WPF erőforrások kezelése

Jánosi-Rancz Katalin Tünde

Sapientia EMTE
tsuto@ms.sapientia.ro

Erőforrások - Resources

- erőforrás alatt általában egy olyan külső objektumot (pl. kép- vagy hangfájlt) értenek, mely az alkalmazás bármely pontjáról elérhető, minden modul által használható
- a WPF-ben (általánosabb) erőforrás lehet bármely külső fájl, sőt bármely osztály példánya, elsősorban:
  - stílusok (`Style`): a felületi elemek egységes megjelenését definiálják
  - sablonok (`Template`): a vezérlők felépülését és adatkötéseit definiálják
  - forgatókönyvek (`StoryBoard`): animációk végrehajtását biztosítják
- lehet:
  - bináris
  - statikus
  - dinamikus (teljesítmény csökkenés)

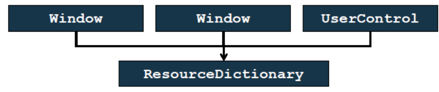
Erőforrások a felületi kódban

- bármely felületi elem (UIElement) tartalmazhat erőforrásokat a `Resources` tulajdonság segítségével

```
<Window ... >
    <Window.Resources>
        ... <!-- eroforrások az egész ablakra -->
    </Window.Resources>
    <Grid Name="LayoutRoot">
        <Grid.Resources>
            ... <!-- rácson belüli eroforrások -->
        </Grid.Resources>
     ...
    </Grid>
</Window>
```

Erőforrásfájlok

- amennyiben több ablak, vagy vezérlő számára biztosítani akarjuk ugyanazt a stílus-, animáció-és sablonkészletet, akkor használhatunk erőforrásfájlokat (ResourceDictionary)
- csak XAML erőforrásokat tartalmazó fájlok
- használatba vehetőek bármely ablakban és egyedi vezérlőben, vagy akár a teljes alkalmazásban (az `App` osztályon keresztül)



Erőforrásfájlok -2

- pl. erőforrásfájl - StyleDict.xaml:

```
<ResourceDictionary ... >
    <Style x:Key=... > <!--stíluselem -->
    ...
</ResourceDictionary>
```

- felhasználása egy ablakban (MainWindow.xaml):

```
...
<Window.Resources>
    <ResourceDictionary Source=”StyleDict.xaml” />
    <!--eroforrásfájl betöltése -->
</Window.Resources>
```

Erőforrások használata

- minden egyes erőforrásunkhoz rendelhetünk egy egyedi azonosítót `x:Key`
- ezzel a kulccsal tudunk majd hivatkozni rá a `StaticResource` elem segítségével:

```
<GridName = ”grid”>
    <Grid.Resources>
        <Style x:Key = ”buttonStyle”> ... </Style>
    </Grid.Resources>
...
<Button Style = ”{StaticResource buttonStyle}”>
```

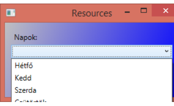
- maga a `Resources` tulajdonság egy asszociatív tömb, amely a kulcsok szerint indexelt, pl.:

```
Style myButtonStyle = (grid.Resources[”buttonStyle”] as Style);
```

DynamicResource

- a StaticResource egyetlen egyszer hajtódik végre betöltéskor. Ha az erőforrás később megváltozik (pl. kódból), akkor a változás nem lesz látható ott, ahol StaticResource-ként volt használva

```
...
xmlns:sys="clr-namespace:System;assembly=mscorlib"
Background="{DynamicResource WindowBackgroundBrush}">

<Window.Resources>
    <sys:String x:Key="ComboBoxTitle">Napok:</sys:String>
    <x:Array x:Key="ComboBoxItems" Type="sys:String">
            <sys:String>Hétfo</sys:String>
            <sys:String>Kedd</sys:String>
            <sys:String>Szerda</sys:String>
    </x:Array>

    <LinearGradientBrush x:Key="WindowBackgroundBrush">
            <GradientStop Offset="0" Color="Silver"/>
            <GradientStop Offset="1" Color="Blue"/>
    </LinearGradientBrush>
</Window.Resources>
<StackPanel Margin="10">
    <Label Content="{StaticResource ComboBoxTitle}" />
    <ComboBox ItemsSource="{StaticResource ComboBoxItems}" />
</StackPanel>
```



Stílusok

Stílusok - Styles

- ahelyett, hogy meghatároznánk minden elem megjelenését, definiálhatunk stílusokat, amelyeket erőforrásokként tárolhatunk
- egyszerre számos elem kinézetét vezérelhetjük egy helyről
- a kód redundanciát kerülhetjük el
- lehetőséget biztosít, hogy cseréljük a stílust futásidőben
- sokkal gyorsabb mint Code Behind-ból
- főbb tulajdonságai:
  - Key
  - TargetType
  - Setters

Stílusok

- a `Setter` -ek segítségével, a `TargetType` -ban megjelölt osztály valamelyik tulajdonságát (`Property`) állíthatjuk, be valamilyen konkrét értékre (`Value`)

```
<Style x:Key="buttonStyle" TargetType="{x:Type Button}">
    <Setter Property="Width" Value="400"/><!--egyszeru érték -->
    <Setter Property="Canvas.Left" Value="200" />
    <Setter Property="RenderTransform">
        <Setter.Value> <!--összetett érték -->
            <TranslateTransform X="100" Y="50" />
        </Setter.Value>
    </Setter>
</Style>
```

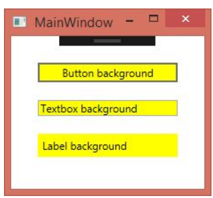
- `TargetType="Button"` is helyes ITT, de az `x:Type` -ot a stílus öröklődések miatt jó odaírni, mert öröklődésnél keresi az első kulcs nélküli típust, és ha annak a `TargetType` -ja nem `x:Type` -al van írva akkor nem találja meg

Stílusok - közös stílus

- a `TargetType` -nak nem lehet megadni több vezérlőt egyszerre, `TargetType=”Button, TextBox”` helytelen!!
- megoldás: TargetType megadás nélkül és `Property=”Control. ”`

```
<Style x:Key=”CommonStyle”>
   <Setter Property=”Control.Background” Value=”Yellow”/>
   <Setter Property=”Control.VerticalAlignment” Value=”Center”/>
   <Setter Property=”Control.Margin” Value=”10”/>
</Style>
```

```
<Button Style=”{StaticResource CommonStyle}” Content=”Button background”/>
<TextBox Style=”{StaticResource CommonStyle}” Text=”Textbox background” />
<Label Style=”{StaticResource CommonStyle}” Content=”Label background”/>
```



Stílusok - Styles

- megadhatóak elemenként, pl.:

```
<Button Content=”BlueButton”>
    <Button.Style>
      <Style TargetType=”{x:Type Button}”>
            <Setter Property=”Foreground” Value=”Blue” />
      </Style>
    </Button.Style>
</Button>
```

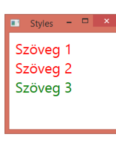
- megadhatóak erőforrásként:

```
<Style x:Key=”buttonStyle” TargetType=”{x:Type Button}”> <!--megadható
     a céltípus is -->
    <Setter Property=”Foreground” Value=”Blue” />
</Style>
...
<Button Style=”{StaticResource buttonStyle}” />
```

Stílusok típusai

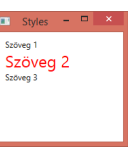
- `implicit`: mennyiben nem adunk meg kulcsot, úgy a stílus az összes megadott típusú elemre érvényes lesz, nem szükséges a `StaticResource` hivatkozás

```
<StackPanel Margin="10">
    <StackPanel.Resources>
        <Style TargetType="{x:Type TextBlock}">
            <Setter Property="Foreground" Value="Red" />
            <Setter Property="FontSize" Value="24" />
        </Style>
    </StackPanel.Resources>
    <TextBlock>Szöveg 1</TextBlock>
    <TextBlock>Szöveg 2</TextBlock>
    <TextBlock Foreground="Green">Szöveg 3</TextBlock>
</StackPanel>
```



- `explicit`: az `x:key` megadásával jelzem, hogy csak akkor akarom használni amikor hivatkozok rá

```
<Window.Resources>
    <Style x:Key="SzövegStílus" TargetType="{x:Type
        TextBlock}">
        <Setter Property="Foreground" Value="Red" />
        <Setter Property="FontSize" Value="24" />
    </Style>
</Window.Resources>
<StackPanel Margin="10">
    <TextBlock>Szöveg 1</TextBlock>
    <TextBlock Style="{StaticResource SzövegStílus}">
        Szöveg 2</TextBlock>
    <TextBlock>Szöveg 3</TextBlock>
</StackPanel>
</Window>
```



NB: az App.xaml-be írt stílusok kulcsa nem fog látszani más fileokban (Intellisense nem segít), Warning-ot ad, de működik

Stílus öröklés

- egy WPF stílus alapozhat egy másik stílusra a `Style.BasedOn` tulajdonság használatával
- megadhatunk egy base stílust, amely meghatározza a közös tulajdonságokat és ebből származtathatunk speciálisabbat
- egy származtatási fa építhető fel, egy stílus az összes őse Setter-ét örökli és azokat felül is írhatja

```
<Style x:Key="baseStyle">
  <Setter Property="FontSize" Value="12" />
  <Setter Property="Background" Value="Orange" />
</Style>

<Style x:Key="boldStyle" BasedOn="{StaticResource baseStyle}">
  <Setter Property="FontWeight" Value="Bold" />
</Style>
```

Eseménykezelés stílusokkal

- a stílusok viselkedést is leírhatnak, eseménykezelőket is rendelhetünk
- a `Style.EventSetter` egy különleges típusú setter, melynek az `Event` tulajdonságában beállított eseményhez, a `Handler` tulajdonságában megadott, eseménykezelő rendelődik

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

```
private void StackPanel_MouseEnter(object sender, MouseEventArgs e)
{ ... }
```

- lásd: Code/WPFAdvanced példát

Triggerek

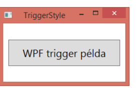
Kioldók - Triggers

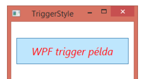
- leggyakrabban stílusokhoz rendelhetjük
- olyan `setterek`, amelyeket egy vagy több feltétel alapján állítunk be
- kioldókkal beállíthatjuk, hogy hogyan reagáljon dinamikusan egy adott vezérlő egy esemény bekövetkeztére, vagy egy tulajdonság megváltozására
- típusai:
  - `Property Triggerek` – meghívódik, amikor egy vezérlő saját property-ének értéke megváltozik (pl. button IsMouseOver)
  - `Data Triggerek` – meghívódik, amikor egy .Net property értéke megváltozik (pl. egy hozzá bindolt adat vmilyen feltételnek eleget tesz)
  - `Event Triggerek` – esemény változásra hívódik meg

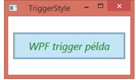
Property Triggerek

- ha a Trigger sora igaz, akkor végrehajtódik a Setter

```
<Window.Resources>
    <Style x:Key="TriggerStyle">
        <Setter Property="Control.FontSize" Value="20"></Setter>
        <Setter Property="Control.HorizontalAlignment" Value="Center"></Setter>
        <Setter Property="Control.Margin" Value="10"></Setter>
        <Setter Property="Control.Foreground" Value="Black"></Setter>
        <Style.Triggers>
            <Trigger Property="Control.IsMouseOver" Value="true">
                <Setter Property="Control.FontStyle" Value="Italic"></Setter>
                <Setter Property="Control.Foreground" Value="Red"></Setter>
                <Setter Property="Control.Background" Value="Yellow"></Setter>
            </Trigger>
            <Trigger Property="Button.IsPressed" Value="true">
                <Setter Property="Control.Foreground" Value="Green"></Setter>
                <Setter Property="Control.Background" Value="Blue"></Setter>
            </Trigger>
        </Style.Triggers>
    </Style>
</Window.Resources>
<Grid>
    <Button Height="50" Width="214" Style="{StaticResource TriggerStyle}"  Content="WPF trigger példa" />
</Grid>
```







Button normal, IsMouseOver, IsPressed

Data Triggerek

- meghívódik amikor egy .Net property értéke megváltozik (pl. egy hozzá bindolt adat vmilyen feltételnek eleget tesz)

```
<TextBlock x:Name="SmallText" Text="Small">
            <TextBlock.Style>
                <Style TargetType="{x:Type TextBlock}" BasedOn="{StaticResource BaseStyle}">
                    <Style.Triggers>
                        <DataTrigger Binding="{Binding ElementName=BigText, Path=IsMouseOver}" Value="True">
                            <Setter Property="FontSize" Value="60"/>
                        </DataTrigger>
                    </Style.Triggers>
                </Style>
            </TextBlock.Style>
</TextBlock>

<TextBlock x:Name="BigText" Text="Big" Grid.Row="1">
            <TextBlock.Style>
                <Style TargetType="{x:Type TextBlock}" BasedOn="{StaticResource BaseStyle}">
                    <Style.Triggers>
                        <DataTrigger Binding="{Binding ElementName=SmallText, Path=IsMouseOver}" Value="True">
                            <Setter Property="FontSize" Value="10"/>
                        </DataTrigger>
                    </Style.Triggers>
                </Style>
            </TextBlock.Style>
</TextBlock>
```

Small Big | Small Big | Small Big

Normal, IsMouseOver a Small-on, IsMouseOver a Big-en

Sablonok - Templates

Sablonok - Templates

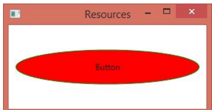
- céljuk a vezérlők tetszőleges testreszabhatósága, azok funkcionalitásának megőrzése mellett

Sablon típus	Leírás
`ControlTemplate`	Lehetővé teszi a vezérlők vizuális szerkezetének a megadását és felülírását
`ItemsPanelTemplate`	Egy ItemsControl itemjeinek megadhatjuk a kinézetét ItemsPanelTemplate hozzárendelésével. Minden ItemsControlnak van egy alapértelmezett ItemsPanelTemplate-je. A MenuItem WrapPanel-t, a StatusBar DockPanel-t használ, és a ListBox VirtualizingStackPanel-t.
`DataTemplate`	Ezek nagyon hasznosak az objektumok grafikus ábrázolásában. Amikor stílusformázunk egy ListBox-ot, alapértelmezetten az itemek toString metódussal íródnak ki. DataTemplate-ket használva felül tudjuk írni ezt a viselkedést, és meghatározhatunk egyéni megjelenést az itemeknek.
`Hierarchical DataTemplate`	Fa típusú objektumok elrendezésére használjuk. Ez a vezérlő támogatja TreeViewItem és a MenuItem-ek elrendezését.

Vezérlő sablon - ControlTemplate

- testreszabhatja a vezérlők teljes megjelenését
- gyakran szerepel stílusban, amely más tulajdonságokat is tartalmazhat
- a sablont erőforrásként kell megadnunk, kulccsal
- a sablonban található tulajdonságokat `TemplateBinding` -gal köthetjük a vezérlőnk bizonyos tulajdonságaihoz

```
<Window.Resources>
    <Style x:Key="DialogButtonStyle" TargetType="Button">
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="{x:Type Button}">
                    <Grid>
                        <Ellipse Fill="{TemplateBinding Background}" Stroke="{TemplateBinding BorderBrush}"/>
                        <ContentPresenter HorizontalAlignment="Center" VerticalAlignment="Center" />
                    </Grid>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>
</Window.Resources>
<Grid Margin="10">
        <Button Style="{StaticResource DialogButtonStyle}" BorderBrush="Green" Background="Red" Content="Button"
             Height="50"/>
</Grid>
</Window>
```



- lásd: Code/ControlTemplate példát

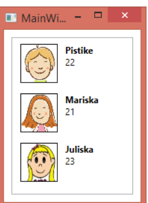
Adatsablonok - DataTemplate

- amit meghatározunk egy DataTemplate-ben, az lesz a vizuális szerkezete az adat objektumunknak
- a tartalommal (Content) bíró vezérlőknek (Button, TextBox) létezik egy ContentTemplate tulajdonsága, melynek egy ilyen adatsablont adhatunk értékül
- a lista vezérlőknek (ListBox, ComboBox) van ItemTemplate tulajdonsága. A listaelemek (egységes) megjelenítését testre szabhatjuk adatsablonokkal
- a ContentControl elemeknek lehet akármilyen tartalma, pl. egy Diák típus objektuma hozzárendelhető egy Button osztályhoz
- példa: ListView ItemTemplate-el

```
<Grid>
    <ListView Margin="10" Name="lvDataBinding">
        <ListView.ItemTemplate>
            <DataTemplate>
                <WrapPanel>
                    <TextBlock Text="Név: " FontSize="24"/>
                    <TextBlock Text="{Binding Nev}" FontSize="24" FontWeight="Bold" />
                    <TextBlock Text=", " />
                    <TextBlock Text="Életkor: "  FontSize="24"/>
                    <TextBlock Text="{Binding Eletkor}" FontSize="24" FontWeight="Bold" />
                    <Image Width="48" Margin="2,2,2,1" Source="{Binding Kep}" />
                </WrapPanel>
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</Grid>
```

ListView ItemTemplate-el

```
...
    InitializeComponent();
    List<Diak> students = new List<Diak>();
    students.Add(new Diak() { Nev = ”Pistike”, Eletkor = 22, Kep = ”images/fiu1.png” });
    students.Add(new Diak() { Nev = ”Mariska”, Eletkor = 21, Kep = ”images/lany1.png” });
    students.Add(new Diak() { Nev = ”Juliska”, Eletkor = 23, Kep = ”images/lany2.png” });
    lvDataBinding.ItemsSource = students;
...
    public class Diak
    {
        public string Nev { get; set; }
        public int Eletkor {get; set; }
        public string Kep { get; set; }
    }
```



lásd: Code/DataTemplate2 példát

Animációk

Animációk - Animation

- gyorsan egymás után vetített képek sorozata
- System.Windows.Media.Animation
- Lineáris animációk
  - egy függőségi tulajdonság értékét fokozatosan változtatja egy kezdő és a végpont között
  - formátuma: `<Típus>Animation`, ahol a `<Típus>` az animáció típusának a neve pl. `Color`, `Double` stb.
- Kulcskocka alapú animációk
  - használjuk ha több értéket kell megadjunk egy animációnak. A függőségi tulajdonság értéke tetszőlegesen megváltoztatható egy adott pillanatban (a kezdő és a vég érték nincs megkötve)
  - formátuma: `<Típus>AnimationUsingKeyFrames`, ahol a `<Típus>` az animáció típusának a neve pl. `String`, `Double` stb.
- Útvonal alapú animációk
  - az adott objektumot, egy általunk meghatározott útvonal követésével, mozgásra bírhatjuk
  - formátuma: `<Típus> AnimationUsingPath`, ahol a `<Típus>` az animáció típusának a neve pl. `Point`, `Double`, `Matrix` stb.

Az animációk főbb elemei

- `TimeLine` (időszalag) - meghatározza, hogy egy érték hogyan változik egy időszakaszon (hogy animáljunk egy double-t - DoubleAnimation osztályt használunk, (van Int32Animation ...)
- Storyboard (forgatókönyv) - animációk kombinálására használjuk
- Triggerek - animációk indításához és leállításához használjuk

Timeline tulajdonságok	**Leírás**
`AutoReverse`	True/False. Az animáció első lejátszás végére érve fordítva is lejátszódik. (visszatér az eredeti értékre)
`SpeedRatio`	A gyermek animáció lejátszási sebességét módosíthatjuk a szülő időszalaghoz viszonyítva. (Default 1. Ha >1 gyorsabb, < 1 lassubb)
`BeginTime`	Az animáció elindításának késleltetésére. Formátuma: óra:perc:másodperc.
`AccelerationRatio` `DecelerationRatio`	0.0 és 1.0 között fogad el értékeket, tulajdonképpen az animáció időtartamának egy bizonyos százaléka, ami alatt az animáció az elején gyorsul/végén lassul.
`Duration`	Az animáció időtartamát adhatjuk meg (óra:perc:másodperc formátumban), From, To, By.
`RepeatBehavior`	Az adott animáció hányszor ismétlődjön meg.
`FillBehavior`	Mi történjen, amikor az animáció véget ért. Két értéke lehet, a HoldEnd és a Stop (a szülő timeline-hoz képest) .

Példa

```
<Grid>
    <Rectangle Height="50" Width="100">
        <Rectangle.Fill>
            <SolidColorBrush x:Name="rectangleBrush" Color="Red" />
        </Rectangle.Fill>
        <Rectangle.Triggers>
            <EventTrigger RoutedEvent="Rectangle.Loaded" >
                <EventTrigger.Actions>
                    <BeginStoryboard>
                        <Storyboard Duration="00:00:06" RepeatBehavior="Forever">
                            <DoubleAnimation Storyboard.TargetProperty="(Rectangle.Width)" Duration="
                                0:0:3" AutoReverse="True" FillBehavior="Stop" RepeatBehavior="Forever"
                                From="100" To="300">
                            </DoubleAnimation>
                            <ColorAnimation Storyboard.TargetName="rectangleBrush" Storyboard.
                                TargetProperty="(SolidColorBrush.Color)" Duration="0:0:3" AutoReverse="
                                True" FillBehavior="Stop" RepeatBehavior="Forever" From="Red" To="Blue"
                                >
                            </ColorAnimation>
                        </Storyboard>
                    </BeginStoryboard>
                </EventTrigger.Actions>
            </EventTrigger>
        </Rectangle.Triggers>
    </Rectangle>
</Grid>
```

lásd: Code/Animation1 példát

Kulcskocka alapú animáció

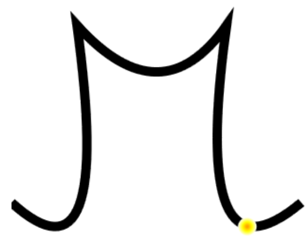
```
<Label Name="animLabel" Margin="20" FontFamily="Verdana">Kattints ide!
    <Label.Triggers>
        <EventTrigger RoutedEvent="Label.MouseDown">
            <BeginStoryboard>
            <Storyboard>
              <StringAnimationUsingKeyFrames Storyboard.TargetName="animLabel" Duration="0:0:9"
              Storyboard.TargetProperty="(Label.Content)" FillBehavior="HoldEnd">
                    <DiscreteStringKeyFrame Value="" KeyTime="0:0:0" />
                    <DiscreteStringKeyFrame Value="M" KeyTime="0:0:1" />
                    <DiscreteStringKeyFrame Value="Mi" KeyTime="0:0:1.5" />
                    <DiscreteStringKeyFrame Value="Min" KeyTime="0:0:2" />
                    <DiscreteStringKeyFrame Value="Mind" KeyTime="0:0:2.5" />
                    <DiscreteStringKeyFrame Value="Mindj" KeyTime="0:0:3" />
                    <DiscreteStringKeyFrame Value="Mindjá" KeyTime="0:0:3.5" />
                    <DiscreteStringKeyFrame Value="Mindjár" KeyTime="0:0:4" />
                    <DiscreteStringKeyFrame Value="Mindját" KeyTime="0:0:4.5" />
                    <DiscreteStringKeyFrame Value="Mindjárt " KeyTime="0:0:5" />
                    <DiscreteStringKeyFrame Value="Mindjárt j" KeyTime="0:0:5.5" />
                    <DiscreteStringKeyFrame Value="Mindjárt jö" KeyTime="0:0:6" />
                    <DiscreteStringKeyFrame Value="Mindjárt jöv" KeyTime="0:0:6.5" />
                    <DiscreteStringKeyFrame Value="Mindjárt jövö" KeyTime="0:0:7" />
                    <DiscreteStringKeyFrame Value="Mindjárt jövök" KeyTime="0:0:7.5" />
                    <DiscreteStringKeyFrame Value="Mindjárt jövök!" KeyTime="0:0:8" />
              </StringAnimationUsingKeyFrames>
            </Storyboard>
        </BeginStoryboard>
    </EventTrigger>
    </Label.Triggers>
</Label>
```

Mind Mindjárt jö

lásd: Code/Animation2 példát

Útvonal alapú animációk

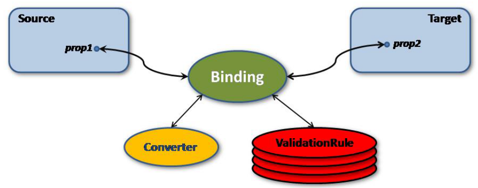
- az adott objektumot, egy általunk meghatározott útvonal követésével, mozgásra bírhatjuk
- formátuma: `<Típus> AnimationUsingPath`, ahol a `<Típus>` az animáció típusának a neve pl. `Point`, `Double`, `Matrix` stb.



Adatkötés - DataBinding

Adatkötés - DataBinding

- System.Windows.Data
- egy technika, amely összeköt két adat/információ forrást (forrás- és a célobjektum) és fenntartja az adatok szinkronizálását
- a célobjektum függőségi tulajdonság (**Dependency Property**) kell hogy legyen (később)
- ha megváltozik az adat UI reprezentációjában bármi, akkor ez automatikusan tükröződni fog az adaton is



- konverter, validálási szabályok opcionálisak
- példa:

A Binding osztály tulajdonságai

- szintaxis:

```
Tulajdonság="{Binding ElementName=Vezérlo, Path=Tulajdonság, Mode=Kapcsolási mód}"
```

Binding tulajdonságok	**Leírás**
`Source`	A forrásobjektum.
`ElementName`	Speciális esetre, mikor a forrásobjektum egy GUI elem.
`Path`	A forrás azon tulajdonságának az „elérési útvonala”, melyet kötni akarunk.
`OneWay Mode`	ha módosul az adatforrás módosul a felület is (default: Label, TextBlock)
`TwoWay Mode`	A cél frissül, amint a forrás módosul, illetve a forrás is frissül, amint a cél módosul.(default: TextBox, CheckBox)
`OneTime Mode`	A cél csak inicializáláskor kap értéket (a forrás alapján).
`Converter`	A konverter objektum.
`StringFormat`	Speciális formátum arra, hogyan legyen megjelenítve a tulajdonság értéke.
`ValidationRules`	A validálási szabályok.

Példák

- 1. a TextBox-ban mindig a Slider aktuális értéke jelenjen meg

```
<StackPanel Margin=”20”>
    <Slider x:Name=”csuszka1” Minimum=”0” Maximum=”100”/>
    <TextBox Text=”{Binding Value, ElementName=csuszka1, Mode=TwoWay}”/>
</StackPanel>
```

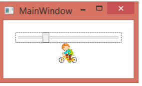
MainWindow

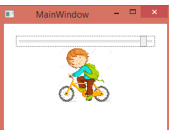
20.2127659574468

túlzottan pontos, Convert

- 2. a kép `mérete` mindig a Slider aktuális értéke szerint jelenjen meg

```
<StackPanel Margin=”20”>
    <Slider x:Name=”csuszka1” Minimum=”10” Maximum=”100”/>
    <Image  Source=”E:\Images\bicycle.jpg”
            Height=”{Binding ElementName=csuszka1, Path=Value}”
            Width=”{Binding ElementName=csuszka1, Path=Value}” />
</StackPanel>
```





NB: amennyiben helytelenül fogalmazzuk meg a kötést, nem fog kivételt dobni az alkalmazás!!! Minden kötési hiba a Visual Studio Output ablakában fog megjelenni, mint trace információ.
NB: ahhoz, hogy az `OneWay`, `TwoWay` megfelelően működjenek, szükség van a forrás objektumnak implementálni egy tulajdonság változás kezelő mechanizmust, pl. az INotifyPropertyChanged.

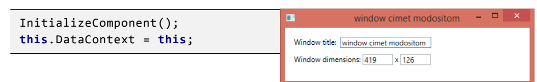
DataContext használata

- a DataContext tulajdonság az alapértelmezett forrása az adatkötésnek. Ha explicit nem adjuk meg a Binding source-ot, akkor a defualtként a DataContext-et fogja venni.
- használat: amikor minden gyermekelemet ugyanahhoz a forráshoz szeretnénk kötni (gyermekelem felül is írhatja). Ha nincs megadva, akkor a logikai fában a szülőtől örökli
- beállíthatjuk a DataContext-nek a `Window`-t is

```
<StackPanel Margin="15">
    <WrapPanel>
        <TextBlock Text="Window title:  " />
        <TextBox Text="{Binding Title, UpdateSourceTrigger=PropertyChanged}" Width="150" />
    </WrapPanel>
    <WrapPanel Margin="0,10,0,0">
        <TextBlock Text="Window dimensions: " />
        <TextBox Text="{Binding Width}" Width="50" />
        <TextBlock Text=" x " />
        <TextBox Text="{Binding Height}" Width="50" />
    </WrapPanel>
</StackPanel>
```

A Window lesz a DataContext:

```
InitializeComponent();
this.DataContext = this;
```



DataContext használata -2

Példa 2: Ez nem azt jelenti, hogy minden elemnek egyetlen DataContext-et kell használni. Egy Panelnek adhatunk más DataContext-et.

```
<StackPanel DataContext="{StaticResource myCustomer}">
    <TextBox Text="{Binding FirstName}"/>
    <TextBox Text="{Binding LastName}"/>
    <TextBox Text="{Binding Street}"/>
    <TextBox Text="{Binding City}"/>
</StackPanel>
```

Frissítési mechanizmusok az `UpdateSourceTrigger` segítségével

- `UpdateSourceTrigger`-el szabályozható, hogy a tulajdonság megváltozásakor (default), a fókusz elvesztésekor, vagy explicit módon akarjuk frissíteni a forrást
- az előbbi példából ha kivesszük a `UpdateSourceTrigger = PropertyChanged`-t, akkor a TextBox módosítás csak a fókusz elvesztésekor látszik a Window címnél
- 3 lehetőség:
  - `PropertyChanged` - default: minden tulajdonság, kivéve a Text tulajdonság
  - `LostFocus` - default: Text tulajdonság
  - `Explicit` - manuálisan
- példa `Explicit` módra:

```
<TextBox Name=" txtWindowTitle " Text="{Binding Title, UpdateSourceTrigger=Explicit}" Width="
    150"/>
<Button Name="btnUpdateSource" Click="btnUpdateSource_Click" Margin="5,0" Padding="5,0">
    ExplicitButton</Button>
```

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

- lásd: Code/Binding/UpdateSourceTrigger példát

RelativeSource Binding

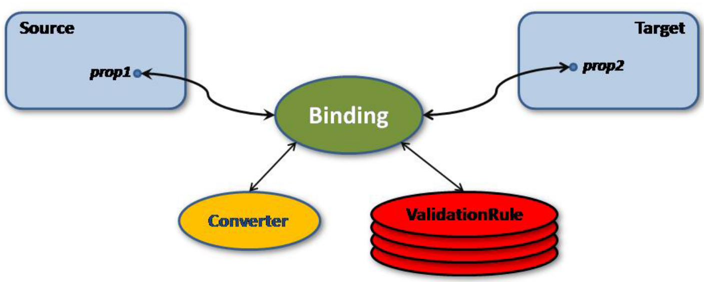
- saját objektumon lévő propertyhez köt
- akkor használjuk, ha meg tudjuk adni a binding forrás helyét - a binding cél helyéhez viszonyítva
- nem tudjuk a saját vagy másik abszolút pozícióját, de tudjuk, hogy ez az előző vagy a következő elem, két szint fölött van, vagy egy bizonyos típusú
- lehetőségek:
  - `Self` - segít, hogy forrás (önmaga) más tulajdonságához köthessünk
  - `TemplatedParent` - segít, hogy a sablonozott szülő objektum tulajdonságaihoz köthessünk
  - `PreviousData` - segít, hogy egy listában az előző item-hez köthessünk
  - `FindAncestor` - típus vagy mélység alapján keres szülő objektumot a logikai fában. Megadható opcionálisan egy level is, mégpedig az, hogy hányadik szintű ezen típusú objektumhoz kívánunk kötni
- Self:

```
<TextBlock Background="Yellow" Text="{Binding RelativeSource={RelativeSource Self},Path=ActualWidth}"/>
```

- FindAncestor:

```
<StackPanel Width="200">
    <StackPanel>
        <StackPanel>
            <TextBlock
                Text="{Binding Width,
                RelativeSource={RelativeSource FindAncestor,
                                               AncestorLevel=3,
                                               AncestorType={x:Type StackPanel}}}"/>

        </StackPanel>
    </StackPanel>
</StackPanel> <!--200-->
```

Konvertálás: Binding.Converter



- magától érthetődő konverziókról nem a programozónak kell gondoskodnia
  - előbbi példában a TextBox.Text tulajdonság `string` típusú, míg a Slider.Value `double`
- mi történik ha a `Boolean` és a `Visibility` tulajdonságokat szeretnénk összekötni??

Konvertálás: Binding.Converter

- ha két különböző típusú tulajdonságot szeretnénk összekötni akkor `ValueConverter` -t használunk
- adatcsere előtt az adat transzformálható a `Binding.Converter` beállításával
- `IValueConverter` -t implementáló osztály
- a konverter bemenő argumentumot, paramétert és még kultúra információt is kap
- A paraméter a kötésnél adható meg, értékei lehetnek:
  - `Convert` : az egyirányú adatkötéshez. Konvertálás a céltípusra, mindenképpen implementálni kell.
  - `ConvertBack` : a kétirányú adatkötéshez. Cél típus konvertálása a forrásra. Egyirányú kötések esetén nem kell megvalósítani.

Példa 1: Boolean to Visibility

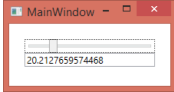
- pl: egy `Boolean` értéket akarunk kötni egy `Visibility` tulajdonsághoz

```
<StackPanel>
    <StackPanel.Resources>
        <BooleanToVisibilityConverter x:Key="boolToVis" />
    </StackPanel.Resources>

    <CheckBox x:Name="chkShowDetails" Content="Show Details" />
    <StackPanel x:Name="detailsPanel" Visibility="{Binding IsChecked,
                ElementName=chkShowDetails, Converter={StaticResource boolToVis}}">
    </StackPanel>
</StackPanel>
```

```
public class BooleanToVisibilityConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
        if (value is Boolean)
        {
            return ((bool)value) ? Visibility.Visible : Visibility.Collapsed;
        }

        return value;
    }

    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo
         culture)
    {
        throw new NotImplementedException();
    }
}
```

NB: Ilyen jellegű előre megírt konverterek léteznek a .NET Frameworkban

Konverter - Példa 2



- túlzottan pontos, alkítsuk át 3 tizedessé
- a konvertálóhoz tartozó kód a következő lesz:

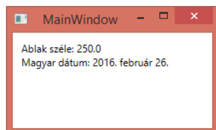
```
public object Convert(object value, Type targetType, object parameter, System.Globalization.
     CultureInfo culture)
{
    double cel = Math.Round((double)value, 3);
    return cel.ToString(culture);
}

public object ConvertBack(object value, Type targetType, object parameter, System.Globalization.
     CultureInfo culture)
{
     try
     {
           return System.Convert.ToDouble(value, culture);
     }
     catch (FormatException) { return null; }
}
```

- a `culture` paraméter azért kerül felhasználásra, mivel Magyar területi beállítások mellet pont helyett vessző választja el a tizedes jegyeket

Binding - egyebek

- ha csak azt akarjuk megváltoztatni, hogy egy bizonyos érték hogyan jelenjen meg, és nem feltétlenül más típussá alakítani, akkor a forrás adatot `StringFormat` -al formázzuk
  - {0}-val kell hivatkozni az adatra

```
<Window x:Class="stringFormat.MainWindow"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:system="clr-namespace:System;assembly=mscorlib"
      Title="MainWindow" Height="150" Width="250" Name="wnd">
    <StackPanel Margin="10">
      <TextBlock Text="{Binding ElementName = wnd, Path= ActualWidth, StringFormat = Ablak
          széle: {0:#,#.0}}" />
      <TextBlock Text="{Binding Source={x:Static system:DateTime.Now}, ConverterCulture='Hu-hu
          ', StringFormat = Magyar dátum: {0:D}}" />
    </StackPanel>
</Window>
```



Binding - egyebek -2

Helytelen binding nem dob kivételt!!! (a Visual Studio Output ablakában mint trace információ)

- `TargetNullValue`
  - ha a binding forrás vagy az elérési útban bármelyik elem null, akkor a `TargetNullValue` értéke adódik át a célnak

```
<TextBox Width="150" Text="{Binding Source={StaticResource object2}, Path=PropertyB,
    BindingGroupName=bindingGroup,
                        TargetNullValue='Oops, I am missing'}" />
```

- `FallbackValue`
  - amikor vmi miatt nem tudja meghatározni a binding értékét, akkor a `FallbackValue` értéke adódik át a célnak

```
<TextBox Text="{Binding EmployeeNames,TargetNullValue='Oops, I am missing',FallbackValue='I
    am default'}"/>
```

Adatkötés változáskövetéssel

Adatforrás változás jelzése

- ahhoz, hogy a cél tükrözze a forrás aktuális állapotát, követni kell az abban történő változásokat
- a forrásnak meg kell valósítania az `INotifyPropertyChanged` interfészt
- a megadott tulajdonság módosításakor kiválthatjuk a `NotifyPropertyChanged` eseményt, ami jelzi a felületnek, mely kötéseket kell frissíteni
- az esemény elküldi a megváltozott tulajdonság nevét, ha ezt nem adjuk meg, akkor az összes tulajdonság változását jelzi
- egyszerűsítésként felhasználhatjuk a `CallerMemberName` attribútumot, amely automatikusan behelyettesíti a hívó tag (tulajdonság) nevét

Adatkötés változáskövetéssel

- a változáskövetés teljes gyűjteményekre is alkalmazható, amennyiben a gyűjtemény megvalósítja az `INotifyCollectionChanged` interfészt
- az `ObservableCollection` típus már tartalmazza az interfészek megvalósítását, ezért alkalmas változó tartalmú gyűjtemények követésére

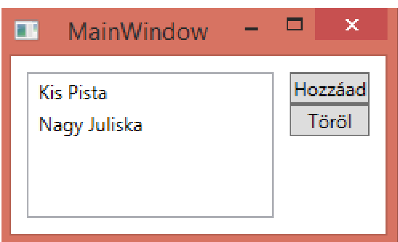
```
private ObservableCollection<User> users = new ObservableCollection<User>();
```

- amennyiben a gyűjtemény, vagy bármely tagjának tulajdonsága változik, azonnal megjelenik a változás

Példa. Miért?

- az UI nem változik a buttonok megnyomása után sem!!

```
<DockPanel Margin="10">
    <StackPanel DockPanel.Dock="Right" Margin="10,0,0,0">
        <Button Name="btnHozzaad" Click="btnHozzaad_Click">Hozzáad</Button>
        <Button Name="btnTorol" Click="btnTorol_Click">Töröl</Button>
    </StackPanel>
    <ListBox Name="lbUsers" DisplayMemberPath="Name"></ListBox>
</DockPanel>
```



Lásd: code/PropertyNotChanged

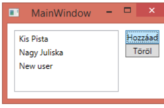
Háttérkód:

```
public partial class MainWindow : Window
{
    private List<User> users = new List<User>(); //figyeld a típusát

    public MainWindow()
    {
        InitializeComponent();
        users.Add(new User() { Name = ”Kis Pista” });
        users.Add(new User() { Name = ”Nagy Juliska” });

        lbUsers.ItemsSource = users;
    }
    private void btnHozzaad_Click(object sender, RoutedEventArgs e)
    {
        users.Add(new User() { Name = ”Uj emberke” });
    }
    private void btnTorol_Click(object sender, RoutedEventArgs e)
    {
        if (lbUsers.SelectedItem != null)
            users.Remove(lbUsers.SelectedItem as User);
    }
}
public class User
{
    public string Name { get; set; }
}
```

Adatkötés változáskövetéssel

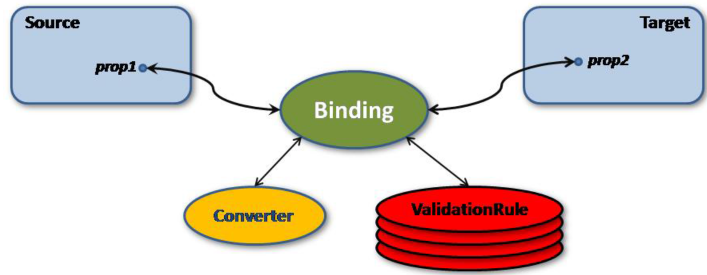
Példa:

```
...
private ObservableCollection<User> users = new ObservableCollection<User>(); //NB: NEM List
...
public class User : INotifyPropertyChanged
{
    private string name;
    public string Name {
                get { return name; }
                set {
                    if(name != value)
                    {
                        name = value;
                        NotifyPropertyChanged(”Name”); // jelezzük a változást
                    }}
                }
    public event PropertyChangedEventHandler PropertyChanged; // az esemény

    public void NotifyPropertyChanged(string propName)
    {
        if(this.PropertyChanged != null)
            this.PropertyChanged(this, new PropertyChangedEventArgs(propName)); //
                eseménykiváltás
    }
}}
// public void NotifyPropertyChanged([CallerMemberName] string propName = null)
// ha paraméter nélkül hívták meg, a hívó nevét helyettesíti be
```

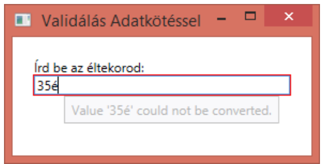


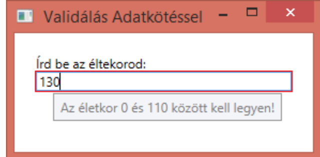
Adatkötés validálással

Validálás



- a validáció ún. ValidationRule-ok segítségével történik, ami az `adatkötési fázisban` ellenőrzi a bemenő információt, egy adott szabály függvényében
- ha az input érvényes, az adatkötés megtörténhet, egyébként pedig meg kell akadályozni
- a felhasználónak visszajelzést kell adni, hogy tudja érvénytelen adatot próbált meg betáplálni





Validáció

- a WPF többfajta validációt támogat, pl:
  - ValidationRules

```
Binding ValidatesOnExceptions = true
```

  - IDataErrorInfo interfész által implementált ellenőrzés

```
Binding ValidatesOnDataErrors = true
```

  - saját validációs szabály is készíthető

A validálás lépései

- I. lépés: textboxhoz hozzáadni `Binding ValidatesOnDataErrors=True`-t

```
<TextBox  Text=”{ Binding Path = Age,  Source= {StaticResource data},
          ValidatesOnDataErrors = True,
          UpdateSourceTrigger = PropertyChanged}”/>
```

Validálás -2

- II. lépés
  - a hivatkozott `Age` a felület mögött álló saját osztályunk egy tulajdonsága
  - az illető osztálynak kell implementálnia az `IDataErrorInfo` interfészt
  - az interfész deklarál egy `indexer` -t, mely a validálandó tulajdonság nevét kapja paraméterül, és melynek függvényében a tulajdonság értékét ellenőrizhetjük

```
public class Person : IDataErrorInfo
{
    public int Szam { get; set; }
    public int Age
        {
            get { return age; }
            set { age = value; }
        }
    public string Error
    { get{ return null;}    }

    public string this[string propertyname]
    {
        get{
        string result = null;
        if (propertyname == ”Age”)
        {
            if (this.age < 0 || this.age > 110)
            {
            result = ”Az életkor 0 és 110 között kell legyen!”;
            }
        }
        return result;
    } }
```

Validálás

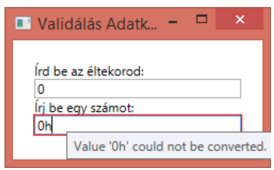
- III. lépés
  - II. lépés után a textbox-ok körül szépen látható a piros keret, de a hibaüzenet még nem jelenik meg
  - a textbox-hoz egy alapértelmezett `stílust` kell létrehozni, melyben egy `trigger` -el figyeljük a textbox `Validation.HasError` tulajdonságát, melynek bool értékéből megtudhatjuk, hogy vajon fellépett-e ilyen hiba
  - amennyiben hiba lépett fel, a kioldó beállítja a textbox `ToolTip` tulajdonságát a hibaüzenet szövegére adatkötés segítségével

```
...
xmlns:src="clr-namespace:Validation">
<Window.Resources>
    <src:Person x:Key="data"/>
    <Style x:Key="textBoxInError" 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>
</Window.Resources>
```

- a textboxhoz hozzáadjuk a stílust

```
 <TextBox Style="{StaticResource textBoxInError}"
           Text="{ Binding Path=Age,  Source={StaticResource data},
           ValidatesOnDataErrors=True, UpdateSourceTrigger=PropertyChanged  }"/>
```

Alapértelmezett validátor



- **NB:** a Szam tulajdonsághoz nem írtunk validáló kódot! Ilyen esetben a WPF – a tulajdonság típusától (int) függő – alapértelmezett validátora fog működésbe lépni
- megnézi, hogy a textbox nem-e üres, és int-re konvertálható-e az oda beírt szöveg

Validálás attribútumokkal - Annotation

- megadhatunk attribútumokat az adattagok előtt, amelyeket akkor használunk fel, amikor validáljuk az adatainkat
- System.ComponentModel.DataAnnotations névtér
- készíthetünk saját attribútumokat is

```
class Person : IDataErrorInfo, INotifyPropertyChanged
{
    private string _name;

    [StringLength(5, ErrorMessage = "Error Msg.")]
    [Required(ErrorMessage = "Name is required.")]
    [DisplayName("Name")]
    public string Name
     {
        get { return _name; }
        set
            {
             _name = value;
             NotifyPropertyChanged("Name");
            }
     }
```

- lásd Code/Validálás

Validálás attribútumokkal - Annotation

- az IDataErrorInfo interfész implementálása:

```
public string Error { get { return null; } }

public string this[string propertyName]
{
    get
    {
    string result = null;
    if (propertyName == ”Name”)
        return ValidateProperty(this.Name, propertyName);
    return result;
    }
}
protected string ValidateProperty(object value, string propertyName)
{
    var info = this.GetType().GetProperty(propertyName);
    IEnumerable<string> errorInfos =
              (from va in info.GetCustomAttributes(true).OfType<ValidationAttribute>()
               where !va.IsValid(value)
               select va.FormatErrorMessage(string.Empty)).ToList();

     if (errorInfos.Count() > 0)
     {
      return errorInfos.FirstOrDefault<string>();
      }
       return null;
}
```