WPF 2

Programowanie Wizualne

Paweł Wojciechowski

Instytut Informatyki, Politechniki Poznańskiej

2022

Oznaczenia slajdów

Slajd ważny

Slajd warty uwagi (bez oznaczenia)

Slajd dodatkowy

DependencyProperties

Podstawowe informacje

- *Dependency Property* stanowi rozszerzenie pojęcia właściwości z .NETa na potrzeby interfejsów użytkownika w WPF
- Jest to podstawowy mechanizm wykorzystywany w animacji elementów interfejsu, wiązaniu danych (*data binding*) oraz obsłudze styli
- DP umożliwiają:
  - informowanie o zmianie wartości (*change notification)*
  - dziedziczenie wartości właściwości
  - zmniejszają koszty przechowywania wartości

Tworzenie *Depencency Property*

- można je dodać tylko do obiektów dziedziczących po *DependencyObject*

```
public static readonly DependencyProperty AquariumGraphicProperty;
```

- obiekt DependencyProperty musi być zawsze statyczny i tylko do odczytu
- obiekt ten należy zarejestrować przed jakimkolwiek użyciem klasy dla której go definiujemy

```
static MainWindow()
{
    AquariumGraphicProperty = DependencyProperty.Register(
                                                "AquariumGraphic",
                                                typeof(Uri),
                                                typeof(MainWindow),
                                                new FrameworkPropertyMetadata(null,
                                                    FrameworkPropertyMetadataOptions.AffectsRender,
                                                    new PropertyChangedCallback(OnUriChanged))
                                                );
}
```

Tworzenie *Dependency Property* (2)

- Rejestrując DP należy podać:
  - jej nazwę
  - typ danych używany przez tę właściwość
  - typ danych, który będzie taką właściwość posiadał
  - opcjonalnie: obiekt *FrameworkPropertyMetadata* z dodatkowymi ustawieniami
  - opcjonalnie: funkcję zwrotną do walidacji wartości
- Ustawienia *FrameworkPropertyMetadata*:

  AffectsArrange,AffectsMeasure, AffectsParrentArrange, AffectsParentMeasure, AffectsRender, BindsTwoWayByDefault, Inherits, IsAnimationProhibited, IsNotDataBindable, DefaultValue, CoerceValueCallback, PropertyChangedCallback

Tworzenie *Dependency Property* (3)

- dodanie *wrapper*-a właściwości

```
public Uri AquariumGraphic
{
    get { return (Uri)GetValue(AquariumGraphicProperty); }
    set { SetValue(AquariumGraphicProperty, value); }
}
```

- walidacja danych nie następuje w setterach/getterach!
- właściwości mogą być współdzielone

```
TextBlock.FontFamilyProperty = TextElement.FontFamilyProperty.AddOwner(typeof(TextBlock));
```

- Właściwości mogą być powiązane – użycie `RegisterAttached()`
- Usuwanie wartości właściwości:

```
Window1.ClearValue(MainWindow.AquariumGraphicProperty);
```

Wykorzystanie DP przez WPF
informowanie o zmianach (*change notification*)

- DP nie uruchamia automatycznie zdarzeń w momencie zmiany wartości
- Uruchamia metodę `OnPropertyChangedCallback()`, która to przekazuje informację o zmianie wartości do dwóch serwisów: wiązania danych i triggerów – chcąc reagować na zmianę wartości można albo utworzyć wiązanie albo napisać trigger
- Niektóre komponenty udostępniają zdarzenia po zmianie wartości np. `ScrollBar – ValueChanged`

Wykorzystanie DP przez WPF
wyznaczanie wartości (*dynamic value resolution)*

- Określenie bazowej wartości właściwości odbywa się na podstawie:
  - Wartości domyślnej,
  - Wartości dziedziczonej,
  - Wartości dla styli
  - lokalnej wartości
- Następnie wartość końcowa określana jest w następujący sposób:
  - Jeżeli wartość opisana jest jakimś wyrażeniem – wyznaczenie jej wartości (data binding i zasoby)
  - Jeżeli wartość jest celem animacji – zastosowanie animacji
  - Wywołanie metody `CoerceValueCallback` w celu „naprawienia” wartości

Walidacja wartości

- Ustawienie wartości DP przebiega następująco:
  - Dostarczana wartość trafia do metody `CoerceValueCallback`, gdzie może zostać zmodyfikowana. Jeśli zmiana jest nieakceptowalna zwracana jest wartość `DependencyProperty.UnsetValue`
  - Następnie uruchamiana jest metoda `ValidateValueCallback`, która zwraca wartość `true` jeśli wartość jest akceptowana. Metoda ta nie ma dostępu do pozostałych właściwości obiektu.
  - Jeśli poprzednie kroki zakończą się sukcesem, zostaje wywołana metoda `PropertyChangedCallback`
- `ValidateValueCallback` musi być statyczna i jest dostarczana jako opcjonalna na etapie rejestracji DP
- `CoerceValueCallback ma następujący nagłówek`

```
private static object CoerceMaximum( DependencyObject d, object value);
```

Przykład działania walidacji

```
ScrollBar bar = new ScrollBar();
bar.Value = 100;
//bar.Value = 1
bar.Minimum = 1;
//bar.Value = 1
bar.Maximum = 200;
//bar.Value = 100
```

Zdarzenia

Model zdarzeń

- zdarzenia (ang. *events)*, czyli wiadomości wysłane przez obiekty informujące o istotnych zmianach
- model zdarzeń WPF zmianie znaczenie zdarzeń .NET - nazywa się on *routed event*
- w strukturze hierarchicznej który z komponentów powinien obsłużyć zdarzenie np. kliknięcia przyciskiem myszy?
- zdarzenie wędruje wzdłuż hierarchii, co pozwala obsłużyć je w najbardziej dogodnym miejscu
- w WPFie 4 dodano obsługę zdarzeń związanych z urządzeniami dotykowymi (*multitouch*)

Dodanie/usunięcie metody obsługi zdarzenia

- XAML

```
<Button Name="button1" Click="button1_Click">Button 1</Button>
```

- Kod

```
button1.Click += new RoutedEventHandler(button1_Click);

button1.Click += button1_Click;
```

- UIElement.AddHandler()

```
button1.AddHandler(ButtonBase.ClickEvent, new RoutedEventHandler(button1_Click));
```

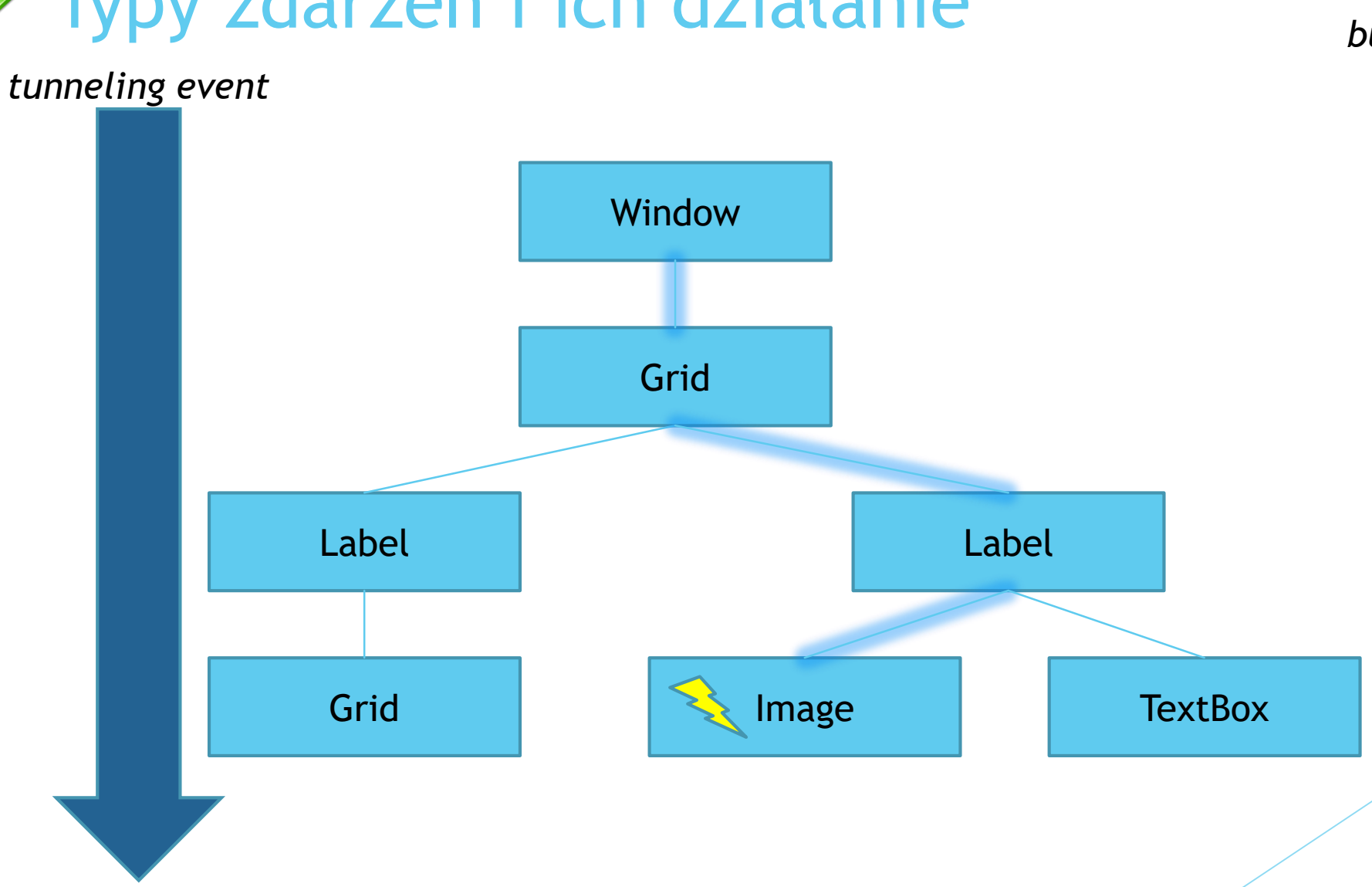
- Usunięcie metody:

```
button1.Click -= button1_Click;

button1.RemoveHandler(ButtonBase.ClickEvent, new RoutedEventHandler(button1_Click));
```

Rodzaje zdarzeń

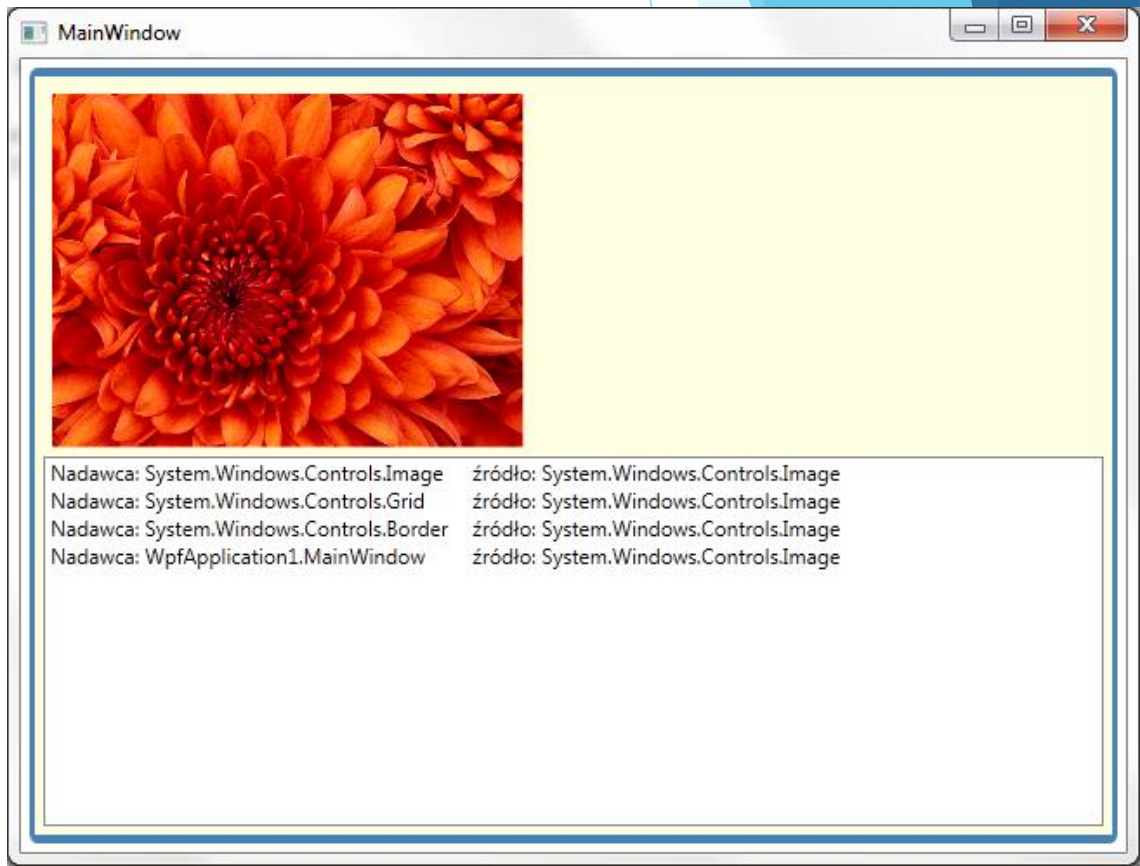
- bezpośrednie (*direct events)* - trafiają do jednego obiektu i nie wędrują po hierarchii komponentów – `Click, MouseEnter`
- „bulgoczące” (*bubbling events)* - podróżują w górę hierarchii np. `MouseDown`
- tunelowane (*tunneling events*) – podróżują w dół hierarchii np. `PreviewMouseDown`

Typy zdarzeń i ich działanie



Przykład zdarzenia *bubbling*

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:WpfApplication1"
        Title="MainWindow" Name="Window1" MouseDown="MouseDownHandler">
    <Border Margin="5" Padding="5" Background="LightYellow" BorderBrush="SteelBlue„
            BorderThickness="3,5,3,5" CornerRadius="3" MouseDown="MouseDownHandler" >

        <Grid MouseDown="MouseDownHandler">
            ...
            <Label Grid.Row="0" Grid.Column="0" VerticalAlignment="Center">
                <Image Source="/WpfApplication1;component/Images/Chrysanthemum.jpg"
                       MouseDown="MouseDownHandler"/>
            </Label>
            <ListBox Name="listBox" Grid.Row="1" Grid.Column="0" />
        </Grid>
    </Border>
</Window>
```



Klasa RoutedEventArgs

- metoda obsługi zdarzenia:

```
private void MouseDownHandler(object sender, RoutedEventArgs rea)
{
    string message = "Nadawca: " + sender.ToString() + "\t źródło: " + rea.Source;
    listBox.Items.Add(message);
}
```

- klasa RoutedEventArgs ma następujące właściwości:
  - Source – obiekt który wywołał zdarzenie (tutaj Image)
  - OriginalSource – najczęściej to samo co wyżej, ale ma zastosowanie w pewnych przypadkach
  - RoutedEvent – zdarzenie, które zostało wywołane
  - Handled – ustawienie wartości na `true` pozwala zatrzymać proces tunelowania lub … (bulgotania ;))

Zdarzenia tunelowane

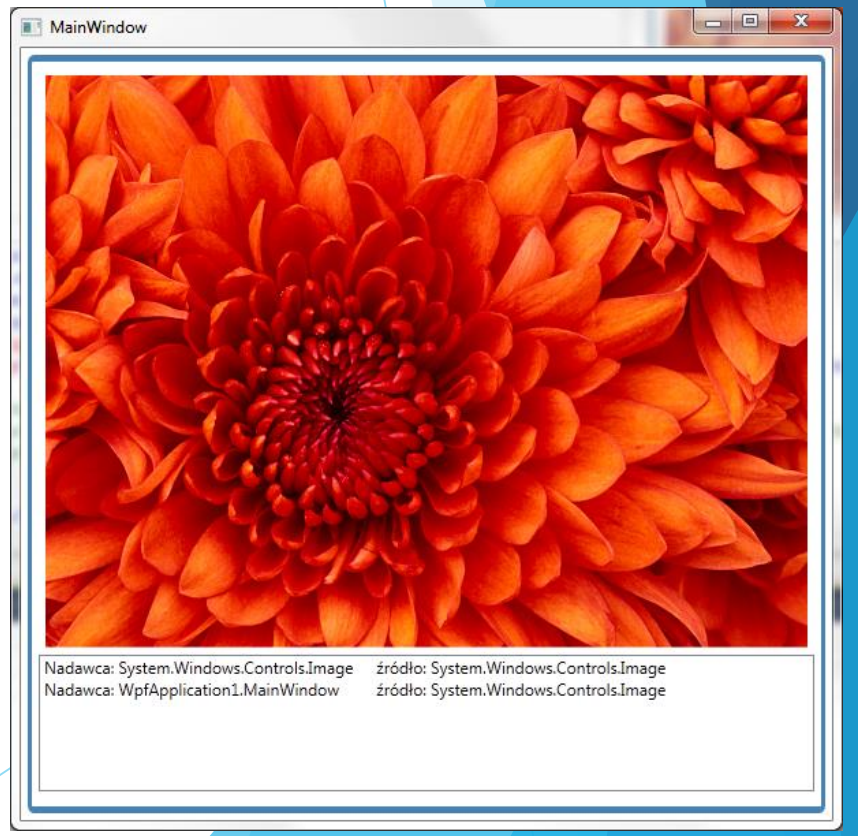
- Łatwo rozpoznawalne – przedrostek `Preview`
- W WPF-ie oba rodzaje zdarzeń są łączone w pary `MouseDown – PreviewMouseDown`
- obłużone zdarzenie tunelowe (`Handled = true`) powoduje brak wywołania zdarzenia *bubbling!* - współdzielenie obiektu `RoutedEventArgs`

Obsługa obsłużonych zdarzeń

- ustawienie parametrów zdarzenia `Handled` na `true` powoduje zatrzymania procesu jego przekazywania
- istnieje jednak możliwość wymuszenia obsługi również obsłużonych zdarzeń

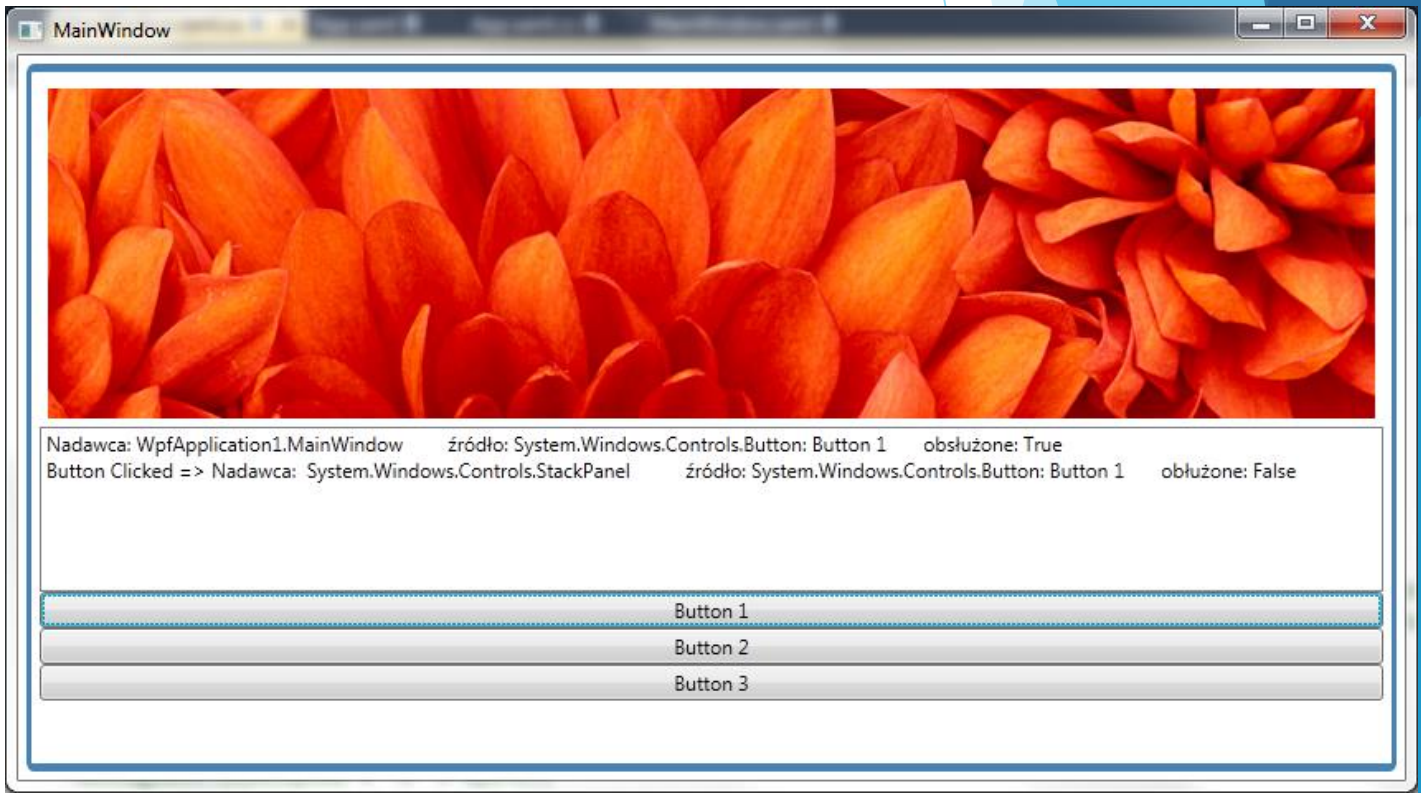
```
Window1.AddHandler(MouseDownEvent,
                   new MouseButtonEventHandler(MouseDownHandler),
                   true);
```

- pytanie: czy takie postępowanie ma sens?



Zdarzenia łączone (*attached*)

```
<StackPanel Name="stackPanel" Button.Click="button1_Click">
    <Button Name="button1">Button 1</Button>
    <Button Name="button2">Button 2</Button>
    <Button Name="button3">Button 3</Button>
</StackPanel>
```



Definiowanie zdarzenia

- Definiowanie zdarzenia wygląda podobnie do definiowania *dependency properties*

```
private static readonly RoutedEvent MyEvent;
```

- rejestracja zdarzenia:

```
MyEvent = EventManager.RegisterRoutedEvent( "MyEvent",                  //nazwa zdarzenia
                                            RoutingStrategy.Bubble,     //rodzaj
                                            typeof(RoutedEventHandler), //delegat metody obsługi
                                            typeof(MainWindow));        //właściciel
```

- *Wrapper:*

```
public event RoutedEventHandler My
{
    add { base.AddHandler(MyEvent, value);}
    remove { base.RemoveHandler(MyEvent, value);}
}
```

Definiowanie zdarzeń (2)

- Zdarzenie współdzielone pomiędzy klasami

```
MyEvent = Mouse.MouseDownEvent.AddOwner(typeof(MainWindow));
```

- Wywołanie zdarzenia

```
RoutedEventArgs rea = new RoutedEventArgs(Mouse.MouseDownEvent, this);
base.RaiseEvent(rea);
```

- spowoduje to wywołanie zdarzenia dla wszystkich metod zarejestrowanych metodą `AddHandler()`.
- konwencja jest taka, że pierwszym parametrem metody obsługi zdarzenia jest obiekt, który to zdarzenie wywołał, a drugim jest obiekt `EventArgs` z dodatkowymi informacjami.

```
private void button1_Click(object sender, RoutedEventArgs e)
{
}
```

Zdarzenia zdefiniowane w WPFie

- Można wyróżnić następujące rodzaje zdarzeń:
  - związane z cyklem życia obiektu (*lifetime)*
  - zdarzenia wejściowe:
    - myszy (*mouse)*
    - klawiatury (*keyboard)*
    - rysika (*stylus)*
    - *multitouch*

Zdarzenia cylku obiektów (*lifetime events*)

- `Initialized` – zostaje wywołane w momencie, gdy obiekt jest już zainicjalizowany i jego właściwości ustawione w XAML zostały nadane (ale nie style)
- `Loaded` – zostaje wywołane, gdy całe okno zostało już zainicjalizowane oraz zostały zaaplikowane style i wiązania danych
- `Unloaded` – wywołane gdy komponent zostaje zwolniony

- kolejność inicjalizowania obiektów jest od najbardziej wewnętrznego do zewnętrznych

Cykl życia obiektu klasy `Window`

Dodatkowe zdarzenia dla okna:

- `SourceInitialized` – wywoływane gdy uchwyt okna (HwndSource) zostaje ustawiony
- `ContentRendered` – okno zostało wyrenderowane po raz pierwszy
- `Activated` – gdy okno staje się aktywne (w wyniku przełączania między oknami)
- `Deactivated` – analogicznie jak wyżej
- `Closing` – wywoływane, gdy okno jest zamykane. Jest to ostatni moment, żeby anulować jego zamknięcie (`CancelEventArgs.Cancel = true`)
- `Closed` – wywołane po tym, gdy okno jest już zamknięte (elementy są wciąż dostępne – przed wywołaniem `Unloaded`)

Zdarzenia obsługi klawiatury

Zdarzenia pojawiają się w następującej kolejności:

- PreviewKeyDown
- KeyDown
- PreviewTextInput
- TextInput
- PreviewKeyUp
- KeyUp

Stosowanie niskopoziomowej obsługi zalecana jest tylko w specjalnych przypadkach.

```
private void TextBox_KeyDown(object sender, KeyEventArgs e)
{
    KeyConverter converter = new KeyConverter();
    MessageBox.Show( "Key: " + converter.ConvertToString(e.Key));
}
```

Klasa KeyEventArgs

Właściwości:

- Handled
- Key
- IsDown – sprawdzenie, czy klawisz Key jest wciśnięty
- IsUp
- IsRepeat
- IsToggled
- KeyStates
- KeyboardDevice:
  - IsKeyDown()
  - IsKeyUp()
  - IsKeyToggled()
  - GetKeyStates()

Klasa Keyboard

Focus

- Komponent może otrzymać *focus* jeżeli ma ustawioną właściwość `Focusable` (domyśnie `true` dla elementów kontrolnych)
- Kolejność w jakiej elementy otrzymują *focus* zależy od:
  - ustawień właściwości `TabIndex` (domyślnie `Int32.MaxValue`)
  - hierarchii obiektów w interfejsie
- Właściwość `IsTabStop` mówi czy element będzie brany pod uwagę w sekwencji przechodzenia *focus'a* klawiszem Tab, ale wartość `false` nie oznacza, że element nie może otrzymać go w inny sposób (programistycznie lub za pomocą myszki)
- Elementy wyłączone (`IsEnabled=false`) i niewidoczne (`Visibility=false`) nie otrzymują *focus'a* bez względu na wartości właściwości: `Focusable`, `IsTabStop` i `TabIndex`

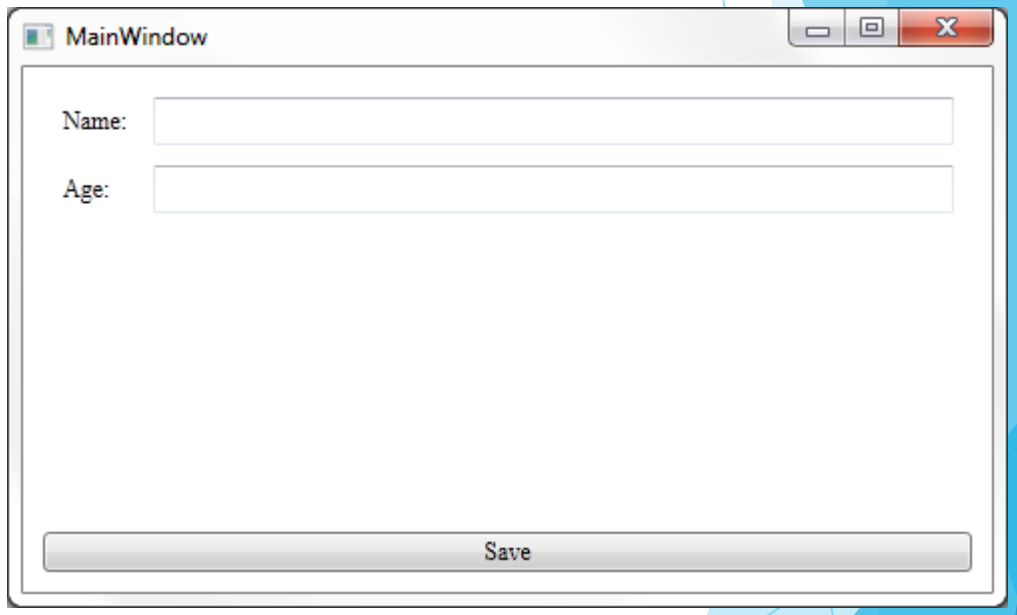
Wiązanie danych

Data binding

Ręczna synchronizacja danych – bez wiązania

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

}
```

```
<TextBox Name="txtName"
         TextChanged="txtName_TextChanged">
</TextBox>

<TextBox Name="txtAge"
         TextChanged="txtName_TextChanged">
</TextBox>

<Button Name="SaveButton"
        Click="SaveButton_Click">Save
</Button>
```

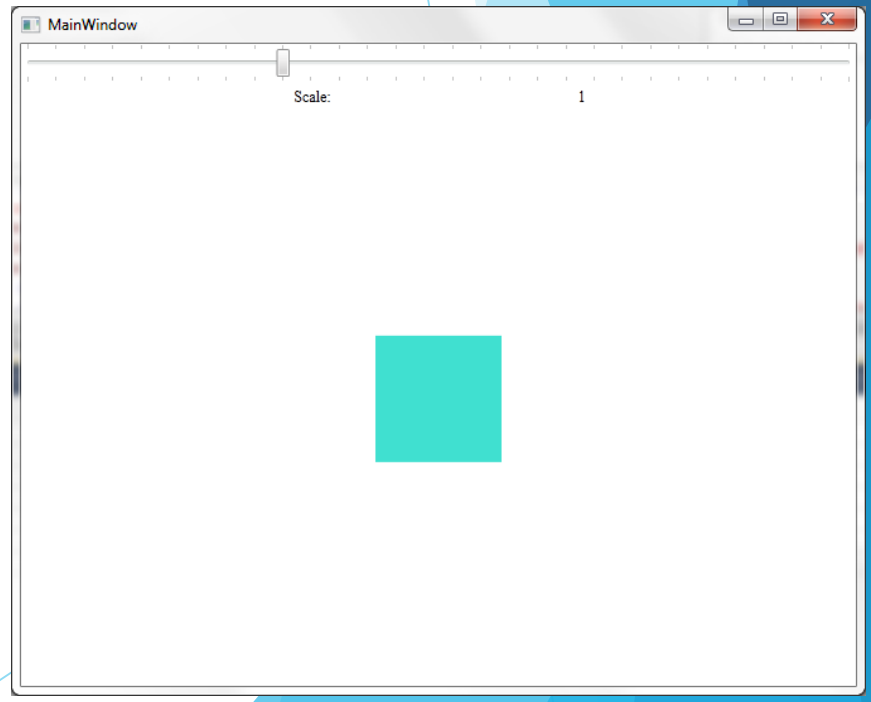


Wiązanie danych

- jest to związek, który mówi skąd dane powinny zostać pobrane (obiekt źródłowy) aby ustawić właściwość w obiekcie docelowym
- właściwość docelowa jest zawsze *dependency property*
- najprostszym zastosowaniem jest wiązanie dwóch takich właściwości
- realizacja wiązania poprzez klasę `Binding` z pakietu `System.Windows.Data`
- niepoprawne wiązanie nie generuje wyjątków

Proste wiązanie właściwości

```
<Slider Name="ScaleSlider" Minimum="0.1" Maximum="3" TickFrequency="0.1"
        IsSnapToTickEnabled="True" Value="1"/>

<TextBox Name="Scaletxt"
         Text="{Binding ElementName=ScaleSlider,Path=Value}" />

<Rectangle Name="rect" Width="100" Height="100"
           HorizontalAlignment="Center"
           VerticalAlignment="Center"
           Fill="Turquoise">

    <Rectangle.RenderTransform>
        <ScaleTransform CenterX="50" CenterY="50"
              ScaleX="{Binding ElementName=ScaleSlider,Path=Value}"
              ScaleY="{Binding ElementName=ScaleSlider,Path=Value}"/>
    </Rectangle.RenderTransform>
</Rectangle>
```




```
Binding binding = new Binding();
binding.Source = ScaleSlider;
binding.Path = new PropertyPath("Value");
binding.Mode = BindingMode.TwoWay;
Scaletxt.SetBinding(TextBox.TextProperty, binding);
```

Tryby wiązania

„Ręczne” ustawienie wartości - zerwanie wiązania!

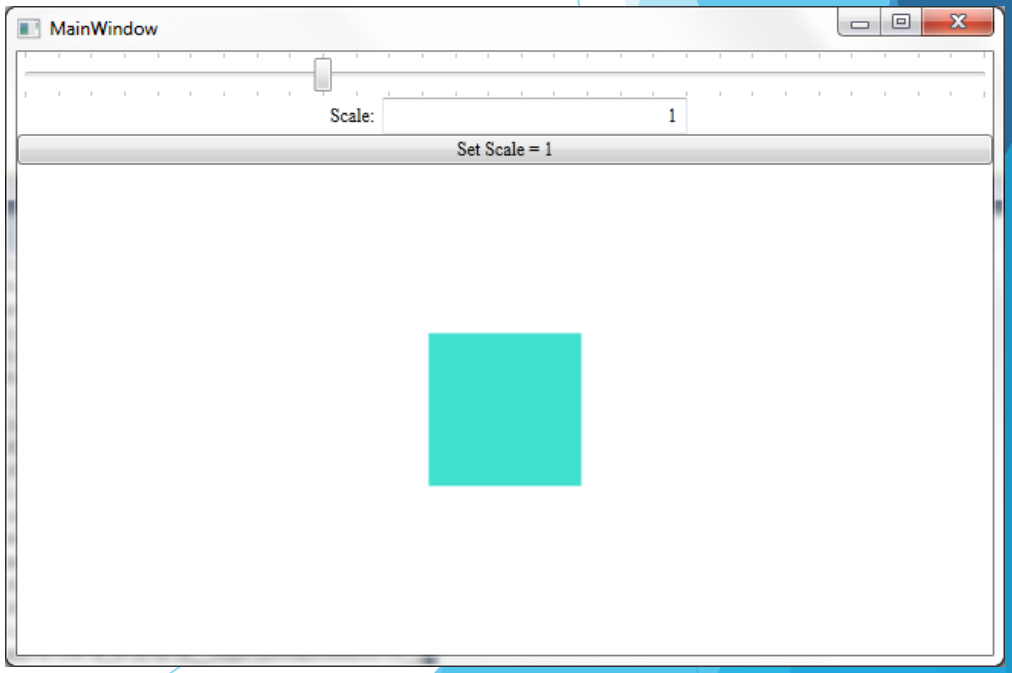
```
private void SetScaleButton_Click(object sender, RoutedEventArgs e)
{
      ScaleTransform st = (ScaleTransform) rect.RenderTransform;
      st.ScaleX = 1;
      st.ScaleY = 1;
}
```

Ustawienie wiązania dwukierunkowego:

```
    ScaleX="{Binding ElementName=ScaleSlider,Path=Value,Mode=TwoWay}"
    ScaleY="{Binding ElementName=ScaleSlider,Path=Value,Mode=TwoWay}"
```

A co będzie, gdy dane będą sprzeczne?

```
private void SetScaleButton_Click(object sender, RoutedEventArgs e)
{
      ScaleTransform st = (ScaleTransform) rect.RenderTransform;
      st.ScaleX = 1;
      st.ScaleY = 2;
}
```



Tryby wiązania

`BindingMode`

- `OneWay` - jednostronne od źródła do celu
- `TwoWay` - dwustronne
- `OneTime` – jednorazowe – w celu inicjalizacji
- `OneWayToSource` – jednostronne od celu do źródła
- `Default` - domyślne – zależne od właściwości

Usunięcie wiązania

- usunięcie pojedynczego wiązania:

```
BindingOperations.ClearBinding(ScaleSlider, Slider.ValueProperty);
```

- usunięcie wszystkich wiązań:

```
BindingOperations.ClearAllBindings(ScaleSlider);
```

- inne zastosowanie klasy `BindingOperations`

```
Binding binding = new Binding();
binding.Source = ScaleSlider;
binding.Path = new PropertyPath("Value");
binding.Mode = BindingMode.TwoWay;
ScaleTransform st = rect.RenderTransform as ScaleTransform;
BindingOperations.SetBinding(st, ScaleTransform.ScaleXProperty, binding);
```

Aktualizacja powiązanych danych

- zmiana wskazania Slider-a automatycznie przenosi się na zmianę rozmiaru prostokąta, podczas gdy wpisanie skali jest uwzględniane dopiero po przeniesieniu *focus*a na inny element.
- O tym kiedy nastąpi aktualizacja decyduje właściwość `Binding.UpdateSourceTrigger` – (jest to typ wyliczeniowy):
  - `PropertyChanged` – aktualizacja natychmiastowa
  - `LostFocus` – po utracie *focusa*
  - `Explicit` – brak aktualizacji do momentu wywołania `BindingExpression.UpdateSource();`
  - `Default` – zależy od ustawień właściwości, a dokładniej od wartości `FrameworkPropertyMetadata.DefaultUpdateSourceTrigger`

Aktualizacja powiązanych danych (2)

```
<TextBox Name="Scaletxt"
         Text="{Binding ElementName=ScaleSlider,Path=Value,UpdateSourceTrigger=PropertyChanged}" />
```

```
<TextBox Name="Scaletxt"
         Text="{Binding ElementName=ScaleSlider,Path=Value,UpdateSourceTrigger=Explicit}" />

BindingExpression be = Scaletxt.GetBindingExpression(TextBox.TextProperty);
be.UpdateSource();
```

Wiązanie do obiektów nie będących elementami wizualnymi

- dane muszą być publicznymi właściwościami obiektu
- dopuszczalne są następujące rozwiązania (zamiast `Binding.ElementName`):
  - Source – wiązanie do obiektu
  - RelativeSource – pozwala wiązać właściwości do właściwości obiektów w hierarchii komponentów
  - DataContext – wiązanie do właściwości `DataContext` komponentu – źródłem zostaje pierwszy element w hierarchii który ma wartość różną od `null`

Wiązanie typu *Source*

```
namespace WpfApplication1
{
    public partial class MainWindow : Window
    {
        public static double DefaultScale {get; set;}

        static MainWindow()
        {
            DefaultScale = 1.0;
        }
    }
}
```

```
 <Window x:Class="WpfApplication1.MainWindow"
         xmlns:local="clr-namespace:WpfApplication1"
         ... >
       ...
      <Slider Name="ScaleSlider"
           Value="{Binding Source={x:Static local:MainWindow.DefaultScale},Mode=OneWay}" />
       ...
```

Wiązania typu *Source*

```
<Window x:Class="WpfApplication1.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"
        xmlns:local="clr-namespace:WpfApplication1"
        Title="MainWindow" >

    <Window.Resources>
        <sys:Double x:Key="DefaultNameX">1.0</sys:Double>
    </Window.Resources>

    ...
    <Slider Name="ScaleSlider" Value="{Binding Source={StaticResource DefaultNameX},Mode=OneWay}" />
```

Wiązania typu RelativeSource

- Wiązanie do siebie samego albo elementu nadrzędnego na nieznanym poziomie w hierarchii
- Wiązanie tego typu wykorzystuje obiekt klasy RelativeSource

```
<TextBlock Text="{Binding
                    RelativeSource={RelativeSource Mode=FindAncestor,AncestorType=Window},
                    Path=Title}"/>
```

Wiązania typu RelativeSource (2)

Tryb wyszukiwania:

- Self – wiązanie do innej właściwości tego samego obiektu
- FindAncestor – wiązanie do rodzica. Podaje się AncestorType – czyli jakiego typu rodzica szukamy, a następnie opcjonalnie AncestorLevel w celu pominięcia zadanej liczby wystąpień tego typu
- PreviousData – wiązanie do poprzedniej wartości w powiązanej liście
- TemplatedParent

Wiązania typu DataContext

```
<Window.Resources>
    <local:Person x:Key="JurekPerson" Name="Jurek Owciak" Age="44"/>
</Window.Resources>


 <StackPanel>
     <TextBlock Text="{Binding Source={StaticResource JurekPerson},Path=Name}"/>
     <TextBlock Text="{Binding Source={StaticResource JurekPerson},Path=Age}"/>
 </StackPanel>

<StackPanel DataContext="{Binding Source={StaticResource JurekPerson}}">
    <TextBlock Text="{Binding Path=Name}"/>
    <TextBlock Text="{Binding Age}"/>
</StackPanel>
```

Dane są wyszukiwane w hierarchii komponentów do czasu napotkania właściwości `DataContext` różnej od `null`.

Konwersja powiązanych danych

- Można wyróżnić dwa rodzaje konwersji:
  - **formatowanie tekstów** – gdy dane są w postaci tekstowej – zazwyczaj zawierają liczby lub daty
  - **konwertery wartości** (*value converters*) – pozwalające na konwersję danych dowolnego typu

```
<Window …
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        …>

<TextBlock Text="{Binding Source={x:Static sys:DateTime.Now},StringFormat=Data: {0:yyyy-MMMM-dd}}"/>
```

Konwertery wartości

```
namespace WpfApplication1
{
    public class ValueToColorConverter:IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)
        {
            double val = (double)value;
            if (val < 1.0)
            {
                return new SolidColorBrush(Colors.Red);
            }
            else
            {
                return new SolidColorBrush(Colors.Black);
            }
        }

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

```
<Window.Resources>
    <local:ValueToColorConverter x:Key="ColorConverter"/>
</Window.Resources>

<Button Foreground="{Binding ElementName=ScaleSlider,Path=Value,
                             Converter={StaticResource ColorConverter}}" >Set Scale = 1
</Button>
```

Konwertery wartości (2)

```
public class ValueToColorConverter:IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)
    {

        double val = (double)value;
        if (val < (double) parameter)
        {
            return new SolidColorBrush(Colors.Red);
        }
        else
        {
            return new SolidColorBrush(Colors.Black);
        }
    }

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

Konwertery wartości (3)

```
<Button Name="SetScaleButton" Click="SetScaleButton_Click"  >
     <Button.Foreground>
          <Binding ElementName="ScaleSlider" Path="Value" Converter="{StaticResource ColorConverter}">

               <Binding.ConverterParameter>
                    <sys:Double>
                        0.5
                    </sys:Double>
               </Binding.ConverterParameter>

          </Binding>
      </Button.Foreground>
      Set Scale = 1
</Button>
```

Konwertery wartości (4)

```
public class ValueToColorConverter:IValueConverter
{
    public double MinWarningThreshold { get; set; }
    public double MaxWarningThreshold { get; set; }

    public object Convert(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)
    {

        double val = (double)value;
        if (val < MinWarningThreshold || val > MaxWarningThreshold)
        {
            return new SolidColorBrush(Colors.Red);
        }
        else
        {
            return new SolidColorBrush(Colors.Black);
        }
    }

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

Konwertery wartości (4)

```
<Button Name="SetScaleButton" Click="SetScaleButton_Click"  >
     <Button.Foreground>
          <Binding ElementName="ScaleSlider" Path="Value">
              <Binding.Converter>
                  <local:ValueToColorConverter
                         MinWarningThreshold="0.5"
                         MaxWarningThreshold="1.5"/>
              </Binding.Converter>

          </Binding>
     </Button.Foreground>
     Set Scale = 1
</Button>
```

Konwertery wielu wiązań (*multibinding)*

```
public class ValueToColorConverter:IMultiValueConverter
   {
       public double MinWarningThreshold { get; set; }
       public double MaxWarningThreshold { get; set; }

       public object Convert(object[] values, Type targetType, object parameter, System.Globalization.CultureInfo culture)
       {
           bool isChecked = (bool)values[0];
           double val = (double)values[1];
           if ( isChecked && ( val < MinWarningThreshold || val > MaxWarningThreshold))
           {
               return new SolidColorBrush(Colors.Red);
           }
           else
           {
               return new SolidColorBrush(Colors.Black);
           }

       }

       public object[] ConvertBack(object value, Type[] targetTypes, object parameter, System.Globalization.CultureInfo culture)
       {
           throw new NotImplementedException();
       }
   }
```

Konwertery wielu wiązań (*multibinding)*

```
<Button Name="SetScaleButton" Click="SetScaleButton_Click"  >
     <Button.Foreground>
         <MultiBinding >
             <MultiBinding.Converter>
                 <local:ValueToColorConverter MinWarningThreshold="0.5" MaxWarningThreshold="1.5"/>
             </MultiBinding.Converter>
             <Binding ElementName="IsScaleWarningEnabled" Path="IsChecked"/>
             <Binding ElementName="ScaleSlider" Path="Value"/>
         </MultiBinding>
     </Button.Foreground>

     Set Scale = 1</Button>

  <CheckBox Name="IsScaleWarningEnabled"  Content="scale warning"/>
```

Sprawdzanie poprawności danych (*Validation*)

- domyślnie, WPF ignoruje wyjątki związane z ustawianiem danych (*settery*)
- Sprawdzanie może odbywać się na dwóch poziomach:
  - danych (*data objects*) – wychwytywanie wyjątków lub implementacja interfejsów `INotifyDataErrorInfo` lub `IDataErrorInfo`
  - walidacja danych na poziomie wiązania danych

```
public class Person
{
    private int _age;
    public string Name { get; set; }
    public int Age
    {
        get { return _age; }
        set
        {
            if (value < 0)
            {
                throw new ArgumentException("Age cannot be negative.");
            }
            else
            {
                _age = value;
            }
        }
    }
}
```

Sprawdzanie poprawności danych (2)
ExceptionValidationRule

```
<TextBox>
    <TextBox.Text>
        <Binding Path="Director.Age" ElementName="Window1">
            <Binding.ValidationRules>
                <ExceptionValidationRule></ExceptionValidationRule>
            </Binding.ValidationRules>
        </Binding>
    </TextBox.Text>
</TextBox>
```

Gdy wystąpi wyjątek przy próbie ustawienia wartości to:

- pojawi się automatycznie czerwona ramka wokół elementu
- właściwość wiązana `Validation.HasError` ustawiana jest na wartość `true`
- Tworzony jest obiekt `ValidationError` i zostaje dodany do kolekcji `Validation.Errors`
- Jeśli wartość właściwości `Binding.NotifyOnValidationError` jest ustawiona na `true`, to wywoływane jest zdarzenie `Validation.Error`

Sprawdzanie poprawności danych (3)
INotifyDataErrorInfo

```
public class Person:INotifyDataErrorInfo
{
        public event EventHandler<DataErrorsChangedEventArgs> ErrorsChanged;
        private string AgeError;

        public System.Collections.IEnumerable GetErrors(string propertyName) {
            return new List<string>(){ AgeError};
        }

        public bool HasErrors {
            get { return !AgeError.Equals(""); }
        }

        private int _age;

        public int Age {
            get { return _age; }
            set {
                if (value < 0) {
                    AgeError = "Age must be greater or equal to 0";
                    if ( ErrorsChanged != null)
                        ErrorsChanged( this, new DataErrorsChangedEventArgs("Age"));
                } else {
                    _age = value;
                    AgeError = "";
                }
            }
        }
}
```

Sprawdzanie poprawności danych (4)
INotifyDataErrorInfo

```
<GroupBox  Validation.Error="TextBox_Error" Header="Person"
   DataContext="{Binding RelativeSource={RelativeSource Mode=FindAncestor,AncestorType=Window},Path=Director}">
    <TextBox Name="AgeTextBox">
        <TextBox.Text>
            <Binding Path="Age" ValidatesOnNotifyDataErrors="True" NotifyOnValidationError="True" Mode="TwoWay"/>
         </TextBox.Text>
    </TextBox>
</GroupBox>
```

```
public partial class MainWindow : Window
{

    public Person Director
    {
        get;
        set;
    }
                //sprawdzenie wewnątrz dowolnej metody
                if (Validation.GetHasError(AgeTextBox))
                {
                    ValidationError ve = Validation.GetErrors(AgeTextBox)[0];
                    MessageBox.Show(ve.ErrorContent.ToString());
                }
   private void TextBox_Error(object sender, ValidationErrorEventArgs e)
   {

       MessageBox.Show(e.Error.ErrorContent.ToString());
   }
```

Reguły sprawdzania danych

```
public class AgeValueRule: ValidationRule
{
    public int MinAge { get; set; }
    public int MaxAge { get; set; }

    public override ValidationResult Validate(object value, System.Globalization.CultureInfo cultureInfo)
    {
        int age = 0;
        try
        {
            if (((string)value).Length > 0)
            {
                age = int.Parse((string)value);
            }
        }
        catch
        {
            return new ValidationResult(false, "Illegal characters.");
        }

        if ((age < MinAge) || (age > MaxAge))
        {
            return new ValidationResult(false, "Age is not in the range <" + MinAge + ";" + MaxAge + ">.");
        }
        else
        {
            return new ValidationResult(true, null);
        }

    }
}
```

Reguły sprawdzania danych (2)

```
<Window.Resources>
    <local:AgeValueRule x:Key="AgeRule"/>

    <Style x:Key="textBoxStyle" TargetType="TextBox">
         <Style.Triggers>
               <Trigger Property="Validation.HasError" Value="true">
                   <Setter Property="ToolTip"
                           Value="{Binding RelativeSource={RelativeSource Self},
                           Path=(Validation.Errors)[0].ErrorContent}"/>
                </Trigger>
            </Style.Triggers>
    </Style>
</Window.Resources>
```


```
<TextBox Name="AgeTxt" Margin="5" Style="{StaticResource textBoxStyle}">
     <TextBox.Text>
         <Binding Path="Director.Age" ElementName="Window1"  >
             <Binding.ValidationRules>
                 <local:AgeValueRule MinAge="18" MaxAge="100"></local:AgeValueRule>
                 <DataErrorValidationRule></DataErrorValidationRule>
             </Binding.ValidationRules>
         </Binding>
     </TextBox.Text>

 </TextBox>
```

Sprawdzanie wielu wartości

```
public override ValidationResult Validate(object value, System.Globalization.CultureInfo cultureInfo)
{
     BindingGroup bg = (BindingGroup)value;
     Person person = (Person)bg.Items[0];
     int age = 0;
     string newAge = (string)bg.GetValue(person, "Age");
     string newName = (string)bg.GetValue(person, "Name");

     try{
         if ((newAge).Length > 0)
            age = int.Parse((string)value);
     }…

     if (newName.Length < MinNameLength)
     {
         return new ValidationResult(false, "Name is to short!!!");
     }

     if ((age < MinAge) || (age > MaxAge))
     {
         return new ValidationResult(false, "Age is not in the range <" + MinAge + ";" + MaxAge + ">.");
     }
     else
     {
         return new ValidationResult(true, null);
     }

 }
```

Sprawdzanie wielu wartości

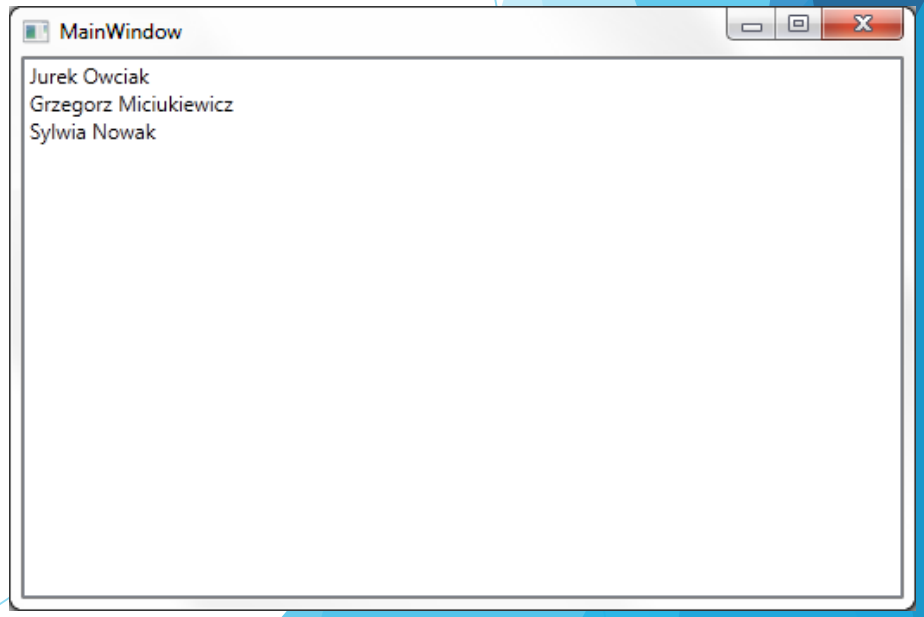
```
<StackPanel DataContext="{Binding ElementName=Window1,Path=Director}">
     <StackPanel.BindingGroup>
          <BindingGroup x:Name="BindingPerson">
               <BindingGroup.ValidationRules>
                    <local:AgeAndNameValueRule MinAge="3" MaxAge="100" MinNameLength="4" />
               </BindingGroup.ValidationRules>
          </BindingGroup>
      </StackPanel.BindingGroup>
      <TextBox Text="{Binding Path=Name}" Margin="5" LostFocus="TextBox_LostFocus" Style="{StaticResource textBoxStyle}"/>
      <TextBox Name="AgeTxt" Margin="5" Style="{StaticResource textBoxStyle}" LostFocus="TextBox_LostFocus">
           <TextBox.Text>
                <Binding Path="Age"  >
                </Binding>
           </TextBox.Text>

      </TextBox>
</StackPanel>
```

```
private void TextBox_LostFocus(object sender, RoutedEventArgs e)
{
     BindingPerson.CommitEdit();
}
```

Wiązanie kolekcji

```
<Window x:Class="WpfApplication3.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" Name="Window1">
    <DockPanel>
        <ListBox Name="PeopleList"
                 ItemsSource="{Binding ElementName=Window1,Path=People}"
                 DisplayMemberPath="Name"/>
    </DockPanel>
</Window>
```

```
private List<Person> _people;
public List<Person> People
{
    get { return _people; }
    set {_people = value;}
}
```

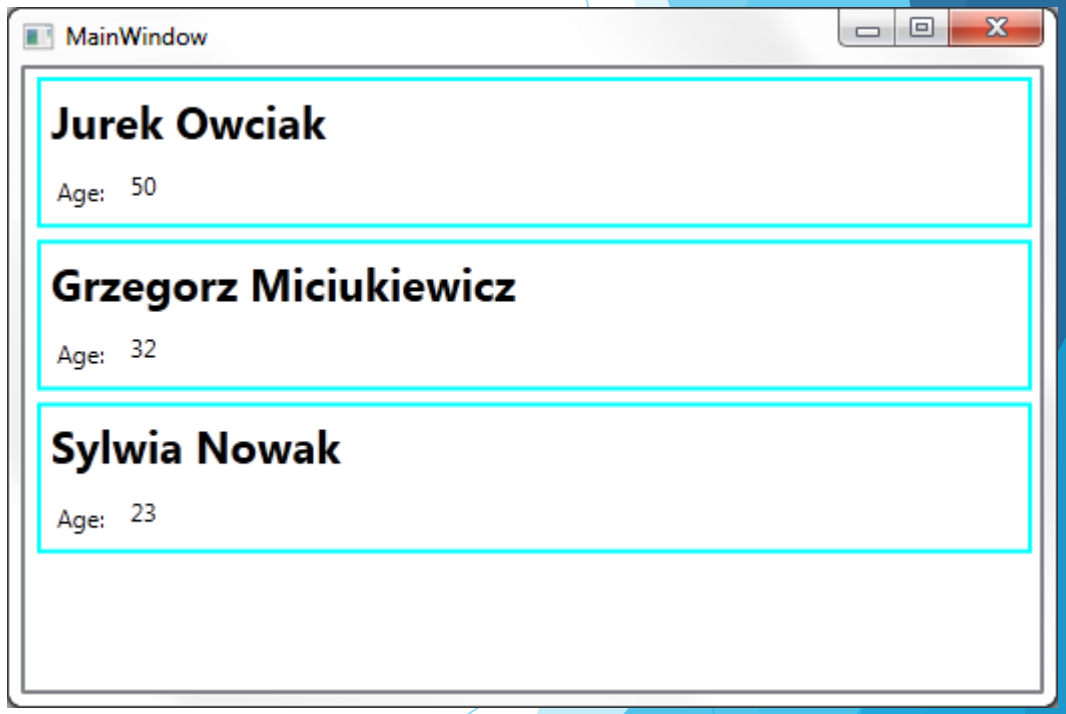


Wiązanie kolekcji

- dla obiektów domyślnie wywoływana jest metoda `ToString()`
- wyświetlenie wartości pojedynczej właściwości przez użycie `DisplayMemberPath`
- użycie szablonów danych (*Data Templates*):
  - Dla obiektów typu *Content* - `ContentTemplate`
  - Dla list - `ItemTemplate`

Wiązanie kolekcji

```
<ListBox Name="PeopleList"
         ItemsSource="{Binding ElementName=Window1,Path=People}"
         HorizontalContentAlignment="Stretch">
    <ListBox.ItemTemplate>
         <DataTemplate>
             <Border BorderBrush="Aqua" BorderThickness="2" Margin="3">
                  <DockPanel>
                       <Label FontWeight="Bold"
                              FontSize="22"
                              Content="{Binding Path=Name}"
                              DockPanel.Dock="Top"/>
                        <StackPanel Orientation="Horizontal">
                            <Label Margin="3">Age:</Label>
                            <Label Content="{Binding Path=Age}" />
                        </StackPanel>
                   </DockPanel>
             </Border>
          </DataTemplate>
      </ListBox.ItemTemplate>
 </ListBox>
```



Przeniesienie szablonu danych do zasobów

```
<Window.Resources>
    <DataTemplate x:Key="PersonTemplate">
        <Border BorderBrush="Aqua" BorderThickness="2" Margin="3">
            <DockPanel>
                <Label FontWeight="Bold" FontSize="22" Content="{Binding Path=Name}"
                       DockPanel.Dock="Top"  VerticalAlignment="Center" />
                <StackPanel Orientation="Horizontal">
                    <Label Margin="3">Age:</Label>
                    <Label Content="{Binding Path=Age}" />
                </StackPanel>
            </DockPanel>

        </Border>
    </DataTemplate>
</Window.Resources>

<DockPanel>
    <ListBox Name="PeopleList" ItemsSource="{Binding ElementName=Window1,Path=People}"
             HorizontalContentAlignment="Stretch"
             ItemTemplate="{StaticResource PersonTemplate}">
    </ListBox>
</DockPanel>
```

Informowanie o zmianie stanu obiektu
INotifyPropertyChanged

```
public class Person
{
        private int _age;
        private string _name;

        public string Name
        {
            get { return _name; }
            set { _name = value; }
        }

        public int Age
        {
            get { return _age;  }
            set { _age = value; }
        }
}

private void Button_Click(object sender, RoutedEventArgs e)
{
     Director.Name = "Marek Sawicki";
}

<TextBox Text="{Binding ElementName=Window1,Path=Director.Name}" />
```

Informowanie o zmianie stanu obiektu (2)

INotifyPropertyChanged

```
public class Person:INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    private int _age;
    private string _name;

    public string Name
    {
        get { return _name; }
        set {
                _name = value;
                if (PropertyChanged != null)
                    PropertyChanged( this, new PropertyChangedEventArgs( "Name"));
            }
    }

    public int Age
    {
        get { return _age;  }
        set {
               _age = value;
                if (PropertyChanged != null)
                    PropertyChanged( this, new PropertyChangedEventArgs( "Age"));
            }
    }
}
```

Informowanie o zmianach w kolekcjach ObservableCollection<>

```
private ObservableCollection<Person> _people;
public ObservableCollection<Person> People
{
    get { return _people; }
    set {_people = value;}
}

public Person Director { get; set; }

public MainWindow()
{
    _people = new ObservableCollection<Person>();

    Director = new Person() { Name = "Jurek Owciak", Age = 50 };
    _people.Add( Director);

    _people.Add(new Person() { Name = "Grzegorz Miciukiewicz", Age = 32 });
    _people.Add(new Person() { Name = "Sylwia Nowak", Age = 23 });
    InitializeComponent();
}

private void Button_Click(object sender, RoutedEventArgs e)
{
    Director.Name = "Marek Sawicki";
    _people.Add(new Person() { Name = "Anna Kowalska", Age = 43 });
}
```

UI Virtualization

- technika tworzenia elementów listy dla dużych list
- komponent `VirtualizingStackPanel`
- domyślnie wykorzystane w: `ListBox, ListView i DataGrid`
- ComboBox i `TreeView` nie wykorzystują `VirtualizingStackPanel`

```
<ComboBox>
    <ComboBox.ItemsPanel>
        <ItemsPanelTemplate>
            <VirtualizingStackPanel></VirtualizingStackPanel>
        </ItemsPanelTemplate>
    </ComboBox.ItemsPanel>
</ComboBox>
```

```
<TreeView VirtualizingStackPanel.IsVirtualizing="True"  … >
```

- Wyłączenie wirtualizacji:
  - użycie pojemnika ScrollViewer
  - wykorzystanie szablonów nie korzystających z `ItemsPanelTemplate`
  - ręczne tworzenie `ListBoxItems`

UI Virtualization (2)

- ponowne wykorzystanie elementów `ListBoxItem`

```
<ListBox VirtualizingStackPanel.VirtualizationMode="Recycling" ... >
```

- rozmiar bufora – ile nadmiarowych stron/elementów będzie gotowych
  - strony

```
<ListBox VirtualizingStackPanel.CacheLength="1"
         VirtualizingStackPanel.CacheLengthUnit="Page" ... />
```

  - elementy

```
<ListBox VirtualizingStackPanel.CacheLength="100"
         VirtualizingStackPanel.CacheLengthUnit="Item" ... />
```

- odłożone przewijanie `Deferred Scrolling`

```
<ListBox ScrollViewer.IsDeferredScrollingEnabled="True" ... />
```

Widoki danych

- wiązanie kolekcji jako źródło danych powoduje automatycznie utworzenie widoku (*view*)
- widok ten umożliwia przeglądanie elementów oraz wspiera takie operacje jak sortowanie, filtrowanie i grupowanie
- obiekty widoku dziedziczą po klasie `CollectionView`, natomiast wyróżnione zostały dwie specjalizowane klasy:
  - ListCollectionView - wiązanie z `ObservableCollection`
  - BindingListCollectionView - wykorzystanie ADO.NET
  - CollectionView – jeśli obiekt danych nie implementuje ani `IBindingList` ani `IList`

Nawigacja po elementach kolekcji

Nawigacja jest możliwa dzięki następującym cechom klasy `CollectionView`

- Właściwości:
  - `CurrentItem`
  - `Count`
  - `CurrentPosition`
- Metody:
  - `MoveCurrentToFirst()`
  - `MoveCurrentToLast()`
  - `MoveCurrentToNext()`
  - `MoveCurrentToPrevious()`
  - `MoveCurrentToPosition()`

Nawigacja po elementach kolekcji

```
private ListCollectionView view;

public MainWindow()
{
    _people = new ObservableCollection<Person>();

    Director = new Person() { Name = "Jurek Owciak", Age = 50 };
    _people.Add( Director);

    _people.Add(new Person() { Name = "Grzegorz Miciukiewicz", Age = 32 });
    _people.Add(new Person() { Name = "Sylwia Nowak", Age = 23 });
    InitializeComponent();
    view = (ListCollectionView) CollectionViewSource.GetDefaultView(PeopleList.ItemsSource);
    this.DataContext = _people;
}

private void Button_Click(object sender, RoutedEventArgs e)
{
    view.MoveCurrentToNext();
}
```

```
<TextBox Text="{Binding Name}" />
```

Filtrowanie kolekcji

```
private void Button_Click(object sender, RoutedEventArgs e)
{
     _people.Add(new Person() { Name = "Anna Kowalska", Age = 43 });
     view.Filter = new Predicate<object>(FilterPerson);
}

public bool FilterPerson(Object item)
{
    Person p = (Person)item;
    return (p.Age > 32);
}
```

```
view.Filter = (o) => ((Person)o).Age > 32;
```

Sortowanie kolekcji

- wykorzystanie właściwości SortDescriptions

```
view.SortDescriptions.Add(new SortDescription("Age", ListSortDirection.Descending));
```

- wykorzystanie właściwości CustomSort

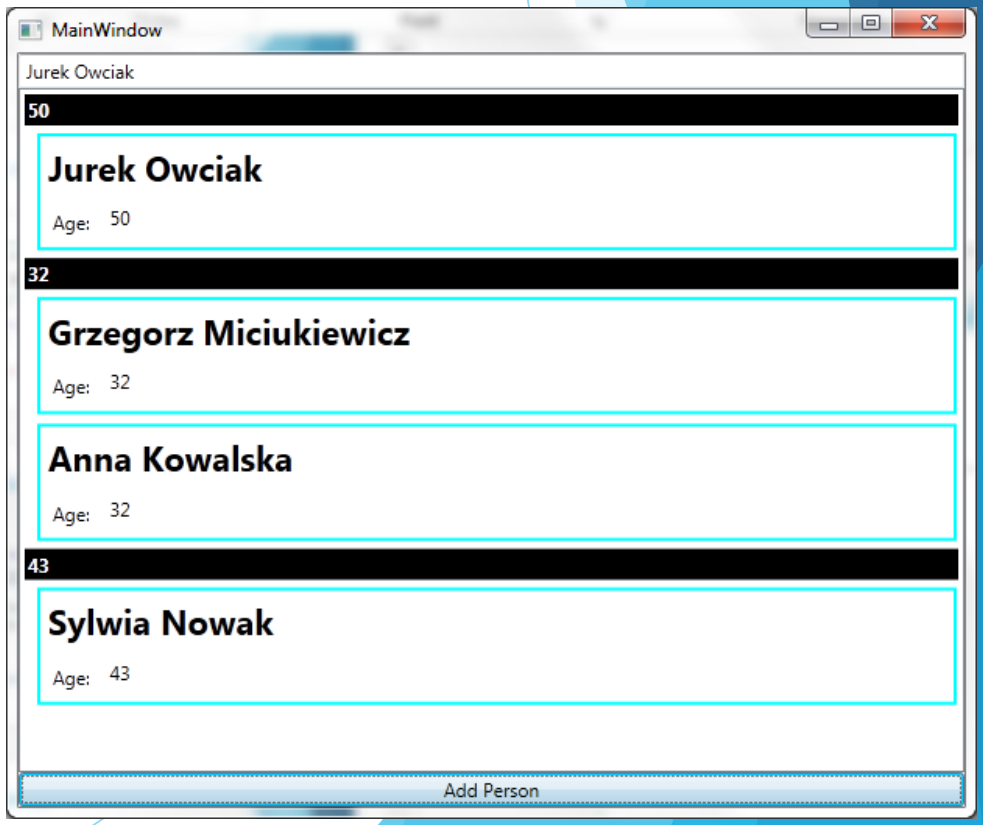
```
view.CustomSort = new SortPersonByAgeAndName();
```

```
public class SortPersonByAgeAndName : IComparer
{
    public int Compare(object _x, object _y)
    {
        Person x = (Person)_x;
        Person y = (Person)_y;
        if (x.Age == y.Age)
        {
            return x.Name.CompareTo(y.Name);
        }
        else
        {
            return x.Age.CompareTo(y.Age);
        }
    }
}
```

Grupowanie elementów kolekcji

```
view.GroupDescriptions.Add(new PropertyGroupDescription("Age"));
```

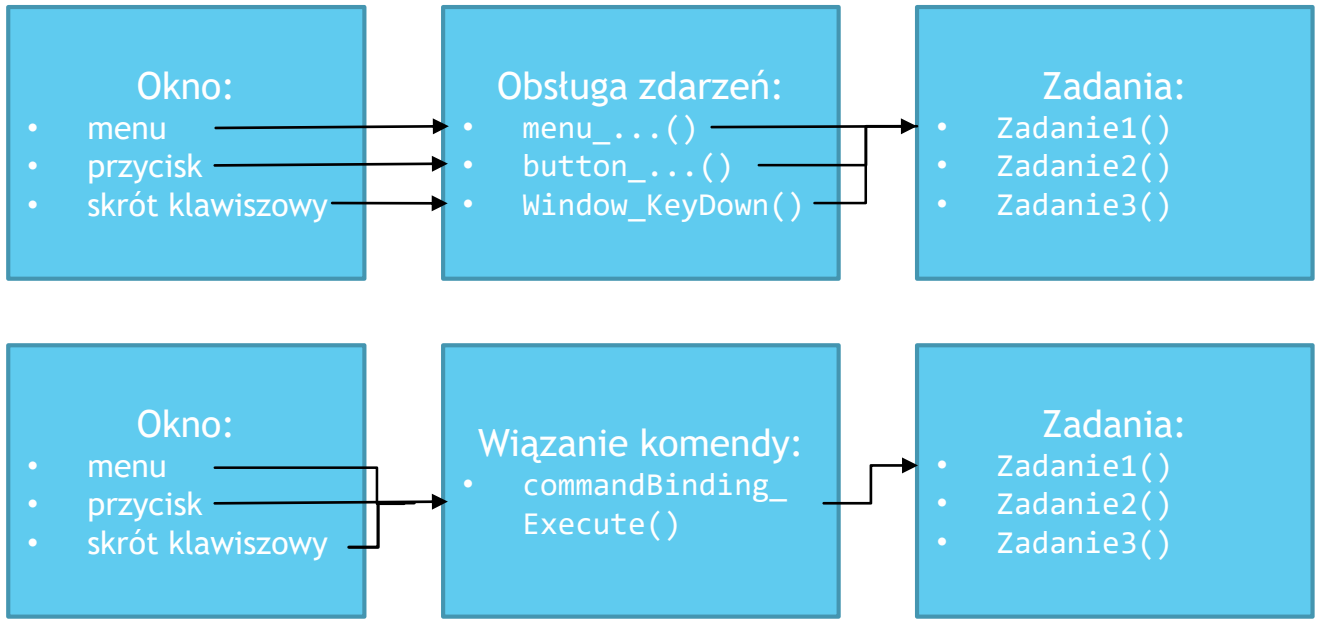
```
<ListBox Name="PeopleList"
         ItemsSource="{Binding ElementName=Window1,Path=People}"
         HorizontalContentAlignment="Stretch"
         ItemTemplate="{StaticResource PersonTemplate}">
    <ListBox.GroupStyle>
        <GroupStyle>
            <GroupStyle.HeaderTemplate>
                <DataTemplate>
                    <TextBlock Text="{Binding Path=Name}"
                               FontWeight="Bold"
                               Foreground="White"
                               Background="Black"
                               Margin="2"
                               Padding="2"/>
                </DataTemplate>
            </GroupStyle.HeaderTemplate>
        </GroupStyle>
    </ListBox.GroupStyle>
</ListBox>
```



System komend (command)

Komendy - idea

- Funkcjonalność aplikacji może zostać podzielona na zadania.
- Każde z zadań może zostać wykonane na wiele sposobów



Komendy – zalety i wady

- Podstawowe zalety:
  - oddelegowanie zdarzeń do odpowiednich komend
  - synchronizacja stanu kontrolek ze stanem związanej komendy
  - część kontrolek obsługuje komendy automatycznie
- Największe ograniczenia (wady)
  - brak mechanizmów śledzenia historii wywoływania komend
  - brak mechanizmu cofnięcia wykonania komendy (*Undo*)
  - ograniczone możliwości stanu komendy

Mechanizm komend – składniki

- Komenda – `Command` – reprezentuje zadanie aplikacji i śledzi możliwość jego wykonania
- Wiązanie wejścia – Input Binding - połączenie konkretnego wywołania (często dodatkowego) z komendą
- Wiązanie komendy – `Command Binding` – połączenie komendy z logiką aplikacji.
- Źródła komendy – Command source – źródło, które wywołało komendę (np. przycisk, opcja menu itp.)
- Cel komendy – Command target – element dla którego komenda została wywołana - może, ale nie musi być istotny

Interfejs ICommand

```
public interface ICommand
{
    void Execute(object parameter);
    bool CanExecute(object parameter);

    event EventHandler CanExecuteChanged;
}
```

- `Execute` – zawiera logikę komendy
- `CanExecute` – stan komendy = wartość `true` jeśli komendę można wywołać i `false` w przeciwnym przypadku
- `CanExecuteChanged` – zdarzenie wywoływane gdy zmieni się stan komendy – jest to sygnał dla wszystkich elementów kontrolnych używających komendy, żeby sprawdziły jej stan przez wywołanie metody `CanExecute()`

Komendy w WPF

- Zależności klas i z czym mamy do czynienia

Klasa implementująca `ICommand` z dodaną obsługą zdarzeń WPF

Klasa dziedzicząca z `RoutedCommand` z dodną właściwością `Text`

Biblioteka komend - podział

Komendy są zdefiniowane jako statyczne właściwości następujących statycznych klas:

- **Komendy aplikacji** (`ApplicationCommands`)

  obsługa schowka: Copy, Cut, Paste oraz dokumentu: New, Open, Save, SaveAs, Print…

- **Komendy nawigacji** (`NavigationCommands`)

  zdefiniowane dla aplikacji bazujących na stronach: BrowseBack, BrowseForward, NextPage, Refresh

- **Komendy edycji** (`EditingCommands`)

  zdefiniowane dla edycji dokumentu: MoveToLineEnd, MoveUpByPage, SelectToLineEnd, ToggleBold, ToggleUnderline

- **Komendy komponentu** (`ComponentCommands`)

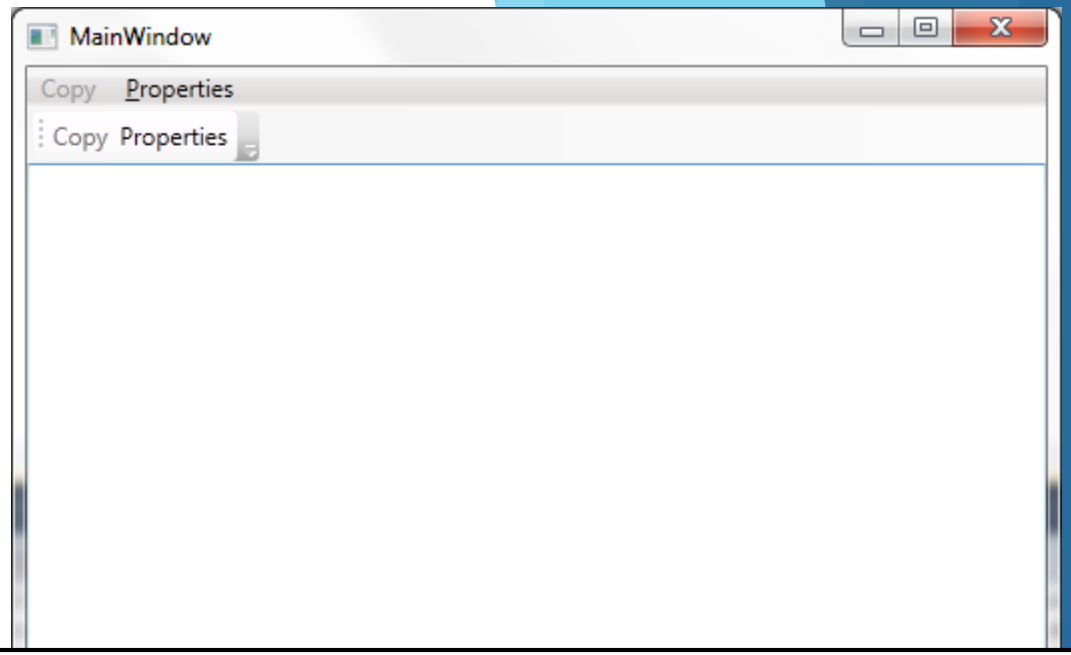
  komendy specyficzne dla komponentu: MoveDown, MoveFocusDown, MoveLeft, MoveRight, MoveToHome, SelectToEnd, SelectToPageDown

- **Komendy mediów** (`MediaCommands`)

  komendy do pracy z multimediami: Play, Pause, NextTrack, IncreaseVolume

Przykład

```
<Window ...>
    <DockPanel LastChildFill="True">
        <Menu DockPanel.Dock="Top">
            <MenuItem Command="ApplicationCommands.Copy"/>
            <MenuItem Header="_Properties" Command="Properties"/>
        </Menu>
        <ToolBarTray DockPanel.Dock="Top">
            <ToolBar>
                <Button Command="Copy" Content="Copy" />
                <Button Command="Properties" Content="Properties"/>
            </ToolBar>
        </ToolBarTray>
        <TextBox />
    </DockPanel>
</Window>
```




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

    InputBinding ib = new InputBinding(
        ApplicationCommands.Properties,
        new KeyGesture(Key.Enter, ModifierKeys.Shift));

    this.InputBindings.Add(ib);

    CommandBinding cb = new CommandBinding(ApplicationCommands.Properties);
    cb.Executed += new ExecutedRoutedEventHandler(cb_Executed);
    this.CommandBindings.Add(cb);
}

void cb_Executed(object sender, ExecutedRoutedEventArgs e)
{
    MessageBox.Show("Command properties");
}
```

Wiązanie komend

- Komponenty, które są związane z komendą są domyślnie wyłączone
- Aby to zmienić należy:
  - zaimplementować logikę dla danej komendy
  - określić kiedy może ona zostać wywołana – jest to opcjonalne – jeśli nie zostanie to zrobione, to komenda będzie aktywna dopóki będzie miała dodane metodę obsługi zdarzenia
  - określić kiedy komenda ma skutek, czyli można zawęzić jej działanie np. do konkretnego komponentu

```
CommandBinding cb = new CommandBinding(ApplicationCommands.Properties);
cb.Executed += new ExecutedRoutedEventHandler(cb_Executed);
this.CommandBindings.Add(cb);
```

Ograniczenie zakresu działania komendy

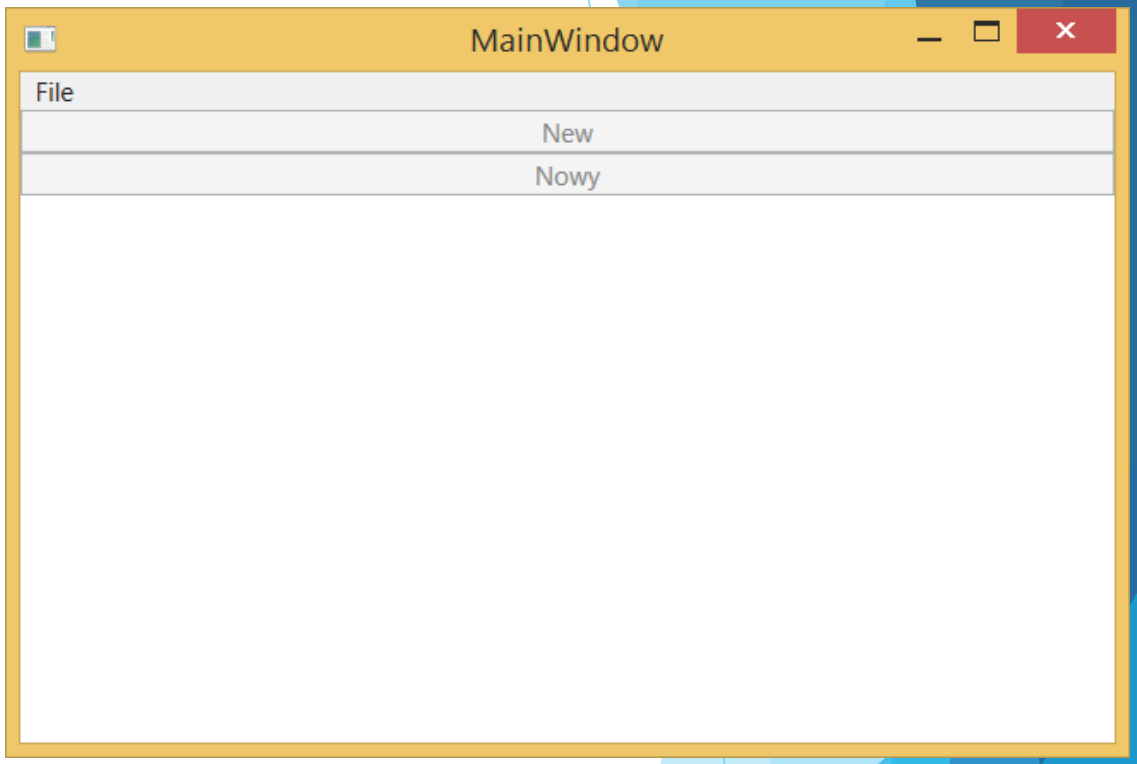
```
public MainWindow()
{
    InitializeComponent();

    InputBinding ib = new InputBinding(
         ApplicationCommands.Properties,
         new KeyGesture(Key.A, ModifierKeys.Control));

    this.InputBindings.Add(ib);

    CommandBinding cb = new CommandBinding(ApplicationCommands.Properties);
    cb.Executed += new ExecutedRoutedEventHandler(cb_Executed);
    this.CommandBindings.Add(cb);
}
```

```
<Window.CommandBindings>
    <CommandBinding Command="Properties" Executed="cb_Executed" />
</Window.CommandBindings>
```

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

    InputBinding ib = new InputBinding(
        ApplicationCommands.Properties,
        new KeyGesture(Key.A, ModifierKeys.Control));

    PropertiesButton.InputBindings.Add(ib);

    CommandBinding cb = new CommandBinding(ApplicationCommands.Properties);
    cb.Executed += new ExecutedRoutedEventHandler(cb_Executed);
    PropertiesButton.CommandBindings.Add(cb);
}
```

```
<Button Name="PropertiesButton" Command="Properties" Content="Properties">
    <Button.CommandBindings>
        <CommandBinding Command="Properties" Executed="cb_Executed"/>
    </Button.CommandBindings>
</Button>
```

Nazwa komendy



Dla Menu – właściwość Command.Text

```
<Menu DockPanel.Dock="Top">
    <MenuItem Header="File">
        <MenuItem Command="ApplicationCommands.New"/>
    </MenuItem>
    <MenuItem Header="_Properties" Command="Properties"/>
</Menu>
```

Dla przycisków:

```
<Button Command="New" Content="{x:Static ApplicationCommands.New}" />

<Button Command="New"
Content="{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"
/>
```

Wyłączanie komend

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

    cb = new CommandBinding(ApplicationCommands.New);
    cb.Executed +=new ExecutedRoutedEventHandler( cb_ExecutedNew);
    cb.CanExecute += new CanExecuteRoutedEventHandler(cb_CanExecute);
    this.CommandBindings.Add(cb);
}

void cb_CanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = IsCommandEnabled.IsChecked ?? true;
}
```

Kontrolki z wbudowaną obsługą komend

- kontrolka z przypisaną komendą, aby być aktywną musi być zostać wcześniej wybrana (mieć *focus*). Gdy np. komenda `Copy` jest związana z przyciskiem co jest celem takiej komendy?
- problem nie dotyczy komponentów w pojemnikach typu: `ToolBar` i `Menu`
- rozwiązanie: zdefiniowanie kto jest celem działania komendy – właściwość `CommandTarget`

```
<Button Command="Copy"
        Content="{x:Static ApplicationCommands.Copy}"
        CommandTarget="{Binding ElementName=txtDescription}"/>
```

- drugim rozwiązaniem jest utworzenie nowego zakresu dla *focus-a*

Kontrolki z wbudowaną obsługą komend

```
<DockPanel LastChildFill="True">
    <Menu DockPanel.Dock="Top">
        <MenuItem Header="Edit">
            <MenuItem Command="Copy"/>
            <MenuItem Command="Paste"/>
        </MenuItem>
        <MenuItem Header="_Properties" Command="Properties"/>
    </Menu>
    <ToolBarTray DockPanel.Dock="Top">
        <ToolBar>
            ...
            <Button Command="Copy" Content="{x:Static ApplicationCommands.Copy}"/>
            <Button Command="Paste" Content="{x:Static ApplicationCommands.Paste}"/>
        </ToolBar>
    </ToolBarTray>
    <StackPanel >
        <StackPanel FocusManager.IsFocusScope="True">
            <Button Command="Copy" Content="{x:Static ApplicationCommands.Copy}"/>
            <Button Command="Paste" Content="{x:Static ApplicationCommands.Paste}"/>
        </StackPanel>
        <CheckBox Margin="5" Name="IsCommandEnabled">Is Command Enabled?</CheckBox>
        <TextBox Text="First Text"/>
        <TextBox Text="Second Text"/>
    </StackPanel>
</DockPanel>
```

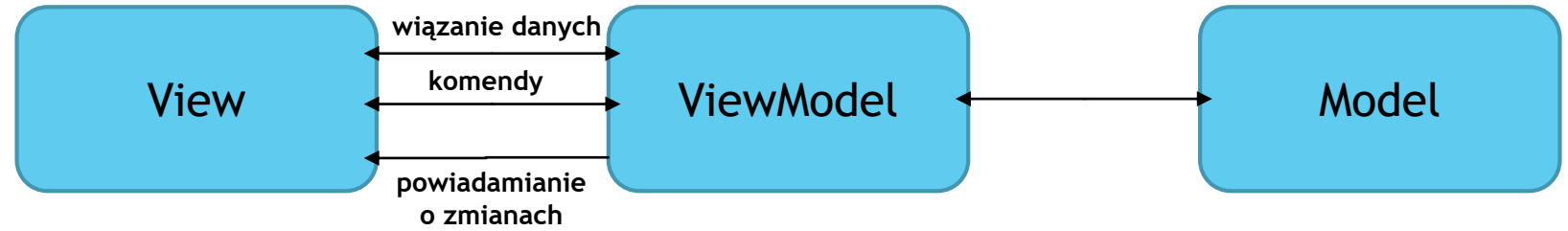
Blokowanie działania wbudowanych komend

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


    CommandBinding commandBinding = new CommandBinding(ApplicationCommands.Copy, null, SupressCommand);
    txt1.CommandBindings.Add(commandBinding);

    KeyBinding keyBinding = new KeyBinding(ApplicationCommands.NotACommand, Key.C, ModifierKeys.Control);
    txt1.InputBindings.Add(keyBinding);
}

private void SupressCommand(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = false;
    e.Handled = true;
}
```

Model MVVM (Model-View-ViewModel)

MVVM (Model-View-ViewModel)

- Wzorzec Model-View-ViewModel jest wzorcem bazującym na MVC (Model-View-Contoller)
- Pozwala odizolować logikę aplikacji od interfejsów użytkownika i danych



Model

- obejmuje logikę biznesową i dane
- z reguły zawiera kod operujący na danych (dostęp do nich, przechowywanie itp.)
- często ta warstwa będzie oddzielnym serwisem/klasą
- powinna być na tyle niezależna, żeby można było jej użyć w innym miejscu – np. do budowy innego rodzaju interfejsu
- nie modyfikuje bezpośrednio ani widoku ani modelu widoku

Widok (View)

Cechy charakterystyczne:

- zawiera elementy interfejsu (Control, UserControl), szablony danych i style
- połączenie z warstwą ViewModel poprzez właściwość DataContext
- elementy kontrolne są powiązane z właściwościami i komendami ViewModelu
- w widoku może konwertować dane oraz można dołączać elementy walidacji danych
- definiuje i obsługuje pewne zachowania związane z prezentacją danych (np. animacje)
- zawartość powinna być (w miarę możliwości) zawarta w kodzie XAML, może jednak zawierać kod, który jest trudny/niemożliwy do realizacji w XAMLu albo taki, który wymaga bezpośredniego odwołania się do elementów interfejsu użytkownika

Widok modelu (ViewModel)

- nie ma referencji do widoku (View)
- definiuje właściwości i komendy, które mogą być związane z interfejsami użytkownika w widoku
- informuje o zmianach stanu danych (INotifyPropertyChanged/INotifyCollectionChanged)
- koordynuje interakcję widoku z danymi
- może łączyć wiele modeli (relacja jeden do wielu)
- definiuje dodatkowe właściwości, których nie ma model
- może zawierać dodatkowe sprawdzanie poprawności danych
- może udostępniać model bezpośrednio do widoku lub może go specjalnie opakowywać (np. w zależności od tego, czy klasa modelu implementuje interfejs INotifyPropertyChanged)
- może definiować stany logiczne prezentowane użytkownikowi