p.DateOfBirth)
                           ).Year
                 };
```

Since, in the above example, the collection resulted by the query consists of instances of an anonymous class (defined by the compiler), one cannot determine the explicit type of the selPersons variable (and, therefore, one does not know what class name to write right after IEnumerable). The var keyword, in C# 3.0, was invented exactly for such cases.

The type of a variable declared by the var keyword is defined precisely, even if developers do not always realize this fact. This type is declared by the compiler, and is the same as the one of the right-hand-side expression used for initializing the variable. For example, the type of x will be double in the following initialization:

```
var x = 15 / 2.3;
```

Besides satisfying the "laziness" of developers, the usage of var is unavoidable when using anonymous classes. We have already seen such an example above, but another good example could be a loop that traverses a collection (selPersons) containing instances of an anonymous class:

```
foreach (var p in selPersons) { ... }
```

One can altogether use two extension methods for projection:

Projecting operators		
Select	T=>TResult	Projects each element to a TResult object.
SelectMany	T=>IEnumerable<TResult>	Projects each element to a collection of TResult objects.

Due to space constraints, we are not going to dig deep into the projecting operators (especially not into SelectMany); on the other hand, we recommend related literature listed in the References section. Among the

various possibilities detailed in literature, we would like to mention only one related to select, namely that of nested subqueries. The point is the following: an expression right after select is allowed to include even another query (which is also allowed to include further ones). In the next example, a list of (system) directories is retrieved, and (hidden) files within each directory as well:

```
System.IO.DirectoryInfo[] dirs = ...;

var query = from d in dirs
            where (d.Attributes & System.IO.FileAttributes.System) == 0
            select new
            {
                DirectoryName = d.FullName,
                Created = d.CreationTime,
                Files = from f in d.GetFiles()
                        where (f.Attributes & System.IO.FileAttributes.Hidden) == 0
                        select new
                        {
                            Filename = f.Name,
                            Length = f.Length
                        }
            };
```

Note that the result of the query will be a collection of anonymous class objects, each of which describes a directory. It is an especially interesting feature that this anonymous class has a Files property, which is a collection of instances of another anonymous class.

4.4. Grouping

In certain queries, one might want to split a collection into smaller chunks, by considering a certain criterion. In comprehension syntax, one can use the keywords group...by for this purpose. For instance, let us group the employees of a company by sections!

```
List<Person> persons;

var personGroups = from p in persons
                   group p.FirstName + " " + p.LastName by p.Section;

foreach (var pGroup in personGroups)
{
    Console.WriteLine("Section: {0}", pGroup.Key);
    foreach (var p in pGroup)
        Console.WriteLine("\t{0}", p);
}
```

As can be seen, in the result of the query, the keys of distinct groups can be accessed via the Key property; in the above example, the key is the name of a section.

Of course, arbitrary expression can be used between the group and by keywords (e.g., instantiation of an anonymous class); in this regard, group...by follows exactly the same rules as select does.[2] Let us show an example about grouping files by extension!

```
System.IO.FileInfo[] files = ...;

var query = from f in files
            group new
            {
                Name = f.Name.ToUpper(),
                Date = f.CreationTime
            } by f.Extension;
```

The sole grouping operator is implemented by the following extension method:

Grouping operator		
GroupBy	T=>TKey [,T=>TResult]	Groups the elements (and transforms them into TResult objects).

Notice that the second parameter of the GroupBy method, which is for customizing projection, is optional.

It is also worth to mention the usage of the into keyword, which is for accessing a grouping via an identifier in a query.

The into keyword is for "saving" a projection, i.e., the resulting collection of a projection can be accessed via the identifier given right after into. The projection can be either a select or a group...by clause.

In the following example, groups of such files are returned whose extensions do not exceed 10 characters, and, furthermore, the resulting groups are sorted by the number of elements, in ascending order.

```
var query = from f in files
            group new
            {
                Name = f.Name.ToUpper(),
                Date = f.CreationTime
            } by f.Extension
            into g
            where g.Key.Length <= 10
```

[2] In fact, group...by is also a projecting operator (such as select). However, the resulting collection is not "flat", but is rather two-level.

```
        orderby g.Count()

        select g;
```

4.5. Join

In applications that use databases (Section XVII), it is essential to join tables. There exist several kinds of joins, e.g., inner join, left join, right join, cross join etc. Besides the tools that have already been introduced in the previous sections, LINQ provides an additional opportunity for connecting collections with each other, and this is available through the join...on...equals triple keyword in the comprehension syntax. Let us see an example that joins a collection of persons (persons) and a collection of travels (travels), and generates a list about who travelled where:

```
var query = from p in persons

        join t in travels on p.Id equals t.PersonId

        select string.Format("{0} {1} travelled to {2}",

                                        p.FirstName, p.LastName, t.Destination);
```

This query returns a collection of such strings:

Mary Butcher travelled to New Zeland

Mary Butcher travelled to Prague

Victor Hugo travelled to Naples

Sándor Kovács travelled to Zalaegerszeg

The above join can be considered typical, since each element of both collections has an identifier that is checked to be equal to each other. By the way, the above join is an inner join, meaning that such persons who did not travel anywhere will not occur in the output.

It is also possible to realize joins on multiple keys, as follows:

```
var query = from x in seqX

  join y in seqY on new { K1 = x.Prop1, K2 = x.Prop2 }

        equals new { K1 = y.Prop3, K2 = y.Prop4 }
```

Here we exploit the fact that the same properties (having the same names and types) are used in both instantiations, therefore the compiler will instantiate the same anonymous class. Thus, equality checking will work as expected.

Comprehension syntax supports even left joins, meaning – in terms of the previous example – that every person will occur in the output, even those ones who have not travelled anywhere. In order to realize this, we need to write an into clause right after the join (c.f. the previous section). The previous example can be altered accordingly:

```
var query = from p in persons

        join t in travels on p.Id equals t.PersonId

        into personTravels

        select new

        {

            PersonName = string.Format("{0} {1}", p.FirstName, p.LastName),
```

```
                    Travels = personTravels
                };
foreach (var pt in query)
{
    Console.WriteLine(pt.PersonName);
    if (pt.Travels.Count() == 0)
        Console.WriteLine(" did not travel anywhere");
    else
    {
        Console.WriteLine(" travelled to:");
            foreach (var t in pt.Travels)
                Console.WriteLine("\t{0}", t.Destination);
    }
}
```

In the above source code, we traverse the query result by a loop, and check each person whether he or she has travelled to at least one place.

All the joins that have been introduced above are implemented by the following two extension methods:

Join operators		
Join	IEnumerable<T2>,T=>TKey,T2=>TKey,(T,T2)=>TResult	Joins a collection of elements of type T and a collection of elements of type T2, and returns a collection of elements of type TResult.
GroupJoin	IEnumerable<T2>,T=>TKey,T2=>TKey,(T,IEnumerable<T2>)=>TResult	Same as above, but the result is further grouped by the elements

		of type T.

Finally, let us show an example that performs multiple joins![3] Let us first use comprehension syntax: besides joining persons and travels, we are also about connecting a collection of travel expenses (expenses) to the query, in order to list who spent how much and where.

```
var query = from p in persons
            join t in travels on p.Id equals t.PersonId
            join e in expenses on t.Id equals e.TravelId
            select new
            {
                PersonName = string.Format("{0} {1}", p.FirstName, p.LastName),
                Amount = e.Amount,
                Place = t.Destination
            };
```

By directly using extension methods, the same result could be achieved, as follows:

```
var query = persons
    .Join(travels, p => p.Id, t => t.PersonId, (p, t) => new
    {
        Person = p,
        Travel = t
    })
    .Join(expenses, pt => pt.Travel.Id, e => e.TravelId, (pt, e) => new
    {
        PersonName = string.Format("{0} {1}",
                                   pt.Person.FirstName, pt.Person.LastName),
        Amount = e.Amount,
        Place = pt.Travel.Destination
    });
```

This example illustrates pretty well how much easier and more intuitive it is to use comprehension syntax, in most cases. On the other hand, calling extension methods directly gives more flexibility.

4.6. Nondeferred Operators

As mentioned before, all the operators introduced in the previous sections provide deferred execution. In a real-world application, eventually it is necessary to “back up” some part of the current query result, such as the whole result or only a single element.

[3] Comprehension syntax supports arbitrary many joins, one after the other.

Conversion. The whole result of a query can be “backed up” into a collection. I.e., whatever will happen with the source of the query in the future, the current query result will be preserved in the exported e.g. array or list. Let us see an example:

```
List<Car> selCars = (from c in cars
                     where c.Manufacturer == "Suzuki"
                     orderby c.ManufactureDate descending
                     select c
                    ).ToList();
```

The following extension methods can be used for exporting/converting:

Conversion operators		
ToArray		Converts an IEnumerable<T> collection into a T[] array.
ToList		Converts an IEnumerable<T> collection into a List<T> list.
ToDictionary	T=>TKey	Converts an IEnumerable<T> collection into a Dictionary<TKey,T> dictionary.

As can be seen, when converting into a Dictionary (i.e., hash table), one must specify what to consider as the key of elements.

Element. If one does not want to export a whole collection but rather a single element, then the following extension methods can be used:

Element operators		
First	[T=>bool]	Returns the collection’s first element (that fulfills the optional condition).
Last	[T=>bool]	Returns the last element (that fulfills the optional condition).
ElementAt	int	Returns the

		element at the given index.

Aggregate. It is necessary to "aggregate" a collection in order to extract certain data, such as the average of elements or even the number of elements. The following aggregating extension methods can be used:

Aggregating operators		
Count	[T=>bool]	Returns the number of elements (that fulfill the optional condition).
Min, Max	[T=>TResult]	Returns the minimal resp. maximal element. Projecting expression can also be specified.
Sum, Average	[T=>TResult]	Returns the sum resp. average of elements. Projecting expression can also be specified.

It is worth to understand the usefulness of the optional T=>TResult parameter, which belongs to certain aggregating operators. Let us assume that our query returns car objects and we would like to calculate their average price. One way to do this:

```
double average = (from c in cars
                  where c.ManufactureDate.Year >= 2005
                  select c
                 ).Average(c => c.Price);
```

Quantifiers. For certain tasks, it might be necessary to check whether (all) the elements of a collection fulfill a given condition. This can be done by using the following extension methods:

Quantifier operators		
Contains	T	Checks whether a given element

		can be found in the collection.
Any	[T=>bool]	Is there any element (that fulfills the optional condition)?
All	T=>bool	Do all the elements fulfill the condition?

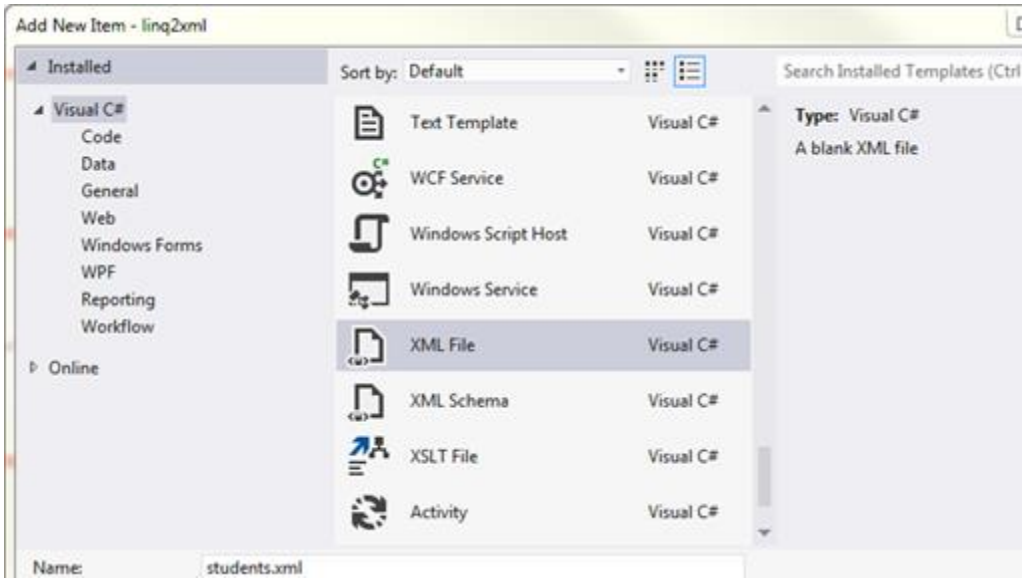
16. fejezet - LINQ to XML (written by Gergely Kovásznai)

In small-scale software, it is common to store source data in files. The same holds for applications that have only limited access to network or external databases (e.g., in Silverlight applications).

Since, in WPF applications, one usually deals with textual or numerical data, it is worth to use text files to store data in. This is nice, but in what format should one store such objects? Do not forget that one must store each property (with its name and value) of individual objects, and, furthermore, some of those properties might be compound ones, i.e., their values might be objects, therefore, again, one must store each property of those objects as well, among which there might be compound ones again, and so on. Long story short, our data should be stored in a hierarchical format.

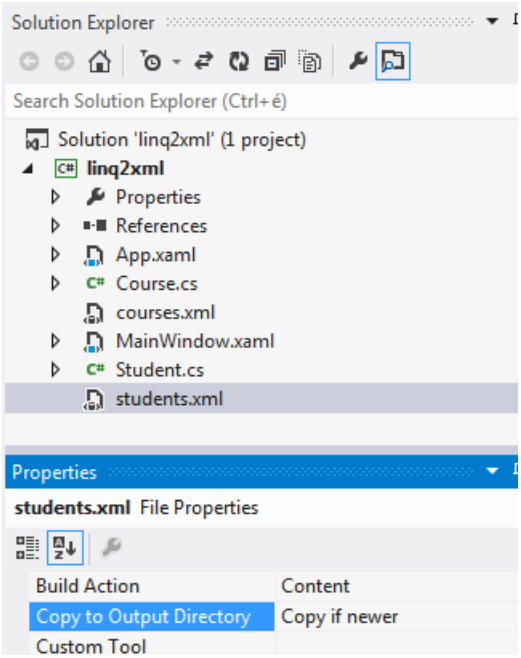
Before starting to invent some own file format, it is worth to check out already existing solutions. It is worth to look around and see what kinds of standards developers can access. For our goals, the XML format is a perfect choice. Furthermore, we as developers can profit from the various tools that support the parsing and handling of XML data. So does the .NET framework. Above .NET 3.5, one can even execute LINQ queries on XML data, too.

Although any text editor can create XML files, it is worth, in our case, to exploit Visual Studio's (Section XVIII.1) support for XML. One can add XML files as new items to a project (c.f. Section XVIII.1.1), as can be seen in Figure 1.



XVI.1. Creating an XML file in Visual Studio

It is worth to copy the XML file that we have just created next to (the exe file of) our application; this we can either do manually or make Visual Studio do it (c.f. Figure 2): among the properties of the XML file (in the Properties window detailed in Section XVIII.1.4), set Build Action to Content, and Copy to Output Directory to Copy if newer. After doing so, whenever we make changes to the XML file, Visual Studio will update the one in the directory of our application, too.



XVI.2. Setting the properties of an XML file in Visual Studio

Let us start to edit the content of our XML file! We have total freedom in choosing the content; it must only meet the syntactical rules of the XML standard. In the next example, we would like to store students' data in an XML file. Let us use a student XML element to represent each student! Let this element have firstName, lastName and sex attributes, which represent the first name, the last name, and the gender of a student, respectively. Let us assume that even the date of birth of each student has to be stored; since date is compound data, we represent it as, for the sake of example, a separate dateOfBirth XML element, which has year, month, and day attributes. It is important that an XML file must contain exactly one root element; we will use the students XML element for this purpose.

```
<?xml version="1.0" encoding="utf-8" ?>

<students>

    <student firstName="Rita" lastName="Poratzki" sex="female">

        <dateOfBirth year="1992" month="07" day="12"/>

    </student>

    <student firstName="Laura" lastName="Ashley" sex="female">

        <dateOfBirth year="1993" month="01" day="09"/>

    </student>

    <student firstName="Steven" lastName="Drabant" sex="male">

        <dateOfBirth year="1991" month="02" day="27"/>

    </student>

    ...

<students>
```

Loading and parsing XML files is supported by several .NET tools. In the followings, we will introduce the toolkit of LINQ to XML, which supports, besides loading and parsing, LINQ queries on XML data. Related classes can be found in the System.Xml.Linq namespace.

1. Loading XML files

When dealing with XML data, the XElement class plays a key role. First, let us introduce two of its static methods, which are for parsing text in XML format! The XElement.Parse method expects a string as parameter, in which one can pass arbitrary text in XML format and make the framework parse it. It is worth to use the XElement.Load method if the text being parsed is located in a file; this method expects a filename as parameter.[1] Both Parse and Load return an instance of the XElement class.

XElement		
Name		Name of the XML element. E.g., string „dateOfBirth” in the case of <dateOfBirth ...>.
Value		If the XML element has a start-tag and an end-tag, and there is some text between them, then this text is returned in this property. E.g., string „Manchester” in the case of <city>Manchester</city>.
Attribute(...)Attributes([...])	XName	Returns the element’s attribute(s) that has/have the given name.
Element(...)Elements([...])	XName	Return the direct descendant element(s) that has/have the given name.
Descendants([...])	XName	Returns all the (direct or indirect) descendant elements.
Parse(...)	string	Parses (the root element of) the XML data passed in the string.
Load(...)	string	Load a file and parses its root element.

The methods that expect a parameter of type XName accept a simple string as well.[2]

All the above methods return XElement objects, except for the Attribute and Attributes methods, which return XAttribute objects. An XML attribute is basically a „name=value” pair, therefore it is sufficient for us to get to know the Name and Value properties of the XAttribute class.

XAttribute		
Name		Name of the

[1] The Load method offers several signatures, which can even read from streams.
[2] The reason for that is an implicit cast operator of the XName class for casting to string.

		XML attribute. E.g., string “year” in the case of year=”1993” .
Value		Value of the XML attribute. E.g., string “1993” in the case of year=”1993” .

In the first line of the next example, we can see how simple it is to load and parse an XML file, in a single line. In order to understand properly that the parsing result is a hierarchy of objects stored in the xStudents variable, let us check the subsequent two commands! The first command accesses the first student XML element, and then reads the value of its firstName attribute (supposed to be equal to „Rita”, based on the previous example). The second command accesses the third student element (i.e., the one at index 2), and then its dateOfBirth element, whose year attribute is accessed, whose value is returned finally; supposed to be equal to „1991”.

```
XElement xStudents = XElement.Load("students.xml");

string firstName = xStudents.Element("student").Attribute("firstName").Value;

string year = xStudents.Elements("student").ElementAt(2)
                        .Element("dateOfBirth").Attribute("year").Value;
```

As this example shows, even numerical data (e.g. 1991) are treated as text; therefore, both the XElement.Value and XAttribute.Value properties are of type string. Developers have to take care of converting texts to numbers; e.g., the last command in the above example could be rewritten as follows: int year = int.Parse(xStudents...Value);

2. Queries

All the query operators that we introduced in Section Hiba! A hivatkozási forrás nem található. can be applied to collections of XML elements and attributes in the same way as to any other collections. The only speciality is the way in which XML elements and attributes are accessed, and, furthermore, the necessity of data conversion. Let us see an instant example! We are going to load the students.xml file of the previous section, and are performing a query on the loaded data (i.e., on XElements). For instance, female students can be filtered as follows:

```
XElement xStudents = XElement.Load("students.xml");

var femaleStudents = from s in xStudents.Elements("student")
                     where s.Attribute("sex").Value == "female"
                     select new
                     {
                         FirstName = s.Attribute("firstName").Value,
                         LastName = s.Attribute("lastName").Value,
                         DateOfBirth = new DateTime(
```

```
                    int.Parse(s.Element("dateOfBirth").Attribute("year").Value),
                    int.Parse(s.Element("dateOfBirth").Attribute("month").Value),
                    int.Parse(s.Element("dateOfBirth").Attribute("day").Value)
                )
            };
```

As can be seen, the query returns instances of an anonymous class. Instead, it would be better (more advisable and safer) to implement an own class for representing students.

As we have already claimed, all the LINQ query operators (e.g., OrderBy, GroupBy, Count etc.) can be used in the usual way on XML data as well. The next example shows how to apply one of them, the Join operator. Let us assume that, besides students' data, we also want to include courses' data in our application! For the sake of example, courses' data will be loaded from another XML file (courses.xml):

```
<?xml version="1.0" encoding="utf-8" ?>
<courses>
    <course id="bd64f" name="Debugging" credits="3"/>
    <course id="z6df3" name="Programming Technologies" credits="3"/>
    <course id="a87d4" name="Visual Computing" credits="4"/>
    <course id="10i7e" name="Genetic Algorithms" credits="2"/>
</courses>
```

It is important that each course has an identifier (id attribute); we are going to use this id to access the course and to realize joins. Let us extend the content of the students.xml file with data that tells us which courses a given student is enrolled for!

```
<?xml version="1.0" encoding="utf-8" ?>
<students>
    <student firstName="Rita" lastName="Poratzki" sex="female">
        <dateOfBirth year="1992" month="07" day="12"/>
        <courses>
            <course id="bd64f" grade="4"/>
            <course id="a87d4" grade="3"/>
        </courses>
    </student>
    <student firstName="Laura" lastName="Ashley" sex="female">
        <dateOfBirth year="1993" month="01" day="09"/>
        <courses>
            <course id="a87d4" grade="5"/>
            <course id="z6df3" grade="2"/>
```

```
        </courses>
    </student>
    <student firstName="Steven" lastName="Drabant" sex="male">
        <dateOfBirth year="1991" month="02" day="27"/>
        <courses>
            <course id="10i7e" grade="3"/>
        </courses>
    </student>
    ...
</students>
```

As can be seen, a course list appears among the data of each student, where each course is represented by its identifier (id) and the grade (grade) that the given student got on the given course.

Our query looks like as follows:

```
var studentsWithCourses = from s in xStudents.Elements("student")
                          select new Student
                          {
                              FirstName = ...,
                              LastName = ...,
                              Sex = ...,
                              DateOfBirth = ...,
                              Courses = from c1 in s.Element("courses").Elements("course")
                                        join c2 in xCourses.Elements("course")
                                        on c1.Attribute("id").Value equals c2.Attribute("id").Value
                                        select new CourseForStudent
                                        {
                                            Name = c2.Attribute("name").Value,
                                            Grade = int.Parse(c1.Attribute("grade").Value)
                                        }
                          };
```

As can be seen, we have defined for each student a Courses collection, in which Name-Grade pairs are stored, where Name is the name of a course and Grade is the grade that the student got on the course. The collection is filled by a join on the student's course list and the global course list (xCourses).

With the query result (studentsWithCourses) in hand, one can build the GUI in Figure 3.

XVI.3. LINQ to XML – Sample application

3. XML Serialization

In this section, we would like briefly to sketch another XML technology, which is not related to LINQ, and is called XML serialization. Serializing objects is, in general, for saving objects into files, and later to load them back (deserialization).[3] XML serialization is a special kind of serialization, which saves objects in XML format. By using the built-in XML serializer in .NET, one can easily save objects (e.g., a list of students) into an XML file, and then load them back easily.

We have seen earlier that the properties of classes (e.g. Student) can easily and directly be represented by XML elements (e.g. DateOfBirth) and XML attributes (e.g. FirstName). This kind of work can be automated as well. The XmlSerializer class in the System.Xml.Serialization namespace is able to serialize a (serializable) object into XML, by automatically inferencing the names of necessary XML elements and attributes.

For the sake of example, let us serialize the studentsWithCourses collection from the previous section! This collection was the result of a LINQ query, and is of type IEnumerable<Student>. Unfortunately, the IEnumerable interface is not serializable, as opposed to its own concrete implementations, such as List. Therefore, if we would like to serialize the results of our LINQ queries, they are worth to be converted to lists by using the ToList() conversion operator (Section Hiba! A hivatkozási forrás nem található.).

```
var studentsWithCourses = (from s in xStudents.Elements("student")
                           select new Student
                           {
                               FirstName = ...,
                               LastName = ...,
                               Sex = ...,
                               DateOfBirth = ...,
                               Courses = (from c1 in s.Element("courses").Elements("course")
```

[3] Precisely speaking, serialization is for mapping an object to a byte stream.

```
                    ...
                    select new CourseForStudent
                    {
                        Name = ...,
                        Grade = ...
                    }).ToList()
            }).ToList();
```

Note that not only the outer query is converted to a list, but also the inner one (which realizes a join)! Since now all the building blocks of our studentsWithCourses collection are serializable (simple types and DateTime are serializable by default), we can serialize the collection. For this, we need to instantiate XmlSerializer, for which we need to specify the type of the object being serialized (List<Student>). For serialization, one must specify a stream (a file stream in the current case), to which the system writes the serialization result. One can (very easily) write source code for serialization as follows:

```
XmlSerializer serializer = new XmlSerializer(typeof(List<Student>));

StreamWriter fileWriter = new StreamWriter("query.xml");

serializer.Serialize(fileWriter, studentsWithCourses);

fileWriter.Close();
```

The resulting query.xml will look as follows:

```
<?xml version="1.0" encoding="utf-8"?>

<ArrayOfStudent xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Student>
    <FirstName>Rita</FirstName>
    <LastName>Poratzki</LastName>
    <Sex>female</Sex>
    <DateOfBirth>1992-07-12T00:00:00</DateOfBirth>
    <Courses>
      <CourseForStudent>
        <Name>Debugging</Name>
        <Grade>4</Grade>
      </CourseForStudent>
      <CourseForStudent>
        <Name>Visual Computing</Name>
        <Grade>3</Grade>
      </CourseForStudent>
```

```
        </Courses>
    </Student>
    ...
</ArrayOfStudent>
```

As can be seen, the resulting XML elements have the same names as the ones of our classes and of their properties. The value of the DateOfBirth property has a strange format; we have, unfortunately, no influence on this, since DateTime is not our own class (therefore it is serialized in a predetermined way). The following questions arise in connection with own classes. Is there any opportunity for serializing a property not as an XML element, but rather as an XML attribute? Can the name of an XML element/attribute differ from the name of the corresponding class/property? Can one rename the ArrayOfStudent element, corresponding to a List<Student> object in the example?

The answer for all the above questions is "yes". In order to solve to first two problems, the System.Xml.Serialization namespace offers .NET attributes. (Do not get confused between .NET attributes and XML attributes!)

.NET attributes make it possible to attach metadata to types (e.g., classes), properties, and methods. Attributes (even several ones) must be written between brackets, and must be located right before the definition of the given type, property, or method. For example:

```
[Serializable]
public class MyClass
{
    [Obsolete]
    [Description("This is a method for ...")]
    public void MyMethod(...) { ... }
    [Category("Numeric")]
  public int MyProp { set; get; }
}
```

The above attributes, which exist in the .NET framework, are shown here as examples. The Serializable attribute declares a given class serializable; Obsolete marks a method obsolete; Description attaches a description to a method/property. Category makes it possible to divide the set of the properties of a class into categories; we can see them arranged in those categories in Visual Studio's Properties Window (Section XVIII.1.4).

We have shown only a few examples for using .NET attributes; c.f. related literature.

In connection with XML serialization, the following .NET attributes are worth to mention:

1. XmlType. To specify the major settings for serializing a class; e.g., how to name the XML element that corresponds to the class.
2. XmlElement, XmlAttribute. To specify whether a property should be serialized as an XML element or as an XML attribute (and to specify related settings).

For instance, these attributes could be used in our Student class as follows:

```
  public enum Sex { female, male }
  [XmlType("StudentWithCourses")]
```

```
public class Student
{
    [XmlAttribute]
    public string FirstName { get; set; }
    [XmlAttribute]
    public string LastName { get; set; }
    [XmlAttribute("Gender")]
    public Sex Sex { get; set; }
    [XmlElement("BirthDate")]
    public DateTime DateOfBirth { get; set; }
    public List<CourseForStudent> Courses { set; get; }
}
```

As can be seen, each Student object will be serialized as a StudentWithCourses XML element. The FirstName, LastName, and Sex properties will become XML attributes, instead of XML elements, and, furthermore, the latter one will be renamed to Gender. For the sake of example, we will serialize the DateOfBirth property as a BirthDate XML element, from now on.

If we have serialized our studentsWithCourses list by using the above settings, then we can notice that the root element is called ArrayOfStudentWithCourses. Even the name of this XML element can be customized in the instantiation of XmlSerializer, as follows:

```
XmlSerializer serializer = new XmlSerializer(typeof(List<Student>),
                                                new XmlRootAttribute("AllStudents"));
```

The result of serializing the list:

```
<AllStudents ...>
    <StudentWithCourses FirstName="Rita" LastName="Poratzki" Gender="female">
        <BirthDate>1992-07-12T00:00:00</BirthDate>
        <Courses>
            <CourseForStudent Name="Debugging" Grade="4" />
            <CourseForStudent Name="Visual Computing" Grade="3" />
        </Courses>
    </StudentWithCourses>
    ...
</AllStudents>
```

Suppose that you send the query.xml file, being generated right now, to someone, and he/she would like to open it in his/her application! One can open an XML file and perform queries on it as introduced in Sections XVI.1 and Hiba! A hivatkozási forrás nem található..

There is another possibility called deserialization. It is about reading back the object that has been serialized, in a direct way, by using the XmlSerializer.Deserialize method. Suppose that we send the query.xml file to someone, who would like to open it in an own application, and then to access its content as a list of students (for which he/she will need our Student and CourseForStudent classes as well). All these things can be done in a few lines of code:

```
XmlSerializer serializer = new XmlSerializer(typeof(List<Student>),
                                                new XmlRootAttribute("AllStudents"));

StreamReader fileReader = new StreamReader("query.xml");

Students = serializer.Deserialize(fileReader) as List<Student>;

fileReader.Close();
```

17. fejezet - LINQ to Entities (written by Gergely Kovásznai)

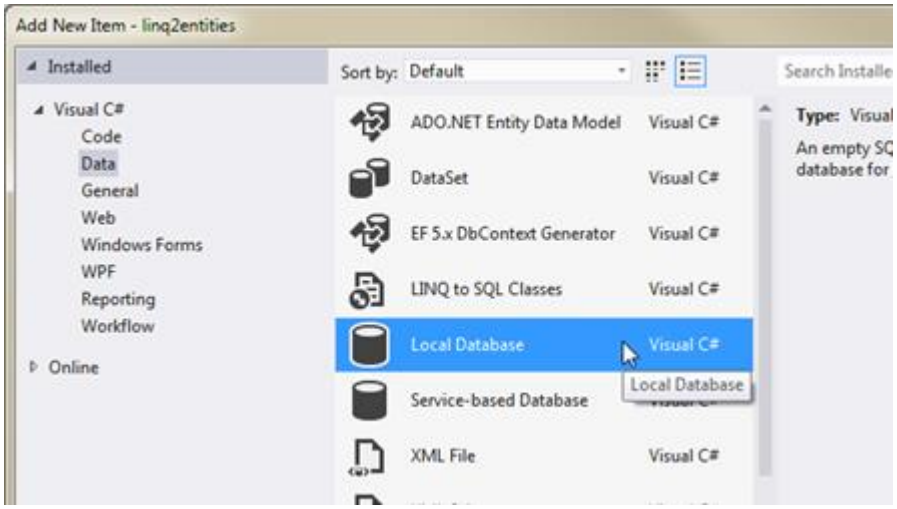
In the previous section, we introduced how to perform queries on data represented in XML. A demand arises for doing the same on SQL databases, either local or remote ones. In contrast to XML files, SQL databases offer a more sophisticated, robust and trusted alternative. State-of-the-art database management systems, such as MS-SQL, Oracle Database, PostgreSQL, or MySQL, provide numerous services, e.g., maintaining large amount of data in an effective, standardized and optimized way, and supporting transaction processing. From our perspective, it is indeed very useful to create and maintain databases easily (by using the appropriate GUI tools), and to access databases, to perform queries, and to manipulate (insert, delete, or modify) data in C# easily. In this section, we only give a practical introduction to this deep topic, and hope that developers, after picking up those introductory tricks, will start their journey along this direction, which offers lots of novelties and beneficial knowledge. For this sake, we are about to show a rapid development process by using modern .NET technologies such as MS-SQL and LINQ to Entities.

1. MS-SQL and Server Explorer

Data in SQL databases are stored in tables, tables consist of columns, each column has some kind of type (e.g., integer, string, date etc.), and, furthermore, certain (groups of) columns may have special roles (e.g., primary key, foreign key), which can be used to realize connections between certain tables. Imagine, for instance, a database in which we store data about chocolats, by using columns such as an identifier (integer), a name (string), and (the identifier of) a manufacturer (integer)! The latter column is a foreign key refers to another table in which manufacturers are stored, by using appropriate columns. For the sake of example, we are going to create the database by using Visual Studio's (Section XVIII.1) corresponding tool. First, add a so-called local database to your project; this database is actually an MS-SQL CE database!

MS-SQL has a slimmed-down version for desktop and mobile applications, called MS-SQL Compact Edition (CE). The point is that the MS-SQL CE engine gets embedded directly in our application. Visual Studio supports MS-SQL CE by default, since it is essential to provide some basic developing tool for creating and managing simple, local databases.

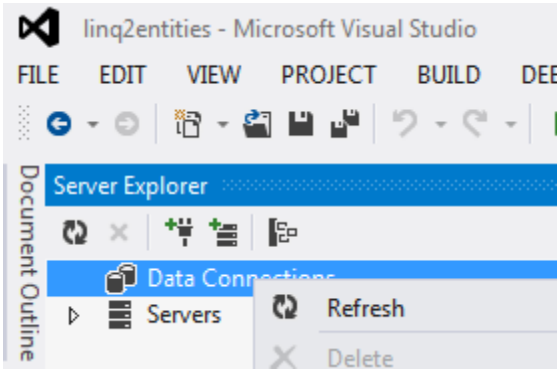
When adding a new item, choose „Local Database" in the „Data" category, as can be seen in Figure 1. Do not forget to give a name to the file that contains the database. As can be seen, the file format for MS-SQL CE is SDF.[1]



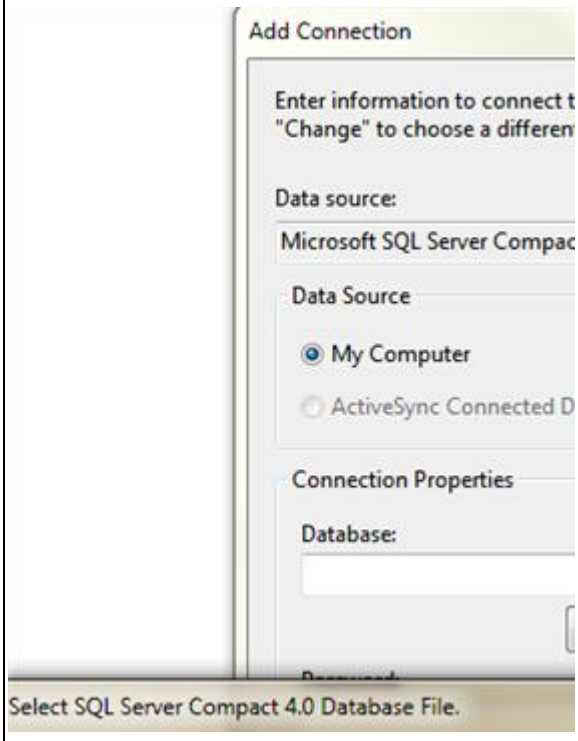
XVII.1. Creating an MS-SQL CE database in Visual Studio

[1] The maximum file size for SDF is 4 GB.

Next, create tables in your (empty) database! For this, it is worth to use a built-in tool in Visual Studio, the Server Explorer.[2] As can be seen in Figures 2 and 3, one has to click on Server Explorer's „Data Connections" item, and then to select „Add Connection". Notice in the window pops up that the tool expects an MS-SQL CE database by default as „Data Source". Click the „Browse" button in order to browse for your SDF file.



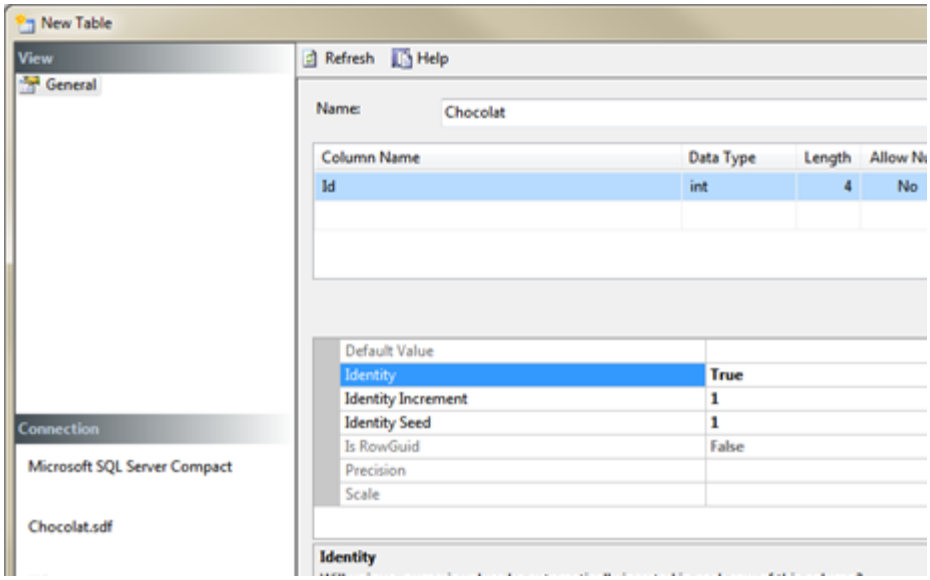
XVII.2. Adding a connection



XVII.3. Setting connection parameters

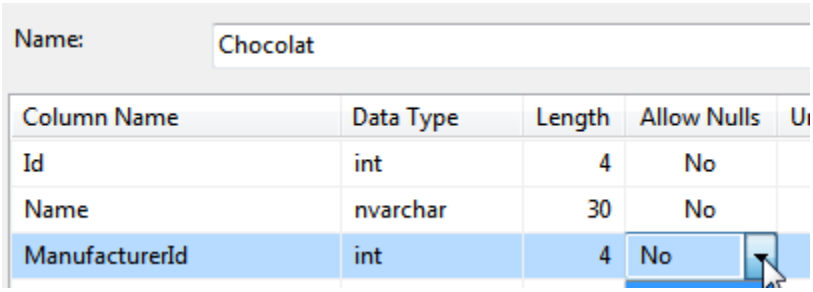
How to create a table in our database? In Server Explorer, right click on the „Tables" item of our database, and then select the „Create Table" menu item. After naming the table (Chocolat), fill out the form for columns. First, add a numerical primary key (Id) to your table, as can be seen in Figure 4. Pay attention to set the type („Data Type") of the key to integer (int), to make it primary key („Primary Key"), and to declare the column as an identifier („Identity").[3]

[2] If you do not see Server Explorer on the interface of Visual Studio (usually on the left), then you can open it on the „View" menu.
[3] As can be seen in Figure 4, it is practical to leave „Identity Increment" equal to 1. By doing so, one instructs the system to increment the value of the identifier by 1 whenever inserting a new record.



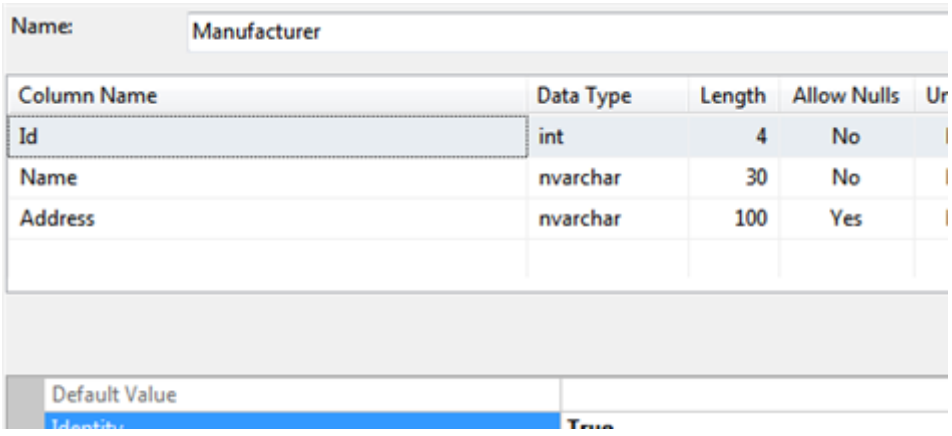
XVII.4. Adding a table to a database I

Add more columns as seen in Figure 5: let the Name column be string (30-character-long nvarchar), ManufacturerId a foreign key (int), and both filled compulsorily („Allows Nulls").



XVII.5. Adding a table to a database II

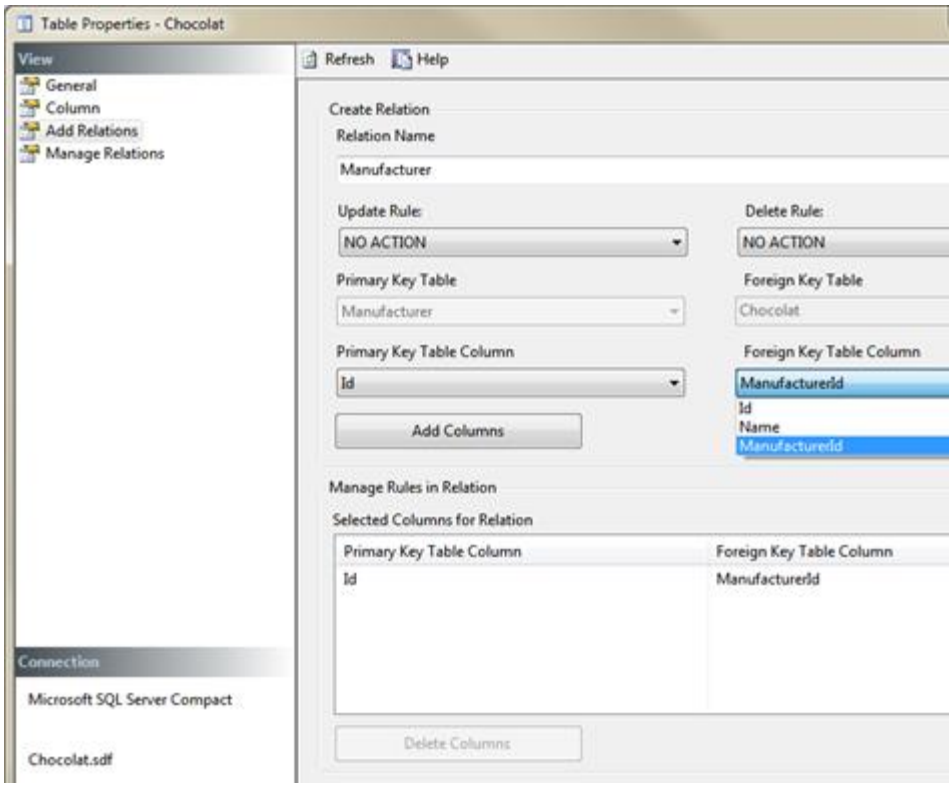
When done, click the „OK" button. In a similar manner, create a „Manufacturer" table as well, as can be seen in Figure 6.



XVII.6. Adding a table to a database III

Notice that, even though the Chocolat table has a ManufacturerId column, we have not yet declared the way for connecting the two tables (i.e., which columns to use). This can be done by adding a so-called „Relation" to the table that references another table: right click on the Chocolat table, select „Table Properties", and then click on „Add Relation". The subsequent steps are shown in Figure 7. First, one need to name the relation (let it be called

Manufacturer this time). Next, select the table to be referenced („Primary Key Table") and its primary key („Primary Key Table Column"). Then, in the table that references („Foreign Key Table") the other, select the column in which the reference itself will be stored („Foreign Key Table Column"). Finally, click the „Add Columns" button and then „OK".



XVII.7. Adding a table to a database IV

For the sake of example, let us create a Material table as well, in which ingredients for chocolats (e.g. cacao mass, cacao buttes, sugar) will be stored. Note that we are about to realize many-to-many relationship between the Chocolat and Material tables! That is, we need to create a junction table (ChocolatMaterial) as well, which contains two „Relations": one to the Chocolat table and one to the Material table.

2. Linq to SQL and Linq to Entities

Accessing an SQL database from an object-oriented application is quite a frequent task to perform for developers. Poor guys must cope with quite a tedious and lingering process every time: to create so-called entity classes that correspond to database tables, and to map columns to the properties of those classes. As mentioned already, this process is really tedious, since entity classes look very alike when one has already figured out, for instance, how to map the primitive data types of the database management system (such as MS-SQL) to the primitive data types of the object-oriented framework (such as .NET), or how to realize the table relations between entity classes. Of course, a bunch of other questions arises as well, such as, for instance, how data manipulations (insert, delete, update) on entity objects can be executed in the database. A developer may implement his/her own solutions for these problems, or apply others' ideas or even ready-to-use toolkits. Since these solutions can mostly be automated, many developers have implemented their own solutions so far, and share with the developer community such useful tools that are capable to generate source code from certain kinds of databases. Those tools are called Object-Relational Mapping (ORM) tools.

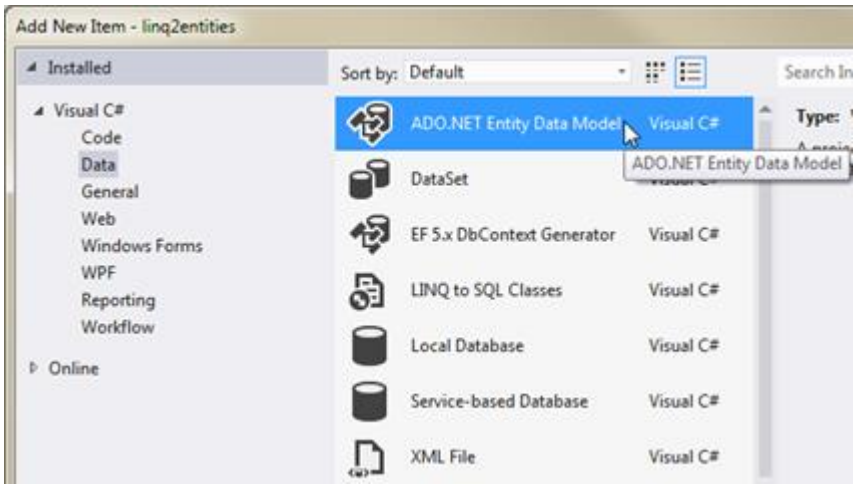
There exist a lot of ORM tools for .NET. Among them, there exist free tools, such as NHibernate, which uses an SQL-like syntax for queries, and there exist even commercial ones, such as Devart LinqConnect, which provides a link to LINQ.[4] We would like to mention here Microsoft's two own solutions. One is Linq to SQL,

[4] NHibernate supports MS-SQL, Oracle Database, PostgreSQL, and MySQL. So do Devart dotConnect and LinqConnect, depending on which release you use; they support the development to special platforms (e.g., Windows Phone, Windows Store) as well, and, furthermore, offer additional services, such as, for instance, visual modelling tools or predefined templates.

which can be considered as an entry-level ORM tool, which puts emphasis on rapid prototyping, (exclusively) on MS-SQL databases. Unfortunately, Microsoft is not always consistent with its own objectives, as illustrated by refusing to add MS-SQL CE 4.0 support to LINQ to SQL (3.5 still supported). Nevertheless, for 4.0 we can use Microsoft's other ORM tool, LINQ to Entities, which is included by the ADO.NET Entity Framework.

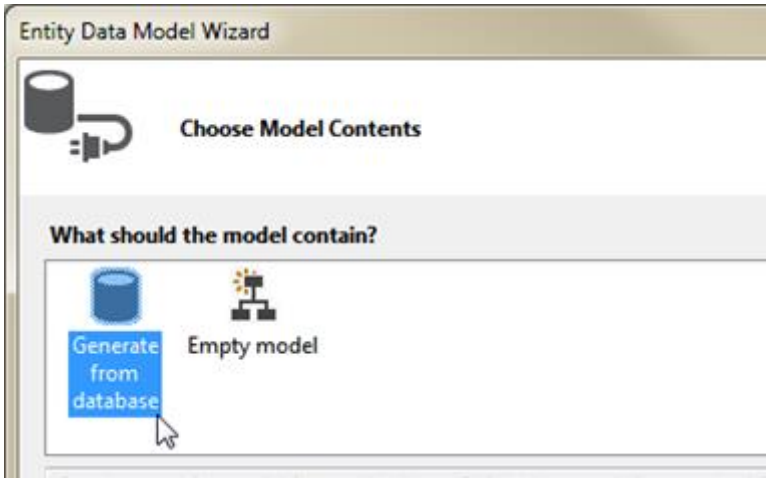
The ADO.NET Entity Framework is an open source ORM framework for .NET. It uses an architecture consists of several abstraction layers, including the database-dependent connector or provider. For numerous database management systems, there exist available (and even official) ADO.NET connectors. Another layer performs a mapping to entity classes, by generating a so-called Entity Data Model (EDM) from a given database, in XML format (EDMX file). Visual Studio offers an Entity Data Model Wizard. The bottom layers of the architecture perform various tasks, from translating (LINQ) queries into SQL, to transaction processing.

There are plenty of books on ADO.NET and Linq to Entities (Freeman & Rattz, 2010). We simply want to give a practical and motivating introduction to this topic, and therefore to show the smashing way to generate entity classes by a few clicks. This can be done even in Visual Studio. Add a new item, a so-called „ADO.NET Entity Data Model" to your project, as can be seen in Figure 8.

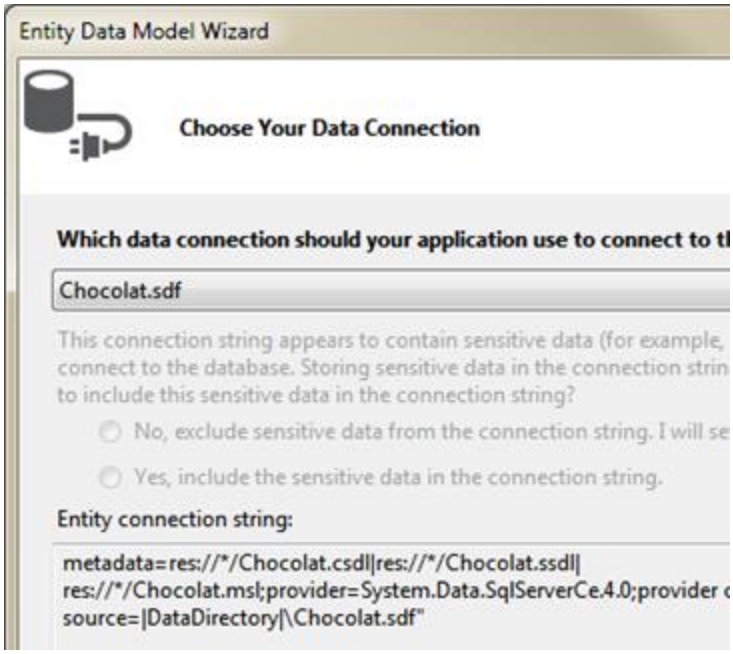


XVII.8. Creating an ADO.NET EDM in Visual Studio I

After choosing the option to generate a model from an existing database (Figure 9), select the database in the window in Figure 10.

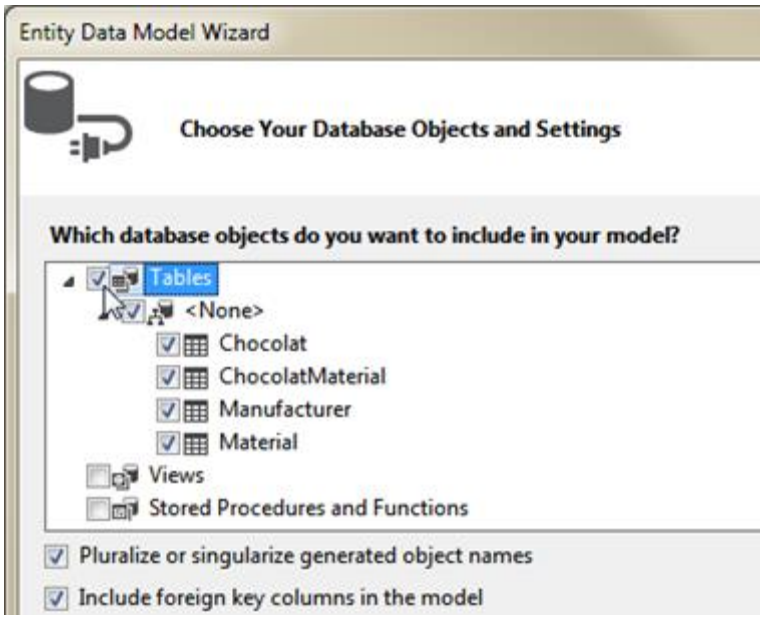


XVII.9. Creating an ADO.NET EDM in Visual Studio II



XVII.10. Creating an ADO.NET EDM in Visual Studio III

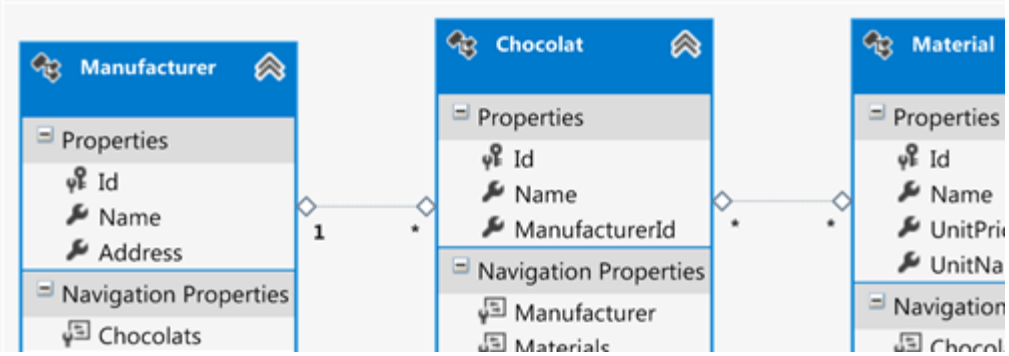
In the drop-down list, one can see databases added with Server Explorer before (if you would like to choose another database, push the „New Connection" button, which redirects you an already known window). The next form can be seen in Figure 11, on which you can select the database items that you would like to map into your model. Note that, in MS-SQL CE, even stored procedures („Stored Procedures and Functions") can be mapped into C#! The option „Pluralize or singularize generated object names" corresponds to the opportunity to give names in plural form to collections; e.g., the collection that contains all the chocolats will be called Chocolats.



XVII.11. an ADO.NET EDM in Visual Studio IV

After using the wizard, the model is generated and displayed in the visual form can be seen in Figure 12. Note that relations between tables have been detected in a correct way, i.e., Manufacturer–Chocolat is a one-to-many relation, but Chocolat–Material is many-to-many! In connection with the latter one, it is especially interesting that the ChocolatMaterial junction table does not even appear in the model. Notice furthermore that relations are represented by separate properties („Navigation Properties") in the entity classes. For example,

Chocolat.Manufacturer represents the manufacturer of a given chocolat, and Manufacturer.Chocolats the collection of all the chocolats produced by a given manufacturer.[5]



XVII.12. Entity Data Model

It is worth to browse, in Solution Explorer, the files generated by the wizard. By clicking the Chocolat.edmx file, the graph shown in Figure 12 appears, even though it is an XML file, as mentioned before. In order to see its XML content, right click on the file, click the „Open With" menu item, and then select the „XML (Text) Editor"!

When opening the Chocolat.edmx section, one can see many other files as well. For us, C# files are the most important. As can be seen, our three entities, according to the model, can be found in the Chocolat.cs, Manufacturer.cs, and Material.cs files. It is worth to take a look into these files, only to see what kinds of solutions and classes are used inside. There is an additional, very important C# file (Chocolat.Context.cs) as well, in which one can find the class that represents the database connection and a link to the tables.

Let us now show how to use the generated entity classes (Chocolat, Manufacturer és Material) in C# code. But first, we need to establish a connection to our database; the wizard makes this very easy by generating a separate „context" class (ChocolatEntities). In order to establish a database connection, we do not have anything else to do but instantiating this class. The next code sample shows a common way to do this; perform the instantiation in the App class in your WPF application, and store the ChocolatEntities instance in a static property (which is therefore initialized in the static constructor). By doing so, the instance can be accessed from any location of the source code, during the entire lifetime of the application.

```
public partial class App : Application
{
    public static ChocolatEntities db;
    static App()
    {
        db = new ChocolatEntities();
    }
}
```

From now on, our database tables can be accessed easily through the corresponding collections in the „context" instance (db.Chocolats, db.Manufacturers, and db.Materials). They all can be used in LINQ queries in the conventional way, for example:

```
var query = from ch in App.db.Chocolats
            where ch.Manufacturer.Address.Contains("Hungary") && ch.Materials.Count >= 3
```

[5] In the wizard, we have enabled the „Pluralize or singularize generated object names" option, therefore all the collections' names are in plural form.

```
        orderby ch.Name

        select ch;
```

As the example shows, the properties that represent table relations can largely unburden developers; as a matter of fact, they can almost completely avoid the usage of joins.[6]

3. Data Manipulations

In LINQ to XML, it is not supported to insert new records into our XML files, or to update or modify existing ones. SQL databases, however, offer such a support, and so do ORM tools; including LINQ to Entities, whose corresponding service is extremely simple to use. Roughly speaking, the framework registers and collects all the manipulations on entity classes, and you can eventually decide to instruct the framework, by calling a special method, to flush those manipulations into the database. This method is SaveChanges()in the „context" class.

It does not take any explaining how to perform updates; simply modify the properties of the entity classes!

```
Chocolat choco;

...

choco.Name = "Twix";

choco.Manufacturer = (

        from m in App.db.Manufacturers

        where m.Name == "Mars, Inc."

        select m

        ).First();

...

App.db.SaveChanges();
```

All the changes in the database can be tracked by using Visual Studio's Server Explorer. However, pay attention! Changes are made not in the original SDF file, but rather on its copy, which is automatically copied to the bin directory of our project every time when compiling. Consequently, if you would like to track database changes in Server Explorer, open the SDF file in the bin directory!

It is also self-explanatory how to perform inserts: create a new entity object, and add it to the appropriate collection in the „context" object! There is, however, a special case of insertion, which is shown in the second half of the code below: a new record gets inserted into a hidden junction table (ChocolatMaterial) behind the scenes, if you add an object (mat) to the appropriate collection (Materials) in an instance (choco) of one of the joint tables.

```
Material mat = new Material

{

    Name = "milk",

    UnitName = "liter",

    UnitPrice = 1.1

};

App.db.Materials.Add(mat);
```

[6] Of course, the framework generates SQL queries out of LINQ queries behind the scenes. Those SQL queries might include joins.

```
...
Chocolat choco = (
                from ch in App.db.Chocolats
                where ch.Name == "Milka Alpine Milk"
                select ch
                ).First();
choco.Materials.Add(mat);
...
App.db.SaveChanges();
```

It is also easy to perform deletes by deleting from appropriate collections. In the first half of the code below, one of the ingredients of a chocolat is deleted, which implies deleting a record from the hidden junction table (ChocolatMaterial). In the second half of the example, we would like to delete a (global) chocolat object, but beforehand we need to empty the collections in the object as well (Materials). If you miss this step, you can easily get an error message (if relations between tables have been properly declared in the database).[7]

```
Chocolat choco = (
                from ch in App.db.Chocolats
                where ch.Name == "Rum Kokos"
                select ch
                ).First();
choco.Materials.Remove((
                from m in choco.Materials
                where m.Name == "rum"
                select m
                ).First());
...
choco.Materials.Clear();
App.db.Chocolats.Remove(choco);
...
App.db.SaveChanges();
```

[7] A lot of database management systems and ORM tools support the so-called cascade delete, which means the following: when a record is deleted, all the records connected to this one are automatically deleted as well. This setting is database-dependent, and, furthermore, is only partly supported by LINQ to Entities (Freeman & Rattz, 2010).

18. fejezet - Development Environments (written by Gergely Kovásznai)

For developing WPF and Silverlight applications, there exist several integrated development environments (IDE). In this section, we are looking into two IDEs developed by Microsoft: Visual Studio and Expression Studio. In the previous sections, we have already got familiar with the usage and certain services of Visual Studio, since this environment is for programming. On the contrary, Expression Studio supports designers' work. Since the core philosophy of WPF (and Silverlight) includes the intense separation of view and code-behind, one can hope that a (XAML) GUI designed in Expression Studio and (C#) source code developed in Visual Studio fit nicely together, and one of them can be easily modified without jeopardizing fitting with the other.

In Section XVIII.1, we are going to recap a few Visual Studio services, in greater depth, supplemented with suggestions. Visual Studio 2012 is the most recent version, supporting WPF 4.5; to be more precise, we are going to use its Professional edition.[1]

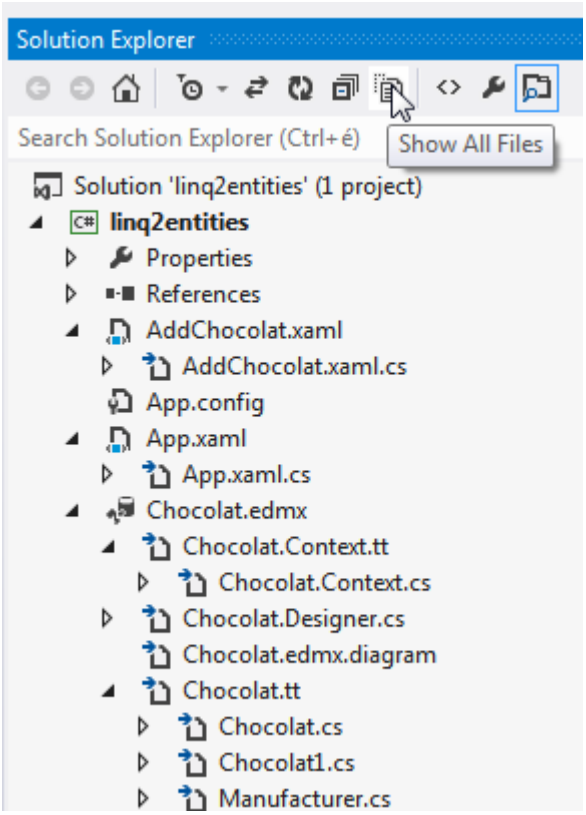
In Section XVIII.2, we will introduce Expression Blend, the most important tool in Expression Studio. The current situation with Expression Studio's versions is quite chaotic, since the most current version, Expression Studio 4 Ultimate, support WPF/Silverlight 4.0, and the upcoming version for WPF 4.5 has not been released yet.[2] Although the installer of Visual Studio 2012 offers us the opportunity to install the new Blend for Visual Studio as well, do not fall for this trick – unless you are intending to develop Window Store apps for Windows 8. In order to develop WPF 4.5 (and Silverlight 5.0) applications, currently we need to download and install another environment called Blend + SketchFlow Preview for Visual Studio 2012.

Finally, in Section XVIII.3, we will look into another tool in Expression Studio, namely Expression Design 4.

1. Visual Studio

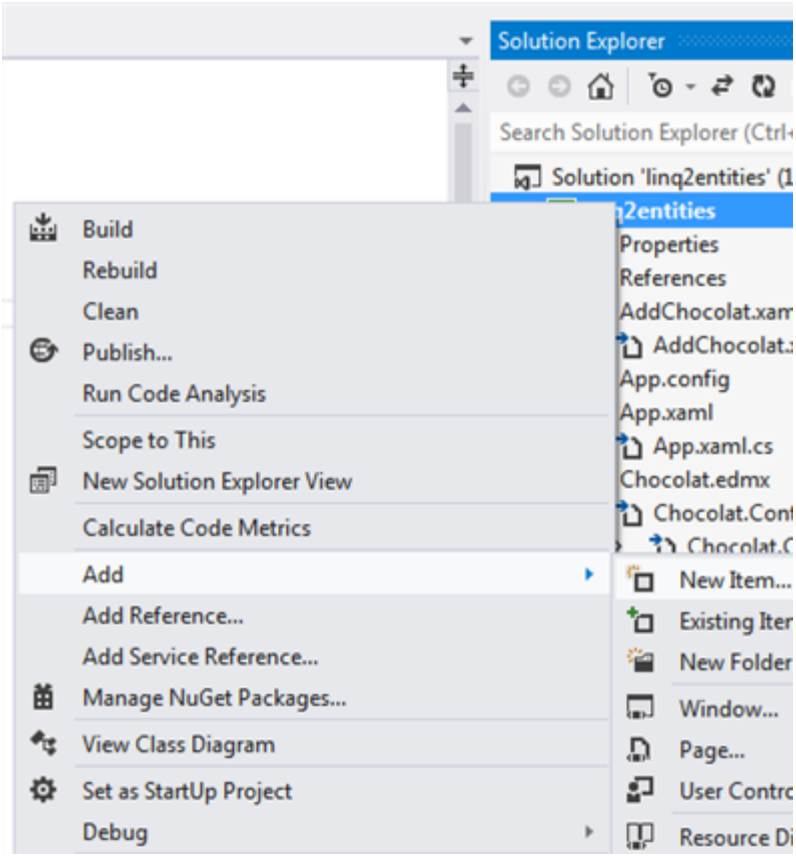
Solution Explorer is usually located on the right-hand-side of Visual Studio's interface (if not, you can find it in the "View" menu). This tool is for browsing the content of your solution. A solution can contain one or more projects; in Figure 1, a solution that contains only one project is shown (this is the rather common case). The individual projects might contain several files. There might be hidden directories and files, e.g., the ones created during compile time; all of these can be displayed by clicking the „Show All Files" button.

[1] Instead, one can also use Visual Studio Express 2012 for Windows Desktop, free of charge.
[2] Microsoft announced to discontinue Expression Studio 4. Some of its tools are no longer supported, so is Expression Design, therefore, probably, no new version can be expected; the current version 4 is available for download at no charge. Blend, as a component in Visual Studio, is henceforward supported.



XVIII.1. Solution Explorer and the "Show All Files" button

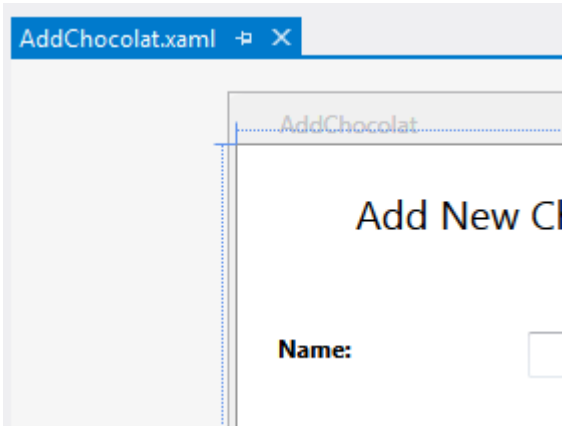
A WPF project always includes an App.xaml and a MainWindow.xaml XAML file, however one can add further XAML (or any) files to a project, as shown in Figure 2. “Behind” each XAML file, there also exists a C# file (with .cs extension), in which the code-behind is stored (e.g., event handlers). A project, of course, can contain lots of other files as well, e.g., image files (Section V.2.3), audio and video files (Section V.2.4), XML files (Section XVI), database files (Section XVII.1), data models (Section XVII.2) together with all the supplementary files (c.f. Figure 1 again).



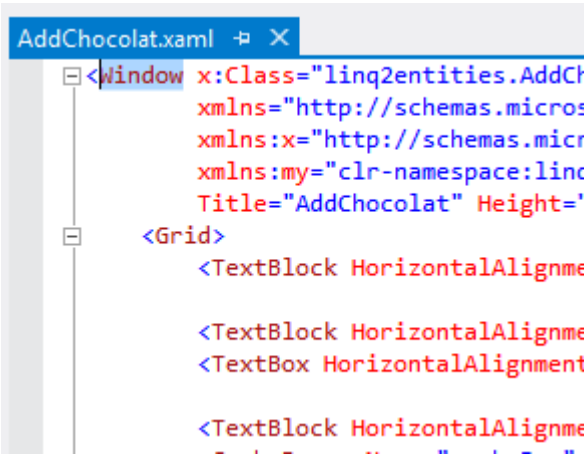
XVIII.2. Adding a new file to a project

1.1. Designer

A XAML file has three different views. The default view can be seen in Figure 3 and is called the Design view, provided by the Designer tool; in this view, one can “draw” the interface of a form. Another view is the XAML view, shown in Figure 4.

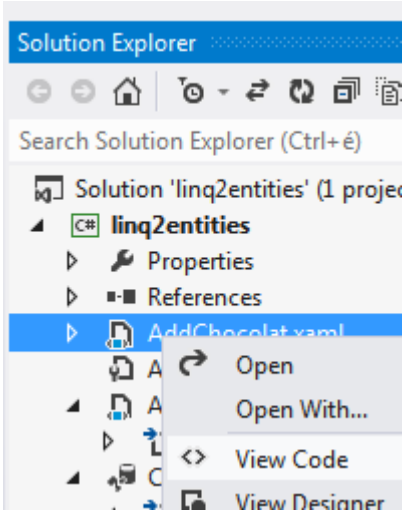


XVIII.3. Designer - Design view

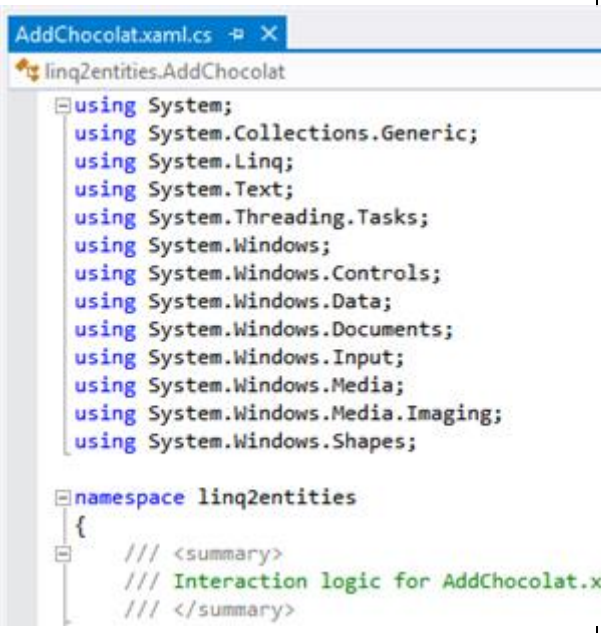


XVIII.4. Designer - XAML view

In Figure 6, the third view called the Code view can be seen, which displays the (C#) code-behind; one can open this view in Solution Explorer (or in the View menu).



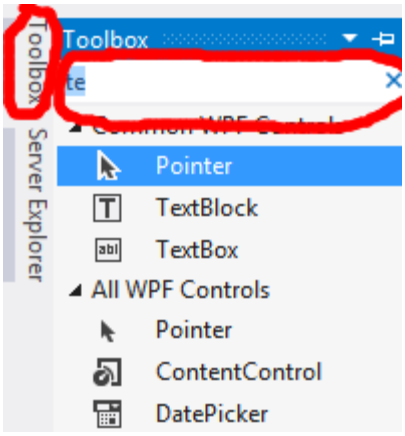
XVIII.5. Opening the Code view



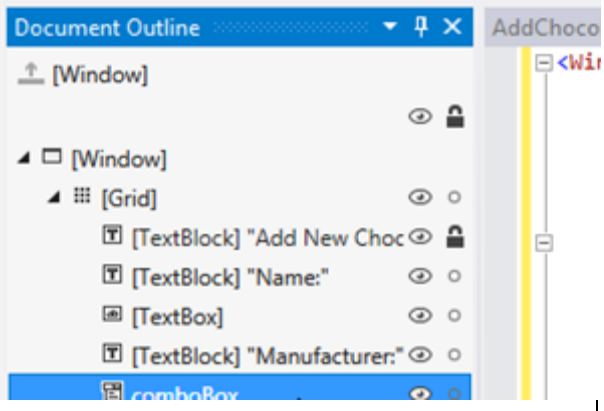
XVIII.6. Code view

1.2. Toolbox and Document Outline

In the Design view, one can place new controllers, one after the other, on a form by using the Toolbox tool, by drag&drop. One usually opens the Toolbox by clicking the vertical button on the left-hand-side of the screen, as it can be seen in Figure 7 (or, alternatively, in the View menu). The Toolbox contains numerous controllers, hence it is a real pain to search among them, and therefore it is expedient to use the search box at the top of the Toolbox.



XVIII.7. Visual Studio: Toolbox



XVIII.8. Visual Studio: Document Outline

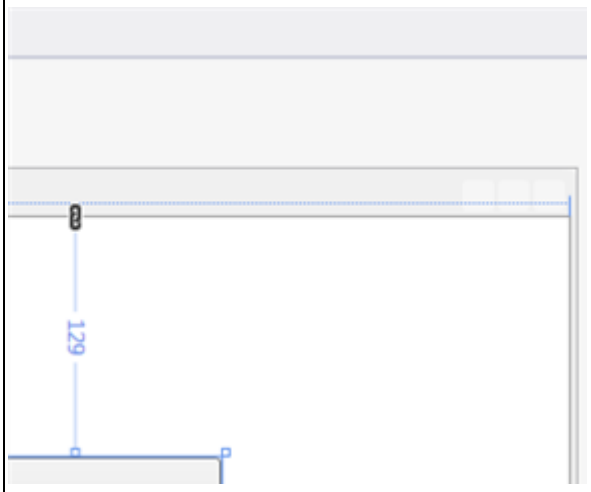
The Document Outline tool (which can be accessed in the View/Other Windows menu) visualizes the (tree) structure of the form, and accesses the individual controllers, as can be seen in Figure 8. Furthermore, one can switch on/off their visibility and lock/unlock them.

1.3. Properties

In the Properties window, which is usually located to the right (and can be switched on/off in the View menu), one can browse and modify the properties of the controller selected in the Designer. The Properties window

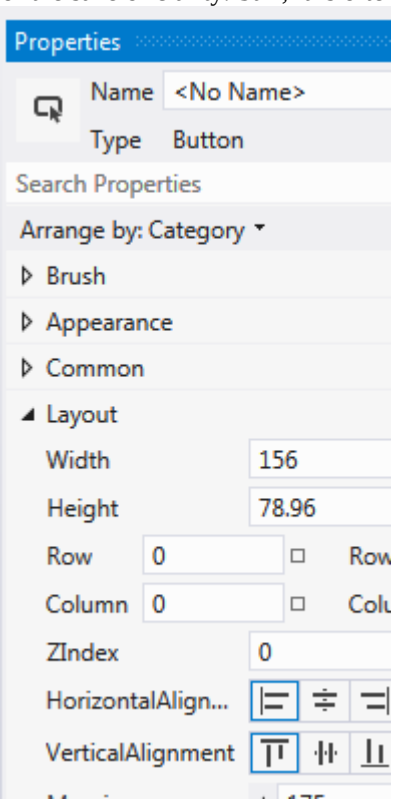
contains two tabs: while the afore-mentioned properties can be browsed on the default tab as shown in Figure 9, the events (and event handlers) of a controller can be browsed on the tab shown in Figure 10.

XVIII.9. Properties tool



XVIII.10. Properties tool - Events

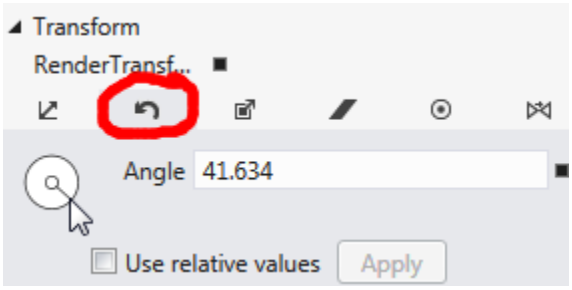
In Figure 9, it can also be seen that properties are divided into categories, for the sake of clarity. Still, it is often quite a challenge to find certain properties; the search box helps here again.



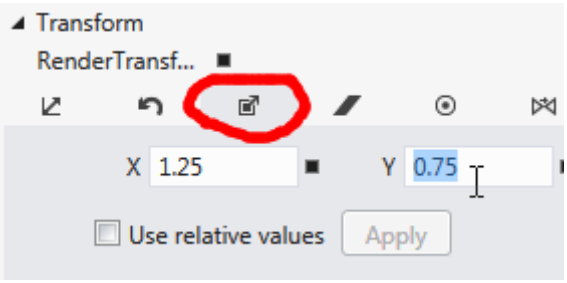
Distinct ways are used for editing the individual properties, but they are quite intuitive; we are not giving details on each, except for the following ones. But first, we would like to pan out about the role of the small checkboxes to the right of the properties (Figure XVIII.11), especially about Reset, i.e., setting the property value to default. This feature might be very important if one uses the Designer to generate XAML code, instead of writing it manually. Of course, the Designer floods the XAML code with lots of (often) unnecessary property settings, which can be cleaned up by Reset.

1.4. Transformations

One can apply transformations (Section Hiba! A hivatkozási forrás nem található.) to controls by using the convenient interface that can be seen in Figures 12 and 13. This interface offers access to the four basic transformations (in the figures you can see two of them), and, on the remaining two tabs, you can also access the transformation center point and the Flip transformation (realized by TranslateTransform).



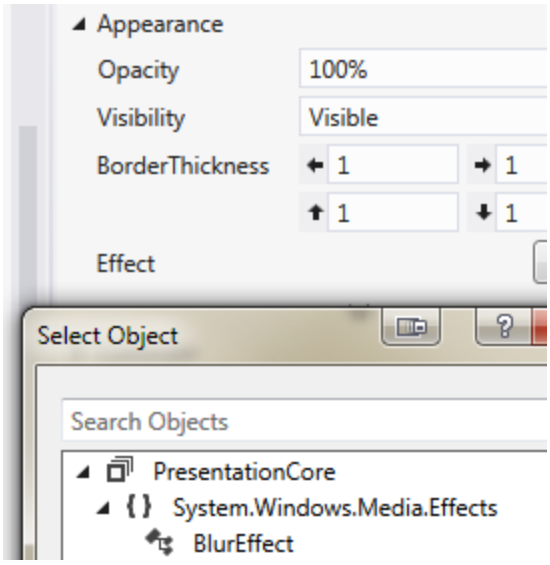
XVIII.12. Transformations - Rotate



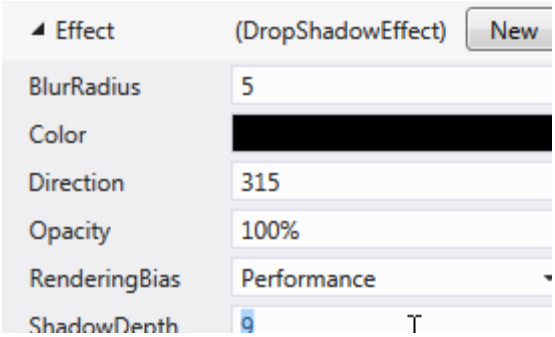
XVIII.13. Transformations - Scale

1.5. Effects

You can add effects (Section IX) to controls by clicking the „New” button in the „Appearance” category. By doing so, new fields appear, according to the selected effect, as can be seen in Figures 14 and 15.



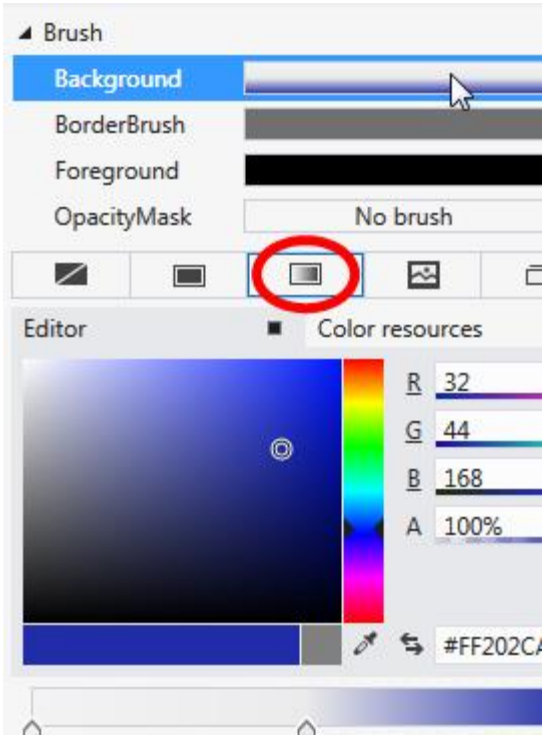
XVIII.14. Effects



XVIII.15. Effects - DropShadowEffect

1.6. Brushes

One can color controls by using brushes (Section Hiba! A hivatkozási forrás nem található.); occasionally, different brushes might be used for coloring e.g. the background, the foreground, or the border. All these can be set in the window shown in Figure 16, by selecting the target of the brush at the top. The tabs below that can be used to select a brush type, e.g., a SolidColorBrush or some kind of gradient brush (this particular case is shown in the figure), or even an ImageBrush. In the case of a gradient brush, the slider in Figure 17 can be used to customize its GradientStops, to add new ones, or to delete existing ones. In the bottom left of this slider (Figure 18), you can select the kind of the gradient brush, i.e., either LinearGradientBrush or RadialGradientBrush.



XVIII.16. Setting a brush

XVIII.17. Gradient stop

XVIII.18. Different kinds of gradient brushes

2. Blend

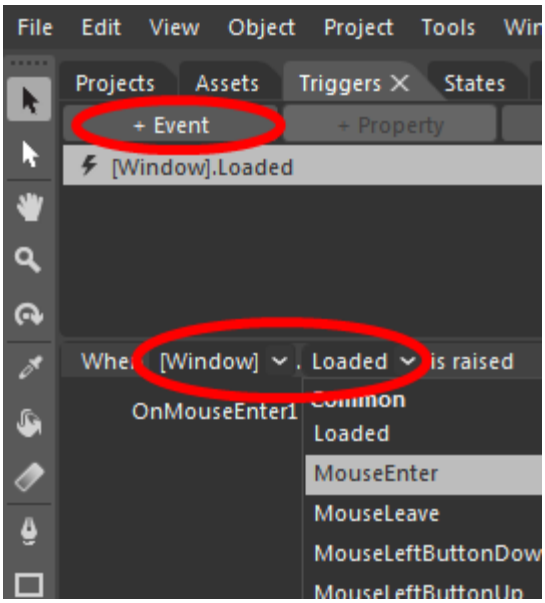
Blend provides an interface that is similar to of Visual Studio in many ways. In Blend, Solution Explorer can be found on the Projects tab, Toolbox on the Assets tab in the upper left corner. Just below that, Objects and Timeline, which corresponds to Document Outline, is located. The Designer and Properties tools are also similar to their counterparts in Visual Studio. In the Properties window, you can set transformations, effects, and brushes in the way as shown in the previous section.

There is one significant difference as compared to what Visual Studio offers, namely how one can edit animations. Blend makes this really easy for designers.

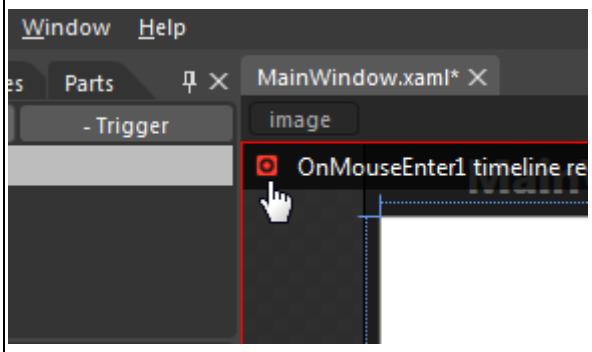
2.1. Triggers and Animations

Let us create a sample program, in which a control will be animated, triggered by an event. Place a control on your form (from the Assets tab). Let us assume that we would like to implement the followings: when the mouse pointer enters the form, we make the invisible control appear gradually, wait for 1 second, and then fade it out. Do not forget to set the (default) Opacity of your control to 0% (Figure 22)!

One can monitor an event by using a trigger (Section X). A trigger can be created on the Triggers tab, next to Assets, by clicking the „+ Event” button. One can specify the control’s event that is supposed to be watched by the trigger, as can be seen in Figure 19. Let us choose Window.MouseEnter! Blend – if we have not added an animation to our project yet – asks whether we would like to create a Storyboard (Section Hiba! A hivatkozási forrás nem található.); Blend is going to give it a unique name (e.g. OnMouseEnter1).



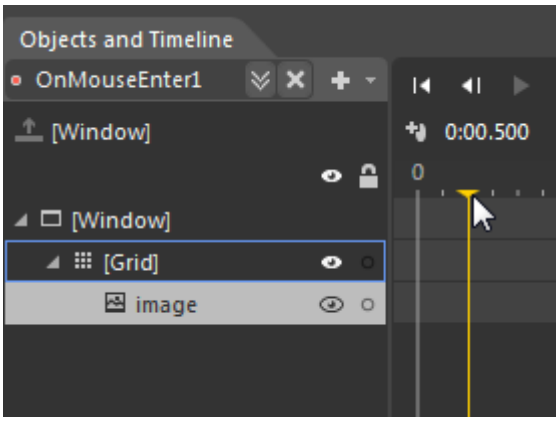
XVIII.19. Adding a trigger



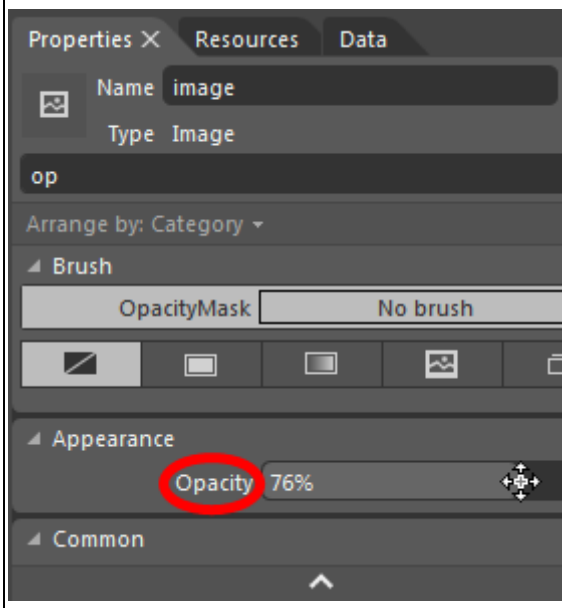
XVIII.20. Timeline recording

One can edit a Storyboard by using the Objects and Timeline tool; although, actually, all our „actions" are recorded in the Storyboard (all what we do, for example, in the Designer or in the Properties). This mode is called Timeline recording, which, when switched on, displays a red frame around the Designer (Figure 20). By clicking the small icon in the upper left corner of this frame, one can (temporarily) exit this mode, or enter back later.

The first step of our animation is supposed to fade in a control during a given time period (e.g., half a second). As can be seen in Figure 21, we therefore set the time on the Timeline to half a second. After doing so, all we are doing is being "recorded" for this specific moment. Thus, select the control you want, and then set its Opacity to 100% (Figure 22)! The animation that you have made so far can be played by using the buttons above the Timeline. Note that WPF calculates the opacity values between two moments – actually, between two keyframes – automatically.

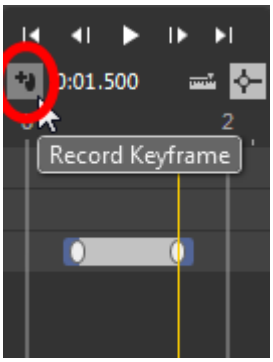


XVIII.21. Timeline – Setting time

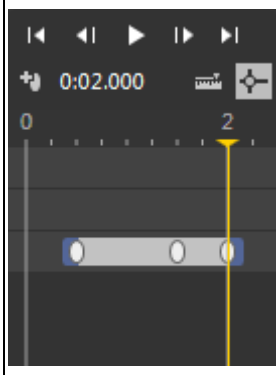


XVIII.22. Setting opacity

Let us suppose that then we would like to leave the control unchanged for 1 second, and, after that, to fade it out in half a second. For this, it is necessary to insert a keyframe at 1.5 seconds, by click the button shown in Figure 23. Last, we only need to set the Timeline to 2 seconds, and then to decrease the opacity of the control to 0%. The final result can be seen in Figure 24. Note that, beside the keyframe we have inserted, Blend has added two other keyframes automatically as well.



XVIII.23. Timeline – Adding a keyframe



XVIII.24. Timeline – Keyframes added automatically and manually

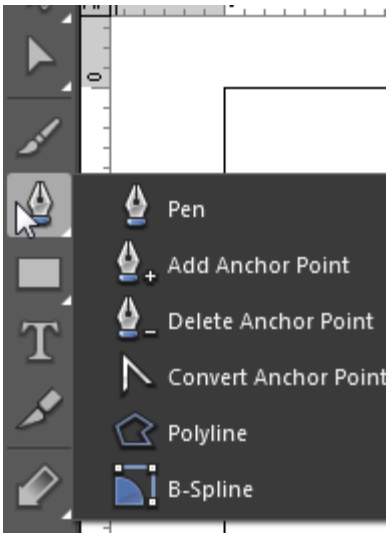
Execute the program and enjoy the animation whenever you are pulling the mouse pointer above the window![3] Of course, you can further improve the animation with several other movements; e.g., if the animated control is an Image of a frog, you could make the frog "hop" by moving the Image up and down from time to time (i.e., from keyframe to keyframe), or by rotating it, and so on.

3. Expression Design

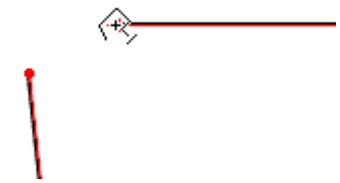
Expression Design is a vector graphics editor by Microsoft, which makes it possible to export figures and shapes into XAML. It is extremely simple to use this software, and, therefore, we would like to introduce it through an illustrative example. Let us edit the Path that was used in the examples in the Sections Hiba! A hivatkozási forrás nem található. and Hiba! A hivatkozási forrás nem található..

From the Toolbox of Expression Design, one can choose building components that are needed for the wanted shape. We are going to use a polyline and a curve (B-Spline); as can be seen in Figure 25, one can display the menu containing these components by right click. First, draw the polyline to form the bottom part of the shape (Figure 26). Then, for the sake of example, modify the polyline: add new anchor points to it, and move one of them (Figure 27). Finally, draw a spline as the missing part of the border (Figure 28). Of source, one might continue with adding design to the shape, e.g., by setting the width or color of lines, or applying transformations, etc. Nevertheless, our aim is now only to export a "bare" shape into XAML; design steps can be applied afterwards to the WPF form that will use this shape (c.f. Sections Hiba! A hivatkozási forrás nem található. and Hiba! A hivatkozási forrás nem található.).

[3] We have had some problem with executing WPF projects from Blend. It is not clear whether this problem originates in Blend + SketchFlow Preview for Visual Studio 2012 or in our own system. Nevertheless, the problem can be solved by opening those projects in Visual Studio (as well), and by compiling and executing them there.

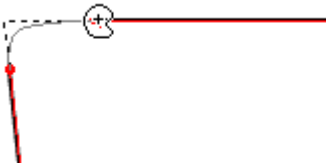


XVIII.25. Toolbox - Polyline, Anchor Point and B-Spline



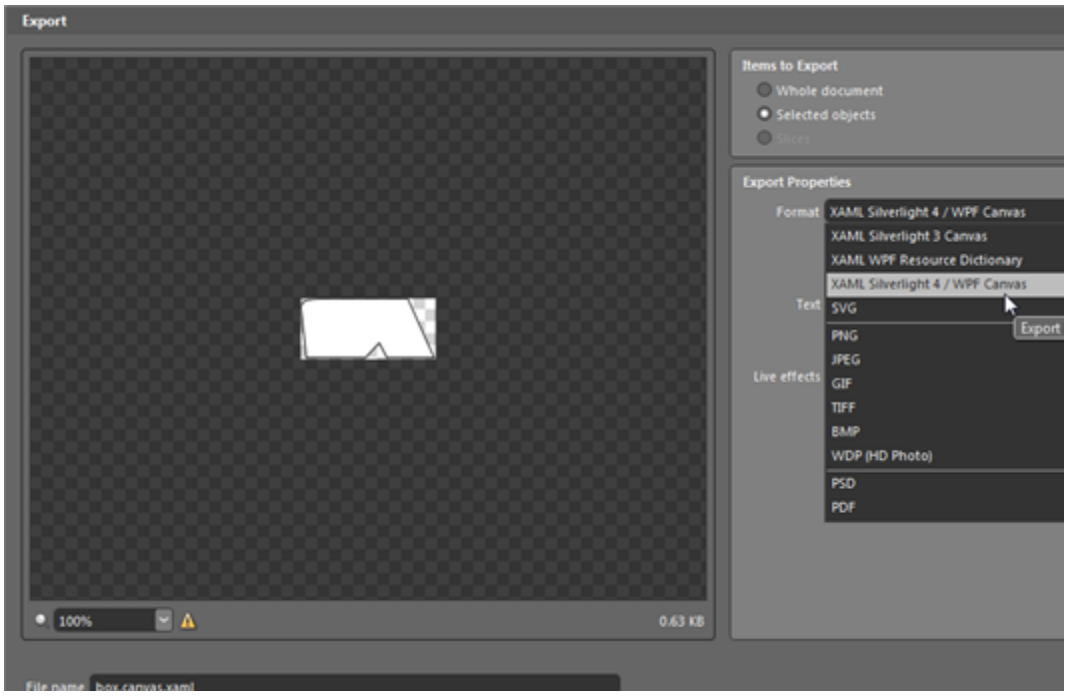
XVIII.26. Drawing a Polyline

XVIII.27. Adding and moving Anchor Points



XVIII.28. Drawing a B-Spline

There is only one thing left to do: to export the shape (into XAML). For this, one need to select the shape, and then to click the „File/Export” menu item. There exist several options for choosing the output format; let use choose „WPF Canvas”, as can be seen in Figure 29.



XVIII.29. Exporting as a WPF Canvas

By doing so, the shape is exported as a Path (included by a Canvas). The Canvas is actually unnecessary (and can be deleted); the Path itself is what matters, and so is, in particular, its Data property, in which the "command sequence" to draw the shape takes place (c.f. Section VII.2.3).[4]

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

        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" Width="112.313"

        Height="50.5" Clip="F1 M 0,0L 112.313,0L 112.313,50.5L 0,50.5L 0,0">

    <Path Width="112.354" Height="50.5" Canvas.Left="-0.041048" Canvas.Top="0"

        Stretch="Fill" StrokeLineJoin="Round" Stroke="#FF000000" Fill="#FFFFFFFF"

            Data="F1 M 21.8125,0.500015C 14.3125,0.500015 6.8125,0.500015 3.3125,2.50002C -0.1875,4.50002 0.312508,8.5 0.812508,12.5L 4.81251,50L 54.3125,50L 65.8125,37.5L 72.3125,50L 111.813,50L 89.8125,0.5L 21.8125,0.500015 Z " />

</Canvas>
```

[4] By using another output format, the "XAML WPF Resource Dictionary", one can export a shape as a DrawingBrush (Section VI.2.5).

19. fejezet - Epilogue

In this lecture note, we lost ourselves in several useful topics, to which a programmer might often face during his/her every day work. First, we got to know the most important and fundamental solutions applied in WPF (and also in other technologies), such as, for instance, triggers, data binding, styles/templates, etc. In order to fulfill the GUI requirements and to make designers satisfied, we looked into creating animations as well. Of course, there are so many other topics even in WPF, e.g., data views, handling media, 3D support, etc.; literature listed in the References might help to elaborate on those topics, and so do the numerous tutorials and forums on the internet. This holds even better for Silverlight, as a technology that is becoming more and more dominantly supported by Microsoft; consider platforms such as Windows Phone or Windows 8! With respect to the latter one, there are, of course, many new directions, one can even develop Windows Store applications as well; however, one can do this even on XAML and C# base.

Among these topics, LINQ seems a little bit as an outsider, since it is much less scenic, for instance, to join two data collections than to rotate a button; nevertheless, such data collections (usually being SQL databases) and ORM tools on the top are together the foundation for modern and robust applications. We particularly recommend the listed literature in these topics, and to get to know other ORM tools as well (some of them were even mentioned).

We wish success in further investigations and professional development!

Bibliográfia

Albahari, J., & Albahari, B. LINQ Pocket Reference. O'Reilly. 2008

Bennage, C., & Eisenberg, R. Tanuljuk meg a WPF használatát 24 óra alatt. Kiskapu. 2009

Freeman, A., & Rattz, J. C. Pro LINQ - Language Integrated Query in C# 2010. APress. 2010

Bennage, Christopher; Eisenberg, Rob - Tanuljuk meg a WPF használatát 24 óra alatt Kiskapu Kiadó 2009

Kovács, E., Hernyák, Z., Radvány, T., & Király, R. A C# programozási nyelv a felsőoktatásban - Programozás tankönyv. 2005.,http://csharptk.ektf.hu/

MacDonald, M. Pro WPF in C# 2012 - Windows Presentation Foundation in .NET 4.5. APress. 2012., http://www.cordis.lu/ist/ka3/digicult/lund_p_browse.htm

Reiter, I. C#. http://www.scribd.com/doc/42063752/Reiter-Istvan-C-2009-350-oldal 2009., http://www.scribd.com/doc/42063752/Reiter-Istvan-C-2009-350-oldal