Rozdział 15.
Przegląd pojemników WPF

Autor: Jacek Matulewski

e-mail: jacek@phys.uni.torun.pl

Fragment książki „Visual Studio 2017. Tworzenie aplikacji Windows w języku C#” (Helion, 2018) udostępniony studentom Wydziału Fizyki, Astronomii i Informatyki Stosowanej UMK w semestrze zimowym 2020/2021.

Do tej pory poznałeś trzy pojemniki (nazywane też mniej elegancko kontenerami) odpowiedzialne za układanie kontrolek w graficznym interfejsie użytkownika. Są to: `Grid`, `StackPanel` i `DockPanel`. W pojemniku `StackPanel` kontrolki ułożone są w poziomie lub pionie w jednej linii. W pojemniku `Grid` można zdefiniować siatkę komórek, w których rozmieszczamy kontrolki. Z kolei `DockPanel` pozwala na „przylepianie” kontrolek do krawędzi okna. To chyba trzy najczęściej wykorzystywane pojemniki.

Czym w ogóle są pojemniki? Są to elementy XAML, których klasy dziedziczą po klasie `Panel` z przestrzeni `System.Windows.Controls`, a które aranżują ułożenie kontrolek znajdujących się w ich wnętrzu (dla których są rodzicami). W odróżnieniu od większości kontrolek, w pojemnikach można umieszczać wiele kontrolek. Temat rozdziału jest dużo szerszy, wspomnę bowiem także o innych kontrolkach, które mogą przechowywać wiele elementów. Mogą to być wszelkiego typu listy, jak kontrolka `ListBox`, kontrolka `ComboBox`, czy kontrolka `TreeView`. Pojemniki w węższym znaczeniu, a więc np.: `Grid`, `StackPanel` czy `DockPanel`, różnią się od kontrolek `ListBox`, `ComboBox` lub `TreeView` tym, że nie dostarczają własnego widoku. Organizują jedynie położenie i wielkość innych kontrolek. Można jednak zmienić np. ich tło. Przycisk nie jest pojemnikiem, bo choć możemy w nim umieścić dowolną kontrolkę, to tylko jedną. Jeżeli chcemy ich więcej, musimy użyć właśnie któregoś z pojemników.

Pojemniki (Layout Containers)

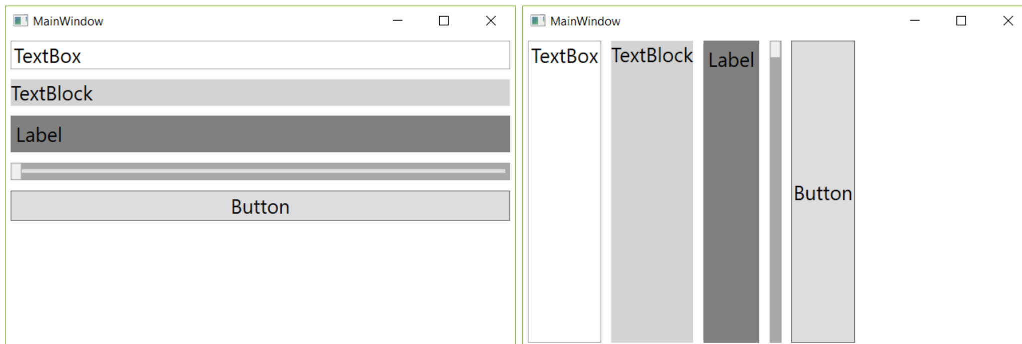
Poniżej przedstawię przegląd pojemników, w których będę prezentował zbiór kilku kontrolek, zawsze tych samych. Nie będzie w tym jakiejś większej filozofii. Chodzi mi przede wszystkim o to, żeby czytelnik zobaczył, co ma do dyspozycji. Nie będę przy tym zagnieżdżał kontrolek, żeby niepotrzebnie nie komplikować przeglądu, ale należy sobie zdawać sprawę, że jest to jak najbardziej możliwe.

Zacznijmy od czegoś, co już dobrze znamy, a więc od pojemnika `StackPanel`. Utwórzmy nową aplikację WPF o nazwie *PojemnikiWPF* i w pliku *MainWindow.xaml* umieśćmy kod z listingu 15.1. Zobaczymy kontrolki ułożone w jednej kolumnie (lewa część rysunku 15.1). Do elementu `StackPanel` możemy dodać atrybut `Orientation` z wartością `Horizontal` i zmienić układ kontrolek na poziomy (prawa część rysunku 15.1).

Listing 15.1. Kilka kontrolek zorganizowanych przez pojemnik StackPanel. Dodałem kolory tła, aby obszary kontrolek były lepiej widoczne

```
<Window x:Class="PojemnikiWPF.MainWindow"
        ...
        Title="MainWindow" Height="350" Width="525"
```

```
            FontSize="20">
        <StackPanel>
            <TextBox Margin="5" Text="TextBox" />
            <TextBlock Margin="5" Text="TextBlock" Background="LightGray" />
            <Label Margin="5" Content="Label" Background="Gray" />
            <Slider Margin="5" Background="DarkGray" />
            <Button Margin="5" Content="Button" />
        </StackPanel>
</Window>
```

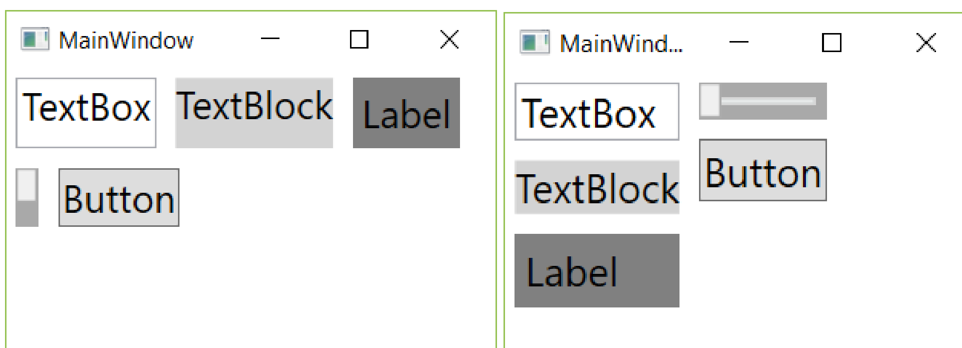


Rysunek 15.1. Kontrolki, których ułożenie jest kontrolowane przez pojemnik StackPanel

Zmieńmy teraz typ pojemnika na `WrapPanel`, nie zmieniając jego zawartości (listing 15.2). Kontrolki zmienią ułożenie: nadal będą układane w pionie lub poziomie (w tym pojemniku też działa atrybut `Orientation`), ale nie będą zajmować całej szerokości lub wysokości okna, a po dotarciu do krawędzi okna będą układane w kolejnej kolumnie lub rzędzie (rysunek 15.2). Wysokość rzędów w ułożeniu poziomym oraz szerokość kolumn w ułożeniu pionowym zależy od wielkości umieszczonych w nich kontrolek — nie można jej ustalić bezpośrednio własnościami pojemnika.

Listing 15.2. Zmiana pojemnika na WrapPanel

```
<Window x:Class="PojemnikiWPF.MainWindow"
        ...
        Title="MainWindow" Height="350" Width="525"
        FontSize="20">
    <WrapPanel Orientation="Vertical">
        <TextBox Margin="5" Text="TextBox" />
        <TextBlock Margin="5" Text="TextBlock" Background="LightGray" />
        <Label Margin="5" Content="Label" Background="Gray" />
        <Slider Margin="5" Background="DarkGray" />
        <Button Margin="5" Content="Button" />
    </WrapPanel>
</Window>
```

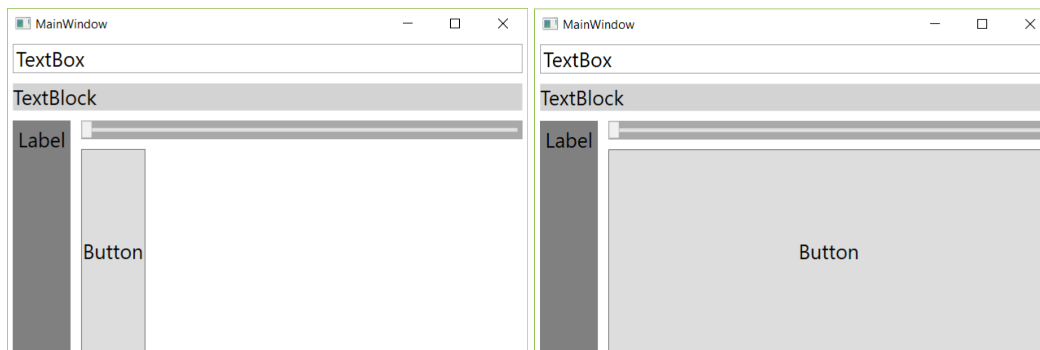


Rysunek 15.2. Jak wyżej, ale dla pojemnika WrapPanel

Kolejny pojemnik, o którym należy wspomnieć, to `DockPanel`. Nie ma on własności `Orientation`. Zamiast tego w każdej kontrolce-dziecku można użyć własności doczepianej `DockPanel.Dock`, która wskaże krawędź okna, do którego kontrolka ma być „przyklejona” (listing 15.3). Kolejność kontrolek z tą samą własnością w kodzie XAML wyznacza ich kolejność ustawienia od krawędzi (pierwsza jest najbliżej). Kolejność kontrolek w kodzie XAML wyznacza również ich pierwszeństwo w rogach okna, gdy są „przyklejone” do różnych krawędzi o wspólnym wierzchołku. Wśród możliwych wartości własności `DockPanel.Dock` nie ma takiej, która oznaczałaby wypełnienie (jak `Fill` w Windows Forms). Wolne miejsce pozostawione w pojemniku przez pozostałe kontrolki-dzieci może wypełnić tylko ostatnia kontrolka, jeżeli ustawimy własność pojemnika `LastChildFill` na `True` (prawa część rysunku 15.3).

Listing 15.3. Użycie pojemnika DockPanel

```
<Window x:Class="PojemnikiWPF.MainWindow"
        ...
        Title="MainWindow" Height="350" Width="525"
        FontSize="20">
    <DockPanel LastChildFill="False">
        <TextBox Margin="5" DockPanel.Dock="Top" Text="TextBox" />
        <TextBlock Margin="5" DockPanel.Dock="Top" Text="TextBlock"
                   Background="LightGray" />
        <Label Margin="5" DockPanel.Dock="Left" Content="Label" Background="Gray" />
        <Slider Margin="5" DockPanel.Dock="Top" Background="DarkGray" />
        <Button Margin="5" Content="Button" />
    </DockPanel>
</Window>
```



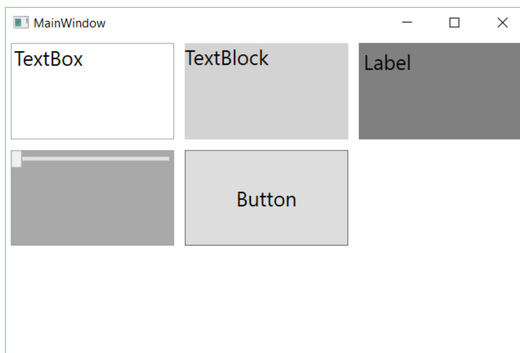
Rysunek 15.3 Jak wyżej, ale dla pojemnika DockPanel. Z prawej ułożenie w sytuacji, gdy atrybut LastChildFill jest ustawiony na True

Zmieńmy teraz pojemnik na `UniformGrid` (listing 15.4). To prosty w działaniu pojemnik, który dzieli okno na komórki o równej szerokości i wysokości. Automatycznie przypisuje poszczególne kontrolki do kolejnych komórek, nadając im rozmiar wypełniający całą komórkę (rysunek 15.4). Przy zmianie rozmiaru okna rozmiary komórek zmieniają się proporcjonalnie.

Listing 15.4. Pojemnik UniformGrid

```
<Window x:Class="PojemnikiWPF.MainWindow"
        ...
        Title="MainWindow" Height="350" Width="525"
        FontSize="20">
    <UniformGrid Columns="3" Rows="3">
        <TextBox Margin="5" Text="TextBox" />
        <TextBlock Margin="5" Text="TextBlock" Background="LightGray" />
        <Label Margin="5" Content="Label" Background="Gray" />
```

```
            <Slider Margin="5" Background="DarkGray" />
            <Button Margin="5" Content="Button" />
        </UniformGrid>
    </Window>
```



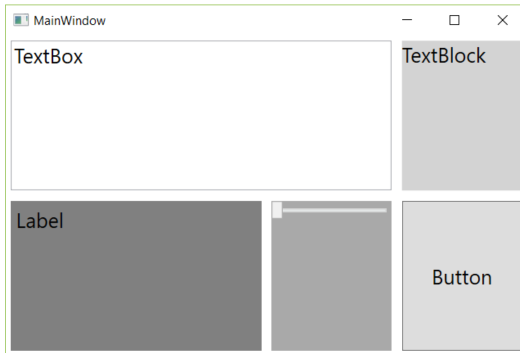
Rysunek 15.4. Jak wyżej, ale dla pojemnika UniformGrid

Gdy tworzymy nowy projekt, domyślnym pojemnikiem kontrolującym położenie kontrolek jest `Grid` (z ang. siatka). Nie będę o niej pisał zbyt obszernie, bo używaliśmy jej dotąd już wiele razy. Siatka pozwala na dowolne ułożenie kontrolek w komórkach, na co, inaczej niż w pojemniku `UniformGrid`, mamy wpływ. Rozmiary komórek możemy też określać, dzieląc siatkę na kolumny i wiersze o wskazanych szerokościach i wysokościach (własności `ColumnDefinitions` i `RowDefinitions`). Rozmiary wierszy możemy określać bezwzględnie, podając liczbę pikseli, lub względnie (zob. użycie gwiazdki na listingu 15.5). Kontrolki przypisujemy do komórek, używając własności doczepianych `Grid.Column` i `Grid.Row` (pierwszy wiersz i pierwsza kolumna mają numer 0). Nie jesteśmy jednak całkowicie ograniczeni strukturą siatki. Możemy użyć własności doczepianych `Grid.ColumnSpan` i `Grid.RowSpan`, żeby rozciągnąć kontrolkę na dwie lub więcej kolumn i wierszy. Przykładem jest kontrolka `TextBox` rozciągnięta na dwie kolumny na listingu 15.5 (rysunek 15.5). Kontrolka nie musi też zajmować całej komórki.

Listing 15.5. Użycie siatki do ułożenia kontrolek

```
<Window x:Class="PojemnikiWPF.MainWindow"
        ...
        Title="MainWindow" Height="350" Width="525"
        FontSize="20">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="2*" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <TextBox Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2" Margin="5"
                 Text="TextBox" />
        <TextBlock Grid.Row="0" Grid.Column="2" Margin="5" Text="TextBlock"
                   Background="LightGray" />
        <Label Grid.Row="1" Grid.Column="0" Margin="5" Content="Label"
               Background="Gray" />
        <Slider Grid.Row="1" Grid.Column="1" Margin="5" Background="DarkGray" />
        <Button Grid.Row="1" Grid.Column="2" Margin="5" Content="Button" />
```

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

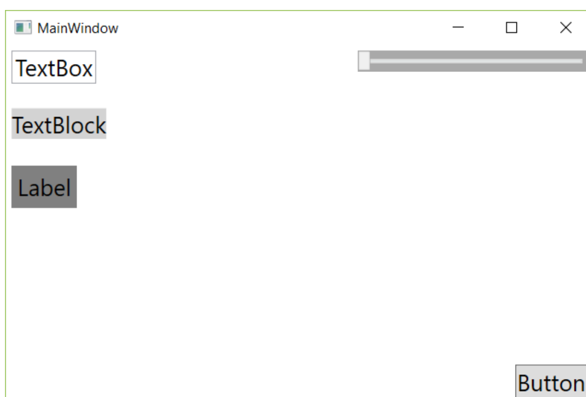


Rysunek 15.5. Jak wyżej, ale dla pojemnika Grid

Ostatnim pojemnikiem WPF jest `Canvas` (z ang. płótno), słusznie kojarzony z grafiką. W tym pojemniku położenie kontrolek jest wyznaczane przez podanie odległości od dwóch wskazanych krawędzi okna (listing 15.6, rysunek 15.6). Pojemnik `Canvas` odtwarza bezwzględne pozycjonowanie kontrolek, jakie znamy z Windows Forms.

Listing 15.6. Pojemnik Canvas

```
<Window x:Class="PojemnikiWPF.MainWindow"
        ...
        Title="MainWindow" Height="350" Width="525"
        FontSize="20">
    <Canvas>
        <TextBox Canvas.Left="0" Canvas.Top="0" Margin="5" Text="TextBox" />
        <TextBlock Canvas.Left="0" Canvas.Top="50" Margin="5" Text="TextBlock"
                   Background="LightGray" />
        <Label Canvas.Left="0" Canvas.Top="100" Margin="5" Content="Label"
               Background="Gray" />
        <Slider Canvas.Right="0" Canvas.Top="0" Width="200" Margin="5"
                Background="DarkGray" />
        <Button Canvas.Right="0" Canvas.Bottom="0" Margin="5" Content="Button" />
    </Canvas>
</Window>
```



Rysunek 15.6. Jak wyżej, ale dla pojemnika Canvas

Wszystkie dostępne w WPF pojemniki podsumowuje tabela 15.1. Warto jeszcze wspomnieć o wygodnym pojemniku, który pojawia się tylko w UWP. Jest nim `RelativePanel`. W tym pojemniku możemy wskazywać relatywne położenie elementów interfejsu użytkownika, decydując, że kontrolka ma leżeć pod, nad, z lewej lub z prawej strony innej kontrolki zidentyfikowanej poprzez nazwę. Dzięki temu powstaje elastyczny interfejs

użytkownika, którym łatwiej zarządzać na różnych rozmiarach ekranu. Wsparciem w tym zakresie jest menedżer stanów wizualnych, w którym można określić ułożenie kontrolek dla różnych typów ekranu.

Tabela 15.1. Podsumowanie pojemników WPF

Klasa/element	**Własności**	**Własności doczepiane**
`StackPanel`	`Orientation`	
`WrapPanel`	`Orientation`	
`DockPanel`	`LastChildFill`	`DockPanel.Dock`
`UniformGrid`	`Columns`, `Rows`	
`Grid`	`ColumnDefinitions`, `RowDefinitions`	`Grid.Column`, `Grid.Row`, `Grid.ColumnSpan`, `Grid.RowSpan`
`Canvas`		`Canvas.Left`, `Canvas.Top`, `Canvas.Right`, `Canvas.Bottom`

Kontrolki ułożenia (Layout Controls)

To nie koniec. Oprócz pojemników organizujących ułożenie kontrolek mamy jeszcze kilka kontrolek, które w różny sposób modyfikują wyświetlanie kontrolek lub ich grup. Wspominam tu o nich, bo są często używane w kontekście pojemników. Świetnym przykładem jest `ScrollViewer`, który pozwala na przewijanie umieszczonej w nim zawartości (jednej kontrolki lub pojemnika z wieloma kontrolkami). Jeżeli umieścimy w nim pole tekstowe, to gdy rozmiar wyświetlanego tekstu będzie przekraczał rozmiar kontrolki, `ScrollViewer` umożliwi jego przewijanie. Podobnie zadziała w przypadku listy. Jeżeli elementów np. w pojemniku `StackPanel` jest tak dużo, że „wystają” poza rozmiar kontrolki, `ScrollViewer` także umożliwi ich przewijanie.

Aby przetestować działanie tej kontrolki, wróćmy do pojemnika `StackPanel` z ułożeniem kontrolek w pionie (listing 15.1) na razie bez kontrolki `ScrollViewer`. Gdy po uruchomieniu aplikacji zmniejszymy wysokość okna, kontrolki zostaną zwyczajnie obcięte. Ale jeżeli pojemnik umieścimy w elemencie `ScrollViewer` (listing 15.7), to po zmniejszeniu wysokości okna aktywny stanie się pasek przewijania pionowego, który pozwoli nam przewinąć zawartość pojemnika tak, że będziemy mogli dostać się do wszystkich umieszczonych w nim kontrolek.

Listing 15.7. Użycie ScrollViewer

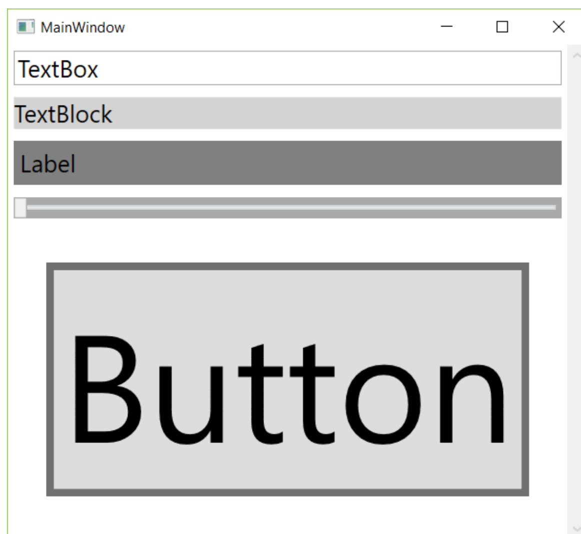
```
<Window x:Class="PojemnikiWPF.MainWindow"
        ...
        Title="MainWindow" Height="350" Width="525"
        FontSize="20">
    <ScrollViewer>
        <StackPanel ScrollViewer.VerticalScrollBarVisibility="Auto"
ScrollViewer.HorizontalScrollBarVisibility="Auto">
            <TextBox Margin="5" Text="TextBox" />
            <TextBlock Margin="5" Text="TextBlock" Background="LightGray" />
            <Label Margin="5" Content="Label" Background="Gray" />
            <Slider Margin="5" Background="DarkGray" />
            <Button Margin="5" Content="Button" />
        </StackPanel>
    </ScrollViewer>
</Window>
```

Kolejną ciekawą kontrolką modyfikującą sposób wyświetlania innych kontrolek jest `ViewBox`. Dla przykładu użyjmy jej na kontrolce `Button` (listing 15.8). Powoduje ona, że kontrolka umieszczona wewnątrz niej stara się zająć całą dostępną przestrzeń. Nie oznacza to jednak prostej zmiany jej rozmiaru. Na kontrolce wewnątrz

ViewBox wykonywana jest bowiem transformacja skalowania od domyślnych rozmiarów do rozmiaru, który wypełni całą szerokość pojemnika StackPanel. Dzięki skalowaniu powiększona będzie nie tylko sama kontrolka, ale także jej zawartość — w naszym przykładzie jest nią kontrolka TextBlock wyświetlająca etykietę w przycisku. Przeskalowane zostaną także marginesy przycisku. Pokazuje to rysunek 15.7.

Listing 15.8. Użycie ViewBox

```
<Window x:Class="PojemnikiWPF.MainWindow"
        ...
        Title="MainWindow" Height="350" Width="525"
        FontSize="20">
    <ScrollViewer>
        <StackPanel ScrollViewer.VerticalScrollBarVisibility="Auto"
                    ScrollViewer.HorizontalScrollBarVisibility="Auto">
            <TextBox Margin="5" Text="TextBox" />
            <TextBlock Margin="5" Text="TextBlock" Background="LightGray" />
            <Label Margin="5" Content="Label" Background="Gray" />
            <Slider Margin="5" Background="DarkGray" />
            <Viewbox Stretch="Uniform">
                <Button Margin="5" Content="Button" />
            </Viewbox>
        </StackPanel>
    </ScrollViewer>
</Window>
```



Rysunek 15.7. Efekt użycia kontrolki ViewBox

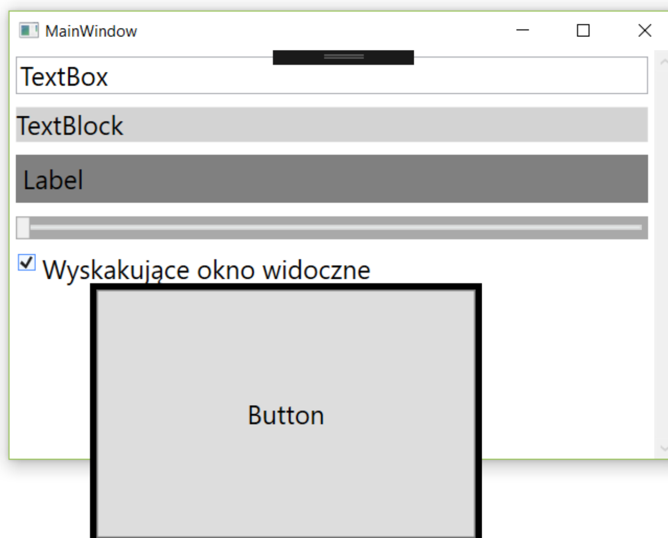
Inną ciekawą kontrolką jest Popup. „Wyjmuje” ona umieszczoną w niej zawartość z okna i umieszcza w dodatkowym wyskakującym okienku. Możemy określić jego wielkość i położenie na ekranie (np. względem położenia myszki). To dodatkowe okno jest widoczne lub ukryte w zależności od wartości własności IsOpen (rysunek 15.8). W przykładzie widocznym na listingu 15.9 związaliśmy tę własność z własnością IsChecked kontrolki pola opcji (CheckBox)[1].

Listing 15.9. Użycie Popup

```
<Window x:Class="PojemnikiWPF.MainWindow"
        ...
```

[1] Więcej o kontrolce Popup przeczytasz na stronie *http://www.c-sharpcorner.com/UploadFile/mahesh/using-xaml-popup-in-wpf/*.

```
        Title="MainWindow" Height="350" Width="525"
        FontSize="20">
    <ScrollViewer>
        <StackPanel ScrollViewer.VerticalScrollBarVisibility="Auto"
ScrollViewer.HorizontalScrollBarVisibility="Auto">
            <TextBox Margin="5" Text="TextBox" />
            <TextBlock Margin="5" Text="TextBlock" Background="LightGray" />
            <Label Margin="5" Content="Label" Background="Gray" />
            <Slider Margin="5" Background="DarkGray" />
            <CheckBox x:Name="checkBox" Margin="5" Content="Wyskakujące okno widoczne"
                      IsChecked="False" />
            <Popup Width="300" Height="200"
                   IsOpen="{Binding ElementName=checkBox, Path=IsChecked}"
                   Placement="MousePoint" HorizontalOffset="50" VerticalOffset="20">
                <Button Margin="5" Content="Button" />
            </Popup>
        </StackPanel>
    </ScrollViewer>
</Window>
```



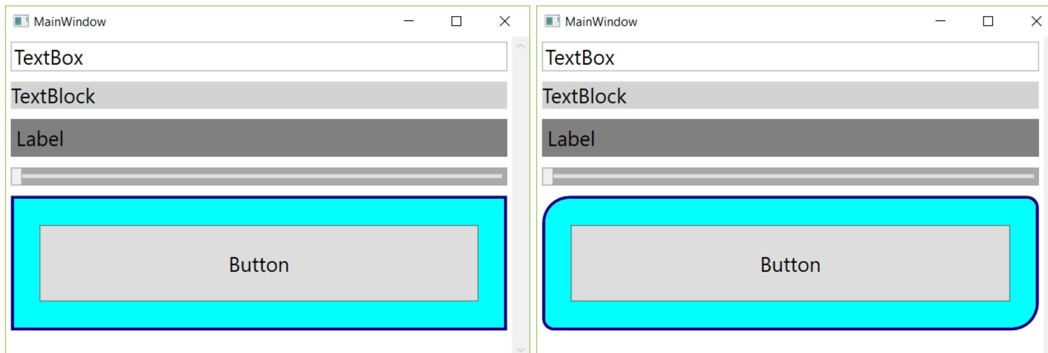
Rysunek 15.8. Efekt użycia kontrolki Popup

Na koniec warto też wspomnieć o prostej, ale bardzo użytecznej kontrolce `Border`, która dodaje obramowanie o podanej grubości i kolorze. Przykład jest widoczny na listingu 15.10. Do czterech wierzchołków dorysowywanego brzegu można dodać zaokrąglenia. Ich promienie ustalamy za pomocą atrybutu `CornerRadius` (prawa część rysunku 15.9).

Listing 15.10. Dodanie zewnętrznych krawędzi za pomocą kontrolki Border

```
<Window x:Class="PojemnikiWPF.MainWindow"
        ...
        Title="MainWindow" Height="350" Width="525"
        FontSize="20">
    <ScrollViewer>
        <StackPanel ScrollViewer.VerticalScrollBarVisibility="Auto"
                    ScrollViewer.HorizontalScrollBarVisibility="Auto">
            <TextBox Margin="5" Text="TextBox" />
```

```
            <TextBlock Margin="5" Text="TextBlock" Background="LightGray" />
            <Label Margin="5" Content="Label" Background="Gray" />
            <Slider Margin="5" Background="DarkGray" />
            <Border BorderBrush="Navy" BorderThickness="3" Background="Cyan"
                    CornerRadius="25,10,25,10">
                <Button Margin="25" Height="100" Content="Button" />
            </Border>
        </StackPanel>
    </ScrollViewer>
</Window>
```



Rysunek 15.9. Efekt użycia kontrolki Border. Z prawej strony zdefiniowany został atrybut CornerRadius

Projektowanie własnego pojemnika

W pojemnikach, które dziedziczą z klasy `Panel`, elementy są organizowane w dwuetapowym procesie. W pierwszym etapie, który można nazwać „pomiarem", badane jest miejsce niezbędne do wyświetlenia elementów, a jednocześnie elementy są powiadamiane o dostępnym dla nich obszarze[2]. W drugim wyznaczane są położenia i rozmiary kontrolek-elementów, jakie zobaczymy na ekranie.

Jeżeli chcemy zaprojektować własny pojemnik, a możemy to zrobić, tworząc klasę dziedziczącą po klasie `Panel`, musimy zdefiniować dwie metody, które są używane w tych dwóch etapach układania elementów: `MeasureOverride` i `ArrangeOverride`. Pierwsza powinna wywołać metodę `Measure` każdej kontrolki-dziecka, powiadamiając je o dostępnej przestrzeni. Może również zbierać informacje o tym, jaki obszar jest potrzebny dla wszystkich kontrolek, i przekazać ją za pomocą wartości zwracanej przez metodę. Druga metoda powinna wywoływać metodę `Arrange` poszczególnych kontrolek-dzieci, wskazując w jej argumentach ich ostateczne położenie i rozmiar.

Listing 15.11 prezentuje bardzo prostą klasę pojemnika, w której obszar przeznaczony dla kontrolek-dzieci jest ustalony przez publiczną własność `ContentSize` (ustawianą z kodu XAML). Jej domyślna wartość wyznacza obszar o rozmiarze 300×200 pikseli. Kontrolki ustawiane są w tym obszarze w losowych pozycjach. Ich rozmiary pozostają równe domyślnym rozmiarom kontrolek, o ile te są większe od rozmiaru wskazanego przez własność `MinimalChildSize` pojemnika; jeżeli nie — stosowany jest rozmiar minimalny. Obie własności zdefiniowane w pojemniku powinny być w zasadzie własnościami zależności (ang. *dependency property*), ale ponieważ jeszcze ich nie znamy, na razie wystarczą nam zwykłe własności. Przez to będziemy mogli ustalać ich wartości z kodu XAML, ale już nie zadziałają wiązania. Zwróć także uwagę, że odświeżenie ułożenia, np. przy zmianie rozmiaru okna, powoduje ponowne losowanie pozycji kontrolek[3]. Klasę nowego pojemnika `RandomPanel` należy umieścić w osobnym pliku, np. *RandomPanel.cs*. Przykład jego użycia jest widoczny na listingu 15.12.

---

[2] Por. informacje o transformacji `LayoutTransform` w rozdziale 12.

[3] W internecie można znaleźć kilka dobrych opisów tworzenia własnych pojemników (hasło „WPF custom panel"). Dwa z nich to *https://www.codeproject.com/Articles/37348/Creating-Custom-Panels-In-WPF* oraz *https://www.codeproject.com/Articles/15705/FishEyePanel-FanPanel-Examples-of-custom-layout-pa*.

Listing 15.11. Definicja własnego pojemnika rozmieszczającego kontrolki w losowych położeniach

```
using System;
using System.Windows;
using System.Windows.Controls;

namespace PojemnikiWPF
{
    public class RandomPanel : Panel
    {
        public Size ContentSize { get; set; } = new Size(300, 200);
        public Size MinimalChildSize { get; set; } = new Size(100, 25);

        protected override Size MeasureOverride(Size availableSize)
        {
            foreach (UIElement child in InternalChildren)
            {
                child.Measure(availableSize);
            }
            return ContentSize;
        }

        private Random random = new Random();

        protected override Size ArrangeOverride(Size finalSize)
        {
            foreach (UIElement child in InternalChildren)
            {
                double x = ContentSize.Width * random.NextDouble();
                double y = ContentSize.Height * random.NextDouble();
                Size childSize = child.DesiredSize;
                if (childSize.Width < MinimalChildSize.Width)
                    childSize.Width = MinimalChildSize.Width;
                if (childSize.Height < MinimalChildSize.Height)
                    childSize.Height = MinimalChildSize.Height;
                child.Arrange(new Rect(new Point(x, y), childSize));
            }
            return finalSize;
        }
    }
}
```

Listing 15.12. Przykład użycia pojemnika RandomPanel

```
<Window x:Class="PojemnikiWPF.MainWindow"
        ...
        Title="MainWindow" Height="350" Width="525"
        FontSize="20">
    <local:RandomPanel ContentSize="100,100" MinimalChildSize="25,25">
```

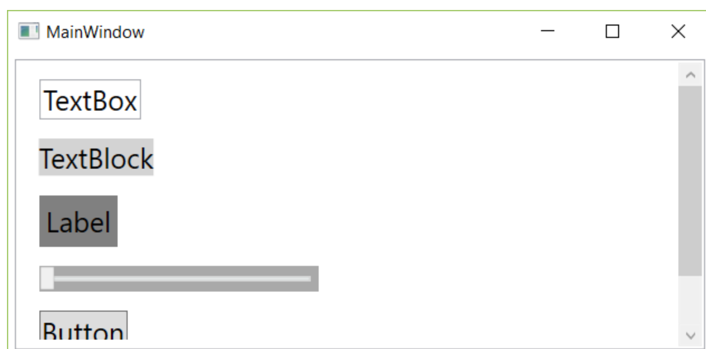
```
            <TextBox Margin="5" Text="TextBox" />
            <TextBlock Margin="5" Text="TextBlock" Background="LightGray" />
            <Label Margin="5" Content="Label" Background="Gray" />
            <Slider Margin="5" Background="DarkGray" />
            <Button Margin="5" Content="Button" />
    </local:RandomPanel>
</Window>
```

Listy (Items Controls)

Wspomniałem, że nie tylko pojemniki mogą przechowywać wiele elementów. Potrafią to także różnego typu listy (`ListBox`, `ListView`), drzewa (`TreeView`) i rozwijane listy (`ComboBox`). Zwykle pokazywane są w nich kolekcje łańcuchów, jednak tak naprawdę mogą przechowywać dowolne kontrolki WPF. I wcale nie są tak różne od pojemników omówionych wyżej (tj. klas dziedziczących po klasie `Panel`). Dla przykładu kontrolka `ItemsControl` ma bardzo podobne działanie jak `StackPanel` w otoczeniu `ScrollViewer` (listing 15.13, rysunek 15.10).

Listing 15.13. Kontrolka ListBox

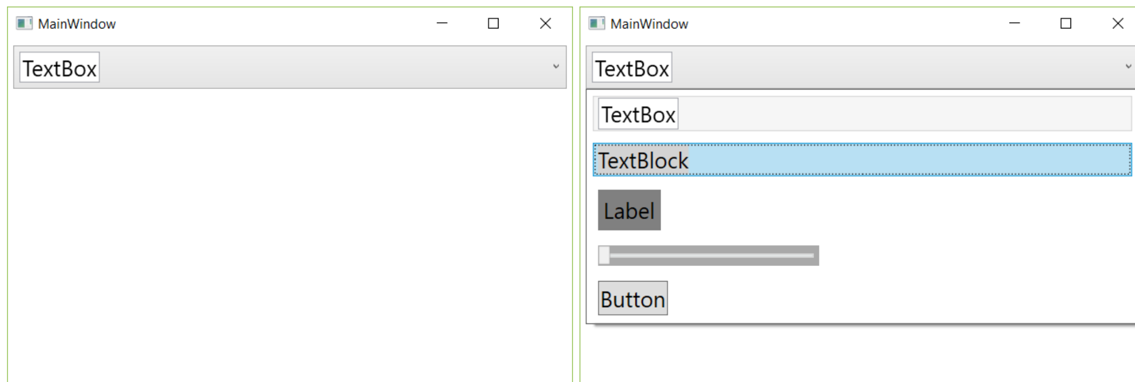
```
<Window x:Class="PojemnikiWPF.MainWindow"
        ...
        Title="MainWindow" Height="350" Width="525"
        FontSize="20">
    <StackPanel>
        <ListBox Margin="5" Padding="5">
            <ListBox.ItemContainerStyle>
                <Style TargetType="ListBoxItem">
                    <Setter Property="Margin" Value="5" />
                </Style>
            </ListBox.ItemContainerStyle>
            <TextBox Text="TextBox" />
            <TextBlock Text="TextBlock" Background="LightGray" />
            <Label Content="Label" Background="Gray" />
            <Slider Width="200" Background="DarkGray" />
            <Button Content="Button" />
        </ListBox>
    </StackPanel>
</Window>
```



Rysunek 15.10. Kontrolka ListBox

Listy pojawiły się już w tej książce. Łatwo się przekonać, że może ona bez problemu przechować zestaw kontrolek, które wyżej umieszczaliśmy w różnych pojemnikach (listing 15.13). Poza ListBox, w WPF są jeszcze dwie inne podobne kontrolki: ListView i ItemsControl. Wszystkie mogą wyświetlać listy elementów (także kontrolek), ale w ItemsControl nie możemy ich zaznaczać. Aby się o tym przekonać, wystarczy w kodzie z listingu 15.13 zmienić element ListBox na ItemsControl i uruchomić aplikację. Klasa ListBox dziedziczy po ItemsControl, dodając do niej właśnie możliwość zaznaczania na liście elementów (a przy okazji też paski przewijania). Z kolei ListView dziedziczy po ListBox, dodając możliwość zmieniania widoków, tj. sposobów, w jakie wyświetlane są elementy.

Ostatnią kontrolką, o której chcę tu wspomnieć, jest ComboBox. Zobaczymy ją, jeżeli w kodzie z listingu 15.11 ponownie zmienimy ListBox na ComboBox. Żadne inne zmiany nie są potrzebne. I w tej kontrolce możliwe jest zaznaczanie elementów — są one wówczas wyświetlane w jej podstawowym, „zwiniętym" widoku (rysunek 15.11).



Rysunek 15.11. Kontrolka ComboBox, czyli rozwijana lista

Szablony

Ważną rzeczą w przypadku list jest możliwość wpływania na to, co jest pokazywane w każdym elemencie listy. Można oczywiście wybrać własność, a nawet całą ścieżkę własności, której wartość jest prezentowana na liście (np. <ListBox ... DisplayMemberPath="ActualWidth">), ale to szablony dają nam pełnię swobody. Szablon pozwala zbudować element listy z dowolnych kontrolek. Domyślnie jest to po prostu etykieta TextBlock, zacznijmy wobec tego od odtworzenia tej domyślnej zawartości (listing 15.14).

Listing 15.14. Szablon wyświetlający jedną etykietę z informacją o szerokości kontrolki

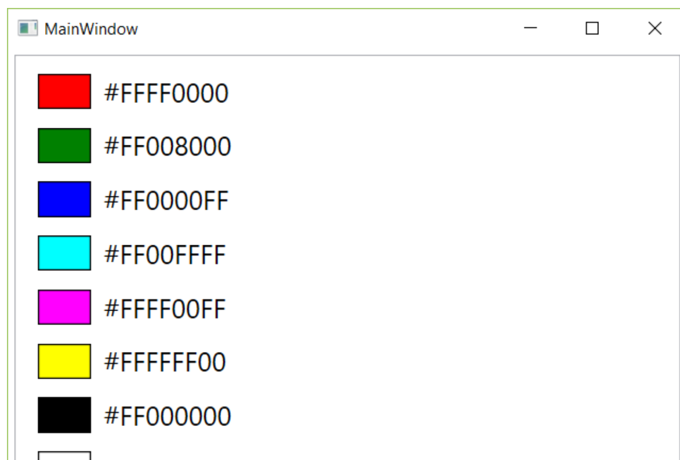
```
<ListBox Margin="5" Padding="5">
    <ListBox.ItemContainerStyle>
        <Style TargetType="ListBoxItem">
            <Setter Property="Margin" Value="5" />
        </Style>
    </ListBox.ItemContainerStyle>
    <ListBox.ItemTemplate>
        <DataTemplate>
            <TextBlock Text="{Binding ActualWidth}" Foreground="Navy" />
        </DataTemplate>
    </ListBox.ItemTemplate>
    <TextBox Text="TextBox" />
    <TextBlock Text="TextBlock" Background="LightGray" />
    <Label Content="Label" Background="Gray" />
    <Slider Width="200" Background="DarkGray" />
    <Button Content="Button" />
</ListBox>
```

W szablonie wiązania odnoszą się do oryginalnych elementów listy i ich własności. W powyższym przykładzie odwołujemy się do jednej z niewielu wspólnych własności kontrolek, a mianowicie szerokości udostępnianej przez własność `ActualWidth`. Niewiele więcej można zrobić, jeżeli lista zawiera elementy różnych typów. Zmieńmy jednak jej zawartość na np. listę pędzli, a więc elementów, które nie mają własnej reprezentacji w oknie, i zbudujmy szablon, który będzie je prezentował (listing 15.15).

Listing 15.15. Zmiana danych i szablonu

```
<ListBox Margin="5" Padding="5">
    <ListBox.ItemContainerStyle>
        <Style TargetType="ListBoxItem">
            <Setter Property="Margin" Value="5" />
        </Style>
    </ListBox.ItemContainerStyle>
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <Rectangle Width="40" Margin="0,0,10,0"
                           Stroke="Black" StrokeThickness="1"
                           Fill="{Binding}" />
                <TextBlock Text="{Binding}" Foreground="{Binding}" />
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
    <SolidColorBrush Color="Red" />
    <SolidColorBrush Color="Green" />
    <SolidColorBrush Color="Blue" />
    <SolidColorBrush Color="Cyan" />
    <SolidColorBrush Color="Magenta" />
    <SolidColorBrush Color="Yellow" />
    <SolidColorBrush Color="Black" />
    <SolidColorBrush Color="White" />
</ListBox>
```

Szablon składa się z prostokąta prezentującego kolor oraz etykiety prezentującej kod szesnastkowy koloru (rysunek 15.12). Zwróć uwagę na wiązanie własności `Fill` prostokąta. Nie podajemy w nim żadnej własności (formalnie jest to pominięty atrybut `Path`), więc własność ta jest wiązana do całego obiektu źródła, czyli pędzla. To ma sens, jednak gdy to samo robimy w przypadku własności `Text` etykiety, to już takiego sensu wydaje się nie mieć. Wskazanie obiektu pędzla prowadzi do użycia metody `ToString` i konwersji na łańcuch, do którego zapisywany jest kod szesnastkowy. Robię tak, bo w obiekcie pędzla nie ma żadnej własności udostępniającej przyjazną nazwę, którą można by wykorzystać.



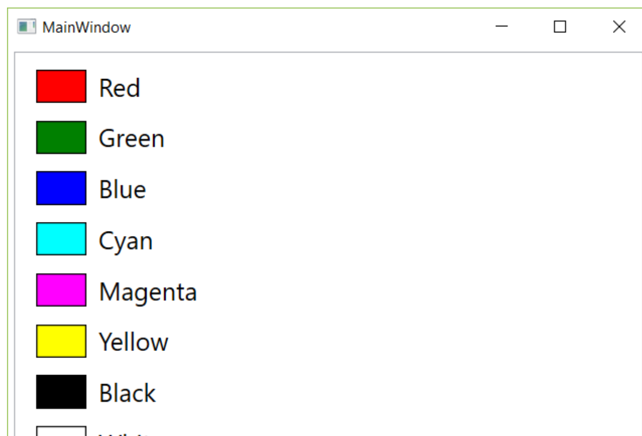
Rysunek 15.12. Lista korzystająca z szablonu do wyświetlania zawartości

Jak ten problem rozwiązać? Niech danymi prezentowanymi na liście nie będą pędzle, a nazwy kolorów (listing 15.16). Wówczas bez zmiany szablonu zmieni się wygląd listy. Etykiety będą po prostu prezentować te nazwy, a prostokąty — właściwe kolory (rysunek 15.13). W tym drugim przypadku nie jest potrzebny żaden konwerter, bo parser XAML jest „przyzwyczajony" do użycia nazw, które są automatycznie konwertowane z użyciem klasy `ColorConverter`. Zwróć uwagę na przestrzeń nazw, którą należy zadeklarować — używana jest tylko w kontrolce `ListBox`, wobec tego definiuje alias `s` tylko w jej obrębie.

Listing 15.16. Zmiana danych prezentowanych na liście

```
<Window x:Class="PojemnikiWPF.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:PojemnikiWPF"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525"
        FontSize="20">
    <StackPanel>
        <ListBox Margin="5" Padding="5"
                 xmlns:s="clr-namespace:System;assembly=mscorlib">
            <ListBox.ItemContainerStyle>
                <Style TargetType="ListBoxItem">
                    <Setter Property="Margin" Value="5" />
                </Style>
            </ListBox.ItemContainerStyle>
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <StackPanel Orientation="Horizontal">
                        <Rectangle Width="40" Margin="0,0,10,0"
                                   Stroke="Black" StrokeThickness="1"
                                   Fill="{Binding}" />
                        <TextBlock Text="{Binding}" />
                    </StackPanel>
                </DataTemplate>
            </ListBox.ItemTemplate>
```

```
            <s:String>Red</s:String>
            <s:String>Green</s:String>
            <s:String>Blue</s:String>
            <s:String>Cyan</s:String>
            <s:String>Magenta</s:String>
            <s:String>Yellow</s:String>
            <s:String>Black</s:String>
            <s:String>White</s:String>
        </ListBox>
    </StackPanel>
</Window>
```



Rysunek 15.13. Lista, w której źródłem jest kolekcja nazw

Zestaw przydatnych list

Jeżeli chcielibyśmy dać użytkownikowi wybór spośród większej liczby kolorów, wymienianie ich po kolei nie byłoby dobrym pomysłem. Czy można jakoś wykorzystać klasy `System.Windows.Media.Colors` lub `System.Windows.Media.Brushes` jako źródło danych listy? Możemy spróbować pewnej sztuczki, polegającej na użyciu klasy `ObjectDataProvider`, która potrafi z dowolnych danych uczynić źródło danych, do którego można określić wiązanie. Wykorzystamy przy tym mechanizm *Reflection*, za pomocą którego odczytamy listę własności klasy `Brushes`:

```
ObjectDataProvider odp = new ObjectDataProvider();
odp.ObjectInstance = typeof(Brushes).GetProperties();
```

To, co przypiszemy do własności `ObjectInstance`, możemy odczytać z własności `Data`, ale może to być źródłem w wiązaniu w kodzie XAML. W powyższym przykładzie zostanie tam umieszczony wynik wywołania metody `Type.GetProperties`, czyli tablica 141 informacji o własnościach klasy `Brushes` zapisanych w obiektach typu `System.Reflection.RuntimePropertyInfo` (rysunek 15.14). Z nich możemy pobrać nazwy tych własności, które są jednocześnie przyjaznymi nazwami kolorów. A to wszystko, czego potrzebujemy.

Name	Value	Type
odp.Data	{System.Reflection.PropertyInfo[141]}	object {System.Reflection.PropertyInfo[]}
[0]	{System.Windows.Media.SolidColorBrush AliceBlue}	System.Reflection.PropertyInfo {System.Reflection.RuntimePropertyInfo}
[1]	{System.Windows.Media.SolidColorBrush AntiqueWhite}	System.Reflection.PropertyInfo {System.Reflection.RuntimePropertyInfo}
[2]	{System.Windows.Media.SolidColorBrush Aqua}	System.Reflection.PropertyInfo {System.Reflection.RuntimePropertyInfo}
[3]	{System.Windows.Media.SolidColorBrush Aquamarine}	System.Reflection.PropertyInfo {System.Reflection.RuntimePropertyInfo}
[4]	{System.Windows.Media.SolidColorBrush Azure}	System.Reflection.PropertyInfo {System.Reflection.RuntimePropertyInfo}
[5]	{System.Windows.Media.SolidColorBrush Beige}	System.Reflection.PropertyInfo {System.Reflection.RuntimePropertyInfo}
[6]	{System.Windows.Media.SolidColorBrush Bisque}	System.Reflection.PropertyInfo {System.Reflection.RuntimePropertyInfo}
[7]	{System.Windows.Media.SolidColorBrush Black}	System.Reflection.PropertyInfo {System.Reflection.RuntimePropertyInfo}
[8]	{System.Windows.Media.SolidColorBrush BlanchedAlmond}	System.Reflection.PropertyInfo {System.Reflection.RuntimePropertyInfo}
[9]	{System.Windows.Media.SolidColorBrush Blue}	System.Reflection.PropertyInfo {System.Reflection.RuntimePropertyInfo}
[10]	{System.Windows.Media.SolidColorBrush BlueViolet}	System.Reflection.PropertyInfo {System.Reflection.RuntimePropertyInfo}
[11]	{System.Windows.Media.SolidColorBrush Brown}	System.Reflection.PropertyInfo {System.Reflection.RuntimePropertyInfo}
[12]	{System.Windows.Media.SolidColorBrush BurlyWood}	System.Reflection.PropertyInfo {System.Reflection.RuntimePropertyInfo}
[13]	{System.Windows.Media.SolidColorBrush CadetBlue}	System.Reflection.PropertyInfo {System.Reflection.RuntimePropertyInfo}

Rysunek 15.14. Dane pobrane z klasy Brushes w debugerze Visual Studio

W kodzie XAML ten sam efekt możemy uzyskać, definiując element:

```
<ObjectDataProvider ObjectInstance="{x:Type Brushes}" MethodName="GetProperties" />
```

Należy go tylko użyć jako źródła danych w wiązaniu z własnością `ListBox.ItemsSource`. Listing 15.17 pokazuje, jak to zrobić. Alternatywnie to źródło można wstawić do zasobów, a w kodzie listy jedynie się do niego odwołać — wówczas nie będą konieczne tak rozbudowane elementy realizujące wiązanie. Ponieważ danymi jest teraz kolekcja obiektów `RuntimePropertyInfo`, a nie łańcuchów, musimy także zmienić szablon danych listy tak, żeby korzystać z własności `Name` tych obiektów.

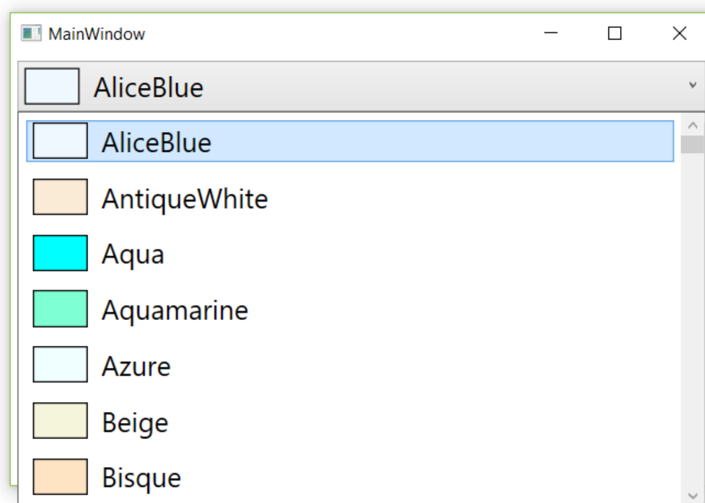
Listing 15.17. Pobranie wszystkich przyjaznych nazw kolorów z klasy Brushes

```
<Window x:Class="PojemnikiWPF.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:PojemnikiWPF"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525"
        FontSize="20">
    <StackPanel>
        <ComboBox Margin="5" Padding="5" SelectedIndex="0"
                  xmlns:s="clr-namespace:System;assembly=mscorlib" >
            <ComboBox.ItemContainerStyle>
                <Style TargetType="ComboBoxItem">
                    <Setter Property="Margin" Value="5" />
                </Style>
            </ComboBox.ItemContainerStyle>
            <ComboBox.ItemTemplate>
                <DataTemplate>
                    <StackPanel Orientation="Horizontal"
                                VirtualizingStackPanel.IsVirtualizing="True">
                        <Rectangle Width="40" Margin="0,0,10,0"
                                   Stroke="Black" StrokeThickness="1"
                                   Fill="{Binding Name}" />
                        <TextBlock Text="{Binding Name}" />
                    </StackPanel>
                </DataTemplate>
            </ComboBox.ItemTemplate>
```

```
                <ComboBox.ItemsSource>
                    <Binding>
                        <Binding.Source>
                            <ObjectDataProvider ObjectInstance="{x:Type Brushes}"
                                                MethodName="GetProperties" />
                        </Binding.Source>
                    </Binding>
                </ComboBox.ItemsSource>
            </ComboBox>
        </StackPanel>
    </Window>
```

Atrybut `VirtualizingStackPanel.IsVirtualizing="True"` powoduje, że nie jest tworzony widok wszystkich elementów listy, a jedynie te, które są potrzebne do wyświetlenia aktualnie widocznej jej zawartości. Przy dużej liczbie elementów to przyspiesza przygotowanie widoku.

Pojemnik `StackPanel` jest nieograniczony w dół i dlatego powoduje, że nie wyświetla się pasek przewijania listy. Aby móc zobaczyć pełną listę kolorów, musimy albo zmienić typ pojemnika na np. `Grid`, albo zmienić typ kontrolki na np. `ComboBox`. Wybrałem to drugie rozwiązanie (rysunek 15.15).

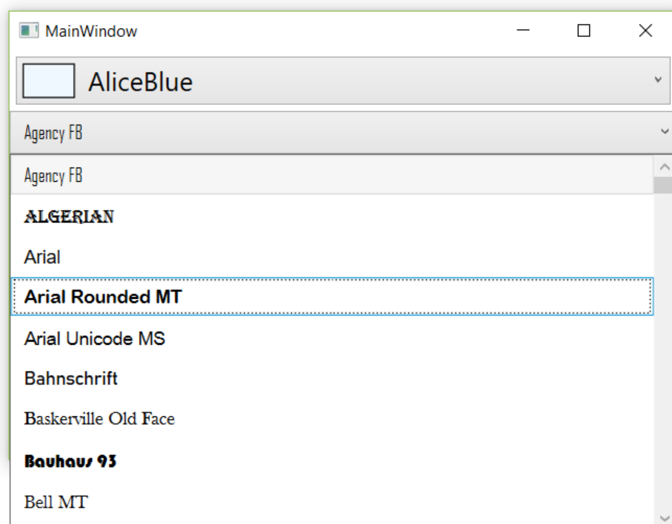


Rysunek 15.15. Lista wszystkich kolorów z klasy Brushes

O wiele prościej jest, gdy klasa udostępnia kolekcję elementów, których bez stosowania *Reflection* można użyć jako źródła danych listy. Tak jest np. w przypadku własności `SystemFontFamilies` zdefiniowanej w klasie `System.Windows.Media.Fonts`. Wówczas korzystając z szablonu, możemy nie tylko pokazać nazwę czcionki, ale również użyć jej do formatowania elementów listy. Pokazuje to listing 15.18 i rysunek 15.16.

Listing 15.18. Rozwijana lista wyświetlająca czcionki

```
<ComboBox ItemsSource="{x:Static Fonts.SystemFontFamilies}" SelectedIndex="0">
    <ComboBox.ItemTemplate>
        <DataTemplate>
            <StackPanel VirtualizingStackPanel.IsVirtualizing="True">
                <TextBlock Text="{Binding Source}" FontFamily="{Binding Source}"
                           FontSize="14" Margin="5" VerticalAlignment="Center" />
            </StackPanel>
        </DataTemplate>
    </ComboBox.ItemTemplate>
</ComboBox>
```



Rysunek 15.16. Rozwijana lista czcionek prezentująca nie tylko ich nazwy, ale również krój

Niestety, uzyskane w ten sposób czcionki nie będą posortowane. Aby je posortować, należy użyć elementu `CollectionViewSource`, wskazując je jako źródła danych i jednocześnie ustawiając jego własność `Source` na zbiór czcionek. Pokazuje to listing 15.19. Efekt jest widoczny na rysunku 15.16.

Listing 15.19. Sortowanie danych w źródle wiązania dla kontrolki ComboBox

```
<ComboBox Margin="5">
    <ComboBox.ItemTemplate>
        <DataTemplate>
            <StackPanel VirtualizingStackPanel.IsVirtualizing="True">
                <TextBlock Text="{Binding Source}" FontFamily="{Binding Source}"
                           FontSize="14" Margin="5" VerticalAlignment="Center" />
            </StackPanel>
        </DataTemplate>
    </ComboBox.ItemTemplate>
    <ComboBox.ItemsSource>
        <Binding>
            <Binding.Source>
                <CollectionViewSource
                    Source="{Binding Source={x:Static Fonts.SystemFontFamilies}}"
                    xmlns:cm="clr-namespace:System.ComponentModel;assembly=WindowsBase">
                    <CollectionViewSource.SortDescriptions>
                        <cm:SortDescription PropertyName="Source" />
                    </CollectionViewSource.SortDescriptions>
                </CollectionViewSource>
            </Binding.Source>
        </Binding>
    </ComboBox.ItemsSource>
</ComboBox>
```

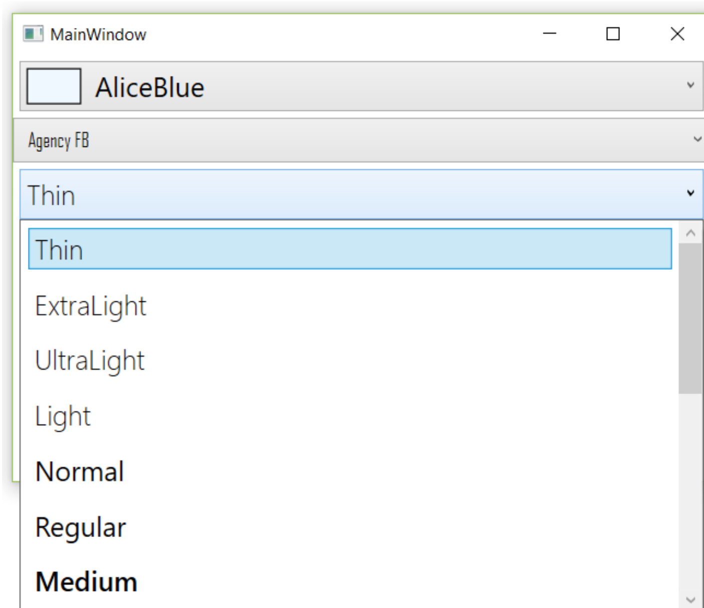
Wykorzystując jeden z powyższych schematów, możemy utworzyć także inne listy, np. listę grubości czcionki, które w WPF są bardzo liczne, albo listę stopnia pochylenia czcionki (listing 15.20, rysunek 15.17).

Listing 15.20. Kod kontrolki prezentującej grubości czcionki w WPF

```
<ComboBox Margin="5" Padding="5" SelectedIndex="0">
```

```
    <ComboBox.ItemTemplate>
        <DataTemplate>
            <StackPanel VirtualizingStackPanel.IsVirtualizing="True">
                <TextBlock Text="{Binding Name}"
                           FontWeight="{Binding Name}"
                           Margin="5" />
             </StackPanel>
        </DataTemplate>
    </ComboBox.ItemTemplate>
    <ComboBox.ItemsSource>
        <Binding>
            <Binding.Source>
                <ObjectDataProvider ObjectInstance="{x:Type FontWeights}"
                                    MethodName="GetProperties" />
            </Binding.Source>
        </Binding>
    </ComboBox.ItemsSource>
</ComboBox>

<ComboBox Margin="5" Padding="5" SelectedIndex="0">
    <ComboBox.ItemTemplate>
        <DataTemplate>
            <StackPanel VirtualizingStackPanel.IsVirtualizing="True">
                <TextBlock Text="{Binding Name}"
                           FontStyle="{Binding Name}"
                           Margin="5" />
            </StackPanel>
        </DataTemplate>
    </ComboBox.ItemTemplate>
    <ComboBox.ItemsSource>
        <Binding>
            <Binding.Source>
                <ObjectDataProvider ObjectInstance="{x:Type FontStyles}"
                                    MethodName="GetProperties" />
            </Binding.Source>
        </Binding>
    </ComboBox.ItemsSource>
</ComboBox>
```



Rysunek 15.17. Widok kontrolki prezentującej grubości czcionki

Nieco inaczej postąpimy w przypadku listy wyświetlającej możliwe ozdobniki czcionki (w praktyce oznacza to różnego rodzaju podkreślenia i przekreślenia). W tym przypadku każdy z nich może być użyty równocześnie z innymi, co oznacza, że użytkownik musi mieć możliwość wybrania dowolnego zbioru elementów. Dlatego w tym przypadku użyjemy listy `ListBox` z własnością `SelectionMode` ustawioną na `Multiple` (listing 15.21).

Listing 15.21. Lista dekoracji tekstu

```
<ListBox Margin="5" Padding="5" SelectionMode="Multiple">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel VirtualizingStackPanel.IsVirtualizing="True">
                <TextBlock Text="{Binding Name}"
                           TextDecorations="{Binding Name}"
                           Margin="5" />
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
    <ListBox.ItemsSource>
        <Binding>
            <Binding.Source>
                <ObjectDataProvider ObjectInstance="{x:Type TextDecorations}"
                                    MethodName="GetProperties" />
            </Binding.Source>
        </Binding>
    </ListBox.ItemsSource>
</ListBox>
```

Zadania

1. W pojemniku `RandomPanel`, w metodzie `ArrangeOverride` dodaj kod wykluczający nakładanie się kontrolek. Zwróć uwagę, że może to prowadzić do nieskończonej pętli, gdy na umieszczenie kolejnej kontrolki nie ma już miejsca. Należy zabezpieczyć się przed taką sytuacją.
2. Zdefiniuj własny pojemnik, który odtwarza działanie pojemnika `UniformGrid`.
3. Zdefiniuj element `ComboBox` wyświetlający kolory z bieżącego zestawu kolorów systemowych (klasa `SystemColors`).
4. W liście wyświetlającej kolory użyj sortowania po składowych RGB tak, żeby kolory występujące obok siebie zmieniały się w sposób płynny.