Windows Presentation Foundation 入門

Ver.1.0.3

目次

図 目次

図 目次

コード 目次

1 はじめに

この章では本書の目的および執筆環境を掲載します。

1.1 目的

本書は Windows Presentation Foundation（以降 WPF）の基本的な使い方などを共有し、WPF 開発技術力向上促進を目的としています。

1.2 注意

Web 上では本書に似た様々なドキュメントやサンプルコードがすでに多数あり、今さら同じような内容をまとめることは無駄なことかもしれませんが、自分なりの使い方や、自分が使っていたときにぶち当たった壁をどのように乗り越えてきたかなどを自分でまとめることで、WPF の理解をさらに深めることができるのではないかという目論見のため本書の執筆に至りました。

本書の前半部分では WPF が提供する標準コントロールの使用例をサンプルとして紹介しています。「8 WPF の基本的な開発手順」では、実際のアプリケーション開発をするときに必要となるデータバインディング機能に関して説明しています。WPF を初めて触る人は、コントロールの使い方を知るために初めから読み進めばいいと思います。これからアプリケーション開発をする人またはしている人は、一度「8 WPF の基本的な開発手順」で説明しているデータバインディング機能を知り、十分に活用してください。

本書の中で示すサンプルコードでは、コンパクトかつわかりやすくするために MainWindow.xaml.cs のようなコードビハインドへの記述が多く見られます。実際のアプリケーション開発においては、「8 WPF の基本的な開発手順」に示すようにデータバインディング機能を使用するため、コードビハインドではなく、ViewModel ないしは Model を用いることになります。

1.3 開発環境

本書は以下の環境で執筆しています。

・Windows7 Professional SP1 32 ビットオペレーティングシステム
・Visual Studio Professional 2013 Update5
・.NET Framework 4.6

2 WPF が提供するコントロール

本章では WPF が提供するコントロールがどのような分類、構造になっているかをおおまかに見ていきます。

2.1 WPF の主要なサブシステム

WPF の主なプログラミングモデルは、マネージコードを通じて公開されます。WPF で扱う UI コントロールの一番の根幹にあるオブジェクトは System.Threading.DispatcherObject クラスとなります。
WPF UI フレームワークの UI 要素におけるクラスの階層を下図に示します。ただし、図中に示しているコントロールは提供されているコントロールの一部です。

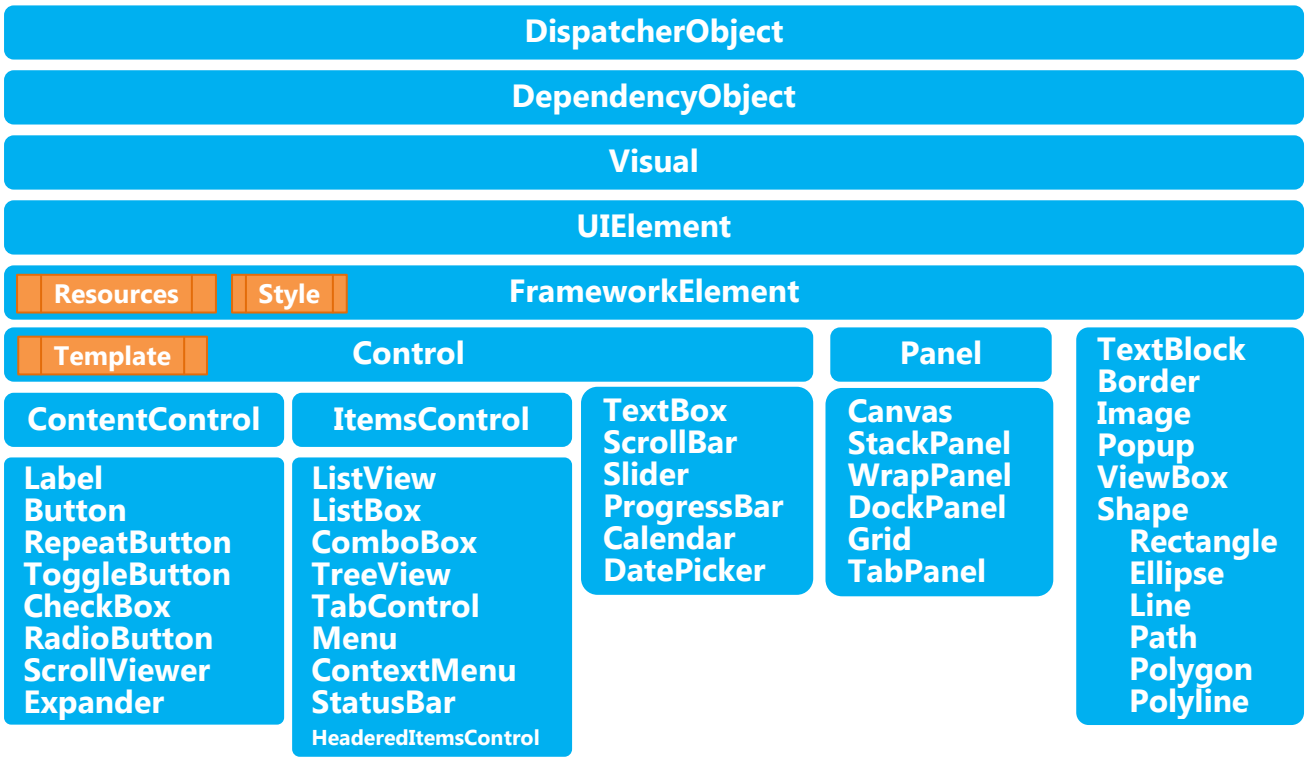


図 2.1：UI に関するコントロールの階層構造

上図で示したコントロールは Visual、UIElement、FrameworkElement が提供する仕組みによって描画処理がおこなわれるため、開発者にとってコード量が少なく済む一方で、コントロールの数が多くなるにつれて描画速度性能が著しく低下することがあります。この問題を回避するために、System.Windows.DependencyObject から派生する System.Windows.Media.Geometry などを使用する方法があります。このことに関しては後ほど説明します。

3 レイアウトに関するコントロール

WPF ではコントロールをレイアウトするための様々なコントロールが提供されています。場面に合ったコントロールを使用することで、使いやすく見映えの良い UI を構築しましょう。

3.1 Canvas コントロール

コントロールを指定座標位置に配置するコントロールです。WPF は相対座標を用いるレイアウトが一般的ですが、絶対座標による配置をしたいときもあります。そんなときに使うパネルがこの Canvas コントロールです。

3.1.1 Left、Top、Right、Bottom 添付プロパティ

Canvas コントロールでコントロールを配置するとき、その配置座標は Left、Top、Right、Bottom 添付プロパティで指定します。サンプルとして Button コントロールを配置します。

コード 3.1：左から 10、上から 50 の位置に Button コントロールを配置する

MainWindow.xaml

```
1 <Window x:Class="Sample_Canvas.MainWindow"
2         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4         Title="MainWindow" Height="200" Width="250"
5         WindowStartupLocation="CenterScreen">
6     <Canvas>
7         <Button Canvas.Left="10" Canvas.Top="50" Content="Click me!" />
8     </Canvas>
9 </Window>
```

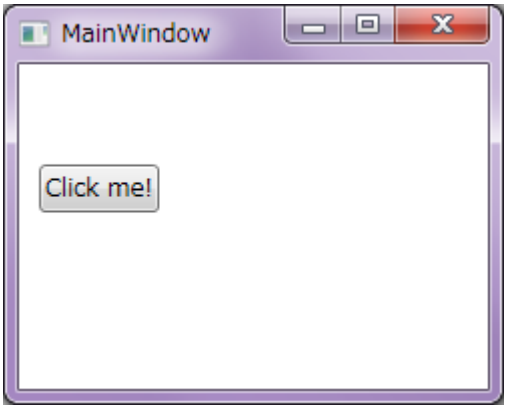


図 3.1：指定された位置にボタンが配置されている

指定している数値はデバイス非依存ピクセル、DIP という単位で表現されています。この単位はデバイスの DPI 設定に関わらず 1 DIP = 1/96 インチと定義されています。そのため、WPF では DPI を意識せずに UI を実装することができます。

3.2 StackPanel コントロール

コントロールを縦または横に並べるためのコントロールです。 "Stack" とは "積む" という意味がありますが、まさしくコントロールを積んでいくイメージです。はみ出たコントロールは、描画処理自体はおこなわれますが、画面上に表示されることはありません。

3.2.1 Orientation プロパティ

StackPanel はコントロールを積んでいくコントロールですが、その積む方向は Orientation プロパティで指定します。まず、サンプルとして Orientation プロパティを特に指定しない StackPanel に Button コントロールを 10 個入れてみます。

コード 3.2：StackPanel コントロールの使用例

MainWindow.xaml

```
1  <Window x:Class="Sample_StackPanel.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainWindow" Height="200" Width="250"
5          WindowStartupLocation="CenterScreen">
6      <StackPanel>
7          <Button Content="No. 1" />
8          <Button Content="No. 2" />
9          <Button Content="No. 3" />
10         <Button Content="No. 4" />
11         <Button Content="No. 5" />
12         <Button Content="No. 6" />
13         <Button Content="No. 7" />
14         <Button Content="No. 8" />
15         <Button Content="No. 9" />
16         <Button Content="No.10" />
17     </StackPanel>
18 </Window>
```

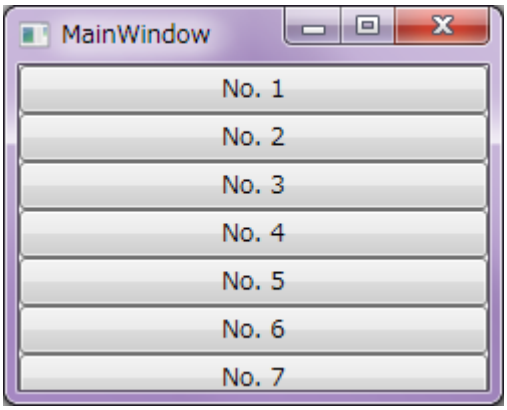


図 3.2：Vertical(既定値) を指定した場合

Orientation プロパティの既定値が Vertical であるため、特に指定しない場合は自動的に縦方向に積まれていきます。Orientation プロパティを Horizontal にすると、コントロールが水平方向に積まれていきます。

コード 3.3：Orientation プロパティを指定する

MainWindow.xaml

```
6  <StackPanel Orientation="Horizontal">
```

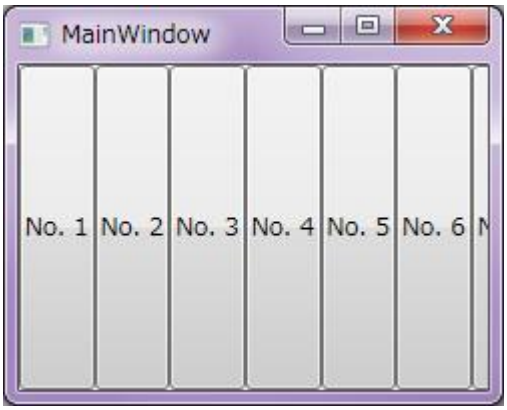


図 3.3：Horizontal を指定した場合

結果を見るとわかると思いますが、垂直方向に並べる場合は幅が、水平方向に並べる場合は高さが自動調整されて配置されます。

3.2.2 ScrollViewer コントロールによるスクロール表示

上記の例では単純に Button コントロールを並べただけですが、ウィンドウサイズが小さいと 10 個すべての Button コントロールが表示できず、ユーザーが操作することもできません。このような場合、スクロールバーを表示するという方法が一般的に用いられますが、これは ScrollViewer コントロールを次のように使用することで簡単に実現できます。

コード 3.4：ScrollViewer コントロールの追加

MainWindow.xaml

```
<Window x:Class="Sample_StackPanel.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="200" Width="250"
        WindowStartupLocation="CenterScreen">
    <ScrollViewer VerticalScrollBarVisibility="Auto">
        <StackPanel>
            <Button Content="No. 1" />
            <Button Content="No. 2" />
            <Button Content="No. 3" />
            <Button Content="No. 4" />
            <Button Content="No. 5" />
            <Button Content="No. 6" />
            <Button Content="No. 7" />
            <Button Content="No. 8" />
            <Button Content="No. 9" />
            <Button Content="No.10" />
        </StackPanel>
    </ScrollViewer>
</Window>
```

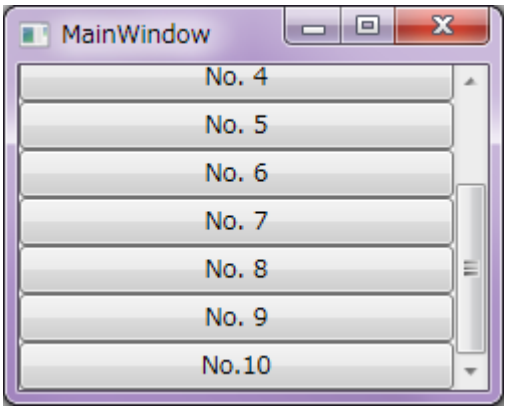


図 3.4：ScrollViewer コントロールによるスクロール

水平方向のスクロールバー表示を設定するプロパティは HorizontalScrollBarVisibility プロパティとなります。

コード 3.5：Orientation プロパティを指定する

MainWindow.xaml

```
6 <ScrollViewer HorizontalScrollBarVisibility="Auto">
7     <StackPanel Orientation="Horizontal">
```

図 3.5：Horizontal を指定した場合

3.3 WrapPanel コントロール

コントロールを縦または横に並べ、はみ出た場合は自動的に次の行または列に並べるコントロールです。

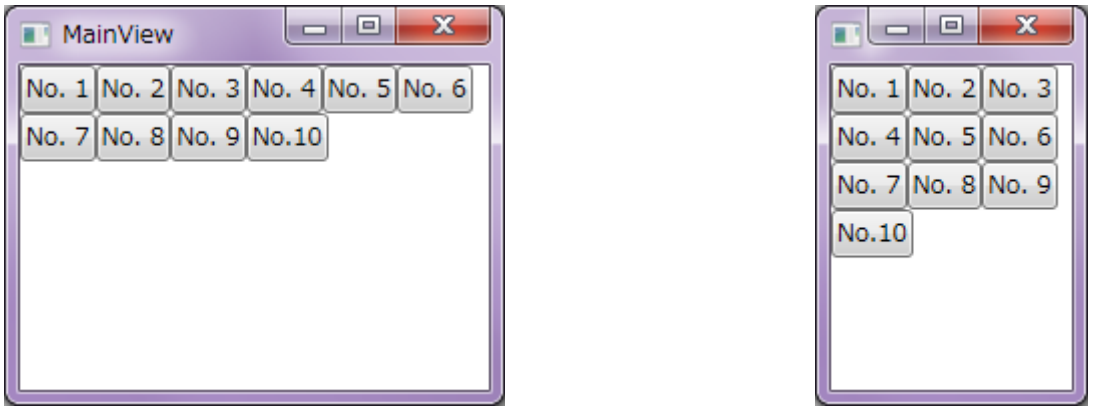
3.3.1 Orientation プロパティ

コントロールを並べる方向は Orientation プロパティで指定します。 サンプルとして Orientation プロパティを特に指定しない WrapPanel に Button コントロールを 10 個入れてみます。

コード 3.6：WrapPanel コントロールの使用例

MainWindow.xaml

```
1  <Window x:Class="Sample_WrapPanel.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          xmlns:sys="clr-namespace:System;assembly=mscorlib"
5          Title="MainView" Height="200" Width="250">
6      <WrapPanel>
7          <Button Content="No. 1" />
8          <Button Content="No. 2" />
9          <Button Content="No. 3" />
10         <Button Content="No. 4" />
11         <Button Content="No. 5" />
12         <Button Content="No. 6" />
13         <Button Content="No. 7" />
14         <Button Content="No. 8" />
15         <Button Content="No. 9" />
16         <Button Content="No.10" />
17     </WrapPanel>
18 </Window>
```



(a) 起動時　　(b) Window の幅を変更した場合

図 3.6：Horizontal(既定値) を指定した場合

Orientation プロパティの既定値は Horizontal であるため、特に指定しない場合は水平方向に積まれていきます。Window のサイズが変更されると、コントロールの配置が再計算され、ボタンの配置が自動的に変化します。Orientation プロパティを Vertical にすると、コントロールが垂直方向に積まれていきます。

コード 3.7：Orientation プロパティを指定する

MainWindow.xaml

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

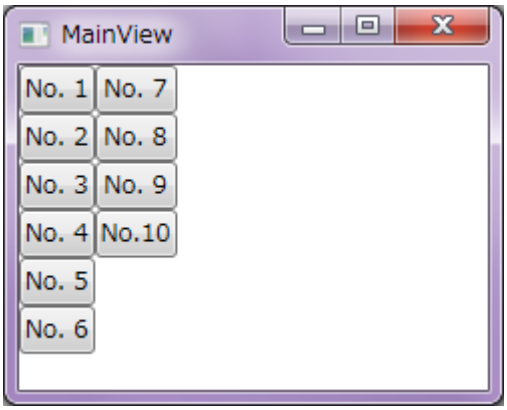


図 3.7：Vertical を指定した場合

WrapPanel コントロールで並べられるコントロールは、その最小サイズに自動調整されます。

3.4 DockPanel コントロール

コントロールを上下左右に張り付くように配置するコントロールです。

3.4.1 Dock 添付プロパティ

DockPanel コントロールでは、XAML で上から書いた順にコントロールの配置が決定されます。そのとき、各コントロールに指定された Dock 添付プロパティを参照して、指定された方向にそのコントロールを張り付けるように配置します。サンプルとして 4 つの Button コントロールを DockPanel に表示する例を示します。

コード 3.8：DockPanel コントロールの使用例

MainWindow.xaml

```
1  <Window x:Class="Sample_DockPanel.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainView" Height="200" Width="250">
5      <DockPanel>
6          <Button DockPanel.Dock="Top" Content="上に張り付くよ！" />
7          <Button DockPanel.Dock="Left" Content="左！" />
8          <Button DockPanel.Dock="Bottom" Content="下に張り付くよ！" />
9          <Button Content="残りの領域を占有するよ！" />
10     </DockPanel>
11 </Window>
```

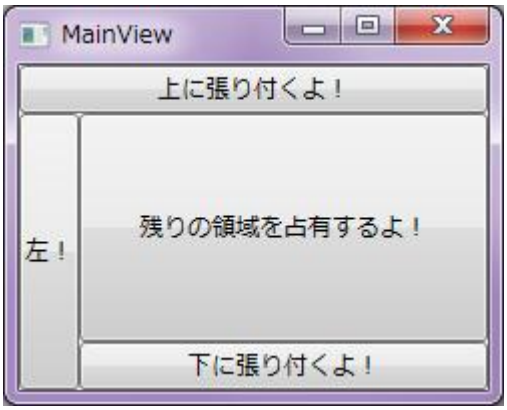


図 3.8：DockPanel コントロールの使用例

この例では、まず上に張り付くボタンを配置した後に、左に張り付くボタンを配置しているため、上に張り付いたボタンはウィンドウの左端から幅めいっぱいに配置されています。これを、左に張り付くボタンを先に記述すると少し結果が変わります。

コード 3.9：順序を入れ替えた場合

MainWindow.xaml

```
6          <Button DockPanel.Dock="Left" Content="左！" />
7          <Button DockPanel.Dock="Top" Content="上に張り付くよ！" />
```

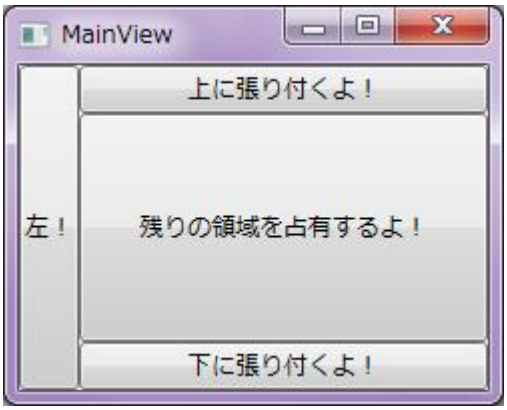


図 3.9：DockPanel コントロールの使用例

左に張り付くコントロールが先に記述されたため、上に張り付くボタンの幅が狭くなっていることがわかります。このように、配置する順序を入れ替えることによって得られるレイアウト結果が異なるため、XAML を記述するときには注意が必要です。

このような DockPanel コントロールは、メニューとステータスバーを持つ UI を実現するときによく使用されます。

コード 3.10：DockPanel コントロールの使用例 2

MainWindow.xaml

```
<Window x:Class="Sample_DockPanel.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="200" Width="250">
    <DockPanel>
        <Menu DockPanel.Dock="Top">
            <MenuItem Header="ファイル (_F)" />
            <MenuItem Header="ヘルプ (_H)" />
        </Menu>

        <StatusBar DockPanel.Dock="Bottom">
            <StatusBarItem Content="準備完了" />
        </StatusBar>

        <Grid Background="Cornsilk">
            <TextBlock Text="メインコンテンツ" />
        </Grid>
    </DockPanel>
</Window>
```

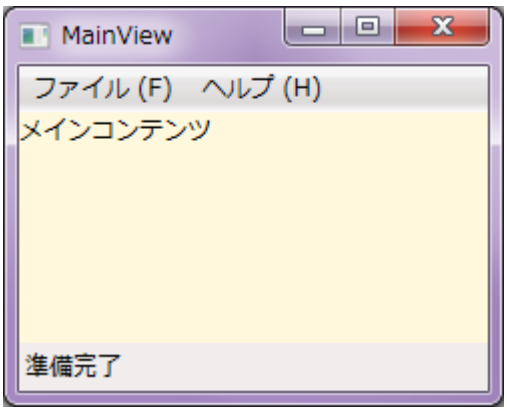


図 3.10：メニューとステータスバーを持つコントロールの実現例

3.5 Grid コントロール

コントロールを格子状に並べるためのコントロールです。RowDefinitions プロパティと ColumnDefintions プロパティで行と列の定義をおこない、Row 添付プロパティと Column 添付プロパティによって各コントロールの配置を決定します。

3.5.1 RowDefinitions プロパティと ColumnDefinitions プロパティ

RowDefinitions プロパティには定義したい行数分だけ RowDefinition を並べることで Grid コントロールの行に関する定義をおこないます。例えば 3 行の行を定義する場合は次のようになります。

コード 3.11：Grid コントロールの使用例

MainWindow.xaml

```
<Window x:Class="Sample_Grid.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="200" Width="250">
    <Grid Background="Cornsilk">
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
    </Grid>
</Window>
```

上記のように記述すると、XAML デザイナー上の表示が次のように 3 行分の領域として認識します。

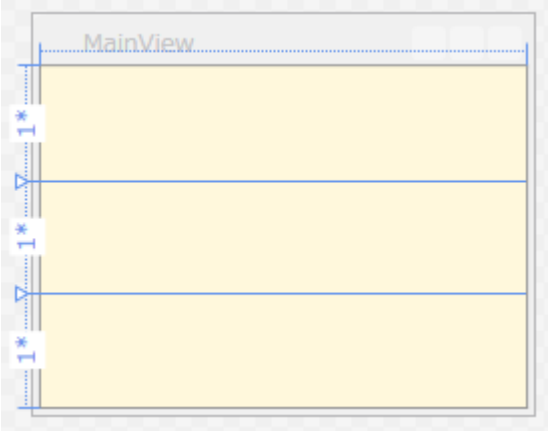


図 3.11：WPF デザイナー上の表示

行の高さは Height プロパティで変更できますが、特に指定しない場合は残っている領域を等分した高さになります。この比率を変更したい場合は "*" を使って表現します。例えば 2 行目を他の行の 2 倍の高さにしたい場合は "2*" と表記します。

コード 3.12：2 行目の高さを他の行の 2 倍にする

MainWindow.xaml

```
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition Height="2*" />
            <RowDefinition />
        </Grid.RowDefinitions>
```

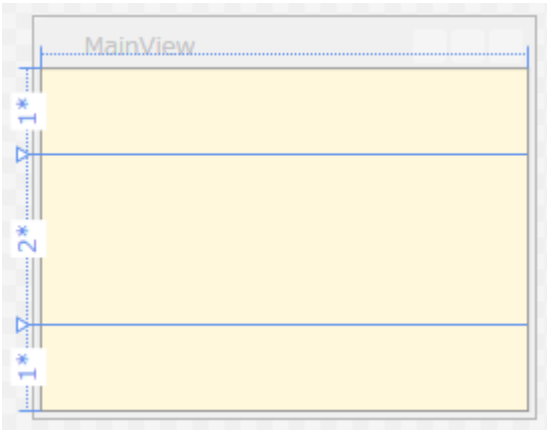


図 3.12 : 2 行目の高さが他の行の 2 倍になっている

もちろん絶対値で指定することもできます。

コード 3.13 : 1 行目を絶対値で指定する

MainWindow.xaml

```
6         <Grid.RowDefinitions>
7             <RowDefinition Height="50" />
8             <RowDefinition Height="2*" />
9             <RowDefinition />
10        </Grid.RowDefinitions>
```

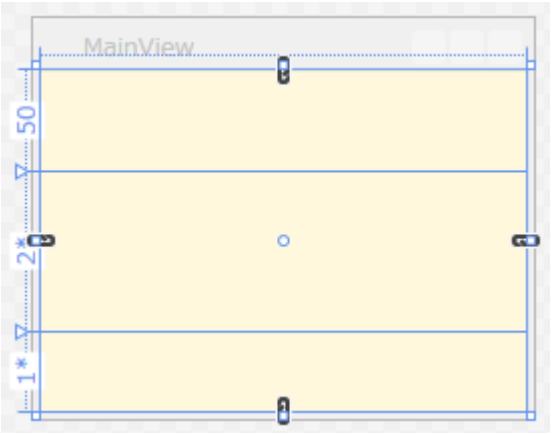


図 3.13 : 1 行目が指定されたので 3 行目と高さが違う

ここで、ウィンドウの高さを 200 から 300 に変更してみます。すると、絶対値で指定されている 1 行目は高さが変わらず、その他の行はウィンドウの高さに応じて高さが変わっていますが、2 行目が 3 行目の 2 倍の高さを持っていることに変わりはありません。

コード 3.14 : Window の高さを 300 に変更する

MainWindow.xaml

```
4         Title="MainView" Height="300" Width="250">
```

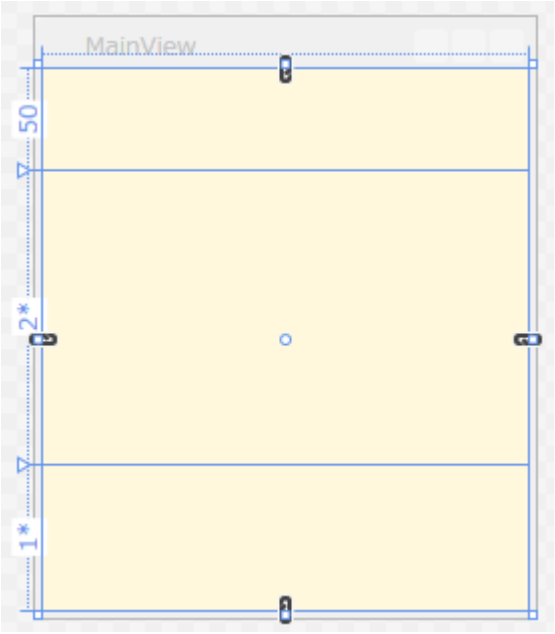


図 3.14：ウィンドウの高さにしたがって行の高さも自動調整される

試しに Button コントロールを置いてみます。

コード 3.15：1 行目を絶対値で指定する

MainWindow.xaml

```
1  <Window x:Class="Sample_Grid.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainView" Height="300" Width="250">
5      <Grid Background="Cornsilk">
6          <Grid.RowDefinitions>
7              <RowDefinition Height="50" />
8              <RowDefinition Height="2*" />
9              <RowDefinition />
10         </Grid.RowDefinitions>
11
12         <Button Content="Click me!"/>
13     </Grid>
14 </Window>
```

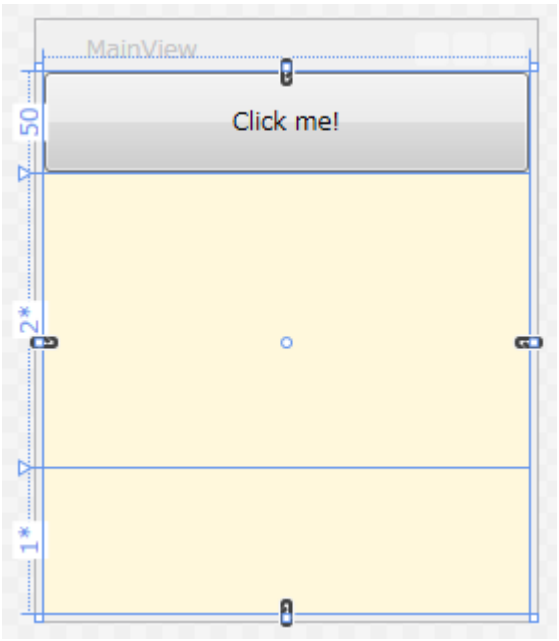


図 3.15：行全体に Button コントロールが配置された

Grid コントロール内でコントロールを配置した場合、特に指定しない限りそのコントロールは 1 行 1 列の位置に配置されます。

この状態では、Button コントロールが Grid コントロールに合わせてサイズが自動調整されています。これを、Grid コントロールが Button コントロールに合わせてサイズが自動調整されるようにするには、RowDefinitions の Height プロパティに "Auto" を指定します。

コード 3.16：Height に Auto を指定する

MainWindow.xaml

```
6         <Grid.RowDefinitions>
7             <RowDefinition Height="Auto" />
8             <RowDefinition Height="2*" />
9             <RowDefinition />
10        </Grid.RowDefinitions>
```

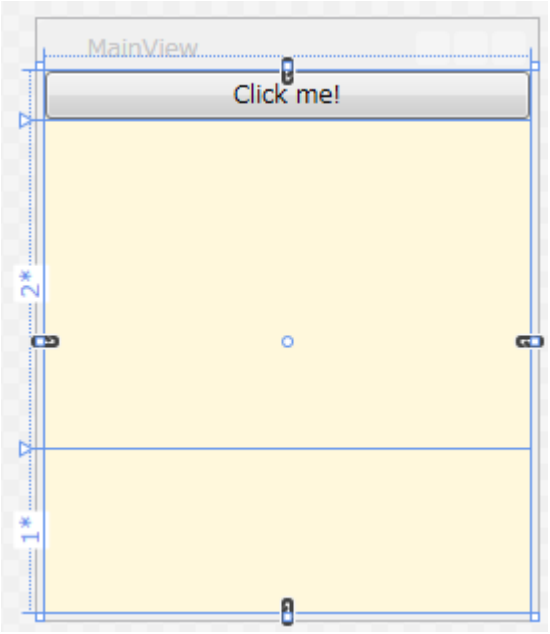


図 3.16：行の高さが Button コントロールに合わせて自動調整されている

また、MinHeight プロパティや MaxHeight プロパティによって行の高さの最小値または最大値を指定することもできます。ウィンドウサイズが変更されたときにレイアウトを崩したくないときなどに使用します。

コード 3.17：行の高さの最大値を指定している

MainWindow.xaml

```
6         <Grid.RowDefinitions>
7             <RowDefinition MaxHeight="40" />
8             <RowDefinition Height="2*" />
9             <RowDefinition />
10        </Grid.RowDefinitions>
```

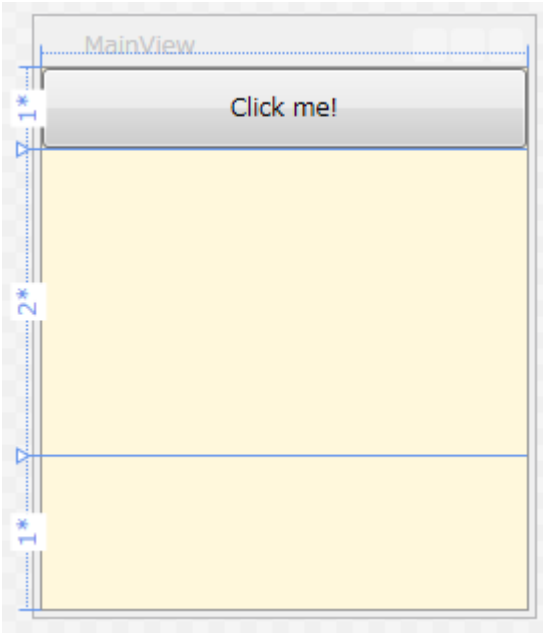


図 3.17：1 行目の高さが最大値となっているため 3 行目の高さより小さい

このように、Height プロパティを指定していないので本来は 3 行目と同じ高さとなるはずですが、MaxHeight プロパティによって高さを制限されているため、ウィンドウサイズによっては 3 行目の高さと異なるようになります。

以上が行に関する定義方法です。列に関しては "Row" の部分が "Column" に変わり、 "Height" の部分が "Width" に変わるだけで、 後はすべて同じなのでここでは説明を省略します。

3.5.2 Row 添付プロパティと Column 添付プロパティ

Grid 内のコントロールの配置を変更するには、Grid コントロールの Row 添付プロパティおよび Column 添付プロパティを使用します。添付プロパティとは、任意のオブジェクトに対して設定できるプロパティで、ここでは配置を指定したいコントロールに対して設定します。

サンプルとして次のような 3 行 3 列の Grid コントロールを対象とします。この Grid コントロールには既に Button コントロールを配置していますが、その位置は特に指定していません。

コード 3.18 : 3 行 3 列の Grid コントロール

MainWindow.xaml

```
1  <Window x:Class="Sample_Grid.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainView" Height="200" Width="250">
5      <Grid Background="Cornsilk">
6          <Grid.RowDefinitions>
7              <RowDefinition />
8              <RowDefinition />
9              <RowDefinition />
10         </Grid.RowDefinitions>
11         <Grid.ColumnDefinitions>
12             <ColumnDefinition />
13             <ColumnDefinition />
14             <ColumnDefinition />
15         </Grid.ColumnDefinitions>
16
17         <Button Content="(2, 3)" />
18     </Grid>
19 </Window>
```

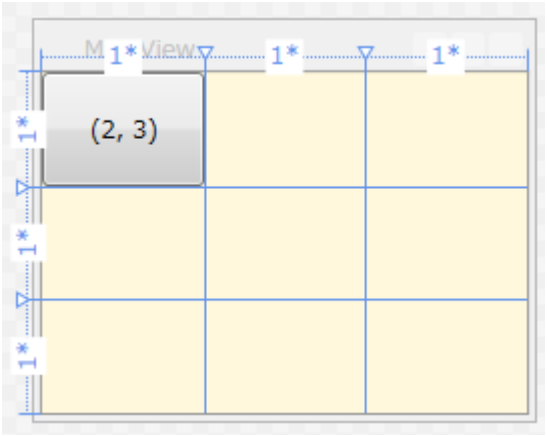


図 3.18 : Button コントロールは 1 行 1 列に配置されている

Button コントロールの配置を特に指定していないため、既定値である 1 行 1 列に表示されています。例えばこの Button コントロールを 3 行 2 列に表示させたい場合、Button コントロールに対して Grid の Row 添付プロパティおよび Column 添付プロパティを次のように指定します。

コード 3.19：添付プロパティを指定する

MainWindow.xaml

```
17          <Button Grid.Row="2" Grid.Column="1" Content="(2, 3)" />
```

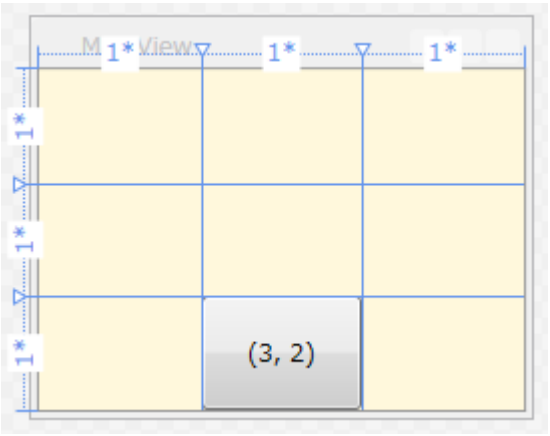


図 3.19：3 行 2 列に表示されている

行番号および列番号は 0 始まりとなるので、3 行 2 列を指定する場合は XAML 上では Row に 3、Column に 2 を指定します。

3.5.3 RowSpan 添付プロパティと ColumnSpan 添付プロパティ

Grid の中で行または列を結合させることもできます。例えば、前述のように 3 行 3 列に表示されている Button コントロールを右隣の行と結合させて配置したいときは、次のように ColumnSpan 添付プロパティを指定します。

コード 3.20：ColumnSpan 添付プロパティを指定する

MainWindow.xaml

```
17          <Button Grid.Row="2" Grid.Column="1" Grid.ColumnSpan="2" Content="(2, 3)" />
```

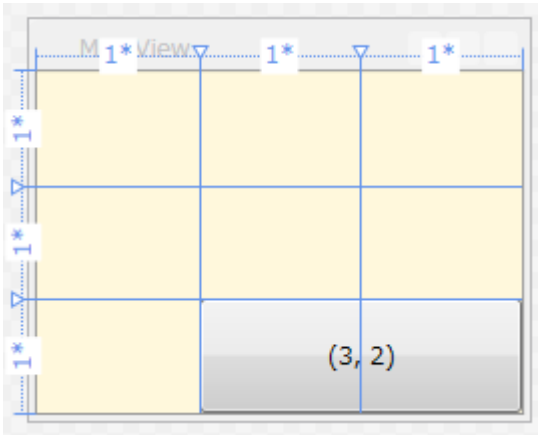


図 3.20：列が結合されて Button コントロールが配置されている

2 列を結合したいので ColumnSpan 添付プロパティに 2 を指定しています。行に関する結合は RowSpan 添付プロパティを使用して同様に指定します。

3.5.4 GridSplitter による高さまたは幅の動的変更

GridSplitter コントロールによって、Grid コントロールの行の高さまたは列の幅をマウス操作で変更できるようになります。次のコードは 3 行分の領域を持つ Grid コントロールの 2 行目に GridSplitter コントロールを配置することで、 1 行目と 3 行目の境界線を動かすような UI を実現しています。

コード 3.21：GridSplitter コントロールの配置

MainWindow.xaml

```
<Window x:Class="Sample_Grid.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="200" Width="250">
    <Grid Background="Cornsilk">
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition Height="Auto" />
            <RowDefinition />
        </Grid.RowDefinitions>

        <GridSplitter Grid.Row="1" Height="5" ResizeDirection="Rows"
HorizontalAlignment="Stretch" />
    </Grid>
</Window>
```

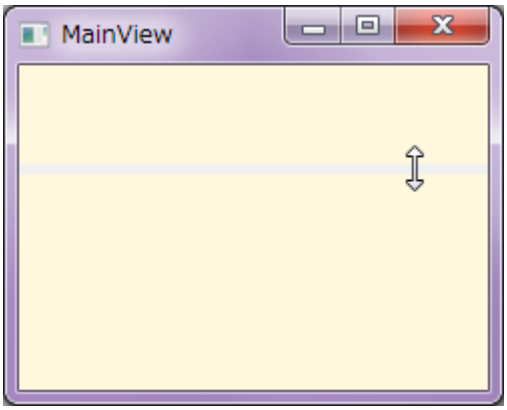


図 3.21：GridSplitter をマウスでドラッグできる

次のコードは 3 列の Grid コントロールで 1 列目と 3 列目に境界線を作るコードです。

コード 3.22：GridSplitter コントロールの配置

MainWindow.xaml

```
<Window x:Class="Sample_Grid.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="200" Width="250">
    <Grid Background="Cornsilk">
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <GridSplitter Grid.Column="1" Width="5" ResizeDirection="Columns"
HorizontalAlignment="Center" VerticalAlignment="Stretch" />
    </Grid>
</Window>
```

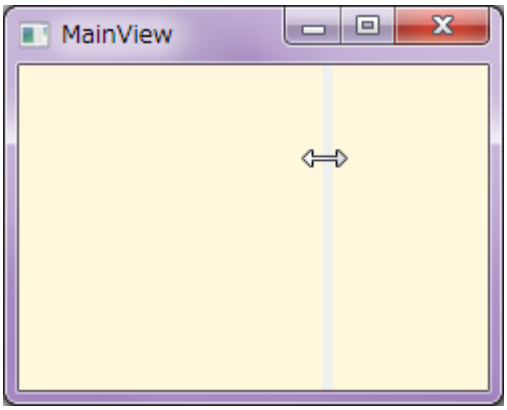


図 3.22：GridSplitter をマウスでドラッグできる

ここで注意しなければならないのは、HorizontalAlignment プロパティに Center を指定しているところです。もし HorizontalAlignment プロパティを指定しない場合、GridSplitter をドラッグ操作しようとしても、なぜか左に動きません。また、右に動かすと左のコンテンツと GridSplitter の間に変な余白ができてしまいます。下図は 1 列目に "ひだり" ボタン、3 列目に "みぎ" ボタンを配置し、GridSplitter の HorizontalAlignment プロパティを指定せずに起動した結果です。

コード 3.23：HorizontalAlignment プロパティを指定しない GridSplitter コントロールの配置

MainWindow.xaml

```
1  <Window x:Class="Sample_Grid.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainView" Height="200" Width="250">
5      <Grid Background="Cornsilk">
6          <Grid.ColumnDefinitions>
7              <ColumnDefinition />
8              <ColumnDefinition Width="Auto" />
9              <ColumnDefinition />
10         </Grid.ColumnDefinitions>
11
12         <Button Content="ひだり" />
13         <GridSplitter Grid.Column="1" Width="5" ResizeDirection="Columns"
   VerticalAlignment="Stretch" />
14         <Button Grid.Column="2" Content="みぎ" />
15     </Grid>
16 </Window>
```

図 3.23：HorizontalAlignment を指定せずに起動して右にドラッグした様子

3.6 パネルコントロールの入れ子

以上、基本的なパネルコントロールを紹介しましたが、実際にコントロールをレイアウトするときは、これらのパネルコントロールを入れ子にしていきます。例えば次のようなコードを書いてみましょう。

コード 3.24：パネルコントロールの入れ子

MainWindow.xaml

```
<Window x:Class="Sample_Panels.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">
    <DockPanel>
        <Button DockPanel.Dock="Top" Content="Top Area" />
        <Button DockPanel.Dock="Bottom" Content="Bottom Area" />
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition />
            </Grid.ColumnDefinitions>

            <ScrollViewer Width="100" Background="Cornsilk">
                <StackPanel>
                    <Button Content="Button A" Margin="10" />
                    <Button Content="Button B" Margin="10" />
                    <Button Content="Button C" Margin="10" />
                    <Button Content="Button D" Margin="10" />
                    <Button Content="Button E" Margin="10" />
                    <Button Content="Button F" Margin="10" />
                    <Button Content="Button G" Margin="10" />
                </StackPanel>
            </ScrollViewer>

            <DockPanel Grid.Column="1">
                <Button DockPanel.Dock="Top" Content="Sub Top Area 1" />
                <Button DockPanel.Dock="Top" Content="Sub Top Area 2" />
                <WrapPanel Background="CornflowerBlue">
                    <Button Content="Button 1" Margin="10" />
                    <Button Content="Button 2" Margin="10" />
                    <Button Content="Button 3" Margin="10" />
                    <Button Content="Button 4" Margin="10" />
                    <Button Content="Button 5" Margin="10" />
                    <Button Content="Button 6" Margin="10" />
                    <Button Content="Button 7" Margin="10" />
                    <Button Content="Button 8" Margin="10" />
                    <Button Content="Button 9" Margin="10" />
                    <Button Content="Button10" Margin="10" />
                </WrapPanel>
            </DockPanel>
        </Grid>
    </DockPanel>
</Window>
```

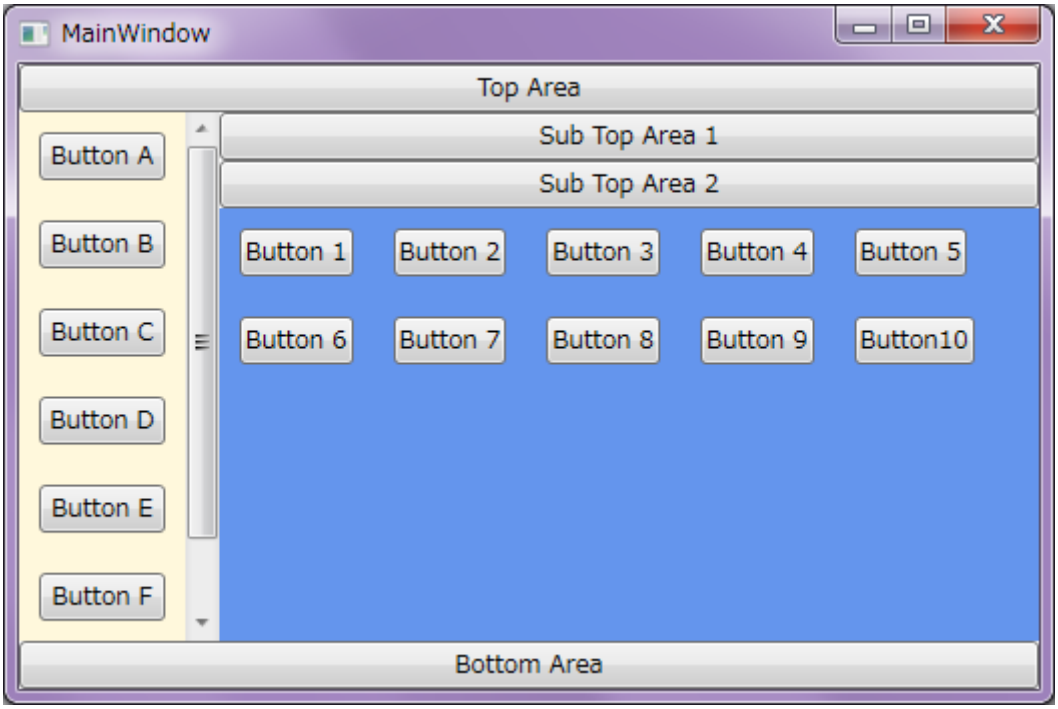


図 3.24：少し複雑なレイアウト例

5 行目の DockPanel コントロールを一番トップレベルのパネルコントロールとして、その中に Grid コントロールが 8 行目で配置されています。またその Grid コントロールの下には StackPanel コントロールや WrapPanel コントロールが配置されています。

複雑なレイアウトでも、慣れてくるとどのパネルコントロールを使えばいいかが自然と見えてくるようになります。また、ひとつのレイアウトでも実現方法はいくつもあります。上記のサンプルは一番最初に DockPanel コントロールを使っていますが、Grid コントロールでも同じことができます。いろいろ試してパネルコントロールを使いこなしてください。

4 ボタンに関するコントロール

この章ではボタンに関するコントロールについて紹介します。

4.1 Button コントロール

ユーザー指令を受け取るもっとも典型的なボタンのコントロールです。 ボタンの中にはテキスト以外にも様々なコンテンツを配置できます。

4.1.1 Content プロパティ

Content プロパティに文字列を指定することで、ボタンにテキストを表示させることができます。

コード 4.1 : Content プロパティに文字列を指定する

MainWindow.xaml

```
1  <Window x:Class="Sample_Button.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainView" Height="200" Width="250">
5      <StackPanel>
6          <Button Content="Click me!" />
7      </StackPanel>
8  </Window>
```

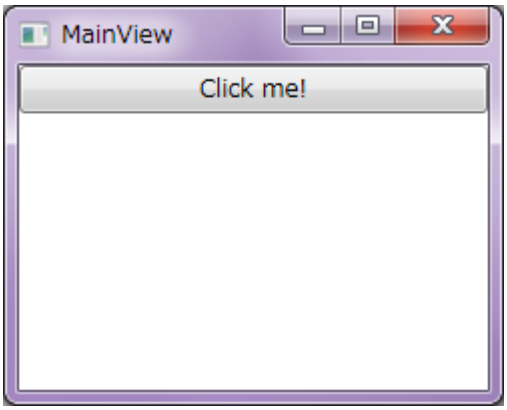


図 4.1 : テキストが表示される

テキスト以外にも様々なコントロールを配置させることができます。例えば次のように別のコントロールを乗せることも簡単にできます。

コード 4.2 : Content プロパティに別のコントロールを指定する

MainWindow.xaml

```
1  <Window x:Class="Sample_Button.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainView" Height="200" Width="250">
5      <StackPanel>
6          <Button>
7              <Border BorderThickness="3" BorderBrush="Green" Padding="4">
8                  <StackPanel Orientation="Horizontal">
```

```
 9                     <Ellipse Fill="Green" Width="16" Height="16" Margin="0,0,10,0" />
10                     <TextBlock Text="Click me!" VerticalAlignment="Center" />
11                 </StackPanel>
12             </Border>
13         </Button>
14     </StackPanel>
15 </Window>
```

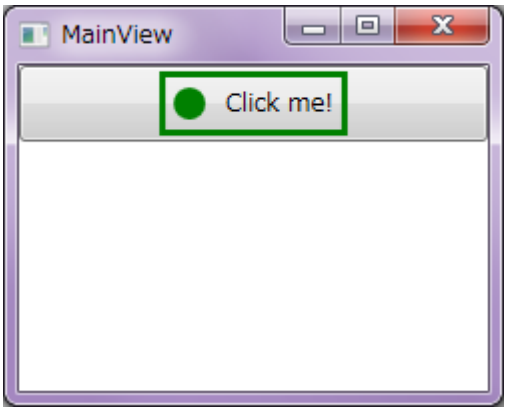


図 4.2：別のコントロールがボタン上に配置される

<Button>～</Button> で挟まれた部分が Content プロパティの値として指定されます。これは、Button コントロールの基本クラスである ContentControl コントロールの ContentPropertyAttribute に Content プロパティが指定されているためです。

実際やるかどうかは別として、同じやり方でボタン上に ListBox コントロールや動画なども乗せることができます。

4.1.2 Command プロパティ

Command プロパティに ICommand インターフェースを実装したクラスを指定することで、ボタンをクリックしたときに指定したクラスの Execute() メソッドが呼ばれるようになります。詳しくは「8.5 ICommand インターフェースの自前実装と具体例」を参照してください。

4.2 RepeatButton コントロール

Button コントロールはクリックするごとにクリックイベントが発生しますが、マウスで押している間ずっとクリックイベントが繰り返し発生し続けるボタンがこの RepeatButton コントロールです。標準コントロールでは ScrollBar コントロールなどに利用されています。

4.2.1 Content プロパティ

見かけ上は Button コントロールと変わらず、Content プロパティも Button コントロールと同じ挙動を示します。詳しくは「4.1.1 Content プロパティ」を参照してください。

4.2.2 Delay プロパティと Interval プロパティ

マウスで押している間ずっとクリックイベントが発生しますが、その繰り返しが始まるタイミングを Delay プロパティで、繰り返す間隔を Interval プロパティで調整できます。設定単位はどちらもミリ秒です。

コード 4.3 : RepeatButton コントロールの使用例

MainWindow.xaml

```
1  <Window x:Class="Sample_RepeatButton.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainView" Height="200" Width="250">
5      <StackPanel>
6          <RepeatButton Content="Repeat Button" Delay="500" Interval="1000"
   Click="RepeatButton_Click" />
7          <Slider Name="slider1" Minimum="0" Maximum="1000" />
8          <TextBlock Text="{Binding Value, ElementName=slider1}" />
9      </StackPanel>
10 </Window>
```

コード 4.4 : RepeatButton コントロールのイベントハンドラ

MainWindow.xaml.cs

```
1  namespace Sample_RepeatButton
2  {
3      using System.Windows;
4
5      /// <summary>
6      /// MainWindow.xaml の相互作用ロジック
       /// </summary>
7      public partial class MainWindow : Window
8      {
9          public MainWindow()
10         {
               InitializeComponent();
           }

           /// <summary>
           /// RepeatButton の Click イベントハンドラ
           /// </summary>
           /// <param name="sender">イベント発行元</param>
           /// <param name="e">イベント引数</param>
           private void RepeatButton_Click(object sender, RoutedEventArgs e)
           {
               this.slider1.Value++;
           }
       }
   }
```

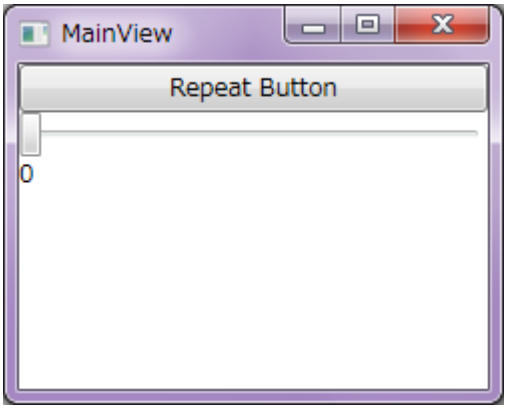


図 4.3：ボタンを押すとスライダーの値が増えていく

RepeatButton コントロールの Click イベントのイベントハンドラに RepeatButton_Click メソッドを登録しており、この中で Slider コントロールの値をインクリメントしています。したがって、RepeatButton コントロールを押し続けると、スライダーの値が増え続けることになります。

今、RepeatButton コントロールの Delay プロパティが 500、Interval プロパティが 1000 となっているため、一度ボタンを押してから 500[ms] 後にまたクリックイベントが発生し、それ以降は 1000[ms] ごとにクリックイベントが発生するようになります。

4.3 ToggleButton コントロール

ボタンが押された状態を記憶するコントロールが ToggleButton コントロールです。一度ボタンを押すと、押された状態を保持し、もう一度ボタンを押すと元に戻ります。

コード 4.5：ToggleButton コントロールの使用例

MainWindow.xaml

```
<Window x:Class="Sample_ToggleButton.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="200" Width="250">
    <StackPanel>
        <ToggleButton x:Name="toggle1" Content="Toggle Button" />
        <TextBlock Text="{Binding IsChecked, ElementName=toggle1}" />
    </StackPanel>
</Window>
```

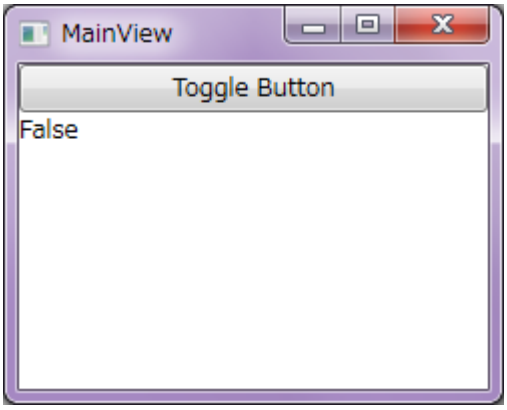


図 4.4：ボタンの状態がテキストで表示される

上記のサンプルコードでは、ToggleButton コントロールに名前を付け、その IsChecked プロパティを TextBlock コントロールからデータバインディング機能で参照しています。

ToggleButton コントロールは自分の状態を IsChecked プロパティで公開しています。つまり、一度ボタンを押すと IsChecked プロパティが true になり、もう一度押すと false に戻ります。IsChecked プロパティが true の場合、視覚的にはボタンが押しこまれた状態のままとなります。

4.3.1 IsThreeState プロパティ

また、IsChecked プロパティは true/false 以外に、null という状態となることもできます。実は、IsChecked プロパティは bool 型ではなく Nullable<bool> 型（コードではよく bool? と書く）、つまり null 許容型の bool 型であるからです。

デフォルトでは、ToggleButton コントロールをクリックすると IsChecked プロパティは true と false の値に切り替わるだけですが、IsThreeState プロパティを true にすると、false → true → null → false … の順に切り替わるようになります。

4.4 CheckBox コントロール

IsChecked プロパティで true または false の状態を示すコントロールです。その挙動は ToggleButton コントロールと同じで、IsThreeState プロパティを true にすることで、IsChecked プロパティを null に切り替えることができるようになります。

コード 4.6：CheckBox コントロールの使用例

MainWindow.xaml

```
1  <Window x:Class="Sample_CheckBox.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainView" Height="200" Width="250">
5      <StackPanel>
6          <CheckBox x:Name="checkbox1" Content="Check me!" IsThreeState="True" />
7          <TextBlock Text="{Binding IsChecked, ElementName=checkbox1}" />
8          <ToggleButton Content="Toggle Button" IsChecked="{Binding IsChecked,
   ElementName=checkbox1}" />
9      </StackPanel>
10 </Window>
```

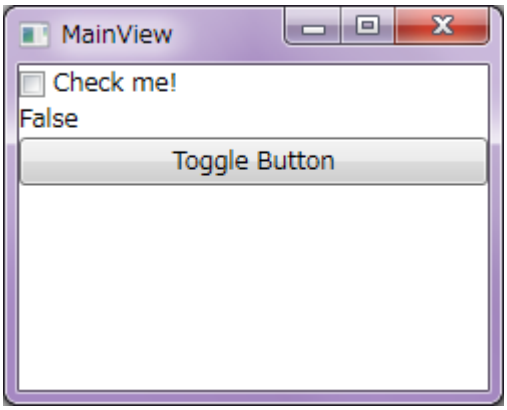


図 4.5：チェックの状態がテキストで表示される

実は CheckBox コントロールは ToggleButton コントロールから派生しているコントロールです。したがって、Button コントロールと同様、Command プロパティを持っています。つまり、CheckBox コントロールでも Button コントロールと同じように ICommand インターフェースを実装したクラスによってコマンド処理をおこなうことができます。しかし、IsChecked プロパティが変更されるタイミングよりも、Command プロパティによってコマンド処理されるタイミングが先となるかは厳密には保証されていません。よって、CheckBox コントロールのチェック状態が変更されるときに処理する内容は、IsChecked プロパティが変更されるタイミングに合わせておこなうようにするのが確実なようです。

4.5 RaidoButton コントロール

複数の選択項目に対してどれかひとつだけを選択させたい場合によく使われるのがこの RadioButton コントロールです。

4.5.1 グループ

コード 4.7：RadioButton コントロールの使用例

MainWindow.xaml

```
<Window x:Class="Sample_RadioButton.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="200" Width="250">
    <StackPanel>
        <RadioButton Content="Radio 1-1" />
        <RadioButton Content="Radio 1-2" />
        <RadioButton Content="Radio 1-3" />
        <Separator />
        <RadioButton Content="Radio 2-1" />
        <RadioButton Content="Radio 2-2" />
    </StackPanel>
</Window>
```

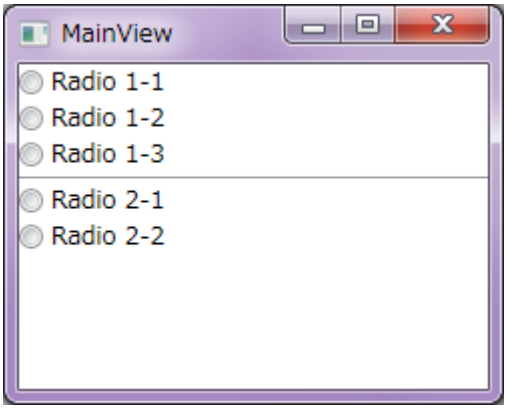


図 4.6：複数の RadioButton コントロール

このサンプルでは、すべての RadioButton コントロールが同じグループに属しているため、どれか一つしか選択できません。Separator コントロールでわけているように複数のグループにしたい場合は、GroupName プロパティを使用します。同一グループの RadioButton コントロールにはすべて同じ名前を指定する必要があるため、XAML コードが少し冗長に見えてしまいます。

コード 4.8：GroupName プロパティによるグルーピング

MainWindow.xaml

```
<Window x:Class="Sample_RadioButton.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="200" Width="250">
    <StackPanel>
        <RadioButton Content="Radio 1-1" GroupName="group1" />
        <RadioButton Content="Radio 1-2" GroupName="group1" />
        <RadioButton Content="Radio 1-3" GroupName="group1" />
        <Separator />
        <RadioButton Content="Radio 2-1" GroupName="group2" />
        <RadioButton Content="Radio 2-2" GroupName="group2" />
    </StackPanel>
```

```
13 </Window>
```

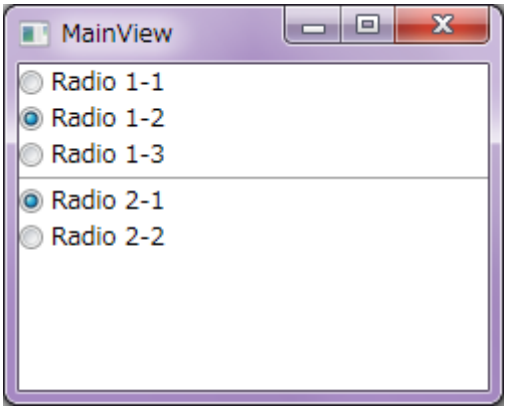


図 4.7：複数の RadioButton コントロールが別グループにわけられている

また、次のように RadioButton コントロールを別のパネルコントロールに配置することで自動的にグルーピングされるようです。

コード 4.9：パネルコントロールによるグルーピング

MainWindow.xaml

```
1  <Window x:Class="Sample_RadioButton.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainView" Height="200" Width="250">
5      <StackPanel>
6          <RadioButton Content="Radio 1-1" />
7          <RadioButton Content="Radio 1-2" />
8          <RadioButton Content="Radio 1-3" />
9          <Separator />
10         <StackPanel>
11             <RadioButton Content="Radio 2-1" />
12             <RadioButton Content="Radio 2-2" />
13         </StackPanel>
14     </StackPanel>
15 </Window>
```

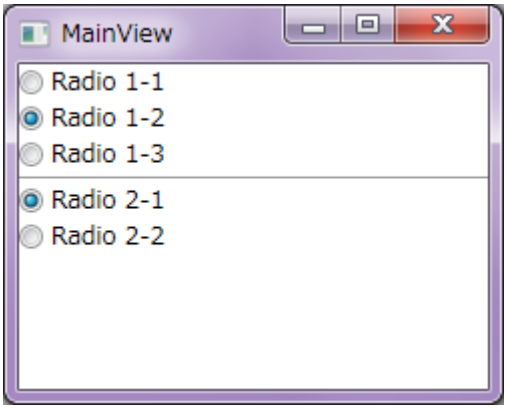


図 4.8：属するパネルコントロールによって別グループにわけられている

実際に使うときは、パネルコントロールによって RadioButton コントロールのグループ分けをするようにし、どうしても異なるパネルコントロールに配置しなければならない場合に GroupName プロパティを使用すると効率が良いように思います。

5 データの表示に関するコントロール

この章ではデータを表示するためのコントロールについて紹介します。

5.1 TextBlcok コントロール

テキストを表示するための一番シンプルなコントロールです。

5.1.1 Text プロパティと Inlines プロパティ

TextBock コントロールでテキストを表示する方法は Text プロパティを指定する方法と、Inlines プロパティを指定する方法の 2 通りあります。

コード 5.1：Text プロパティの使用例

MainWindow.xaml

```
1  <Window x:Class="Sample_TextBlock.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainView" Height="200" Width="250">
5      <StackPanel>
6          <TextBlock Text="サンプルテキスト" Background="Cornsilk" />
7          <TextBlock Background="LightBlue">
8              サンプルテキスト
9          </TextBlock>
10         <TextBlock Text="Text プロパティの値">
11             さらにテキストを指定
12         </TextBlock>
13     </StackPanel>
14 </Window>
```

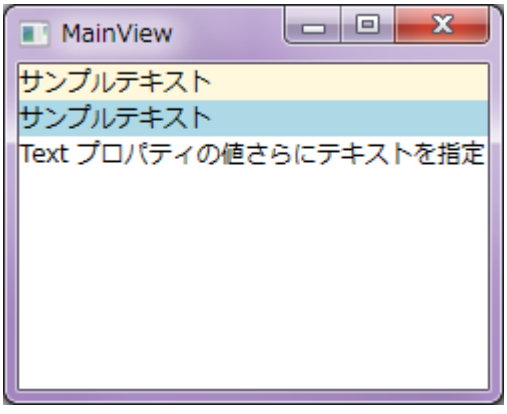


図 5.1：Text プロパティによるテキスト表示

Inlines プロパティは Text プロパティのように属性として記述することはできません。使用するときは <TextBlock>～</TextBlock> で挟む形で指定します。
また、Text プロパティと Inlines プロパティを同時に指定した場合は、Text プロパティに指定したテキストに加えて Inlines プロパティに指定したテキストが同時に表示されるようなので、使用時には注意が必要です。

Text プロパティはシンプルに 1 行だけのテキストを表示する場合に使用します。Inlines プロパティは改行や一部フォント色の変更など細かい指定をする場合に使用します。

5.1.2 Inlines プロパティによるテキストの構成方法

Inlines プロパティにそのままテキストを指定すると Text プロパティを使用した場合と同様の結果が得られますが、Inline 要素を活用することで多彩なテキスト表現が実現できます。

例えば次のような記述では一部のフォント色を変更できます。

コード 5.2：Inlines プロパティの使用例

MainWindow.xaml

```
<Window x:Class="Sample_TextBlock.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="200" Width="250">
    <StackPanel>
        <TextBlock Text="赤文字青文字ただの文字" Background="Cornsilk" />
        <TextBlock Background="LightBlue">
            <Run Text="赤文字" Foreground="Red" />
            <Run Text="青文字" Foreground="Blue" />
            <Run Text="ただの文字" />
        </TextBlock>
    </StackPanel>
</Window>
```

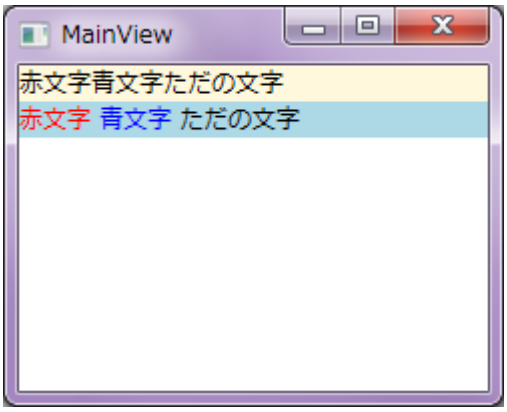


図 5.2：一部のフォント色が変更されている

Inlines プロパティでは Run クラスによってテキストを指定します。このとき、Inlines プロパティは InlineCollection というコレクションであるため、複数の Run クラスを並べて配置することで、InlineCollection に子要素を追加することができます。したがって、上記のサンプルでは Run クラスを 3 つ持つ InlineCollection を TextBlock コントロールの Inlines プロパティに指定していることになります。

それぞれの Run クラスに別々の Foreground プロパティを指定することで、一部のフォント色が変更された 1 行のテキストが表示できます。ただし、Run クラスでテキストを区切ると、その区切りに若干の余白が生まれ、Text プロパティによって同じ文字を指定したときより幅広となるため、画面設計時には注意が必要です。

複数行のテキストにしたい場合は、LineBreak クラスを使用します。

コード 5.3：LineBreak クラスの使用例

MainWindow.xaml

```
<Window x:Class="Sample_TextBlock.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="200" Width="250">
    <StackPanel>
        <TextBlock>
```

```
7             <Run Text="あけまして" />
8             <LineBreak />
9             <Run Text="おめでとうございます。" />
10        </TextBlock>
11    </StackPanel>
12 </Window>
```

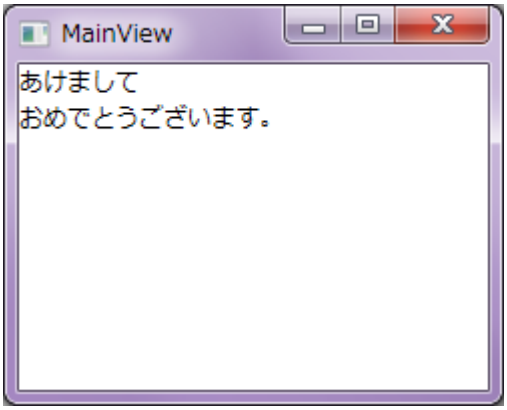


図 5.3：文字列を改行して表示する

あるデータをテキストとして表示する場合、Text プロパティにデータバインディングすることで表示することができますが、そのデータを説明するためのテキストを同時に表示したい場合、別の TextBlock コントロールの Text プロパティで表示する方法と、ひとつの TextBock コントロールの Inlines プロパティを使用して表示する方法の 2 通りがあります。

コード 5.4：データバインディングによるデータ表示

MainWindow.xaml

```
1  <Window x:Class="Sample_TextBlock.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainView" Height="200" Width="250">
5      <StackPanel>
6          <Slider x:Name="slider1" TickFrequency="1" IsSnapToTickEnabled="True" />
7
8          <StackPanel Orientation="Horizontal">
9              <TextBlock Text="只今の値は " VerticalAlignment="Bottom" />
10             <TextBlock Text="{Binding Value, ElementName=slider1}" FontSize="24"
   Foreground="LightSteelBlue" VerticalAlignment="Bottom" />
11             <TextBlock Text=" でございます。" VerticalAlignment="Bottom" />
12         </StackPanel>
13
14         <TextBlock>
15             <Run Text="只今の値は" />
16             <Run Text="{Binding Value, ElementName=slider1}" FontSize="24"
   Foreground="LightSteelBlue" />
17             <Run Text="でございます。" />
18         </TextBlock>
19     </StackPanel>
20 </Window>
```

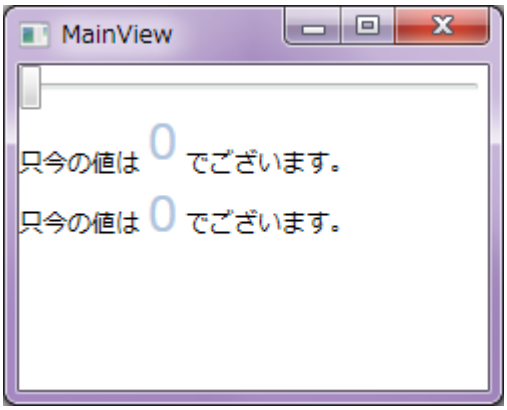


図 5.4：Inlines プロパティによるテキスト表示ならベースラインが共通なので綺麗に見える

一見どちらも同じように見えますが、Inlines プロパティを使用したほうが、同じベースライン上でテキストが表示されているため、レイアウトとして綺麗に見えます。また、XAML の記述としても Text プロパティによる実現では無駄にパネルコントロールを配置したり、テキスト表示とは関係のないプロパティ値を指定したりしないといけないため煩雑になってしまいます。

Run クラスや LineBreak クラスの他にも様々なクラスによってテキスト表示を装飾できます。

コード 5.5：その他のクラスの使用例

MainWindow.xaml

```
<Window x:Class="Sample_TextBlock.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="200" Width="250">
    <StackPanel>
        <TextBlock>
            <Run Text="Plain text" />
            <LineBreak />
            <Bold>
                <Run Text="Bold text" />
            </Bold>
            <LineBreak />
            <Hyperlink>
                <Run Text="Hyperlink text" />
            </Hyperlink>
            <LineBreak />
            <Italic>
                <Run Text="Italic text" />
            </Italic>
            <LineBreak />
            <Span>
                <Run Text="Span" />
            </Span>
            <LineBreak />
            <Underline>
                <Run Text="Underline" />
            </Underline>
        </TextBlock>
    </StackPanel>
</Window>
```

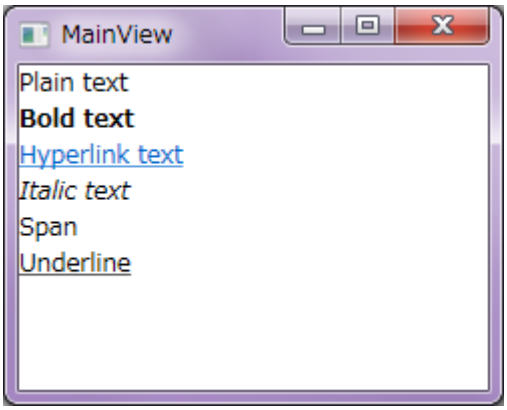


図 5.5：TextBlock コントロールだけでもいろいろな装飾ができる

5.1.3 TextAlignment プロパティ

"テキストを中央揃えにしたい" ということはよくあることですが、これを実現する方法は実は 2 通りあります。ひとつは TextAlignment プロパティを使用する方法です。

まずシンプルに Window コントロールに TextBock コントロールをひとつだけ配置します。

コード 5.6：TextBlock のみを配置

MainWindow.xaml

```
1 <Window x:Class="Sample_TextBlock.MainWindow"
2         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4         Title="MainView" Height="200" Width="250">
5     <TextBlock Text="Sample Text" Background="LightSkyBlue" />
6 </Window>
```

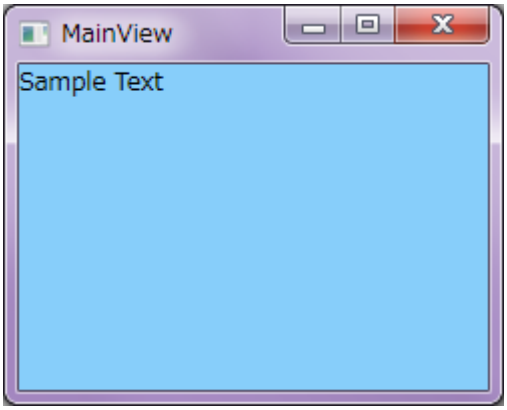


図 5.6：Window コントロールいっぱいに広がる TextBlock コントロール

背景色でもわかるように、TextBlock コントロールは親パネルである Window コントロール全体に延びるようにして配置されています。この状態で TextAlignment プロパティに Center を指定してみましょう。

コード 5.7：TextAlignment プロパティによる中央揃え

MainWindow.xaml

```
1 <Window x:Class="Sample_TextBlock.MainWindow"
2         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4         Title="MainView" Height="200" Width="250">
5     <TextBlock Text="Sample Text" TextAlignment="Center" Background="LightSkyBlue" />
6 </Window>
```

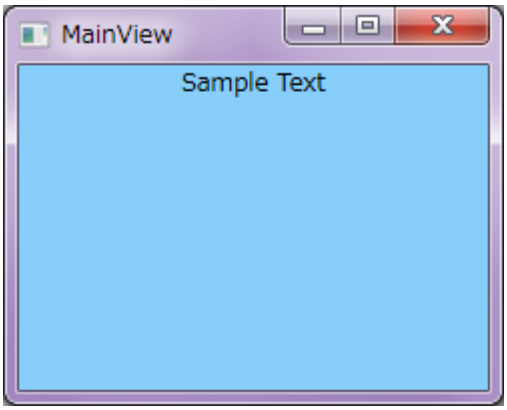


図 5.7：テキストが TextBlock コントロールの中央に配置される

文字通りテキストが TextBlock コントロールの中央に表示されるようになります。

一方、TextAlignment プロパティを指定せずに、HorizontalAlignment プロパティによってテキストを中央に表示することもできます。

コード 5.8：HorizontalAlignment プロパティによる中央揃え

MainWindow.xaml

```
1 <Window x:Class="Sample_TextBlock.MainWindow"
2         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4         Title="MainView" Height="200" Width="250">
5     <TextBlock Text="Sample Text" HorizontalAlignment="Center" Background="LightSkyBlue" />
6 </Window>
```

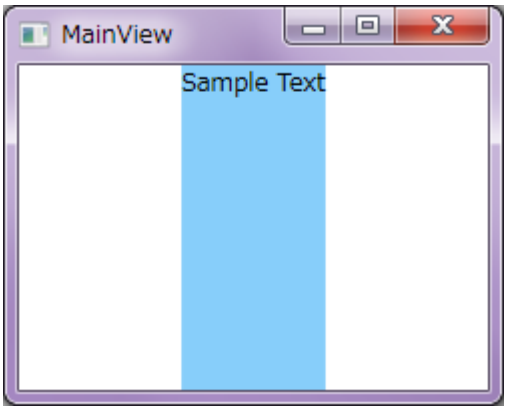


図 5.8：TextBlock コントロールが Window コントロールの中央に配置される

HorizontalAlignment プロパティは FrameworkElement クラスで定義されているプロパティで、コントロールの水平位置を調整するためのプロパティです。このプロパティに Center を指定しているため、TextBlcok コントロールが親コントロールである Window コントロールに対して中央の位置になるように配置されるようになります。

テキストの表示状態だけを見るとどちらのプロパティで変更しても同じ結果ですが、コントロールの配置として異なることに注意しなければなりません。例えばこの TextBlock コントロールに MouseDown イベントを登録した場合、HorizontalAlignment プロパティで中央に配置してしまうと、余白部分となったところをマウスでクリックされてもイベントが発生しないからです。

TextAlignment プロパティは水平方向のテキスト位置を調整するためのプロパティですが、垂直方向のテキスト位置を調整するためのプロパティはありません。垂直方向はコントロールの配置を調整する VerticalAlignment プロパティを使用するしかないようです。

コード 5.9：VerticalAlignment プロパティによる中央揃え

MainWindow.xaml

```
<Window x:Class="Sample_TextBlock.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="200" Width="250">
    <TextBlock Text="Sample Text" VerticalAlignment="Center" Background="LightSkyBlue" />
</Window>
```

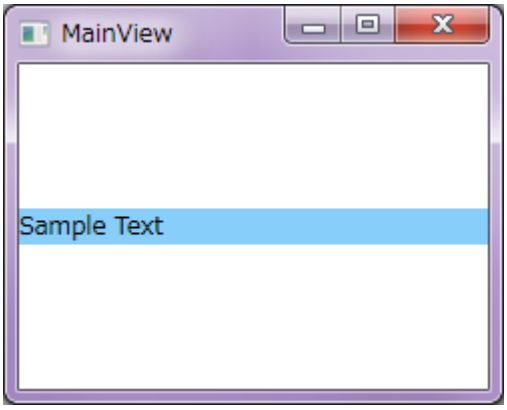


図 5.9：TextBlock コントロールが Window コントロールの垂直方向の中央に配置される

しかし、これではやはり TextBlock コントロールの領域が自動調整されてしまうので、具合が悪い場合があります。こういった場合は、例えば Border コントロールで TextBlock コントロールをラッピングするなどして、別のコントロールを配置し、イベントなどはラッピングした Border コントロールのほうで認識するようにしたほうがいいかもしれません。

コード 5.10：Border コントロールによるラッピング

MainWindow.xaml

```
<Window x:Class="Sample_TextBlock.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="200" Width="250">
    <Border Background="Cornsilk">
        <TextBlock Text="Sample Text" VerticalAlignment="Center" Background="LightSkyBlue" />
    </Border>
</Window>
```

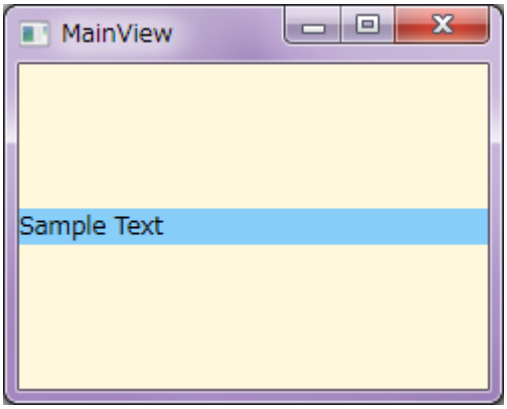


図 5.10：Border コントロールの中で垂直方向の中央に配置される TextBlock コントロール

5.2 Label コントロール

5.2.1 Target プロパティ

例えば TextBox コントロールを配置したとき、それがなんのための文字入力なのかを説明するテキストを配置することは一般的にあります。このとき、Label コントロールを使用してテキストを配置すると、アクセスキーを利用することができるようになり非常に便利です。アクセスキーとは、メニューなどによく表示されている "(F)" のように、アルファベットによるキーボードショートカットキーです。

以下のサンプルで Label コントロールによるアクセスキーを確認しましょう。

コード 5.11：Label によるアクセスキー機能の実現

MainWindow.xaml

```
<Window x:Class="Sample_Label.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="200" Width="250">
    <StackPanel>
        <StackPanel Orientation="Horizontal">
            <Label Content="Name(_N) : " Target="{Binding ElementName=textbox_name}"
HorizontalAlignment="Right" />
            <TextBox x:Name="textbox_name" Width="150" Margin="0,2" />
        </StackPanel>

        <StackPanel Orientation="Horizontal">
            <Label Content="Age(_A) : " Target="{Binding ElementName=textbox_age}"
HorizontalAlignment="Right" />
            <TextBox x:Name="textbox_age" Width="150" Margin="0,2" />
        </StackPanel>

        <!-- ダメな使用例 -->
        <Label Target="{Binding ElementName=textbox_memo}">
            <StackPanel Orientation="Horizontal">
                <!-- アクセスキーとして認識されない -->
                <TextBlock Text="Memo(_M) : " />
                <TextBox x:Name="textbox_memo" Width="150" Margin="2,0" />
            </StackPanel>
        </Label>
    </StackPanel>
</Window>
```

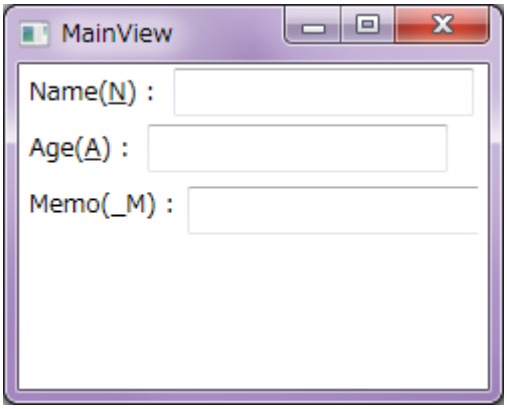


図 5.11：Alt + キーで指定した TextBox コントロールにフォーカスが移る

Label コントロールの Content プロパティに "_F" というように、"_" アンダーバーと組み合わせたアルファベットがアクセスキーとして自動認識されます。アクセスキーは Alt を押しながら指定のアルファベットのキーを

押すことで認識されます。そして、Target プロパティにコントロールを指定することで、アクセスキーが押されたときに指定されたコントロールにフォーカスが移るようになります。上記のサンプルでは、Alt を押しながら "N" または "A" を押したとき、それぞれの隣の TextBox コントロールにフォーカスが移ります。

TextBlock コントロールではテキストを表示することはできますが、アクセスキーに対応していないため、上記のサンプルのように "_M" と指定した部分はそのまま表示されてしまいますし、「Alt + M」キーを押してもフォーカスが移ることはありません。

5.3 TextBox コントロール

ユーザーがテキストを入力できるコントロールです。Text プロパティが表示するテキストあるいは入力されたテキストとなります。

5.3.1 IsReadOnly プロパティ

TextBox コントロールは本来ユーザーに入力してもらうためのコントロールですが、条件によっては入力させたくない場合があります。このとき、IsReadOnly プロパティを true にすることによって、テキストを選択することはできますが、入力を受け付けなくなるようになります。元々入力させたくはないけれど、そのテキストをコピーできるようにしておきたい場合なども、TextBlock コントロールではなく、IsReadOnly プロパティを true にした TextBox コントロールを使うこともあります。

5.3.2 AcceptsReturn プロパティ

通常の TextBox コントロールは 1 行のテキストのみを表示しますが、AcceptsReturn プロパティを true に指定することで、Enter キーによる改行ができるようになります。

5.3.3 HorizontalScrollBarVisibility プロパティ

横方向に長いテキストが入力されている場合、カーソル操作でテキスト表示をスクロールすることもできますが、HorizontalScrollBarVisibility プロパティを設定することで、スクロールバーによるスクロールもできるようになります。

5.3.4 VerticalScrollBarVisibility プロパティ

AcceptsReturn プロパティを true にして複数行の表示をおこなっているとき、TextBox コントロールの高さからはみ出てしまう場合、VerticalScrollBarVisibility プロパティを設定しておくことで、縦方向のスクロールバーでスクロールできるようになります。

コード 5.12 : Image コントロールの使用例

MainWindow.xaml

```
1  <Window x:Class="Sample_TextBox.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainView" Height="300" Width="300">
5      <StackPanel>
6          <TextBox x:Name="textbox1" Text="Input something." />
7          <TextBlock Text="{Binding Text, ElementName=textbox1}" />
8          <TextBox Text="Read-only text." IsReadOnly="True" />
9          <TextBox Text="長いテキストが入力されると自動的に水平方向のスクロールバーが表示されるようになります。" HorizontalScrollBarVisibility="Auto" />
10         <TextBox Text="改行できるようになります。" AcceptsReturn="True" />
11         <TextBox Text="改行していくとスクロールバーが表示されます。" AcceptsReturn="True" VerticalScrollBarVisibility="Auto" Height="80" />
12     </StackPanel>
13 </Window>
```

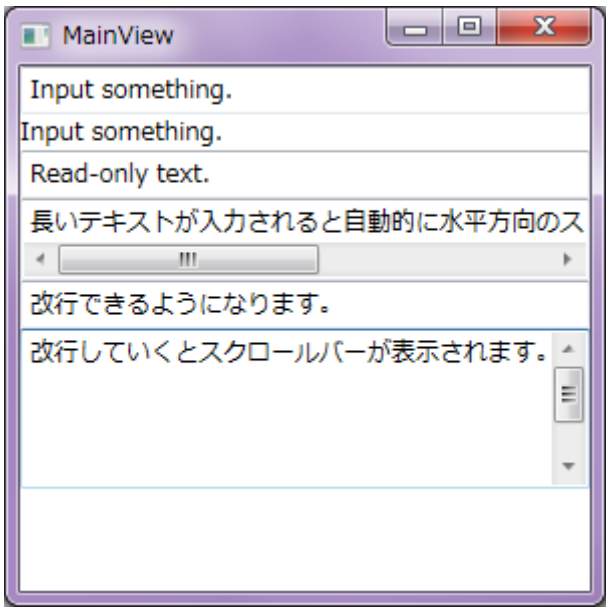


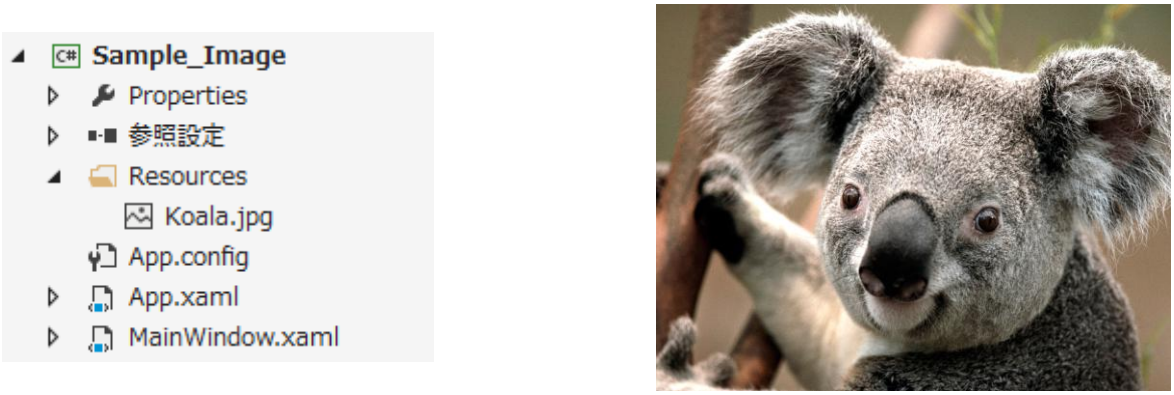
図 5.12：TextBox コントロールの使用例

5.4 Image コントロール

画像を表示するときに使うコントロールです。

5.4.1 Source プロパティ

画像のソースを指定します。ここではサンプル画像として Koala.jpg という画像ファイルをプロジェクトに追加します。



(a) プロジェクトツリー (b) 画像

図 5.13：サンプルとして Koala.jpg 画像ファイルをプロジェクトに追加

この画像ファイルを Source プロパティに指定する場合、現在編集している .xaml ファイルから見た相対パスを指定します。

コード 5.13：Image コントロールの使用例

MainWindow.xaml

```
1  <Window x:Class="Sample_Image.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainView" Height="200" Width="250">
5      <StackPanel>
6          <Image Source="Resources/Koala.jpg" />
7      </StackPanel>
8  </Window>
```

図 5.14：ソースで指定された画像が表示される

コードから画像を指定することもできます。

コード 5.14：Image コントロールに名前を付けておく

MainWindow.xaml

```
<Window x:Class="Sample_Image.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="200" Width="250">
    <StackPanel>
        <Image x:Name="image1" />
    </StackPanel>
</Window>
```

コード 5.15：jpeg 画像を読み込む

MainWindow.xaml.cs

```
namespace Sample_Image
{
    using System;
    using System.Windows;
    using System.Windows.Media.Imaging;

    /// <summary>
    /// MainWindow.xaml の相互作用ロジック
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            var jpeg = new BitmapImage(new Uri(@"Resources/Koala.jpg", UriKind.Relative));
            this.image1.Source = jpeg;
        }
    }
}
```

図 5.15：コードから指定された画像が表示される

5.4.2 Stretch プロパティ

Image コントロールの領域内でどのように画像を表示するかをこの Stretch プロパティで決定します。Stretch プロパティに指定できる値は System.Windows.Media.Stretch 列挙体で、その列挙子には次のようなものがあります。Image コントロールでは Uniform が既定値となっています。

表 5.1：Stretch プロパティに指定できる値

値	説　明
None	画像のサイズをそのままに表示する。

Uniform	縦横比を維持しながら対象の領域に収まるように画像を拡大/縮小する。こうすることで必ず画像全体が見えるようになる。
UniformToFill	縦横比を維持しながら対象の領域が画像で埋まるように画像を拡大/縮小する。はみ出た部分は表示されない。
Fill	縦横比に関係なく、対象の領域を埋めるように画像を拡大/縮小する。

次のサンプルコードでは、上表に示す Stretch 列挙体を ComboBox コントロールで選択することで、Image コントロールの Stretch プロパティを動的に変更できます。Stretch 列挙体は PresentationCore.dll で提供されていて、System.Windows.Media 名前空間に属しているため、エイリアス media を追加しています。

コード 5.16：Stretch プロパティを動的に変更するサンプル

MainWindow.xaml

```
<Window x:Class="Sample_Image.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:media="clr-namespace:System.Windows.Media;assembly=PresentationCore"
        Title="MainView" Height="200" Width="250">
    <DockPanel Background="Pink">
        <ComboBox x:Name="combobox1" DockPanel.Dock="Top" SelectedIndex="0">
            <ComboBox.ItemsSource>
                <x:Array Type="{x:Type media:Stretch}">
                    <media:Stretch>None</media:Stretch>
                    <media:Stretch>Uniform</media:Stretch>
                    <media:Stretch>UniformToFill</media:Stretch>
                    <media:Stretch>Fill</media:Stretch>
                </x:Array>
            </ComboBox.ItemsSource>
        </ComboBox>
        <Image Source="Resources/Koala.jpg" Stretch="{Binding SelectedValue,
ElementName=combobox1}" />
    </DockPanel>
</Window>
```

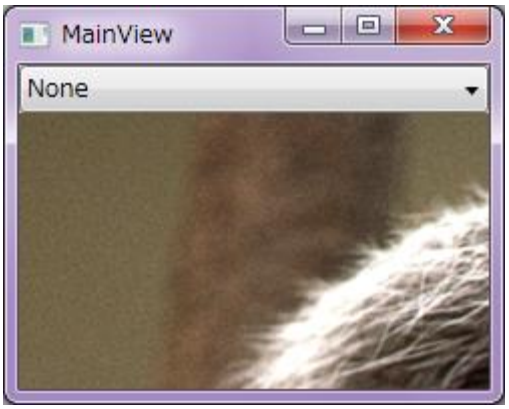


図 5.16：指定された Stretch の値にしたがって画像の表示方法が変わる

6 コレクションの表示に関するコントロール

この章ではデータコレクションを表示するためのコントロールについて紹介します。

6.1 ItemsControl コントロール

ItemsControl コントロールは、データやコントロールなどを複数持った配列やコレクションをソースとして、そのひとつひとつを指定された方法にしたがって並べるためのコントロールです。以降で紹介する ListBox コントロールや ComboBox コントロールなどはすべてこの ItemsControl コントロールから派生しています。

このコントロールのカスタマイズ方法を知っておくだけで UI デザインの幅が格段に広がります。WPF 開発者にとっては必須項目といってもいいくらいなので習得しておくことをお勧めします。

6.1.1 ItemsSource プロパティ

まず、単純に文字列である string 型を複数持つデータを ItemsSource プロパティに指定してみましょう。

コード 6.1 : ItemsControl コントロールを配置する

MainWindow.xaml

```
1  <Window x:Class="Sample_ItemsControl.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainView" Height="300" Width="250">
5      <StackPanel>
6          <ItemsControl x:Name="itemsControl1" />
7      </StackPanel>
8  </Window>
```

コード 6.2 : ItemsSource プロパティに string 型配列を指定する

MainWindow.xaml.cs

```
1  namespace Sample_ItemsControl
2  {
3      using System.Windows;
4
5      /// <summary>
6      /// MainWindow.xaml の相互作用ロジック
7      /// </summary>
8      public partial class MainWindow : Window
9      {
10         public MainWindow()
11         {
12             InitializeComponent();
13
14             this.itemsControl1.ItemsSource = new string[]
15             {
16                 "項目 1",
17                 "項目 2",
18                 "項目 3",
19                 "項目 4",
20                 "項目 5",
21                 "項目 6",
22                 "項目 7",
```

```
                "項目 8",
                "項目 9",
                "項目 10",
            };
        }
    }
}
```

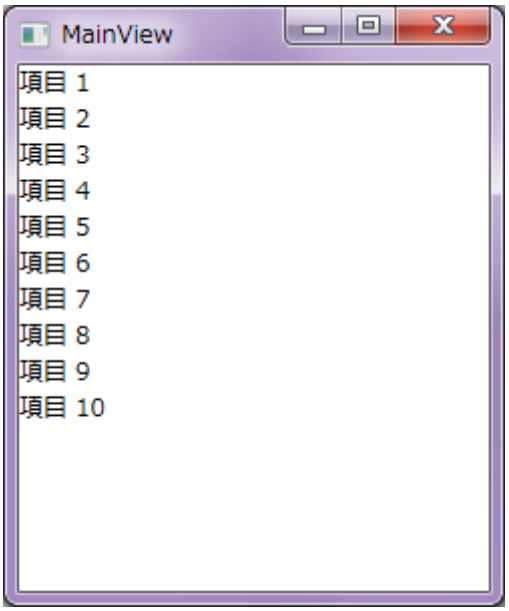


図 6.1：アイテムが縦並びに配置される

指定された文字列がひとつずつ配置されました。int 型や bool 型の場合は、それぞれの値が string 型に変換されて配置されます。以下のサンプルは bool 型のコレクションを与えたときの動作を示しています。

コード 6.3：ItemsSource プロパティに bool 型配列を指定する

MainWindow.xaml.cs

```
namespace Sample_ItemsControl
{
    using System.Windows;

    /// <summary>
    /// MainWindow.xaml の相互作用ロジック
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            this.itemsControl1.ItemsSource = new bool[]
            {
                true,
                true,
                false,
                false,
                true,
                false,
                false,
                true,
```

```
                false,
                true,
                true,
            };
        }
    }
}
```

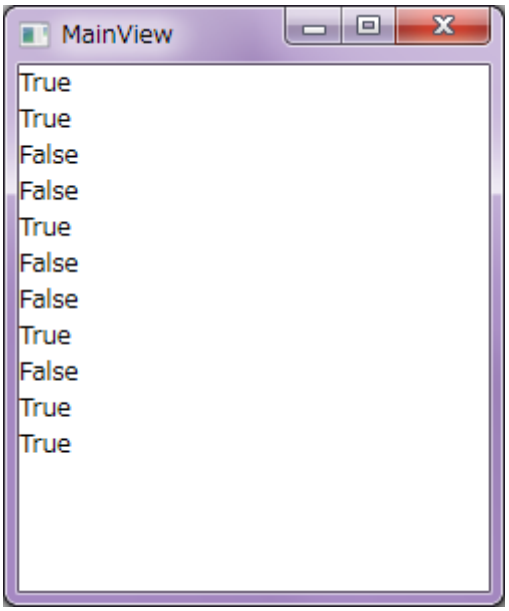


図 6.2：アイテムが縦並びに配置される

それでは、独自のクラスを ItemsSource プロパティに指定するとどうなるでしょうか。ここでは、以下のような Person クラスを考えます。

コード 6.4：Person クラスの定義

Person.cs

```
namespace Sample_ItemsControl
{
    /// <summary>
    /// 人物データを表します。
    /// </summary>
    public class Person
    {
        /// <summary>
        /// 名前を取得または設定します。
        /// </summary>
        public string Name { get; set; }

        /// <summary>
        /// 年齢を取得または設定します。
        /// </summary>
        public int Age { get; set; }

        /// <summary>
        /// 認証済みかどうかを取得または設定します。
        /// </summary>
        public bool IsAuthenticated { get; set; }
    }
}
```

この Person クラスを利用して、ItemsControl コントロールの ItemsSource プロパティにコレクションデータを設定します。先ほどは単純に配列を指定していましたが、ここでは System.Linq による Enumerable 列挙体を生成し、これを ItemsSource プロパティに代入しています。余談ですが、コレクションデータの生成や操作に関しては System.Linq が非常に有用ですので、C# プログラミングをする人は習得しておくべきでしょう。

コード 6.5：ItemsControl コントロールを配置する

MainWindow.xaml

```
<Window x:Class="Sample_ItemsControl.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="400" Width="250">
    <StackPanel>
        <ItemsControl x:Name="itemsControl1" />
    </StackPanel>
</Window>
```

コード 6.6：Person クラスの列挙子を ItemsSource プロパティに指定する

MainWindow.xaml.cs

```
namespace Sample_ItemsControl
{
    using System.Linq;
    using System.Windows;

    /// <summary>
    /// MainWindow.xaml の相互作用ロジック
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            this.itemsControl1.ItemsSource = Enumerable.Range(0, 10).Select(i => new Person()
            {
                Name = "サンプル " + i + "太郎",
                Age = i,
                IsAuthenticated = i % 3 != 0,
            });
        }
    }
}
```

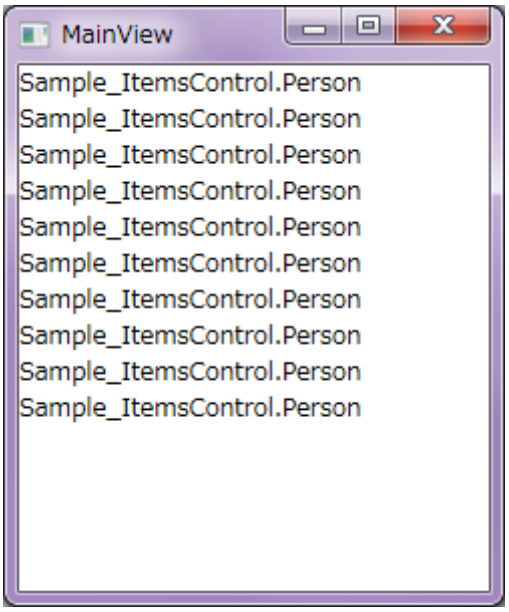


図 6.3：Person クラスが縦並びに配置される

結果の通り、独自のクラスのコレクションを ItemsSource プロパティに指定すると、そのクラス名がそのまま列挙されることになります。これは、ItemsControl コントロールの既定の動作が、与えられたアイテムを文字列として配置するようになっているためです。int 型や bool 型がデータとして見えるように配置されたのは、int 型や bool 型に自身を文字列に変換する ToString() メソッドが定義されているからです。

それでは、試しに Person クラスにも ToString() メソッドを定義してみましょう。ToString() メソッドは元々 object 型で定義されているため、これをオーバーライドする形で記述します。

コード 6.7：Person クラスに ToStriong() メソッドを実装する

Person.cs

```
1  namespace Sample_ItemsControl
2  {
3      /// <summary>
4      /// 人物データを表します。
5      /// </summary>
6      public class Person
7      {
8          /// <summary>
9          /// 名前を取得または設定します。
10         /// </summary>
11         public string Name { get; set; }
12
13         /// <summary>
14         /// 年齢を取得または設定します。
15         /// </summary>
16         public int Age { get; set; }
17
18         /// <summary>
19         /// 認証済みかどうかを取得または設定します。
20         /// </summary>
21         public bool IsAuthenticated { get; set; }
22
23         /// <summary>
24         /// インスタンスを文字列として表現します。
25         /// </summary>
26         /// <returns>インスタンスを表現した文字列を返します。</returns>
27         public override string ToString()
```

```
28          {
29              return string.Format("{0} ({1})\t{2}", this.Name, this.Age, this.IsAuthenticated);
30          }
31      }
32  }
```

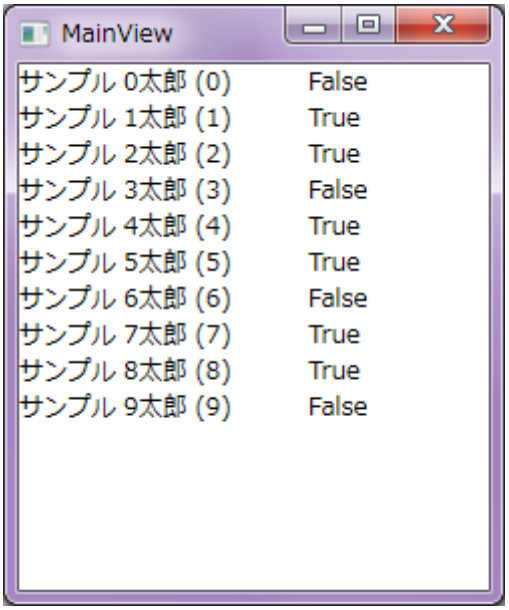


図 6.4：ToStriong() メソッドによって変換された文字列が表示されている

このように ToString() メソッドで表示する文字列を整形することができますが、以降で紹介する DisplayMemberPath プロパティや ItemTemplate プロパティなどを用いる方法のほうが一般的です。

6.1.2 DisplayMemberPtah プロパティ

このプロパティにプロパティ名を指定することで、並べたアイテムを表示するときにそのプロパティの値を表示するようにできます。

例えば次のような Person クラスを定義します。

コード 6.8：Person クラスの定義

Person.cs

```
1   namespace Sample_ItemsControl
2   {
3       /// <summary>
4       /// 人物データを表します。
5       /// </summary>
6       public class Person
7       {
8           /// <summary>
9           /// 名前を取得または設定します。
10          /// </summary>
11          public string Name { get; set; }
12      }
13  }
```

この Person クラスのコレクションを ItemsControl コントロールの ItemsSource プロパティに指定すると、Sample_ItemsControl.Person というクラス名が羅列されてしまいますが、同時に DisplayMemberPath プロパティを指定することでそのプロパティの値を並べることができます。

コード 6.9：DisplayMemberPath プロパティを指定する

MainWindow.xaml

```
<Window x:Class="Sample_ItemsControl.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="300" Width="250">
    <StackPanel>
        <ItemsControl x:Name="itemsControl1" DisplayMemberPath="Name"/>
    </StackPanel>
</Window>
```

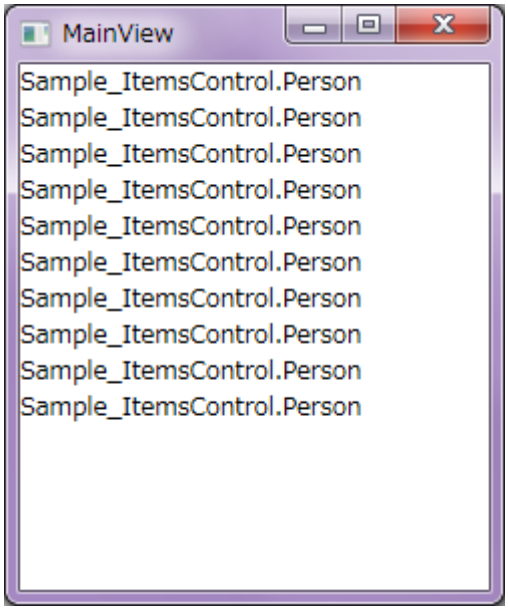
コード 6.10：ItemsSource プロパティに Person クラスの列挙子を指定する

MainWindow.xaml.cs

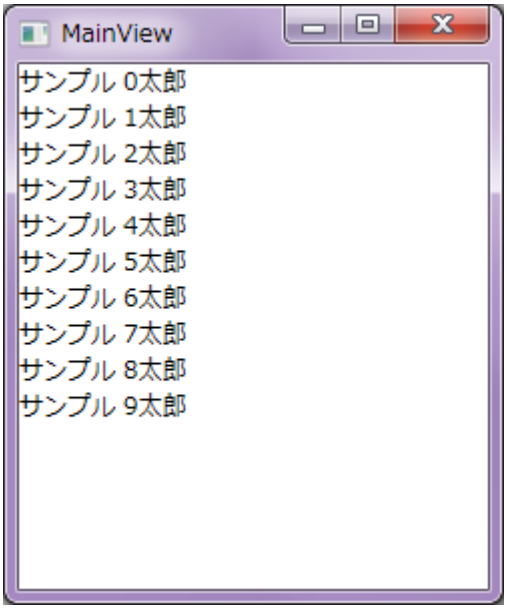
```
namespace Sample_ItemsControl
{
    using System.Linq;
    using System.Windows;

    /// <summary>
    /// MainWindow.xaml の相互作用ロジック
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            this.itemsControl1.ItemsSource = Enumerable.Range(0, 10).Select(i => new Person()
            {
                Name = "サンプル " + i + "太郎",
            });
        }
    }
}
```



(a) DisplayMemberPath プロパティを指定しない場合



(b) DisplayMemberPath プロパティを指定した場合

図 6.5：Person クラスの Name プロパティが表示される

DisplayMemberPath プロパティと ItemTemplate プロパティはどちらか一方しか設定できません。ComboBox コントロールのように並べるアイテムがテキストだけの場合は DisplayMemberPath プロパティを、それ以外で装飾したアイテムを表現したい場合は ItemTemplate プロパティを、というような使い分けになると思います。

6.1.3 ItemsPanel プロパティ

ItemsControl コントロールは与えられたコレクションデータのアイテムをひとつずつ並べるコントロールですが、その並べ方を指定することができます。その指定方法として、ItemsPanel、ItemTemplate、ItemContainerStyle プロパティを使用します。場合によっては Template プロパティを指定することもあります。

これらのプロパティによって下図のようにアイテムが配置されます。Template プロパティもしくは ItemsPanel プロパティでアイテムを並べるためのパネルを指定します。その上で、ItemContainerStyle プロパティで指定された Style にしたがって各アイテムの挙動が決定され、ItemTemplate プロパティで指定された DataTemplate によって各アイテムが表されます。

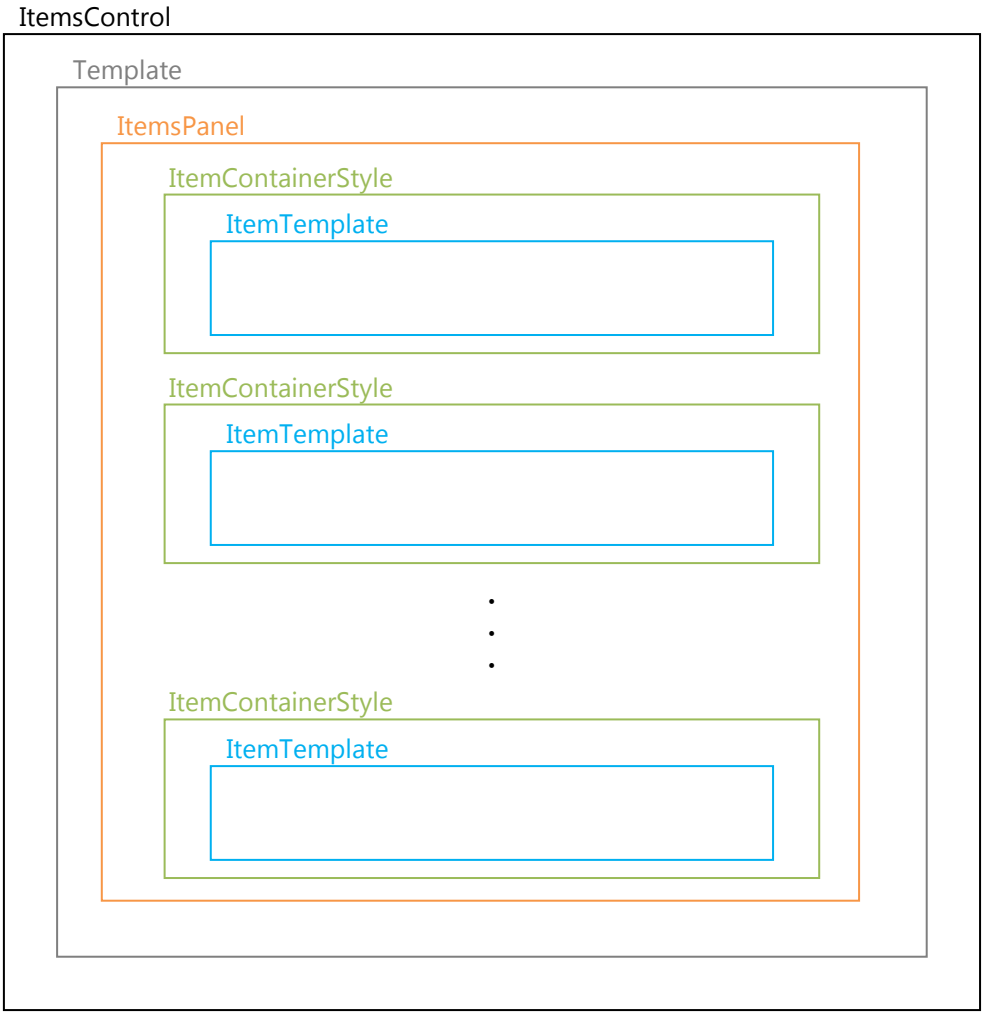


図 6.6：ItemsControl コントロールにおけるそれぞれのプロパティの役割

ここでは ItemsPanel プロパティについて見ていきます。

ItemsPanel プロパティでは、アイテムを並べるためのパネルを指定します。パネルということなので、StackPanel コントロールや WrapPanel コントロールなどが指定できます。これまでの動作を見る限り、デフォルトでは StackPanel コントロールが指定されているようです。

それでは、実際に ItemsPanel プロパティを変更してみましょう。

コード 6.11：ItemsControl コントロールを配置する

MainWindow.xaml

```
<Window x:Class="Sample_ItemsControl.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="300" Width="250">
    <StackPanel>
        <ItemsControl x:Name="itemsControl1">
            <ItemsControl.ItemsPanel>
                <ItemsPanelTemplate>
                    <StackPanel Orientation="Horizontal" />
                </ItemsPanelTemplate>
            </ItemsControl.ItemsPanel>
        </ItemsControl>
    </StackPanel>
</Window>
```

コード 6.12：Person クラスの定義

Person.cs

```
namespace Sample_ItemsControl
{
    /// <summary>
    /// 人物データを表します。
    /// </summary>
    public class Person
    {
        /// <summary>
        /// 名前を取得または設定します。
        /// </summary>
        public string Name { get; set; }

        /// <summary>
        /// 年齢を取得または設定します。
        /// </summary>
        public int Age { get; set; }

        /// <summary>
        /// 認証済みかどうかを取得または設定します。
        /// </summary>
        public bool IsAuthenticated { get; set; }

        /// <summary>
        /// インスタンスを文字列として表現します。
        /// </summary>
        /// <returns>インスタンスを表現した文字列を返します。</returns>
        public override string ToString()
        {
            return this.Name;
        }
    }
}
```

コード 6.13：ItemsSource プロパティに Person クラスの列挙子を指定する

MainWindow.xaml.cs

```
namespace Sample_ItemsControl
{
    using System.Linq;
```

```
 4      using System.Windows;
 5
 6      /// <summary>
 7      /// MainWindow.xaml の相互作用ロジック
 8      /// </summary>
 9      public partial class MainWindow : Window
10      {
11          public MainWindow()
12          {
13              InitializeComponent();
14
15              this.itemsControl1.ItemsSource = Enumerable.Range(0, 10).Select(i => new Person()
16              {
17                  Name = "サンプル " + i + "太郎",
18                  Age = i,
19                  IsAuthenticated = i % 3 != 0,
20              });
21          }
22      }
23  }
```

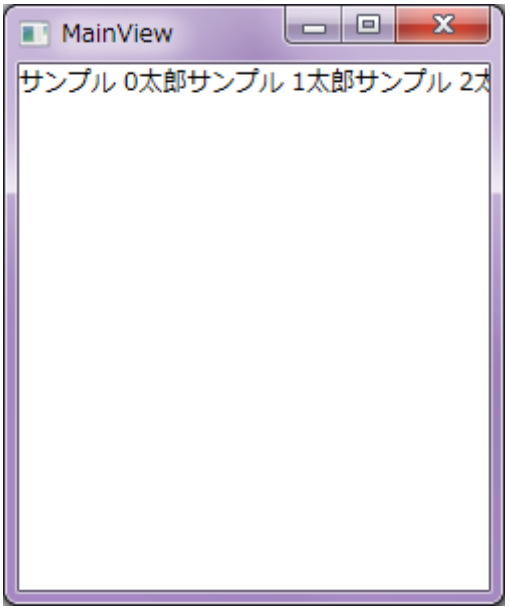


図 6.7：アイテムが横並びに配置される

ItemsPanel プロパティに横並びの StackPanel コントロールを指定したため、Person クラスが横並びになっていることがわかります。しかし、StackPanel コントロールであるため、はみ出たアイテムはそのまま表示されなくなってしまっています。

StackPanel コントロールではなく、WrapPanel コントロールにした場合は次のようになります。

コード 6.14：ItemsControl コントロールを配置する

MainWindow.xaml

```
 8                  <ItemsPanelTemplate>
 9                      <WrapPanel Orientation="Horizontal" />
10                  </ItemsPanelTemplate>
```

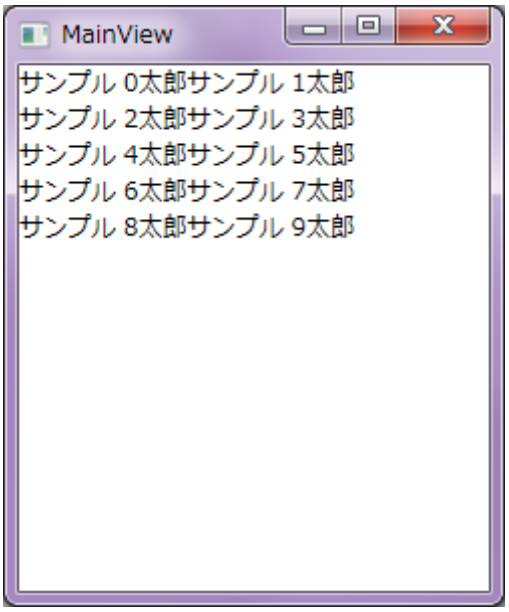


図 6.8：アイテムが横並びではみ出るアイテムは次の行に配置される

WrapPanel コントロールを指定したため、はみ出たアイテムも自動的に表示できる位置に再配置されています。

Grid コントロールや Canvas コントロールなどもパネルコントロールなので、ItemsPanel プロパティに指定できますが、子要素を並べるためのパネルコントロールではないため、そのまま使ってしまうとアイテムが重なってしまうことになります。

(a) Grid コントロールを指定した場合

(b) Canvas コントロールを指定した場合

図 6.9：Grid コントロールや Canvas コントロールはそのままでは使えない

Grid コントロール上でアイテムを配置するには Margin プロパティや Grid.Row、Grid.Column 添付プロパティを指定する必要があります。また、Canvas コントロール上では Canvas.Left、Canvas.Top 添付プロパティなどを指定する必要があります。これらを指定するには ItemTemplate プロパティなどで工夫すれば指定できます。

6.1.4 ItemTemplate プロパティ

これまでは Person クラスを見える形で表示するために ToString() メソッドをオーバーライドしていましたが、これでは文字列としての表現しか得られません。ItemTemplate プロパティでは様々なコンテンツによってアイテムを表現するための DataTemplate が指定できます。

コード 6.15：ItemsControl コントロールを配置する

MainWindow.xaml

```
<Window x:Class="Sample_ItemsControl.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="300" Width="250">
    <StackPanel>
        <ItemsControl x:Name="itemsControl1">
            <ItemsControl.ItemTemplate>
                <DataTemplate>
                    <StackPanel>
                        <TextBlock Text="{Binding Name, StringFormat='{}氏名 : {0}'}" />
                        <TextBlock Text="{Binding Age, StringFormat='{}年齢 : {0}'}" />
                        <StackPanel.Style>
                            <Style TargetType="StackPanel">
                                <Setter Property="Background" Value="Cornsilk" />
                                <Style.Triggers>
                                    <DataTrigger Binding="{Binding IsAuthenticated}"
Value="False">
                                        <Setter Property="Background" Value="LightPink" />
                                    </DataTrigger>
                                </Style.Triggers>
                            </Style>
                        </StackPanel.Style>
                    </StackPanel>
                </DataTemplate>
            </ItemsControl.ItemTemplate>
        </ItemsControl>
    </StackPanel>
</Window>
```

コード 6.16：Person クラスの定義

Person.cs

```
namespace Sample_ItemsControl
{
    /// <summary>
    /// 人物データを表します。
    /// </summary>
    public class Person
    {
        /// <summary>
        /// 名前を取得または設定します。
        /// </summary>
        public string Name { get; set; }

        /// <summary>
        /// 年齢を取得または設定します。
        /// </summary>
        public int Age { get; set; }

        /// <summary>
        /// 認証済みかどうかを取得または設定します。
```

```
20          /// </summary>
21          public bool IsAuthenticated { get; set; }
22
23          /// <summary>
24          /// インスタンスを文字列として表現します。
25          /// </summary>
26          /// <returns>インスタンスを表現した文字列を返します。</returns>
27          public override string ToString()
28          {
29              return this.Name;
30          }
31      }
32  }
```

コード 6.17：ItemsSource プロパティに Person クラスの列挙子を指定する

MainWindow.xaml.cs

```
1   namespace Sample_ItemsControl
2   {
3       using System.Linq;
4       using System.Windows;
5
6       /// <summary>
7       /// MainWindow.xaml の相互作用ロジック
8       /// </summary>
9       public partial class MainWindow : Window
10      {
11          public MainWindow()
12          {
13              InitializeComponent();
14
15              this.itemsControl1.ItemsSource = Enumerable.Range(0, 10).Select(i => new Person()
16              {
17                  Name = "サンプル " + i + "太郎",
18                  Age = i,
19                  IsAuthenticated = i % 3 != 0,
20              });
21          }
22      }
23  }
```

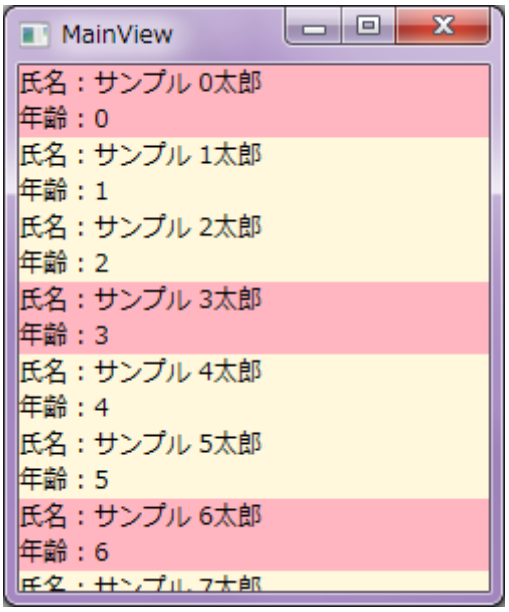


図 6.10：指定された ItemTemplate プロパティにしたがって配置されている

ItemsPanel プロパティは特に指定していないため、アイテムは縦並びになりますが、アイテムの表現は ItemsPanel プロパティに指定された DataTemplate にしたがっています。

ItemTemplate プロパティで DataTemplate を指定するとき、その DataContext はアイテムとなります。したがって、データバインディングをするときはアイテムを基準として考える必要があります。
例えば、アイテムは Name プロパティを持っているため、TextBlock コントロールに Name プロパティをバインディングとして指定することができます。
また、ここでは IsAuthenticated プロパティを直接表示せず、StackPanel コントロールの背景色を変更するためのトリガとして使用しています。

このように、文字列だけでなく様々な形で各アイテムを表現できます。DataTemplate として指定できるため、例えば Button コントロールや Image コントロールも配置できるため、一見 ItemsControl を使っているとは思えないような多彩な表現ができます。

6.1.5 ItemContainerStyle プロパティ

ItemContainerStyle プロパティでは、各アイテムに対する Style を指定します。ItemTemplate プロパティで指定する DataTemplate でほぼ同じことが実現できることもありますが、例えば表示のみの役割を ItemTemplate プロパティで、アイテム間のレイアウト調整や Trigger による動的な挙動は ItemContainerStyle プロパティで指定するなど、自分なりのやり方を決めると使いやすくなります。

ただし、ItemsControl コントロールでは、各アイテムを並べるとき、それぞれを ContentPresenter コントロールに入れ、それをひとつのコンテナとして扱います。つまり、ItemContainerStyle プロパティに指定する Style は ContentPresenter コントロールを対象とした Style にする必要があります。しかし、ContentPresenter コントロールは、System.Windows.FrameworkElement クラスから派生しているコントロールであるため、System.Windows.Controls.Control クラスが提供しているような、例えば Background プロパティや Template プロパティなどが使えません。このことに関しては図 2.1 を参照するとわかりやすいと思います。
このことから、ここでは特に ItemContainerStyle プロパティを指定したサンプルは掲載しません。ItemContainerStyle プロパティのサンプルは「6.2 ListBox コントロール」などのサンプルを参考にしてください。

6.2 ListBox コントロール

ListBox コントロールは、複数のアイテムを並べ、ひとつまたは複数を選択できるコントロールです。System.Controls.Primitives.Selector コントロールから派生したコントロールで、下表のようなプロパティが追加されています（System.Controls.Primitives.Selector コントロールは ItemsControl コントロールから派生しています）。

表 6.1：ListBox コントロールで追加されているプロパティ

プロパティ名	型	説　明
IsSynchronizedWithCurrentItem（Selector から継承）	bool?	Selector で SelectedItem を Items プロパティの現在の項目と同期するかどうかを取得または設定します。
SelectedIndex（Selector から継承）	int	現在の選択範囲のうち、最初のアイテムのインデックスを取得または設定します。選択範囲が空の場合は -1 を返します。
SelectedItem（Selector から継承）	object	現在選択されている最初の項目を取得または設定します。選択範囲が空の場合は null を返します。
SelectedItems	IList	現在選択されている項目を取得します。
SelectedValue（Selector から継承）	object	SelectedValuePath を使用して取得される SelectedItem の値を取得または設定します。
SelectedValuePath（Selector から継承）	string	SelectedItem から SelectedValue を取得するために使用するパスを取得または設定します。
SelectionMode	SelectionMode	ListBox の選択動作を取得または設定します。

ここでは次のような Person クラスを使って説明します。

コード 6.18：Person クラスの定義

Person.cs

```
1  namespace Sample_ListBox
2  {
3      /// <summary>
4      /// 人物データを表します。
5      /// </summary>
6      public class Person
7      {
8          /// <summary>
9          /// 名前を取得または設定します。
10         /// </summary>
11         public string Name { get; set; }
12
13         /// <summary>
14         /// 年齢を取得または設定します。
15         /// </summary>
16         public int Age { get; set; }
17
18         /// <summary>
19         /// 認証済みかどうかを取得または設定します。
20         /// </summary>
21         public bool IsAuthenticated { get; set; }
22     }
23 }
```

6.2.1 IsSynchronizedWithCurrentItem プロパティ

このプロパティは、同じデータを ItemsSource プロパティに指定された複数の ListBox コントロールで、SelectedItem プロパティを常に同じオブジェクトにしたいときに使用します。

コード 6.19 : ItemsControl コントロールを配置する

MainWindow.xaml

```
<Window x:Class="Sample_ListBox.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="300" Width="400">
    <StackPanel Orientation="Horizontal" Background="Cornsilk" Margin="10">
        <TextBlock Text="ListBox" Margin="0,0,4,0" />
        <ListBox x:Name="listbox1" DisplayMemberPath="Name" SelectionMode="Extended"
IsSynchronizedWithCurrentItem="True" />
        <TextBlock Text="ListBox2" Margin="10,0,4,0" />
        <ListBox x:Name="listbox2" DisplayMemberPath="Name"
IsSynchronizedWithCurrentItem="True" />
    </StackPanel>
</Window>
```

コード 6.20 : ItemsSource プロパティに Person クラスの列挙子を指定する

MainWindow.xaml.cs

```
namespace Sample_ListBox
{
    using System.Linq;
    using System.Windows;

    /// <summary>
    /// MainWindow.xaml の相互作用ロジック
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            this.listbox1.ItemsSource = Enumerable.Range(0, 10).Select(i => new Person()
            {
                Name = "サンプル " + i + "太郎",
                Age = i,
                IsAuthenticated = i % 3 != 0,
            });
            this.listbox2.ItemsSource = this.listbox1.ItemsSource;
        }
    }
}
```

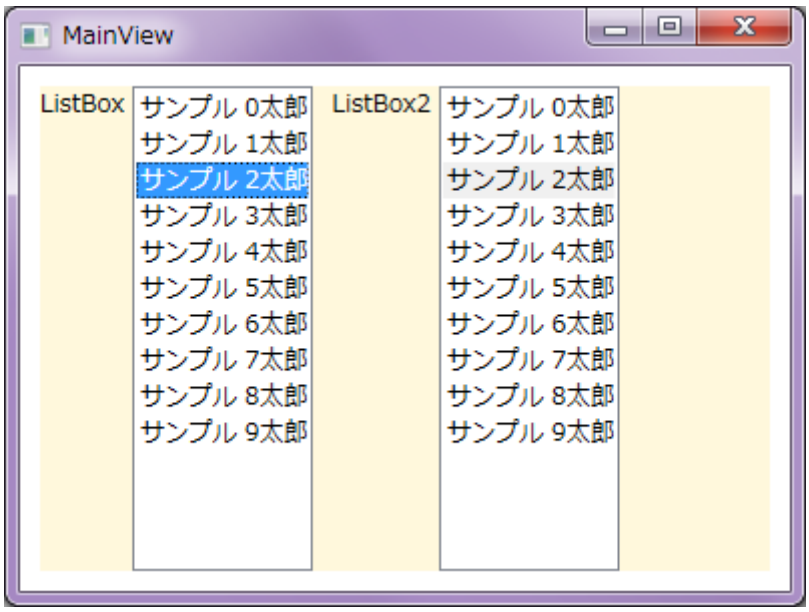


図 6.11：指定された ItemTemplate プロパティにしたがって配置されている

どちらの ListBox コントロールでアイテムを選択しても、必ず両方同じアイテムが選択されるようになります。

6.2.2 SelectionMode プロパティ

アイテムをひとつだけ選択できるようにするのか、または複数選択できるようにするのかを SelectionMode プロパティによって指定します。SelectionMode プロパティに指定できる値は System.Windows.Controls.SelectionMode 列挙体で、その列挙子には次のようなものがあります。ListBox コントロールでは Single が既定値となっています。

表 6.2：SelectionMode プロパティに指定できる値

値	説　明
Extended	ユーザーは、Shift キーを押しながら連続した複数の項目を選択できます。Ctrl キーを押しながら連続していない複数の項目を選択することもできます。
Multiple	ユーザーは、変換キーを押さなくても複数の項目を選択できます。
Single	ユーザーは、一度に 1 つの項目しか選択できません。

6.2.3 SelectedIndex プロパティ

ItemsSource プロパティまたは Items プロパティに指定したコレクションの先頭を 0 として、現在選択しているアイテムのインデックスを取得または設定します。ただし未選択状態は -1 として表現されます。また、複数のアイテムを選択している場合は、最初に選択したアイテムのインデックスとなります。

SelectedIndex プロパティの既定値は -1 のため、指定しないと起動直後は未選択状態となります。これを回避するときは、SelectedIndex プロパティに例えば 0 を指定しておくと、始めからアイテムが選択された状態となります。

SelectedItem プロパティと連動しているため、どちらかを変更するともう片方もそれに該当する値に変更されます。

6.2.4 SelectedItem プロパティ

現在選択されているアイテムを取得または設定します。複数のアイテムを選択している場合は、最初に選択したアイテムになります。
SelectedIndex プロパティと連動しているため、どちらかを変更するともう片方もそれに該当する値に変更されます。

6.2.5 SelectedItems プロパティ

SelectionMode プロパティに Single 以外を設定し、複数のアイテムを選択したとき、選択しているアイテムすべてを IList として取得できます。

このプロパティは読み取り専用のため注意が必要です。

6.2.6 SelectedValue プロパティ

SelectedItem プロパティは選択されたアイテムそのものを取得または設定しますが、SelectedValue プロパティは、SelectedValuePath プロパティによって指定されたプロパティパスの値を取得または設定することができます。

6.2.7 SelectedValuePath プロパティ

SelectedValue プロパティによって取得または設定したいプロパティ名を指定します。

6.2.8 ItemContainerStyle プロパティの使用例

ItemContainerStyle プロパティによって、アイテムコンテナの Style を指定することができます。「0

ItemTemplate プロパティ」のように、ItemTemplate プロパティだけでもアイテムそれぞれの外観を変更することはできますが、ItemContainerStyle プロパティによってコンテナそのものの設定がおこなえるため、例えば選択状態の外観を簡単に変更できます。

ItemTemplate プロパティや ItemContainerStyle プロパティなどの関係については図 6.6 を参照してください。

コード 6.21：ItemContainerStyle プロパティを指定した ListBox コントロール

MainWindow.xaml

```
<Window x:Class="Sample_ListBox.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="300" Width="200">
    <ListBox x:Name="listbox1" Grid.Column="1">
        <ListBox.ItemContainerStyle>
            <Style TargetType="ListBoxItem">
                <Setter Property="Background" Value="White" />
                <Setter Property="Template">
                    <Setter.Value>
                        <ControlTemplate TargetType="ContentControl">
                            <Border Background="{TemplateBinding Background}">
                                <ContentPresenter />
                            </Border>
                        </ControlTemplate>
                    </Setter.Value>
                </Setter>
                <Style.Triggers>
                    <Trigger Property="IsMouseOver" Value="True">
                        <Setter Property="Background" Value="LightGray" />
                    </Trigger>
                    <Trigger Property="IsSelected" Value="True">
                        <Setter Property="Background" Value="Plum" />
                    </Trigger>
                </Style.Triggers>
            </Style>
        </ListBox.ItemContainerStyle>
        <ListBox.ItemTemplate>
            <DataTemplate>
                <StackPanel>
                    <TextBlock Text="{Binding Name, StringFormat='{}氏名 : {0}'}" />
                    <TextBlock Text="{Binding Age, StringFormat='{}年齢 : {0}'}" />
                </StackPanel>
            </DataTemplate>
        </ListBox.ItemTemplate>
    </ListBox>
</Window>
```

コード 6.22：ItemsSource プロパティに Person クラスの列挙子を指定する

MainWindow.xaml.cs

```
namespace Sample_ListBox
{
    using System.Linq;
    using System.Windows;

    /// <summary>
    /// MainWindow.xaml の相互作用ロジック
    /// </summary>
    public partial class MainWindow : Window
    {
```

```
11          public MainWindow()
12          {
13              InitializeComponent();
14
15              this.listbox1.ItemsSource = Enumerable.Range(0, 10).Select(i => new Person()
16              {
17                  Name = "サンプル " + i + "太郎",
18                  Age = i,
19                  IsAuthenticated = i % 3 != 0,
20              });
21          }
22      }
23  }
```

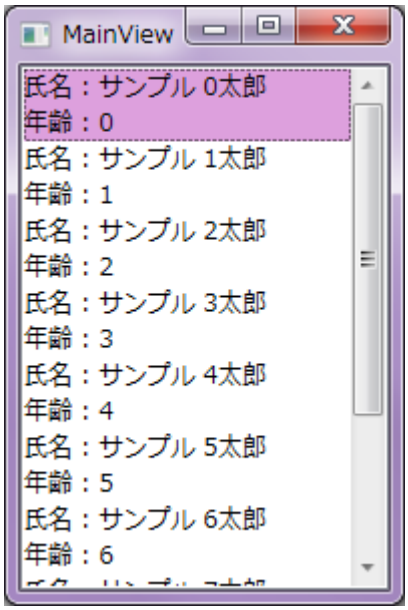


図 6.12：指定された ItemContainerStyle プロパティによって色が変更されている

6.3 ComboBox コントロール

ComboBox コントロールはドロップダウンリスト形式でアイテムを選択するコントロールです。ListBox コントロールと同様に System.Controls.Primitives.Selector コントロールから派生したコントロールで、下表のようなプロパティが追加されています（System.Controls.Primitives.Selector コントロールは ItemsControl コントロールから派生しています）。したがって、Selector コントロールから継承する SelectedItem プロパティなどは ListBox コントロールと同様に使用できます。詳細は「6.2 ListBox コントロール」を参照してください。
その他の ComboBox コントロール特有のプロパティを下表にまとめます。

表 6.3：ListBox コントロールで追加されているプロパティ

プロパティ名	型	説　明
IsDropDownOpen	bool	コンボ ボックスのドロップダウンが現在開かれているかどうかを示す値を、取得または設定します。
IsEditable	bool	ComboBox のテキスト ボックス内のテキストの編集を有効または無効にする値を取得または設定します。
IsReadOnly	bool	選択専用モードを有効にする値を取得または設定します。選択専用モードでは、コンボ ボックスの内容は選択可能ですが、編集することはできません。
IsSelectionBoxHighlighted	bool	SelectionBoxItem が強調表示されるかどうかを取得します。
MaxDropDownHeight	double	コンボ ボックス ドロップダウンの最大の高さを取得または設定します。
SelectionBoxItem	object	選択ボックスに表示される項目を取得します。
SelectionBoxItemStringFormat	string	選択ボックス内の選択した項目が文字列として表示される場合に、その書式を指定する複合文字列を取得します。
SelectionBoxItemTemplate	DataTemplate	選択ボックスの内容の項目テンプレートを取得します。
ShouldPreserveUserEnteredPrefix	bool	ComboBox がユーザー入力を保持するか、または一致する項目で入力を置き換えるかどうかを示す値を取得または設定します。
StaysOpenOnEdit	bool	ユーザーがテキスト ボックスに情報を入力しているときに、ドロップ ダウン コントロールが開いたままになるかどうかを取得または設定します。
Text	string	現在選択されている項目のテキストを取得または設定します。

IsDropDownOpen プロパティによってドロップダウンリストが開いているかどうか、もしくは開くかどうかを確認、設定できます。

IsEditable プロパティは false が既定値となっているため、通常の ComboBox コントロールは用意されているアイテムから選択するだけで、それ以外のアイテムを設定することはできません。しかし、場合によってはユーザーが任意に入力できるようにしたいこともあります。このとき、IsEditable プロパティを true にすると、ComboBox コントロールを選択すると TextBox コントロールと同様にテキスト入力を受け付けるようになります。

IsReadOnly プロパティは、IsEditable プロパティを true にすることによって表示される TextBox コントロールに対する IsReadOnly プロパティを設定するためのものです。したがって、IsEditable プロパティと IsReadOnly プロパティを同時に true にすると、用意されているアイテムをドロップダウンリストから選択することでしかアイテム選択できませんが、選択されたアイテムのテキストをコピーすることができるようになります。

StaysOpenOnEdit プロパティは、IsEditable プロパティが true のとき、ドロップダウンリストを開いた状態でテキストを編集できるようにするためのプロパティです。ComboBox コントロール右端のボタンを押してドロップダウンリストを表示させた後、TextBox コントロールの領域をクリックすると、通常はドロップダウンリストが閉じられてしまいますが、StaysOpenOnEdit プロパティを true にすると、同様の操作をしても閉じないようになります。

以下のサンプルでは、ComboBox コントロールに様々なプロパティを設定したときの動作の違いを見るために、同じコレクションデータを持った複数の ComboBox コントロールを配置しています。
コレクションデータは XmlDataProvider クラスによってリソースに定義しています。今後変更がないような固定値を定義する場合はこのような方法が便利です。

コード 6.23：ComboBox コントロールのプロパティ設定例

MainWindow.xaml

```
<Window x:Class="Sample_ComboBox.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="200" Width="250">
    <Window.Resources>
        <XmlDataProvider x:Key="People" XPath="/People/*">
            <x:XData>
                <People xmlns="">
                    <Person Name="Hanako Tanaka" Age="23" IsAuthenticated="true" />
                    <Person Name="Tarou Tanaka" Age="21" IsAuthenticated="true" />
                    <Person Name="Mitsuba Tanaka" Age="16" IsAuthenticated="false" />
                    <Person Name="Yotsuba Tanaka" Age="16" IsAuthenticated="false" />
                    <Person Name="Jirou Tanaka" Age="12" IsAuthenticated="true" />
                </People>
            </x:XData>
        </XmlDataProvider>
    </Window.Resources>

    <StackPanel>
        <ComboBox ItemsSource="{Binding Source={StaticResource People}}"
                  DisplayMemberPath="@Name"
                  SelectedIndex="0"
                  />

        <TextBlock Text="Editable ComboBox :" />
        <ComboBox ItemsSource="{Binding Source={StaticResource People}}"
                  DisplayMemberPath="@Name"
                  IsEditable="True"
                  ShouldPreserveUserEnteredPrefix="True"
                  />

        <TextBlock Text="Editable but ReadOnly ComboBox :" />
        <ComboBox ItemsSource="{Binding Source={StaticResource People}}"
                  DisplayMemberPath="@Name"
                  IsEditable="True"
                  IsReadOnly="True"
                  />

        <TextBlock Text="StaysOpenOnEdit ComboBox :" />
        <ComboBox ItemsSource="{Binding Source={StaticResource People}}"
                  DisplayMemberPath="@Name"
                  IsEditable="True"
                  ShouldPreserveUserEnteredPrefix="True"
                  StaysOpenOnEdit="True"
                  />
```

```
46      </StackPanel>
47  </Window>
```

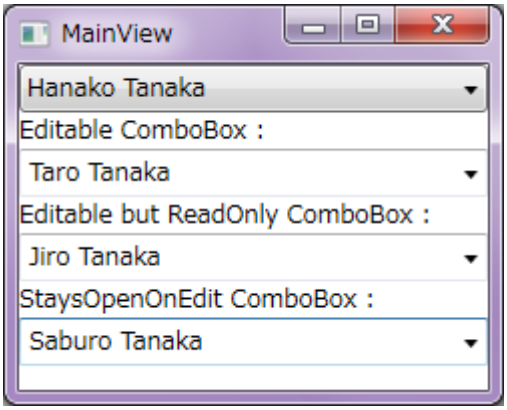


図 6.13：ComboBox コントロールの使用例

6.4 ListView コントロール

データをリスト形式で表示するためのコントロールです。ListView コントロールは ListBox コントロールから派生し、データを表示するための View プロパティを持っています。つまり、データの表示形式を View プロパティで設定しますが、後は ListBox コントロールと同様の動作となります。

コード 6.24：Person クラスの定義

Person.cs

```
1  namespace Sample_ListView
2  {
3      /// <summary>
4      /// 人物データを表します。
5      /// </summary>
6      public class Person
7      {
8          /// <summary>
9          /// 名前を取得または設定します。
10         /// </summary>
11         public string Name { get; set; }
12
13         /// <summary>
14         /// 年齢を取得または設定します。
15         /// </summary>
16         public int Age { get; set; }
17
18         /// <summary>
19         /// 認証済みかどうかを取得または設定します。
20         /// </summary>
21         public bool IsAuthenticated { get; set; }
22     }
23 }
```

コード 6.25：ComboBox コントロールのプロパティ設定例

MainWindow.xaml

```
1  <Window x:Class="Sample_ListView.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainView" Height="200" Width="250">
5      <Grid>
6          <ListView x:Name="listview1">
7              <ListView.View>
8                  <GridView ColumnHeaderToolTip="People">
9                      <GridViewColumn Header="Full Name" DisplayMemberBinding="{Binding Name}" />
10                     <GridViewColumn Header="Age" DisplayMemberBinding="{Binding Age}" />
11                 </GridView>
12             </ListView.View>
13         </ListView>
14     </Grid>
15 </Window>
```

コード 6.26：ItemsSource プロパティに Person クラスの列挙子を指定する

MainWindow.xaml.cs

```
1  namespace Sample_ListView
2  {
3      using System.Linq;
4      using System.Windows;
5
6      /// <summary>
```

```
    /// MainWindow.xaml の相互作用ロジック
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            this.listview1.ItemsSource = Enumerable.Range(0, 10).Select(i => new Person()
            {
                Name = "サンプル " + i + "太郎",
                Age = i,
                IsAuthenticated = i % 3 != 0,
            });
        }
    }
}
```

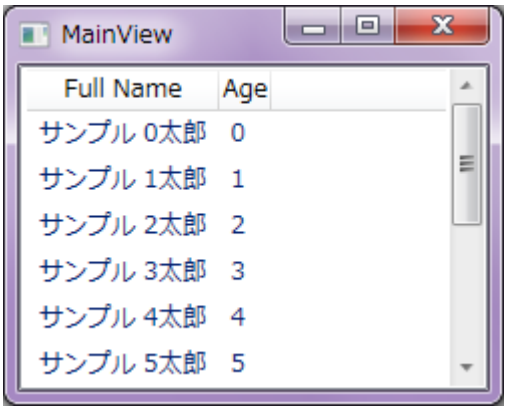


図 6.14：ComboBox コントロールの使用例

View プロパティには GridView クラスを指定し、GridViewColumn クラスで各列のデータ表示方法を指定します。上記のサンプルでは GridViewColumn クラスの DisplayMemberBinding プロパティに表示したいプロパティ名を指定することで、シンプルにテキストを表示させています。また、Header プロパティには列ヘッダに表示する文字列を指定しています。

GridViewColumn クラスの CellTemplate プロパティを使用することで、様々なデータ表示方法を指定することができます。CellTemplate プロパティは DataTemplate 型のため、例えば次のように別のコントロールを配置させることができます。

コード 6.27：ComboBox コントロールのプロパティ設定例

MainWindow.xaml

```
<Window x:Class="Sample_ListView.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="200" Width="320">
    <Grid>
        <ListView x:Name="listview1">
            <ListView.View>
                <GridView ColumnHeaderToolTip="People">
                    <GridViewColumn Header="Full Name" DisplayMemberBinding="{Binding Name}" />
                    <GridViewColumn Header="Age" DisplayMemberBinding="{Binding Age,
StringFormat='{}{0}才'}" />
                    <GridViewColumn Header="Custom Column">
                        <GridViewColumn.CellTemplate>
                            <DataTemplate>
```

```
14                                  <CheckBox IsChecked="{Binding IsAuthenticated}">
15                                      <TextBlock>
16                                          <Run Text="{Binding Name, Mode=OneWay}" />
17                                          <LineBreak />
18                                          <Run Text="{Binding Age, Mode=OneWay, StringFormat='{}{0}
    才'}" />
19                                      </TextBlock>
20                                  </CheckBox>
21                              </DataTemplate>
22                          </GridViewColumn.CellTemplate>
23                      </GridViewColumn>
24                  </GridView>
25              </ListView.View>
26          </ListView>
27      </Grid>
28  </Window>
```

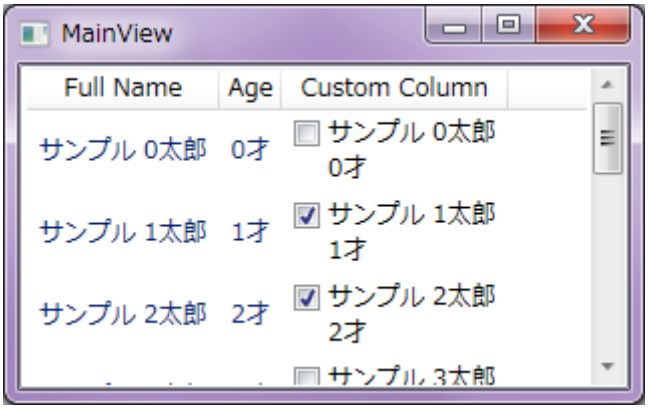


図 6.15：ComboBox コントロールの使用例

また、GridViewColumn クラスには HeaderTemplate プロパティや HeaderContainerStyle プロパティなど、ヘッダに関するテンプレートも指定できるので、非常に幅広いカスタマイズができます。

6.5 DataGrid コントロール

DataGrid コントロールは、ListView コントロールと同様にアイテムをリスト形式で表示するコントロールですが、ListView コントロールが System.Windows.Controls.Primitives.Selector クラスから派生しているのに対し、DataGrid コントロールは System.Windows.Controls.Primitives.MultiSelector クラスから派生しています。

コード 6.28：Person クラスの定義

Person.cs

```
namespace Sample_DataGrid
{
    /// <summary>
    /// 人物データを表します。
    /// </summary>
    public class Person
    {
        /// <summary>
        /// 名前を取得または設定します。
        /// </summary>
        public string Name { get; set; }

        /// <summary>
        /// 年齢を取得または設定します。
        /// </summary>
        public int Age { get; set; }

        /// <summary>
        /// 性別を取得または設定します。
        /// </summary>
        public Gender Gender { get; set; }

        /// <summary>
        /// 認証済みかどうかを取得または設定します。
        /// </summary>
        public bool IsAuthenticated { get; set; }
    }

    /// <summary>
    /// 性別を表します。
    /// </summary>
    public enum Gender
    {
        /// <summary>
        /// 男性を表します。
        /// </summary>
        Male,

        /// <summary>
        /// 女性を表します。
        /// </summary>
        Female,
    }
}
```

コード 6.29：DataGrid コントロールの配置

MainWindow.xaml

```
<Window x:Class="Sample_DataGrid.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
        Title="MainView" Height="200" Width="320">
    <DataGrid ItemsSource="{Binding People}" />
</Window>
```

コード 6.30：ItemsSource プロパティに Person クラスの列挙子を指定する

MainWindow.xaml.cs

```
namespace Sample_DataGrid
{
    using System.Linq;
    using System.Windows;

    /// <summary>
    /// MainWindow.xaml の相互作用ロジック
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            this.DataContext = this;

            People = Enumerable.Range(0, 10).Select(i => new Person()
            {
                Name = "サンプル " + i + "太郎",
                Age = i,
                IsAuthenticated = i % 3 != 0,
            }).ToArray();
        }

        public Person[] People { get; set; }
    }
}
```

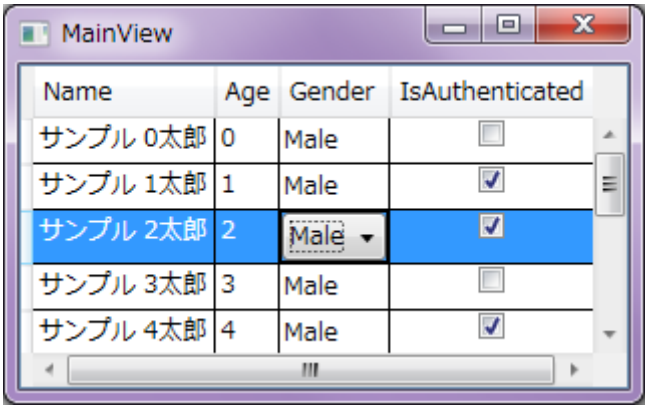


図 6.16：DataGrid コントロールの使用例

DataGrid コントロールは、デフォルト設定で、表示するプロパティの型によって表示形式が自動的に変更されます。bool 型の場合はチェックボックスが表示され、それ以外はテキストとして表示されます。また、ダブルクリックすることで各アイテムの要素を編集して変更することができますが、string 型や int 型の場合は TextBox コントロール、enum 型の場合は ComboBox コントロールとなります。列ヘッダはプロパティ名がそのまま表示されます。

このように、クラスのプロパティの型によって自動的に列が生成されるのは便利のように思えますが、実際のアプリケーションに実装しようとした場合、プロパティ名がそのままヘッダになっていい状況はほとんどないし、bool 型だからといって必ずしも CheckBox コントロールを配置する必要があるとは限りません。

AutoGenerateColumns プロパティを false にすることによって、このような列が自動的に生成される動作を禁止することができます。

コード 6.31：AutoGenerateColumns プロパティの指定

MainWindow.xaml

```
1  <Window x:Class="Sample_DataGrid.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainView" Height="200" Width="320">
5      <DataGrid ItemsSource="{Binding People}" AutoGenerateColumns="False" />
6  </Window>
```

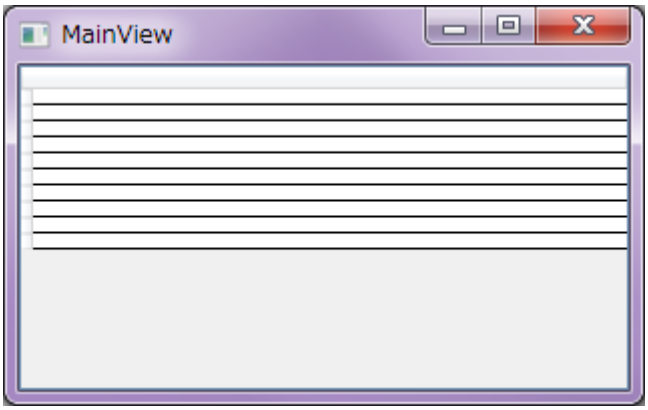


図 6.17：列の自動生成を禁止したのでどのアイテムも列を持っていない

AutoGenerateColumns プロパティに false を指定すると、列の自動生成がおこなわれないため、各アイテムの行が表示されるだけで、データが何も表示されない状態になってしまいます。列を個別に指定するには、Columns プロパティに下表に示すクラスを必要なだけ羅列する必要があります。

表 6.4：DataGrid コントロールの Columns プロパティに指定できるクラス

プロパティ名	説　明
DataGridCheckBoxColumn	アイテムの要素を CheckBox コントロールとして表示する。IsChecked プロパティでチェックの ON/OFF を切り替える。
DataGridComboBoxColumn	アイテムの要素を ComboBox コントロールとして表示する。ItemsSource プロパティに ComboBox コントロールで選択できるアイテムを指定する。
DataGridHyperlinkColumn	アイテムの要素を Hyperlink テキストとして表示する。
DataGridTemplateColumn	CellTemplate プロパティに DataTemplate クラスを指定することで多彩な表現ができる。また、CellEditingTemplate プロパティには編集モードで表示するための DataTemplate クラスを指定できる。
DataGridTextColumn	シンプルにテキストを表示する。編集モードでは TextBox コントロールとなる。

どのクラスにも共通して、列ヘッダに表示するコンテンツを指定するための Header プロパティと、このセルに表示するデータを指定するための Binding プロパティがあります。また、IsReadOnly プロパティを true にすることで、そのセルが編集モードにならないようにできます。他にも、列ヘッダの Style を指定するための HeaderStyle プロパティや、セルの Style を指定するための CellStyle プロパティもあるため、ちょっとしたカスタマイズも簡単にできます。

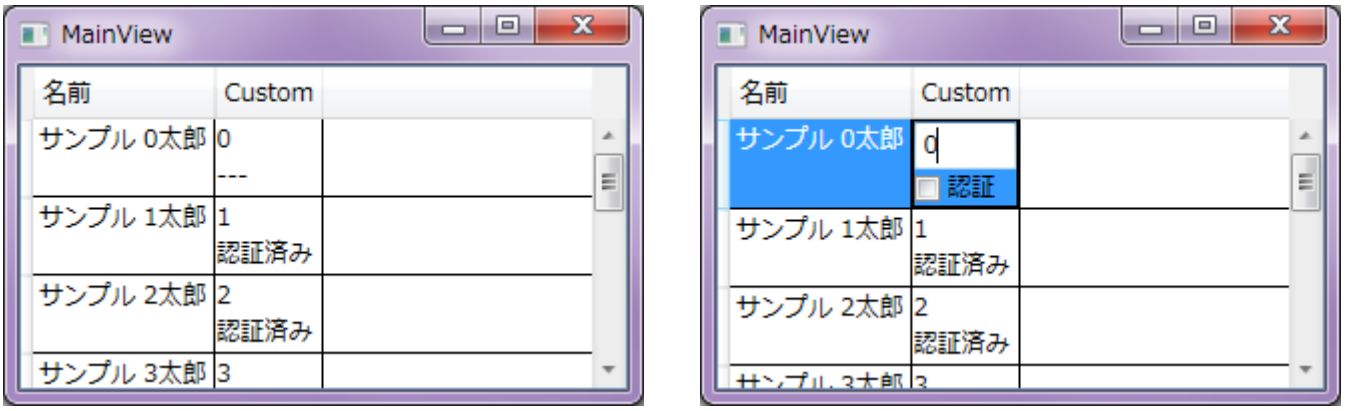
ちょっと複雑なことをする場合は DataGridTemplateColumn クラスを使用します。これは他のクラスと違って、データを表示するための DataTemplate クラスを DataGridTemplateColumn.CellTemplate プロパティに、編集モードが必要な場合はそのための DataTemplate クラスを DataGridTemlateColumn.CellEditingTemplate プロパティに指定する必要があります。

下記のサンプルでは、DataGridTemplateColumn クラスを使用して、通常の表示と編集モードでの表示を変更しています。

コード 6.32：AutoGenerateColumns プロパティの指定

MainWindow.xaml

```
<Window x:Class="Sample_DataGrid.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="200" Width="320">
    <DataGrid ItemsSource="{Binding People}" AutoGenerateColumns="False">
        <DataGrid.Columns>
            <DataGridTextColumn Header="名前" Binding="{Binding Name}" />
            <DataGridTemplateColumn Header="Custom">
                <DataGridTemplateColumn.CellTemplate>
                    <DataTemplate>
                        <StackPanel>
                            <TextBlock Text="{Binding Age}" />
                            <TextBlock>
                                <TextBlock.Style>
                                    <Style TargetType="TextBlock">
                                        <Setter Property="Text" Value="---" />
                                        <Style.Triggers>
                                            <DataTrigger Binding="{Binding IsAuthenticated}" Value="True">
                                                <Setter Property="Text" Value="認証済み" />
                                            </DataTrigger>
                                        </Style.Triggers>
                                    </Style>
                                </TextBlock.Style>
                            </TextBlock>
                        </StackPanel>
                    </DataTemplate>
                </DataGridTemplateColumn.CellTemplate>
                <DataGridTemplateColumn.CellEditingTemplate>
                    <DataTemplate>
                        <StackPanel>
                            <TextBox Text="{Binding Age}" />
                            <CheckBox Content="認証" IsChecked="{Binding IsAuthenticated}" />
                        </StackPanel>
                    </DataTemplate>
                </DataGridTemplateColumn.CellEditingTemplate>
            </DataGridTemplateColumn>
        </DataGrid.Columns>
    </DataGrid>
</Window>
```



(a) 通常の表示 (b) 編集モード

図 6.18：DataGridTemplateColumn クラスの使用例

7 その他の表示に関するコントロール

この章ではデータを表示するための支援をおこなうコントロールについて紹介します。

7.1 Border コントロール

コントロールに枠の装飾を施すためのコントロールです。

コード 7.1：Border コントロールの配置

MainWindow.xaml

```
<Window x:Class="Sample_Border.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="200" Width="250"
        Background="Cornsilk">
    <StackPanel Margin="20" Background="White">
        <Border Background="Plum" BorderBrush="Black" BorderThickness="1">
            <TextBlock Text="囲まれるコンテンツ1" />
        </Border>

        <Border BorderBrush="Black" BorderThickness="1,4,8,12" Margin="0,10,0,0">
            <TextBlock Text="囲まれるコンテンツ2" />
        </Border>

        <Border BorderBrush="Green" BorderThickness="2" CornerRadius="4" Margin="0,10,0,0">
            <TextBlock Text="囲まれるコンテンツ3" />
        </Border>
    </StackPanel>
</Window>
```

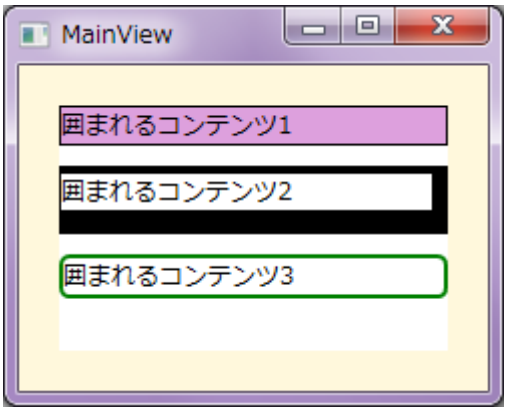


図 7.1：Border コントロールの使用例

このように、BorderBrush プロパティに境界線の色、BorderThickness プロパティに境界線の太さを指定します。BorderBrush プロパティは Brush 型なので、サンプルのように色名を指定することで単純な塗潰しの色を指定することもできますが、LinearGradientBrush クラスなどもしていできます。BorderThickness プロパティは Thickness 構造体で、数値をひとつだけ指定すると 4 辺とも同じ太さになりますが、4 つの数値を別々に指定することで 4 辺別々の太さにもできます。数値は左、上、右、下の順で並べることになります。また、2 つの数値のみを指定することで、左右と上下に対して数値を指定することもできます。

CornerRadius プロパティは、4 隅の丸みを指定できます。0 が既定値となっていますが、数値を大きくすればするほど角が丸くなります。CornerRadius プロパティも BorderThickness プロパティと同様に 2 つまたは 4 つの数値によって指定することもできます。

7.2 ScrollViewer コントロール

与えられた領域からはみ出てしまうコントロールに対してスクロールバーを表示するためのコントロールです。

コード 7.2：ScrollViewer コントロールの配置

MainWindow.xaml

```
 1 <Window x:Class="Sample_ScrollViewer.MainWindow"
 2         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 3         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 4         Title="MainWindow" Height="200" Width="250"
 5         WindowStartupLocation="CenterScreen">
 6     <ScrollViewer VerticalScrollBarVisibility="Auto">
 7         <StackPanel>
 8             <Button Content="No. 1" />
 9             <Button Content="No. 2" />
10             <Button Content="No. 3" />
11             <Button Content="No. 4" />
12             <Button Content="No. 5" />
13             <Button Content="No. 6" />
14             <Button Content="No. 7" />
15             <Button Content="No. 8" />
16             <Button Content="No. 9" />
17             <Button Content="No.10" />
18         </StackPanel>
19     </ScrollViewer>
20 </Window>
```

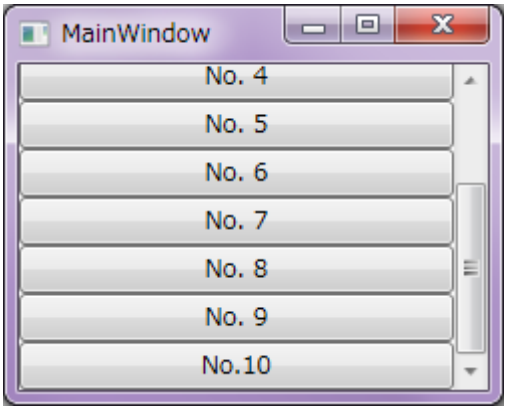


図 7.2：ScrollViewer コントロールによるスクロール

水平方向のスクロールバー表示を設定するプロパティは HorizontalScrollBarVisibility プロパティとなります。

コード 7.3：Orientation プロパティを指定する

MainWindow.xaml

```
 6 <ScrollViewer HorizontalScrollBarVisibility="Auto">
 7     <StackPanel Orientation="Horizontal">
```

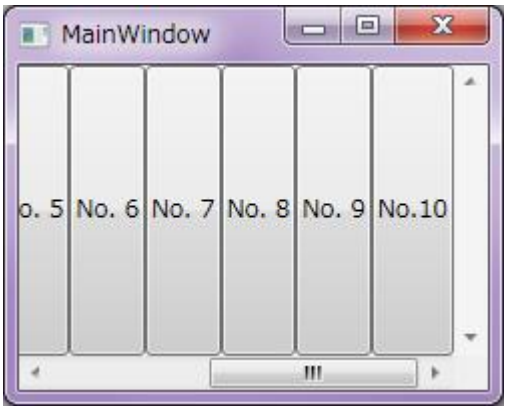


図 7.3：Horizontal を指定した場合

7.3 ViewBox コントロール

コンテンツを拡大/縮小して表示することができるコントロールです。拡大/縮小する方法を Stretch プロパティに、拡大のみ/縮小のみ/両方おこなうことを切り替えるために StretchDirection プロパティを指定します。

Stretch プロパティには下表のような Stretch 列挙体を指定します。ViewBox コントロールでは Uniform が既定値となっています。

表 7.1：Stretch プロパティに指定できる値

値	説 明
None	コンテンツのサイズをそのままに表示する。
Uniform	縦横比を維持しながら対象の領域に収まるようにコンテンツを拡大/縮小する。こうすることで必ずコンテンツ全体が見えるようになる。
UniformToFill	縦横比を維持しながら対象の領域が画像で埋まるようにコンテンツを拡大/縮小する。はみ出た部分は表示されない。
Fill	縦横比に関係なく、対象の領域を埋めるようにコンテンツを拡大/縮小する。コンテンツ全体が見えるようになる。

コード 7.4：ViewBox コントロールの配置

MainWindow.xaml

```
<Window x:Class="Sample_ViewBox.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="200" Width="420">
    <StackPanel Orientation="Horizontal">
        <Viewbox Width="80" Height="80" Stretch="None">
            <Button Content="Button" />
        </Viewbox>
        <Viewbox Width="80" Height="80" Stretch="Fill">
            <Button Content="Button" />
        </Viewbox>
        <Viewbox Width="80" Height="80" Stretch="Uniform">
            <Button Content="Button" />
        </Viewbox>
        <Viewbox Width="80" Height="80" Stretch="UniformToFill">
            <Button Content="Button" />
        </Viewbox>
    </StackPanel>
</Window>
```

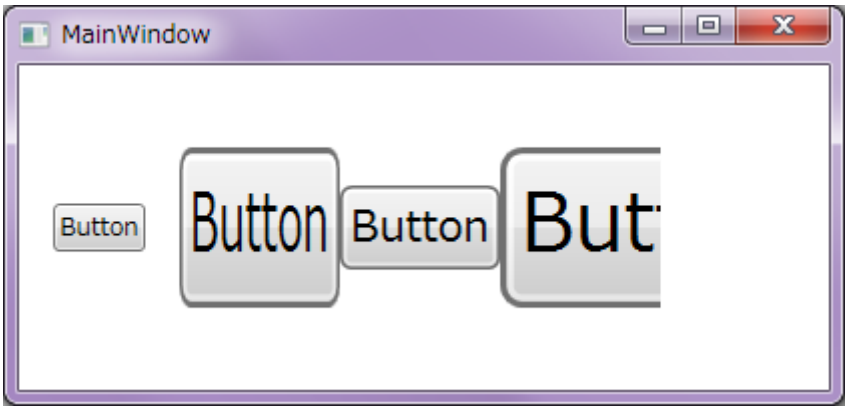


図 7.4：ViewBox コントロールによる拡大表示

7.4 Popup コントロール

Window コントロールや Panel コントロールにオーバーラップして表示するためのコントロールです。

コード 7.5：Popup コントロールの配置

MainWindow.xaml

```
<Window x:Class="Sample_Popup.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="200" Width="250">
    <Grid>
        <ToggleButton x:Name="button1"
                      Content="ポップアップ表示"
                      HorizontalAlignment="Center"
                      VerticalAlignment="Center"
                      IsChecked="{Binding IsOpen, ElementName=popup1}"
                      />
        <Popup x:Name="popup1"
               PlacementTarget="{Binding ElementName=button1}"
               Placement="Bottom"
               StaysOpen="False"
               HorizontalOffset="0"
               VerticalOffset="1">
            <Border Background="White"
                    BorderBrush="Black"
                    BorderThickness="1"
                    Height="30"
                    Padding="10,5">
                <TextBlock Text="ポップアップ"
                           TextAlignment="Center"
                           VerticalAlignment="Center"
                           />
            </Border>
        </Popup>
    </Grid>
</Window>
```

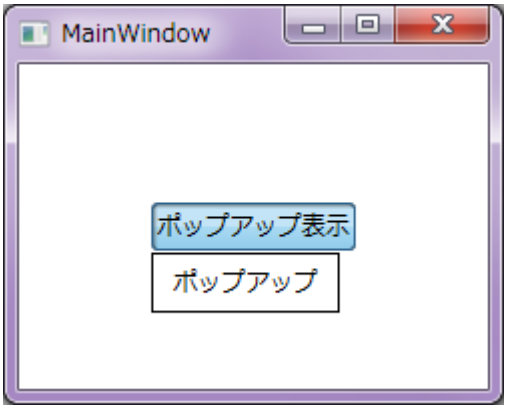


図 7.5：ボタンを押すとポップアップが表示される

Popup コントロールは IsOpen プロパティが true のときに表示され、false のときは表示されません。この切り替えを制御できるようにするために、上記のサンプルでは ToggleButton コントロールの IsChecked プロパティとデータバインディング機能で結んでいます。こうすることで、ToggleButton コントロールをクリックすることで Popup コントロールの表示/非表示を切り替えられるようになります。

StaysOpen プロパティに false が指定されている場合、ポップアップ表示中に Popup コントロール以外でマウスのボタンを押すと自動的に Popup コントロールの IsOpen プロパティが false になり、非表示になります。StaysOpen プロパティを true にしておくと自動的に非表示にならなくなります。

Popup コントロールの表示位置は PlacementTarget プロパティと Placement プロパティで指定します。PlacementTarget プロパティではポップアップの基準位置を決めるためのコントロールを指定します。上記のサンプルでは ToggleButton コントロールを基準としています。Placement プロパティでは、基準に対してどの位置に表示するかを指定します。上記のサンプルでは ToggleButton コントロールの下部に表示するために Placement.Bottom を指定します。

HorizontalOffset プロパティで水平方向の、VerticalOffset プロパティで垂直方向の相対位置を指定することで Popup コントロールの表示位置を微調整できます。

7.5 Path クラスによる描画

本節では、System.Windows.Shapes.Path クラスを用いた幾何学図形の描画方法を紹介します。

7.5.1 Path クラス

System.Windows.Shapes.Path クラスは曲線や複雑な図形を描画するための FrameworkElement です。Data プロパティに 描画するための図形を表現した System.Windows.Media.Geometry クラスをセットします。

Data プロパティはあくまでも図形の定義のみで、色や線の太さといった情報は Fill プロパティや Stroke プロパティ、StrokeThickness プロパティで指定します。この仕組みから、ひとつの Path クラスで複数の塗潰し色を持った図形を描画することはできません。描画に使用する主なプロパティを下表にまとめます。この他に StrokeDashArray プロパティなど細かいものもあります。

表 7.2：描画のための主なプロパティ

プロパティ名	説 明
Fill	図形の内部を塗潰す方法を指定する System.Windows.Media.Brush を取得または設定します。
Stroke	境界線を描画する方法を指定する System.Windows.Media.Brush を取得または設定します。
StrokeThickness	境界線の太さを取得または設定します。

7.5.2 Geometry クラス

幾何学図形を定義するための抽象基底クラスです。実際に使用するクラスは下表に示すような派生クラスとなります。

表 7.3：図形を定義するための Geometry クラス派生のクラス

クラス名	説 明
LineGeometry	直線を定義します。
EllipseGeometry	楕円を定義します。
RectangleGeometry	四角形を定義します。
CombinedGeometry	2 つの Geometry オブジェクトを組み合わせることで図形を定義します。
GeometryGroup	他の Geometry オブジェクトで構成することで、複合ジオメトリとして定義します。
PathGeometry	円弧、曲線、楕円、直線、四角形で構成できる複雑な図形を定義します。
StreamGeometry	PathGeometry の代替となる軽量なものです。データバインディング、アニメーション、変更には対応していません。

特に GeometryGroup クラスは、Geometry クラスから派生しているすべてのクラスをいくつでも組み合わせることができるため、どんな複雑な図形でも描画することができます。

7.5.3 LineGeometry クラス

LineGeometry クラスを使うことで直線を描画できます。設定する主なプロパティを下表にまとめます。開始点と終点があれば直線が描画できるので、非常にシンプルです。点の座標は Path クラスを配置する領域の左上点が原点となります。

表 7.4：LineGeometry クラスの主なプロパティ

プロパティ名	説　明
StartPoint	線の開始点を取得または設定します。
EndPoint	線の終点を取得または設定します。

例えば次のコードでは、原点から右下にかけて直線を描画しています。Geometry クラスはあくまでも図形の定義をおこなうだけで、色に関しては Path クラスが指定します。LineGeometry クラスで定義する図形は線だけなので、Path クラスの Stroke プロパティによってその線色を指定します。また、StrokeThickness プロパティによって線の太さも指定できます。

コード 7.6：LineGeometry の使用例

MainView.xaml

```
1  <Window x:Class="Tips_Path.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainWindow" Height="350" Width="525">
5      <Grid>
6          <Path Stroke="Black" StrokeThickness="2">
7              <Path.Data>
8                  <LineGeometry StartPoint="0,0" EndPoint="10,10" />
9              </Path.Data>
10         </Path>
11     </Grid>
12 </Window>
```

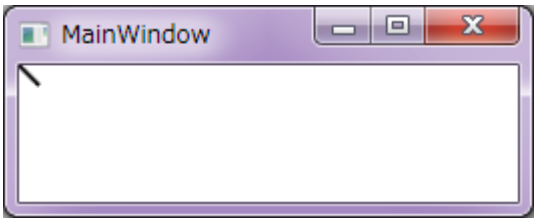


図 7.6：指定された点を結ぶ直線が描画される

7.5.4 EllipseGeometry クラス

EllipseGeometry クラスを使うことで円または楕円を描画できます。設定する主なプロパティを下表にまとめます。

表 7.5：EllipseGeometry クラスの主なプロパティ

プロパティ名	説　明
Center	中心点を取得または設定します。
RadiusX	横軸方向の半径を取得または設定します。
RadiusY	縦軸方向の半径を取得または設定します。

次のコードでは横長の楕円を描画しています。Center プロパティは既定値で座標 (0, 0)、つまり左上点となっているので、このサンプルでは半径と同じ値を指定することで図形がぴったり表示されるようにしています。

コード 7.7：EllipseGeometry の使用例

MainView.xaml

```
1  <Window x:Class="Tips_Path.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainWindow" Height="350" Width="525">
5      <Grid>
6          <Path Fill="SkyBlue" Stroke="Black" StrokeThickness="2">
7              <Path.Data>
8                  <EllipseGeometry Center="20,10" RadiusX="20" RadiusY="10" />
9              </Path.Data>
10         </Path>
11     </Grid>
12 </Window>
```

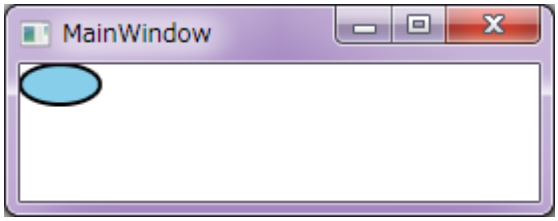


図 7.7：横長の楕円が描画される

7.5.5 RectangleGeometry クラス

RectangleGeometry クラスを使うことで四角形を描画できます。設定する主なプロパティを下表にまとめます。

表 7.6：RectangleGeometry クラスの主なプロパティ

プロパティ名	説　明
Rect	四角形のサイズを取得または設定します。
RadiusX	四角形の角に丸みを付けるために使用する楕円の横軸方向の半径を取得または設定します。
RadiusY	四角形の角に丸みを付けるために使用する楕円の縦軸方向の半径を取得または設定します。

次のコードでは角の丸い四角形を描画しています。Rect プロパティではサイズの他に左上座標も指定できます。ここでは原点を指定しているので、Path クラスが配置されている領域の左上点に四角形が配置されます。

コード 7.8：RectangleGeometry の使用例

MainView.xaml

```
1  <Window x:Class="Tips_Path.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainWindow" Height="350" Width="525">
5      <Grid>
6          <Path Fill="SkyBlue" Stroke="Black" StrokeThickness="2">
7              <Path.Data>
8                  <RectangleGeometry Rect="0,0,100,100" RadiusX="10" RadiusY="10" />
9              </Path.Data>
10         </Path>
11     </Grid>
12 </Window>
```

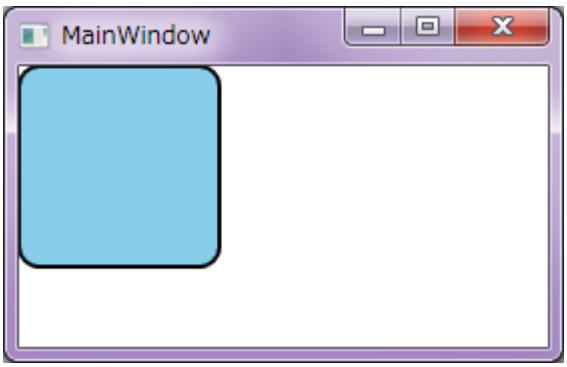


図 7.8：角の丸い四角形が描画される

7.5.6 CombinedGeometry クラス

CombinedGeometry クラスを使うことで 2 つの Geometry クラスを結合することができます。設定する主なプロパティを下表にまとめます。

表 7.7：CombinedGeometry クラスの主なプロパティ

プロパティ名	説　明
Geometry1	最初の Geometry オブジェクトを取得または設定します。
Geometry2	2 番目の Geometry オブジェクトを取得または設定します。
GeometryCombineMode	2 つの Geometry オブジェクトを結合する方法を取得または設定します。

次のコードでは 2 つの EllipseGeometry オブジェクトを結合して描画しています。geometryCombineMode プロパティに Xor を指定しているため、2 つのジオメトリが重なり合う部分がくり抜かれて描画されています。

コード 7.9：CombinedGeometry の使用例

MainView.xaml

```
1  <Window x:Class="Tips_Path.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainWindow" Height="150" Width="300" Background="Cornsilk">
5      <Grid>
6          <Path Fill="SkyBlue" Stroke="Black" StrokeThickness="2">
7              <Path.Data>
8                  <CombinedGeometry GeometryCombineMode="Xor">
9                      <CombinedGeometry.Geometry1>
10                         <EllipseGeometry Center="20,20" RadiusX="20" RadiusY="20" />
11                     </CombinedGeometry.Geometry1>
12                     <CombinedGeometry.Geometry2>
13                         <EllipseGeometry Center="20,20" RadiusX="6" RadiusY="6" />
14                     </CombinedGeometry.Geometry2>
15                 </CombinedGeometry>
16             </Path.Data>
17         </Path>
18     </Grid>
19 </Window>
```

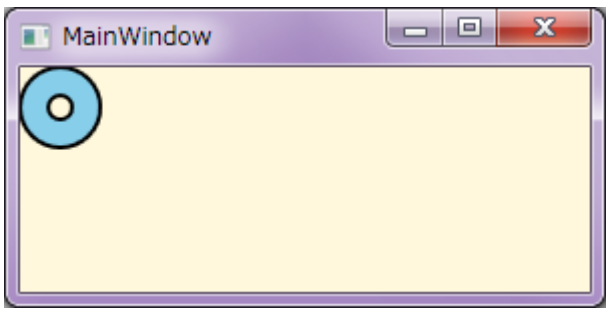


図 7.9：2 つの楕円の組み合わせが描画される

7.5.7 GeometryGroup クラス

複数の Geometry クラスを組み合わせる場合は GeometryGroup クラスがよく使われます。設定する主なプロパティを下表にまとめます。

表 7.8：GeometryGroup クラスの主なプロパティ

プロパティ名	説 明
FillRule	この GeometryGroup に含まれるオブジェクトの交差する領域がどのように結合されるかを取得または設定します。
Children	組み合わせる Geometry オブジェクトを含む GeometryCollection を取得または設定します。

次のコードでは 3 つの EllipseGeometry オブジェクトを結合して描画しています。FillRule プロパティによって描画結果が変わるため、かなり複雑な図形が描画できます。

コード 7.10：GeometryGroup の使用例

MainView.xaml

```
1  <Window x:Class="Tips_Path.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainWindow" Height="150" Width="300" Background="Cornsilk">
5      <Grid>
6          <Path Fill="SkyBlue" Stroke="Black" StrokeThickness="2">
7              <Path.Data>
8                  <GeometryGroup>
9                      <EllipseGeometry Center="20,20" RadiusX="20" RadiusY="20" />
10                     <EllipseGeometry Center="20,20" RadiusX="14" RadiusY="14" />
11                     <EllipseGeometry Center="20,20" RadiusX="6" RadiusY="6" />
12                 </GeometryGroup>
13             </Path.Data>
14         </Path>
15     </Grid>
16 </Window>
```

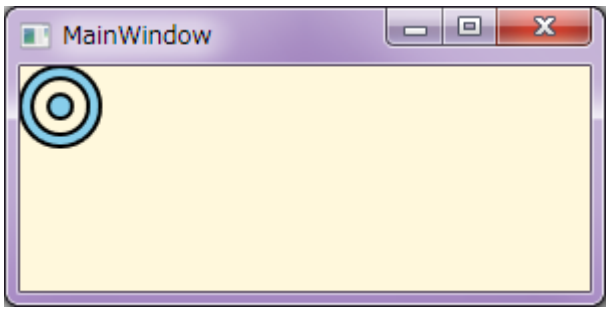


図 7.10：3 つの楕円の組み合わせが描画される

7.5.8 PathGeometry クラス

PathGeometry クラスは、直線、曲線、円弧の組み合わせから成る図形を定義するときに使います。設定する主なプロパティを下表にまとめます。

表 7.9：PathGeometry クラスの主なプロパティ

プロパティ名	説　明
FillRule	この GeometryGroup に含まれるオブジェクトの交差する領域がどのように結合されるかを取得または設定します。
Figures	パスのコンテンツを記述する PathFigure オブジェクトのコレクションを取得または設定します。

次のコードでは 2 つの PathFigure オブジェクトを Figures プロパティに暗黙的に指定することで、複雑な図形を描画しています。1 つ目の PathFigure オブジェクトでは、LineSegment クラスを使用して四角形を描画しています。2 つ目の PathFigure オブジェクトでは、ArcSegment クラスを使用して円弧を描画しています。PathGeometry クラスの FillRule プロパティに EvenOdd を指定しているため、2 つの PathFigure オブジェクトが重なり合う部分がくり抜かれて描画されています。

コード 7.11：PathGeometry の使用例

MainView.xaml

```
<Window x:Class="Tips_Path.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="150" Width="300" Background="Cornsilk">
    <Grid>
        <Path Fill="SkyBlue" Stroke="Black" StrokeThickness="2">
            <Path.Data>
                <PathGeometry FillRule="EvenOdd">
                    <PathFigure StartPoint="20,20" IsClosed="True">
                        <LineSegment Point="20,30" />
                        <LineSegment Point="30,30" />
                        <LineSegment Point="30,20" />
                    </PathFigure>
                    <PathFigure StartPoint="50,0" IsClosed="True">
                        <ArcSegment Point="40,40" Size="20,10" />
                    </PathFigure>
                </PathGeometry>
            </Path.Data>
        </Path>
    </Grid>
</Window>
```

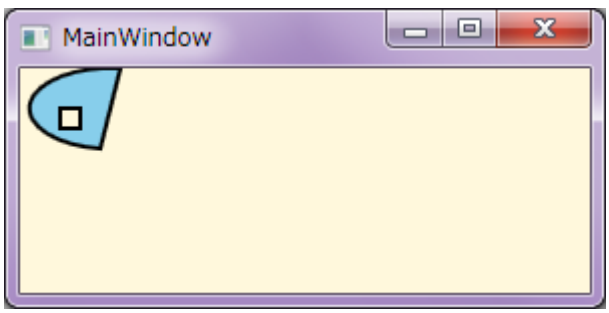


図 7.11：複数の PathFigure オブジェクトの組み合わせが描画される

7.5.9 StreamGeometry クラス

PathGeometry クラスを使用することで複雑な図形を定義することができますが、例えば 10,000 個の線から成る図形だった場合、描画にかかる時間は膨大になります。このような場合、データバインディングなどの機能を省略した StreamGeometry クラスを使用することで、描画時間を大幅に短縮できます。

StreamGeometry クラスの使い方は特殊で、Stream をオープンしなければならないため、必ず C# コード上で作業することになります。

コード 7.12：StreamGeometry を使用するために Path オブジェクトを配置する

MainView.xaml

```
<Window x:Class="Tips_Path.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="150" Width="300" Background="Cornsilk"
        Loaded="Window_Loaded">
    <Canvas>
        <Path x:Name="path1" Fill="SkyBlue" Stroke="Black" StrokeThickness="2" />
    </Canvas>
</Window>
```

コード 7.13：StreamGeometry の使用例

MainView.xaml.cs

```
namespace Tips_Path
{
    using System.Windows;
    using System.Windows.Media;
    using System.Windows.Shapes;

    /// <summary>
    /// MainWindow.xaml の相互作用ロジック
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            var geometry = new StreamGeometry();
            geometry.FillRule = FillRule.EvenOdd;

            using (var context = geometry.Open())
            {
                context.BeginFigure(new Point(0, 0), true, true);
                context.LineTo(new Point(20, 0), true, true);
                context.ArcTo(new Point(20, 20), new Size(10, 10), 0.4, false,
SweepDirection.Clockwise, true, true);
                context.LineTo(new Point(0, 20), true, true);
            }
            geometry.Freeze();
            this.path1.Data = geometry;
        }
    }
}
```

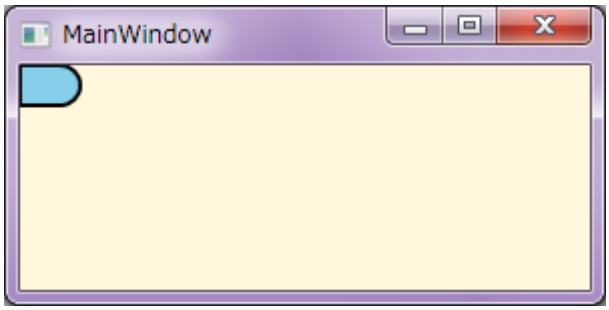


図 7.12：描画結果は他の Geometry クラスと同じようになる

7.5.10 パス マークアップ構文

ジオメトリを指定するとき、よりコンパクトに指定するときに使用できる、強力かつ複雑なミニ言語というものがあります。Path クラスの Data プロパティに Geometry クラスを指定するときや、PathGeometry クラスの Figures プロパティに PathFigureCollection を指定するときに使うことができます。

ミニ言語の構文はシンプルで、覚えると簡単な図形はこのミニ言語のみで定義できるようになります。その構文を下表にまとめます。特殊な値として、Infinity、-Infinity、NaN も使えます。

表 7.10：パス マークアップ構文

種別	構文	説 明
点の構文	*x,y*	数値をカンマ区切りで記述します。 *x*：点の x 座標 *y*：点の y 座標
移動コマンド	M *startPoint* または m *startPoint*	新しい図形の始点を指定します。 大文字の M は startPoint が絶対値であることを示し、小文字の m は startPoint が前の点へのオフセットであるか、前の点が存在しない場合は (0, 0) であることを示します。
線コマンド	L *endPoint*	現在の点と指定した終点を直線で結びます。
横線コマンド	H *x*	現在の点と指定した *x* 座標を横線で結びます。
縦線コマンド	V *y*	現在の点と指定した *y* 座標を縦線で結びます。
3 次ベジエ曲線コマンド	C *p1 p2 endPoint*	指定した 2 つの制御点 (*p1* と *p2*) を使用して、現在の点と指定した終点の間に 3 次ベジエ曲線を作成します。 *p1*：曲線の開始接線を決定する制御点 *p2*：曲線の終了接線を決定する制御点
2 次ベジエ曲線コマンド	Q *p1 endPoint*	指定した制御点を使用して、現在の点と指定した終点の間に 2 次ベジエ曲線を作成します。
平滑 3 次ベジエ曲線コマンド	S *p2 endPoint*	現在の点と指定した終点の間に 3 次ベジエ曲線を作成します。1 つ目の制御点は、現在の点を基準として前のコマンドの 2 つ目の制御点に点対称となります。前のコマンドが存在しない場合や、前のコマンドが 3 次ベジエ曲線コマンドまたは平滑 3 次ベジエ曲線コマンドでなかった場合、1 つ目の制御点は現在の点と同一のものと見なされます。 *p2*：曲線の終了接線を決定する制御点
平滑 2 次ベジエ曲線コマンド	T *p1 endPoint*	現在の点と指定した終点の間に 2 次ベジエ曲線を作成します。制御点は、現在の点を基準として前のコマンドの制御点に点対称となります。前のコマンドが存在しない場合や、前のコマンドが 2 次ベジエ曲線コマンドまたは平滑 2 次ベジエ曲線コマンドでなかった場合、制御点は現在の点と同一のものと見なされます。 *p1*：曲線の開始接線を決定する制御点
楕円の円弧コマンド	A *size rotationAngle isLargeArcFlag sweepDirectionFlag endPoint*	
	現在の点と指定した終点の間に楕円の円弧を作成します。 size：円弧の x 半径と y 半径 rotationAngle：円弧の回転角度 isLargeArcFlag：円弧の角度が 180 度を超える場合は 1 に設定します。それ以外の場合は 0 を設定します。 sweepDirectionFlag：円弧を正の角の方向に描画する場合は 1 を設定します。それ以外の場合は 0 を設定します。	

閉じるコマンド	Z	現在の図形を終了し、現在の点と図形の開始点を結ぶ線を作成します。このコマンドは、図形の最後のセグメントと最初のセグメントの間に線結合を作成します。

次のコードでは、Path クラスの Data プロパティに対してミニ言語による Geometry の指定をおこなうことで、StreamGeometry クラスを指定することになります。

コード 7.14：StreamGeometry の作成例

MainView.xaml

```
<Window x:Class="Tips_Path.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="180" Width="320" Background="Cornsilk">
    <Canvas>
        <Path Data="M 10,100 C 10,300 300,-200 300,100"
              Fill="SkyBlue"
              Stroke="Black"
              StrokeThickness="2"
              />
    </Canvas>
</Window>
```

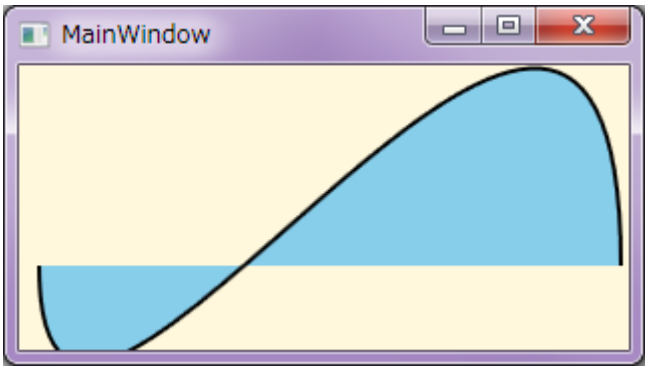


図 7.13：ミニ言語で簡単に曲線を描画できる

次のコードでは、PathGeometry クラスの Figures プロパティに対してミニ言語による PathFigureCollection の指定をおこなうことで、Path クラスの Data プロパティに PathGeometry クラスを指定しています。

コード 7.15：PathFigureCollection の作成例

MainView.xaml

```
<Window x:Class="Tips_Path.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="180" Width="320" Background="Cornsilk">
    <Canvas>
        <Path Fill="SkyBlue" Stroke="Black" StrokeThickness="2">
            <Path.Data>
                <PathGeometry Figures="M 10,100 C 10,300 300,-200 300,100" />
            </Path.Data>
        </Path>
    </Canvas>
</Window>
```

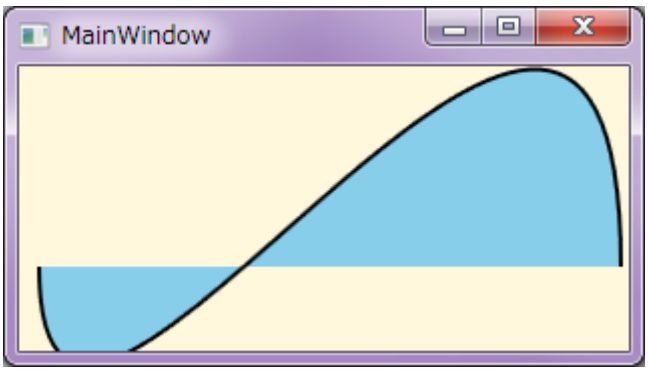


図 7.14：ミニ言語で簡単に曲線を描画できる

7.5.11 文字列を Geometry に変換する

Path として扱えるのは楕円や四角形などの図形だけですが、文字列を Geometry に変換することで、文字列を図形として扱えるようにもなります。文字列を Geometry に変換するには、文字列を FormattedText クラスで生成する必要があります。そして、FormattedText クラスの BuildGeometry() メソッドを使って変換します。

文字列を Geometry に変換した後、塗潰し色を指定したり境界線の色や太さを指定したりすることができます。塗潰し色を変更するだけでは、TextBlock コントロールでテキストを表示するほうが簡単で処理速度も速いですが、透明色の文字列は TextBlock コントロールでは表現できません。せっかくなので、ここでは透明色の文字列を表現するためのクラスを定義し、これを実際に使ってみます。

透明色の文字列を表現するためのクラスを CreativeText クラスとして次のようなコードで定義します。

コード 7.16：透明色の文字列でくり抜いた四角形を描画するクラス

CreativeText.cs

```
namespace WpfApplication2
{
    using System.Globalization;
    using System.Windows;
    using System.Windows.Media;

    /// <summary>
    /// 指定された文字列でくり抜いた四角形を表現するためのクラスです。
    /// </summary>
    public class CreativeText : FrameworkElement
    {
        /// <summary>
        /// Text 依存関係プロパティの定義
        /// </summary>
        public static readonly DependencyProperty TextProperty =
DependencyProperty.Register("Text", typeof(string), typeof(CreativeText), new
FrameworkPropertyMetadata(null, FrameworkPropertyMetadataOptions.AffectsRender));

        /// <summary>
        /// テキストを取得または設定します。
        /// </summary>
        public string Text
        {
            get { return (string)GetValue(TextProperty); }
            set { SetValue(TextProperty, value); }
        }

        /// <summary>
        /// 描画処理オーバーライド
        /// </summary>
        /// <param name="drawingContext">描画のためのコンテキストが渡されます。</param>
        protected override void OnRender(DrawingContext drawingContext)
        {
            var formattedText = new FormattedText(
                this.Text,
                CultureInfo.CurrentUICulture,
                FlowDirection.LeftToRight,
                new Typeface("Consolas"),
                80,
                Brushes.Black);
            var textGeometry = formattedText.BuildGeometry(new Point(0, 0));
            var rectGeometry = new RectangleGeometry(new Rect(new Size(formattedText.Width,
formattedText.Height)));
```

```
41              var geometry = new CombinedGeometry(GeometryCombineMode.Xor, rectGeometry,
    textGeometry);
42              drawingContext.DrawGeometry(Brushes.SkyBlue, new Pen(Brushes.Black, 2.0),
    geometry);
43          }
44      }
45  }
```

12～24 行目では Text 依存関係プロパティを定義しています。こうすることで、XAML 上から任意のテキストを指定できるようになります。他にも背景色なども指定できるようにするための依存関係プロパティを定義してもいいですが、コードが煩雑になってしまうため、ここでは Text 依存関係プロパティのみとします。

描画処理をおこなう OnRender() メソッドをオーバーライドして、描画内容を自分で指定しています。32 行目で指定されたテキストを使って FormattedText クラスを生成し、39 行目で BuildGeometry() メソッドを使って Geometry に変換しています。

今回は透明色の文字列を表現するために、透明でない四角形と結合させます。この四角形を 40 行目で定義し、41 行目で CombinedGeometry として結合しています。

最後に、42 行目で結合した Geometry を描画しています。

以上で定義した CreativeText クラスの使用例が次のコードになります。ウィンドウに背景色を指定していますが、テキストの部分がその背景色となっていることから、文字列が透明色であることがわかります。

コード 7.17：CreativeText クラスの使用例

MainWindow.xaml

```
1  <Window x:Class="WpfApplication2.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          xmlns:local="clr-namespace:WpfApplication2"
5          Title="MainWindow" Height="200" Width="730"
6          Background="Cornsilk">
7      <Grid>
8          <local:CreativeText Text="Transparent text" />
9      </Grid>
10 </Window>
```

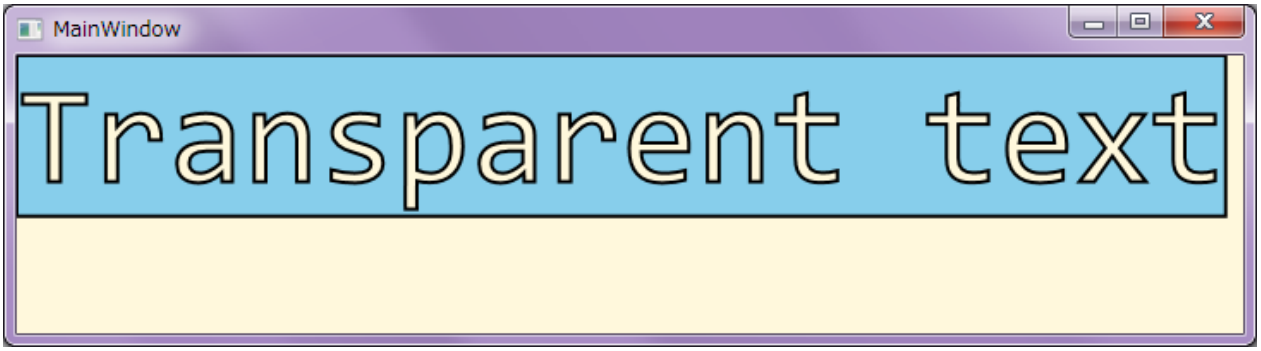


図 7.15：透明色の文字列でくり抜かれた四角形が表示されている

8 WPF の基本的な開発手順

この章では WPF の基本的な開発手順として、まず MVVM パターンの基本的な考え方について紹介します。また、実際のコーディング作業をする上で MVVM パターンを意識した内部構造の作成方法を紹介します。その後、データバインディング機能の要となる INotifyPropertyChanged インターフェースおよび ICommand インターフェースの実装例を紹介し、データバインディング機能の基礎について簡単に紹介します。

8.1 MVVM パターンについての基本的な考え方

はじめに、MVVM パターンはあくまでもひとつの方針である、ということを覚えておいてください。必ずしもこのパターンにはめる必要はありませんが、理想的なパターンを知っておくことで、自分の開発スタイルが理想からどれだけかけ離れているか、もしくは近いかを測ることができると思います。理想形を知っているのと知らないのとでは大きな差があるので、参考程度に一読し、実践できる部分は実践するように意識してください。

MVVM パターン（Model-View-ViewModel Pattern）とは、アプリケーションの内部構造を Model、View、ViewModel の 3 つに大別して開発をおこなうソフトウェアアーキテクチャの 1 つです。最大の特徴は、ロジックと UI が分離されることで、プログラマーとデザイナーの分業が比較的容易となることです。どちらも同じ人が担当する場合でも、データを操作するプログラム部分と見た目を表現するデザイン部分が明確に区切られるため、メンテナンスが非常にやりやすくなります。

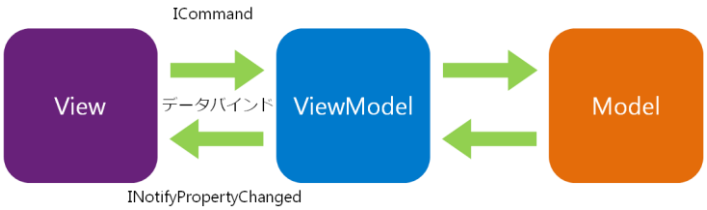


図 8.1：MVVM パターン概略図

3 つの役割をおおまかにいうと、Model はアプリケーションの中核となる処理、View はユーザーが直接触れる GUI、ViewModel は View に表示する情報の保持および変換をそれぞれ担当します。これらは理想的には完全に分離できるはずで、実際のコードもこれだけ綺麗に分かれるとアプリケーションの内部構造が非常に明確で、いわゆるスパゲティコードにならずに済みます。

MVVM パターンは、ロジックと UI（Model と View）の結合を疎にすることが目的です。ロジックと UI の結合を疎にすることで、中～大規模なアプリケーション開発の生産性が向上されることが期待されます。逆にロジックと UI の結合が密になると、GUI をデザインするときにロジックの内部構造をすべて把握する必要があり、GUI のデザインをするのにロジック部分の知識が必要となってしまいます。小規模なアプリケーションであれば、GUI のデザインと同時にロジックの部分も変更することは容易ですが、中～大規模なアプリケーションになってくるとそう簡単にはいきません。まして複数の人が関わるアプリケーション開発において、そのような作業が困難であることは明白です。

一方、小規模なアプリケーション開発の場合、内部構造を限定する MVVM パターンは単なる足枷になり得ます。実際のコードに触れるとわかると思いますが、MVVM パターンを意識し過ぎると、「これは ViewModel の役割だからこっちにコードを書くのはおかしい」とか、「これは View の操作だから ViewModel から触るのは NG」とかいったことが発生します。しかし、MVVM パターンに則った内部構造を実現するために、これらの問題を解決することに注力すべきではありませんし、往々にしてできあがるものは複雑な仕組みとなってしまいます。これでは開発の生産性を向上させるためのソフトウェアアーキテクチャとしては本末転倒な話です。「MVVM パターンとして見るとこれは NG かも知れないけど、こっちのほうがコードも見やすいし、メンテナンスも問題ないね。」と判断できればそれでいいのです。完全に MVVM パターンに当てはまらなければならない、なんてことは誰も言っていません。つまり、MVVM パターンはあくまでもひとつの方針として考え、必ずしも MVVM パターンに縛られる必要はないということです。もちろん小規模アプリケーション開発でも、MVVM パターンを意識して作り込むこと

で後々のメンテナンス性に有利に働くかもしれません。あるアプリケーション開発において、Model、View、ViewModel の役割をきちんと線引きできていれば、あるいはその他の役割を持った別のものがあったとしてもそれがそのアプリケーションの開発スタイルとなるのではないでしょうか。MVVM パターンはあくまでもひとつの理想形と考え、そこから自分たちの開発スタイルをどのようにすべきかを考えることで、自分たちにとってより良いアプリケーション開発のフレームワークが見えてくると思います。

ここでは MVVM パターンの理想形について説明します。次に、Model、View、ViewModel の 3 つの分類がそれぞれどんな役割を果たすのかを見ていきます。

8.1.1 Model

Model はそのアプリケーションの最も中核となるロジックの集まりです。極端にいうと、そのアプリケーションを CUI として動作させるコードの集まりが Model となります。したがって、Model は View にも ViewModel にも依存してはいけません。 また、アプリケーションの中核となるため、Model がなければアプリケーションは成り立ちません。というよりは、UI 以外でアプリケーションが満たすべき要求仕様のほんとんどは Model によって実現しているということです。

GUI とは関連のない部分を担っているため、データ構造の定義やアプリケーション外部とのやり取りはすべて Model がおこないます。外部とのやり取りには、ファイルアクセス、ソケット通信、データベースアクセスなどがあります。また、アプリケーション内部で発生したエラーや例外処理も、そのほとんどすべては Model がおこなうべき処理となります。

8.1.2 View

View はユーザーが直接触れる GUI の部分です。View はユーザーからの指令や、それに付随するイベントを発行し、これを ViewModel に伝える必要があります。また、ViewModel が公開する情報をグラフィカルに変換してユーザーに見せます。ユーザーからの指令を ViewModel に伝える仕組みは WPF に組み込まれているため、開発者が意図的にコードを追加する必要はありません。

ViewModel が公開する情報を知る必要があるため、View は ViewModel に依存して作られなければなりません。ただし、View は ViewModel の内部構造を完全に把握する必要はありません。これは、 ロジックと UI の疎結合を目指す MVVM パターンの方針と矛盾してしまうからです。つまり、View が ViewModel のインスタンスを持つ必要はありません。したがって、XAML 上で ViewModel を使用することはありません。

WPF の開発では、GUI のコードはすべて XAML によっておこないます。C# や VB などのコードからコントロールを配置することもできますが、それらはすべて XAML で実現できる上に、XAML でしか実現できない機能も存在します。したがって、コードビハインドを用いてコードを書く必要はありません。ただし、動的なコントロール配置など、標準的な動作以外のことを実現しようとすると、XAML だけで実現することが複雑になってしまう場合があります。この場合、カスタムコントロールやユーザーコントロールを用いることで、複雑な動作をするコントロールを定義することもできます。

8.1.3 ViewModel

ViewModel はユーザーに見せる情報を Model から参照して保持し、必要であれば変換して View に公開します。また、View から受け取ったユーザーからの指令を翻訳して Model を操作するのも ViewModel の役割です。

原則として View と ViewModel は 1 対 1 対応で、複数の View を持つ場合は ViewModel も複数となります。また、ViewModel は Model の影となる存在です。ViewModel 同士に所有関係はあり得ますが、連携をしてはいけません。複数の Model が連携している状況で、さらに ViewModel 同士も連携してしまうと、それだけ内部構造がややこしくなってしまうからです。

8.1.4 複数の View を持つアプリケーション

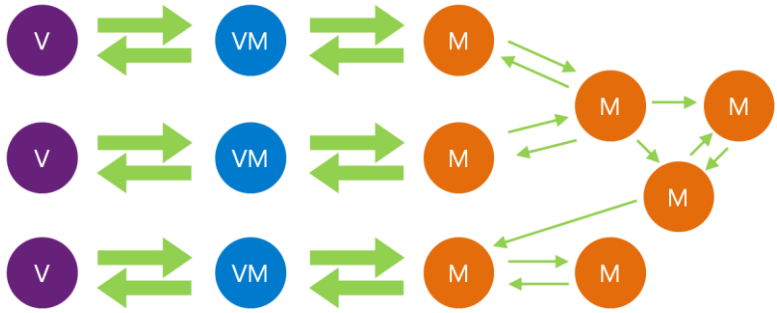


図 8.2：複数の View と Model によるデータ連携

大抵の場合、アプリケーションは複数の画面を持っています。このとき、誰が新たな View のインスタンスを生成すればいいか悩みます。View のコードビハインドで生成した場合、生成された View は、 元の View の所有物となるため、元の View が Close() されると、生成された View は迷子になってしまいます。ところが、ViewModel や Model から View のインスタンスを生成してしまうのは MVVM パターンとしてマナー違反となります。このような場合は、MVVM パターンではなく、MVPVM パターンを用いるという手段もありますが、下図のように、Application クラスがあらかじめ View と ViewModel の Type 情報を保持しており、イベントまたはメソッドによって View インスタンスの生成および削除の管理をおこなうという方法もあります。

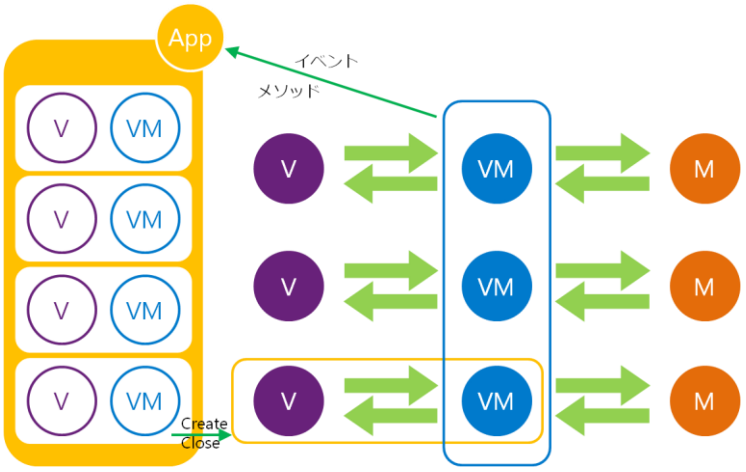


図 8.3：Application クラスによる View-ViewModel インスタンス管理

8.1.5 まとめ

Model、View、View Model それぞれの特徴を簡単にまとめると次のようになります。

- Model はアプリケーションの中核
  CUI として動作させようとして作る部分が Model になり得ます。View にも ViewModel にも依存しません。アプリケーション外部とのやり取りやエラー処理も Model でおこないます。

- View は UI を担当
  ViewModel に依存し、公開されるプロパティを知る必要がありますが、ViewModel のインスタンスを知る必要はありません。ユーザー指示を ViewModel に通知する義務がありますが、これは WPF の仕組みでおこなわれるため、開発者が意図的にコードを追加する必要はありません。

- ViewModel は View に見せる情報を保持
  View からの指示にしたがって Model を操作し、結果を View に公開します。また、それらの情報を保持します。ViewModel のプロパティに変更があった場合は View に通知しなければいけません。

8.2 MVVM パターンを意識した基本プロジェクト作成方法

WPF によるデスクトップアプリケーションを開発する場合、Visual Studio では "WPF アプリケーション" というプロジェクトを選択して開発を進めます。新規プロジェクトを作成するとき、図 8.4 のように Visual C# の下にある "WPF アプリケーション" を選択し、ソリューション名を決定して下さい。

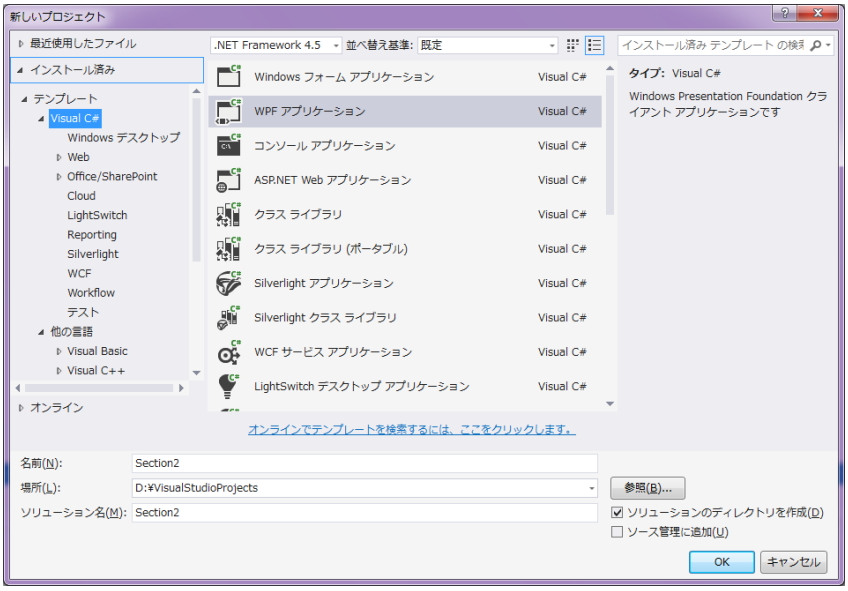


図 8.4：新しいプロジェクト作成ダイアログ

WPF アプリケーションのデフォルトの内部構造は図 8.5 のように App クラスと MainWindow クラスのみとなっています。また、参照設定を見ると、コンソールアプリケーションに比べて PresentationCore、PresentationFramework、System.Xaml、WindowsBase の外部参照が追加されています。

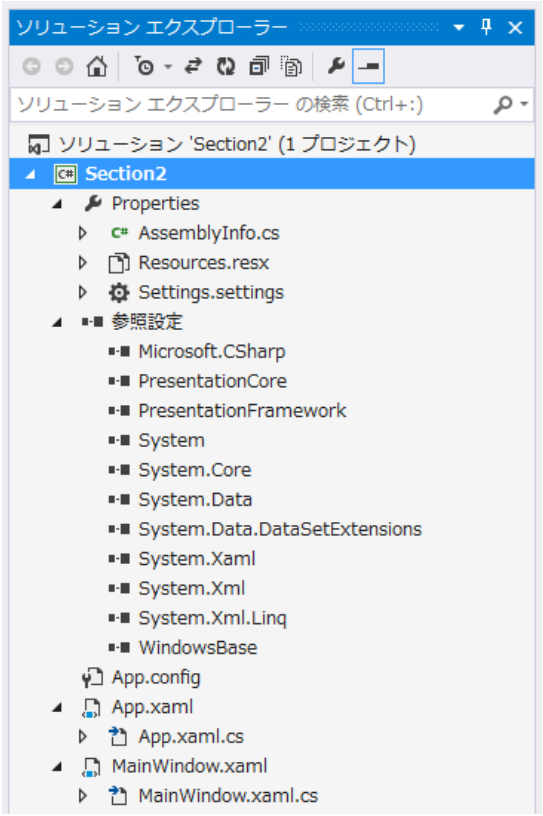


図 8.5：デフォルトの内部構造

WPF は UI とロジックを明確に切り分けて開発を進めることができる MVVM パターンによるプログラミングを支援するシステムとして知られています。しかし、これを有効活用するにはデフォルトの内部構造では少し不便ですので、使いやすいように変更して使います。

具体的には次のような作業となります。

・MainWindow.xaml（および MainWindow.xaml.cs）を削除
・"Views"、"ViewModels"、"Models" という名前のフォルダをツリーに追加
・"Views" フォルダに "MainView.xaml" をウィンドウとして追加
・"ViewModels" フォルダに "MainViewModel.cs" をクラスとして追加
・App.xaml 内で定義されている StartupUri 属性を削除
・App.xaml.cs 内で OnStartup() メソッドをオーバーライド、編集

以上の作業をおこなった後の内部構造は図 8.6 のようになります。また、編集した App.xaml および App.xaml.cs はコード 8.1、コード 8.2 のようになります。

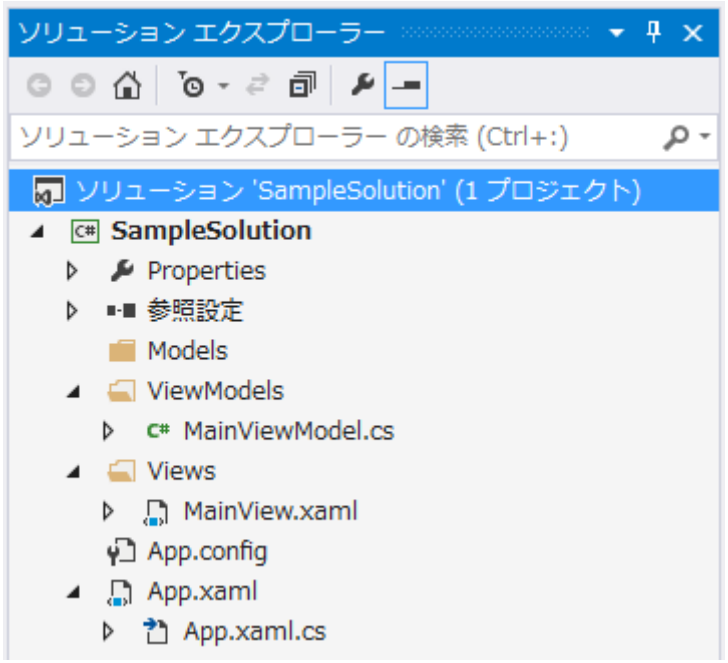


図 8.6：MVVM パターンを意識した内部構造

コード 8.1：StartupUri プロパティを削除した App.xaml

App.xaml

```
1  <Application x:Class="Section2.App"
2               xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3               xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
4      <Application.Resources>
5
6      </Application.Resources>
7  </Application>
```

コード 8.2：OnStartup メソッドを override した App.xaml.cs

App.xaml.cs

```
1  namespace Section2
2  {
3      using System.Windows;
4      using Section2.Views;
5      using Section2.ViewModels;
6
```

```
7       /// <summary>
8       /// App.xaml の相互作用ロジック
9       /// </summary>
10      public partial class App : Application
11      {
12          protected override void OnStartup(StartupEventArgs e)
13          {
14              base.OnStartup(e);
15
16              var w = new MainView();
17              var vm = new MainViewModel();
18
19              w.DataContext = vm;
20              w.Show();
21          }
22      }
23  }
```

このように MainView の DataContext に対して MainViewModel を指定することで、MainView と MainViewModel の間でデータバインディング機能を用いたデータのやり取りができるようになります。しかし、データバインディング機能を使用するためには、ViewModel 側は INotifyPropertyChanged インターフェースや ICommand インターフェースを実装したプロパティを公開する必要があります。

8.3 簡単な UI の作成

データバインディング機能を説明する前に、簡単な UI を作成します。MainView.xaml をコード 8.3 のように編集します。縦にコントロールを並べる StackPanel コントロールの中に、TextBox コントロール、TextBlock コントロール、Button コントロールをひとつずつ配置しただけのシンプルな画面です。これをコンパイルすると図 8.7 のような画面が表示されます。

コード 8.3：コントロールを配置した MainView

MainView.xaml

```
1  <Window x:Class="Section2.Views.MainView"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainView" Height="300" Width="300">
5      <StackPanel>
6          <TextBox Text="Hello world." />
7          <TextBlock Text="Hello world." />
8          <Button Content="Click me." />
9      </StackPanel>
10 </Window>
```

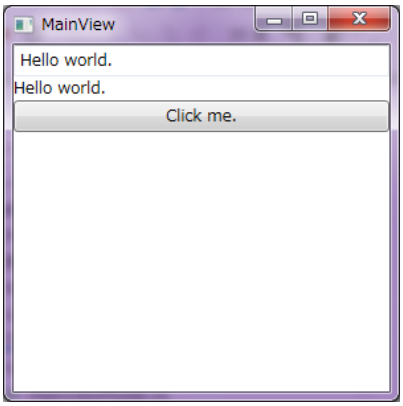


図 8.7：サンプル画面

8.4 INotifyPropertyChanged インターフェースの自前実装と具体例

データバインディング機能を使うために、ViewModel 側に INotifyPropertyChanged インターフェースを実装したプロパティを定義します。すでに公開されている YKToolkit.Controls.dll などのライブラリを利用することで INotifyPropertyChanged インターフェースが既に実装されたクラスを扱うことができますが、ここでは敢えて INotifyPropertyChanged インターフェースを自前で実装して、その内部構造を知ってもらおうと思います。

INotifyPropertyChanged インターフェースを MainViewModel に実装すると次のようなコードになります。

コード 8.4 : INotifyPropertyChanged を実装した MainViewModel クラス

MainViewModel.cs

```
1  namespace Section2.ViewModels
2  {
3      using System.ComponentModel;
4
5      public class MainViewModel : INotifyPropertyChanged
6      {
7          #region INotifyPropertyChanged のメンバ
8          /// <summary>
9          /// プロパティ変更時に発生します。
10         /// </summary>
11         public event PropertyChangedEventHandler PropertyChanged;
12         #endregion INotifyPropertyChanged のメンバ
13
14         /// <summary>
15         /// PropertyChanged イベントを発行します。
16         /// </summary>
17         /// <param name="propertyName">プロパティ名を指定します。</param>
18         protected void RaisePropertyChanged(string propertyName)
19         {
20             var h = PropertyChanged;
21             if (h != null)
22                 h(this, new PropertyChangedEventArgs(propertyName));
23         }
24     }
25 }
```

INotifyPropertyChanged インターフェースの目的は、View 側に ViewModel のプロパティが変更されたことを通知することです。したがって、ViewModel 側で公開しているプロパティに変更があったときに、RaisePropertyChanged() メソッドをコールする必要があります。

具体例として、Text プロパティおよび Result プロパティを次のように定義します。それぞれの get アクセサでは、private フィールドの内容をそのまま返しています。set アクセサでは、private フィールドで保持している値と異なる値がセットされようとしたとき、private フィールドの内容を更新するとともに RaisePropertyChanged() メソッドをコールすることで、対応するプロパティが変更されたことを View 側に通知しています。

ここでは Text プロパティが変更されたとき、すべて大文字にした文字列を Result プロパティに設定するようにしています。

コード 8.5 : プロパティの追加とその変更通知

MainViewModel.cs の一部

```
1  private string text;
2  /// <summary>
3  /// 文字列を取得または設定します。
4  /// </summary>
5  public string Text
6  {
7      get { return text; }
8      set
9      {
```

```
        if (text != value)
        {
            text = value;
            Result = text.ToUpper();
            RaisePropertyChanged("Text");
        }
    }
}

private string result;
/// <summary>
/// 処理結果を取得または設定します。
/// </summary>
public string Result
{
    get { return result; }
    set
    {
        if (result != value)
        {
            result = value;
            RaisePropertyChanged("Result");
        }
    }
}
```

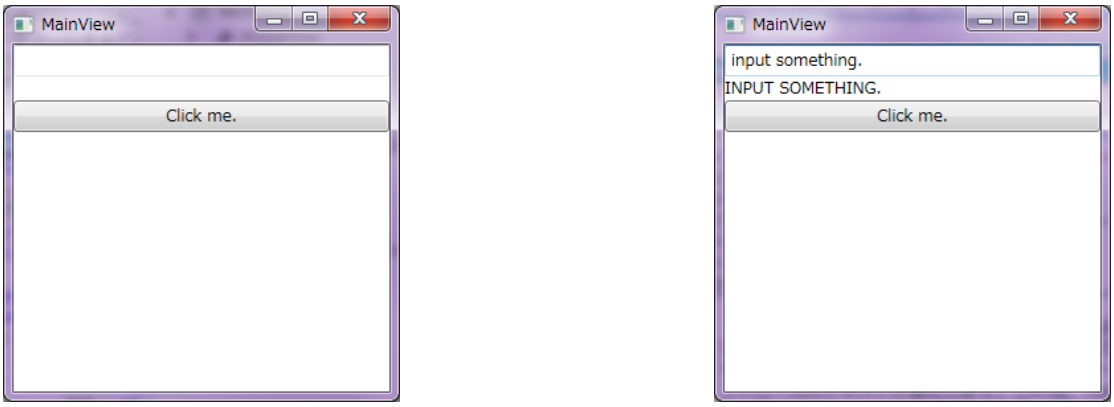
次に、これらのプロパティを参照するように、MainView を次のように変更します。

コード 8.6：データバインディング機能を利用した MainView

MainView.xaml

```
<Window x:Class="Section2.Views.MainView"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="300" Width="300">
    <StackPanel>
        <TextBox Text="{Binding Text}" />
        <TextBlock Text="{Binding Result}" />
        <Button Content="Click me." />
    </StackPanel>
</Window>
```

コンパイル、実行すると、Buton コントロールおよび TextBox コントロールの中身が MainViewModel で定義した Text プロパティの内容と一致するようになるため、起動直後は空白となります。TextBox コントロール内のテキストを変更した後、TextBox コントロールのキーボードフォーカスを外す（Tab キーを押す）と、TextBlock コントロールのテキストが変化します。これは、Text プロパティの変更によって Result プロパティが変更され、その変更が TextBlock コントロールの Text プロパティに伝わっているからです。



(a) 起動直後 (b) テキスト変更後

図 8.8：Text プロパティがバインディングされた画面

View 側からのプロパティ変更通知のタイミングは設定によって変更できます。デフォルト値は LostFocus といって、フォーカスを失ったときに通知されるようになっています。テキスト内容を変更した時点で MainViewModel にその変更が通知されるようにする場合は、PropertyChanged という設定値にする必要があります。具体的なコードは次のようになります。

コード 8.7：UpdateSourceTrigger を指定したデータバインディング

MainView.xaml の一部

```
1 <TextBox Text="{Binding Text, UpdateSourceTrigger=PropertyChanged}" />
```

8.5 ICommand インターフェースの自前実装と具体例

次に、ボタンを押すことによって実行されるコマンドを、データバインディング機能を用いて ViewModel 側に記述する方法を紹介します。コマンドを使用するには、ICommand インターフェースを実装したプロパティが必要になります。前節と同様に、すでに公開されている YKToolkit.Controls.dll などのライブラリを利用することで ICommand インターフェースが既に実装されたクラスを扱うことができますが、ここでは敢えて ICommand インターフェースを自前で実装して、その内部構造を知ってもらおうと思います。

DelegateCommand クラスという ICommand インターフェースを実装するクラスを次のように定義します。

コード 8.8：DelegateCommand クラスの定義

DelegateCommand.cs

```
1  namespace Section2
2  {
3      using System;
4      using System.Windows.Input;
5
6      public class DelegateCommand : ICommand
7      {
8          /// <summary>
9          /// コマンドの実体を保持します。
10         /// </summary>
11         private Action<object> _execute;
12
13         /// <summary>
14         /// コマンドの実行可能判別処理の実態を保持します。
15         /// </summary>
16         private Func<object, bool> _canExecute;
17
18         /// <summary>
19         /// 新しいインスタンスを生成します。
20         /// </summary>
21         /// <param name="execute">コマンドの実体を指定します。</param>
22         public DelegateCommand(Action<object> execute)
23             : this(execute, null)
24         {
25         }
26
27         /// <summary>
28         /// 新しいインスタンスを生成します。
29         /// </summary>
30         /// <param name="execute">コマンドの実体を指定します。</param>
31         /// <param name="canExecute">コマンドの実行可能判別処理の実体を指定します。</param>
32         public DelegateCommand(Action<object> execute, Func<object, bool> canExecute)
33         {
34             _execute = execute;
35             _canExecute = canExecute;
36         }
37
38         /// <summary>
39         /// CanExecuteChanged イベントを発行します。
40         /// </summary>
41         public static void RaiseCanExecuteChanged()
42         {
43             CommandManager.InvalidateRequerySuggested();
44         }
45
46         #region ICommand のメンバ
```

```
47          /// <summary>
48          /// コマンドの実行可能判別処理を実行します。
49          /// </summary>
50          /// <param name="parameter">コマンドパラメータを指定します。</param>
51          /// <returns>コマンドが実行可能であるとき true を返します。</returns>
52          public bool CanExecute(object parameter)
53          {
54              return _canExecute == null ? true : _canExecute(parameter);
55          }
56
57          /// <summary>
58          /// コマンドの実行可能判別条件が変更されたときに発生します。
59          /// </summary>
60          public event System.EventHandler CanExecuteChanged
61          {
62              add { CommandManager.RequerySuggested += value; }
63              remove { CommandManager.RequerySuggested += value; }
64          }
65
66          /// <summary>
67          /// コマンドを実行します。
68          /// </summary>
69          /// <param name="parameter">コマンドパラメータを指定します。</param>
70          public void Execute(object parameter)
71          {
72              if (_execute != null)
73                  _execute(parameter);
74          }
75          #endregion ICommand のメンバ
76      }
77  }
```

ICommand インターフェースは、実際にコマンドの内容を実施する Execute() メソッドと、このコマンド自体が実行可能かどうかを判別するための CanExecute() メソッドを実装する必要があります。コマンドの内容や実行可能かどうかの条件をここで固定させる必要はないため、_execute および _canExecute という private フィールドを用意し、コマンドの中身や実行可能判別の処理を外部から指定できるようにしています。

実行可能かどうかに影響するような変更があった場合、RaiseCanExecuteChanged() メソッドを呼び出すことで、CanExecuteChanged イベントを発行し、その変更を View 側に伝えます。ここでは、CanExecuteChanged イベントを CommandManager.RequerySuggested イベントに委任しているため、こちらのイベントを発生させることで CanExecuteChanged イベントを発生させています。ただし、同じ View（ViewModel）に複数のコマンドが存在し、この RaiseCanExecuteChanged() メソッドを呼び出した場合、すべてのコマンドに対して再評価がおこなわれるため、各コマンドに対して呼ぶ必要はありません。このことから、RaiseCanExecuteChanged() メソッドは static なメソッドとして定義しています。

具体例として、ClearCommand プロパティを次のように定義します。get アクセサによるプロパティ値取得タイミングで、private フィールドである clearCommand 変数が null のときのみ DelegateCommand クラスをインスタンス化しています。コマンドの中身は Text プロパティを空にするという処理で、Text プロパティが既に空である場合は実行不可であるという判別をおこなっています。

コード 8.9：ClearCommand プロパティの定義

MainViewModel.cs の一部

```
1   private DelegateCommand clearCommand;
2   /// <summary>
3   /// 文字列をクリアするコマンドを取得します。
4   /// </summary>
5   public DelegateCommand ClearCommand
6   {
7       get
8       {
```

```
9           if (clearCommand == null)
10              clearCommand = new DelegateCommand(
11              _ => Text = string.Empty,
12              _ => !string.IsNullOrEmpty(Text)
13              );
14          return clearCommand;
15      }
16  }
```

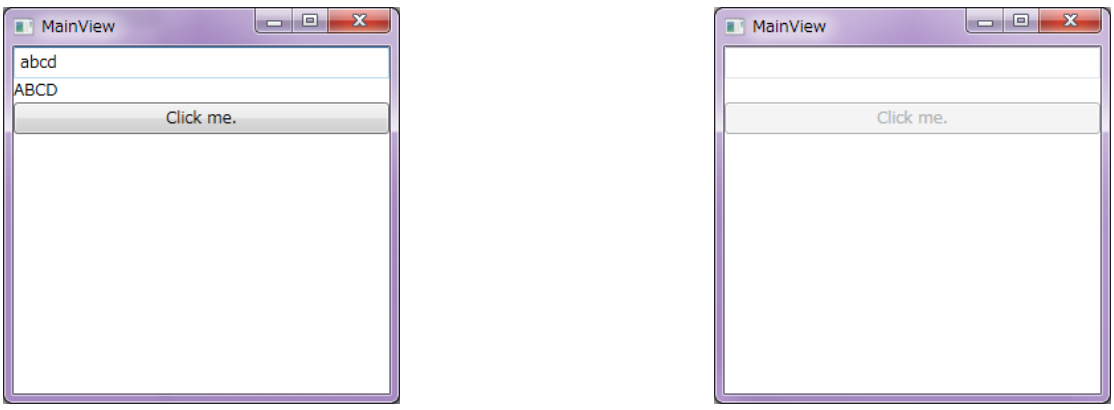
次に、このプロパティを参照するように、MainView を次のように変更します。

コード 8.10：データバインディング機能を利用した MainView

MainView.xaml

```
1  <Window x:Class="Section2.Views.MainView"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainView" Height="300" Width="300">
5      <StackPanel>
6          <TextBox Text="{Binding Text, UpdateSourceTrigger=PropertyChanged}" />
7          <TextBlock Text="{Binding Result}" />
8          <Button Content="Click me." Command="{Binding ClearCommand}" />
9      </StackPanel>
10 </Window>
```

コンパイル、実行すると、起動直後は Text プロパティが空であるため、Button コントロールの Command プロパティの実行可能判別処理によって Button コントロールの IsEnabled プロパティが false となっています。TextBox コントロールに任意の文字列を入力して Text プロパティに文字列が設定されると、Button コントロールの実行可能判別処理によって Button コントロールの IsEnabled プロパティが true となり、ボタンを押せるようになります。また、ボタンを押すと ClearCommand のコマンド実行処理がおこなわれ、Text プロパティが空になるため、TextBox コントロールのテキストが空になります。



(a) テキストを入力するとボタンが有効になる　　(b) ボタンを押すと TextBox コントロールが空になる

図 8.9：ClearCommand プロパティがバインディングされた画面

9 おわりに

本書は WPF の基本的な使い方についてまとめることで、WPF による開発技術力向上促進を目的として作成しました。自分がわからなくて web で調査したことをこうしてドキュメント化することで、自分の中でも整理することができ、またいろんな人たちと情報を共有することができるようになったのではないかと思います。

ここに掲載している内容はかなり初歩的な内容で、上級者の方々から見れば無用の長物かもしれませんが、今後 WPF による開発を始めようとする方や、既に WPF を知っている方のリファレンスとして活用していただければ幸いです。

改訂履歴

改訂年月日	バージョン	概　要
2015.10.17	1.0.0	初版リリース。
2015.12.16	1.0.1	第 8 章を校正。
2016.08.18	1.0.2	Path クラスによる描画 を追加。
2016.08.25	1.0.3	外部公開のため細かい部分を修正。

Windows Presentation Foundation 入門

2015.10.17 初版
2016.08.25 改訂